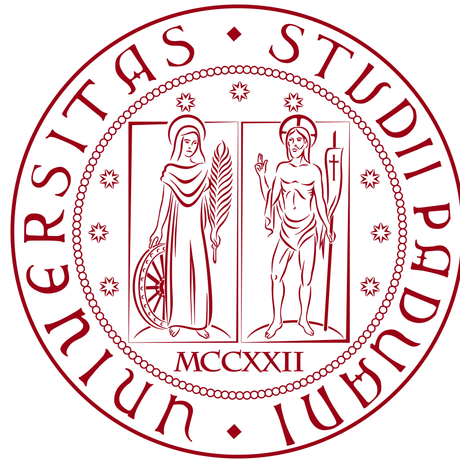
Università degli Studi di Padova

Department of Mathematics "Tullio Levi-Civita"

Bachelor's Degree in Computer Science



A Practical Approach to Face Verification: An End-to-End Implementation

Supervisor

Prof. Ballan Lamberto

Graduand

Filippini Giovanni

Academic Year 2023-2024

© Filippini Giovanni, September 2024. All rights reserved. Bachelor's Thesis: "*A Practical Approach to Face Verification: An End-to-End Implementation*", Università degli Studi di Padova, Department of Mathematics "Tullio Levi-Civita".

"I have no special talents. I am only passionately curious."

Albert Einstein

Acknowledgements

To my family - my parents, my sister, my relatives - you are the foundation upon which my aspirations are constructed. In both moments of triumph and hours of despair, your unwavering belief has served as a guiding light, a North Star, that has illuminated the path forward.

And to my friends - the old guards and the new allies, the high school comrades and the college ones - your words bridged the chasms of uncertainty. Some of you have walked beside me for years, others joined along the way, but each of you has left an indelible mark.

To all of you, this thesis stands as a testament to your impact. You saw potential where I saw limitations, you offered hands when I stumbled, you celebrated victories I couldn't yet see.

Thank you, for everything.

Abstract

The aim of this thesis is to present the project that was carried out during the internship at DuckMa Srl. The objective was to develop a *Proof of Concept (PoC)*$_{\boldsymbol{G}}$ for a face verification system that would encompass three core functionalities: data collection and preprocessing, the ability to determine whether a person present in an image or video feed matches the vector selected from a data structure, and the removal of a specific person, given their facial representation. Furthermore, it was essential to ascertain the feasibility of operating a face verification system on devices with constrained computational resources.

This paper will examine the fundamental principles of computer vision, including the techniques and algorithms utilized, as well as the technologies and tools employed in the project's development and architecture. Ultimately, the outcomes and conclusions derived from the experience will be presented, along with a comparison of the objectives set and those achieved, and with the potential future development.

Contents

1 Introduction **1**

1.1 The Company . . . 1

1.1.1 The DuckMa Method . . . 1

1.2 The Idea . . . 3

1.2.1 The Difficulties . . . 3

1.3 Structure of the Document . . . 4

1.4 Typographical Norms . . . 5

2 What is Computer Vision? **7**

2.1 The Evolution of Computer Vision . . . 7

2.2 The Fundamentals . . . 10

2.2.1 Features . . . 10

3 DeVisu, a Facial Verification System **15**

3.1 The Project . . . 15

3.1.1 Face Recognition vs. Face Verification . . . 16

3.2 Requirements . . . 16

3.2.1 Mandatory Requirements . . . 17

3.2.2 Desirable Requirements . . . 18

3.2.3 Optional Requirements . . . 18

3.3 Planning . . . 19

4 Technologies and Tools **23**

4.1 Framework and Libraries . . . 23

4.1.1 Connexion . . . 23

4.1.2 Face Recognition 23
4.1.3 Flask 24
4.1.4 OpenAPI 24
4.1.5 OpenCV 25
4.1.6 Python 25
4.1.7 SQLite 26
4.1.8 Uvicorn 26
4.2 Hardware 27
4.2.1 Raspberry Pi 5 27
4.3 Development Tools 28
4.3.1 Visual Studio Code 28
4.3.2 Git 28
4.3.3 GitHub 29

5 Project Encoding 31
5.1 System Architecture 32
5.2 Adding a new Person 33
5.2.1 Storing the Data 36
5.3 Vecotrization 37
5.3.1 Data Preprocessing 38
5.3.2 Storing the Vectors 40
5.4 Verification of a Person 41
5.4.1 Vectors Comparison 42
5.5 Removal of a Person 45

6 Testing and Validation 47
6.1 Unit Testing 47
6.1.1 Vectorization 47
6.1.2 API Calls 49
6.1.3 Utils 50
6.1.4 Routing 51
6.2 Integration Testing 52
6.2.1 Testing the Registration Process 52

6.2.2 Testing the Verification Process 52
6.2.3 Testing the Deletion Process 53
6.3 System Testing 53
6.3.1 Performance Testing 53
6.3.2 Compliance 54
6.4 Accuracy of the Algorithm 55
6.4.1 Implementation 56
6.4.2 Results 58

7 Conclusions **61**
7.1 Achieved Goals 61
7.2 Future Development 62
7.3 Knowledge Acquired 62
7.4 Final Considerations 63

Acronyms and abbreviations **i**

Glossary **ii**

Bibliography **v**

Webliography **vi**

List of Figures

1.1 DuckMa's logo 1

2.1 Example of Haar-like features 12

3.1 Logo of the Proof of Concept project 15

4.1 Connexion's logo 23
4.2 Flask's logo 24
4.3 OpenAPI's Initiative logo 24
4.4 OpenCV's logo 25
4.5 Python's logo 25
4.6 SQLite's logo 26
4.7 Uvicorn's logo 26
4.8 The Raspberry Pi 5 27
4.9 Visual Studio Code's logo 28
4.10 Git's logo 28
4.11 GitHub's logo 29

5.1 The application follows a monolithic architecture 32
5.2 Image capturing 33
5.3 Vecotrization of an image 37
5.4 Preprocessing 38
5.5 verification of a person 41
5.6 Remove a person from the database 45

List of Tables

3.1 Table of the mandatory requirements 17
3.2 Table of the desirable requirements 18
3.3 Table of the optional requirements 18
3.4 Weekly tasks planning 21

6.1 Table showing the accuracy of the system as the tolerance changes 58

7.1 Table showing the status of the requirements 61

List of Codes

5.1 Initializing the Haar-like Features classifier 34
5.2 Detecting faces in the captured image 34
5.3 Capturing an image when a single face is detected 34
5.4 Capturing and saving the image of the face 35
5.5 Route for storing the new person in the database 36
5.6 Generate a vector given the image of a face (Part 1) 37
5.6 Generate a vector given the image of a face (Part 2) 38
5.7 Vectorization of an image 39
5.8 Base64 encoding of the vector 40
5.9 Base64 decoding of the vector 40

5.10 Get a user from the database, given the six digit code 41
5.11 Calculate the distance between two vectors 42
5.12 Handling the result of the verification process (Part 1) 44
5.12 Handling the result of the verification process (Part 2) 44
5.13 Deletion of a user, given an six digit code 45
5.14 Handling the result of the verification process 46

6.1 Test for generating the vector form an image without a face . . 48
6.2 Test for generating the vector form an image 48
6.3 Test for return an error when a user is not found 49
6.4 Test for saving a person to the database 49
6.5 Simulating the correct initialization of the camera 50
6.6 Simulating an error while initializing the camera 50
6.7 Test for adding a person with a POST call (Part 1) 51
6.7 Test for adding a person with a POST call (Part 2) 51
6.8 Generating the base face vector for the verification process . . . 56
6.9 Evaluation of the performance (Part 1) 57
6.9 Evaluation of the performance (Part 2) 57
6.10 Calculating the accuracy of the system from the given data . . . 58

Chapter 1

Introduction

1.1 The Company

DuckMa is a software company based in Rezzato, Italy, founded over 10 years ago in Brescia. Their mission is to meet needs of the costumers, acting as a trusted partner by adopting new technologies and optimal solutions. They utilize a consultative approach, collaborating closely with a diverse list of clients, ranging from startups to established businesses and larger companies, to analyze their needs and devise a strategic road map with them.

Figure 1.1: DuckMa's logo.

1.1.1 The DuckMa Method

When working with a wide range of clients, each with unique needs, it's important to have an established *Way of Working*$_G$, and over the course of a decade of experience, DuckMa identified several issues that required an improvement in their development processes. Initially, they applied a single-step approach, en-

compassing the entire process from customer needs analysis to post-development support. However, this approach frequently resulted in two problematic situations: underestimated budgets, which did not cover actual development costs, or overestimated budgets, which did not offer good value for money for the customer.

The resulting methodology, called "*DuckMa Method*", enables the client to know with certainty the cost and time frame for the development, and participate actively in each stage of the process.
The method comprises three phases: **To Define**, **To Build**, and **To Grow**.

- **Define**: The fist phase entails the analysis of the client's needs, which is used to define the application's macro functionality and user journey. A *wireframe* is created to establish the connection between screens, and a *moodboard* is developed to determine the visual style. The discovery process allows the client's needs to be fully scrutinized and aligned with the technical functionality, ensuring that every aspect of the app meets the client's needs. At the end of this phase, detailed documentation of functional and technical specifications is provided.

- **Build**: In this phase, the static design of the app screen and a high-fidelity prototype are developed. This phase involves gathering feedback from the client and stakeholders, optimizing the user interface and user experience. A project manager oversees the team of developers, and the app is released with quality assurance and testing. The client receives free assistance with bug fixing and process optimization.

- **Grow**: In the final phase, the app is released to digital stores and ongoing support is provided, in terms of to fully support the latest version of Android and iOS, and a continuous feedback from the end user by engaging with them throughout the store review. Additionally, the client receives user behavior analysis and integration with customer relationship management systems, ensuring that the app remains aligned with market needs and generates returns on investment.

1.2 The Idea

After discussing with the company about a project that could be carried during the internship, their proposal consisted in the development of a Proof of Concept regarding a facial verification system optimized for deployment on devices with limited computational resources, such as a Raspberry Pi. This system would consist of three fundamental functionalities:

- **Data gathering**: This initial phase involves acquiring the data needed to extract a person's facial characteristics, typically obtained from a picture taken by the application.
  - **Image processing**: After acquiring the image, the system should preprocess them to ensure a homogeneous data set.
- **Vectorization of the data**: Once the image is processed, it's transformed into a vector representing the person's unique features, and then this data is stored in a data structure.
- **Face verification**: With the vector uploaded, the system can initiate a recognition process to identify the person based on their facial characteristics.

1.2.1 The Difficulties

Significant challenges arise when working with computationally limited devices and building services that require a significant amount of computing resources, among which a slow down of the device due to the complexity of the processes involved or the risk of breaking the device if not properly manged. These limitations can also compromise the effectiveness and efficiency of the system, slowdown in processing time or decrease the accuracy of the program. To mitigate these problems, it is crucial to employ algorithms with low computational costs without sacrificing accuracy. Additionally, a product development approach that leverages every aspect of the device's capabilities is essential, and

by optimizing resource utilization and adopting efficient algorithms, the negative impacts of computational limitations can be minimized, ensuring acceptable performance levels.

1.3 Structure of the Document

The document is structured in six chapters, in addition to the first introductory chapter, each of which addresses a distinct aspect of the internship:

- Chapter Two: "What is Computer Vision?", provides an introduction to computer vision, its applications, and the basic concepts that are used in the field.
- Chapter Three: "DeVisu, a Facial Verification System", presents the *DeVisu* project, the requirements, and the planning that was followed to develop the project.
- Chapter Four "Technologies and Tools", shows the technologies and tools that were used to develop the project.
- Chapter Five: "Project Encoding", describes the development process of the project, the challenges faced, and the solutions that were implemented.
- Chapter Six: "Test and Validation", presents the testing strategy and the results obtained in the various tests.
- Chapter Seven: "Conclusions", which comprises the achieved goals and a summary of this experience, describing also the future development that could be done.

1.4 Typographical Norms

To facilitate the reading and understanding of the text, the following typographical conventions have been adopted:

- The glossary, located at the end of this document, defines acronyms, abbreviations, and ambiguous or uncommonly used terms mentioned;
- For the first occurrence of those terms, the following nomenclature is used: *Database*$_{\boldsymbol{G}}$.
- Terms in foreign languages, or forming part of technical jargon, are highlighted with italics, for example: *Convolutional Neural Network*.

Chapter 2

What is Computer Vision?

For describing in a simple way what computer vision is, we could say that "*If AI enables computers to think, computer vision enables them to see, observe and understand*"[1].
Although the term "computer vision" may suggest the capability of computers to perceive the world around them in a manner analogous to human perception, in reality computers are only capable of "reasoning" with numerical data. Therefore, for computers to process information, it must first be translated into binary code.

2.1 The Evolution of Computer Vision

The field of computer vision is not something that has been introduced in recent years, although some of the biggest advances have been made in the last decade, but the basic concepts and techniques, such as *image processing* and *pattern recognition*, were developed back in the mid-20th century. A significant milestone was the development of the Mark I Perceptron[2] by Frank Rosenblatt in 1957. The Perceptron, an early form of *Artificial Neural Network*$_G$, laid the groundwork for future advancements in *machine learning* and computer

[1] IBM. *What is computer vision?* URL: https://www.ibm.com/topics/computer-vision. (Accessed: May 26, 2024)

[2] The *Perceptron* is an algorithm, proposed by Warren McCulloch and Walter Pitts in 1943, that belongs to the category of supervised learning methods. It classifies inputs, represented as vectors, determining whether the input belongs to a specific class of objects.

vision. This work demonstrated, ahead of its time, the potential for machines to recognize patterns and learn from visual data, a concept that would prove revolutionary in the decades to come.

The 1960s saw further advancements, with researchers like Lawrence Roberts developing algorithms for extracting 3D geometric information from 2D images.

In the 1970s and 1980s, the field witnessed substantial improvements in the development of algorithms specifically designed for vision-related tasks. These advancements included sophisticated techniques for *edge detection*, *image segmentation*, and *feature extraction*. Notable contributions included the Canny edge detector, developed by John F. Canny in 1986, which remains widely used today for its effectiveness in identifying boundaries within images.

During this period, Kunihiko Fukushima's Neocognitron[3] introduced the concept of *local feature integration*[4]. This new approach would later transform into the modern *Convolutional Neural Networks (CNN)* $_{\boldsymbol{G}}$, a cornerstone of contemporary deep learning in computer vision.

This field of research continued to evolve rapidly in the 1990s, with the introduction of *statistical learning* techniques that brought a new level of sophistication to image analysis, like the development of algorithms like SIFT (Scale-Invariant Feature Transform) by David Lowe in 1999, which allowed the identification of objects regardless of scale, rotation, or partial occlusion.

A notable milestone came in 2001 with the introduction of the Viola-Jones framework for real-time face detection. This algorithm, which we will discuss later, became particularly significant for its practical applications and robustness in real-time scenarios by enabling rapid face detection in images and video streams, laying the foundation for numerous applications in security, consumer electronics, and social media.

The 2000s ushered in the era of *Deep Learning* $_{\boldsymbol{G}}$, which fundamentally

[3] The *Neocognitron* was an Artificial Neural Network developed primarily for the recognition of Japanese handwritten characters, as well as for pattern recognition.

[4] This technique involves detecting and combining small, distinct elements from an image to form a comprehensive understanding of the visual input. In the case of the *Neocognitron*, allowing the recognition of complex patterns, such as Japanese handwritten characters, by hierarchically integrating features from simple edges to complex shapes.

changed the landscape of computer vision. Convolutional Neural Networks, popularized by researchers like Yann LeCun, enabled more accurate and efficient image recognition and analysis. The breakthrough moment came in 2012 when Alex Krizhevsky, Ilya Sutskever, and Geoffrey Hinton demonstrated the power of deep Convolutional Neural Networks in the ImageNet competition[5] by achieving unprecedented accuracy in image classification.
The usage of deep learning made it possible to expand the application of vision-related tasks into numerous fields, like in autonomous vehicles for perceiving and interpreting the environment, enabling safer navigation and decision-making, medical imaging, with significant advancements in the detection and diagnosis of diseases from X-rays, MRIs, and other imaging modalities, and in robotics, by enhancing the visual perception of robots for more sophisticated manipulation and interaction with the physical world.

The 2010s saw further refinements and innovations in deep learning architectures for computer vision, with techniques like transfer learning, where models pre-trained on large datasets are fine-tuned for specific tasks, that expanded the accessibility and applicability of computer vision technologies.

Nowadays, computer vision stands at the forefront of the artificial intelligence revolution that we are all living and experiencing. It represents a powerful convergence of image processing, machine learning, and progressively increasing computational power. This synergy empowers machines to perceive and interpret the visual world with unprecedented accuracy and complexity.

[5] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. "ImageNet Classification with Deep Convolutional Neural Networks". In: *Advances in Neural Information Processing Systems*. Ed. by F. Pereira et al. Vol. 25. Curran Associates, Inc., 2012. URL: https://proceedings.neurips.cc/paper_files/paper/2012/file/c399862d3b9d6b76c8436e924a68c45b-Paper.pdf

2.2 The Fundamentals

As previously said, the purpose of computer vision is to enable computers to interpret and understand visual information, mimicking the human vision, and this is possible thanks to the development of algorithms and techniques to extract meaningful insights from an image or video. Fundamental concepts in computer vision include image processing, feature extraction, object detection, image segmentation and pattern recognition.
For the sake of this thesis, we are going to discuss the key technology implemented in the project: features.

2.2.1 Features

In the case of computer vision, a feature refers to a distinctive piece of information within an image that can be used to describe and characterize the content of the image. The extraction of these features involves identifying and extracting meaningful data from the given data, such as edges, textures, shapes, colors, and other visual patterns, and it serves as the foundation for more advanced computer vision tasks, enabling the system to understand and interpret the visual information.

Features can be broadly categorized into low-level and high-level features. Low-level features are basic visual attributes like edges, corners, and intensity gradients, which can be extracted directly from the image data, and often combined and processed to derive higher-level features that capture more complex and semantically meaningful information.
High-level features, on the other hand, represent more abstract concepts and may include shapes, textures, or even specific objects or patterns, typically obtained by combining and analyzing low-level features using machine learning algorithms or specialized feature extraction techniques.

Haar-like

Haar-like features are a type of low-level feature, particularly effective in detecting patterns and variations in intensity within an image. They operate by evaluating the difference in pixel values between adjacent rectangular regions, allowing for the identification of edges, corners, and texture variations.
This type of feature is named after the Hungarian mathematician Alfred Haar due to its usage of the Haar wavelet, which was proposed in 1910 by the Hungarian mathematician[6]. Wavelets are mathematical functions that represent a signal or an image in terms of a series of oscillations at different scales and positions[7].
The Haar wavelet is the simplest possible wavelet, defined as:

$$\psi(t) = \begin{cases} 1 & 0 \leq t < 1/2 \\ -1 & 1/2 \leq t < 1 \\ 0 & \text{otherwise} \end{cases}$$

This wavelet function can be scaled and translated to create a set of basis functions that can be used to decompose a signal or image into its constituent wavelet coefficients[8]. The Haar-like features used in computer vision are derived from these wavelet coefficients, capturing basic visual attributes such as edges, lines, and intensity gradients within a given region of an image.

In this case, it is a set of shaped feature comprised of two colors, black and white, that can capture the presence of certain visual structures in an image, such as edges, lines, and corners. This approach was proposed in 2001, when two researchers at the Massachusetts Institute of Technology, Paul Viola and Michael Jones, published a paper[9] showcasing that this approach can be used to

[6] Alfred Haar. “Zur Theorie der orthogonalen Funktionensysteme”. In: *Mathematische Annalen* 69.3 (Sept. 1910), pp. 331–371. ISSN: 1432-1807. DOI: 10.1007/BF01456326

[7] Wavelets are employed in signal processing, image compression, and a multitude of other applications due to their capacity to efficiently represent both minute and substantial features of a signal or image.

[8] The wavelet coefficients represent the contribution of each wavelet basis function to the original signal or image, allowing for a compact and efficient representation.

[9] P. Viola and M. Jones. “Rapid object detection using a boosted cascade of simple

detect faces, even in real-time applications, due to the efficiency of the algorithm. This was also the main reason for implementing them in the project.

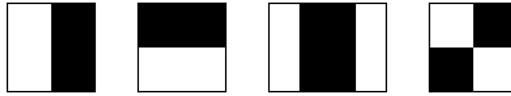


Figure 2.1: Example of Haar-like features.[10]

There are three basic types of Haar-like features:

- **Two-Rectangle Feature**: This computes the sum of the pixel intensities in two adjacent rectangles. It can be used, for example, for identifying the area of the eyes.
- **Three-Rectangle Feature**: This compares the sum of pixel intensities in three adjacent rectangles, typically aligned horizontally or vertically. We can imagine this be used for identifying the area of the nose, with one white and two black rectangles.
- **Four-Rectangle Feature**: This feature compares the sum of pixel intensities in four adjacent rectangles arranged in a 2x2 grid. It can be used when dealing with a wider area, identifying different sections of a face.

The way Features works is simple but really clever: since some parts of the face are brighter or darker than others (for example the area of the eyes is darker than the tip of the nose) those features are calculated by taking the sum of pixel intensities in adjacent rectangular regions of the image and then finding the difference between these sums.
Because Haar-like features are based on simple rectangular patterns, they are considered low-level features since they capture only low-level visual information and do not directly represent complex shapes, textures, or high-level semantic

features". In: (2001). DOI: 10.1109/CVPR.2001.990517

[10] Indif. *VJ featureTypes*. Own work, CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=11681210. Accessed: June 4, 2024. Personal revisit.

concepts. This results in a much more efficient way to recognize faces than other deep learning-based solutions, even in real-time scenarios.

Mathematical Formulation

Given an image represented as a two-dimensional array $I(x, y)$ that represents the pixel intensities:

Integral Image (Summed-area Table) To quickly compute the sum of pixel intensities over any rectangular area, an integral image $II(x, y)$ is used. The integral image at any point (x, y) is defined as:

$$II(x, y) = \sum_{i=0}^{x} \sum_{j=0}^{y} I(i, j)$$

This allows the sum of pixel intensities in any rectangular region to be calculated using only four references to the integral image:

$$R = II(x_2, y_2) - II(x_2, y_1 - 1) - II(x_1 - 1, y_2) + II(x_1 - 1, y_1 - 1)$$

where (x_1, y_1) and (x_2, y_2) are the top-left and bottom-right corners of the rectangle R.

Feature Calculation For a given Haar-like feature, the sum of pixel intensities for each rectangle is obtained by calculating the integral of the image. Then, the feature value is determined by computing the weighted sum of these sums.

- **Two-Rectangle Feature**: Suppose the feature consists of two adjacent rectangles, R_1 and R_2. The feature value f is:

  $$f = \sigma(R_1) - \sigma(R_2)$$

  where $\sigma(R_i)$ represents the sum of pixel intensities over the rectangle R_i.

- **Three-Rectangle Feature**: For three adjacent rectangles, R_1, R_2, and R_3, the feature value is:

$$f = \sigma(R_1) - \sigma(R_2) + \sigma(R_3)$$

- **Four-Rectangle Feature**: For four rectangles arranged in a 2x2 grid, the feature value is:

$$f = (\sigma(R_1) + \sigma(R_4)) - (\sigma(R_2) + \sigma(R_3))$$

In the above formulations, $\sigma(R_i)$ represents the sum of pixel intensities over the rectangle R_i, computed using the Integral Image method previously defined.

Chapter 3

DeVisu, a Facial Verification System

3.1 The Project

Figure 3.1: Logo of the Proof of Concept project.

The Proof of Concept, from now on called *DeVisu*, revolves around the development of a facial verification system designed to operate on computationally limited devices, such as the Raspberry Pi platform. The primary objective is to create a solution that can effectively perform facial recognition tasks while adhering to the limitations imposed by resource-constrained hardware.

The application's core functionality encompasses three main operations: the possibility of adding a new person to the database, verifying the identity of a person based on a previously saved vector, and removing individuals from the database. The verification process involves capturing a live image of the individual using the device's camera, extracting facial features from the captured image, and comparing them with the facial features stored in the database. If

a match is found within a predefined threshold of similarity, the individual's identity is successfully verified, and an appropriate response is provided to the user.

3.1.1 Face Recognition vs. Face Verification

The term *face recognition* is often employed as an overarching concept, similarly to artificial intelligence, encompassing a wide range of solutions. At the foundational level, we can talk about *face detection*, the core component of face-related technologies, which is solely dedicated to determine the presence of a face in an image or video feed, without delving into the identification of the individual.

Face recognition technology works by comparing a given image to a database of known individuals. The system scans the image to identify unique facial characteristics and features that are specific to each person, much like how we might recognize a friend in a crowd by noticing their distinctive facial traits.

On the other hand, *face verification* involves matching a given facial image with a specific stored version of that face, which is usually represented as a mathematical model. This process is similar to the verification methods used in electronic passports, where a taken photo is compared to the one stored in a chip inside the document to confirm the person's identity.

We can say that face recognition systems have a one-to-many matching process, where the system compares the input image to all the images in the database to find a match. In contrast, face verification systems have a one-to-one matching process, where the system compares the input image to a single stored image to determine if they match, this being the main reasons why face verification is faster and less computationally intensive than face recognition.

3.2 Requirements

In order to define the features and functionalities that the Proof of Concept should have, a set of requirements has been defined, in agreement with the

company, and divided into three categories, according to their importance and necessity:

- **ROB**: for mandatory requirements, binding as the primary element required by the developer;
- **RDE**: for desirable requirements, not binding or strictly necessary, but of recognizable added value;
- **ROP**: for optional requirements, representing added value not strictly necessary for the completion of the project.

In addition, every acronym is accompanied by a number that indicates the corresponding requirement in the list.

3.2.1 Mandatory Requirements

Requirement	Description
ROB-1	Implementation of a system for capturing image/photo.
ROB-2	Saving the representation of a person's face within a data structure.
ROB-3	Implementation of a system that can tell whether or not a previously stored face belongs to a specific person.
ROB-4	The application should compile and run correctly on a Raspberry Pi.

Table 3.1: Table of the mandatory requirements.

3.2.2 Desirable Requirements

Requirement	Description
RDE-1	Implement a graphical user interface (GUI) to enhance the usability of the application.
RDE-2	Ensure a homogeneous system structure, thereby eliminating the necessity for the use of separate software components.
RDE-3	Determine if the image or video feed in question has the with the vector selected from the database.
RDE-4	The user should be able to delete their own face from the system.

Table 3.2: Table of the desirable requirements.

3.2.3 Optional Requirements

Requirement	Description
ROP-1	The application should run natively on the Raspberry Pi.
ROP-2	Integrate an Neural Processing Unit in order to enhance the overall performance of a system.

Table 3.3: Table of the optional requirements.

3.3 Planning

Week	Planned activities
First week *Study of the technologies involved*	During the first week, the following activities were planned: • Meeting with stakeholders to discuss and clarify any doubts about the requirements and requests related to the system to be developed; • Study of the different facial recognition algorithms; • In-depth study of the technologies involved.
Second week *Software integration*	During this week, the activities carried out were: • Analysis of the technologies involved; • Study of face verification algorithms suitable for the project.
Third week *Design of the system*	During this week, the following things were conducted: • Design of the software structure and its core components; • Configuration of the device and development environment; • Tests on system and library integration.

Continuing on the next page

Week	**Planned activities**
Fourth week *Beginning of the development of the Proof of Concept*	In this phase, which marked the beginning of the development: • Configuration of the development environment; • Setup of the API and database handling; • Development of the vectorization system.
Fifth week *Development of the face verification system*	Over the course of the week, the activities carried out were the folllowing: • Creation of the pipeline for the preprocessing of the images; • To write the System responsible for verify if the vectors correspond to the same person; • Initial drafting of the project documentation.
Sixth week *Development of the person's removal feature*	Following the completion of the verification system, this phase was initiated: • Development of the deletion process, in order to remove a person from the database; • Continued working on the documentation of the project.

Continuing on the next page

Week	Planned activities
Seventh week *Completion of the Proof of Concept*	With the end of the development of the proof of concept, these tasks where carried out: • Review and improvement of the developed code; • System test with various faces and different face capture and recognition conditions; • Integration of the latest developments within the document; • Preparation of materials for the final presentation;
Eighth week *Testing and validation*	In this last week, the final activities where carried out: • Validation and verification of the results obtained with respect to what was set; • Final review of the results based on the requirements previously defined; • Showcase of the Proof of Concept to the company and delivery of the product.

Table 3.4: Weekly tasks planning.

Chapter 4

Technologies and Tools

4.1 Framework and Libraries

4.1.1 Connexion

Figure 4.1: Connexion's logo.

Connexion is a Python library that simplifies the development of RESTful APIs[1] by automatically validating and routing the requests based on an OpenAPI specification, which defines the API's structure and behavior.

4.1.2 Face Recognition

Face Recognition is a library that utilizes the dlib library's face detection and alignment algorithms and enables efficient face detection, encoding (converting face images into numerical representations), and comparison of faces within images or videos.

[1]**RE**presentational **S**tate **T**ransfer API, architectural style for designing networked applications. It relies on a stateless, client-server communication model and uses standard HTTP to interact with resources, identified by URLs.

4.1.3 Flask

Figure 4.2: Flask's logo.

Flask is a lightweight, open-source Python *Web Server Gateway Interface (WSGI)*$_{G}$ framework built with a small core and easy-to-extend architecture. It follows the microframework philosophy by providing a minimalistic core with a low-level, modular design that allows developers to choose and integrate only the components they need for their specific project. Despite its minimal footprint, Flask can be extended with a rich ecosystem of third-party libraries and plugins, enabling it to scale up to handle more complex applications.

4.1.4 OpenAPI

Figure 4.3: OpenAPI's Initiative logo.

OpenAPI (formerly known as Swagger) is an open-source specification for designing, building, and documenting RESTful APIs by providing a standardized, language-agnostic interface for describing the structure of an API, including its endpoints, operations, input/output data models, and authentication methods. One of the key benefits of OpenAPI is its ability to automatically generate interactive documentation, client libraries in various programming languages, and server stubs, streamlining the API development process.

4.1.5 OpenCV

Figure 4.4: OpenCV's logo.

OpenCV is a open-source library aimed at accelerating the development of computer vision applications. Originally developed by Intel Research in 1999, it has since evolved into a comprehensive toolbox supported by an active community of contributors. Written primarily in C and C++, OpenCV also provides language bindings for Python and Java, allowing developers to leverage its capabilities across multiple programming languages. The library offers a rich set of algorithms and functions for a wide range of computer vision tasks and machine learning-based vision algorithms.

4.1.6 Python

Figure 4.5: Python's logo.

Python is a versatile, high-level programming language known for its simplicity, readability, and versatility. It supports multiple programming paradigms, including procedural, object-oriented, and functional programming, allowing developers to adopt the approach best suited for their project. One of Python's key strengths lies in its extensive ecosystem of libraries and frameworks, covering areas such as web development, data analysis and scientific computing, machine learning and artificial intelligence, and automation.

4.1.7 SQLite

Figure 4.6: SQLite's logo.

SQLite is a self-contained, *Serverless*$_{\boldsymbol{G}}$, and transactional *Structured Query Language (SQL)*$_{\boldsymbol{G}}$ database engine widely used for its simplicity, reliability, and ease of integration into applications. Unlike traditional database management systems, SQLite stores the entire database in a single file, making it a reliable solution for embedded applications, prototyping, and small to medium-sized projects.

4.1.8 Uvicorn

Figure 4.7: Uvicorn's logo.

Uvicorn is a *Asynchronous Server Gateway Interface (ASGI)*$_{\boldsymbol{G}}$ server implementation for Python, built on top of the asyncio library. Designed to be lightweight and efficient, Uvicorn is tailored to serve asynchronous web applications and APIs, and it leverage the Python's async/await syntax and the ASGI protocol to provide high-performance, non-blocking I/O operations, making it well-suited for handling concurrent connections and long-running tasks.

4.2 Hardware

4.2.1 Raspberry Pi 5

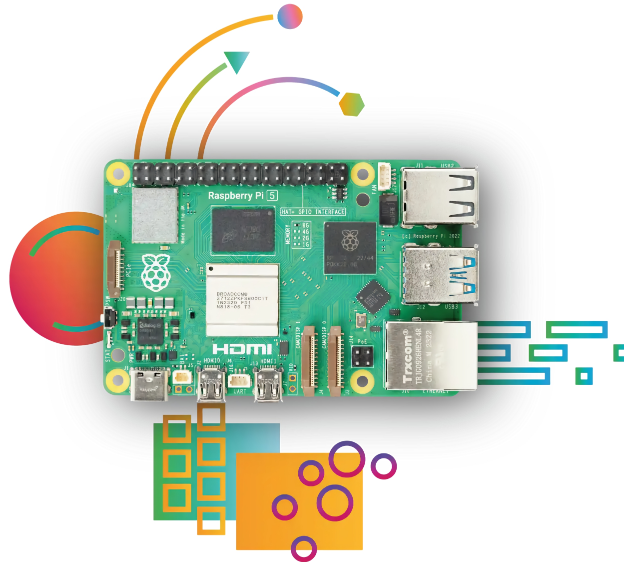


Figure 4.8: The Raspberry Pi 5.[2]

The Raspberry Pi is a single-board computer that has become popular choice for a wide range of applications, including simple educational tools, complex Internet of Things projects, and even some commercial uses.
The latest iteration of the Raspberry Pi series, the Raspberry Pi 5, exhibits notable enhancements in comparison to its predecessors, by including a more powerful processor and enhanced connectivity, thanks to the first custom chip, developed by the Raspberry Foundation, for I/O handling, which collectively make it particularly well-suited for more demanding projects, such as real-time processing.

[2]Raspberry Pi Foundation. *raspberry-pi-5*. Own work, https://www.raspberrypi.com/products/raspberry-pi-5/. Accessed: May 30, 2024. 2023

4.3 Development Tools

4.3.1 Visual Studio Code

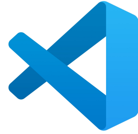


Figure 4.9: Visual Studio Code's logo.

Visual Studio Code, also known as VSCode, is a code editor developed by Microsoft. It is known for its lightweight nature, yet it offers a comprehensive set of features that cater to various programming needs. These include syntax highlighting, intelligent code completion, integrated debugging, and support for version control systems.

4.3.2 Git

Figure 4.10: Git's logo.

Git is a widely used version control system designed to handle everything from small to very large projects with speed and efficiency. It tracks changes in the source code, allowing developers to collaborate, manage code revisions, and maintain a complete history of their work, ensuring that every developer working copy of the codebase is also a complete repository with full version history.

4.3.3 GitHub

Figure 4.11: GitHub's logo.

GitHub is a web-based hosting platform for handling Git-based projects. It also offers collaborative software development, with features like pull requests, issue tracking, and project management tools in order to facilitate teamwork and streamline the development process.

Chapter 5

Project Encoding

DeVisu is a browser-based application, where the front-end is build using Flask and traditional web technologies like HTML, CSS, and JavaScript, with the Jinja2 templating engine serving as the integrating element, and the Django template language to dynamically render HTML pages. This approach decouples the presentation logic from the application logic, in order to have a more robust and flexible system. The fundamental structure of the HTML pages is defined in the `base.html` file, which serves as a blueprint for the entire application. This file encapsulates the common elements, such as the HTML boilerplate code, links to CSS and JavaScript files, and placeholders for injecting dynamic content.

Individual page templates inherit from the `base.html` file, overriding specific blocks to inject their unique content. This facilitates code reusability and maintainability by avoiding duplication and promoting a consistent look and feel across the application. For instance, the `result.html` page is responsible for displaying the outcomes of operations involving the addition, verification, or deletion of a person from the database. It leverages on Jinja2's control structures and variable substitution to dynamically render different HTML elements based on the operation type and the result, success or error, allowing for a single template to fulfill to multiple use cases, promoting code efficiency and reducing redundancy.

The database is implemented using SQLite, and because it stores data in a

single file, it's easy to deploy and manage, by providing at the same time support for standard SQL queries, ensuring compatibility with existing tools and libraries.
However, to facilitate the communication between the application and the database, and to prevent direct access to it, a set of *Application Programming Interface (API)*$_{\boldsymbol{G}}$ has been added, in order to act as intermediaries, handling the tasks of creating, retrieving, and deleting data within the database. Leveraging these APIs allows the application to interact with the database efficiently, ensuring data integrity and providing a smooth user experience.
The implementation of APIs introduces a layer of abstraction, separating the application's presentation logic from the underlying data management operations, but this choice improves the code organization and maintainability, but also enhancing a possible scalability of the project, as the they can be independently extended or modified to accommodate future requirements.

5.1 System Architecture

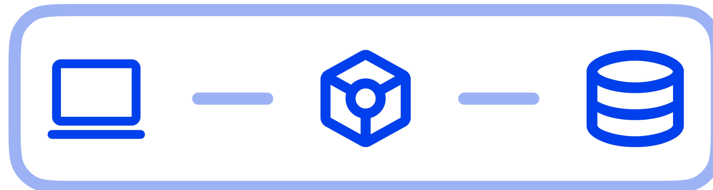


Figure 5.1: The application follows a monolithic architecture.

The Proof of Concept follows a monolithic architecture, where the application logic, API, and database interactions are tightly coupled within a single codebase.

The application layer handles user interface rendering, user interactions, and camera management through various routes. It leverages the `render_template` function from Flask to render HTML templates and interact with the user. The camera functionality is encapsulated in the `VideoCamera` class from the `cameraUtils.py` module, which manages camera operations and image capture.

The API layer is responsible for handling the requests and interacting with

the database, by defining the routes using the add_url_rule method from Connexion, such as for creating a new person (/users POST), reading a specific person (/users/<int:userId> GET), and deleting a person (/users/<int:userId> DELETE). The database interactions are handled within the users.py module, which contains functions for performing *Create, Read, Update, Delete (CRUD)*$_{\boldsymbol{G}}$ operations.

The application integrates with a separate hf_vectorizer.py module for generating and comparing face vectors used for user verification and deletion processes, as well as for encoding and decoding the vector into Base64[1].
The application communicates with the API by making HTTP requests to the defined routes. When creating a new user, the application sends a POST request to the /users route with the user data in the request body. Similarly, for user verification or deletion, the application retrieves the user data by sending a GET request to the /users/by_otp/<string:otp> route with the provided six digit code.

5.2 Adding a new Person

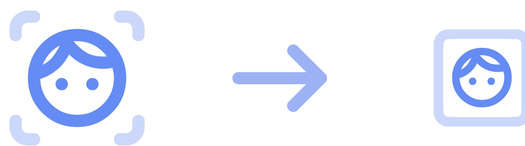


Figure 5.2: Image capturing.

When a user initiates the process of adding a new person to the database, the application first prompts the user to enter the person's name. After the name is provided, the application begins to capture an image from the camera feed.

[1]Base64 is a binary-to-text encoding scheme used to rapresent binary data using 64 ASCII characters.

```
self.face_cascade = cv2.CascadeClassifier(cv2.data.haarcascades +
↪ 'haarcascade_frontalface_default.xml')
```

Code 5.1: Initializing the Haar-like Features classifier.

The image capturing process leverages the Haar-like Features algorithm for face detection. This algorithm is implemented using the OpenCV library, specifically the `cv2.CascadeClassifier` class, and the `haarcascade_frontalface_default.xml` dataset is leveraged in order to perform the face detection.

```
gray_frame = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
faces = self.face_cascade.detectMultiScale(gray_frame,
↪ scaleFactor = 1.15, minNeighbors = 5, minSize = (30, 30))
```

Code 5.2: Detecting faces in the captured image.

The captured frame is first converted to grayscale, and the `self.face_cascade.detectMultiScale()` method is used to detect faces, if any is present. If a face is found, the coordinates of the face region are used to extract and crop the face from the original image.

```
if len(faces) == 1:
    if self.face_detected_time is None:
        self.face_detected_time = time.time()
    elif time.time() - self.face_detected_time >= 3:
        self.capture_image(frame, faces[0])
        self.face_detected_time = None
        return "captured"
elif len(faces) > 1 or len(faces) == 0:
    self.face_detected_time = None
else:
    print("No face detected.")
    self.face_detected_time = None
```

Code 5.3: Capturing an image when a single face is detected.

The application continuously processes the frames that comes from the camera, searching for faces using the `self.face_cascade.detectMultiScale()` method.

Upon the detection of a single face, the application waits for a period of two seconds before capturing the image, in order to have a consistent result. On the other hand, if no face or more then one are found, the timer is reset to zero.

```
def capture_image(self, frame, face_coords):
    self.captured_image_path.path = ""
    (x, y, w, h) = face_coords
    padding = 20

    x1 = max(x - padding, 0)
    y1 = max(y - padding, 0)
    x2 = min(x + w + padding, frame.shape[1])
    y2 = min(y + h + padding, frame.shape[0])

    face_img = frame[y1:y2, x1:x2]

    img_filename = "captured_image{}.jpg"
    ↪   .format(time.strftime("%Y%m%d-%H%M%S"))
    cv2.imwrite(img_filename, face_img)
    path = os.path.abspath(img_filename)

    if os.path.exists(path):
        print(f"Image successfully saved as {path}")
    else:
        print(f"Failed to save image as {path}")

    self.captured_image_path.path = path
    return path
```

Code 5.4: Capturing and saving the image of the face.

The `capture_image()` method is responsible for extracting the face region from the frame, applying padding to ensure the entire face is captured, and saving the face image to a file.

The captured image is saved with a filename that includes the current date and time, in order to ensure the uniqueness of the image generated for each capture, the path of the saved image is stored in the `self.captured_image_path` object, which can be accessed later for further processing.

5.2.1 Storing the Data

```
@app.route("/add_result")
def add_result():
    global g_name, g_otp, g_vector

    new_user = {
        "id": 0, # Placeholder that will be ignored by the API
        "name": g_name,
        "otp": str(g_otp),
        "vector": base64_encoder(g_vector)
    }

    response = requests.post(BASE_URL, json=new_user)

    if response.status_code == 201:
        created_user = response.json()
        print(f"User created: {created_user}")
        tmp_user, tmp_otp = g_name, g_otp
        return render_template("result.html", step=4,
         ↪  operation="add", result="Person added correctly!",
         ↪  username=tmp_user, otp=tmp_otp)
    else:
        print(f"Error creating user: {response.text}")
        return render_template("result.html", step=4,
         ↪  operation="add", result="Error creating user",
         ↪  username=tmp_user, otp=tmp_otp)
```

Code 5.5: Route for storing the new person in the database.

The /add_result route is responsible for saving the new person's data in the database. It first constructs a dictionary `new_user` containing the person's name (stored in `g_name`), the six digit code that identifies the person (stored in `g_otp`), and the Base64-encoded vector representation of their face (obtained by calling `base64_encoder(g_vector)`). The dictionary is then sent as JSON data in a POST request to the API endpoint specified by `BASE_URL`. If the server responds with a status code of 201 (Created), indicating that the new user was successfully added to the database, the function renders an HTML template with a success message, showing the person's name and their six digit code that can be later used for verifying a person or removing it from the database. If the

server responds with any other status code, an error message is displayed in the result.html template.

5.3 Vecotrization

Figure 5.3: Vecotrization of an image.

The image of the face is passed to a function that generates the vector representation using the face_recognition library by encoding the essential features of the face.

```
def get_face_vector(image_path):
    try:
        image = cv2.imread(image_path)
        if image is None:
            print(f"Failed to read image from path:
             ↪  {image_path}")
            return None
        face_locations = face_recognition.face_locations(image)
            if len(face_locations) == 0:
                print("No faces detected in the image.")
                return None

            top, right, bottom, left = face_locations[0]
            face_image = image[top:bottom, left:right]
            image_resized = cv2.resize(face_image, (512, 512))
            face_vector =
             ↪  face_recognition.face_encodings(image_resized)

        if len(face_vector) == 0:
            print("Failed to generate face vector from the
             ↪  detected face.")
            return None
        return face_vector[0]
```

Code 5.6: Generate a vector given the image of a face (Part 1).

```
    except Exception as e:
        print(f"Error occurred during face vector generation:
        ↪  {e}")
        return None
```

Code 5.6: Generate a vector given the image of a face (Part 2).

The function `get_face_vector(image_path)` is used to extract a facial feature vector from a given image. It begins by loading the image specified by `image_path` using OpenCV's `imread()` method. If the image fails to load, the function prints an error message and returns `None`.

5.3.1 Data Preprocessing

Figure 5.4: Preprocessing.

After capturing the image, the next step is to preprocess the image to extract the face characteristics. This is achieved by using the same Haar-like Features algorithm employed during the image capture process.

Next, it identifies the locations of faces within the image using `face_recognition.face_locations()`. If no faces are found, the function returns `None`. If a face is detected, the coordinates (top, right, bottom, left) of the first detected face are used to extract the face region from the image.

This face region is then resized to 512x512 pixels using `cv2.resize()` to ensure uniform resolution, which helps in consistent generation of the facial feature vector. The function then attempts to compute the facial feature vector for the image using `face_recognition.face_encodings()`. If no encoding can be generated, it returns `None`; otherwise, it returns the first encoding found.

```
@app.route("/add_vectorization")
def add_vectorization():
    global g_otp, g_vector

    # ... (camera initialization and image capture)

    img_path = camera.get_captured_path()
    if img_path is None or img_path == "":
        # ...
        return render_template("result.html", error="Image path
         ↪ is not available. Please capture an image first.",
         ↪ step=4, operation="add")

    vectorizer = hf_vectorizer.get_face_vector(str(img_path))
    # ...

    if vectorizer is None:
        return render_template("result.html", error="Failed to
         ↪ generate face vector from the captured image.",
         ↪ step=4, operation="add")

    set_global_vector(vectorizer)
    set_global_otp(generate_otp())

    return render_template("add_vectorization.html", step=3)
```

Code 5.7: Vectorization of an image.

The `/add_vectorization` route handles the vectorization process for the captured image. It first ensures that the camera is initialized and that a valid image path is available from the previous step, and if either of these conditions is not met, an error message is displayed, and the camera and any captured images are cleaned up.

If a valid image path is available, the `hf_vectorizer.get_face_vector()` function is called to generate a vector representation of the captured face image. In case the functions fails to generate a vector, an error message is displayed.

When the vector is successfully generated, it is stored in the global variable `g_vector` using `set_global_vector()`. Additionally, a six-digit code is generated with the `generate_otp()` and stored in the global variable `g_otp`, assigned using the `set_global_otp()` function.

5.3.2 Storing the Vectors

The vectors generated cannot be stored in their original form in the database. Consequently, they are encoded in Base64 and then decoded when retrieved.

```
def base64_encoder(vector):
    face_vector_bytes = vector.tobytes()
    face_vector_base64 =
     ↪  base64.b64encode(face_vector_bytes).decode('utf-8')
    return face_vector_base64
```

Code 5.8: Base64 encoding of the vector.

This function performs the conversion of a vector into a Base64-encoded string. It initiates the process by transforming the NumPy[2] array vector into a bytes object through the `tobytes()` method. The byte representation is then encoded into a Base64 string using the Python function `base64.b64encode()`, which produces a byte string. To make it suitable for storage or transmission as a text string, it is further decoded into a UTF-8 string using the Python method `decode('utf-8')`.

```
def base64_decoder(base64_string):
    decoded_bytes = base64.b64decode(base64_string)
    vector = np.frombuffer(decoded_bytes, dtype=np.float64)
    return vector
```

Code 5.9: Base64 decoding of the vector.

In order to reverse the encoding retrieved from the database and make it usable for verification-related tasks, the Python `base64.b64decode()` module is employed. This is followed by the transformation of the byte sequence into a NumPy array using the `np.frombuffer()` function, with the data type specified as `np.float64` to align with the original vector's format.

[2]NumPy is a library for scientific computing in Python, particularly useful for array and matrix operations.

5.4 Verification of a Person

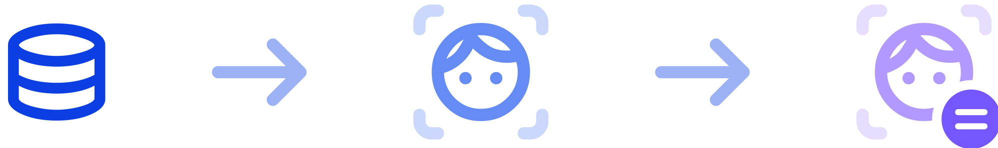


Figure 5.5: Verification of a person.

Another functionality of the application enables users to verify whether two individuals possess the same vector by comparing the vector generated from a captured image with one previously stored in the database.

This process is initiated by the user providing the six digit code associated with their registered facial data. This code is displayed when a face is saved into the database.

```
def get_user_by_otp(otp):
    otp = otp.strip()
    url = f"{BASE_URL}/by_otp/{otp}"
    response = requests.get(url)

    if response.status_code == 200:
        return response.json()
    elif response.status_code == 404:
        return None
    else:
        return None
```

Code 5.10: Get a user from the database, given the six digit code.

The function takes a six digit code as input and sends a GET request to the API endpoint `/users/by_otp/<otp>`, constructed using the `BASE_URL` and the provided six digit code. If the response status code is 200 (OK), the function returns the *JSON*$_G$ representation of the user data. If the response status code is 404 (Not Found), the function returns `None`, indicating that no user with the given code was found in the database. For any other response status code, the function also returns `None`.

When the `get_user_by_otp` function is called, if a user with the given six digit code exists in the database, the response contains the user's information, including their Base64-encoded version of their face vector. The Base64-encoded face vector is decoded using the `base64_decoder` function from the `hf_vectorizer` module, and the resulting vector is stored in the `g_obtained_vector` global variable. Subsequently, the user is prompted to capture an image of the person they want to verify from the `/verify_capture` route. The captured image is processed by the `get_face_vector` function from `hf_vectorizer`, which generates a new face vector. If the vectors match, meaning `compare_vectors` returns True, the verification is considered successful and a success message is displayed to the user. Otherwise, a failure message is shown.

5.4.1 Vectors Comparison

Once the facial image is captured, the `compare_vectors` function is used to compare a vector obtained from the database (`db_vector`) with a vector obtained from from an image (`camera_vector`).

```
def compare_vectors(db_vector, camera_vector, tolerance=0.45):
    if db_vector is None or len(db_vector) == 0 or camera_vector
    ↪  is None:
        return False
    arracy_vct1 = np.atleast_2d(db_vector)
    arracy_vct2 = np.atleast_2d(camera_vector)

    distances = np.linalg.norm(arracy_vct1 - arracy_vct2, axis=1)
    return np.any(distances <= tolerance)
```

Code 5.11: Calculate the distance between two vectors.

Its purpose is to determine if these two vectors are similar enough, based on a given tolerance value. The function takes two mandatory arguments, `db_vector` and `camera_vector`, and an optional `tolerance` value, whose default value is 0.45.

The Euclidean distance metric used in this function measures the straight-line distance between the two vectors in an n-dimensional space, where n is the

length of the vectors. The smaller the distance, the more similar the vectors are.

Initially, the function performs input validation by checking if the `db_vector` is not None, has a non-zero length, and if the `camera_vector` is not None. If any of these conditions are not met, the function returns False, indicating that the vectors cannot be compared.

After the validation of the input, the function converts the input vectors into 2D arrays using NumPy's `np.atleast_2d` function, this conversion is necessary due to the `np.linalg.norm` function, which is used later for distance calculation, that expects a 2D array as input.

Once the conversation is done, the function calculates the Euclidean distances between the `db_vector` and `camera_vector` in an n-dimensional space, given by:

$$d(db_\vec{vector}, camera_\vec{vector}) = \sqrt{\sum_{i=1}^{n}(a_i - b_i)^2}$$

where a_i and b_i are the i-th components of the $db_\vec{vector}$ and $camera_\vec{vector}$ vectors, respectively, and n is the length of the vectors. This calculation is performed using NumPy's `np.linalg.norm` function, along the axis=1 (row-wise) for the 2D arrays, in order to ensure that the distances are calculated between corresponding elements of the vectors.

Finally, the function checks if any of the calculated distances are less than or equal to the specified `tolerance` value. If at least one distance satisfies this condition, the function returns True, indicating that the vectors are similar within the given tolerance. Otherwise, it returns False.

```
@app.route("/verify_check")
def verify_check():
    global g_obtained_vector

    # ... (camera initialization and image capture)

    generated_vector =
    ↪  hf_vectorizer.get_face_vector(str(img_path))
    # ...

    if g_obtained_vector is None:
        return render_template("result.html", error="No face
        ↪  detected during verification.", step=3,
        ↪  operation="verify")
```

Code 5.12: Handling the result of the verification process (Part 1).

```
    if generated_vector is None:
        return render_template("result.html", error="Failed to
        ↪  generate face vector from the captured image.",
        ↪  step=3, operation="verify")

    if not compare_vectors(g_obtained_vector, generated_vector):
            print("Verification failed.")
            return render_template("result.html",
            ↪  result="Verification failed.", step=3,
            ↪  operation="verify")
        else:
            print("Verification successful.")
            return render_template("result.html",
            ↪  result="Verification successful!", step=3,
            ↪  operation="verify")
```

Code 5.12: Handling the result of the verification process (Part 2).

Through the `/verify_check` route, the captured image is processed by the `get_face_vector` function from the `hf_vectorizer` module to generate a new face vector. The `compare_vectors` function from the same module is used to compare the newly generated face vector with the decoded vector obtained from the database (stored in `g_obtained_vector`). If the vectors match, meaning `compare_vectors` returns True, a success message is then displayed to the user, otherwise if they do not match, a failure message is shown instead.

5.5 Removal of a Person

Figure 5.6: Remove a person from the database.

The application also provides a mechanism for removing a person's facial data from the database. Similar to the verification process, the user is first prompted to enter the six digit code of the person to remove.

```
def delete_user_by_otp(otp):
    url = f"{BASE_URL}/by_otp/{otp}"
    response = requests.get(url)

    if response.status_code == 200:
        user = response.json()
        user_id = user["id"]
        delete_url = f"{BASE_URL}/{user_id}"
        delete_response = requests.delete(delete_url)

        if delete_response.status_code == 200:
            print(f"User with OTP {otp} deleted successfully.")
        else:
            print(f"Failed to delete user with OTP {otp}. Error:
            ↪  {delete_response.text}")
    else:
        print(f"User with OTP {otp} not found.")
```

Code 5.13: Deletion of a user, given an six digit code.

The `delete_user_by_otp` handles the deletion of a user from a database by leveraging on the code given to the user as an identifier. It begins by constructing a URL using the provided six digit code and sending a GET request to an endpoint defined by `BASE_URL`. This request aims to retrieve user details associated with the given code from the database. If the server responds with a status code of 200, it indicates the user has been found, and it proceeds to extract the user ID from the JSON response.

With the user ID in hand, the function constructs the same URL as for the verification process and issues a DELETE request to this endpoint. If this request is successful, confirmed by a 200 status code, the it prints a confirmation message stating the user has been deleted successfully. However, if the deletion request fails or if the initial GET request does not find a user (i.e., it returns a status code other than 200), it prints an error message. If a user with the given six digit code exists, the Base64-encoded face vector obtained from the database is decoded using `base64_decoder` and the resulting vector is stored in the `g_obtained_vector` global variable, as well as the code provided.

```
@app.route("/delete_check")
def delete_check():
    global g_obtained_vector, g_otp
    # ...

    if not compare_vectors(g_obtained_vector, generated_vector):
        print("Deletion failed.")
        return render_template("result.html", result="Deletion
         ↪  failed.", step=3, operation="delete")
    else:
        delete_user_by_otp(g_otp)
        print("Deletion successful!")
        return render_template("result.html", result="Deletion
         ↪  successful!", step=3, operation="delete")
```

Code 5.14: Handling the result of the verification process.

The `/delete_check` route works in the same way as the verification one, the only difference being what they do at the end. In this case, if the vectors matches, the `delete_user_by_otp` function is called to delete the user with the given six digit code from the database.

Chapter 6

Testing and Validation

After having completed the development of the application, it was really important to check the system's reliability, accuracy, and performance by using various testing methodologies, including unit testing, integration testing, and system testing, as well as the validation processes used to verify the application's compliance with requirements and its effectiveness in real-world scenarios.

6.1 Unit Testing

Unit testing involves trying individual components or functions in isolation by confirming that each unit of code behaves as expected. In the context of this project, four key components have been tested: **Vectorization**, **API Calls**, **Utils** and **Routing**.

6.1.1 Vectorization

The vectorization process is responsible for extracting facial features from images and converting them into vectors. Unit tests for this process covered scenarios like vector generation, handling of images, and proper error management, for example in case failures during vector generation.

```
def test_get_face_vector_no_face(self):
    with patch('cv2.imread', return_value=np.random.randint(0,
    ↪   256, (100, 100, 3), dtype=np.uint8)):
        with patch('face_recognition.face_locations',
        ↪   return_value=[]):
            vector = hf_vectorizer.get_face_vector('test.jpeg')
            self.assertIsNone(vector)
```

Code 6.1: Test for generating the vector form an image without a face.

```
def test_get_face_vector_valid_image(self):
        with patch('cv2.imread',
        ↪   return_value=np.random.randint(0, 256, (100, 100, 3),
        ↪   dtype=np.uint8)):
            with patch('face_recognition.face_locations',
            ↪   return_value=[(10, 30, 60, 20)]):
                with patch('face_recognition.face_encodings',
                ↪   return_value=[np.random.rand(128)]):
                    vector =
                    ↪   hf_vectorizer.get_face_vector('test.jpeg')
                    self.assertIsInstance(vector, np.ndarray)
                    self.assertEqual(vector.shape, (128,))
```

Code 6.2: Test for generating the vector form an image.

6.1.2 API Calls

The system interacts with external APIs for CRUD operations with the database. The unit tests for API calls focused on verifying the correct handling of successful and failed API responses, as well as edge cases like user not found or invalid inputs.

```
@patch('app.requests.get')
def test_get_user_by_otp_not_found(self, mock_get):
    mock_response = MagicMock()
    mock_response.status_code = 404
    mock_get.return_value = mock_response

    user = get_user_by_otp('invalid_otp')
    self.assertIsNone(user)
```

Code 6.3: Test for return an error when a user is not found.

```
@patch('app.requests.get')
def test_get_user_by_otp_success(self, mock_get):
    mock_response = MagicMock()
    mock_response.status_code = 200
    mock_response.json.return_value = {'id': 1, 'name': 'John
    ↪  Doe', 'otp': '123456', 'vector': 'base64vector'}
    mock_get.return_value = mock_response

    user = get_user_by_otp('123456')
    self.assertEqual(user, {'id': 1, 'name': 'John Doe', 'otp':
    ↪  '123456', 'vector': 'base64vector'})
```

Code 6.4: Test for saving a person to the database.

6.1.3 Utils

Utility functions play a key role in supporting various operations within the system. And unit testing for these functions covered areas such as camera initialization and sharing, image file management, and helper functions for encoding and decoding face vectors.

```
@patch('app.VideoCamera')
    def test_initialize_camera_success(self, mock_video_camera):
        mock_instance = MagicMock()
        mock_video_camera.return_value = mock_instance
        mock_instance.camera_status = "Success"
        result = initialize_camera()
        self.assertEqual(result, "(app.initialize_camera)Camera
         ↪  initialized successfully.")
```

Code 6.5: Simulating the correct initialization of the camera.

```
@patch('app.VideoCamera')
def test_initialize_camera_error(self, mock_video_camera):
    mock_instance = MagicMock()
    mock_video_camera.return_value = mock_instance
    mock_instance.camera_status = "Error"
    result = initialize_camera()
    self.assertEqual(result, "(app.initialize_camera)Error:
     ↪  Failed to open the camera.")
```

Code 6.6: Simulating an error while initializing the camera.

6.1.4 Routing

The system's routes handle user interaction and request processing. Unit testing for routing focuses on verifying the correct handling of various HTTP methods (GET, POST, etc.) by simulating client requests and validating responses, testing the behavior of route functions under various input conditions, and ensuring proper redirection and rendering of templates.

```
def test_add_name_get(self):
    response = self.app.get('/add_name')
    self.assertEqual(response.status_code, 200)
```

Code 6.7: Test for adding a person with a POST call (Part 1).

```
@patch('app.set_global_name')
def test_add_name_post(self, mock_set_global_name):
    data = {'username': 'John Doe'}
    response = self.app.post('/add_name', data=data,
     ↪  follow_redirects=True)
    self.assertEqual(response.status_code, 200)
    mock_set_global_name.assert_called_once_with('John Doe')
```

Code 6.7: Test for adding a person with a POST call (Part 2).

6.2 Integration Testing

The purpose of integration testing is to verify the correct interaction between different components or modules of a system, with the goal of identifying potential problems that may arise from the integration of multiple components. All of the following tests have been tested with different people under different conditions to cover a comprehensive range of scenarios.

6.2.1 Testing the Registration Process

The first test to do is to validate the end-to-end user registration process, being at the center of the application. This process involves several steps:

- Capturing an image of the user using the camera;
- Generating a face vector from the captured image using the face vectorizer;
- Saving the generated face vector and the user's information in the database.

This ensures that the camera, face vectorizer, and database processes work together in an integrated manner.

6.2.2 Testing the Verification Process

The second integration test is to validate the user verification process. This process involves:

- Capturing a new image of the user using the camera;
- Retrieving the stored face vector of the user from the database from the given six digit code;
- Generating a face vector from the new image;
- Comparing the new face vector with the retrieved face vector to verify the user's identity.

By validating that the system is capable of accurately comparing face vectors and verifying user identities, we can confirm that the user verification process is reliable.

6.2.3 Testing the Deletion Process

The final integration test involves the user deletion process. This process includes:

- Capturing a new image of the user using the camera;
- Retrieving the stored face vector of the user from the database from the given six digit code;
- Removing the user's face vector and related information from the database;
- Ensuring that any references to the user's data in other components are properly handled and do not cause errors.

The integration test verifies that the database operations for deleting user data are executed correctly and that all components are able to handle the absence of the user's data.

6.3 System Testing

System testing involves testing the entire system as a whole, simulating real-world scenarios and user interactions, which helps in identifying issues that may arise from the interaction of different components and the overall system behavior.

6.3.1 Performance Testing

The objective of performance testing is to evaluate the system's performance under various conditions, including high user load, large amounts of data, and resource-constrained environments.

The application has been tested on both a traditional computer and a Raspberry Pi 5, which was the aim of this paper. In both cases, the application performed as expected and was able to complete the tasks without requiring significant resources.
Furthermore, to confirm the results, the application was also tested on a Raspberry Pi 4, a less powerful model than the Pi 5. The application successfully compiled and ran as expected.

6.3.2 Compliance

The initial goal of the project was to develop a facial recognition system, but after discussions with stakeholders, the focus shifted to implementing a facial verification system. This change was influenced by several factors, including the recent enactment of the AI Act[1], which highlighted important regulatory considerations, as well as an assessment of the company's existing projects, which revealed a stronger focus on verification technologies, and the available support and resources within the company were found to be more suitable for a verification system.

[1] European Commission. *AI Act | Shaping Europe's digital future*. URL: https://digital-strategy.ec.europa.eu/en/policies/regulatory-framework-ai. (Accessed: June 12, 2024)

6.4 Accuracy of the Algorithm

A key aspect was to define the accuracy of the algorithm used in the project. To do this, a binary classification test was performed based on the proportion of correct predictions (both true positives and true negatives) out of the total number of cases examined.
The accuracy is calculated as:

$$Accuracy = \frac{TP+TN}{TP+TN+FP+FN}$$

where:

- **True Positive (TP)**: Correct identification of the right person;
- **True Negative (TN)**: Correct rejection when no match should be found;
- **False Positive (FP)**: Incorrect match with the wrong person (Type I error);
- **False Negative (FN)**: Failure to match with the correct person (Type II error).

This metric represents the ratio of correct predictions (both true positives and true negatives) to the total number of cases evaluated, providing a straightforward interpretation of overall performance.

The process relies on the Face-Dataset—2400-IMG-and-80-LABELS dataset, which comprises:

- 2,400 total images;
- 80 distinct classes (famous individuals);
- 30 images per class.

6.4.1 Implementation

```
def generate_reference_vectors(image_folder):
    reference_vectors = {}
    for person in range(1, 81):
        image_path = os.path.join(image_folder,
         ↪  f"{person}_1.png")
        if os.path.exists(image_path):
            vector = get_face_vector(image_path)
            if vector is not None:
                reference_vectors[person] = vector
            else:
                print(f"Failed to generate vector for person
                 ↪  {person}")
        else:
            print(f"Reference image for person {person} not
             ↪  found")
    return reference_vectors
```

Code 6.8: Generating the base face vector for the verification process.

The `generate_reference_vectors` function creates a dictionary of reference vectors for the 80 individuals in the dataset. It iterates through person IDs 1 to 80, attempts to load the first image (`"person_1.png`) for each person, and generates a face vector using the `get_face_vector` function. Successfully generated vectors are stored in the dictionary with the person's ID as the key. The function also handles the case where images are missing or vector generation fails, by providing appropriate error messages.

```python
def test_accuracy(image_folder, reference_vectors):
    results = {'TP': 0, 'FP': 0, 'TN': 0, 'FN': 0}
    processed_images = 0
    failed_images = 0
    correct_match = False

    for filename in os.listdir(image_folder):
        person_id = int(filename.split('_')[0])
        if f"{person_id}_1" in filename:
            continue

        test_vector = get_face_vector(os.path.join(image_folder,
        ↪   filename))
        if test_vector is None:
            failed_images += 1
            continue

        processed_images += 1

        for ref_person, ref_vector in reference_vectors.items():
            is_match = compare_vectors(ref_vector, test_vector,
            ↪   tolerance=0.5)
            correct_match = (results['TP'] += 1 if person_id ==
            ↪   ref_person and is_match else results['FP'] += 1
            ↪   if not person_id == ref_person and is_match else
            ↪   correct_match)

            if correct_match:
                break
```

Code 6.9: Evaluation of the performance (Part 1).

```python
        if not correct_match:
            results['FN'] += 1 if person_id in reference_vectors
            ↪   else results['TN'] += 1
```

Code 6.9: Evaluation of the performance (Part 2).

The `test_accuracy` function evaluates the performance of the face verification system. It processes all images in the specified folder (excluding the reference images), generates face vectors for each, and compares them against the reference vectors.

The function tracks **TP**, **FP**, **TN**, and **FN**. It handles cases where face vector generation fails and provides a comprehensive assessment of the system's accuracy across all test images.

```
def calculate_accuracy(results):
    total = sum(results.values())
    accuracy = (results['TP'] + results['TN']) / total if total >
    ↪ 0 else 0

    print("Overall Results:")
    print(f"True Positives: {results['TP']}")
    print(f"True Negatives: {results['TN']}")
    print(f"False Positives: {results['FP']}")
    print(f"False Negatives: {results['FN']}")
    print(f"Accuracy: {accuracy:.2%}")
```

Code 6.10: Calculating the accuracy of the system from the given data.

Finally, the `calculate_accuracy` function computes and shows the overall performance metrics of the face recognition system. It calculates the accuracy as the ratio of correct predictions (True Positives and True Negatives) to the total number of predictions.

6.4.2 Results

Tolerance	TP	FP	FN	TN	Accuracy
0.45	513	114	1	595	51.27%
0.50	803	113	4	303	74.90%
0.55	945	98	52	128	85.28%
0.60	837	59	302	25	73.26%

Table 6.1: Table showing the accuracy of the system as the tolerance changes.

We can see that the face recognition system's performance varies significantly with different tolerance levels. At a strict tolerance of *0.45*, which is the default value of the `compare_vectors` function, the system shows a low accuracy (51.27%) due to numerous false negatives. As the tolerance increases to

0.55, the accuracy peaks at 85.28%, striking an optimal balance between true positives and false negatives. However, further increasing the tolerance to 0.60 leads to a decline in accuracy (73.26%) as false positives spike.

At the end, we can see that the optimal level of tolerance is *0.55*, in order to maximizes the system's ability to correctly identify individuals while minimizing both false positives and negatives, demonstrating the delicate balance required in tuning such systems for real-world applications.

Chapter 7

Conclusions

7.1 Achieved Goals

Requirement	Status
ROB-1	Achieved.
ROB-2	Achieved
ROB-3	Achieved.
ROB-4	Achieved.
RDE-1	Achieved.
RDE-2	Achieved
RDE-3	Achieved.
RDE-4	Achieved.
ROP-1	Partially Achieved.
ROP-2	Partially Achieved.

Table 7.1: Table showing the status of the requirements.

All mandatory and desirable requirements were achieved and implemented in the application. On the other hand, the optional requirements were partially accomplished. In the case of the first one, **ROP-1**, due to the nature of the product, which is a Progressive Web App, it cannot be considered a true native application. The second one, **ROP-2**, was partially fulfilled because the Google's Coral TPU that was provided by the company, is specifically designed

to work with Google-related libraries and frameworks, such as TensorFlow, and even after testing and trying to work with the desired device, no real advantage where shown in the project, which uses OpenCV and a *pyhtonic* implementation of the Dlib library for computer vision related tasks.

7.2 Future Development

The *DeVisu* application has laid a solid foundation for a facial verification-based system, where the algorithms and architecture implemented in the project have demonstrated the feasibility of leveraging computer vision techniques on computationally limited hardware. However, this could be a beginning for a broader paradigm shift towards seamless and intuitive identity management solutions.

In the current form of the project, the face verification process is performed on the same device where the user data is stored, limiting the utility of the application to a single device, which is not practical in real-world scenarios. To address this limitation, a possible solution could be a centralized server to store user data and facilitate communication between devices.

One such promising endeavor of this paradigm is the HandsFree project, currently under development at DuckMa, which aims to harness the advantages of face verification associated to a unique *tokenization* system for a hands-free interaction. The foundation laid by *DeVisu* could serve as a stepping stone for this ambitious project, accelerating its development and ensuring a robust, secure, and user-friendly solution.

7.3 Knowledge Acquired

I had the opportunity to develop and consolidate several technical skills in the field of software development. In particular, I dedicated myself to a comprehensive study of the technologies concerning the implementation of facial verification systems, and improved the ability to research and identify solutions emerged as an essential component of my training during the internship. I also

enhanced my proficiency in effectively managing information, utilizing research tools and deepening my critical analysis skills, enabling me to identify appropriate solutions to some of the challenges encountered during the development of the *DeVisu* project.

Finally, the internship has proven to be effective in applying the knowledge that I have acquired during my studies and to gain new skills. In particular, I have been able to deepen my understanding of software development, the management of resources, and computer vision.
Furthermore, I had the chance to work in a professional environment, interacting with experienced colleagues, and to learn new working and organizational methodologies.

7.4 Final Considerations

In conclusion, the internship at DuckMa, which I am grateful in light of the willingness shown ad the opportunity given, represents a significant experience for my professional and personal development. The technical skills acquired, the ability to research and identify solutions, the ability to organize work, and the effective interaction with the reference figure collectively contribute to an enhancement of my profile and the advancement of my educational trajectory.

Acronyms and abbreviations

API Application Programming Interface. 32

ASGI Asynchronous Server Gateway Interface. 26

CNN Convolutional Neural Networks. 8

CRUD Create, Read, Update, Delete. 33

PoC Proof of Concept. iv

SQL Structured Query Language. 26

WSGI Web Server Gateway Interface. 24

Glossary

Application Programming Interface A set of rules and protocols for building and interacting with software applications. APIs specify how software components should interact and allow different software systems to communicate with each other. 32

Artificial Neural Network Artificial Neural Networks are computational models inspired by the human brain's network of neurons, consisting in interconnected nodes, or neurons, organized in layers: an input layer, one or more hidden layers, and an output layer. Each neuron receives inputs, processes them using weights and biases, and passes the result through an activation function to determine the output. 7

Asynchronous Server Gateway Interface Specification that serves as a standard interface between asynchronous Python web servers, applications, and frameworks, by extends the capabilities of the WSGI to support asynchronous communication patterns. 26

Convolutional Neural Network Convolutional Neural Networks are a class of deep learning algorithms primarily used for processing and analyzing visual data, designed to automatically and adaptively learn spatial hierarchies of features from input images. 8

CRUD Refers to the four basic operations of persistent storage, which are Create, Read, Update, and Delete. These operations are at the foundation of database management and are used in software development to manage data. 33

Database A database is an organized collection of data that can be easily accessed, managed, and updated. It stores information in a structured format, typically using tables, to facilitate efficient retrieval and manipulation of data by various users and applications. 5

Deep Learning Deep learning is a subset of machine learning, which itself falls under the broader umbrella of artificial intelligence. It involves training artificial neural networks to recognize patterns and make decisions based on a large datasets. Deep learning models, often consisting of many layers (hence the word "deep"), can automatically learn to extract complex features from raw input data. 8

JSON A lightweight data interchange format that is easy for humans to read and write and easy for machines to parse and generate. It is based on a subset of the JavaScript Programming Language. 41

Proof of Concept A Proof of Concept is a preliminary demonstration to showcase the feasibility and potential of a new idea, technology, or product. It aims to verify that the concept can be successfully developed and implemented, helping stakeholders assess its viability before committing significant resources. iv

Serverless The term "serverless" refers to a type of architecture in which the cloud provider dynamically manages the allocation of machine resources. A serverless application runs in stateless compute containers that are event-triggered, ephemeral (may last for only one invocation), and fully managed by the cloud provider. 26

SQL A standardized language used for managing and manipulating relational databases. It is used for tasks such as querying data, updating records, inserting new data, and defining database structures. 26

Way of Working The Way of Working is a set of principles, practices, and methodologies that guide how individuals and teams approach their tasks

and collaborations within an organization. It encompasses elements such as communication protocols, decision-making processes, project management frameworks, and cultural norms. 1

Web Server Gateway Interface Protocol that establishes and describes communications and interactions between servers and web applications written in Python. 24

Bibliography

Books

Palma, M. and W. Maraschini. *Enciclopedia della matematica*. Le Garzantine. Garzanti, 2013, pp. 560, 866–867. ISBN: 9788811505259. URL: https://books.google.it/books?id=4Y0zLgEACAAJ.

Articles

Fukushima, Kunihiko. "Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position". In: *Biological cybernetics* 36.4 (1980), pp. 193–202. DOI: 10.1007/BF00344251.

Haar, Alfred. "Zur Theorie der orthogonalen Funktionensysteme". In: *Mathematische Annalen* 69.3 (Sept. 1910), pp. 331–371. ISSN: 1432-1807. DOI: 10.1007/BF01456326 (cit. on p. 11).

Roberts, Lawrence Gillman. "Machine Perception of Three-Dimensional Solids". In: (July 1963). URL: https://dspace.mit.edu/handle/1721.1/11589.

Viola, P. and M. Jones. "Rapid object detection using a boosted cascade of simple features". In: (2001). DOI: 10.1109/CVPR.2001.990517 (cit. on p. 11).

Webliography

Django Software Foundation. *The Django template language*. URL: https://docs.djangoproject.com/en/5.0/ref/templates/language/. (Accessed: June 5, 2024).

European Commission. *AI Act | Shaping Europe's digital future*. URL: https://digital-strategy.ec.europa.eu/en/policies/regulatory-framework-ai. (Accessed: June 12, 2024) (cit. on p. 54).

IBM. *What is computer vision?* URL: https://www.ibm.com/topics/computer-vision. (Accessed: May 26, 2024) (cit. on p. 7).

OpenCV. *OpenCV: Cascade Classifier*. URL: https://docs.opencv.org/3.4/db/d28/tutorial_cascade_classifier.html. (Accessed: May 20, 2024).

Pallets Project. *Flask Documentation*. URL: https://flask.palletsprojects.com/en/3.0.x/. (Accessed: May 19, 2024).

Regula. *Face Recognition vs. Face Verification in Identity Verification: The Expert Explanation*. URL: https://regulaforensics.com/blog/face-recognition-vs-face-verification/. (Accessed: May 31, 2024).

Shufti Pro. *Understanding Facial Identification: Face Verification vs. Face Recognition*. URL: https://shuftipro.com/blog/understanding-facial-identification-face-verification-vs-face-recognition/. (Accessed: May 29, 2024).

Macte nova virtute, puer, sic itur ad astra.