Verificación en escenarios multinivel con presencia de robots asistentes

Proyecto creado por Kevin David Rosales Santana

Supervisado por:
Modesto Fernando Castrillón Santana
José Javier Lorenzo Navarro

Grado en Ingeniería Informática
Julio de 2020

Agradecimientos

A mi familia y amigos, quienes no han parado de ser los pilares fundamentales con los que compartir mis experiencias en esta carrera académica y personal.

Al PDI y el PAS, por su constante contribución al aprendizaje del alumnado y la adaptación a la modalidad virtual en estos últimos meses. En especial, a mis tutores por la paciencia que han tenido guiándome a lo largo de la mención y en el presente proyecto.

Al personal sanitario, quienes han estado luchando día tras día sin descanso en primera línea de batalla contra el COVID-19.

Resumen

La introducción de robots de servicio como asistentes en edificios enfrentándose a problemas diferentes a los existentes actualmente, como el guiado de personas en una única planta, será una realidad en años venideros. Uno de estos es la cooperación bajo interacción hombre-máquina entre varios robots ubicados en diversos pisos de dicho edificio, planteando una comunicación donde uno, localizado en una planta distinta, reconozca a la persona orientada inicialmente en otra entre varias, continuando la colaboración. Para resolver este problema de verificación de identidad se usará el conjunto de datos AveRobot, cuyas condiciones de captura no ideales implican un reto que será encarado estudiando tecnologías biométricas faciales y analizando técnicas de detección, generación y distancia de vectores descriptores o redes neuronales.

Abstract

Service robots incorporation as building assistants confronting different problems from currently existing ones, like the person guidance on a single floor, will be a reality in the incoming years. One of these problems is the cooperation under human-robot interaction between various robots located on distinct building levels, considering a communication where one, located on a different floor, is capable of recognising a person initially assisted on another one between various individuals to continue the collaboration. AveRobot dataset will be used in order to solve this identity verification problem, whose non-ideal recording conditions imply a challenge that will be faced studying facial biometrics technologies and analysing detection, embedding generation and distance or neural networks techniques.

Índice general

Índice de figuras

Índice de tablas

Índice de fragmentos de código

Capítulo 1

Introducción

> By far the greatest danger of Artificial Intelligence is that people conclude too early that they understand it.
>
> Eliezer Yudkowsky [1]

Uno de los campos que genera más interés dentro de la inteligencia artificial es la interacción hombre-máquina, la cual es capaz de plantear retos muy variados en escenarios diversos. Este proyecto trata de resolver un problema de verificación de identidad a través de personas que realizan una interacción de este estilo hacia unos robots a los cuales se les desea otorgar la capacidad de diferenciar si dos personas son la misma o no haciendo uso de cámaras de vídeo que incorporan como sensores.

Por ello, asumiendo que la presencia de robots asistentes será más común en edificios públicos, se considera que se tienen varios pisos o plantas y que se quiere realizar la tarea de verificación previamente mencionada entre plantas. En otras palabras, se pretende verificar que la persona que ha sido captada en un piso del edificio equipado con esta tecnología, sea la que se esté captando en otro piso.

La finalidad es proporcionar un nivel de seguridad tecnológica que no deba requerir de otros medios actuales ya estudiados en profundidad por la comunidad científica y habitualmente usados como sensores biométricos fisiológicos basados en huella digital o reconocimiento de iris, ni requerir la participación (e incluso el contacto, siendo este limitado en la época de crisis sanitaria en la que se encuentra la humanidad sumergida por el SARS-CoV-2) que dichas técnicas requieren.

Por este motivo, se propone realizar la tarea de verificación mediante el rostro. Se considera esta opción y no cualquier otra que use imágenes de cuerpo entero debido a que al ser una verificación en escenarios multinivel, se debe contar con el hecho de que una persona pueda cambiar levemente su atuendo a medida que cambia de piso. Por ejemplo, se podría pensar en un individuo que se equipara con un abrigo (o se desprendiera de este) de camino hacia otra planta.

Por otro lado, también se pretende realizar un estudio del alcance que puede llegar a lograr un reconocimiento usando únicamente elementos faciales, usando diferentes técnicas de recogida de fotogramas, de detección, de obtención de descriptores, pasando por la gestión de volúmenes de información amplios para evaluar de distintas maneras y de forma exhaustiva sistemas basados en cálculo de matrices de distancias euclídeas o coseno o basados en redes neuronales a través de las mejores muestras recogidas por los robots asistentes del par de vídeos a verificar.

Se debe destacar que la metodología que se ha usado a lo largo del proyecto se encuentra basada en prototipos, mediante los cuales se han añadido diferentes modificaciones y módulos que han permitido ir obteniendo resultados variados en función de las conclusiones analizadas.

Para llevar a cabo dicho estudio, se ha utilizado el conjunto de datos *AveRobot* [2] (ver figura 1.1), el cual fue recopilado en el Edificio Departamental de Informática y Matemáticas en otoño de 2018, formando la captura de 111 individuos con 8 sensores distintos distribuidos en tres pisos del módulo III. Se explicará de forma más profunda el conjunto de datos en el apartado 5.1 de la presente memoria.

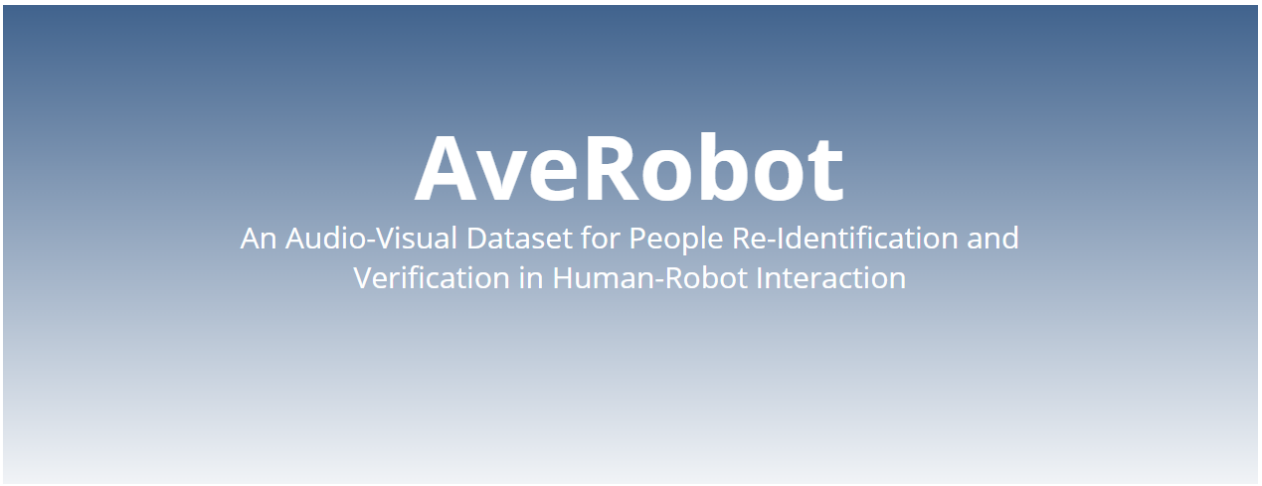


Figura 1.1: Página oficial del conjunto de datos *AveRobot* [3]

1.1 Normativa y Legislación

En el presente proyecto, como se relatará en profundidad a lo largo de esta memoria, se muestran imágenes pertenecientes al conjunto de datos titulado *AveRobot*, debiendo mencionar que no se vulneran los derechos de imagen de las personas físicas utilizadas en el documento.

El artículo 18, apartado 1, de la Constitución Española consta de la siguiente afirmación: “Se garantiza el derecho al honor, a la intimidad personal y familiar y a la propia imagen”.

A estos efectos, la Ley Orgánica 3/2018, de 5 de diciembre de Protección de Datos Personales y garantía de los derechos digitales establece en el Artículo 6 “Tratamiento basado en el consentimiento del afectado” del Título II lo siguiente:

> *1. De conformidad con lo dispuesto en el artículo 4.11 del Reglamento (UE) 2016/679, se entiende por consentimiento del afectado toda manifestación de voluntad libre, específica, informada e inequívoca por la que este acepta, ya sea mediante una declaración o una clara acción afirmativa, el tratamiento de datos personales que le conciernen.*
>
> *2. Cuando se pretenda fundar el tratamiento de los datos en el consentimiento del afectado para una pluralidad de finalidades será preciso que conste de manera específica e inequívoca que dicho consentimiento se otorga para todas ellas.*
>
> *3. No podrá supeditarse la ejecución del contrato a que el afectado consienta el tratamiento de los datos personales para finalidades que no guarden relación con el mantenimiento, desarrollo o control de la relación contractual.*

Todas las personas físicas que se mostrarán en el documento han firmado un acuerdo de consentimiento de divulgación de sus imágenes originadas en el conjunto de datos previamente mencionado para fines académicos o científicos como el presente Trabajo de Fin de Grado. Dichos acuerdos se encuentran almacenados a cargo de uno de los tutores de este proyecto, José Javier Lorenzo Navarro.

Capítulo 2

Objetivos, competencia específica y planificación temporal

Los objetivos trabajados y cumplidos a lo largo del proyecto (de acuerdo con el documento TFT01) son los que se listan a continuación:

- Estudio de las tecnologías disponibles para la detección y caracterización de personas.
- Diseño del conjunto experimental.
- Adaptación de las técnicas de verificación al contexto multinivel.
- Evaluación de técnicas de verificación de la identidad.
- Pruebas de rendimiento y validación.

Siguiendo el documento TFT01, se ha cubierto la siguiente competencia específica del título principalmente (se adjunta la correspondiente justificación):

Competencia Específica	**Definición**	**Justificación**
CP04	Capacidad para conocer los fundamentos, paradigmas y técnicas propias de los sistemas inteligentes y analizar, diseñar y construir sistemas, servicios y aplicaciones informáticas que utilicen dichas técnicas en cualquier ámbito de aplicación.	Se ha construido un sistema basado en conocimientos adquiridos sobre los sistemas inteligentes tras un análisis previo sobre sus requisitos y un diseño basado en iteraciones para su correcta funcionalidad. Dichas técnicas permiten que el sistema haya obtenido la capacidad para verificar en escenarios multinivel.

Tabla 2.1: Competencia específica principal cubierta

Además, de acuerdo al mismo documento TFT01, se ha cumplido con la siguiente planificación temporal estimada:

Fases	**Duración estimada (horas)**	**Tareas**
Estudio previo / Análisis	50	Tarea 1.1: Estudio de las tecnologías para detección de personas en contextos de interacción hombre-máquina
		Tarea 1.2: Familiarización con las tecnologías de identificación de personas
		Tarea 1.3: Familiarización con técnicas de clasificación
Diseño / Desarrollo / Implementación	150	Tarea 2.1: Verificación basada en información facial, del cuerpo y audio
		Tarea 2.2: Adaptación al contexto multinivel
		Tarea 2.3: Combinación de pistas
Evaluación / Validación / Prueba	70	Tarea 3.1: Diseño de experimentos de validación
		Tarea 3.2: Evaluación de resultados
Documentación / Presentación	30	Tarea 4.1: Redacción de la memoria del TFG
		Tarea 4.2: Realización de la presentación
		Tarea 4.3: Ensayos de la presentación

Tabla 2.2: Planificación temporal

Donde se debe destacar que en el desarrollo finalmente se centró el proyecto en la verificación basada en información facial exclusivamente, desechando la verificación basada en información del cuerpo y audio.

Capítulo 3

Herramientas usadas

A la hora de describir las herramientas utilizadas en el proyecto, se dividirá por su cometido distinto en tres apartados: herramientas de desarrollo, herramientas de gestión y equipo de trabajo.

3.1 Herramientas de desarrollo

En el caso de las herramientas de desarrollo, se diferenciará entre herramientas generales y bibliotecas principales (o *libraries*).

3.1.1 Herramientas generales

3.1.1.1 Python

Python [4] (ver figura 3.1) es un lenguaje de programación interpretado de alto nivel creado en 1991 por Guido van Rossum cuya principal característica es la mejora de la legibilidad en su código. Además, es multiparadigma, con soporte de programación funcional, orientada a objetos e imperativa. Se debe mencionar que se trata de un lenguaje de programación de tipado dinámico y multiplataforma.

Actualmente, se encuentra gestionado por *Python Software Foundation* [5], ofreciendo una licencia de código abierto (*Python Software Foundation License* [6]) compatible con la Licencia Pública General de GNU a partir de la versión 2.1.1.

Python se ha convertido en uno de los lenguajes más usados en ciencia de datos y en inteligencia artificial debido principalmente a la cantidad de bibliotecas relacionadas que se encuentran disponibles [7], como las mostradas en la sección 3.1.2.

En este proyecto, se ha hecho uso de la versión 3.7.1.

Figura 3.1: Logotipo de *Python* [Fuente]

3.1.1.2 Jupyter Notebook

Jupyter Notebook [8] (ver figura 3.2) es un entorno informático interactivo web. Un documento *Jupyter Notebook* (con extensión `.ipynb`) se trata de un documento JSON que contiene una lista de celdas de entrada o salida cuyo contenido puede ser de tipo código, texto (usando *Markdown* [9]), fórmulas matemáticas, gráficos...

Un documento *Jupyter Notebook* puede generar varios formatos de salida como *HTML*, *LaTeX*, *Markdown*, *PDF*, *Python*... Por este motivo, resultó de gran utilidad a la hora de presentar los resultados de una forma cómoda y ordenada haciendo uso de los diferentes módulos base que contiene el proyecto.

Jupyter Notebook puede conectarse como entorno a diferentes núcleos (*kernels*) para permitir la programación en diversos lenguajes. En este proyecto, se conectó al núcleo por defecto, el cual permite programar en *Python*. Al gestionar el núcleo a lo largo del documento, se logra una cierta optimización de los recursos informáticos del equipo informático que lo ejecuta.

Figura 3.2: Página oficial de *Project Jupyter* [Fuente]

3.1.1.3 Git

Como se menciona en el apartado 3.2.2, *Git* [10] es un software de control de versiones creado por Linus Torvalds en 2005 para el desarrollo del núcleo (*kernel*) de Linux con el fin de organizar el mantenimiento de versiones de aplicaciones (en especial, cuando adquieren un número razonable de ficheros de código fuente o existen varios programadores trabajando en un mismo proyecto).

En consecuencia, su objetivo consiste en llevar un registro de los cambios en dichos ficheros y en coordinar estas modificaciones de forma eficiente, manteniendo la integridad de los datos

y un soporte para flujos de trabajo no lineales mediante ramas (ver figura 3.3). Se trata de un software libre que se encuentra bajo los términos de la versión 2 de la Licencia Pública General de GNU [11].

Figura 3.3: Ejemplo de ramas (*branches*) en un control de versiones de *Git* [Fuente]

3.1.1.4 IntelliJ IDEA Ultimate

IntelliJ IDEA [12] (ver figura 3.4) es un IDE (entorno de desarrollo integrado) destinada al desarrollo de programas informáticos creado por *JetBrains* [13] en 2001. En este caso, se ha hecho uso de la edición comercial o *Ultimate* [14] de forma gratuita gracias a la situación de estudiante universitario.

Su elección se origina en la cantidad de facilidades que ofrece a la hora de trabajar y a la experiencia obtenida a lo largo de los últimos años en su uso. Mediante extensiones (*plugins*) se puede obtener soporte para el lenguaje principal usado en este proyecto: *Python*. Además, también ofrece compatibilidad con Jupyter Notebook e interfaz para control de versiones (en este caso, realizada mediante *Git* y repositorios de *GitHub*).

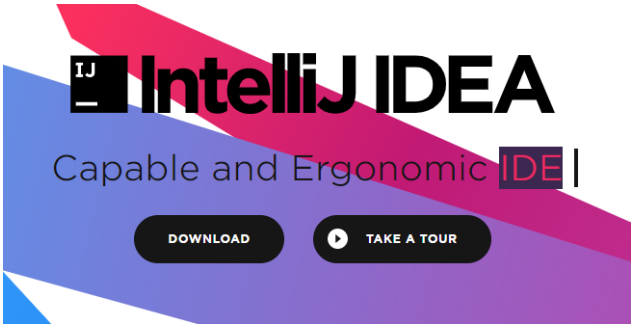


Figura 3.4: Página oficial de *IntelliJ IDEA* [Fuente]

3.1.2 Bibliotecas principales

3.1.2.1 TensorFlow

TensorFlow [15] es una biblioteca de código abierto destinada al aprendizaje automático desarrollada por Google Brain inicialmente para el uso interno de Google [16]. Sin embargo, en 2015 fue lanzado bajo una Licencia Apache 2.0.

Actualmente, la biblioteca no solamente funciona en Python, sino que también trata de dar soporte a otros lenguajes como JavaScript con TensorFlow.js o a varias plataformas móviles.

Como la implementación de programas usando directamente la biblioteca de TensorFlow puede resultar algo compleja, se puede hacer uso de APIs de alto nivel más intuitivas como Keras. En el sistema de verificación, por tanto, se estará haciendo uso de TensorFlow como el núcleo donde se sostiene la mencionada API.

Es importante mencionar que a la hora de instalarse esta biblioteca y Keras, se deben realizar los pasos pertinentes para hacer uso de la GPU y optimizar el tiempo de ejecución de las redes neuronales usadas.

3.1.2.2 Keras

Keras [17] (ver figura 3.5) es una biblioteca de código abierto destinada principalmente al desarrollo de redes neuronales escrita en Python cuyo autor principal es el ingeniero de Google [16] François Chollet. Puede utilizar como núcleo de funcionamiento a TensorFlow, entre otros.

Fue diseñada principalmente para permitir el desarrollo de redes neuronales profundas de una manera rápida, sencilla y amigable para el usuario, incluyendo una extensa y detallada documentación.

En el sistema de verificación, existen múltiples lugares donde se hace uso de esta biblioteca (y con ello de TensorFlow) como por ejemplo en la carga de generadores de descriptores en FaceNet (Inception ResNet v1) o en la construcción de redes neuronales *fully connected* para realizar la tarea de verificación (ver apartado 7.2).

Figura 3.5: Logotipo de Keras [17]

3.1.2.3 OpenCV

OpenCV [18] (ver figura 3.6) es una biblioteca libre basada en el lenguaje de programación C y C++ de visión por computador (*Computer Vision)* originalmente desarrollada por Intel en 1999. A lo largo de sus más de 20 años de trayectoria, se ha ido convirtiendo en una de las bibliotecas más populares de visión artificial gracias en gran parte a ser multiplataforma y tener una extensa documentación.

OpenCV es utilizado con diversos objetivos como detección de movimiento, reconstrucción 3D a partir de imágenes, reconocimiento de objetos...

En este proyecto, una biblioteca no oficial basada en OpenCV [19] será utilizada, entre otros, para el Generador de fotogramas desde vídeos.

Figura 3.6: Logotipo de OpenCV [18]

3.1.2.4 Dlib

Dlib [20] (ver figura 3.7) es una biblioteca *open-source* (o de código abierto) creada en 2002 multiplataforma de propósito general basada en el lenguaje de programación C++ que se encuentra bajo la licencia de software Boost. Incluye diversos componentes *software* independientes de temática relacionada principalmente con el aprendizaje automático.

En el caso de este Trabajo de Fin de Grado, se hará uso de su versión [21] para Python para la detección de caras haciendo uso de su modelo HOG y MMOD (ver sección 4.2). A su vez, también se utilizará su modelo para la detección de puntos faciales [22] (ver sección 6.2.2.4).

Figura 3.7: Logotipo de Dlib [20]

3.1.2.5 MTCNN

MTCNN [23] o *Multi-task Cascaded (Convolutional) Neural Networks* (Redes Neuronales (Convolucionales) Multitarea en Cascada) se define en su artículo como un *framework* (marco de trabajo) que trata de explotar la correlación que existe entre la tarea de la detección facial y el alineamiento de rostro para mejorar sus resultados. En la sección 4.1 se describirá más en profundidad dicho *framework*.

David Sandberg, creador de una implementación popular en TensorFlow de FaceNet (ver apartado 4.3), realizó una implementación [24] de MTCNN en dicho repositorio. Dicha implementación sirvió a su vez de referencia para la implementación de detección facial [25] de Iván de Paz Centeno usada en este proyecto. Su funcionamiento y desarrollo serán explicados también en el apartado 6.2.2.1.

3.2 Herramientas de gestión

3.2.1 Trello

Trello [26] es un software de administración de proyectos (cuyo desarrollador actual es la empresa *Atlassian* [27]) con cliente para sistemas operativos móviles y con interfaz web cuyo cometido es organizar proyectos. La razón de su uso es originada por la variada cantidad de tareas que se han llevado a cabo en este Trabajo de Fin de Grado, la cual provoca que (con el fin de obtener una cierta autonomía y continuidad en el ritmo de trabajo) se use algún sistema de tablero *Kanban*.

De forma resumida, un tablero *Kanban* [28] es una de las herramientas que pueden ser usadas para implementar una metodología *Kanban* en una administración de un proyecto personal o empresarial. La visualización de un tablero *Kanban* consiste en varias etapas de un proceso usando tarjetas (en este caso, virtuales) que representan tareas y columnas que representan dichas etapas del proceso (ver figura 3.8). Las columnas más comunes son las *to-do* (en espera), *doing* (realizando) y *done* (completado). En este caso, también se añadieron las columnas "consultas" y "consultas respondidas" (para controlar las dudas y sus respuestas procedentes de las tutorías semanales) y una columna "canceladas de momento" (para almacenar tareas canceladas que pudieran resurgir en un futuro como un nuevo foco de ideas para el proyecto).

Por último, cabe destacar que esta metodología se reforzó en la asignatura Prácticas Externas del Grado en Ingeniería Informática y en una de las charlas formativas sobre gestiones de proyectos pertenecientes a la asignatura que involucra a este proyecto.

Figura 3.8: Ejemplo de visualización de un tablero *Kanban* [Fuente]

3.2.2 GitHub

GitHub [29] es una plataforma cuyo objetivo es alojar proyectos en repositorios haciendo uso del sistema de control de versiones *Git*.

En el caso de este proyecto [30], se ha realizado un control de versiones básico[1].(ver figura 3.9) lo cual ha aportado seguridad a la integridad del proyecto y un seguimiento más sencillo de la evolución de este.

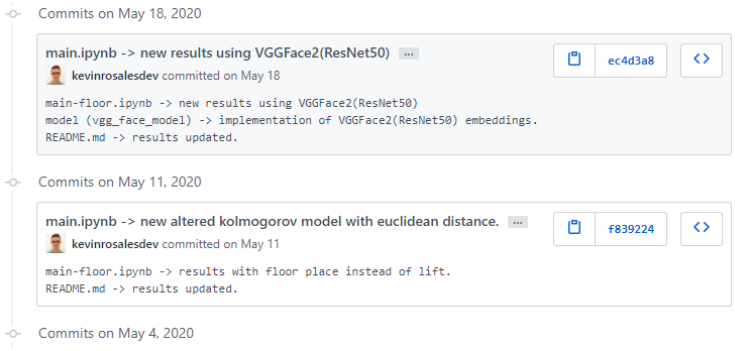


Figura 3.9: Muestra parcial del historial de versiones del repositorio original

3.2.3 Google Drive

Google Drive [31] es un servicio de alojamiento de archivos creado por *Google* [16] en 2012. *Google Drive* ofrece 15 GB de almacenamiento gratuito para el alojamiento de archivos de usuarios sin planes de pago y acceso a través del navegador y de aplicaciones móviles.

[1]El control de versiones tuvo que ser eliminado cuando se publicó el repositorio para proteger los derechos de las identidades que se encontraban en versiones antiguas de los cuadernos Jupyter y no firmaron el acuerdo de consentimiento de divulgación de sus imágenes (ver apartado de Normativa y Legislación)

En este caso, *Google Drive* fue el encargado de almacenar las salidas de algunas de las distintas fases del proyecto con el fin de obtener una cierta persistencia y seguridad al guardar los datos de forma redundante en la nube (ver figura 3.10).

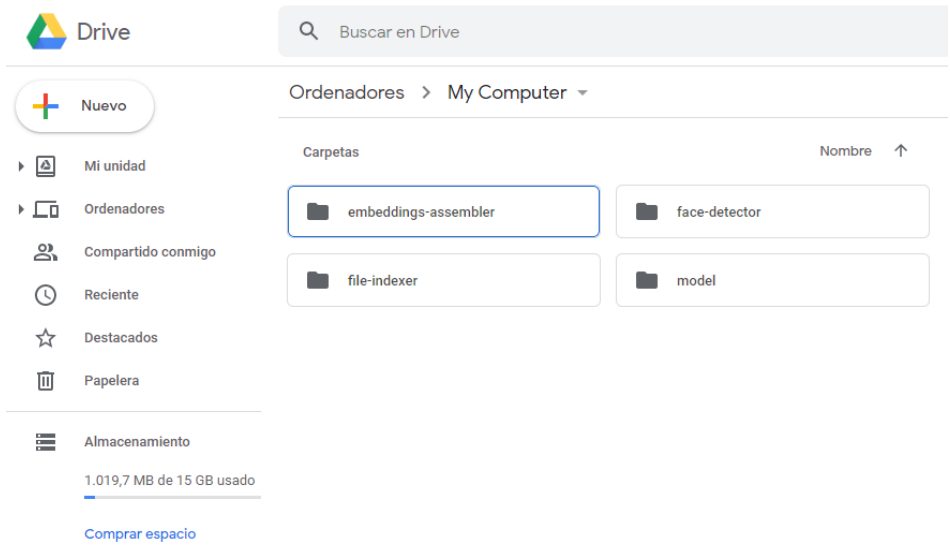


Figura 3.10: Almacenamiento de ficheros del TFG en la nube mediante Google Drive

3.2.4 Google Hangouts

Google Hangouts [32] es una aplicación de mensajería multiplataforma creada por *Google* [16] en 2013.

A lo largo del proyecto, se utilizó *Google Hangouts* como medio para realizar reuniones semanales hasta principios del mes de mayo a través de videollamadas debido a que las reuniones presenciales fueron canceladas por la crisis sanitaria provocada por el COVID-19.

3.2.5 Microsoft Teams

Microsoft Teams [33] es una plataforma de comunicación que combina, entre otros, chat, reuniones de vídeo y almacenamiento de archivos creada por *Microsoft* [34] en 2017.

A partir de principios del mes de mayo, se decidió sustituir *Google Hangouts* por *Microsoft Teams* por cuestiones relacionadas a la calidad de las videollamadas en las reuniones.

3.2.6 Overleaf

Overleaf [35] (ver figura 3.11) es un editor LaTeX [36] en línea creado por la compañía del mismo nombre *Overleaf* [37] en 2014. Se ha hecho uso de su plan gratuito para redactar la presente memoria del Trabajo de Fin de Grado.

Figura 3.11: Logotipo oficial de *Overleaf* [Fuente]

3.2.7 WeTransfer

WeTransfer [38] es un servicio de transferencia de ficheros accesible a través de navegadores web creado por *WeTransfer* [39] en 2009 y que permite enviar, de manera gratuita, ficheros de hasta 2 GB.

En el caso del proyecto, se ha utilizado para el traspaso de varios ficheros de salida de las distintas fases del trabajo en caso de que estos guardaran datos de peso demasiado elevado que impidieran enviarlos por correo ordinario universitario a los tutores del proyecto.

3.2.8 Campus virtual de la ULPGC y página oficial de la EII

Ambos elementos han sido utilizados como herramientas para el seguimiento de toda la información [40] y de las actividades planteadas a lo largo de la asignatura que envuelve el actual trabajo.

3.3 Equipo de trabajo

El análisis, el diseño, el desarrollo, la evaluación y la ejecución del proyecto se ha realizado en un equipo con las siguientes características:

Sistema Operativo	Microsoft Windows 10 Pro (64-bits)
GPU	NVIDIA GeForce GTX 960
CPU	Intel(R) Core(TM) i5-4670K CPU @ 3.40GHz (4 CPUs)
Memoria RAM	8 GB
Placa base	Asus H97M-E

Tabla 3.1: Equipo de trabajo

Capítulo 4

Estado del arte

La verificación, la identificación y la reidentificación [41] de identidades se han convertido a lo largo de la historia de la visión por computador en varios de los problemas con mayor interés por parte de los investigadores debido a sus grandes utilidades, especialmente cuando se trata de realizar dichas tareas en tiempo real bajo escenarios no ideales donde en caso de usar únicamente las caras (estos problemas pueden resolverse haciendo uso de imágenes de cuerpo entero), estas pueden presentar una baja calidad de iluminación, de definición, mostrarse parcialmente o estar rotadas.

Si bien los tres problemas se pueden categorizar dentro de las tareas de reconocimiento (en el caso de este proyecto, de tipo facial) se debe mencionar brevemente la diferencia que existe entre ellos:

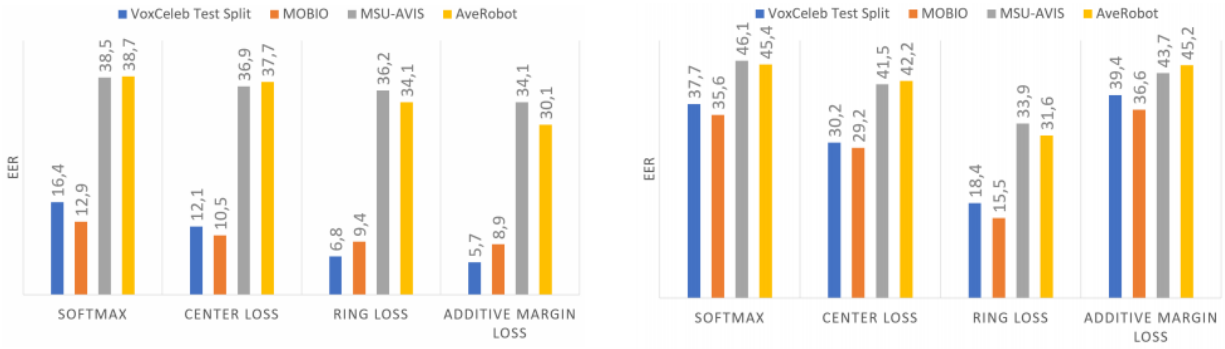
- La verificación (como se mencionó en el Capítulo 1) trata de, dadas dos imágenes que contienen una persona, determinar si son la misma.
- La identificación es el proceso de búsqueda de la identidad de una persona (a través de una imagen suya) en una base de datos con identidades de personas previamente introducidas.
- La reidentificación tiene como objetivo determinar si una persona (mediante su imagen) ya ha sido anteriormente visualizada por el sistema y, en caso afirmativo, identificarla.

El estudio realizado por el Departamento de Matemáticas y Ciencias de la Computación de la Universidad de Cagliari en Italia y el Instituto Universitario de Sistemas Inteligentes y Aplicaciones Numéricas en Ingeniería (SIANI) de la Universidad de Las Palmas de Gran Canaria (ULPGC) sobre *AveRobot* [2] muestra resultados donde se puede observar la dificultad que posee una tarea de verificación cuando se realiza bajo el escenario no ideal mencionado previamente.

Concretamente, en dicho artículo se propone (como es habitual en los procesos de interacción hombre-robot) realizar la tarea de verificación apoyándose tanto en el rostro físico del individuo como en su voz o habla, aportando modalidades biométricas separadas con el objetivo de conseguir obtener una buena calidad en el sistema.

Sin embargo, aun haciendo uso de ambas modalidades, los resultados que se obtienen con respecto al EER o *Equal Error Rate* (ver sección 7.1.2.2) muestran la dificultad de realizar dicha verificación usando rostros cuando se desarrolla bajo un conjunto de datos no controlado o semicontrolado como *AveRobot* o *MSU-AVIS* [42], generando un EER de 30.1 % (o 0.301 sobre 1) en el mejor de los casos con el conjunto de datos que se tratará en el presente Trabajo de Fin de Grado (ver figura 4.1a).

Además, se debe tener en cuenta que el ruido presente en los audios de las grabaciones de *AveRobot* generado por situaciones como puertas abriéndose, conversaciones ambientales o los propios dispositivos que actúan de robots asistentes provoca que dicho conjunto de datos se convierta también en uno de los más desafiantes con respecto a la verificación mediante voz, logrando como mejor EER un 31.6 % (o 0.316 sobre 1) (ver figura 4.1b).



(a) Resultados de verificación mediante rostros (b) Resultados de verificación mediante voz

Figura 4.1: EER de la verificación usando ResNet-50 (ver apartado 4.4) entrenado con varias funciones de pérdida sobre el conjunto de entrenamiento de VoxCeleb y probado con diferentes conjuntos de datos (se muestra *AveRobot* en amarillo) [2]

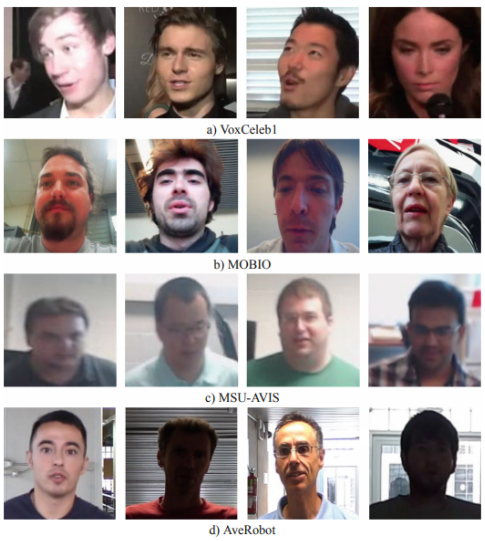


Figura 4.2: Muestras de rostros de los conjuntos de prueba usados en la figura 4.1 [43]

Dicho artículo concluye remarcando que los resultados demuestran que *AveRobot* es uno de los conjuntos de datos que supone un mayor reto por la cantidad de condiciones sin control que contiene y advirtiendo de la necesidad de un entendimiento mejor de la reacción de los algoritmos existentes actualmente destinados a la verificación frente a escenarios capturados de manera no ideal (ver comparación de conjuntos de prueba de la figura 4.2).

Continuando con los estudios sobre la verificación también se puede analizar un artículo de los mismos autores [43] posterior donde se plantea usar ambas modalidades biométricas fusionadas para realizar la verificación. En este caso, se recogen características del rostro y de la voz en paralelo y se explora si uniéndolas se es capaz de mejorar la tasa de acierto del sistema de verificación.

Para llegar a tal fin, se diseña la arquitectura neuronal representada en la figura 4.3 que fusiona los datos de ambos tipos de modalidades biométricas para a continuación estudiar su comportamiento sobre varias funciones de pérdida y conjuntos de datos:

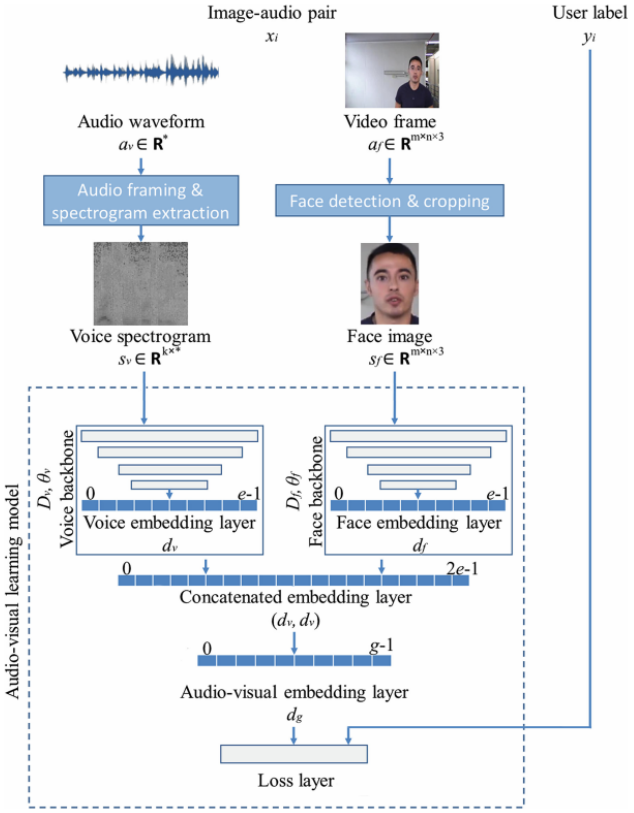


Figura 4.3: Arquitectura neuronal propuesta en el artículo para una fusión multibiométrica [43]

Respecto a los resultados sobre *AveRobot*, tras probar varias combinaciones que hacen uso de ambas modalidades biométricas de un modo u otro, se tuvo como EER más bajo un 31.2 % (o 0.312 sobre 1), como se muestra en la figura 4.4. No obstante, en dicha figura se puede visualizar cómo usar ambos tipos de modalidades biométricas al mismo tiempo puede dar mejores resultados que usar las modalidades de manera individual en el sistema planteado.

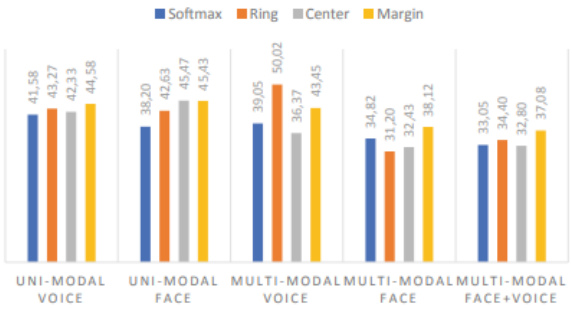
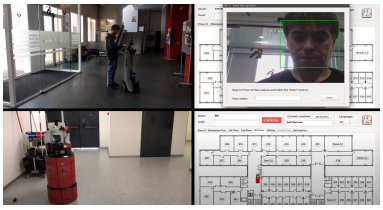
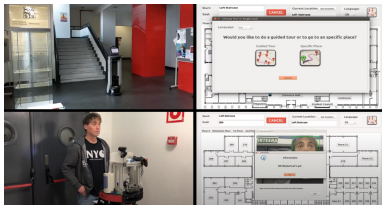


Figura 4.4: EER de la verificación en *AveRobot* haciendo uso de varias combinaciones de modalidades respecto a la voz y al rostro y de diversas funciones de pérdida [43]

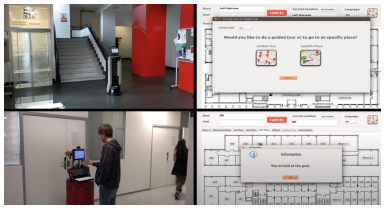
Estudiando a los escenarios multinivel, se debe mencionar al estudio [44] realizado por la Facultad de Informática de la Universidad del País Vasco junto al ya mencionado SIANI de la ULPGC donde se propone una aplicación (GidaBot) diseñada para administrar un equipo de robots que actúen de guías turísticos en edificios de múltiples plantas. Como los recorridos de una persona física pueden afectar a más de un piso, se propone una comunicación entre los robots que permita verificar a una persona ya asistida en un piso distinto anteriormente por otro robot para continuar orientándola por el edificio, como se muestra en la figura 4.5 mediante la arquitectura representada en la figura 4.6.



(a) Llegada de la persona al sistema con el primer robot en un piso X



(b) Verificación de la persona con el segundo robot en un piso Y



(c) Asistencia del segundo robot como guía turístico en un piso Y

Figura 4.5: Recorrido de una persona física siendo guiada por varios robots en un escenario multinivel [45]

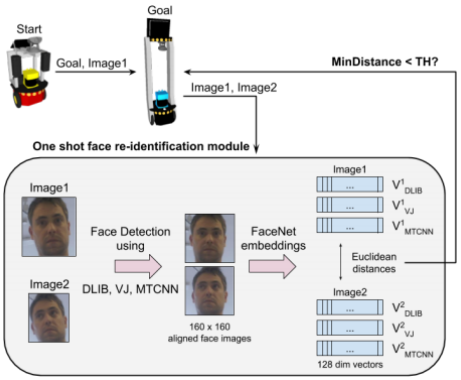


Figura 4.6: Arquitectura básica del sistema de verificación usada en el estudio [44]

Un ejemplo de biblioteca que incluye procesos de reconocimiento facial con verificación puede ser DeepFace [46] (ver figura 4.7) de Sefik Ilkin Serengil, quien implementa la verificación basada en un umbral. Dicho tipo de verificación será uno de los utilizados a lo largo del proyecto (ver sección 7.1.1).

Según su autor, un proceso de verificación mediante rostros tiene como idea fundamental el reconocimiento facial y se basa en cuatro fases principalmente: detección (sección 6.2), alineamiento (sección 6.2.3), representación vectorial o generación de descriptores (sección 6.3) y verificación (secciones 7.1 y 7.2) haciendo uso del concepto de distancias entre descriptores (sección 6.5).

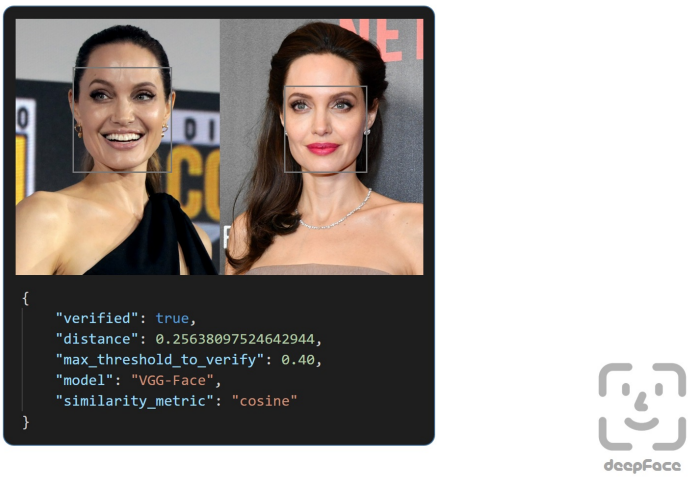


Figura 4.7: Ejemplo de verificación con DeepFace [46]

Respecto a las tecnologías relacionadas con detección facial, también se hará uso de detectores tradicionales (como puede ser un detector basado en Histograma de Gradientes Orientados [47]) y de detectores recientes (como MTCNN [23]).

Además, en DeepFace [46], se usaron cinco tipos distintos de generadores de descriptores. Entre ellos, se destaca el uso de VGG-Face [48] y Google FaceNet [49], los cuales serán investigados y evaluados a lo largo del proyecto.

A continuación se detalla cómo funciona la infraestructura de los detectores y generadores de descriptores que se usarán:

4.1 Detector MTCNN

El primero de los tres detectores que se utilizarán será MTCNN. A continuación, se explicará cómo funciona dicho detector [50]:

1. El primer paso que realiza MTCNN es recoger como entrada una imagen. En el caso del presente proyecto, dicha imagen se corresponderá a cada uno de los fotogramas de

un determinado vídeo, mientras que en un sistema real sería la imagen que estuviera captando en ese momento el robot asistente. Dicha imagen es redimensionada a varios tamaños con el objetivo de localizar diferentes tamaños de caras en el fotograma (pirámide de imágenes).

2. A continuación, se utilizan núcleos (o *kernels*) de tamaño 12x12 que se van desplazando a lo largo de la imagen. Dichas porciones de la imagen son pasadas a la red *P-Net* (ver figura 4.8), la cual es la encargada de devolver las coordenadas de un cuadro delimitador (más conocido por el término en inglés *Bounding box*). Dicho núcleo va moviéndose 2 píxeles tras cada escaneo, lo que reduce la complejidad de la computación sin sacrificar de manera muy significativa la precisión (ya que las caras de las personas en general serán de más de 2 píxeles). Gracias al redimensionamiento realizado en el primer paso, el detector es capaz de detectar caras de menor tamaño (por el tamaño del núcleo en una imagen de gran tamaño) y a su vez, posee la habilidad de detectar caras de mayor tamaño (por el tamaño del núcleo en una imagen de menor tamaño).

    - Para aquellas caras devueltas que estén superpuestas, se utiliza el NMS o *Non-Maximum Suppression*. Este algoritmo recoge aquellos núcleos que se superponen una cantidad elevada de veces y poseen un gran nivel de confianza (ver sección 6.2.1.1), ya que probablemente se traten de la misma persona. Este proceso es imprescindible dado a que MTCNN tiene la capacidad de detectar varios rostros en una misma imagen (aunque en el caso de *AveRobot*, se propone un individuo únicamente por vídeo).

3. Como P-Net ha sido entrenado previamente, devuelve cuadros delimitadores relativamente precisos para las caras detectadas. En este paso es necesario recoger las coordenadas devueltas (las cuales probablemente hayan usado caras de imágenes redimensionadas) para escalar dichas coordenadas a la imagen original.

    - Podría existir el caso de que un cuadro delimitador contuviera parte de la imagen fuera de los márgenes originales. En este caso, MTCNN realiza un relleno con valores a 0 (píxeles negros). En el caso de *AveRobot*, aunque la mayoría de los individuos tienen su cabeza dentro de los márgenes de los dispositivos que los detectan, existen una serie de vídeos donde su rostro sale de los márgenes del dispositivo. Estos casos, si bien son aislados, supondrán un problema a la hora de verificar dada la calidad de la detección y deberán ser evitados usando las Medidas de calidad de detecciones.

4. En el siguiente paso, se normalizan los valores de los píxeles de dichos cuadros delimitadores redimensionados a 24x24 (que actualmente se encuentran entre 0 y 255) para transformarlos a un intervalo de [-1, 1]. Para ello, se aplica la siguiente fórmula:

$$newPixel_{value} = \frac{pixel_{value} - 127,5}{127,5}$$

5. Tras normalizar los valores de los píxeles, se introducen en la siguiente red, la R-Net (ver figura 4.8). Dicha R-Net tiene como objetivo presentar unos cuadros delimitadores más precisos junto a su confianza. Tras esto, se vuelven a eliminar aquellos cuadros

delimitadores con confianza más baja y se procede a volver a realizar el algoritmo NMS. Evidentemente, se vuelven a escalar a sus coordenadas originales.

6. La última red que compone el *framework* de MTCNN es la red O-Net (ver figura 4.8). Dicha red (cuyos cuadros delimitadores deben ser redimensionados a 48x48) devuelve 3 salidas: las coordenadas del cuadro delimitador, las coordenadas de 5 puntos faciales o *landmarks* (1 en el ojo izquierdo, 1 en el ojo derecho, 1 en nariz, 1 en la parte izquierda de la boca y otro en la parte derecha de la boca) y la confianza. Tras esta salida, se vuelve a realizar el mismo procedimiento que se ha desarrollado previamente: eliminar cuadros delimitadores con confianza baja, escalar las coordenadas (en esta ocasión también de los puntos faciales) y ejecutar el algoritmo NMS.

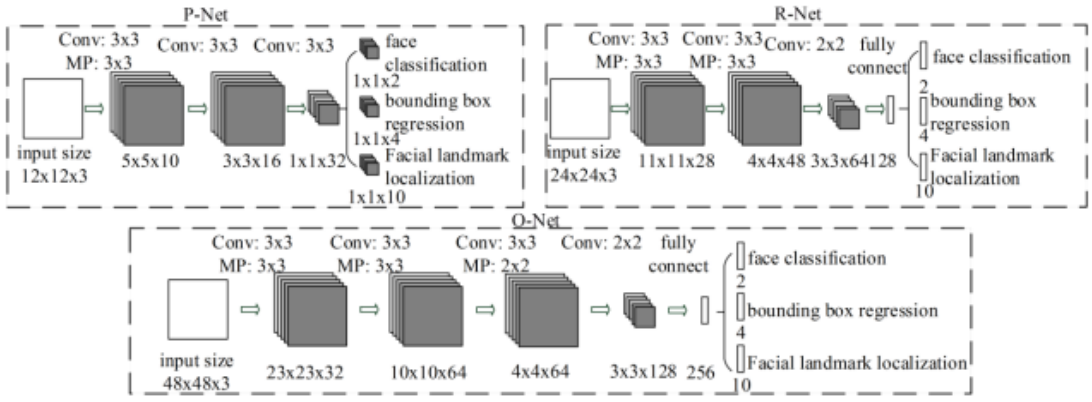


Figura 4.8: Arquitectura de P-Net, R-Net y O-Net [23]

La implementación de este detector en el proyecto será explicada en el apartado 6.2.2.1.

4.2 Detectores de DLIB

En el caso de Dlib, se han utilizado dos detectores distintos: el Detector DLIB - HOG y el Detector DLIB - MMOD.

4.2.1 Detector DLIB - HOG

El segundo detector usado y el primero de los detectores de Dlib que se propone es el detector HOG. Dicho detector [51] hace uso del clásico HOG [47] o *Histogram of Oriented Gradients* (Histograma de Gradientes Orientados) combinado con un clasificador lineal, una pirámide de imágenes (redimensionamiento mencionado en el paso 1 del algoritmo del *framework* del Detector MTCNN) y un esquema de detección mediante ventana deslizante.

En general, HOG es capaz de detectar muchos tipos de objetos semirígidos, por lo que se considera un detector de propósito general. En este caso, se utilizará evidentemente para la detección de caras.

A continuación, se describirá cómo funciona este tipo de detector [52]:

1. En primer lugar, se recaban muestras con el objeto que se quiere detectar y se extraen descriptores HOG de dichas muestras (ver figura 4.9). En este caso, el modelo se ha entrenado con rostros.

    - Unos descriptores HOG se corresponden a la salida de un Histograma de Gradientes Orientados. Este descriptor de características mide la orientación de los gradientes en diferentes porciones de la imagen (se hace uso de una ventana deslizante para generar los diversos histogramas que componen dicha imagen).

Figura 4.9: Ejemplo de un HOG [Fuente]

2. A continuación, se realiza la misma acción con las muestras que no contienen el objeto que se quiere detectar y se vuelven a extraer los descriptores HOG.
3. Se entrena un modelo (por ejemplo, una SVM, la cual se encuentra explicada en el paso 4 de la sección 6.2.1.2) con las muestras positivas y negativas.

    - Tras el entrenamiento se puede utilizar una ventana deslizante que permita localizar rostros en determinadas porciones de la imagen.
    - Se puede realizar un estudio de los falsos positivos que muestren que un determinado histograma de gradientes de una porción de la imagen corresponde a un rostro inexistente. Dichas muestras se pueden volver usar de nuevo para re-entrenar el modelo, de modo que se trate de evitar dichos falsos positivos. A esta técnica se le llama *hard-negative mining*.

4. Una vez el modelo ha sido entrenado haciendo uso de la pirámide de imágenes redimensionadas, se encuentra listo para detectar el objeto con el que fue entrenado (en el caso de este proyecto, se detectan caras de personas).

    - En este paso se puede aplicar el NMS o *Non-Maximum Suppresion*, de manera que se eliminen los cuadros delimitadores redundantes y solapados.

En este caso, su implementación en el proyecto será explicada en el apartado 6.2.2.2.

4.2.2 Detector DLIB - MMOD

El segundo de los detectores de Dlib y el último detector utilizado que se propone es el detector MMOD. Dicho detector [53] hace uso de una red neuronal convolucional (*Convolutional Neural Network* o CNN). Para ello, carga un modelo pre-entrenado que tiene como objetivo detectar caras en imágenes. Según la descripción de dicho detector, es más preciso que el modelo basado en HOG (ver sección 4.2.1), pero necesita mayor potencia computacional para ejecutarse. Por este motivo, se recomienda encarecidamente ejecutarlo haciendo uso de la GPU (aprovechando las capacidades que puede llegar a ofrecer en el cómputo de las redes neuronales).

El modelo de detección de caras, disponible en uno de los repositorios del creador de Dlib [22] (`mmod_human_face_detector.dat.bz2`), se basa en la detección de objetos de margen máximo [54] (*Max-Margin Object Detection* o MMOD). El algoritmo que sigue el detector [55] se detalla a continuación de manera breve, pues al ser una red convolucional posee bastantes características en común con el Detector MTCNN:

1. Se crea una pirámide de imágenes como se realizó con MTCNN y se recogen todas ellas en una única de dimensión mayor.
2. Se aplica la CNN a dicha imagen combinada mediante una ventana deslizante. Como la imagen combinada contiene la pirámide de imágenes, la CNN es capaz de detectar las caras siendo invariante a escala.
    - Como la imagen posee la pirámide de forma combinada, no es necesario ir por cada una de las imágenes de la pirámide por separado ya que la imagen formada ya las contiene a todas ellas, con lo que se optimiza el proceso.
3. Tras obtener todas las zonas probables de localización, se realiza un NMS y se devuelven los cuadros delimitadores.
    - En este paso es necesario mencionar que se encuentra la detección de objetos de margen máximo (MMOD), la cual funciona como un optimizador que recorre todas las ventanas de la imagen y mejora la tasa de acierto del sistema de detección respecto al número de detecciones fallidas y falsos positivos, intentando colaborar con el correcto funcionamiento del NMS otorgando un margen a las detecciones que provoca que la red neuronal aprenda a evitar zonas donde no se mostrarán rostros.

En este último caso de detector, su implementación en el proyecto será descrita en la sección 6.2.2.3.

4.3 FaceNet (Inception ResNet v1)

El primer generador de descriptores que se utilizará será FaceNet [49], el cual es un sistema desarrollado en 2015 por Google [16] que aprende la transformación de una imagen de

una cara a un espacio euclídeo compacto donde las distancias son las encargadas de medir la similitud entre dichas caras. Con dicho mapeo, tareas como el reconocimiento facial, la verificación o el agrupamiento (o *clustering*) pueden ser implementadas con facilidad usando dichos *embeddings* como vectores de características o descriptores.

En concreto, FaceNet usa una red neuronal convolucional profunda entrenada para optimizar el descriptor (en lugar de una capa intermedia para extraer el descriptor como se ha realizado en otros modelos). Para el entrenamiento, FaceNet utiliza las denominadas tripletas, las cuales están compuestas de tres imágenes: una original, otra con la misma persona y otra con distinta persona.

FaceNet posee varias estructuras para su modelo. En este proyecto, se hará uso de la implementación de David Sandberg [56] para TensorFlow de la arquitectura Inception ResNet v1.

Concretamente, el modelo que se propone en el artículo de FaceNet [49] es el que se puede visualizar en la figura 4.10:

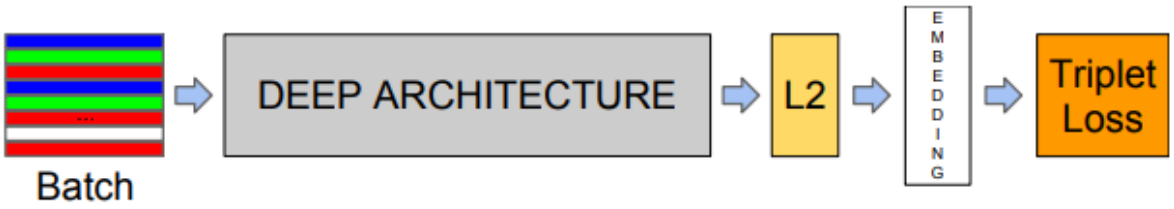


Figura 4.10: Estructura del modelo de FaceNet [49]

Donde, como se puede observar:

- La entrada pasa por la arquitectura que implementa FaceNet y que tiene como finalidad generar una salida con descriptores. En el caso de este proyecto y como fue mencionado previamente, la arquitectura usada será Inception ResNet v1 (ver figura 4.11).
    - En el sistema se hará uso de una implementación en Keras [57] basada en el trabajo realizado por David Sandberg [56].
    - Esta versión de Inception ResNet v1 es levemente modificada respecto al modelo original mostrado en la figura 4.11, debido a que la entrada pasa a ser una imagen de dimensiones de 160x160 y la salida con *Softmax* desaparece como es común en los generadores de descriptores para dar lugar a una salida sin función de activación. Se puede observar esta implementación en el repositorio de FaceNet para Tensorflow [58]. La salida será un vector de 128 dimensiones.
- Tras obtener la salida de la arquitectura de la red convolucional profunda, se realiza una normalización L2 para obtener el descriptor final.
    - El modelo se ha entrenado usando el conjunto de datos *MS-Celeb-1M* [59], como se ha realizado también con el generador de descriptores basado en VGGFace2 - ResNet50.

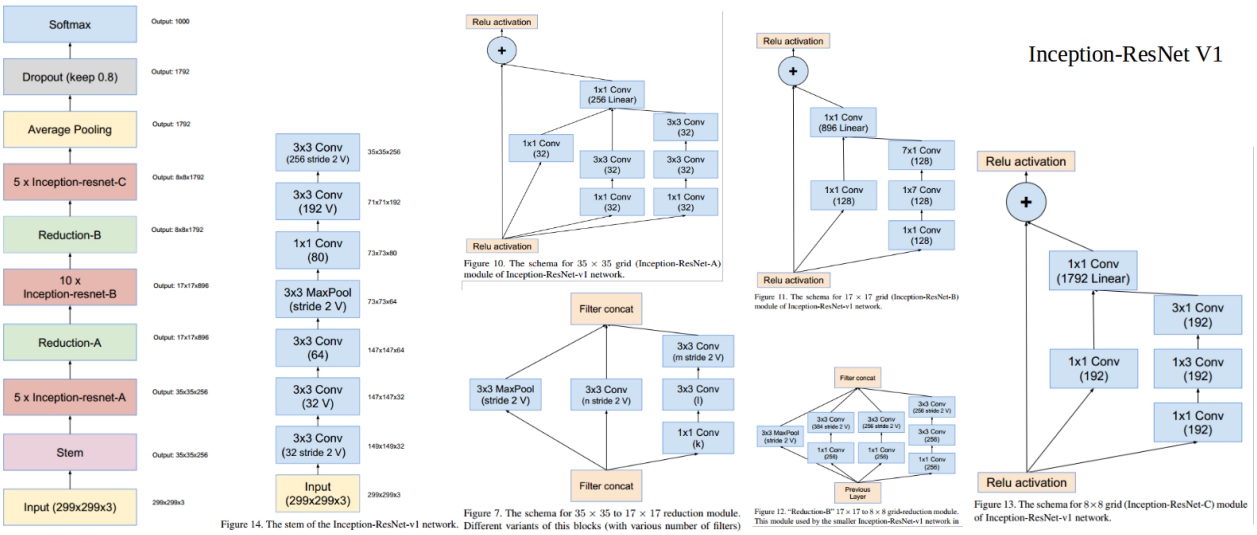


Figura 4.11: Estructura del modelo de Inception ResNet v1 [Fuente]

- En la fase de entrenamiento, los rostros son la entrada de la red a modo de tripleta. Como se mencionó previamente, se pretende optimizar una función de error donde el descriptor de la imagen con la misma persona sea cercano a la imagen original y el descriptor con la persona distinta se diste lo máximo posible, como se puede analizar en las figuras 4.12 y 4.13:

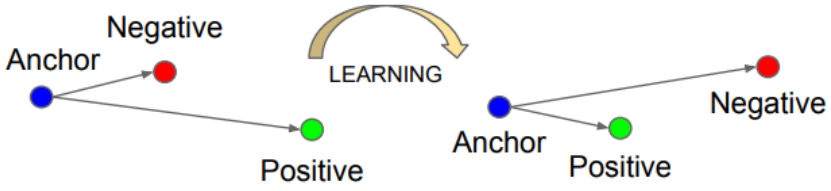


Figura 4.12: Representación de la función de pérdida basada en una tripleta [49]

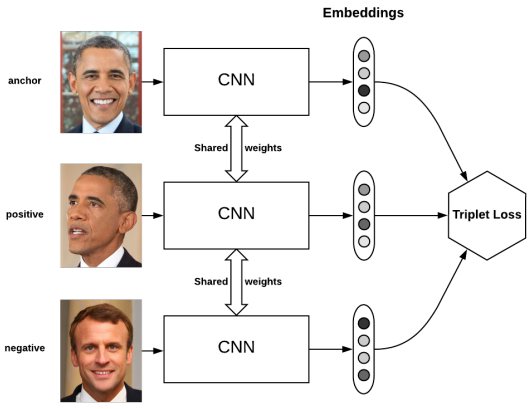


Figura 4.13: Ejemplo de la función de pérdida basada en una tripleta [Fuente]

La implementación de FaceNet en el proyecto será descrita en la sección 6.3.1.

4.4 VGGFace2 - ResNet50

El segundo y último generador de descriptores que se usará será ResNet50 apoyado sobre VGGFace2 [48], el cual se trata de un conjunto de datos (o *dataset*) de un tamaño voluminoso destinado al reconocimiento facial desarrollado por *Visual Geometry Group* [60] de la Universidad de Oxford [61]. Las imágenes que contiene han sido descargadas desde búsquedas de imágenes en Google [16] y contiene una gran extensión de variaciones en lo que refiere a pose, emociones, edad, iluminación, origen étnico y profesión (ver figura 4.14). En concreto, se trata de un conjunto de datos que posee:

- Más de 9.000 identidades variadas.
- Más de 3.300.000 imágenes faciales capturadas *in the wild*. Es decir, se trata de un conjunto de datos con muestras que no han sido diseñadas ni construidas originalmente con una finalidad relacionada con la investigación.
- De media 362 imágenes por cada identidad, siendo un número que varía entre 87 y 843.
- Respecto a la distribución por sexo, la división del conjunto de datos en entrenamiento/test y la distribución de tamaño del rostro se tienen las siguientes gráficas circulares:

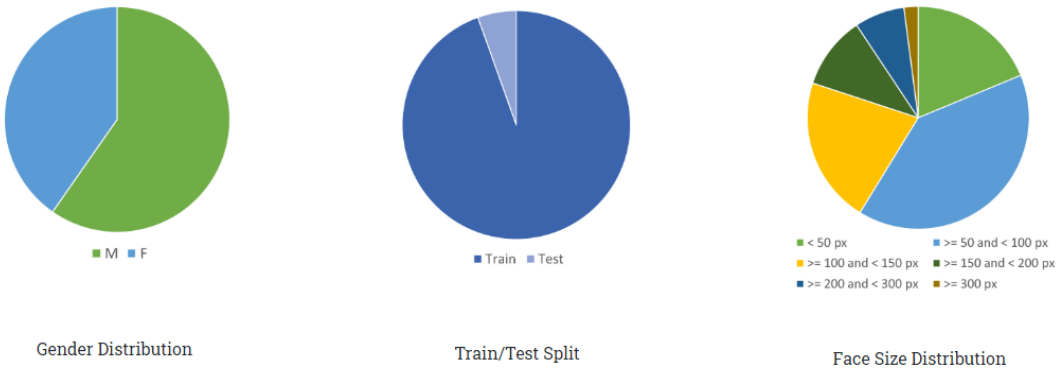


Figura 4.14: Distribución por sexo, división del conjunto de datos de entrenamiento y test y distribución del tamaño del rostro en VGGFace2 [Fuente]

Respecto a los modelos que utilizan dicho conjunto de datos como entrenamiento realizando *fine-tune*[1] teniendo como base de entrenamiento *MS-Celeb-1M* [59] existe una gran variedad de ellos:

[1] *Fine Tuning*: técnica que recoge los pesos de una red neuronal previamente entrenada y la utiliza como inicialización para un nuevo modelo entrenado con nuevos datos del mismo dominio. En este caso, imágenes de rostros. Para realizar esta tarea se pueden seguir varias estrategias, como volver a entrenar toda la red neuronal o detener el entrenamiento para algunos de los pesos o incluso capas. Con ello, se logra entrenar una red a raíz de otra que ya era capaz de aprender una serie de características para un problema similar.

Architecture	Feat dim	Pretrain	TAR@FAR = 0.001	TAR@FAR = 0.01
ResNet-50	2048	N	0.878	0.938
ResNet-50	2048	Y	0.891	0.947
SE-ResNet-50	2048	N	0.888	0.949
SE-ResNet-50	2048	Y	0.908	0.956
ResNet-50-256D	256	Y	0.898	0.956
ResNet-50-128D	128	Y	0.904	0.956
SE-ResNet-50-256D	256	Y	0.912	0.965
SE-ResNet-50-128D	128	Y	0.910	0.959

Tabla 4.1: Modelos entrenados sobre VGGFace2 [Fuente]

Se debe resaltar que para la detección facial se utiliza MTCNN (ver apartado 4.1) con una redimensión de un factor de 0.3 (excepto que la extensión provocara una salida de los bordes de la detección). Todos los modelos entrenados han usado la función de pérdida de *cross-entropy*.

En este proyecto, y tal como se indica en el título, se utilizará la red neuronal ResNet-50 [62] pre-entrenada, la cual ofrece un descriptor de 2048 dimensiones teniendo como entrada un tensor de píxeles de dimensión 224x224x3 y una tasa de aceptación real (TAR) de 0.891 y de 0.947 con un FAR (ver apartado 7.1.2.2) de 0.001 y de 0.01 respectivamente.

Para usar ResNet-50 entrenado sobre VGGFace2 se ha utilizado la biblioteca `keras-vggface` [63]. Los modelos de dicha biblioteca parten de las redes originales de Caffe[2] sobre Tensorflow (ver apartado 3.1.2.1). Concretamente, se requieren las siguientes versiones (también son válidas para hacer funcionar al generador de descriptores basado en FaceNet (Inception ResNet v1)):

- Keras v2.2.4
- Tensorflow v1.14.0

Respecto a la arquitectura de ResNet-50, se tiene la representación mostrada en la figura 4.15:

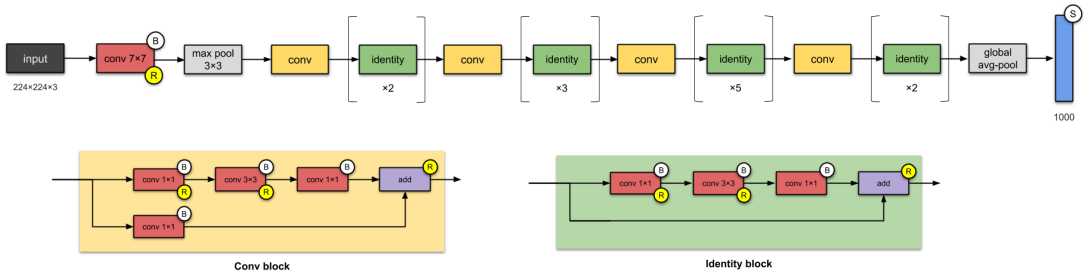


Figura 4.15: Arquitectura de ResNet-50 [Fuente]

[2]Caffe: *framework* para aprendizaje profundo creado por Yangqing Lia y desarrollado por *Berkeley AI Research* (BAIR) y su comunidad.

Donde se debe tener en cuenta la leyenda descrita en la figura 4.16:

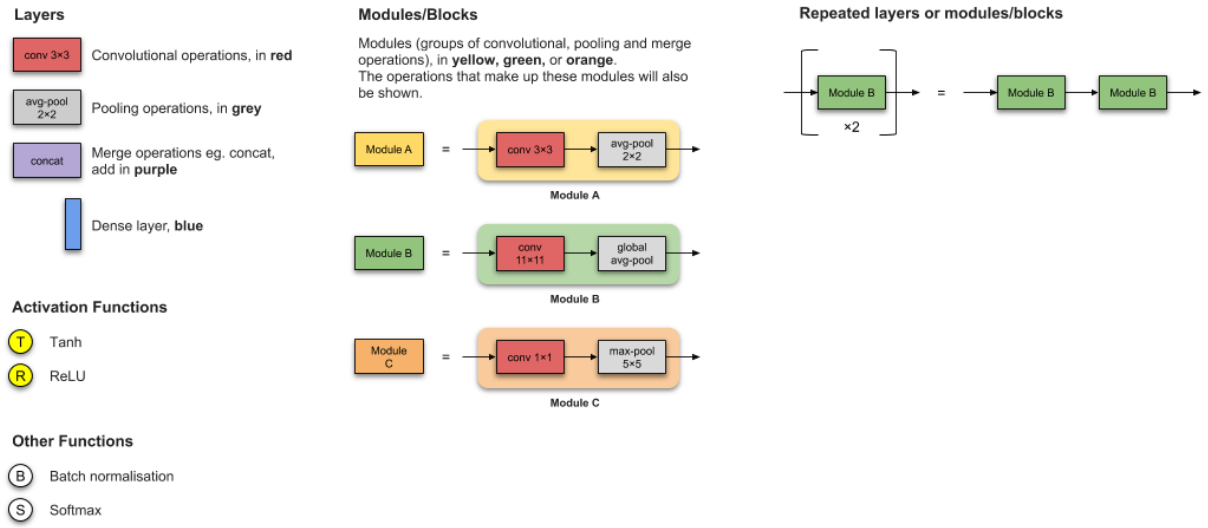


Figura 4.16: Leyenda de arquitectura de ResNet-50 [Fuente]

Donde la última capa *fully connected* ha sido modificada para que contuviera la misma cantidad de identidades con las que se entrena con VGGFace2. En este caso, al no incluirse dichas capas, al utilizar el modelo se tendrá como salida el resultado vectorizado con 2048 dimensiones deseado.

Además, en la implementación de `keras-vggface` de dicha arquitectura[3] se podrá observar cómo concuerda la estructura de la red neuronal de la figura 4.15 con la desarrollada por la biblioteca.

La implementación de este generador de descriptores en el proyecto será mostrada en la sección 6.3.2.

Una vez se ha explicado el entorno global en el que se situará el presente proyecto, el cual parte desde la calidad de los conjuntos de datos para verificación y en concreto la de *AveRobot* con distintas formas de encararlo, pasando de manera somera por el concepto de verificación en escenario multinivel y finalizando por una breve introducción al proceso de verificación estándar incluyendo el marco teórico que envuelve a los detectores y generadores de descriptores que se usarán, se tiene una idea más general de la localización de este proyecto dentro de la comunidad de investigación computacional.

[3]Enlace a la implementación: https://github.com/rcmalli/keras-vggface/blob/master/keras_vggface/models.py#L207

Capítulo 5

Descripción del conjunto de datos

Este capítulo tiene como objetivo describir en qué consiste exactamente *AveRobot* [2], el cual, como se mencionó en la Introducción, servirá como conjunto de datos para el estudio de un escenario multinivel. Además, se comentará brevemente qué subconjuntos de *AveRobot* fueron escogidos para el análisis y evaluación de resultados en el presente proyecto. Por último, se detallará la selección de conjuntos de test y entrenamiento para cada uno de los subconjuntos elegidos, lo que generará definitivamente la entrada a los sistemas que se construirán a lo largo del Trabajo de Fin de Grado.

5.1 La entrada: el conjunto de datos *AveRobot*

Como se mencionó previamente en el Capítulo 1, la verificación se estudiará utilizando el escenario multinivel *AveRobot* [2]. La primera parte del presente capítulo se destinará íntegramente a la descripción de la distribución del conjunto de datos y a la explicación de los subconjuntos escogidos, analizando la calidad de vídeos que se pueden encontrar en las muestras seleccionadas para el presente proyecto.

5.1.1 Estadísticas de *AveRobot*

AveRobot, el conjunto de datos audiovisual destinado a la reidentificación y verificación de personas en el contexto de interacción hombre-máquina, contiene una serie de vídeos cortos (de una duración aproximada de 10 segundos) en los que los participantes (personas físicas) de manera individual (en principio) comentan frases de corta duración simulando estar en escenarios con presencia de robots asistentes.

Respecto a sus características concretas, se tiene que *AveRobot* posee:

- 111 individuos de diferentes nacionalidades (por ejemplo, de España, China o República de la India), edad (teniendo una media de 27 años con una desviación estándar de 11) y altura (con una media de 1,74 metros de altura y una desviación estándar de 10 centímetros).
- 2.664 vídeos filmados desde tres localizaciones distintas para cada uno de los tres pisos (dado a que el escenario es multinivel) de un edificio universitario común (en este caso, el módulo III del Edificio Departamental de Informática y Matemáticas):
  - Escaleras.
  - Ascensor.
  - Pasillo.
- 8 dispositivos de captación de vídeo por usuario. La tabla 5.1 describe la relación entre los pisos y los dispositivos utilizados en cada uno de estos.
- 9 frases por usuario. Cada uno de los 111 individuos captados aparece 24 veces a lo largo del conjunto de datos ($V\acute{i}deos/usuario = \sum_{piso=1}^{3} N_{escenarios} * N_{c\acute{a}maras_{piso}} = 3 * 2 + 3 * 3 + 3 * 3 = 24$), mencionando 9 frases distintas seleccionadas de una lista predefinida de 34. Se pueden escuchar varias frases como:
  - *I would like to meet Prof. Castro.*
  - *Where's the lift?*
  - *I'm afraid I didn't get that.*

En la siguiente tabla se explican algunas especificaciones relevantes sobre los dispositivos utilizados para la producción del conjunto de datos *AveRobot*:

ID	Model	Type	Resolution	FpS	Format	Height(cm)	Floor
1	Casio Exilim EXFH20	Compact Camera	1280 x 720	30	AVI	130	0
2	Huawei P10 Lite	Smartphone Camera	1920 x 1080	30	MP4		
3	Sony HDR-XR520VE	Video Camera	1920 x 1080	30	MTS	120	1
4	Samsung NX1000	Compact Camera	1920 x 1080	30	MP4		
5	iPhone 6S	Smartphone Camera	1280 x 720	30	MOV		
6	Sony DCR-SR90	Video Camera	702 x 576	25	MPG	150	2
7	Olympus VR310	Compact Camera	1280 x 720	30	AVI		
8	Samsung Galaxy A5	Smartphone Camera	1280 x 720	30	MP4		

Tabla 5.1: Especificaciones relevantes sobre los dispositivos usados en el conjunto de datos *AveRobot* [2]

Se debe mencionar que todos los vídeos cuyo formato no era `.mp4` fueron convertidos a dicho formato.

Los vídeos del conjunto de datos *Averobot* son anotados de manera que se tiene conocimiento de la identidad del individuo, el piso y la localización donde se grabó, la frase comentada y el ID del dispositivo (correspondiéndose al ID mostrado de la tabla 5.1).

En concreto, se tienen 3 carpetas (relacionándose cada una de ellas con un piso distinto) cuya nomenclatura de los vídeos en su interior es la siguiente:

`personId_floor_location_sentenceId_deviceId`

Por ejemplo:

`01_01_01_29_01.mp4`

Se corresponde a un vídeo de la identidad 1, en el piso 1, en la localización 1 (escaleras), realizando la locución con ID 29 (*Where's the lift?*) y usando la cámara con ID 1 (*Casio Exilim EXFH20*).

El sexo, la edad y la altura de cada participante también se encuentra incluido en un fichero de metadatos separado de los vídeos.

Con todas estas características y de acuerdo a la finalidad del conjunto de datos a la hora de ser diseñado descrito en sus artículos [2] [43], *AveRobot* está preparado para probar diferentes aplicaciones bajo escenarios de interacción hombre-máquina (*Human-Robot Interaction* o HRI), entre los que se encuentran por ejemplo:

- Verificación audiovisual. En el caso del presente proyecto, se hará uso del apartado visual para realizar la verificación.
- Reidentificación audiovisual.
- Reconocimiento del discurso audiovisual.

A continuación (en la figura 5.1) se muestran una serie de estadísticas que muestran la distribución del sexo por edad, de la altura de los participantes y de las frases comentadas respectivamente:

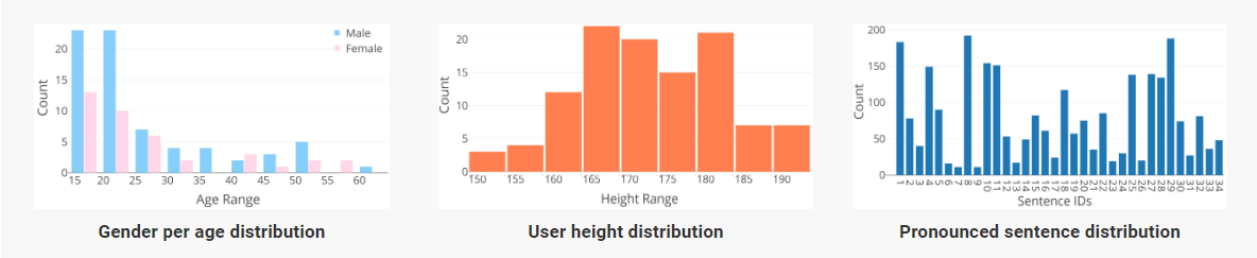


Figura 5.1: Estadísticas sobre la distribución de diferentes características del conjunto de datos *AveRobot* [3]

5.1.2 Subconjuntos de *AveRobot* escogidos y calidad de vídeos

Dada la extensión de *AveRobot* (2.664 vídeos), se ha decidido escoger una serie de subconjuntos de vídeos con el fin de realizar un análisis y evaluación de resultados dependientes de la calidad original de los datos.

Es imprescindible mencionar que el conjunto de datos supone un desafío en problemas relacionados con la verificación debido a que se abordan 111 identidades en condiciones de captura poco controladas por la variedad de sensores, resoluciones y capacidad de enfoque de las videocámaras, condiciones de iluminación y estabilidad en la grabación, alejándose en términos generales de un escenario "ideal".

A esto se le suman dificultades en menor medida provocadas por situaciones forzadas donde los individuos cambian parte de su complementos faciales, como puede ser el hecho de usar en un piso unas gafas para quitárselas en otro. Además, hay una serie de vídeos donde aparecen en su fondo otros individuos, lo que puede ser vital a la hora de construir la infraestructura del sistema de verificación.

Es por ello por lo que se hará uso a lo largo del estudio de la verificación en escenarios multinivel de los siguientes cuatro subconjuntos de datos, mostrando a continuación qué calidad esperar de cada uno de ellos:

Subconjunto de vídeos	Cámara del piso 1	Cámara del piso 2	Cámara del piso 3	Localización	Dificultad de verificación
A	2	3	8	Aleatoria (escalera, ascensor o pasillo)	Difícil
B	2	5	8	Ascensor	Normal
C	2	5	8	Pasillo	Muy difícil
D	2	5	8	Escalera	Muy difícil

Tabla 5.2: Subconjuntos de vídeos de *AveRobot* utilizados a lo largo del proyecto

5.1.2.1 Subconjunto de vídeos A

En el caso del subconjunto de vídeos A (ver figura 5.2), se tiene que las cámaras 2, 3 y 8 son las usadas. El hecho de que la cámara de vídeo 3 sea utilizada se convierte en una dificultad añadida debido a que no posee un sensor progresivo y produce el efecto de entrelazado (técnica que será explicada en la sección 6.1.2).

Otro problema que contiene este subconjunto es el hecho de la diversidad de localizaciones (siendo esta aleatoria), dado a que implica una mayor variedad de imágenes de la misma identidad en distintos pisos y se disminuye la facilidad de verificarla.

(a) Identidad 11 en la planta 1 en pasillo

(b) Identidad 11 en la planta 2 en ascensor

Figura 5.2: Ejemplo de fotogramas de la identidad 11 en el subconjunto de vídeos A

5.1.2.2 Subconjunto de vídeos B

En el subconjunto de vídeos B (ver figura 5.3), se tiene una verificación con una dificultad más estándar. En concreto (y como se repetirá en el resto de subconjuntos) se utiliza en el piso 2 la cámara de vídeo 5 (sustituyendo a la cámara de vídeo 3). Con esto, el problema del efecto de entrelazado desaparece dado a que, como se explicará en la sección 6.1.2, las cámaras de los *smartphones* usados son progresivas (aunque esto conlleve una pérdida de resolución desde 1920 x 1080 a 1280 x 720).

Además, como se mostró previamente en la tabla 5.2, en este y en los siguientes subconjuntos la localización es única (en este caso, ascensor), eliminando gran parte de la variedad de imágenes entre distintos pisos mencionada en el subconjunto de la sección 5.1.2.1 (donde la localización era aleatoria).

(a) Identidad 9 en la planta 1 en ascensor

(b) Identidad 9 en la planta 2 en ascensor

Figura 5.3: Ejemplo de fotogramas de la identidad 9 en el subconjunto de vídeos B

5.1.2.3 Subconjunto de vídeos C

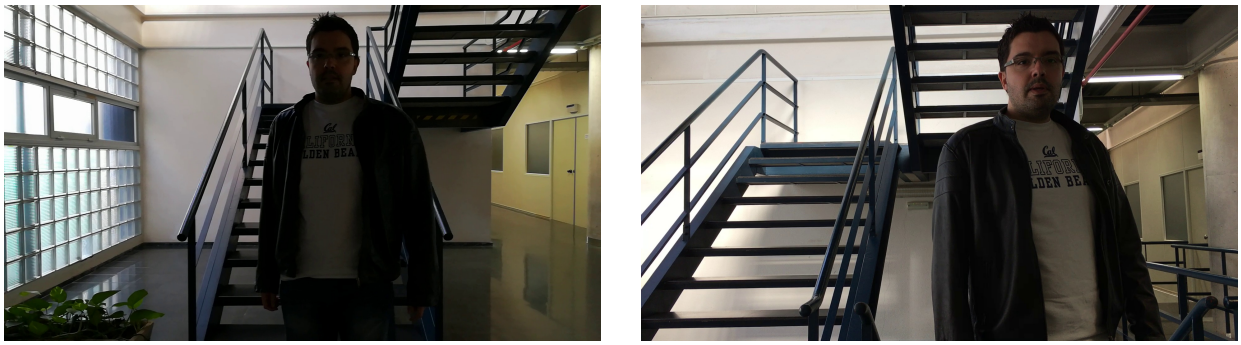
Respecto al subconjunto de vídeos C (ver figura 5.4), se tienen las mismas características que con el Subconjunto de vídeos B, con la excepción de que la localización es intercambiada por el pasillo en lugar del ascensor. Esta modificación provoca que, dada la calidad de los vídeos ubicados en el pasillo, se incremente la dificultad de la verificación.

(a) Identidad 108 en la planta 1 en pasillo (b) Identidad 108 en la planta 3 en pasillo

Figura 5.4: Ejemplo de fotogramas de la identidad 108 en el subconjunto de vídeos C

5.1.2.4 Subconjunto de vídeos D

En el subconjunto de vídeos D (ver figura 5.5), como ha sucedido con los anteriores dos subconjuntos, se ha realizado un cambio para ubicarlo en la última localización de *AveRobot* no usada en exclusividad aún, es decir, en la localización de la escalera (teniendo en cuenta que la calidad de estos vídeos es similar a los captados en el pasillo).



(a) Identidad 90 en la planta 1 en escalera (b) Identidad 90 en la planta 2 en escalera

Figura 5.5: Ejemplo de fotogramas de la identidad 90 en el subconjunto de vídeos D

5.2 Selección de conjuntos de test y entrenamiento

Una vez se tienen bien diferenciados los subconjuntos de *AveRobot* que se escogerán para los experimentos, se debe seleccionar la proporción que se destinará a las pruebas (test) y la que tendrá como objetivo un posible entrenamiento (*train*).

En primer lugar, se recuerda que los subconjuntos recogen los vídeos que pertenecen a una localización determinada (o aleatoria en caso de ser el Subconjunto de vídeos A) de cada identidad en cada piso con un solo dispositivo por planta (ver tabla 5.1 y 5.2 para más información).

Esto provoca que se modifique el hecho de tener 24 vídeos por persona (como se calculó en el apartado 5.1.1) a tener 3 vídeos por persona. Teniendo en cuenta que el conjunto de datos contiene un total de 111 individuos, se tienen $111 * 3 = 333$ vídeos en cada subconjunto.

El conjunto de identidades se ha dividido en dos subconjuntos disjuntos: un conjunto con 70 identidades (210 vídeos, 63.06 %) que formará el conjunto de entrenamiento y otro de 41 identidades (123 vídeos, 36.94 %) que formará el conjunto de test (ver figura 5.6).

Es importante mencionar el hecho de que la división debe realizarse sobre las identidades en lugar de sobre los vídeos. Esto es debido a que tanto las pruebas como los entrenamientos se deben poder realizar con pares de vídeos que pertenezcan a la misma identidad. En otras palabras, debe existir disponibilidad de los vídeos de cada identidad de los conjuntos de test y entrenamiento en todos los pisos.

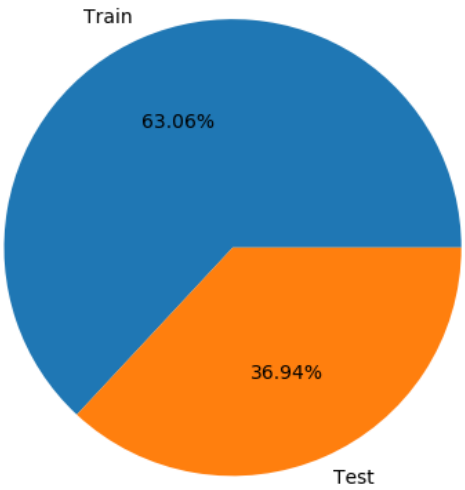


Figura 5.6: Distribución de los conjuntos de test y entrenamiento en los experimentos

Para lograr obtener estos conjuntos, se ha desarrollado el módulo `fileindexer` disponible en el repositorio del proyecto [30], el cual hace uso de las bibliotecas `os` [64], `random` [65], `datetime` [66], `glob` [67] y `pathlib` [68].

El fichero del módulo (`file_indexer.py`) contiene dos funciones explicadas a continuación:

Función	Descripción
`create_indexer_files(files_pairs=1, training_identities_number=55, test_identities_number=56, cameras=(2, 3, 8), place=None)`	Genera pares de ficheros `.txt` de entrenamiento y test con identidades aleatorias tras realizar una comprobación de la coherencia de los valores de sus parámetros (por ejemplo, que las cámaras escogidas para cada piso estén disponibles en dicho piso) haciendo uso de la función `create_content(...)`.
`create_content(identities_list, identity, cameras, place)`	Realiza búsquedas en el directorio donde se localiza *AveRobot* con el objetivo de encontrar los 3 vídeos correspondientes con una identidad (1 vídeo por piso) respetando las cámaras y la localización pedidas por parámetros. Se incluye la posibilidad de que la localización sea aleatoria (utilizada en el Subconjunto de vídeos A).

Tabla 5.3: Funciones presentes en el fichero `file_indexer.py`

Resulta destacable el hecho de describir cómo se hace uso de las expresiones regulares de la consola de *Unix* mediante `glob` para buscar los vídeos apropiados para una determinada identidad teniendo en cuenta la nomenclatura de los vídeos de *AveRobot* (ver apartado 5.1.1):

```
first_floor = glob.glob(os.getenv('VIDEOS_ROUTE') + "/averobot_floor_01/"
                        + str(identities_list[identity]).zfill(2)
                        + "_01_" + place[0] + "_*_" + str(cameras[0]).zfill(2)
                        + ".mp4")

second_floor = glob.glob(os.getenv('VIDEOS_ROUTE') + "/averobot_floor_02/"
                         + str(identities_list[identity]).zfill(2)
                         + "_02_" + place[1] + "_*_" + str(cameras[1]).zfill(2)
                         + ".mp4")

third_floor = glob.glob(os.getenv('VIDEOS_ROUTE') + "/averobot_floor_03/"
                        + str(identities_list[identity]).zfill(2)
                        + "_03_" + place[2] + "_*_" + str(cameras[2]).zfill(2)
                        + ".mp4")
```

Fragmento de código 5.1: Búsqueda personalizada de vídeos de una identidad en el conjunto de datos *AveRobot* [Fuente]

Si se estudia el fichero del repositorio [30] `main-lift.ipynb` en la sección 2.2, se podrá observar un ejemplo para obtener un par de ficheros (test y entrenamiento) para la zona del ascensor:

```
file_indexer.create_indexer_files(files_pairs=1, training_identities_number=70,
                                  test_identities_number=41, cameras=(2,5,8), place=2)
```

Fragmento de código 5.2: Creación de par de ficheros de test y entrenamiento en la zona del ascensor para el Subconjunto de vídeos B [Fuente]

La estructura final de dichos ficheros es similar a la que se muestra en el siguiente ejemplo:

```
averobot_floor_01/59_01_02_04_02.mp4
averobot_floor_02/59_02_02_10_05.mp4
averobot_floor_03/59_03_02_11_08.mp4
averobot_floor_01/51_01_02_19_02.mp4
averobot_floor_02/51_02_02_15_05.mp4
averobot_floor_03/51_03_02_25_08.mp4
averobot_floor_01/37_01_02_08_02.mp4
averobot_floor_02/37_02_02_15_05.mp4
averobot_floor_03/37_03_02_06_08.mp4
                    .
                    .
                    .
averobot_floor_01/42_01_02_01_02.mp4
averobot_floor_02/42_02_02_19_05.mp4
averobot_floor_03/42_03_02_30_08.mp4
averobot_floor_01/101_01_02_01_02.mp4
averobot_floor_02/101_02_02_08_05.mp4
averobot_floor_03/101_03_02_10_08.mp4
averobot_floor_01/86_01_02_10_02.mp4
averobot_floor_02/86_02_02_25_05.mp4
averobot_floor_03/86_03_02_08_08.mp4
```

Capítulo 6

Desarrollo

El desarrollo del proyecto ha constado de una metodología basada en prototipos con diferentes modificaciones y módulos. Por ello, al describir el presente capítulo se hablará de las distintas fases que lo componen (en la figura 6.1 se muestran los principales módulos del sistema).

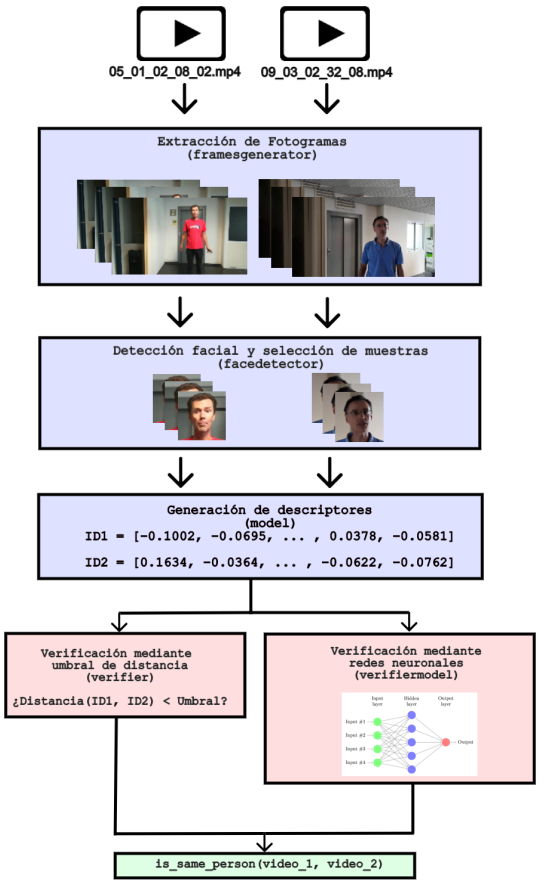


Figura 6.1: Diagrama de bloques con los principales módulos del sistema

6.1 Extracción de fotogramas

En el sistema de verificación que se propone en este proyecto, es necesario trasladar el concepto de captación a través de los robots asistentes a imágenes independientes que permitan un correcto tratamiento de los datos a lo largo de dicho sistema.

Por este motivo, tras haber hecho una división de *AveRobot* en una serie de subconjuntos (ver sección 5.1) y seleccionado la proporción de test y entrenamiento (ver sección 5.2), es necesario diseñar un módulo que permita descomprimir un vídeo en fotogramas (o *frames*) independientes.

6.1.1 Generador de fotogramas desde vídeos

El módulo `framesgenerator` presente en el repositorio del proyecto [30] permite llevar a cabo la labor relacionada con la generación de fotogramas desde los vídeos de *AveRobot*.

Para lograr esta funcionalidad, se ha hecho uso de las siguientes nuevas bibliotecas: `cv2` [19] (ver sección 3.1.2.3), `re` [69], `numpy` [70] y `pillow` [71].

En este caso, tan solo se tiene una función en el módulo:

Función	Descripción
`generate_frames(filename)`	Genera los fotogramas dado el nombre de un vídeo de *AveRobot*. Realiza un tratamiento especial para aquellos vídeos captados por videocámaras que generan un efecto de entrelazado.

Tabla 6.1: Función presente en el fichero `frames_generator.py`

Mediante el siguiente algoritmo sencillo y con ayuda de las utilidades de la biblioteca de OpenCV para Python, se descomprime el vídeo en fotogramas separados para su tratamiento posterior:

```
video = cv2.VideoCapture(os.getenv('VIDEOS_ROUTE') + "/" + filename)
count = 1
while True:
    state, image = video.read()
    if not state:
        break

    [...]

    cv2.imwrite(directory + "/frame-" + str(count).zfill(3) + ".jpg", image)
    count += 1

video.release()
```

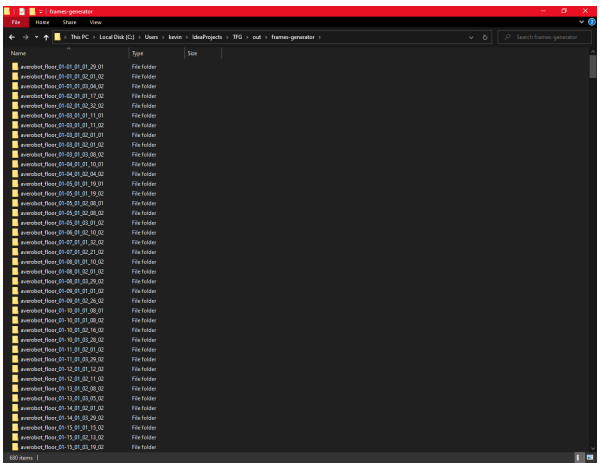
Fragmento de código 6.1: Generación de fotogramas desde vídeos de *AveRobot* [Fuente]

Haciendo uso de la función previamente detallada, se puede proceder a generar todos los fotogramas de todos los vídeos de cualquier subconjunto planteado en las anteriores secciones. Es en esta situación cuando sucede la conexión entre el módulo `fileindexer` (ver sección 5.2) y el módulo `framesgenerator`, ya que este último comenzará a descomprimir en fotogramas los vídeos cuyos nombres son proporcionados por los ficheros de texto del primero. Dicho comportamiento se puede analizar a continuación (sección 2.3.2 del fichero `main-lift.ipynb`):

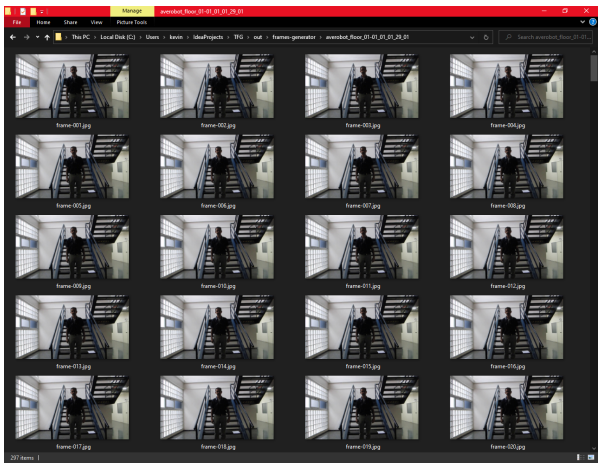
```
file_indexer_file = open(test_file)
videos = file_indexer_file.readlines()
[...]
for video in videos:
    frames_generator.generate_frames(video.strip())
    [...]

file_indexer_file.close()
```

Fragmento de código 6.2: Generación de fotogramas desde vídeos de *AveRobot* en uno de los subconjuntos de test [Fuente]

Tras la generación de los fotogramas, se obtendrá una serie de carpetas como las que se muestran en la figura 6.2 que contendrán los fotogramas de los vídeos descritos en los ficheros de texto:



(a) Carpetas con los vídeos descomprimidos



(b) Fotogramas del vídeo `01_01_01_29_01.mp4`

Figura 6.2: Ejemplo de salida del módulo `framesgenerator`

6.1.2 Desentrelazado de fotogramas

La exploración entrelazada [72] (usada en las videocámaras de *AveRobot*) es un método de adquisición de imágenes cuyo funcionamiento se basa en dividir la imagen que se quiere transmitir en dos campos o fotogramas formados por líneas pares e impares.

La implantación de la exploración entrelazada en la televisión analógica vino dada por ser una forma pragmática de comprimir una imagen que, en lugar de transmitir fotogramas a 25 Hz, transmitía este tipo de fotogramas mezclados a 50 Hz a costa de reducir la definición vertical y empeorar en consecuencia la visualización de la imagen.

Al realizar una exploración entrelazada y utilizar dos fotogramas para formar uno, se evita una posible sensación de parpadeo (o *flicker*).

Otro método de adquisición de imágenes es la exploración progresiva, la cual propone un método de exploración secuencial (usada en *smartphones* de *AveRobot*) de las líneas de una imagen con un barrido sucesivo de todas ellas para componer la imagen. Sin embargo, se requiere mayor ancho de banda para la transmisión que en la exploración entrelazada.

Figura 6.3: Diferencia entre fotogramas con exploración entrelazada y progresiva [Fuente]

En el caso de *AveRobot*, se tiene que las videocámaras 3 y 6 (ver tabla 5.1) proporcionan vídeo entrelazado (el cual provoca que a la hora de generar un fotograma de un vídeo grabado con ese dispositivo, se observen dos fotogramas mezclados, como se muestra en la figura 6.3).

Para el sistema de verificación, es imprescindible evitar esta situación de entrelazado de fotogramas. Por ello, se inicia un algoritmo para eliminar dicho efecto (aunque esto conlleve una pérdida de líneas horizontales y por tanto, de calidad de la imagen).

En primer lugar, se detecta si el vídeo que se está descomprimiendo en fotogramas en el módulo `framesgenerator` ha sido grabado con un dispositivo con dicho efecto haciendo uso de la biblioteca `re` (mencionada previamente en el apartado 6.1.1):

```
used_camera = int(re.findall('\d+', filename)[5])
if used_camera == 3 or used_camera == 6:
    deinterlace_frame = True
else:
    deinterlace_frame = False
```

Fragmento de código 6.3: Identificación de los vídeos con dispositivos con efecto de entrelazado [Fuente]

A continuación, si el fotograma se está obteniendo desde un dispositivo de estas características, se procede a eliminar la mitad de las líneas horizontales usando la biblioteca `pillow` (mencionada en la sección 6.1.1) para, por último, redimensionar la imagen al tamaño original. En este proceso, se recoge el píxel "vecino más cercano" (`Image.NEAREST`) a la hora de gestionar la redimensión que elimina la mitad de las líneas horizontales del fotograma:

```
if deinterlace_frame:
    image = Image.fromarray(image)
    size = list(image.size)
    image = image.resize([size[0], int(size[1]/2)], Image.NEAREST)
    image = numpy.array(image.resize(size))
```

Fragmento de código 6.4: Eliminación del efecto de entrelazado eliminando la mitad de líneas horizontales [Fuente]

Al haber eliminado la mitad de las líneas horizontales, se ha eliminado uno de los fotogramas que constituían el fotograma entrelazado, suprimiendo dicho efecto y obteniendo inevitablemente una definición vertical de menor calidad que un fotograma generado por exploración progresiva en el proceso.

Para finalizar esta sección, se mostrará un ejemplo en la figura 6.4 de desentrelazado en uno de los fotogramas de *AveRobot*:

(a) Fotograma con efecto de entrelazado

(b) Fotograma sin efecto de entrelazado

Figura 6.4: Ejemplo de eliminación de efecto de entrelazado con la identidad 1

6.2 Detección facial y selección de muestras

Una vez se han obtenido los fotogramas, se debe comenzar el desarrollo de las detecciones faciales y la selección de muestras, que servirá como núcleo a la hora de comenzar a comparar los diferentes individuos a verificar para decidir si se tratan del mismo o no.

6.2.1 Medidas de calidad de detecciones

En primer lugar, se analizarán diferentes medidas que indican la calidad de una detección facial. Es imprescindible obtener dicha calidad con el fin de recoger aquellas detecciones que puedan otorgar mayor cantidad de detalle de los individuos a verificar.

En concreto, se ha calculado en todos los casos una suma ponderada de las siguientes tres medidas propuestas: el *Confidence* propio del detector ($conf_{det}$), la medida BRISQUE ($brisque$) y la Medida de contorno ($cont$):

$$confidence = \omega_1 * conf_{det} + \omega_2 * brisque + \omega_3 * cont$$

Teniendo en cuenta que:

$$\omega_1 + \omega_2 + \omega_3 = 1$$
$$conf_{det}, brisque, cont \ \epsilon \ [0, 100]$$

6.2.1.1 *Confidence* propio del detector

Una de las medidas que se ha tenido en cuenta es la propia "confianza" (o *confidence*) devuelta por el detector.

La gran mayoría de detectores desarrollados en los últimos años generan este parámetro numérico junto a los resultados de la propia detección con el objetivo de entregar información valiosa al usuario que esté haciendo uso de dicho detector sobre la confianza, certeza o seguridad que posee la utilidad sobre la detección resultante. Así pues, una confianza mayor dará lugar a una detección más definida y clara mientras que una confianza menor resultará en una detección más ambigua y compleja.

Se debe mencionar en este punto que los desarrolladores utilizan dicha confianza para definir límites en la capacidad de detección de objetos (o, en este caso, rostros de personas). Generalmente, se suele utilizar un "umbral" que trata de descartar aquellas detecciones que no son lo suficientemente aptas como para considerarse un cuerpo detectado en el sistema.

Por este motivo, el umbral de la confianza del detector se relaciona normalmente con los falsos positivos y los falsos negativos, dado a que los primeros serán inversamente proporcionales y los segundos proporcionales a una confianza mayor. En otras palabras, con un umbral de mayor confianza se reducirán los resultados donde la detección que se ha realizado realmente era errónea y se incrementarán los casos donde las detecciones se debían haber procesado y no llegaron a hacerlo. Evidentemente, y en líneas generales, con un umbral del confianza

menor se produce el efecto adverso; se incrementan los falsos positivos y se reducen los falsos negativos.

Cada uno de los detectores propuestos (secciones 4.1, 4.2.1 y 4.2.2) propone un parámetro de confianza con cierta diversidad (ver figura 6.5) que será explicada en sus respectivos apartados junto a su tratamiento con el resto de parámetros de medidas de calidad en las detecciones.

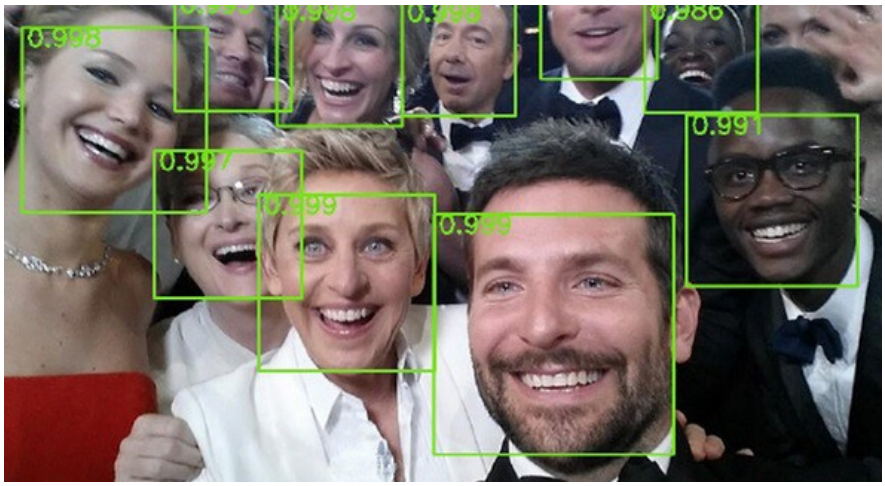


Figura 6.5: Ejemplo del parámetro de confianza de un detector en sus resultados [Fuente]

6.2.1.2 BRISQUE

Antes de explicar qué es una medida BRISQUE [73], se debe explicar qué es un IQA o *Image Quality Assessment* (Evaluación de la Calidad de Imagen).

Un algoritmo IQA trata de recoger como entrada una imagen para a continuación devolver una puntuación de su calidad como salida. Existen tres tipos de IQA:

- *Full-Reference IQA*: se tiene además de la imagen a evaluar, una imagen de referencia "limpia" (no distorsionada). Se usa principalmente para conocer la calidad de un algoritmo de comprensión de imagen, donde se puede comparar la original (o no distorsionada) y dicha imagen tras la mencionada compresión.
- *Reduced-Reference IQA*: similar a la *Full-Reference IQA* con la diferencia de que la imagen original no sirve de referencia completa, sino de añadido de información sobre ella (por ejemplo, una imagen con marca de agua).
- *Objective Blind IQA* o *No-Reference IQA*: se tiene una única imagen con el objetivo de medir su calidad sin comparar con ninguna otra imagen de referencia. BRISQUE es una medida sin referencia, por lo que se encuentra dentro del *No-Reference IQA*.

Una vez explicados los algoritmos IQA, se procederá a explicar en qué consiste BRISQUE [74] o *Blind/Referenceless Image Spatial Quality Evaluator*.

BRISQUE, como se explicó previamente, se trata de una medida de calidad de imagen sin referencia. Entre menor sea su valor, se considera que la imagen tiene mayor calidad. Dicho

valor se calcula mediante los siguientes pasos descritos de manera somera, tal y como se representa en la figura 6.6:

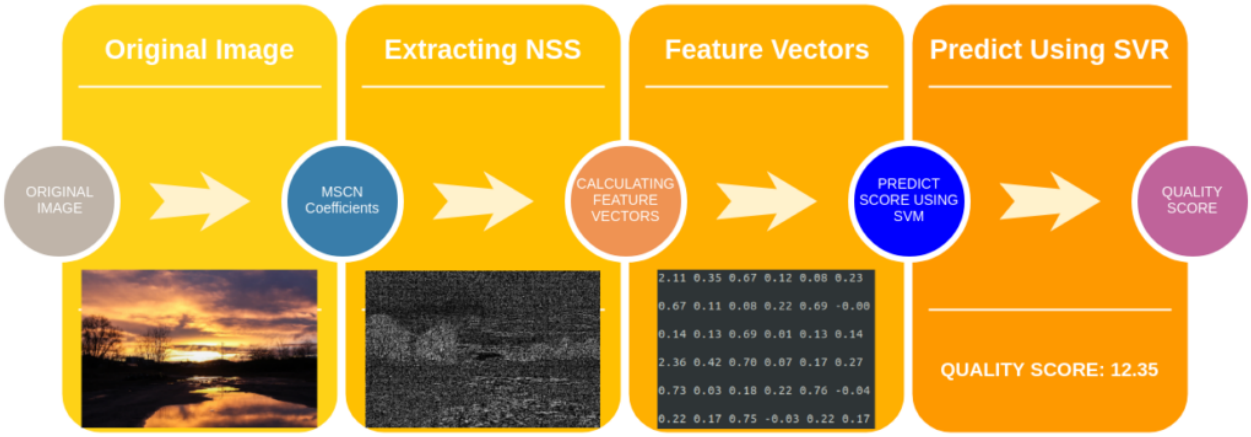


Figura 6.6: Pasos para calcular la medida BRISQUE [73]

1. Se recoge la imagen a la cual se le quiere calcular la medida de calidad.
2. Se extraen las estadísticas de escena natural (o NSS). Para ello, se debe tener en cuenta que la intensidad de los píxeles de las imágenes originales difiere de las que están distorsionadas y que existe la posibilidad de normalizar dichas intensidades con el fin de que la distribución resultante sea más pronunciada. Con este paso se pretende medir la desviación de dicha distribución como medida de la cantidad de distorsión/deformación de la imagen.
3. Se calculan los vectores de características (o *feature vectors*). Una vez se han derivado varias imágenes para el cálculo de las NSS desde la original, estas son usadas para desarrollar vectores de dimensión 36x1. Para ello, se encajan en distribuciones gaussianas generalizadas básicas y asimétricas, desde donde se extraen parámetros numéricos relacionados con la forma, la media y la desviación.
4. Por último, se hace uso de dichos vectores y de una SVM o *support vector machine* (Máquina de Vector Soporte [75]) para calcular un valor definitivo que mida la calidad de la imagen. Se recuerda que el objetivo de una SVM como algoritmo de aprendizaje supervisado es construir hiperplanos óptimos en el que el margen de separación entre dos clases de los datos sea maximizado dado unos vectores de soporte de un subconjunto de entrenamiento que son usados con el objetivo de aprender a clasificar o a realizar una regresión.

En el caso de este proyecto, se ha hecho uso de la biblioteca `image-quality` [76], la cual proporciona una medida BRISQUE dada una imagen del sistema de ficheros del sistema operativo.

Para ello, tras utilizar las bibliotecas ya mencionadas `cv2` (sección 3.1.2.3) y `pillow` [71] para convertir el fotograma con la detección facial a escala de grises y, posteriormente, a una imagen de `pillow`, se recoge dicha medida.

```
face_matrix = self.prepare_face(detection, directory)
gray_scale_image = cv2.cvtColor(face_matrix, cv2.COLOR_BGR2GRAY)
pil_gray_scale_image = Image.fromarray(gray_scale_image)
detection['confidence'] = [...] (1 / (1 + brisque.score(pil_gray_scale_image)))
```

Fragmento de código 6.5: Cálculo de medida BRISQUE en el módulo `facedetector`

Se recuerda que la medida BRISQUE indica una mejor calidad de imagen cuanto menor sea. Por este motivo, se realiza la fórmula que se puede visualizar en la cuarta línea del fragmento de código 6.5.

$$brisque = \frac{1}{1 + brisque_{score}}$$

Con lo que, sabiendo que una medida BRISQUE con los fotogramas de *AveRobot* oscila entre [0-100], se tiene que la confianza otorgada por parte de BRISQUE en el sistema será de [0-1]. Evidentemente, el denominador contiene la suma de la unidad para evitar dividir por 0 y asegurar un valor máximo de 1.

Por último, se adjunta en la figura 6.7 unas muestras del valor BRISQUE para una detección de la identidad 79 del conjunto de datos *AveRobot*, a la cual se le va aplicando progresivamente un desenfoque gaussiano:

(a) Detección original (b) BRISQUE: 85.58 (c) BRISQUE: 97.71 (d) BRISQUE: 98.06

Figura 6.7: Medida BRISQUE para una detección de la identidad 79

6.2.1.3 Medida de contorno

La tercera y última medida que participa en la suma ponderada del parámetro de confianza es la medida de contorno.

La medida de contorno trata la proporción de píxeles en una imagen que indican bordes o siluetas. Para detectar dichas siluetas en una imagen se hace uso del detector de contornos de Canny implementado en OpenCV mediante `Canny(imagen, threshold_1, threshold_2)`.

La detección de bordes de Canny [77] es un algoritmo desarrollado por John F. Canny en 1986. Se compone de los siguientes pasos:

1. Reducción de ruido. Usando un filtro gaussiano, se trata de limpiar el posible ruido de la imagen para facilitar la detección de bordes.

2. Búsqueda del gradiente de la intensidad de la imagen. La imagen es filtrada por un núcleo (o *kernel*) de Sobel (ver figura 6.8) en las direcciones horizontales y verticales para obtener la primera derivada en ambas direcciones. Con esta información, se puede encontrar para cada píxel el gradiente del borde y la dirección. El gradiente de la dirección es perpendicular a los bordes y se redondea a uno de los ángulos que representan la dirección vertical, horizontal o diagonal.

(a) Imagen original [Fuente] (b) Imagen tras núcleo de Sobel [Fuente]

Figura 6.8: Transformación de imagen filtrada por un núcleo de Sobel

3. A continuación, se eliminan aquellos píxeles que no constituyen ningún borde. Para lograr esto, cada píxel es revisado para verificar si es un máximo local con su vecino en la dirección del gradiente calculado. Tras este proceso, la imagen pasa a estar formada de píxeles únicamente blancos para los bordes y negros para el resto de la imagen.

4. Por último, se realiza un "umbralizado de histéresis", con el objetivo de seleccionar los bordes que realmente lo son y los que no. Es en este último tramo donde se hace uso de los parámetros `threshold_1` y `threshold_2`, siendo respectivamente el umbral menor y el umbral mayor. Todos los píxeles que estén por debajo del umbral menor no son bordes y aquellos cuyo valor sea superior al umbral mayor lo son. El resto son estudiados dependiendo de su conectividad con el resto de bordes (aquellos que recientemente se decidieron clasificar como tales). Este último paso también elimina ruido generado por pequeños píxeles dado a que se considera que un borde debe ser una línea de un mínimo de píxeles.

En el caso de este proyecto, se convierte la detección facial a escala de grises para, a continuación, hacer uso de la función de OpenCV descrita previamente. Por último, y teniendo en cuenta que la imagen obtendrá valores negros (representados por un 0) y valores blancos (representados mediante un 255), se calcula la proporción de píxeles de la imagen considerada

como bordes.

```
face_matrix = self.prepare_face(detection, directory)
gray_scale_image = cv2.cvtColor(face_matrix, cv2.COLOR_BGR2GRAY)
pil_gray_scale_image = Image.fromarray(gray_scale_image)
edges_image = cv2.Canny(gray_scale_image, 100, 200)
detection['confidence'] = [...] (10 * (numpy.sum(edges_image / 255) /
                                (edges_image.shape[0] * edges_image.shape[1]))) [...]
```

Fragmento de código 6.6: Cálculo de medida de contorno en el módulo `facedetector`

Dado a que, en circunstancias comunes, un 10 % de una imagen de un rostro puede llegar a contener bordes, el resultado de la siguiente operación:

$$outline_{rate} = \frac{\sum white_{pixels}}{total_{pixels}}$$

Será multiplicado por 10. De este modo, si una imagen tiene un 10 % de píxeles de bordes o contorno, se considerará que tiene la proporción adecuada y se considerará una detección de buena calidad (teniendo en cuenta el resto de Medidas de calidad de detecciones).

Como se describió en la figura 6.7, se expondrá un ejemplo en la figura 6.9 del cálculo de la medida de contorno con la misma detección que se mostró tras aplicaciones progresivas de desenfoques gaussianos (los resultados en este caso representan la proporción sin ser multiplicados por 10):

(a) Detección original (b) Ratio de bordes: 0.0161 (c) Ratio de bordes: 0.0037 (d) Ratio de bordes: 0.0011

Figura 6.9: Medida de contorno para una detección de la identidad 79

6.2.2 Detección facial

En esta sección, se explicará la implementación de los distintos detectores usados en el proyecto. Concretamente, se hablará del uso del Detector MTCNN, del Detector DLIB - HOG y del Detector DLIB - MMOD, finalizando por la Extracción de elementos faciales para los dos últimos.

6.2.2.1 Implementación del detector MTCNN

A continuación, se detallará la implementación del Detector MTCNN (descrito en el estado del arte en la sección 4.1).

Tras la generación de fotogramas desde vídeos (explicado en el apartado 6.1), se comenzará a procesar todas sus detecciones faciales.

Con este objetivo, se ha desarrollado el módulo `facedetector` y, con la finalidad de hacer uso de la biblioteca explicada en el apartado 3.1.2.5, el fichero `mtcnn_detector.py`. Las nuevas bibliotecas, además de `termcolor` [78] y `matplotlib` [79], que se utilizan en este nuevo módulo del sistema de verificación son las siguientes:

- `pickle` [80]: implementa una serialización (y deserialización) de una estructura de un objeto de Python. Dicha serialización permitirá archivar los resultados de las detecciones en diferentes ficheros dependientes del vídeo original con el objetivo de no repetir las detecciones de un tipo siempre y cuando ya hayan sido previamente procesadas.
- `FaceNormalizationUtils`: fichero con utilidades para la normalización o alineamiento de caras proporcionado por uno de los tutores del presente Trabajo de Fin de Grado, Modesto Fernando Castrillón Santana.

En concreto, las funciones que se pueden encontrar en dicho fichero son las siguientes:

Función	Descripción
`get_false_negative_rate(false_negative_samples, true_positive_samples)`	Obtiene la proporción de falsos negativos dada una cantidad de falsos negativos y positivos reales.
`get_false_positive_rate(false_positive_samples, true_negative_samples)`	Obtiene la proporción de falsos positivos dada una cantidad de falsos positivos y negativos reales.

Tabla 6.2: Funciones presentes en el fichero `mtcnn_detector.py`

Respecto al objeto `MTCNNDetector` y sus métodos se tiene:

Método	Descripción
`__init__(self, max_detections=10, pickle_results=True, image_results=False, force_update=False, normalize_face='None', confidence_range=0.00001, frames_directory='frames-generator/'`	Inicializa el objeto `MTCNNDetector` personalizado a una serie de parámetros.
`process_video(self, frames_folder_name)`	Procesa detecciones de los fotogramas de un vídeo y realiza diversas acciones dependiendo de los parámetros con los que se inicializó el objeto.
`prepare_results_list(self, detection_list, current_max_confidence)`	Devuelve una lista de detecciones que trata de recoger las mejores procesadas con un margen de separación en la confianza entre ellas para evitar detecciones de fotogramas cercanos temporalmente (ver sección 6.2.4.2).
`detect_face(self, image)`	Detecta un rostro dada una imagen.
`@staticmethod prepare_face(face, directory)`	Devuelve un *array* de 160x160 con el rostro detectado.
`save_pickle_results(self, pickle_data, pickle_directory)`	Guarda en un fichero `.pickle` los datos generados por `process_video(...)`

Tabla 6.3: Métodos presentes en el fichero `mtcnn_detector.py`

```
[{
    'box': [277, 90, 48, 63],
    'keypoints': {
        'nose': (303, 131),
        'mouth_right': (313, 141),
        'right_eye': (314, 114),
        'left_eye': (291, 117),
        'mouth_left': (296, 143)
    },
    'confidence': 0.99851983785629272
}]
```

Fragmento de código 6.7: Ejemplo de salida de una detección de la biblioteca `MTCNN` [Fuente]

El algoritmo que sigue la detección haciendo uso de la biblioteca de MTCNN es el siguiente:

```
for index, frame in enumerate(video_frames):
    [...]
    if frame.startswith("frame-") and frame.endswith(".jpg"):
        pixels_array = numpy.asarray(Image.open(directory + frame))
        detection = self.detect_face(pixels_array)
        if detection is not None:
            detection['frame_number'] = frame
            face_matrix = self.prepare_face(detection, directory)
            gray_scale_image = cv2.cvtColor(face_matrix, cv2.COLOR_BGR2GRAY)
            pil_gray_scale_image = Image.fromarray(gray_scale_image)
            edges_image = cv2.Canny(gray_scale_image, 100, 200)
            detection['confidence'] = 0.0125 * (10 * (numpy.sum(edges_image / 255) /
                                      (edges_image.shape[0] * edges_image.shape[1])) \
                                      + 0.975 * detection['confidence'] \
                                      + 0.0125 * (1 / (1 + brisque.score(pil_gray_scale_image)))
            [...]
            detection_list.append(detection)
```

Fragmento de código 6.8: Procesamiento de detecciones haciendo uso de MTCNN [Fuente]

Como se observa en el fragmento de código 6.8, se comienza fotograma a fotograma a detectar si existen caras de personas en ellos. Tras una detección exitosa, se incluye el número de fotograma dentro del diccionario resultante (ver Ejemplo de salida de una detección de la biblioteca `MTCNN`) y se comienza a realizar el proceso explicado en el apartado 6.2.1, donde una suma ponderada de los tres marcadores de calidad de imagen otorgarán un valor numérico a la calidad de la detección. En este caso, y debido a los valores altos que suele proporcionar como confianza propia MTCNN, se ha decidido tras varias pruebas y evaluaciones que la suma ponderada para este detector sea la siguiente:

$$confidence = 0.975 * conf_{det} + 0.0125 * brisque + 0.0125 * cont$$

Es necesario mencionar que la biblioteca devuelve los rostros detectados ordenados por la confianza del propio detector. Como en principio (y aunque no se cumpla en la totalidad del conjunto de datos) solo existe un individuo en cada vídeo, se propone el siguiente algoritmo para devolver la detección de la biblioteca:

```
def detect_face(self, image):
    detection = self.detector.detect_faces(image)
    if detection:
        return detection[0]

    return None
```

Fragmento de código 6.9: Implementación del método `detect_face(self, image)` en MTCNN [Fuente]

Haciendo uso del método estático desarrollado `prepare_face(face, directory)`, se podrá realizar el tratamiento de la matriz de píxeles de un rostro a través de un cuadro delimitador desde su fotograma original.

Introduciéndole como parámetro la salida modificada de MTCNN (ver fragmento de código 6.7) y tras obtener la matriz de píxeles de la imagen original, se recogen las coordenadas devueltas por el detector (punto de la esquina superior izquierda, ancho y alto). Después de obtener dichas coordenadas, se recoge la porción de la matriz de píxeles de la imagen original correspondiente a la cara detectada para por último devolverla con un tamaño de 160x160.

El siguiente fragmento de código es un ejemplo donde se construye el objeto `MTCNNDetector` y se detectan los rostros de los fotogramas de los vídeos:

```
mtcnn_d = mtcnn_detector.MTCNNDetector(max_detections=4, pickle_results=True,
                                       image_results=False, force_update=False,
                                       normalize_face='resize', confidence_range=0.00001)

for video in test_videos_out:
    mtcnn_d.process_video(video.strip())
```

Fragmento de código 6.10: Ejemplo de detección de caras en el sistema de verificación con MTCNN [Fuente]

Los resultados son almacenados en ficheros `.pickle` con la estructura mostrada en el fragmento de código 6.7 añadiéndole el número de fotograma (`detection['frame_number']`).

6.2.2.2 Implementación del detector DLIB - HOG

En este apartado se explicará la implementación del Detector DLIB - HOG (explicado en el estado del arte en la sección 4.2.1).

En el caso de este proyecto, se ha hecho uso del módulo desarrollado `facedetector`, el cual contiene el fichero `dlib_detector.py`.

Dicho fichero utiliza la implementación de la biblioteca `dlib` para Python. La estructura del fichero es similar a la del fichero `mtcnn_detector.py` dado a que ambos tienen el objetivo de desarrollar la misma salida (cuadro delimitador, puntos faciales y confianza).

Debido a dicha similitud, las tablas de funciones y métodos (tablas 6.2 y 6.3) no se volverán a mostrar, dado a que la única diferencia se encuentra en el constructor y en el añadido del método `get_facial_landmarks(...)` (la implementación interna de los métodos sí que difiere respecto a lo desarrollado en `mtcnn_detector.py`):

Método	Descripción
`__init__(self, max_detections=10, pickle_results=True, image_results=False, force_update=False, detector_type='mmod', confidence_range=0.02, frames_directory='/out/frames-generator/'`	Inicializa el objeto `DLIBDetector` personalizado a una serie de parámetros.
`get_facial_landmarks(self, pixels_array, face)`	Devuelve puntos faciales situados en los mismos lugares que devuelve MTCNN (ver apartado 6.2.2.4).

Tabla 6.4: Métodos presentes en el fichero `dlib_detector.py`

Es importante señalar que para hacer uso del detector HOG de Dlib se ha desarrollado el siguiente fragmento de código:

```
if detector_type == 'hog':
    self.detector = dlib.get_frontal_face_detector()
```

Fragmento de código 6.11: Creación del detector HOG de Dlib [Fuente]

También es necesario mencionar que respecto a la medida de calidad de la detección, teniendo en cuenta que los valores de confianza devueltos por HOG (ver sección 6.2.1.1) no son tan altos y tras varios tests, se ha propuesto la siguiente suma ponderada:

$$confidence = 0.5 * conf_{det} + 0.25 * brisque + 0.25 * cont$$

Respecto al método `detect_face(...)`, ha sufrido el siguiente cambio en su implementación con HOG:

```
def detect_face(self, image):
    if self.detector_type == 'hog':
        detection, scores = self.detector.run(image, 0, 0)[:2]
    [...]

    if detection and scores:
        return [detection[0], scores[0]]

    return None
```

Fragmento de código 6.12: Implementación del método `detect_face(self, image)` en Dlib-HOG [Fuente]

En este caso, cuando se ejecuta el `run` del detector, sus parámetros se corresponden respectivamente a la imagen, a si realizar *upsampling* [1] y a establecer un umbral (*threshold*) donde un valor menor dará lugar a que se admitan detecciones que el detector Dlib-HOG encuentre con una confianza baja. La variable `detection` recogerá el cuadro delimitador y la variable `scores` la confianza del detector (ver apartado 6.2.1.1).

De nuevo, como *AveRobot* es un conjunto de datos en los cuales en circunstancias normales aparece una sola persona por vídeo, se recoge únicamente la primera detección encontrada.

Como se mostró con el Detector MTCNN, se recogerían las detecciones para los vídeos de la siguiente manera (presente en el cuaderno de Jupyter `main-lift.ipynb` del repositorio del proyecto [30]):

```
dlib_d = dlib_detector.DLIBDetector(max_detections=4, pickle_results=True,
                                    image_results=False, force_update=False,
                                    detector_type='hog', confidence_range=0.003)

for video in test_videos_out:
    dlib_d.process_video(video.strip())
```

Fragmento de código 6.13: Ejemplo de detección de caras en el sistema de verificación con Dlib-HOG [Fuente]

Los resultados, como ya se mencionó en la explicación del Detector MTCNN son almacenados en ficheros `.pickle` con la estructura mostrada en el fragmento de código 6.7 añadiéndole el número de fotograma (`detection['frame_number']`).

6.2.2.3 Implementación del detector DLIB - MMOD

En esta sección se explicará la implementación del Detector DLIB - MMOD (detallado en el estado del arte en el apartado 4.2.2).

[1] *Upsampling*: técnica utilizada en visión por computador que consiste en redimensionar la imagen a mayor tamaño para poder detectar más objetos alargando en consecuencia el tiempo de ejecución y aumentando los recursos dedicados.

En el caso del proyecto, también se ha proporcionado la implementación en el módulo `facedetector` (concretamente, en el fichero `dlib_detector.py` al igual que la implementación del Detector DLIB - HOG).

Por ello, la implementación de este detector tan solo se diferencia en los siguientes factores:

- El detector MMOD de Dlib se inicializa de la siguiente manera:

```
elif detector_type == 'mmod':
    self.detector = dlib.cnn_face_detection_model_v1('src/facedetector/mmod_human_face_detector.dat')
```

Fragmento de código 6.14: Creación del detector MMOD de Dlib [Fuente]

- Dados los valores de confianza devueltos por MMOD (ver sección 6.2.1), tras varias pruebas, se propone la siguiente suma ponderada:

$$confidence = 0.7 * conf_{det} + 0.15 * brisque + 0.15 * cont$$

- En el método `detect_face(...)` se realiza la siguiente implementación del detector MMOD:

```
def detect_face(self, image):
    [...]
    elif self.detector_type == 'mmod':
        detection = self.detector(image, 0)
        if detection:
            scores = [detection[0].confidence]
            detection = [detection[0].rect]

    if detection and scores:
        return [detection[0], scores[0]]

    return None
```

Fragmento de código 6.15: Implementación del método `detect_face(self, image)` en Dlib-MMOD [Fuente]

  En este caso, cuando se ejecuta el detector, se le puede indicar, al igual que con el Detector DLIB - HOG si se desea realizar *upsampling* de la imagen. Si se ha detectado el rostro del vídeo, se recoge tanto su confianza (`scores[0]`) como su cuadro delimitador (`detection[0]`) y se devuelven.

- Para recoger las detecciones de todos los vídeos, se realizaría la misma llamada del cuaderno de Jupyter `main-lift.ipynb` mostrada en el fragmento de código 6.13 cambiando el valor del parámetro `detector_type = 'hog'` a `'mmod'`. Los resultados, al igual que con el Detector DLIB - HOG, se almacenan en ficheros `.pickle` con la estructura del fragmento de código 6.7 añadiéndole también el número de fotograma detectado (`detection['frame_number']`).

6.2.2.4 Extracción de elementos faciales

En el caso de ambos detectores de Dlib y a diferencia de MTCNN, no devuelven por defecto algún punto facial de referencia (como los ojos, nariz o boca). Por este motivo, si se quiere obtener tales referencias es necesario hacer uso de *68 Face Landmarks*, el cual es un modelo de detección de 68 puntos faciales llamado `shape_predictor_68_face_landmarks.dat.bz2` disponible en el repositorio de modelos [22] de Dlib.

El detector de puntos faciales se basa en la idea [81] de usar un conjunto de árboles de regresión que permitan estimar las posiciones de los elementos faciales desde un subconjunto de intensidades de píxeles para tratar de lograr predicciones en tiempo real. El *framework* que se propone hace uso de gradientes para entrenar dichos árboles de regresión. En concreto, el algoritmo que se realiza, de forma resumida, es el siguiente [82]:

1. Se recoge como entrada del entrenamiento imágenes etiquetadas manualmente con las coordenadas de las regiones de cada estructura facial a detectar.
2. En el entrenamiento, el conjunto de árboles estiman las posiciones de los puntos faciales (sabiendo la distancia que guardan entre ellos) haciendo uso de las intensidades de los píxeles mencionados previamente.
3. En el caso de este detector de puntos faciales (*68 Face Landmarks*), se utilizó la plantilla que se muestra en la figura 6.10 para definir las coordenadas y, por tanto, los puntos a detectar:

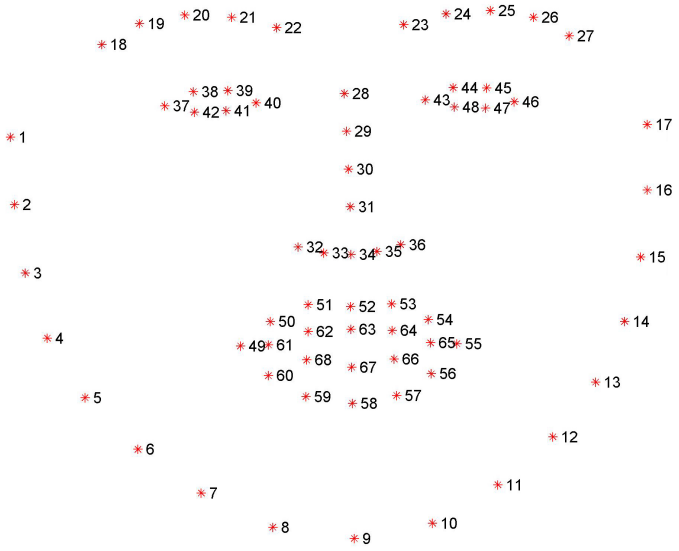


Figura 6.10: 68 puntos faciales a detectar con *68 Face Landmarks* [82]

Para el proceso de entrenamiento se utilizó el *iBUG 300-W face landmark dataset* [83].

Es importante destacar el hecho de que existen más tipos de detectores de puntos faciales, por ejemplo el *5 face landmarks* (de Dlib también). El hecho de escoger el detector de 68 puntos faciales es debido a desear obtener la misma calidad en los puntos que MTCNN.

En el proyecto, *68 Face Landmarks* se encuentra presente en el módulo `facedetector` (concretamente en el fichero `dlib_detector.py`). Concretamente, el modelo se inicializa en el constructor (al igual que se realizó con los detectores):

```
self.predictor68 = dlib.shape_predictor("src/facedetector/shape_predictor_68_face_landmarks.dat")
```

Fragmento de código 6.16: Inicialización de *68 Face Landmarks* [Fuente]

Tras realizar una detección y devolver su correspondiente cuadro delimitador, se llama al método `get_facial_landmarks(...)`, el cual se encarga de devolver 5 coordenadas coincidentes con las devuelve MTCNN (ojo izquierdo, ojo derecho, nariz y parte izquierda y derecha de la boca):

```
def get_facial_landmarks(self, pixels_array, face):
    points = self.predictor68(pixels_array, face)

    if points is not None:
        return [(points.part(49).x, points.part(49).y),
                (points.part(55).x, points.part(55).y),
                (points.part(34).x, points.part(34).y),
                (points.part(38).x, points.part(38).y),
                (points.part(45).x, points.part(45).y)]

    return None
```

Fragmento de código 6.17: Detección de puntos faciales [Fuente]

Nótese en el fragmento de código 6.17 cómo, tras recoger la matriz de píxeles de la imagen original `pixels_array` y el cuadro delimitador `face`, se ejecuta el detector para devolver las 5 coordenadas pedidas si ha conseguido lograr los puntos. La figura 6.10 indica qué punto se está devolviendo exactamente.

A continuación, y con la finalidad de devolver exactamente la misma salida que realiza el objeto `MTCNNDetector`, se incluyen los puntos en el diccionario de salida y, en consecuencia, en el `.pickle` generado.

```
points = self.get_facial_landmarks(pixels_array, detection_box)

if points is not None:
    detection['keypoints'] = {'nose': points[2],
                              'mouth_right': points[1],
                              'right_eye': points[4],
                              'left_eye': points[3],
                              'mouth_left': points[0]}
```

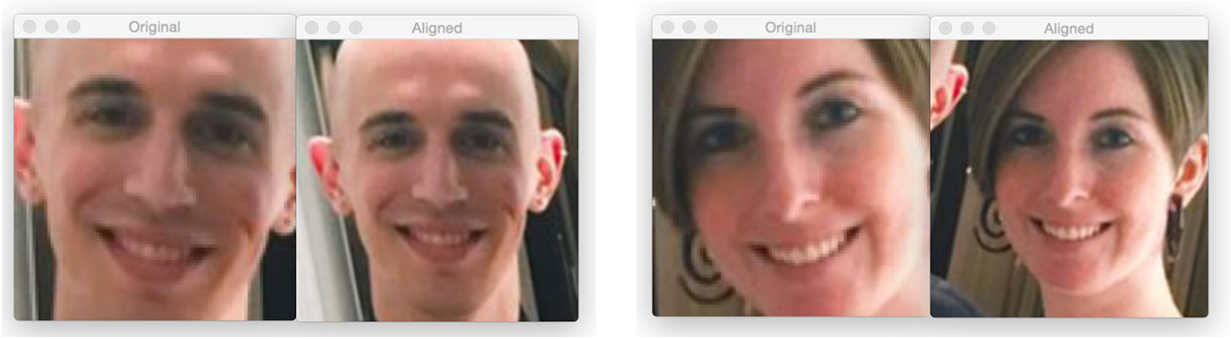
Fragmento de código 6.18: Inclusión de puntos faciales en el diccionario de salida [Fuente]

Si no se detectan puntos faciales, se considera que la detección original no tenía la suficiente calidad que se requiere en el sistema y se desecha.

6.2.3 Normalización de la detección

En la gran mayoría de sistemas de verificación que se han desarrollado a lo largo de los últimos años se lleva a cabo un proceso de normalización (también llamado alineamiento) en la detección. Este proceso tiene como objetivo que un rostro que lleva consigo una cierta inclinación se normalice, rectifique o enderece hasta que se encuentre centrado.

Dicha normalización, en casos generales, conlleva una mejora en el sistema de verificación debido a que todos los rostros que se vectorizan en la Generación de descriptores poseen la misma orientación (ver ejemplo de la figura 6.11). Se proponen a continuación dos técnicas de alineamiento y una de redimensión usadas en el proyecto.



(a) Ejemplo 1 (b) Ejemplo 2

Figura 6.11: Ejemplo de normalización/alineamiento de caras [84]

6.2.3.1 Normalización y recorte estático

La primera técnica que se propone es la normalización con un recorte estático del resultado. Para lograr realizar dicha normalización, como se mencionó en las bibliotecas usadas en el Detector MTCNN, se usará las utilidades otorgadas por uno de los tutores del presente proyecto, Modesto Fernando Castrillón Santana (basadas en una implementación de C++ [85]).

Dichas utilidades se encuentran en un fichero llamado `FaceNormalizationUtils`, en el cual se encuentra el objeto `Normalization`, que se inicializa siempre y cuando se haya seleccionado la opción que permite dicho alineamiento:

```
if self.normalize_face == 'basic_double_detection' or self.normalize_face == 'basic_cut':
            self.normalizatorHS = faceutils.Normalization()
```

Fragmento de código 6.19: Inicialización de módulo de normalización de caras [Fuente]

Si la opción se encuentra activa y se pretende realizar un recorte estático a continuación (`normalize_face = 'basic_cut'`), se ejecutará un proceso que se puede dividir en dos:

1. Uso de las utilidades para alinear el rostro.

- En primer lugar, se divide la imagen haciendo uso de la biblioteca de OpenCV para dividir la imagen a color en tres matrices separadas BGR (se recuerda que OpenCV usa BGR como espacio de colores en lugar de RGB).
- A continuación, se alinea cada una de las imágenes "grises" (formada por las matrices de píxeles separadas) haciendo uso de las coordenadas de los ojos.
  - Se recoge un cierto margen proporcional a la distancia entre los ojos respecto a la imagen original. Esto permitirá que tras rotar la imagen se reduzca la probabilidad de tratar con márgenes en negro. Evidentemente, y aunque no sea una situación muy común en *AveRobot*, si el rostro se encuentra en uno de los bordes de la imagen original, los márgenes negros también se mostrarán al rotar la imagen.
  - Se utiliza el ángulo arcotangente entre ambos ojos para el cálculo de la inclinación que posee el rostro.
  - A continuación, se realiza una rotación de la matriz bidimensional que permite alinear la cara respecto al ángulo de inclinación calculado previamente haciendo uso de funciones proporcionadas por OpenCV.
  - Se produce la transformación necesaria para asignar como nuevos puntos de referencia la nueva posición de los ojos en la imagen, es decir, las coordenadas $(66, 62)$ y $(92, 62)$ para el ojo izquierdo y derecho respectivamente. Tras ello, se recorta la imagen con el rostro alineado a unas dimensiones de 159 x 155.
- Se fusionan las tres matrices de píxeles para formar de nuevo la imagen a color alineada. Como se puede observar en la figura 6.12, existe una pérdida de definición de la imagen tras el proceso de alineamiento.
  - La imagen alineada se guarda con el nombre `n-<frame>.jpg` en la misma salida que el módulo de generación de frames, con el objetivo de recortar dicha imagen a la hora de proceder a la Generación de descriptores en lugar de la original, obteniendo el rostro alineado.

(a) Ejemplo 1 sin alineamiento

(b) Ejemplo 1 tras alineamiento

(c) Ejemplo 2 sin alineamiento

(d) Ejemplo 2 tras alineamiento

Figura 6.12: Transición de la identidad 5 desde el fotograma original hasta el alineamiento básico

2. Recorte o *cropping* estático.

    - En este caso, al tratarse de un recorte estático, se presupone que la cara se encontrará siempre desde el píxel (35,39) de la imagen alineada con una anchura y altura de 80 (ver figura 6.13).

(a) Ejemplo 1 tras alineamiento

(b) Ejemplo 1 tras recorte estático

(c) Ejemplo 2 tras alineamiento

(d) Ejemplo 2 tras recorte estático

Figura 6.13: Transición de la identidad 5 desde el alineamiento básico hasta el recorte estático

Se puede encontrar en el fichero `mtcnn_detector.py` del módulo `facedetector` la correspondiente implementación en el sistema:

```
for index, detection in enumerate(detection_list):
    [...]
    img = numpy.asarray(Image.open(directory + detection['frame_number']))
    b, g, r = cv2.split(img)

    [...] # Se normalizan cada una de las matrices por separado

    norm_bgr = cv2.merge((b_norm, g_norm, r_norm))
    [...]
    elif self.normalize_face == 'basic_cut':
        detection['box'] = [35, 39, 80, 80]
        detection['frame_number'] = "n-" + detection['frame_number']
        normalized_detection_list.append(detection)
```

Fragmento de código 6.20: Implementación de normalización con recorte estático [Fuente]

6.2.3.2 Normalización y doble detección

En el caso de la técnica de normalización y doble detección, se propone cambiar el paso 2 del proceso explicado en la sección 6.2.3.1. En lugar de realizar un recorte estático en la imagen presuponiendo que el rostro se encuentra en un punto en concreto de la imagen alineada siempre, se realiza otra detección. En el caso de MTCNN, el resultado es el que se muestra en la figura 6.14:

(a) Ejemplo 1 tras alineamiento

(b) Ejemplo 1 tras nueva detección

(c) Ejemplo 2 tras alineamiento

(d) Ejemplo 2 tras nueva detección

Figura 6.14: Transición de la identidad 5 desde el alineamiento básico hasta la nueva detección

Se puede observar en la figura 6.14 cómo la imagen resultante es más cercana al rostro que la salida que se muestra con el recorte estático en la figura 6.13.

En este caso, la implementación de una nueva detección en el sistema tras realizar la normalización se realizaría de la siguiente manera:

```
if self.normalize_face == 'basic_double_detection':
    new_detection = self.detect_face(norm_bgr)

    if new_detection is not None:
        new_detection['frame_number'] = "n-" + detection['frame_number']
        normalized_detection_list.append(new_detection)
```

Fragmento de código 6.21: Implementación de la doble detección tras la normalización [Fuente]

En el caso de que no sea posible realizar la nueva detección, el fotograma se desecha por falta de suficiente calidad.

6.2.3.3 Redimensionamiento de la detección

La última técnica que se propone es redimensionar la detección final de los detectores (la cual muestra una proporción facial parecida a la figura 6.14b o a la figura 6.14d) para adaptarla a una imagen más común para los modelos de Generación de descriptores sin alinearla previamente.

Para realizar dicho redimensionamiento, se recogen los puntos faciales y se calcula la distancia entre los ambos ojos en la coordenada X y la distancia entre los ojos y la boca en la coordenada Y. Tras el cálculo de dichas distancias, el cuadro delimitador:

- Se ampliará en su anchura (dimensión horizontal) proporcionalmente a la distancia entre los ojos ($Dist_x$) en la coordenada X.
- Se ampliará en su altura (dimensión vertical) proporcionalmente a la distancia entre los ojos y la boca ($Dist_y$) en la coordenada Y.

Los detectores usados no proporcionan un tamaño fijo en la detección que devuelven. Teniendo en cuenta que la distancia entre ojos y la distancia entre los ojos y la boca también es variable, el cuadro delimitador resultante del redimensionamiento de la detección no posee unas dimensiones fijas ($3 * Dist_x$ x $3 * Dist_y$), pero sí una proporción similar a otros cuadros delimitadores (dependerá de las características del rostro de la persona detectada en sí). Como con el resto de detecciones, se cambiará el tamaño de la detección en función de lo que exija el modelo usado en la Generación de descriptores.

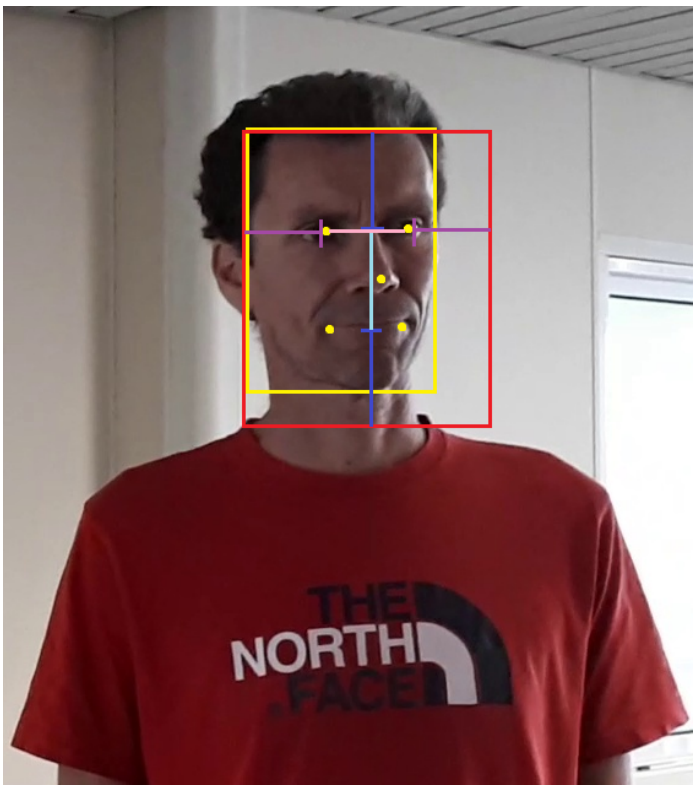


Figura 6.15: Ejemplo de redimensionamiento de la detección en la identidad 5

En la figura 6.15, la estructura coloreada de color amarillo representa la salida del detector (en este caso, MTCNN). Al aplicar la redimensión mencionada, teniendo en cuenta la distancia entre ojos (representada con la línea de color rosa) y la distancia entre los ojos y la boca (representada por el color azul celeste), se amplía el cuadro delimitador en su anchura y altura hasta lograr obtener las dimensiones mostradas por el rectángulo rojo.

6.2.4 Selección de muestras

En esta sección se describirá cómo se realiza la selección de muestras, explicando primero cómo se puede ajustar un número máximo de detecciones adecuado por vídeo para a continuación tratar de recoger las mejores teniendo en cuenta su distancia temporal.

6.2.4.1 Selección del número máximo de detecciones recogidas por vídeo

Según se ha mostrado en la inicialización de ambos detectores (fragmentos de código 6.10 y 6.13), existe un parámetro llamado `max_detections`. Una correcta selección del número de detecciones por vídeo (o en el caso del sistema final por captación) puede resultar determinante en la tasa de acierto del sistema de verificación.

Dicho parámetro, por tanto, representa el número máximo de detecciones que se desean realizar. Existe la posibilidad de que no se llegue al número total deseado por el hecho de que el contenido sea de insuficiente calidad como para llegar a dicha cantidad (aún con las técnicas empleadas para realizar las detecciones como *upsampling* o el Desentrelazado de fotogramas). En el caso de que el sistema sea incapaz de detectar algún rostro, lanzará un error notificando al administrador o al usuario en cuestión.

Se debe tener en cuenta que el número de detecciones máximas del sistema:

- Si es bajo (por ejemplo, 1), no dejará al sistema suficientes muestras para realizar una buena comparativa de dichas detecciones en un escenario multinivel. Por ello, se estaría en una situación donde la elección vendría determinada por una suposición de la buena calidad de las pocas detecciones realizadas y sus correspondientes descriptores.
- Si es alto (por ejemplo, 10), dejará al sistema demasiadas muestras con el objetivo de realizar la comparativa en el mencionado escenario multinivel. En este caso, aumentaría el posible ruido proporcionado por detecciones que pudieran llegar a incorporar consigo una peor calidad por forzar al sistema a recoger más de estas.

En este proyecto, se propone usar una cantidad de 4 detecciones por vídeo siempre que sea posible. No obstante, se observarán los resultados de recoger una única muestra escogida manualmente a raíz de dichas 4 detecciones en los modelos 6 y 7 de la tabla 7.5.

6.2.4.2 Recogida de las mejores detecciones con distancia temporal

El último aspecto que se tratará sobre la selección de muestras será cómo se realiza la recogida de las mejores detecciones. Las detecciones resultantes de un vídeo (o de una captación en el sistema definitivo) deben respetar dos puntos principalmente:

- Deben contener los mayores valores de confianza. Se recuerda que dichos valores de confianza se calculan a raíz de tres parámetros distintos explicados en el apartado 6.2.1. Este algoritmo se aplica después de obtener dicha confianza final en cada detección.
- Deben poseer una cierta distancia temporal entre ellos con el objetivo de evitar fotogramas seguidos que contengan prácticamente la misma pose en el rostro y aporten poca nueva información al sistema. En este proyecto, se planteará que aquellas detecciones que hayan obtenido una confianza muy similar probablemente apenas contengan cambios en sus imágenes, por lo que pueden ser consecutivas o cercanas en el tiempo.

Con el fin de conseguir un equilibrio entre ambos requisitos, se ha desarrollado un algoritmo que trata de recoger las mejores detecciones desde el punto de vista del parámetro de confianza

introducido en cada una de ellas a la par que se realizan saltos descartando aquellas que tengan dicha confianza muy similar. Para lograrlo:

1. Se recoge la confianza máxima de todas las detecciones de los fotogramas del vídeo.
2. Siempre que aún no se haya rellenado la lista de detecciones:
    - Se recogen una serie de intervalos de detecciones previamente ordenadas por su confianza y se separan por un parámetro de rango de confianza `confidence_range`, que marcará entre cuántas detecciones se podrá escoger solo una con el objetivo de no recoger aquellas con una confianza similar y, por tanto, posiblemente cercanas temporalmente.
    - Se procede a recorrer cada una de las series de intervalos de confianza de la detección y se recoge un fotograma aleatorio de cada serie. Todos los fotogramas no recogidos de esa serie pasan a pertenecer a otro conjunto de detecciones no recogidas.
3. Si se ha rellenado la lista de detecciones teniendo en cuenta que su longitud deseada es `max_detections` (ver apartado 6.2.4.1), se devuelve. Si no es así, se recogen las detecciones que faltan para llegar a esta cantidad del conjunto de detecciones no recogidas ordenadas por confianza.

Con este algoritmo se consiguen las mejores detecciones con una cierta distancia temporal que permite introducir información novedosa sobre las personas en el sistema. La implementación del algoritmo en Python se puede observar en el método `prepare_results_list(...)` de los detectores del módulo `facedetector`:

```
def prepare_results_list(self, detection_list, current_max_confidence):
    new_detection_list = list()
    not_chosen_detections = list()
    confidence = current_max_confidence
    while len(new_detection_list) < self.max_detections and confidence > 0:
        detection_set = [detection for detection in detection_list
                         if confidence - self.confidence_range <= detection['confidence'] <= confidence]

        if detection_set:
            new_detection_list.append(detection_set.pop(random.randrange(len(detection_set))))
            not_chosen_detections.extend(detection_set)

        confidence -= self.confidence_range

    if len(new_detection_list) < self.max_detections:
        new_detection_list.extend(not_chosen_detections[:self.max_detections - len(new_detection_list)])

    return new_detection_list
```

Fragmento de código 6.22: Algoritmo de recogida de las mejores detecciones con distancia temporal [Fuente]

Finalmente, la distribución de confianzas resultantes en los tres detectores tiene una desviación típica distinta. Por este motivo, tras varias pruebas, se recomienda que la serie de intervalos de detecciones esté sometida a un `confidence_range` con los siguientes valores:

$$MTCNN = 0.00001 \qquad DLIB_{HOG} = 0.003 \qquad DLIB_{MMOD} = 0.002$$

6.3 Generación de descriptores

Una vez se tienen las detecciones, se deben generar los descriptores de las características faciales de la persona (*face embedding*). El objetivo de la obtención de descriptores es utilizarlos en métodos de comparación entre estos con la finalidad de determinar si provienen o no de la misma persona en el sistema de verificación.

Para la generación de descriptores, se propone por un lado del uso de FaceNet (Inception ResNet v1) y por otro VGGFace2 - ResNet50.

6.3.1 Implementación de FaceNet (Inception ResNet v1)

En esta sección se explicará la implementación de FaceNet (Inception ResNet v1) (explicado en el estado del arte en la sección 4.3).

En el generador de descriptores se hará uso por primera vez de las bibliotecas `keras` [17] (ver sección 3.1.2.2) y `scikit-learn` [86].

A la hora de inicializar el modelo basado en FaceNet, se hace uso del módulo `model`. Concretamente, en el fichero `facenet_keras_model.py`, se puede observar la inicialización del objeto que generará los descriptores.

```
def __init__(self, norm_type='l2', embedding_type='None', pickle_results=True,
             csv_results=False, print_model_info=False, force_update=False):
```

Fragmento de código 6.23: Inicialización del modelo FaceNet en el sistema de verificación [Fuente]

Respecto a las funciones y métodos del objeto `FacenetKerasModel`, se tiene:

Función	Descripción
`load_pickle_results(pickle_directory, embedding_type='None')`	Carga un fichero de salida del módulo `facedetector`. El parámetro `embedding_type` indica qué tipo de detección se quiere cargar.
`save_pickle_results(pickle_data, pickle_directory, embedding_type='None')`	Guarda los descriptores en un fichero `.pickle`.
`save_csv_results(pickle_data, pickle_directory, embedding_type='None')`	Guarda los descriptores en un fichero `.csv`.

Tabla 6.5: Funciones presentes en el fichero `facenet_keras_model.py`

Método	Descripción
`__init__(self, norm_type='l2', embedding_type='None', pickle_results=True, csv_results=False, print_model_info=False, force_update=True)`	Inicializa el objeto `FacenetKerasModel` personalizado a una serie de parámetros.
`get_model(self)`	Carga el modelo del objeto. En este caso, la implementación en Keras de FaceNet basada en Inception ResNet v1.
`predict_data(self, frames_folder_name, face_detection_results)`	Dadas unas detecciones de unos fotogramas, comienza a generar descriptores de los rostros. Si se ha indicado al construir el objeto, también normaliza el resultado y lo guarda de forma persistente en un fichero `.pickle` o `.csv`.
`@staticmethod prepare_face(face, directory)`	Devuelve la matriz de píxeles de una detección desde un fotograma. En el caso de FaceNet, es necesario dividir todos los píxeles entre 255 para que los valores de estos se encuentren en el intervalo [0,1]

Tabla 6.6: Métodos presentes en el fichero `facenet_keras_model.py`

Respecto a la implementación de FaceNet en el sistema de verificación, es necesario destacar varios detalles. En primer lugar, y como se mencionó anteriormente, la carga del modelo pre-entrenado se realiza utilizando Keras [17]:

```
model = load_model('src/model/facenet_keras.h5', compile=False)
```

Fragmento de código 6.24: Carga del modelo FaceNet usando la biblioteca `keras` [Fuente]

Además, es necesario destacar, como se mencionó en la explicación de los métodos (ver tabla 6.6), que el modelo está preparado para recibir imágenes a color en formato RGB con valores de la matriz de píxeles entre 0 y 1. Por este motivo, en el método `prepare_face(...)`, es necesario tomar en consideración este hecho con el objetivo de conseguir una entrada de dimensiones 160x160x3:

```
return numpy.divide(numpy.asarray(Image.fromarray(face_matrix)
            .resize((160, 160))), 255).reshape((1, 160, 160, 3))
```

Fragmento de código 6.25: Adaptación de la imagen de entrada al modelo FaceNet pre-entrenado [Fuente]

Por último, y como es común en el uso de redes neuronales, se ejecuta el `predict(input)` del modelo con la entrada ya preprocesada y apropiada:

```
vectorized_faces = self.model.predict(prepared_faces_matrix)
```

Fragmento de código 6.26: Predicción para obtener descriptores con ambos modelos propuestos

Tras este proceso, y dependiendo del parámetro `norm_type` mencionado en el fragmento de código 6.23, se puede realizar una normalización L1 o L2 (encontrándose esta última por defecto).

Por último, se guarda el resultado en formato `.csv` y en formato `.pickle` si así se ha personalizado el objeto creado, generándose un contenido persistente que contiene una lista donde cada miembro es un vector de 128 dimensiones, siendo el descriptor buscado. En el caso de este proyecto, cada *item* de la lista del fichero generado se trata del descriptor de cada una de las detecciones guardadas según el criterio usado en el proceso de detección de los rostros (ver apartado 6.2).

Por ejemplo, y como se puede visualizar en el apartado 2.5.1 del fichero del repositorio [30] `main-lift.ipynb`, se pueden generar los descriptores de la siguiente forma:

```
fkm = facenet_keras_model.FacenetKerasModel(norm_type='l2', embedding_type='resize',
                                            pickle_results=True, csv_results=True,
                                            print_model_info=True, force_update=True)

for video in train_videos_out:
    data = fkm.predict_data(video, facenet_keras_model.load_pickle_results(video,
                                embedding_type='resize'))
```

Fragmento de código 6.27: Generación de los descriptores usando el módulo `model` del sistema de verificación con FaceNet

Donde se puede analizar cómo en este caso se crea el generador de descriptores de FaceNet, el cual realiza una normalización L2 sobre unas detecciones con redimensionamiento (ver apartado 6.2.3.3), guardando los resultados en formato `.pickle` y `.csv`, imprimiendo por pantalla información del modelo y forzando la actualización en caso de existir algún resultado para el vídeo que se esté tratando, para a continuación recoger cada uno de los títulos con su nomenclatura específica y realizar una predicción sobre sus detecciones redimensionadas.

6.3.2 Implementación de VGGFace2 - ResNet50

En este apartado se explicará la implementación de VGGFace2 - ResNet50 (explicado en el estado del arte en la sección 4.4).

Como se puede analizar en el módulo `model` del repositorio del proyecto [30] en el interior del fichero `vgg_face_model.py`, la estructura del fichero es muy similar a la explicada perteneciente a FaceNet (Inception ResNet v1). Por este motivo, no se volverá a explicar.

Gracias a la siguiente línea de código, se puede acceder por medio de la biblioteca `keras-vggface` al modelo entrenado de ResNet-50, donde se muestra cómo la última capa *fully_connected* no se carga mediante el parámetro `include_top` para obtener el descriptor y que las dimensiones de entrada son 224x224x3:

```
model = VGGFace(model='resnet50', include_top=False, input_shape=(224, 224, 3), pooling='avg')
```

Fragmento de código 6.28: Carga del modelo ResNet-50 por medio de la biblioteca `keras-vggface`

Tras cargar una lista de los tensores de entrada que contienen los píxeles de los rostros detectados, se procede a llamar a la siguiente función:

```
prepared_faces_matrix = preprocess_input(prepared_faces_matrix, version=2)
```

Fragmento de código 6.29: Preprocesado de la entrada de ResNet-50

La función `preprocess_input(input, version)` es la encargada de hacer un preprocesamiento automático de la entrada para adecuarla a ResNet-50. Por este motivo no es necesario, al contrario que se realizó con FaceNet (Inception ResNet v1), dividir la matriz de píxeles entre 255 para obtener valores entre 0 y 1 (siendo este rango el generalmente utilizado a la hora de tratar con matrices de píxeles procedentes de imágenes). Se recuerda que la versión 2 (parámetro `version`) se debe escoger si se está tratando con este modelo ResNet-50, debido a que se hace uso de VGGFace2.

El resto del proceso (desde la predicción para obtener los descriptores hasta el guardado persistente de estos en ficheros) coincide con el realizado con FaceNet (Inception ResNet v1), con lo que no se volverá a exponer. En el fragmento de código 6.30, se puede visualizar cómo se comenzarían a generar descriptores usando el presente modelo en comparación a cómo se generaba con FaceNet (Inception ResNet v1) en el fragmento de código 6.27.

```
vfm = vgg_face_model.VGGFaceModel(norm_type='l2', embedding_type='resize',
                                  pickle_results=True, csv_results=True,
                                  print_model_info=True, force_update=True)

for video in test_videos_out:
    data = vfm.predict_data(video, vgg_face_model.load_pickle_results(video,
                            embedding_type='resize'))
```

Fragmento de código 6.30: Generación de los descriptores usando el módulo `model` del sistema de verificación con ResNet-50

6.3.3 Normalización

Tras generar los descriptores (tanto con FaceNet como con VGGFace2) es necesario realizar un proceso de normalización de sus valores en los vectores con la finalidad de que su distribución sea más regular.

Por este motivo, el parámetro `norm_type` permitirá elegir si se desea no realizar ningún tipo de normalización (`'None'`), una normalización L1 (`'l1'`) o una normalización L2 (`'l2'`).

6.3.3.1 L1

La normalización L1 sigue la siguiente fórmula: $L_1(X) = \dfrac{X}{||X||_1}$

Por ello, el vector normalizado resultante será el resultado de recoger cada elemento del vector original para a continuación dividirlo entre la suma de todos los elementos del vector.

Con esto se logra que los valores resultantes se encuentren en un intervalo de [-1, 1], lo que facilita el tratamiento y mejora la precisión de distancias en las siguientes fases del sistema.

6.3.3.2 L2

Es la norma más utilizada en el tratamiento de vectores en el área de la inteligencia artificial y la que se usará en los prototipos del sistema de verificación.

La normalización L2 sigue la siguiente fórmula: $L_2(X) = \frac{X}{||X||_2}$

En este caso, el vector normalizado resultante será el resultado de recoger cada elemento que forma el vector original para dividirlo entre el módulo de dicho vector.

De esta manera, también se obtienen unos valores resultantes en un intervalo de [-1, 1], con lo que se logra además una mejora en el tratamiento de los datos y en la precisión de las distancias usadas en el resto del proyecto.

En concreto, y por su forma de calcularse, L2 tiende a generalizar mejor ya que distribuye la importancia de cada una de las características internas del vector de una manera más efectiva (siendo sensible a *outliers* o valores atípicos) que L1.

```
if self.norm_type == 'l2':
    vectorized_faces = normalize(vectorized_faces, norm='l2')
elif self.norm_type == 'l1':
    vectorized_faces = normalize(vectorized_faces, norm='l1')
```

Fragmento de código 6.31: Normalización de descriptores en el sistema de verificación

6.4 Construcción del *DataFrame* con los datos relevantes para la verificación

Una vez se ha finalizado de extraer los descriptores de las detecciones, y con ello, la información que se va a comparar con el objetivo de verificar en el sistema planteado, se hace necesario realizar una estructura de datos que almacene toda la información.

En el caso de este proyecto, dicha estructura de datos debería contener los descriptores logrados por cada uno de los fotogramas tratados, además de otros metadatos que mostraran una idea de la procedencia de dichos fotogramas (en el caso del escenario multinivel, un dato sustancial sería el piso) o la ID asignada a la persona que se está tratando.

Se propone, por tanto, la siguiente organización en los datos a tratar:

VídeoID/CaptaciónID	ID	Piso	Descriptor-NDimensional
averobot_floor_01-03_01_02_01_02/	3	1	Descriptor-detección-1
averobot_floor_01-03_01_02_01_02/	3	1	Descriptor-detección-2
averobot_floor_01-03_01_02_01_02/	3	1	Descriptor-detección-3
averobot_floor_01-03_01_02_01_02/	3	1	Descriptor-detección-4
...			
averobot_floor_03-85_03_02_08_08/	85	3	Descriptor-detección-1
averobot_floor_03-85_03_02_08_08/	85	3	Descriptor-detección-2
averobot_floor_03-85_03_02_08_08/	85	3	Descriptor-detección-3
averobot_floor_03-85_03_02_08_08/	85	3	Descriptor-detección-4

Tabla 6.7: Ejemplo de estructura de datos que se desea construir en el sistema de verificación

Donde la columna:

- **VídeoID/CaptaciónID** se corresponde a un identificador del vídeo.
- **ID** es la identidad asignada a la persona física que aparece en el vídeo. En un sistema real, dicha ID sería un valor autogenerado en el primer piso donde se realice la verificación con el objetivo de que en el piso donde se desee identificar si la persona es o no una determinada identidad sea usada.
- **Piso** es la planta donde se ha realizado la grabación.
- **Descriptor-NDimensional** se trata del descriptor procesado en la Generación de descriptores. Sus dimensiones en el caso de este proyecto pueden ser 128 (en caso de haberse utilizado FaceNet (Inception ResNet v1)) o 2048 (si se ha hecho uso de VGGFace2 - ResNet50). Se debe mencionar que dentro de una misma grabación y dependiendo del número de detecciones realizadas y posteriormente procesadas, se tendrán más o menos descriptores por vídeo.

El encargado de realizar esta tarea será el fichero `embeddings_assembler.py` situado en el interior del módulo `model`. La principal biblioteca que se utilizará será `pandas` [87], quien implementa una herramienta de manipulación y de análisis de datos sencilla de usar, flexible y eficiente. Dicha biblioteca introduce una estructura de datos llamada *DataFrame* (o marco de datos) que permitirá adquirir una distribución de los datos parecida a la analizada en la tabla 6.7.

A continuación, se exponen las funciones de dicho fichero:

Función	Descripción
`load_panda_embeddings(pickle_file_name)`	Carga un *DataFrame* con los datos del sistema de verificación.
`load_pickle_results(pickle_directory, embedding_type='None')`	Carga los descriptores generados de una grabación. También devuelve el ID de la persona y el piso.
`save_pickle_results(dataframe, name, embedding_type='None')`	Guarda el *DataFrame* generado en formato `.pickle`.
`save_csv_results(dataframe, name, embedding_type='None')`	Guarda el *DataFrame* generado en formato `.csv`.
`assemble_embeddings(frame_folders_name_list, embedding_type='None', pickle_results=True, csv_results=True, name=None)`	Reúne los descriptores generados de una serie de vídeos según una serie de parámetros.
`create_pandas_df(face_embeddings, videos_ids_list)`	Crea un *DataFrame* de `pandas` con la estructura de la tabla 6.7.

Tabla 6.8: Funciones presentes en el fichero `embeddings_assembler.py`

Tras reunir los datos numéricos que se pueden visualizar en la tabla 6.7, se crea el *DataFrame* haciendo uso de `pandas` con las cabeceras indicando qué metadato se muestra en cada columna. Por último, se inserta el ID del vídeo o de la captación como la primera columna de la estructura de datos.

```
def create_pandas_df(face_embeddings, videos_ids_list):
    header = ['id', 'floor']
    header.extend(list(range(0, face_embeddings.shape[1]-len(header))))
    df = pd.DataFrame(face_embeddings, columns=header)

    df.insert(0, "video_id", videos_ids_list)

    return df
```

Fragmento de código 6.32: Generación del *DataFrame* con los datos relevantes para la verificación

En consecuencia, las dimensiones del *DataFrame* deberán ser:

$$Embeddings_{number} * (3 + Embedding_{length})$$

Por último, se adjunta un ejemplo de la apariencia final del *DataFrame*:

video_id	**id**	**floor**	**0**	**1**	**2**	**...**	**125**	**126**	**127**
averobot_floor_01-03_01_02_01_02/	3.0	1.0	-0.100246	-0.069571	-0.039903		0.037884	-0.058198	0.016351
averobot_floor_01-03_01_02_01_02/	3.0	1.0	-0.091518	-0.020731	-0.067700		0.012988	-0.053477	0.030616
averobot_floor_01-03_01_02_01_02/	3.0	1.0	-0.078686	-0.007150	-0.050450	...	-0.020420	-0.061162	0.007005
averobot_floor_01-03_01_02_01_02/	3.0	1.0	-0.088945	-0.063194	-0.112910		0.064195	-0.041558	-0.044219
averobot_floor_02-03_02_02_30_05/	3.0	2.0	-0.014646	-0.081424	-0.088336		-0.158368	0.090631	-0.021257
...									
averobot_floor_02-85_02_02_25_05/	85.0	2.0	0.217501	0.009896	0.059669		-0.068217	-0.096064	0.007097
averobot_floor_03-85_03_02_08_08/	85.0	3.0	0.161357	0.047597	0.162354		-0.015687	-0.071712	-0.066290
averobot_floor_03-85_03_02_08_08/	85.0	3.0	0.203430	0.013746	0.029775	...	-0.148488	-0.095411	-0.069836
averobot_floor_03-85_03_02_08_08/	85.0	3.0	0.127501	0.067244	0.048223		-0.069896	-0.028896	-0.057982
averobot_floor_03-85_03_02_08_08/	85.0	3.0	0.163408	0.036488	0.118511		-0.062293	-0.076279	-0.072922

Tabla 6.9: Ejemplo de *DataFrame* con descriptores generados por FaceNet (Inception ResNet v1) usando el Detector DLIB - HOG con Redimensionamiento de la detección en el Subconjunto de vídeos B de entrenamiento

6.5 Distancias euclídeas y coseno del *DataFrame*

Una vez se tiene una estructura de datos ordenados con los que poder realizar la verificación en el presente escenario multinivel, se debe clarificar cómo se realizarán las mediciones de distancias entre los distintos descriptores procedentes de las identidades.

La lógica de la distancia entre descriptores generados es tratar de ser cercanos entre sí en caso de proceder del mismo individuo y lejanos en caso contrario. Dichas distancias permitirán decidir si dos grabaciones contienen o no a la misma persona, siendo este el objetivo del sistema de verificación.

El encargado del cálculo y carga de distancias euclídeas y coseno en los subconjuntos de vídeos a través de los *DataFrames* formados será el módulo `distancematrixcalculator`, presente en el repositorio del proyecto [30].

6.5.1 Uso de distancias euclídeas

La primera distancia que se empleará será la distancia euclídea o euclidiana.

Como se mencionó previamente, el módulo `distancematrixcalculator` tendrá el objetivo de calcular las distancias (tanto euclídeas como coseno) entre los distintos descriptores computados. El módulo se compone de dos ficheros principales:

Fichero	Descripción
`distances_matrix_calculator.py`	Contiene funciones para el cálculo de distancias euclídeas o coseno usando descriptores procedentes de un par de vídeos distintos o del mismo vídeo.
`distances_matrix_loader.py`	Contiene funciones para la carga de listas con distancias euclídeas o coseno calculadas con las funciones de `distances_matrix_calculator.py` usando un *DataFrame* de pandas originadas en el mismo vídeo, en pares de vídeos con la misma identidad y en pares de vídeos con distinta identidad.

Tabla 6.10: Ficheros presentes en el módulo `distancematrixcalculator`

En este módulo, además, se incorpora una nueva biblioteca llamada `deprecated` [88], la cual permite haciendo uso del decorador `@deprecated` marcar determinadas clases, funciones o métodos como obsoletas.

Concretamente, en el fichero `distances_matrix_calculator.py` se tienen las siguientes funciones:

Función	Descripción
`get_different_video_range(embedding_matrix_1, embedding_matrix_2, distance_type='euclidean', min_percentile=10, max_percentile=90)`	Devuelve una matriz de distancias euclídeas o coseno procedentes de los descriptores de dos vídeos distintos, además de los dos percentiles que se indiquen de dicha matriz.
`get_same_video_range(embedding_matrix, distance_type='euclidean', min_percentile=10, max_percentile=90)`	Devuelve una matriz de distancias euclídeas o coseno originada de comparar los descriptores de un mismo vídeo entre sí, además de los dos percentiles que se indiquen de dicha matriz.

Tabla 6.11: Funciones presentes en el fichero `distances_matrix_calculator.py`

Mientras en el fichero `distances_matrix_loader.py` se tienen las siguientes:

Función	Descripción
get_same_video_distance_matrices(videos_embeddings, distance_type='euclidean')	Obtiene una lista de matrices de distancias euclídeas o coseno de descriptores junto con sus percentiles por defecto (10 y 90) procedentes de todos los vídeos con ellos mismos.
get_same_person_distances_matrices(videos_embeddings, distance_type='euclidean')	Obtiene una lista de matrices de distancias euclídeas o coseno de descriptores junto con sus percentiles por defecto (10 y 90) procedentes de todos los pares de vídeos con la misma identidad.
get_different_people_distance_matrices(videos_embeddings, distance_type='euclidean')	Obtiene una lista de matrices de distancias euclídeas o coseno de descriptores junto con sus percentiles por defecto (10 y 90) procedentes de todos los pares de vídeos con distinta identidad.

Tabla 6.12: Funciones presentes en el fichero `distances_matrix_loader.py`

Respecto a la implementación interna del cálculo de distancias euclídeas, se ha usado la biblioteca `sklearn` de nuevo:

```
def get_different_video_range(embedding_matrix_1, embedding_matrix_2,
                              distance_type='euclidean',
                              min_percentile=10, max_percentile=90):
    [...]
    if distance_type == 'euclidean':
        distance_matrix = euclidean_distances(embedding_matrix_1, embedding_matrix_2)
    [...]

    return [numpy.percentile(distance_matrix, min_percentile, interpolation='nearest'),
            numpy.percentile(distance_matrix, max_percentile, interpolation='nearest'),
            distance_matrix.reshape(-1)]
```

Fragmento de código 6.33: Cálculo de distancias euclídeas y devolución de sus percentiles y su matriz resultante [Fuente]

6.5.2 Uso de distancias coseno

La segunda distancia que se propone es la denominada distancia coseno.

En general, y como se observará a lo largo de la sección 7.1, las distancias coseno suelen contener valores menores a los generados por las distancias euclídeas.

Respecto a la implementación, se hace uso de los mismos ficheros (ver tabla 6.10) y las mismas funciones (ver tablas 6.11 y 6.12). Para adquirir la distancia coseno en lugar de la distancia euclídea tan solo se debe cambiar el valor del parámetro `distance_type` a `'cosine'` en lugar de `'euclidean'`.

Respecto a la implementación interna, también se hace uso de la biblioteca `sklearn`:

```
elif distance_type == 'cosine':
    distance_matrix = cosine_distances(embedding_matrix_1, embedding_matrix_2)
```

Fragmento de código 6.34: Cálculo de distancias coseno [Fuente]

Una vez se tiene una serie de funciones que permiten obtener las distancias euclídeas y coseno desde un determinado *DataFrame* que se actualiza con todos los datos indispensables para el sistema de verificación, se puede comenzar a realizar varios prototipos con su respectiva evaluación.

En consecuencia, se entra en la fase final del proyecto: en los Prototipos y evaluación.

Capítulo 7

Prototipos y evaluación

Una vez se han obtenido los descriptores (organizándolos en un *DataFrame* similar al mostrado en la tabla 6.7) y sus distancias, se debe comenzar la última fase del proyecto: la construcción de diversos prototipos y la evaluación de sus resultados. En concreto, en este capítulo se estudiará el diseño de sistemas basados en umbrales y en redes neuronales, con su correspondiente valoración. Por último, se realizará un breve resumen con los resultados de los mejores sistemas construidos.

7.1 Verificación mediante umbral y evaluación

El primer tipo de sistema de verificación en escenarios multinivel será el basado en un umbral. En primer lugar, se explicará en qué consiste dicho umbral para, a continuación, analizar las técnicas de evaluación del sistema que se plantearán para cada uno de los prototipos con el objetivo de presentarlos finalmente.

7.1.1 Verificación basada en umbral de distancia

El tipo de verificación que se realizará en el presente apartado de desarrollo se basará en un umbral. Dicho umbral es un valor numérico que trata de separar o dividir las distancias entre dos descriptores (ya sea euclídea o coseno (ver apartado 6.5)) en dos intervalos con la finalidad de clasificar si se tratan o no de la misma persona.

Tomando en consideración que a menor distancia entre estos, mayor probabilidad existe de que se traten de la misma identidad y que a mayor distancia, menor probabilidad de que se traten de la misma persona, la verificación de identidad basada en umbral trata de calcular un valor para dicho umbral a lo largo de todo el conjunto de datos que logre minimizar el error en el sistema (tanto para falsos rechazos como para falsas aceptaciones) tras la separación mencionada.

Por ejemplo, si se recogieran 4 fotogramas por cada vídeo (ver sección 6.2.4.1), se podría formar una matriz de distancias (ya sea euclídeas o coseno) como la que se muestra a continuación:

		Vídeo B			
		Descriptor 1	**Descriptor 2**	**Descriptor 3**	**Descriptor 4**
Vídeo A	**Descriptor 1**	1.054	1.122	1.321	0.867
	Descriptor 2	0.840	0.995	1.011	0.998
	Descriptor 3	1.432	1.322	1.276	1.114
	Descriptor 4	0.890	0.854	1.075	0.942

Tabla 7.1: Ejemplo de matriz de distancias euclídeas

Si, a continuación, se fuera ajustando el umbral explicado para clasificar cuándo se considera que dos descriptores pertenecen a un mismo individuo o no, se tendría una serie de distancias que acreditarían una clasificación u otra. Además, si se estableciera por ejemplo que con un mínimo de 50 % de distancias euclídeas por debajo del umbral se puede afirmar que ambos vídeos contienen el mismo individuo, se tendrían los siguientes resultados (las distancias en verde se encuentran por debajo del umbral definido y las distancias en rojo se encuentran por encima):

	Descriptor 1	**Descriptor 2**	**Descriptor 3**	**Descriptor 4**
Descriptor 1	1.054	1.122	1.321	0.867
Descriptor 2	0.840	0.995	1.011	0.998
Descriptor 3	1.432	1.322	1.276	1.114
Descriptor 4	0.890	0.854	1.075	0.942

(a) Umbral: 1.000

	Descriptor 1	**Descriptor 2**	**Descriptor 3**	**Descriptor 4**
Descriptor 1	1.054	1.122	1.321	0.867
Descriptor 2	0.840	0.995	1.011	0.998
Descriptor 3	1.432	1.322	1.276	1.114
Descriptor 4	0.890	0.854	1.075	0.942

(b) Umbral: 1.100

Tabla 7.2: Evolución de proporción de valores por debajo del umbral tras incrementarlo

En este ejemplo, en la tabla 7.2a, se tiene un umbral con valor 1.000, dejando una proporción de un 43.75 % de las distancias por debajo de dicho umbral. Según la decisión argumentada previamente, al no igualar o superar el 50 % de los valores de la matriz, la decisión final del sistema de verificación para este par de vídeos sería negativa (es decir, los vídeos no contienen a la misma persona).

Si, en cambio y como se muestra en la tabla 7.2b, se tiene un umbral con valor 1.100, dejando una proporción de un 62.5 % de valores por debajo de dicho umbral, al superar al 50 % mencionado, el sistema de verificación daría una respuesta positiva para este par de vídeos (es decir, los vídeos contienen a la misma persona).

Por ello, en un sistema que desarrolle la verificación haciendo uso de un umbral fijo se tendrán que optimizar los siguientes dos parámetros:

- Umbral: este parámetro se optimizará cuando el error producido por pares de vídeos con la misma identidad y pares de vídeos con distinta identidad sea el mismo. Dicho error, llamado EER (*Equal Error Rate*), será explicado en la sección 7.1.2.2.
- Confianza: se corresponde al 50 % del ejemplo. Aunque dicho 50 % pueda resultar un valor lógico, se debe buscar una proporción de los valores de la matriz que permita

ajustar lo mejor posible la verificación. Dicha búsqueda se realizará en la sección 7.1.2.3. Además, dicho parámetro permitirá personalizar si se quieren evitar más los falsos negativos o los falsos positivos.

Por último, es necesario mencionar que la confianza es un parámetro exclusivo de la utilización de vídeos y del sistema de votación por mayoría. Una verificación basada en umbral de distancia no tiene por qué tener la capacidad de optimizar dicha confianza (por ejemplo, en caso de que se estuvieran comparando imágenes independientes en lugar de fotogramas de vídeos). En consecuencia, el umbral es el encargado de minimizar el EER para, a continuación, en este caso, ser aplicado en vídeos que disponen de varios fotogramas. En caso de que se escoja un solo fotograma como máximo por vídeo (ver sección 6.2.4.1) mediante `max_detections=1`, el parámetro de confianza dejará de ser práctico evidentemente.

7.1.1.1 Obtención de pares de vídeos para los tests

En primer lugar, y una vez se ha obtenido el *DataFrame* con los datos necesarios para las pruebas (ver sección 6.4), se debe construir una herramienta que permita obtener todos los pares de vídeos disponibles, tanto los que contengan a la misma identidad (también llamados pares de genuinos) como los que no (también llamados pares de impostores). Por este motivo, se introduce el módulo `accuracymeter`.

Dicho módulo contiene un fichero llamado `accuracy_meter.py`, el cual contiene las siguientes tres funciones:

Función	Descripción
`get_same_person_videos_ids_pairs(videos_embeddings)`	Obtiene todos los pares de vídeos con la misma identidad desde el *DataFrame* construido.
`get_different_people_videos_ids_pairs(videos_embeddings)`	Obtiene todos los pares de vídeos con distinta identidad desde el *DataFrame* construido.
`get_accuracy(videos_embeddings, ids_pairs_list, distance_type='euclidean', confidence=0.5, threshold=1.193, print_iterations=False)`	Obtiene la tasa de acierto del sistema dado un *DataFrame* y una lista de pares de vídeos. Permite personalizar si se quiere usar la distancia euclídea o coseno, la confianza y el umbral.

Tabla 7.3: Funciones presentes en el fichero `accuracy_meter.py`

Mediante la biblioteca `itertools` [89], la cual implementa funciones que crean iteradores eficientes, se procederá a realizar las combinaciones necesarias de vídeos desde el *DataFrame* para formar las listas de pares de vídeos para las pruebas:

```
def get_different_people_videos_ids_pairs(videos_embeddings):
    id_pairs = set()
    for possible_combination in itertools.combinations(videos_embeddings.iterrows(), 2):
        if possible_combination[0][1]['id'] != possible_combination[1][1]['id'] and \
           possible_combination[0][1]['floor'] != possible_combination[1][1]['floor']:

            id_pairs.add((possible_combination[0][1]['video_id'],
                          possible_combination[1][1]['video_id']))
    return list(id_pairs)
```

Fragmento de código 7.1: Implementación de la obtención de pares de vídeos de diferente identidad desde el *DataFrame* en el módulo `accuracymeter` [Fuente]

De esta manera, se podrán obtener todos los pares de vídeos disponibles para realizar las pruebas adecuadas, incluyendo los que tienen la misma identidad, los que tienen distinta y una mezcla:

```
dataframe = embeddings_assembler.load_panda_embeddings(dataframe_name + '.pickle')
same_person_videos_id_pairs_list = accuracy_meter.get_same_person_videos_ids_pairs(dataframe)
different_person_videos_id_pairs_list = accuracy_meter.get_different_people_videos_ids_pairs(dataframe)
[...] # Se desordena con la misma semilla ambas listas
mixed_videos_id_pairs_list = same_person_videos_id_pairs_list.copy()
mixed_videos_id_pairs_list.extend(different_person_videos_id_pairs_list)
```

Fragmento de código 7.2: Generación de pares de vídeos usando el módulo `accuracymeter` del sistema de verificación

7.1.2 Técnicas de evaluación del sistema

Tras haber introducido la teoría que sostiene a los prototipos que se presentarán en esta sección, se procederá a explicar las distintas técnicas de evaluación del sistema de verificación. Se debe recalcar que la evaluación de distancias (apartado 7.1.2.1) y el cálculo de EER (apartado 7.1.2.2) se realiza sobre el conjunto de entrenamiento. Una vez se tiene el umbral con menos EER para dicho conjunto de entrenamiento, se procede a usarlo para evaluar el *accuracy* con su mejor índice de confianza (apartado 7.1.2.3) y a realizar varios tests de funcionamiento del sistema (apartado 7.1.2.4) sobre el conjunto de test.

7.1.2.1 Evaluación de distancias

La primera técnica de evaluación que se explicará será la evaluación de distancias. Dicha evaluación se sostiene en la idea explicada a lo largo de la sección 6.5. Es decir, se podrá validar que el sistema está funcionando de mejor o peor manera dependiendo de cuánta distancia separa a la distribución de distancias entre descriptores de vídeos con la misma identidad y la distribución de distancias entre descriptores de vídeos con distinta identidad. Además, se propone visualizar cómo se comporta la distribución las distancias de descriptores de un vídeo consigo mismo, debiendo este de situarse por debajo de las otras distribuciones lógicamente.

Para lograr visualizar dicha evaluación, se hace uso del módulo `distancesmatrixcalculator` (concretamente, del fichero `distances_matrix_loader.py` explicado en la tabla 6.12). Una vez se tienen en listas las matrices de distancias y una serie de percentiles, se hace uso de las utilidades desarrolladas `utils.distance_plotter` y `utils.distance_plotter_boxplot` para generar gráficas similares a las siguientes:

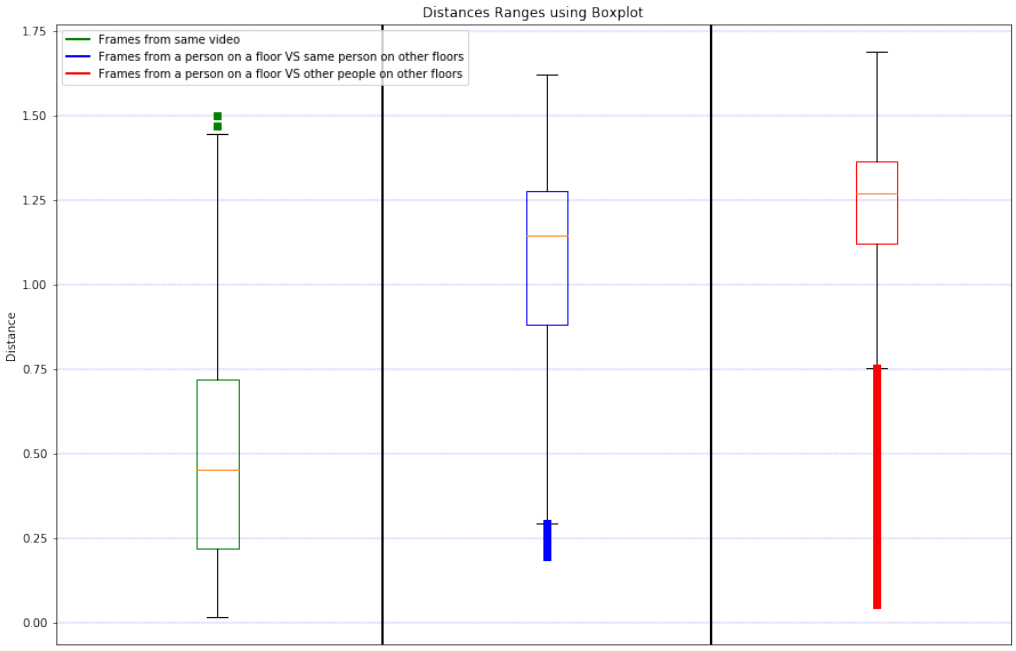


Figura 7.1: Representación de distancias generadas entre descriptores mediante diagramas de cajas de las matrices de distancias concatenadas

Gracias a la representación de la figura 7.1, se puede observar cómo la sección coloreada de verde (representando distancias de un vídeo consigo mismo) debería quedar por debajo de las otras dos distribuciones de distancias.

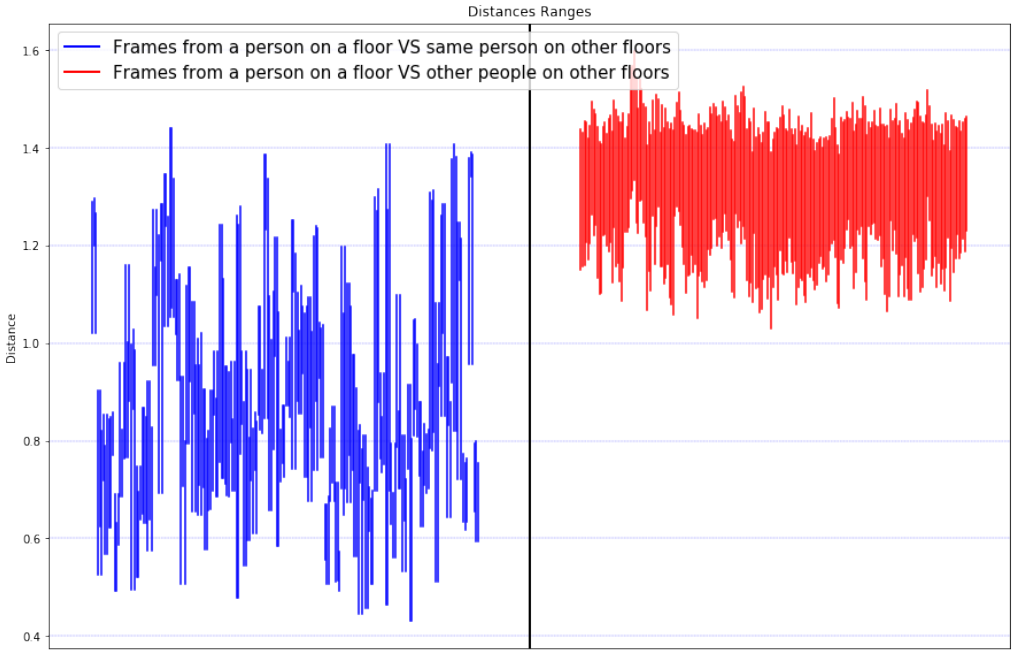


Figura 7.2: Representación de distancias generadas entre descriptores mediante los percentiles de las matrices de distancias generadas

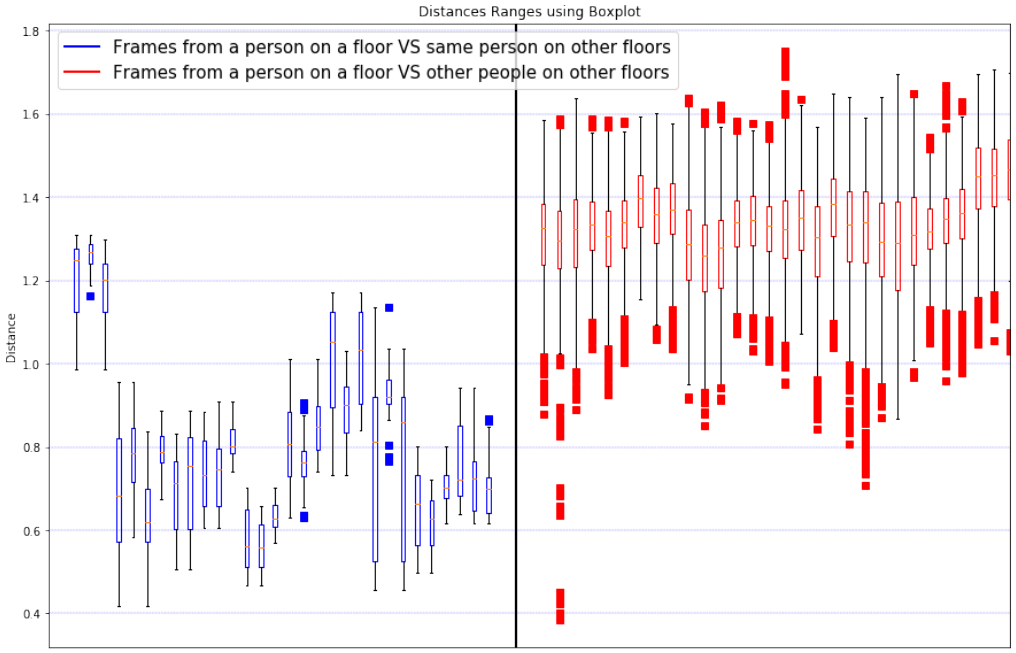


Figura 7.3: Representación de distancias generadas entre descriptores mediante diagramas de cajas de las matrices de distancias

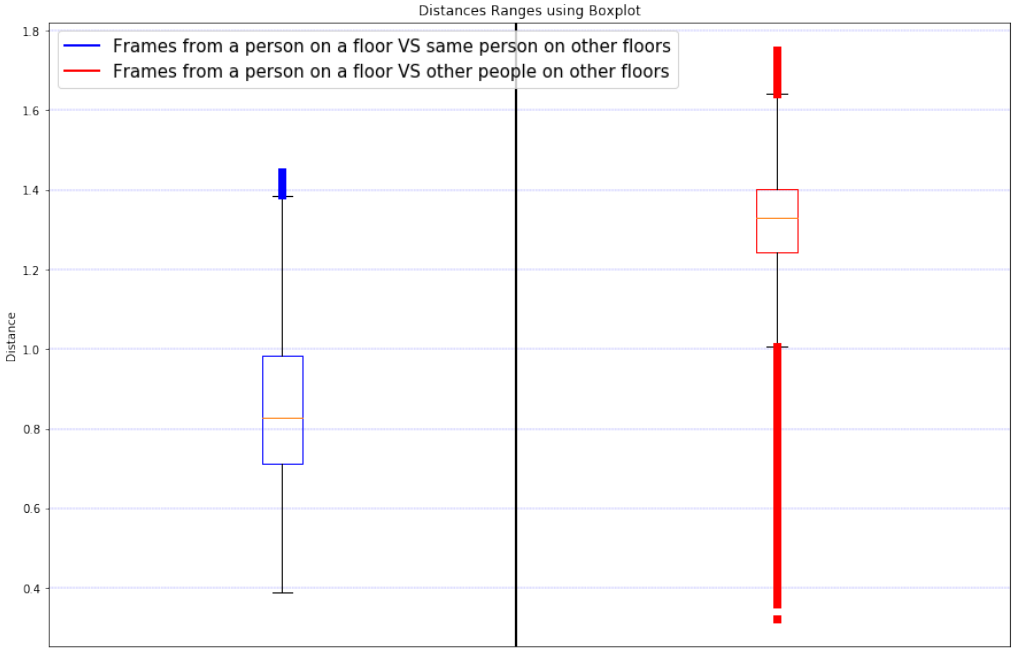


Figura 7.4: Representación de distancias generadas entre descriptores mediante diagramas de cajas de las matrices de distancias concatenadas

No se debe olvidar que el objetivo de un sistema de verificación basado en un umbral es dibujar una línea horizontal ficticia que separe las secciones azules (distancias de descriptores de vídeos con la misma identidad) de las secciones rojas (distancias de descriptores de vídeos con distinta identidad). Con esta herramienta se tiene una estimación visual del valor del umbral.

La diferencia entre las tres representaciones, como se muestra en el pie de figura de cada una de ellas, consiste en recoger los percentiles de las matrices de distancias generadas con los pares de vídeos (figura 7.2), en recoger cada una de las matrices de forma separada para generar diagramas de cajas y observar su distribución (figura 7.3) y en recoger todas las matrices concatenándolas en una sola para cada una de las secciones que se pretenden estudiar (figura 7.4), siendo esta última la que clarifica de mejor manera las distribuciones de distancias que se desean estudiar y la que se mostrará a lo largo de los prototipos.

```
dpb = distance_plotter_boxplot.DistancePlotterBoxplot(numpy.concatenate(same_person_euclidean_distance_list),
                                                      numpy.concatenate(different_people_euclidean_distance_list))
dpb.draw_plot()
```

Fragmento de código 7.3: Generación de una representación similar a la figura 7.4 usando la utilidad `distance_plotter_boxplot`

7.1.2.2 FAR, FRR y EER

La segunda técnica de evaluación que se propone es el cálculo del EER [90] o *Equal Error Rate* (Ratio de error equivalente). Para explicar dicho error, es necesario aclarar en qué consiste el FRR o *False Rejection Rate* (Ratio de rechazo falso) y el *False Acceptance Rate* (Ratio de aceptación falsa):

- FRR: muestra el error que se produce con aquellos pares de vídeos que deberían haberse verificado de manera positiva y su salida fue negativa. En otras palabras, es el error producido por descartar verificaciones originalmente genuinas.
- FAR: muestra el error que se genera con los pares de vídeos que se correspondían con una verificación negativa y su salida fue positiva. Es decir, es el error producido por aceptar verificaciones formadas por impostores.

Teniendo en cuenta ambas definiciones, el EER es el punto donde ambos errores se igualan cuando se incrementa progresivamente el umbral. Por ello, si se obtiene el umbral donde se produce dicho EER, se obtendrá el mejor umbral disponible con el *DataFrame* que se esté utilizando.

Evidentemente, al incrementar el umbral de aceptación de una distancia producida por dos descriptores, se estará facilitando la salida positiva del sistema de verificación y con ello, el FRR deberá reducirse en consecuencia. Por el contrario, si se decrementa dicho umbral de aceptación, se estará dificultando la salida positiva del sistema de verificación y, por lo tanto, el FAR deberá reducirse.

Con el fin de visualizar esta segunda técnica de evaluación del sistema, se hace uso de la utilidad `threshold_plotter.py` para generar representaciones similares a la de la figura 7.5:

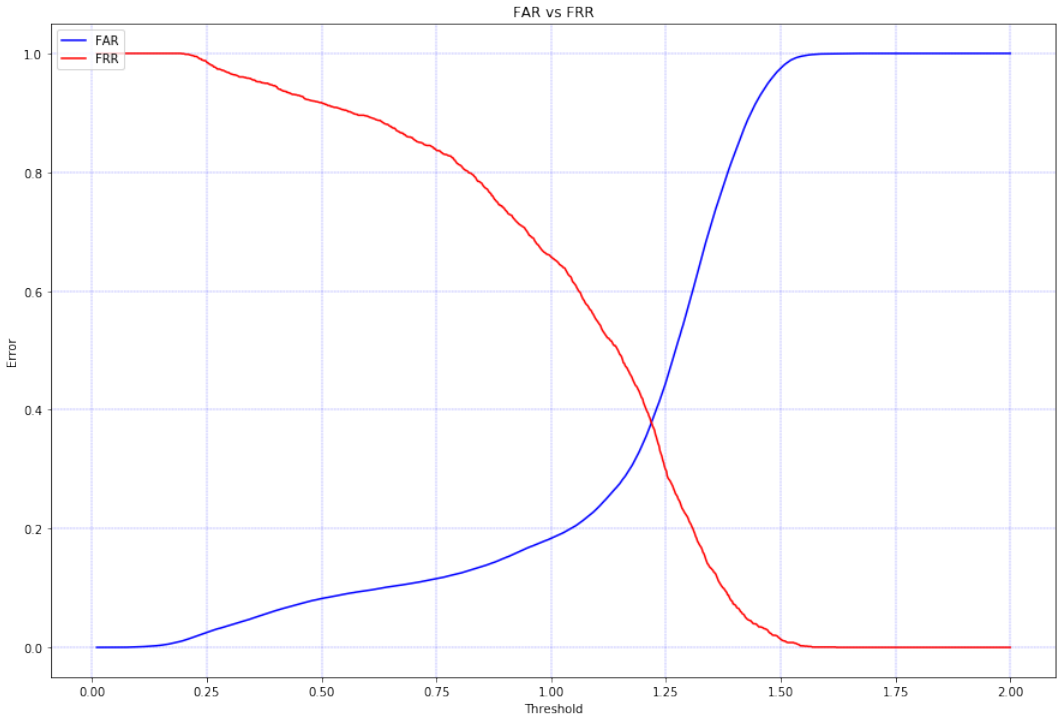


Figura 7.5: Representación de FAR, FRR y ERR para el cálculo del mejor umbral (en este caso, EER: 0.38 y mejor umbral: 1.219)

La utilidad desarrollada `threshold_plotter.py` hace uso para representar la gráfica de la figura 7.5 del módulo `thresholdvalidator`. Dicho módulo contiene el fichero `threshold_validator.py`, que posee las siguientes funciones:

Función	Descripción
`get_false_acceptance_rate(false_acceptances_number, total_samples)`	Calcula el FAR.
`get_false_rejection_rate(false_rejections_number, total_samples)`	Calcula el FRR.
`print_errors(false_rejection_rate, false_acceptance_rate)`	Imprime por pantalla con un formato estético el FAR y el FRR.
`measure_threshold(threshold, original_embedding_matrix, comparing_embedding_matrix, total_columns_with_same_person, distance_type='euclidean', print_results=False)`	Dado un umbral (*threshold*) y dos matrices de descriptores a comparar (siendo la primera perteneciente a un vídeo de una identidad y la segunda una concatenación entre procedentes de la misma y distinta identidad), devuelve el FAR y el FRR, entre otros.

Tabla 7.4: Funciones presentes en el fichero `threshold_validator.py`

En concreto, la implementación interna de la medición de la calidad del umbral es similar al fragmento de código que se muestra a continuación:

```
def measure_threshold(threshold, original_embedding_matrix, comparing_embedding_matrix,
                      total_columns_with_same_person,
                      distance_type='euclidean', print_results=False):
    [...]

    if distance_type == 'euclidean':
        distance_matrix = euclidean_distances(original_embedding_matrix, comparing_embedding_matrix)
    elif distance_type == 'cosine':
        distance_matrix = cosine_distances(original_embedding_matrix, comparing_embedding_matrix)

    fa_number = numpy.where(distance_matrix[:, total_columns_with_same_person:] < threshold)[0].shape[0]
    fr_number = numpy.where(distance_matrix[:, :total_columns_with_same_person] > threshold)[0].shape[0]

    genuine_samples = distance_matrix.shape[0] * total_columns_with_same_person
    impostor_samples = distance_matrix.shape[0] * \
                       (distance_matrix.shape[1] - total_columns_with_same_person)

    false_acceptance_rate = get_false_acceptance_rate(fa_number, impostor_samples)
    false_rejection_rate = get_false_rejection_rate(fr_number, genuine_samples)

    [...]

    return [false_acceptance_rate, false_rejection_rate, genuine_samples,
            impostor_samples, fa_number, fr_number]
```

Fragmento de código 7.4: Cálculo de FAR y FRR en el módulo `thresholdvalidator` [Fuente]

Recorriendo por cada una de las identidades sus matrices de descriptores y comparándolas con matrices de descriptores de muestras de otros pisos procedentes de esas mismas identidades y de otras identidades se van obteniendo una serie de valores FAR y FRR. Realizando medias de dichos valores se obtiene la gráfica representada en la figura 7.5 y, calculando el punto de intersección entre ambos errores, el EER, y con ello, el mejor umbral para el sistema de verificación propuesto con el *DataFrame* usado.

```
tp = threshold_plotter.ThresholdPlotter(dataframe, threshold_min=0.01, threshold_max=2)

tp.draw_plot()

best_threshold, eer = tp.calculate_best_threshold_and_eer()

print("\nBest threshold:", best_threshold)
print("EER:", eer)
```

Fragmento de código 7.5: Generación de una representación similar a la figura 7.5 usando la utilidad `threshold_plotter` y el módulo `thresholdvalidator`

7.1.2.3 Cálculo del mejor índice de confianza y *accuracy*

Como se mencionó en la sección 7.1.1, existen dos parámetros personalizables en un sistema de verificación basado en un umbral: el propio umbral y la confianza (si se realiza mediante vídeos). La tercera técnica de evaluación que se propone es la tasa de acierto del sistema (o *accuracy*). Por ello, para el cálculo de la confianza óptima, que proporcionará la mejor tasa de acierto dado un EER, se utilizará una técnica muy similar a la del cálculo de dicho error.

Concretamente, se irán obteniendo distintas tasas de acierto del sistema a raíz de cada una de las listas de pares de vídeos generadas en la sección 7.1.1.1. Se introducirán cada uno de los pares de vídeos de dichas listas en el sistema de verificación y a raíz de la proporción de aciertos que se obtenga mientras se incrementa la confianza, se obtendrá la que provoca una mayor tasa de acierto. Evidentemente:

- Una confianza alta provocará que prácticamente todas las muestras de la matriz de distancias deban cumplir con permanecer por debajo del umbral propuesto para dar un resultado positivo. En consecuencia, si existe algún valor atípico alto, la salida será errónea.
- Una confianza baja provocará que el sistema devuelva una salida positiva si una proporción baja de los valores de la matriz de distancias se encuentra por debajo del umbral. En otras palabras, un valor atípico bajo originará una salida equivocada.

Gracias a la utilidad desarrollada `confidence_plotter.py`, se podrá obtener una gráfica similar a la figura 7.6, donde se pueden visualizar tres líneas:

- La línea azul representa la tasa de acierto con pares de vídeos formados por la misma identidad. Evidentemente, entre menor proporción de valores sea necesario para dar una salida positiva, mayor tasa de acierto obtendrá.
- La línea roja representa la tasa de acierto con pares de vídeos formados por distinta identidad. Lógicamente, entre mayor proporción de valores se requiera para otorgar una salida positiva, mayor tasa de acierto poseerá.
- La línea verde representa la tasa de acierto con pares de vídeos formados por la misma y por distinta identidad con un *undersampling*[1] de los pares de la última. En circunstancias normales, en la gráfica deberá situarse entre la línea roja y azul siempre.

La distribución de las líneas dependerá directamente de la dimensión de la matriz de distancias a la que se le ejerce la confianza como proporción. En el punto real donde las tres líneas horizontales se encuentren más cerca entre sí, se podrá calcular qué confianza se ha usado para obtener las mejores tasas de acierto. En el caso de la figura 7.6, la confianza fue de un 50.5 %.

Internamente, `confidence_plotter.py` hace uso de la función `get_accuracy(...)` explicada en la tabla 7.3.

[1] *Undersampling*: técnica en la que se reduce la cantidad de muestras de una clase con la finalidad de tratar de igualarla con otras

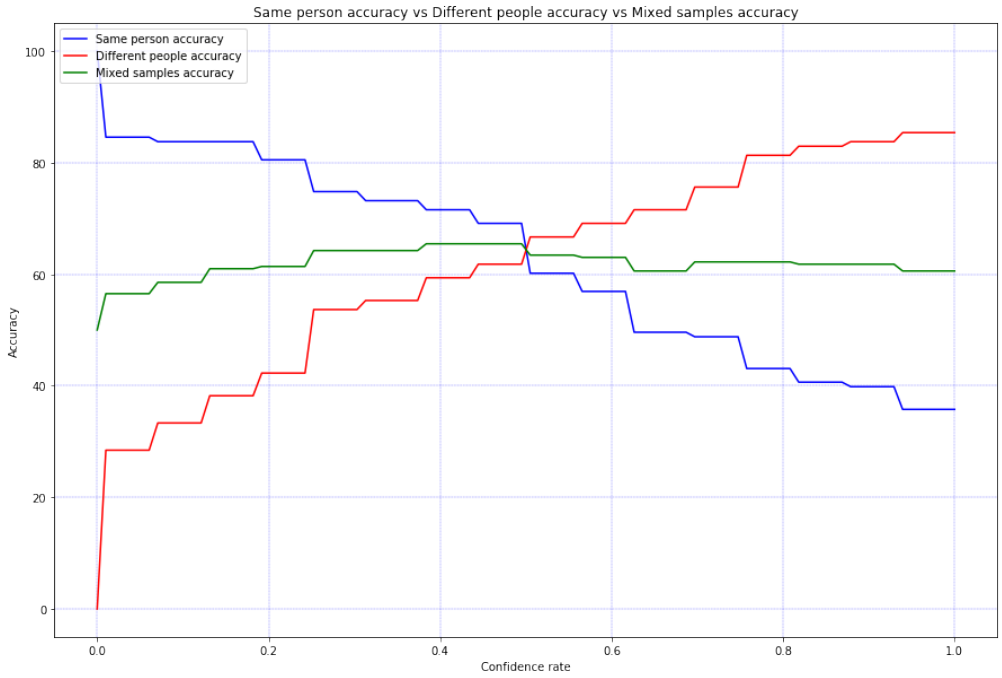


Figura 7.6: Representación de la tasa de acierto para el cálculo de la mejor confianza (en este caso, tasas de acierto: 60.163 %, 66.667 % y 63.415 % y mejor confianza: 0.505)

```python
def get_accuracy(videos_embeddings, ids_pairs_list, distance_type='euclidean', confidence=0.5,
                 threshold=1.193, print_iterations=False):
    verifier = Verifier(videos_embeddings, distance_type, confidence, threshold)
    success_samples = 0
    for index, ids_pair in enumerate(ids_pairs_list):
        [...]
        first_id = videos_embeddings[videos_embeddings['video_id'] == ids_pair[0]]['id'].unique()
        second_id = videos_embeddings[videos_embeddings['video_id'] == ids_pair[1]]['id'].unique()

        verifier_result = verifier.is_same_person(ids_pair[0], ids_pair[1])
        if first_id == second_id and verifier_result['is_same_person']:
            success_samples += 1
        elif first_id != second_id and not verifier_result['is_same_person']:
            success_samples += 1
    return round(success_samples*100/len(ids_pairs_list), 3)
```

Fragmento de código 7.6: Cálculo del *accuracy* dado un umbral y una confianza usando un *DataFrame* y una lista de pares de vídeos en el módulo `accuracymeter` [Fuente]

7.1.2.4 Tests de funcionamiento del sistema

La cuarta y última técnica de evaluación que se propone para un sistema de verificación basado en umbral es recoger 20 pares de vídeos (10 pares de vídeos con la misma identidad y 10 pares de vídeos con distinta identidad) aleatorios y, haciendo uso del módulo `verifier` (que será explicado con mayor profundidad en la sección 7.3), del *DataFrame* y de los mejores parámetros de umbral y de confianza calculados, probar si sus salidas son correctas y estimar si el error y el *accuracy* calculado sigue su lógica de acierto.

Además, al realizarse sobre el conjunto de test (al igual que el cálculo del parámetro de confianza y la tasa de acierto), se puede comprobar de manera individual (mostrando el resultado de cada par de vídeos) si el umbral calculado con el conjunto de entrenamiento es válido para el conjunto de test (situación analizada también mediante la tasa de acierto en la sección anterior).

En el cuaderno Jupyter, dichas pruebas se realizan de la siguiente manera:

```
vf = verifier.Verifier(dataframe, confidence=0.505, threshold=1.219)
sp_vipl_copy = same_person_videos_id_pairs_list.copy()
dp_vipl_copy = different_person_videos_id_pairs_list.copy()
random.shuffle(sp_vipl_copy)
random.shuffle(dp_vipl_copy)
print("[Same person tests]")
for ids_pair in sp_vipl_copy[:10]:
    isp_result = vf.is_same_person(ids_pair[0], ids_pair[1])
    [...]
    print("[" + ids_pair[0] + " VS " + ids_pair[1] + ":\t" + str(isp_result) + "]")

print("[Different person tests]")
for ids_pair in dp_vipl_copy[:10]:
    isp_result = vf.is_same_person(ids_pair[0], ids_pair[1])
    [...]
    print("[" + ids_pair[0] + " VS " + ids_pair[1] + ":\t" + str(isp_result) + "]")
```

Fragmento de código 7.7: Generación de varios tests de funcionamiento de la verificación

Lo que genera una salida similar a la siguiente:

```
[Same person tests]
[averobot_floor_01-48_01_03_08_02/ VS averobot_floor_03-48_03_03_30_08/:        {'is_same_person': True, 'confidence': 0.625}]
[averobot_floor_01-101_01_03_15_02/ VS averobot_floor_02-101_02_03_20_03/:      {'is_same_person': True, 'confidence': 1.0}]
[averobot_floor_02-110_02_03_14_03/ VS averobot_floor_03-110_03_02_03_08/:      {'is_same_person': False, 'confidence': 1.0}]
[averobot_floor_01-90_01_01_29_02/ VS averobot_floor_02-90_02_03_10_03/:        {'is_same_person': True, 'confidence': 1.0}]
[averobot_floor_01-52_01_01_08_02/ VS averobot_floor_02-52_02_03_25_03/:        {'is_same_person': True, 'confidence': 1.0}]
[averobot_floor_01-104_01_02_08_02/ VS averobot_floor_02-104_02_01_05_03/:      {'is_same_person': True, 'confidence': 1.0}]
[averobot_floor_02-39_02_02_11_03/ VS averobot_floor_03-39_03_02_22_08/:        {'is_same_person': True, 'confidence': 1.0}]
[averobot_floor_01-51_01_02_19_02/ VS averobot_floor_03-51_03_02_25_08/:        {'is_same_person': True, 'confidence': 1.0}]
[averobot_floor_02-15_02_03_12_03/ VS averobot_floor_03-15_03_02_27_08/:        {'is_same_person': True, 'confidence': 0.75}]
[averobot_floor_02-105_02_02_22_03/ VS averobot_floor_03-105_03_03_08_08/:      {'is_same_person': False, 'confidence': 0.688}]

[Different person tests]
[averobot_floor_01-08_01_01_10_02/ VS averobot_floor_02-16_02_02_28_03/:       {'is_same_person': True, 'confidence': 1.0}]
[averobot_floor_01-88_01_02_08_02/ VS averobot_floor_02-92_02_03_10_03/:       {'is_same_person': False, 'confidence': 1.0}]
[averobot_floor_03-35_03_02_13_08/ VS averobot_floor_01-101_01_03_15_02/:      {'is_same_person': False, 'confidence': 0.938}]
[averobot_floor_01-83_01_01_27_02/ VS averobot_floor_02-32_02_01_15_03/:       {'is_same_person': False, 'confidence': 0.5}]
[averobot_floor_01-104_01_02_08_02/ VS averobot_floor_02-09_02_03_18_03/:      {'is_same_person': False, 'confidence': 1.0}]
[averobot_floor_01-80_01_03_29_02/ VS averobot_floor_02-32_02_01_15_03/:       {'is_same_person': False, 'confidence': 0.5}]
[averobot_floor_03-15_03_02_27_08/ VS averobot_floor_02-92_02_03_10_03/:       {'is_same_person': True, 'confidence': 0.75}]
[averobot_floor_01-11_01_03_29_02/ VS averobot_floor_02-39_02_02_11_03/:       {'is_same_person': False, 'confidence': 1.0}]
[averobot_floor_03-98_03_01_27_08/ VS averobot_floor_01-45_01_01_28_02/:       {'is_same_person': False, 'confidence': 1.0}]
[averobot_floor_03-91_03_02_34_08/ VS averobot_floor_01-53_01_02_01_02/:       {'is_same_person': False, 'confidence': 1.0}]
```

Fragmento de código 7.8: Resultado de varios tests de funcionamiento de la verificación

Donde se puede observar si se ha verificado correctamente y, además, la confianza que el sistema de verificación devuelve del resultado propuesto (la confianza surge a partir de la proporción de números de la matriz de distancias que cumplieron la decisión final del sistema).

Por último, una vez se tienen los identificadores de los pares de vídeos que fallan en el sistema de verificación se puede hacer uso de la utilidad `failed_frames_pairs_generator.py` para, haciendo un proceso de ingeniería inversa, mostrar por pantalla en qué detecciones está fallando el sistema de verificación y comprobar si tiene coherencia el error.

7.1.3 Sistemas construidos sobre FaceNet y VGGFace2

En esta sección, se adjunta una tabla con una descripción breve de cada uno de los 10 prototipos o modelos construidos para la verificación basada en umbral de distancia probados sobre el Subconjunto de vídeos B usando FaceNet (Inception ResNet v1) y VGGFace2 - ResNet50 como generadores de descriptores con su correspondiente evaluación:

Nº Modelo	Descripción	Mejor umbral	EER	Mejor confianza	*Accuracy*[2]
1	El modelo se basa en una detección sin ninguna normalización con MTCNN con 4 fotogramas máximos por vídeo, recogiendo los descriptores con FaceNet (Inception ResNet v1) y L2 y haciendo uso de distancias euclídeas en el cálculo de la similitud entre dichos descriptores.	1.16	0.105	31.3 %	93.09 %
2	Similar al Modelo 1, con la diferencia de aplicar una redimensión de la detección. El Redimensionamiento de la detección proporciona que los rostros de entrada de los generadores de descriptores sean más apropiados al tener dimensiones parecidas a las caras con las que fueron entrenados, lo que mejora los resultados.	1.13	0.083	44.4 %	94.31 %
3	Parecido al Modelo 1, diferenciándose en realizar una normalización básica basada en una doble detección (ver sección 6.2.3.2). Como se puede observar, la normalización trae consigo la pérdida de definición mencionada al explicarse y la doble detección implica el uso de una proporción de las caras demasiado cercana con respecto a las que se usaron para entrenar FaceNet o VGGFace2, lo que disminuye la tasa de acierto e incrementa el error.	1.19	0.167	44.4 %	87.81 %
4	Semejante al Modelo 1, salvo por el hecho de realizar una normalización básica basada en un recorte estático (ver apartado 6.2.3.1). Al usar la normalización (o alineamiento) de rostros, se tiene una pérdida de calidad de la imagen que no permite que el Modelo mejore en cuestión de tasa de acierto o error.	1.12	0.109	25.3 %	93.09 %
5	Similar al Modelo 2, con la diferencia de usar distancias coseno en lugar de distancias euclídeas (ver sección 6.5). Los resultados son muy similares pero se puede observar cómo, debido a la distribución de distancias coseno, el umbral disminuye su valor.	0.64	0.085	44.4 %	94.31 %
6	Parecido al Modelo 2. No obstante, se recoge únicamente el mejor fotograma de cada vídeo manualmente entre los 4 que otorga la selección automática[3] (ver apartado 6.2.4.1).	1.10	0.057	-	95.94 %
7	Semejante al Modelo 6[3]. Sin embargo, se utilizan distancias coseno en lugar de distancias euclídeas.	0.61	0.060	-	95.53 %
8	Similar al Modelo 2, diferenciándose en usar el Detector DLIB - MMOD en lugar de MTCNN. En este caso, se producen unos resultados peores que con MTCNN.	1.17	0.145	50.5 %	90.65 %
9	Parecido al Modelo 2, salvo por el hecho de usar el Detector DLIB - HOG en lugar de MTCNN. Se produce de nuevo un error más alto que con MTCNN, pero más bajo que con el Detector DLIB - MMOD.	1.13	0.107	62.6 %	95.12 %
10	Semejante al Modelo 2, con la diferencia de usar el generador de descriptores VGGFace2 - ResNet50 en lugar de FaceNet. Este sistema se convierte en el que contiene menos error y mayor tasa de acierto de todos los diseñados y el que se tomará de referencia en futuros estudios comparativos.	1.03	0.017	13.1 %	100 %

Tabla 7.5: Sistemas de verificación basados en umbrales de distancia y resultados de su evaluación

[2]El cálculo del *accuracy* se basa en la tasa de acierto calculada con la misma cantidad de pruebas con pares de vídeos de la misma identidad y de distinta identidad en el conjunto de test.

[3]En este modelo, y como se ha mencionado a lo largo del capítulo, no tiene sentido hacer uso de una confianza al solo disponer de un fotograma por vídeo. Además, al haber realizado una selección manual, tampoco se puede considerar un sistema con posibilidad de implementación real.

Se adjuntan varias representaciones gráficas de la evaluación del mejor modelo diseñado para la verificación basada en umbral de distancia (Modelo 10 de la tabla 7.5) para el Subconjunto de vídeos B haciendo uso de las utilidades desarrolladas:

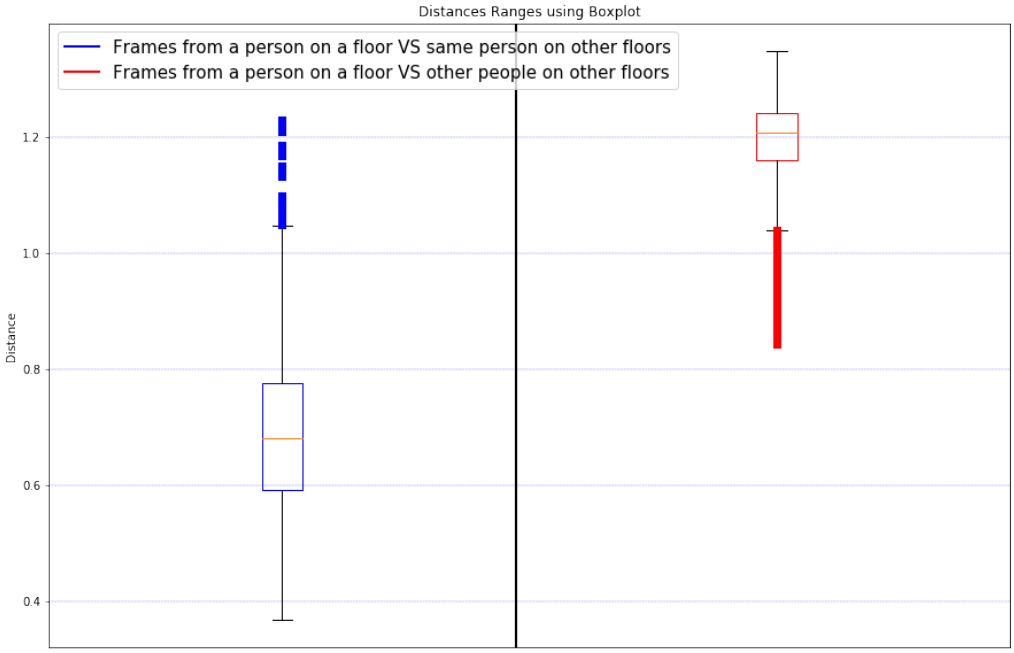


Figura 7.7: Distribución de distancias euclídeas del conjunto de entrenamiento

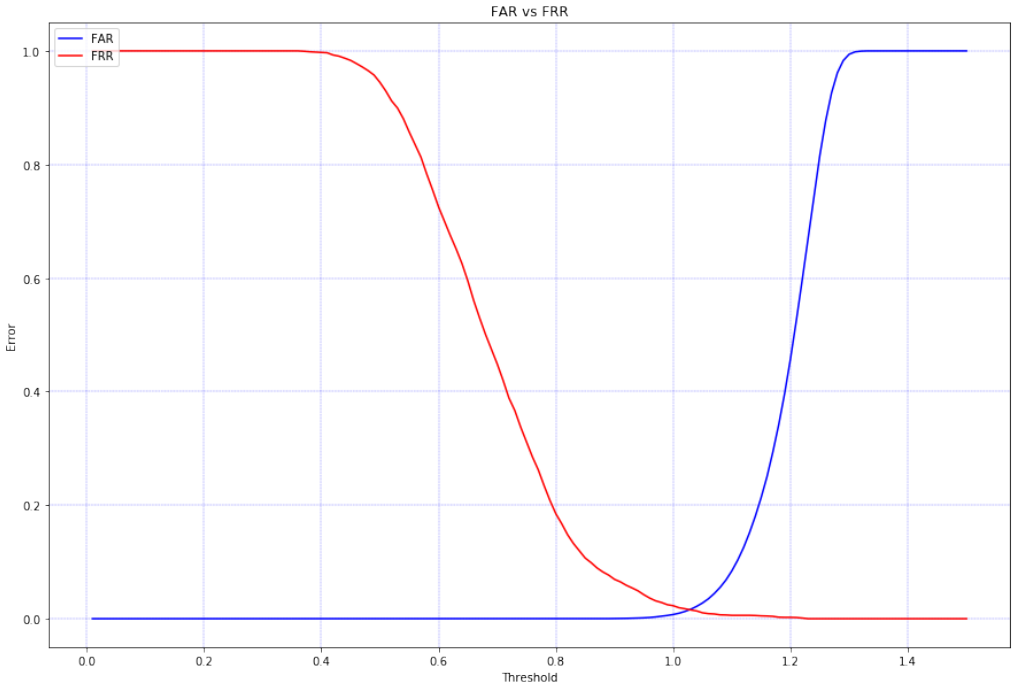


Figura 7.8: Representación de EER del conjunto de entrenamiento (en este caso, mejor umbral: 1.03 y EER: 0.017)

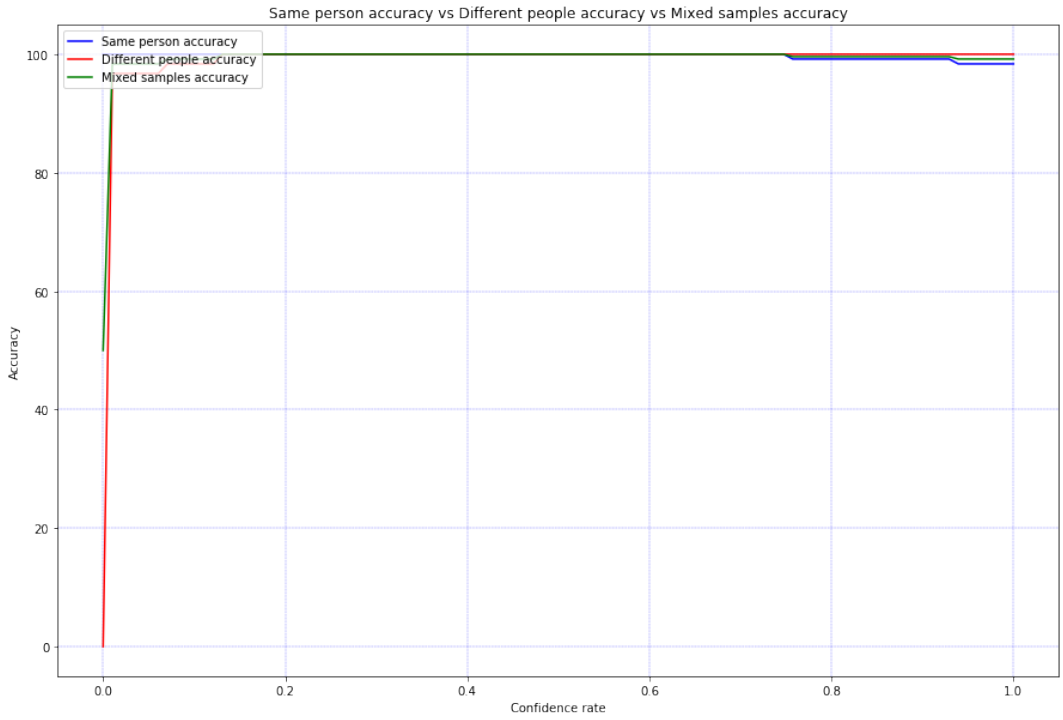
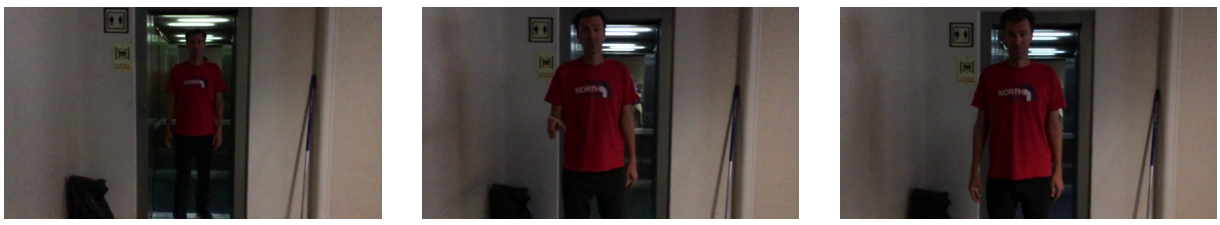


Figura 7.9: Representación de la tasa de acierto respecto a la confianza sobre el conjunto de test usando el umbral hallado en la figura 7.8 (en este caso, mejor confianza: 0.131 (13.1 %) y tasa de acierto: 100.0 %)

Se pueden observar estas gráficas y otras para el resto de modelos construidos en los cuadernos de Jupyter Notebook presentes en el repositorio del proyecto [30].

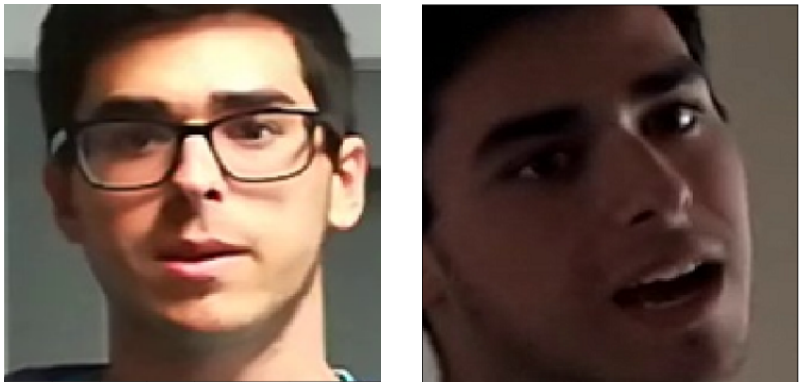
Es necesario mencionar en este punto que el Detector DLIB - HOG tiene ciertas dificultades para detectar alguna muestra en determinados vídeos si estos son de baja calidad, incluso realizando *upsampling*. Por este motivo, y aun consiguiendo la mejor tasa de acierto bajo FaceNet (Inception ResNet v1) con selección automática de fotogramas, se considera un prototipo inestable y carente de robustez (ver figura 7.10).



(a) Muestra 1 (b) Muestra 2 (c) Muestra 3

Figura 7.10: Ejemplo de vídeo con identidad 5 indetectable para el Detector DLIB - HOG

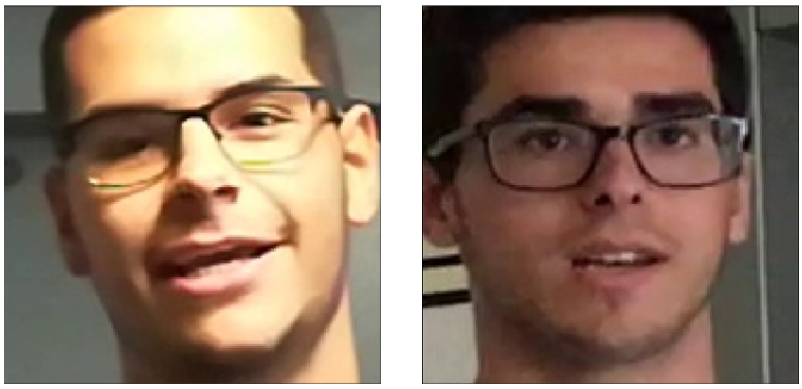
A continuación, se ilustra un ejemplo de falso negativo en el sistema (en este caso, para el modelo 5 de la tabla 7.5), donde se puede observar cómo los casos fallidos pueden llevar consigo circunstancias no previstas:



(a) En piso 1 (b) En piso 3

Figura 7.11: Falso negativo con la identidad 38 debido a un cambio en sus accesorios faciales de un piso a otro

Además, se puede observar cómo los falsos positivos (en este caso, para el modelo 7 de la tabla 7.5) también tienen cierta coherencia debido al parecido de las identidades:



(a) Identidad 37 (b) Identidad 38

Figura 7.12: Falso positivo entre dos identidades parecidas

A lo largo de los documentos de Jupyter Notebook del repositorio [30], se pueden observar más falsos positivos y falsos negativos de los sistemas propuestos para cada uno de los subconjuntos descritos en la tabla 5.2.

7.2 Verificación mediante redes neuronales y evaluación

El segundo y último tipo de sistema de verificación en escenarios multinivel será el basado en redes neuronales. Primero, se procederá a explicar en qué consiste el uso de redes neuronales destinadas a tareas de verificación para, a continuación, analizar diversas técnicas de evaluación de los prototipos que se desarrollaron finalizando con una presentación de estos.

7.2.1 Verificación basada en redes neuronales

Si se reflexiona sobre en qué consistía la verificación basada en umbral de distancia (ver apartado 7.1.1), se recordará que su objetivo era una clasificación en la que existían dos clases posibles: mismo o distinto individuo.

Este escenario justifica que una red neuronal, la cual puede basarse en el hecho de tratar de conseguir una clasificación de clases haciendo uso de neuronas artificiales con entrada y salida conectadas entre sí transmitiéndose señales, pueda ser un componente ideal para tratar de ser el cimiento de un sistema de verificación.

En términos más concretos, si previamente se consideró un umbral como un valor numérico que trataba de separar o dividir las distancias entre dos descriptores en dos intervalos distintos para clasificar, ahora se plantea realizar una red neuronal (sometida a un proceso de aprendizaje previo) *fully connected* que, dada una entrada formada por los descriptores de los *DataFrames* (y en algunos modelos las distancias euclídeas de los pares a comparar), logre dar la misma salida binaria: mismo o distinto individuo. Por ello, se plantea una arquitectura similar a la representada en la figura 7.13:

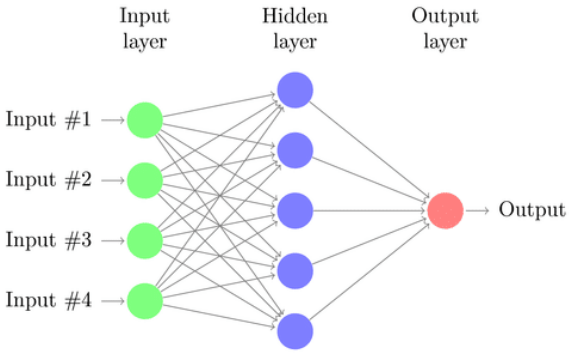


Figura 7.13: Arquitectura de una red neuronal con una neurona de salida [Fuente]

Además, en algunos prototipos se propone el uso de descriptores procedentes de distintos detectores que puedan dotar a la red neuronal de varias "opiniones" sobre una identidad, provocando que la red neuronal sea capaz de aprender detecciones dispares de la misma persona y mejore su tasa de acierto.

En esta ocasión, el módulo `verifiermodel` será el encargado de realizar todas las tareas de verificación mediante redes neuronales. Los métodos del objeto `NNVerifier` procedentes del único fichero del módulo `nnverifier.py` se describen a continuación:

Método	Descripción
`__init__(self, train_videos_embeddings_1, test_videos_embeddings_1, train_videos_embeddings_2=None, test_videos_embeddings_2=None, mlp_type='default', select_one_frame=False, use_one_detector=False, use_kolmogorov_theorem=False, use_euclidean_distance=False)`	Inicializa el verificador basado en redes neuronales según una lista de parámetros.
`get_embedding_pairs(self, pair_type='train')`	Obtiene pares de descriptores que servirán de entrada en los diferentes modelos de redes neuronales que se diseñarán. Dichas entradas se devuelven separadas en si contienen o no a la misma persona, sirviendo para el proceso de entrenamiento. Devuelve la distancia euclídea entre descriptores si se requiere por las circunstancias del modelo.
`create_mlp(self, print_summary=True)`	Crea el perceptrón multicapa con las características con las que se inicializó el objeto `NNVerifier` usando Keras.
`train_model(self, same_person_embeddings_pairs_train, different_people_embeddings_pairs_train, batch_size=50000, epochs=15, same_person_ed=None, different_people_ed=None)`	Tras preparar la entrada del entrenamiento a unas dimensiones pertinentes y ajustar las salidas que se deberían tener en cada caso, se entrena el modelo. En el proceso, se realiza un *undersampling* de la entrada cuya salida es "distinta persona" debido a la existencia de mayores pares de este tipo.
`evaluate(self, same_person_embeddings_pairs_test, different_people_embeddings_pairs_test, same_person_ed=None, different_people_ed=None)`	Tras preparar la entrada para las pruebas con las dimensiones adecuadas y ajustar las salidas lógicas para cada muestra, se evalúa el modelo haciendo uso del evaluador de Keras. Se devuelve la tasa de acierto (*accuracy*) y el error o pérdida (*loss*).
`is_same_person(self, id_video_1, id_video_2)`	Devuelve si dos vídeos de un *DataFrame* (desde su ID) contienen al mismo individuo. Para ello, prepara la entrada adecuándola al `NNVerifier` construido y, dependiendo de la proporción de resultados positivos de las muestras a comparar, se devuelve que ambas captaciones pertenecen a la misma persona o no.
`save_model(self, model_name)`	Guarda el modelo entrenado.
`load_model(self, model_name)`	Carga un modelo pre-entrenado.

Tabla 7.6: Funciones presentes en el fichero `nnverifier.py`

Respecto a la implementación interna, cabe destacar el manejo previo que se debe realizar con los datos que servirán de entrada para cada uno de los prototipos que se desarrollarán en las siguientes secciones.

Keras a la hora de diseñar redes neuronales permite modelos secuenciales [91], donde se definen las capas añadiéndolas consecutivamente a un modelo y la API funcional [92], donde se permite maniobrar más con la arquitectura de la red, pudiendo por ejemplo añadir un nuevo dato en medio de la arquitectura de la red neuronal (ver modelo 7 y 8 de la tabla 7.7):

```
if not self.use_euclidean_distance: # Modelo secuencial
    self.model = Sequential()
    if self.use_kolmogorov_theorem:
        self.model.add(Dense(input_dim, input_dim=input_dim, activation='relu'))
        self.model.add(Dense((2*input_dim)+1, activation='relu'))
        self.model.add(Dense(1, activation='sigmoid'))
    [...]
else: # API funcional
    ed_input = Input(shape=[1], name='euclidean_distance')
    input_layer = Input(shape=[input_dim])
    hidden_layer_1 = Dense(input_dim, activation='relu')(input_layer)
    hidden_layer_2 = Dense(int(input_dim/2), activation='relu')(hidden_layer_1)
    hidden_layer_2_ed = concatenate([hidden_layer_2, ed_input])
    output_layer = Dense(1, activation='sigmoid')(hidden_layer_2_ed)

    self.model = Model(inputs=[input_layer, ed_input], outputs=output_layer)
```

Fragmento de código 7.9: Arquitectura parcial de la red neuronal planteada [Fuente]

Una vez se tienen los datos listos, se realiza el proceso de entrenamiento, extrayendo de dicho conjunto de entrenamiento un 20 % (mediante el parámetro `validation_split`) para probar el conjunto de validación con el objetivo de detectar el *overfitting* o sobre-entrenamiento:

```
self.model.compile(loss='binary_crossentropy', optimizer='rmsprop', metrics=['accuracy'])
[...]
self.history = self.model.fit(X_train, Y_train, epochs=epochs, batch_size=batch_size,
                               verbose=1, validation_split=0.2)
```

Fragmento de código 7.10: Compilación y entrenamiento del modelo [Fuente]

Respecto a la evaluación, Keras proporciona funciones para el cálculo del error (*loss*, no confundir con el EER de la sección 7.1.2.2) y la tasa de acierto (*accuracy*):

```
evaluation_results = self.model.evaluate(X_test, Y_test, verbose=1)
```

Fragmento de código 7.11: Evaluación del modelo [Fuente]

Se debe mencionar que la evaluación se ha realizado con un conjunto de datos de test no visto por el modelo antes en su respectiva fase de entrenamiento (ver sección 5.2).

7.2.1.1 Obtención de las entradas para el entrenamiento y test

Como se mencionó previamente, para cada uno de los prototipos se generan todos los pares de descriptores disponibles y se gestionan según la entrada requerida por cada red diseñada. En términos matemáticos, el número de pares p_n que se generan haciendo uso del método `get_embedding_pairs(...)` el siguiente:

$$p_n = MaxDetections_{number}{}^{2*Detectors_{number}} * Identities * Comparing_{identities} * Floors_{number} * 2$$

Dado a que dicho número vendrá dado por el número de detecciones máximas por cada identidad $MaxDetections_{number}$ elevado al número de detectores usados $Detectors_{number}$ por 2 y multiplicado por el número de identidades del conjunto $Identities$, el número de identidades

a comparar $Comparing_{identities}$, el número de pisos $Floors_{number}$ y por 2 (debido a que un par de descriptores (a, b) también es introducido como (b, a)).

Por ejemplo, en el sistema con el modelo 1 que se explicará en la tabla 7.7, sustituyendo los valores para las muestras procedentes del conjunto de entrenamiento en los pares con distinta identidad queda:

$$p_n = 4^{2*2} * 70 * 69 * 3 * 2 = 7418880$$

Mientras que respecto a los pares con la misma identidad queda:

$$p_n = 4^{2*2} * 70 * 1 * 3 * 2 = 107520$$

El método `get_embedding_pairs(...)` devuelve por pantalla la cantidad de combinaciones para la red que ha conseguido generar.

7.2.2 Técnicas de evaluación del sistema

En este apartado, al igual que se realizó en el apartado 7.1.2, se explicará cómo se evaluarán los prototipos desarrollados.

7.2.2.1 *Accuracy* y *Loss* en los conjuntos de entrenamiento, validación y test

La primera técnica de evaluación se basa en el propio funcionamiento de Keras, quien a medida que va realizando el entrenamiento con su subconjunto apropiado (y con el subconjunto de validación extraído como se realizó en el fragmento de código 7.10) va mostrando en cada iteración o *epoch* el *accuracy* (tasa de acierto) y *loss* (error) que va obteniendo el sistema para dichos subconjuntos de entrenamiento y validación.

Además, gracias al método desarrollado `evaluate(...)`, que se apoya en el método con mismo nombre de Keras (ver fragmento de código 7.11), se logra conseguir una medida de tasa de acierto (*accuracy*) y de error (*loss*) para el conjunto de test (conjunto no visto previamente por el modelo cuando se entrenó).

7.2.2.2 Tests de funcionamiento del sistema

Respecto a la segunda técnica de evaluación, se proponen unos tests del funcionamiento del sistema que funcionan de manera muy similar a los expuestos en la sección 7.1.2.4. En el cuaderno de Jupyter se pueden visualizar varias secciones donde se aplica esta técnica de evaluación.

Además, la utilidad desarrollada `failed_frames_pairs_generator.py` también resulta funcional con los sistemas basados en redes neuronales, con lo que se pueden volver a visualizar las detecciones de los casos fallidos (y por lo tanto, los falsos positivos y los falsos negativos de nuevo).

7.2.3 Sistemas construidos sobre FaceNet

A continuación, se adjunta una tabla con los 9 modelos que tratan de resolver la verificación basándose en redes neuronales y sus respectivos resultados probados sobre el Subconjunto de vídeos B usando FaceNet (Inception ResNet v1) como generador de descriptores:

Nº Modelo	Descripción	Entrada	*Accuracy*[4]	*Loss*[4]
1	Su arquitectura se basa en una entrada con doble "opinión" proporcionada por el Detector DLIB - MMOD y MTCNN. Así pues, para un par de captaciones a verificar, se concatenan 4 descriptores (2 por captación) procedentes de ambos detectores.	2 descriptores de 128 desde 2 captaciones $(2 * 2 * 128 = 512)$	82.9 %	0.422
2	Parecido al Modelo 1, diferenciándose en que se produce una resta entre la concatenación de descriptores procedentes de las dos captaciones. La diferencia entre los descriptores añade información valiosa que mejora el *accuracy* de la red.	2 descriptores de 128 desde 2 captaciones restados entre sí $(2 * 128 = 256)$	88.5 %	0.358
3	Similar al Modelo 2, pero cuando produce la resta, a continuación se recoge el valor absoluto del resultado. De esta manera, se evita que el orden del par de vídeos afecte al resultado (lo que produce una mejora del *accuracy* del sistema).	2 descriptores de 128 desde 2 captaciones restados entre sí $(2 * 128 = 256)$	92.3 %	0.209
4	Parecido al Modelo 3. No obstante, se utiliza únicamente la mejor detección automática de las captaciones. Estudiando el resultado, se analiza una disminución del *accuracy* originada por descender el número de detecciones (ver apartado 6.2.4.1).	2 descriptores de 128 desde 2 captaciones restados entre sí $(2 * 128 = 256)$	90.0 %	0.264
5	Su arquitectura implementa una entrada con una única "opinión" proporcionada por MTCNN. Por ello, para un par de captaciones a verificar, se utilizan 2 descriptores (únicamente 1 por captación) procedentes de dicho detector. A continuación, se restan ambos descriptores y se calcula el valor absoluto del resultado. Como se observa en el resultado, usar una sola "opinión" mejora el *accuracy*.	1 descriptor de 128 desde 2 captaciones restados entre sí $(1 * 128 = 128)$	92.8 %	0.206
6	Similar al modelo 5 usando el Teorema de Kolmogorov[5] aplicado a la construcción de arquitecturas de redes neuronales. Dicho teorema estabiliza el entrenamiento y mejora el *accuracy* a la par que reduce el error.	1 descriptor de 128 desde 2 captaciones restados entre sí $(1 * 128 = 128)$	93.3 %	0.183
7	Semejante al modelo 6, salvo por el hecho de introducir la distancia euclídea entre ambos descriptores como una nueva entrada en la red. Como se observa en el resultado, la influencia (baja en este caso) de la distancia euclídea añade ruido a la red debido a que la resta de por sí ya añadía una información similar, lo que disminuye su *accuracy*.	1 descriptor de 128 desde 2 captaciones restados entre sí $(1 * 128 = 128)$ y distancia euclídea entre dichos descriptores (1)	88.2 %	0.417
8	Parecido al modelo 7, salvo que se modifica levemente el Teorema de Kolmogorov. Además, se introduce la distancia euclídea en una capa posterior para que influya más en la red. Como se puede analizar en el *accuracy* del sistema, una mayor influencia perjudica a la red por añadir la información redundante explicada en la descripción del modelo 7.	1 descriptor de 128 desde 2 captaciones restados entre sí $(1 * 128 = 128)$ y distancia euclídea entre dichos descriptores (1)	86.8 %	0.342
9	Similar al modelo 6, con la diferencia de mantener los 2 descriptores originales tras restarlos usando el valor absoluto. El resultado se ve levemente perjudicado por la leve relación generadora de ruido para la red que existe entre el resultado de una resta de descriptores y los descriptores originales.	1 descriptor de 128 desde 2 captaciones restados entre sí $(1 * 128 = 128)$ y los dos descriptores originales $(2 * 128 = 256)$	92.6 %	0.240

Tabla 7.7: Sistemas de verificación basados en redes neuronales y resultados de su evaluación

[4]Tanto al *accuracy* como el *loss* fue calculado usando el conjunto de test (ver apartado 7.2.2.1). En el README.md del repositorio [30], se tienen además las distintas tasas de acierto y errores para los conjuntos de entrenamiento y validación.

[5]Teorema de Kolmogorov: adaptado a las redes neuronales [93], dada una función continua $f : I^N \rightarrow R^M$ donde I se encuentra en un intervalo cercano a [0,1] (como resulta en esta implementación), f puede ser representado por una red neuronal teniendo N neuronas de entrada, $2N + 1$ neuronas en la capa oculta y M salidas.

7.3 La salida: verificación con el método `is_same_person (video_1, video_2)`

Según la Universidad Nacional de Córdoba, un sistema se define [94] como:

> *Un conjunto de partes o elementos (subsistemas) relacionados entre sí mediante una cadena de actividades que buscan alcanzar un objetivo planteado. Reciben entradas del medio en forma de datos, energía, información, y proveen salidas al entorno en carácter de información, materia, etc.*

A lo largo del desarrollo, se ha ido describiendo cada uno de los módulos que componen el presente sistema desde que se comenzó con su propia entrada (ver sección 5.1).

Recordando la definición planteada, y teniendo en mente el módulo `verifier` (orientado a verificación mediante umbral) y `verifiermodel` (para verificación mediante redes neuronales), se procede a visualizar el fragmento de código que, dados dos IDs de vídeos, es capaz de verificar, con una confianza calculada, si ambos contienen o no a la misma identidad:

```
def is_same_person(self, id_video_1, id_video_2):
    [...] # Se calcula la matriz de distancias con los descriptores de los dos vídeos a comparar
    same_person_samples = numpy.where(distance_matrix < self.threshold)[0].shape[0]
    same_person_rate = same_person_samples / distance_matrix.shape[0]
    if same_person_rate >= self.confidence:
        return {'is_same_person': True,
                'confidence': round(same_person_rate, 3),
                'distance_matrix': distance_matrix}

    return {'is_same_person': False,
            'confidence': round(1 - same_person_rate, 3),
            'distance_matrix': distance_matrix}
```

Fragmento de código 7.12: Verificación con el método `is_same_person(video_1, video_2)` en el módulo `verifier` [Fuente]

Una vez se han observado los resultados de ambos tipos de verificación analizando cómo el modelo 10 descrito en la tabla 7.5 por parte de la verificación mediante umbral y el modelo 6 descrito en la tabla 7.7 por parte de la verificación mediante redes neuronales han sido los que han obtenido mejores resultados, se propone a continuación y para finalizar con el presente capítulo de la memoria del Trabajo de Fin de Grado, ampliar la tabla 5.2 con la tasa de acierto y el error obtenido con dichos sistemas para los conjuntos de *AveRobot* propuestos:

						Con redes neuronales		Con umbral de distancia	
Subconjunto de vídeos	**Cámara del piso 1**	**Cámara del piso 2**	**Cámara del piso 3**	**Localización**	**Dificultad de verificación**	**Loss**	**Accuracy**	**EER**	**Accuracy**
A	2	3	8	Aleatoria (escalera, ascensor o pasillo)	Difícil	0.884	64.9 %	0.084	95.12 %
B	2	5	8	Ascensor	Normal	0.183	93.3 %	0.017	100 %
C	2	5	8	Pasillo	Muy difícil	0.790	59.2 %	0.092	91.46 %
D	2	5	8	Escalera	Muy difícil	0.624	65.5 %	0.103	92.68 %

Tabla 7.8: Subconjuntos de vídeos de *AveRobot* utilizados en el proyecto con sus resultados haciendo uso del modelo 6 de la tabla 7.7 y del modelo 10 de la tabla 7.5 respectivamente

Capítulo 8

Conclusiones

8.1 Aportaciones

Como se mencionó someramente en la Introducción, este proyecto ha tenido diversos objetivos a la hora de aportar una evolución o avance en diferentes ámbitos. En concreto, se deben destacar las aportaciones que se han querido realizar en los siguientes tres entornos:

En el **entorno socio-económico** se ha aportado:

- Un sistema capaz de añadir un nivel alto de seguridad en escenarios multinivel (es decir, de varios pisos).
- La capacidad de adquirir este sistema a un precio reducido, dado a que únicamente requiere de cámaras y sistemas de una calidad media para obtener buenos resultados (ver calidad de las cámaras utilizadas en *AveRobot* y las utilizadas en la evaluación del proyecto en las tablas 5.1 y 5.2) dado a que el resto de elementos usados son gratuitos o de código abierto (*open-source*).

En el **entorno técnico** se ha aportado:

- La facilidad de implementación del sistema a raíz de unos conocimientos básicos de instalaciones informáticas (ver Manual de Usuario entregado junto a esta memoria).

En el **entorno científico** se ha aportado:

- Siendo este el principal entorno que se ha querido tratar, un estudio sobre la verificación en escenarios multinivel. Dicho estudio implica un análisis, diseño y desarrollo de cada una de las fases que implica el proyecto con su correspondiente evaluación.
- Unos resultados satisfactorios usando el rostro como fuente de la verificación, alejándose de sensores biométricos fisiológicos ya estudiados en profundidad por la comunidad científica tales como la huella digital y el reconocimiento de iris.

8.2 Comparativa de resultados existentes de *AveRobot*

Finalizando con la memoria de este Trabajo de Fin de Grado, se desea resaltar una pequeña comparativa de los resultados obtenidos con el proceso de verificación en escenarios multinivel con presencia de robots asistentes respecto a los artículos relacionados con el conjunto de datos *AveRobot* descritos en el Estado del arte [2] [43] que trataban de lograr dicho proceso de verificación con la mayor tasa de acierto posible.

En concreto, se recuerda que en dichos artículos se logró un EER mínimo de 0.301 y de 0.312 respectivamente (ver figuras 4.1 y 4.4). Teniendo en cuenta los resultados obtenidos en la tabla 7.8, se tiene que en el conjunto donde la localización es aleatoria (pudiendo originarse en la escalera, en el ascensor o en el pasillo indistintamente) y donde incluso se hace uso de cámaras con efecto de entrelazado (ver sección 6.1.2), se logra un EER de 0.084.

Este resultado, por tanto, es un indicador de que se ha logrado reducir el error del proceso de verificación respecto a dichos artículos en un **72.09 %** y en un **73.08 %** respectivamente.

8.3 Trabajo futuro

No obstante, aunque se trate de una mejora destacable, la estructura del sistema de verificación propuesto en el proyecto sugiere varias rutas donde avanzar con el objetivo de lograr mayor robustez y tasa de acierto aún. A continuación, se plantean cuatro de ellas:

- En primer lugar, como se realizó en uno de los artículos mencionados en el Estado del arte [43], se podría plantear el uso de varias modalidades biométricas en lugar de emplear únicamente los rostros. Por ejemplo, se demostró cómo la verificación mediante voz combinada con la facial puede transformarse en mejoras de tasa de acierto en el sistema respecto al uso de cada una de ellas por separado.
- En segundo lugar, también se podría analizar el uso de una verificación mediante redes neuronales (ver sección 7.2) que hiciera uso de descriptores generados por VGGFace2 - ResNet50, los cuales otorgan actualmente los mejores resultados del proyecto en lugar de los desarrollados por FaceNet (Inception ResNet v1). Se recuerda que varios modelos basados en verificación con redes neuronales lograron superar a algunos de los sistemas viables que hacían uso de un umbral estático (ver apartado 7.1.3) basados en FaceNet (Inception ResNet v1), así que podría existir una mejora si se adaptaran dichos modelos a VGGFace2 - ResNet50.
- En tercer lugar, se podría estudiar el uso de otras herramientas de alineamiento o normalización de detecciones de rostros (ver sección 6.2.3). Si bien es cierto que la calidad de imágenes de muchas captaciones/grabaciones de *AveRobot* dificulta una normalización en la que no se pierda aún más calidad, se podría investigar los últimos sistemas de alineamiento en busca de uno que permitiera realizar dicha normalización facial al mismo tiempo que no existe (o existe levemente) una pérdida de definición en la imagen.

- En cuarto y último lugar, se podría tratar de trasladar el sistema de verificación desarrollado a lo largo del proyecto para que tratara con grabaciones de otro conjunto de datos ubicado en un escenario multinivel, en busca de obtener un mayor número de resultados. No obstante, como se advirtió en el artículo de *AveRobot* [2], el conjunto de datos usado en este proyecto se trata de uno de los más complicados de tratar actualmente.

8.4 Reflexión personal

Para concluir, también se quería añadir una pequeña reflexión filosófica de lo que ha supuesto un trabajo que concluye mi paso por el Grado en Ingeniería Informática.

Más allá de encontrarme muy satisfecho por haber logrado una tasa de acierto tan alta respecto a la que se presentaron en los artículos de *AveRobot* [2] [43] y una posible base para futuras investigaciones del mismo entorno, he de decir que este Trabajo de Fin de Grado me ha servido para tratar de sintetizar la gran mayoría de conocimientos adquiridos a lo largo de la carrera: desde los cimientos matemáticos, pasando por una arquitectura modular y ordenada aprendida desde la Ingeniería del Software y llegando a los paradigmas relacionados con la inteligencia artificial y la visión por computador enseñados en la especialidad de Computación, entre otros.

Recordando a la cita textual de la Introducción de Eliezer Yudkowsky [1]:

> *By far the greatest danger of Artificial Intelligence is that people conclude too early that they understand it.*

Lo que se ha tratado de lograr a lo largo del proyecto ha sido ir más allá del uso de distintas herramientas para lograr obtener una evaluación de los resultados. Se ha tratado de estudiar cada uno de los componentes del sistema de verificación, su funcionamiento, su objetivo y su razón de implementación. Se ha tratado de aprender como estudiante antes de dejar el aprendizaje al propio sistema.

Quizá el mayor peligro de la inteligencia artificial nunca estuvo en el poder que adquiere cada día. Quizá el riesgo estuvo en su uso. Quizá la amenaza siempre ha estado en usarla sin entender lo que realmente ocurría en su infraestructura.

Y si bien aún queda mucho por aprender y muchos avances por realizar, no puedo dejar de estar orgulloso de poder llevar conmigo una curiosidad constante en este campo tecnológico.

Y eso, me aporta mucho.
Gracias a todos los que colaboraron y colaborarán en hacerlo posible.

Bibliografía

[1] Eliezer Yudkowsky. «Artificial Intelligence as a Positive and Negative Factor in Global Risk». En: *Global Catastrophic Risks* (2008).

[2] Mirko Marras. y col. «AveRobot: An Audio-visual Dataset for People Re-identification and Verification in Human-Robot Interaction». En: *Proceedings of the 8th International Conference on Pattern Recognition Applications and Methods - Volume 1: ICPRAM,* INSTICC. SciTePress, 2019, págs. 255-265. ISBN: 978-989-758-351-3. DOI: 10.5220/0007690902550265.

[3] Universidad de Las Palmas de Gran Canaria. *Página oficial de AveRobot.* URL: http://mozart.dis.ulpgc.es/averobot/. (accedido: 04.07.2020).

[4] PSF. *Página oficial de Python.* URL: https://www.python.org/. (accedido: 24.05.2020).

[5] PSF. *Información sobre PSF (Python Software Foundation).* URL: https://www.python.org/psf-landing/. (accedido: 24.05.2020).

[6] PSF. *PSF License Agreement for Python 3.8.3.* URL: https://docs.python.org/3/license.html#psf-license-agreement-for-python-release. (accedido: 24.05.2020).

[7] PSF. *Python Package Index (PyPI).* URL: https://pypi.org/. (accedido: 24.05.2020).

[8] Project Jupyter. *Página oficial de Project Jupyter.* URL: https://jupyter.org/. (accedido: 24.05.2020).

[9] Matt Cone. *Markdown Guide.* URL: https://www.markdownguide.org/. (accedido: 24.05.2020).

[10] Linus Torvalds. *Git.* URL: https://git-scm.com/. (accedido: 24.05.2020).

[11] FSF. *GNU - General Public License v2.* URL: https://www.gnu.org/licenses/old-licenses/gpl-2.0.html. (accedido: 24.05.2020).

[12] JetBrains. *IntelliJ IDEA.* URL: https://www.jetbrains.com/es-es/idea/. (accedido: 24.05.2020).

[13] JetBrains. *Página oficial de JetBrains.* URL: https://www.jetbrains.com/. (accedido: 24.05.2020).

[14] JetBrains. *IntelliJ IDEA: Diferencias entre versiones.* URL: https://www.jetbrains.com/es-es/idea/features/editions_comparison_matrix.html. (accedido: 24.05.2020).

[15] Google Brain. *Página oficial de TensorFlow.* URL: https://www.tensorflow.org/. (accedido: 12.06.2020).

[16] Google. *Información sobre Google LLC.* URL: https://about.google/. (accedido: 23.05.2020).

[17] Keras team. *Página oficial de Keras.* URL: https://keras.io/. (accedido: 11.06.2020).

[18] OpenCV team. *Página oficial de OpenCV.* URL: https://opencv.org/. (accedido: 05.06.2020).

[19] Olli-Pekka Heinisuo. *Biblioteca 'cv2'*. URL: https://github.com/skvark/opencv-python. (accedido: 04.06.2020).
[20] Davis E. King. *Página oficial de Dlib*. URL: http://dlib.net/. (accedido: 07.06.2020).
[21] Davis E. King. *Repositorio de Dlib*. URL: https://github.com/davisking/dlib. (accedido: 07.06.2020).
[22] Davis E. King. *Repositorio de modelos de Dlib*. URL: https://github.com/davisking/dlib-models. (accedido: 07.06.2020).
[23] Kaipeng Zhang y col. «Joint Face Detection and Alignment Using Multitask Cascaded Convolutional Networks». En: *IEEE Signal Processing Letters* 23.10 (oct. de 2016), págs. 1499-1503. ISSN: 1558-2361. DOI: 10.1109/lsp.2016.2603342. URL: http://dx.doi.org/10.1109/LSP.2016.2603342.
[24] David Sandberg. *Repositorio de implementación de MTCNN en FaceNet*. URL: https://github.com/davidsandberg/facenet/tree/master/src/align. (accedido: 07.06.2020).
[25] Iván de Paz Centeno. *Repositorio de implementación de MTCNN para Keras*. URL: https://github.com/ipazc/mtcnn. (accedido: 07.06.2020).
[26] Atlassian. *Trello*. URL: https://trello.com/. (accedido: 22.05.2020).
[27] Atlassian. *Página oficial de Atlassian*. URL: https://www.atlassian.com/. (accedido: 22.05.2020).
[28] Kanbanize. *Información sobre tableros Kanban*. URL: https://kanbanize.com/es/recursos-de-kanban/primeros-pasos/que-es-tablero-kanban. (accedido: 22.05.2020).
[29] Microsoft. *GitHub*. URL: https://github.com. (accedido: 22.05.2020).
[30] Kevin Rosales. *Repositorio del código programado para el TFG*. URL: https://github.com/kevinrosalesdev/tfg. (accedido: 28.06.2020).
[31] Google. *Google Drive*. URL: https://www.google.com/drive/. (accedido: 23.05.2020).
[32] Google. *Google Hangouts*. URL: https://hangouts.google.com/. (accedido: 23.05.2020).
[33] Microsoft. *Microsoft Teams*. URL: https://www.microsoft.com/en/microsoft-365/microsoft-teams/group-chat-software. (accedido: 23.05.2020).
[34] Microsoft. *Página oficial de Microsoft*. URL: https://www.microsoft.com/. (accedido: 23.05.2020).
[35] Overleaf. *Editor LaTeX de Overleaf*. URL: https://www.overleaf.com/. (accedido: 23.05.2020).
[36] LaTeX. *LaTeX Project*. URL: https://www.latex-project.org/. (accedido: 23.05.2020).
[37] Overleaf. *Información sobre Overleaf*. URL: https://www.overleaf.com/about. (accedido: 23.05.2020).
[38] WeTransfer. *WeTransfer*. URL: https://wetransfer.com/. (accedido: 23.05.2020).
[39] WeTransfer. *Sobre WeTransfer*. URL: https://wetransfer.com/about. (accedido: 23.05.2020).
[40] EII. *Información sobre TFT*. URL: https://www.eii.ulpgc.es/tb_university_ex/?q=tft. (accedido: 24.05.2020).
[41] Yujiang Wang y col. *A real-time and unsupervised face Re-Identification system for Human-Robot Interaction*. 2018. arXiv: 1804.03547 [cs.CV].
[42] Anurag Chowdhury y col. «MSU-AVIS dataset: Fusing Face and Voice Modalities for Biometric Recognition in Indoor Surveillance Videos». En: *In Proceeding of International Conference on Pattern Recognition*. Beijing, China, ago. de 2018.
[43] Mirko Marras y col. «Deep Multi-biometric Fusion for Audio-Visual User Re-Identification and Verification». En: *Pattern Recognition Applications and Methods*. Ed. por Maria

De Marsico, Gabriella Sanniti di Baja y Ana Fred. Cham: Springer International Publishing, 2020, págs. 136-157. ISBN: 978-3-030-40014-9.

[44] Igor Rodriguez y col. «Personal Guides: Heterogeneous Robots Sharing Personal Tours in Multi-Floor Environments». En: *Sensors* 20.9 (2020). ISSN: 1424-8220. DOI: 10.3390/s20092480. URL: https://www.mdpi.com/1424-8220/20/9/2480.

[45] RSAIT - Robotika eta Sistema Autonomoen Ikerketa Taldea. *Personal guides: heterogeneous robots sharing personal tours in multi-floor environments*. URL: https://www.youtube.com/watch?v=5c4RDQg5Rsc. (accedido: 03.07.2020).

[46] Sefik Ilkin Serengil. *deepface*. 2020. URL: https://github.com/serengil/deepface.

[47] N. Dalal y B. Triggs. «Histograms of oriented gradients for human detection». En: 1 (2005), 886-893 vol. 1.

[48] Qiong Cao y col. *VGGFace2: A dataset for recognising faces across pose and age*. 2017. arXiv: 1710.08092 [cs.CV].

[49] Florian Schroff, Dmitry Kalenichenko y James Philbin. «FaceNet: A unified embedding for face recognition and clustering». En: *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)* (jun. de 2015). DOI: 10.1109/cvpr.2015.7298682. URL: http://dx.doi.org/10.1109/CVPR.2015.7298682.

[50] Chi-Feng Wang. *How Does A Face Detection Program Work? (Using Neural Networks)*. URL: https://towardsdatascience.com/how-does-a-face-detection-program-work-using-neural-networks-17896df8e6ff. (accedido: 07.06.2020).

[51] Davis E. King. *Código de ejemplo para DLIB - HOG*. URL: http://dlib.net/face_detector.py.html. (accedido: 07.06.2020).

[52] Adrian Rosebrock. *Histogram of Oriented Gradients and Object Detection*. URL: https://www.pyimagesearch.com/2014/11/10/histogram-oriented-gradients-object-detection/. (accedido: 07.06.2020).

[53] Davis E. King. *Código de ejemplo para DLIB - MMOD*. URL: http://dlib.net/cnn_face_detector.py.html. (accedido: 10.06.2020).

[54] Davis E. King. *Max-Margin Object Detection*. 2015. arXiv: 1502.00046 [cs.CV].

[55] Guillermo Gallud Baños. *Reconocimiento de emociones humanas y su aplicación a la Robótica Social*. 2019.

[56] David Sandberg. *Implementación de FaceNet para TensorFlow*. URL: https://github.com/davidsandberg/facenet. (accedido: 11.06.2020).

[57] Hiroki Taniai. *Implementación de FaceNet para Keras*. URL: https://github.com/nyoki-mtl/keras-facenet. (accedido: 11.06.2020).

[58] David Sandberg. *Implementación de FaceNet para TensorFlow - Arquitectura*. URL: https://github.com/davidsandberg/facenet/blob/master/src/models/inception_resnet_v1.py. (accedido: 11.06.2020).

[59] Yandong Guo y col. *MS-Celeb-1M: A Dataset and Benchmark for Large-Scale Face Recognition*. 2016. arXiv: 1607.08221 [cs.CV].

[60] VGG. *Página oficial de Visual Geometry Group de la Universidad de Oxford*. URL: http://www.robots.ox.ac.uk/~vgg/. (accedido: 31.05.2020).

[61] Universidad de Oxford. *Página oficial de la Universidad de Oxford*. URL: http://www.ox.ac.uk/#. (accedido: 31.05.2020).

[62] Kaiming He y col. *Deep Residual Learning for Image Recognition*. 2015. arXiv: 1512.03385 [cs.CV].

[63] Refik Can Malli. *Repositorio de la biblioteca keras-vggface*. URL: https://github.com/rcmalli/keras-vggface. (accedido: 31.05.2020).

[64] PSF. *Biblioteca 'os'*. URL: https://docs.python.org/3/library/os.html. (accedido: 04.06.2020).

[65] PSF. *Biblioteca 'random'*. URL: https://docs.python.org/3/library/random.html. (accedido: 04.06.2020).

[66] PSF. *Biblioteca 'datetime'*. URL: https://docs.python.org/3/library/datetime.html. (accedido: 04.06.2020).

[67] PSF. *Biblioteca 'glob'*. URL: https://docs.python.org/3/library/glob.html. (accedido: 04.06.2020).

[68] PSF. *Biblioteca 'pathlib'*. URL: https://docs.python.org/3/library/pathlib.html. (accedido: 04.06.2020).

[69] PSF. *Biblioteca 're'*. URL: https://docs.python.org/3/library/re.html. (accedido: 04.06.2020).

[70] NumPy Team. *Biblioteca 'numpy'*. URL: https://numpy.org/. (accedido: 04.06.2020).

[71] Alex Clark. *Biblioteca 'pillow'*. URL: https://pillow.readthedocs.io/en/stable/. (accedido: 04.06.2020).

[72] Samsung. *Diferencia entre exploración entrelazada y exploración progresiva*. URL: https://www.samsung.com/latin/support/tv-audio-video/what-is-interlaced-and-progressive-video/. (accedido: 05.06.2020).

[73] Kushashwa Ravi Shrimali. *Explicación de medida BRISQUE*. URL: https://www.learnopencv.com/image-quality-assessment-brisque/. (accedido: 06.06.2020).

[74] Anish Mittal, Anush Moorthy y Alan Bovik. «No-Reference Image Quality Assessment in the Spatial Domain». En: *IEEE transactions on image processing : a publication of the IEEE Signal Processing Society* 21 (ago. de 2012). DOI: 10.1109/TIP.2012.2214050.

[75] MathWorks. *Explicación breve de máquinas de vectores de soporte (SVM)*. URL: https://www.mathworks.com/discovery/support-vector-machine.html. (accedido: 06.06.2020).

[76] Ricardo Ocampo. *Biblioteca 'image-quality'*. URL: https://github.com/ocampor/image-quality. (accedido: 06.06.2020).

[77] OpenCV team. *Tutorial Canny Edge Detection*. URL: https://docs.opencv.org/trunk/da/d22/tutorial_py_canny.html. (accedido: 06.06.2020).

[78] Konstantin Lepa. *Biblioteca 'termcolor'*. URL: https://pypi.org/project/termcolor/. (accedido: 07.06.2020).

[79] The Matplotlib development team. *Biblioteca 'matplotlib'*. URL: https://matplotlib.org/. (accedido: 07.06.2020).

[80] PSF. *Biblioteca 'pickle'*. URL: https://docs.python.org/3/library/pickle.html. (accedido: 07.06.2020).

[81] Vahid Kazemi y Josephine Sullivan. «One Millisecond Face Alignment with an Ensemble of Regression Trees». En: (jun. de 2014). DOI: 10.13140/2.1.1212.2243.

[82] Adrian Rosebrock. *Facial landmarks with dlib, OpenCV, and Python*. URL: https://www.pyimagesearch.com/2017/04/03/facial-landmarks-dlib-opencv-python/. (accedido: 10.06.2020).

[83] Christos Sagonas y col. «300 Faces In-The-Wild Challenge: database and results». En: *Image and vision computing* 47 (mar. de 2016), págs. 3-18. ISSN: 0262-8856. DOI: 10.1016/j.imavis.2016.01.002.

[84] Adrian Rosebrock. *Face Alignment with OpenCV and Python*. URL: https://www.pyimagesearch.com/2017/05/22/face-alignment-with-opencv-and-python/. (accedido: 10.06.2020).

[85] Modesto Fernando Castrillón Santana. *Repositorio de ENCARA2*. URL: https://github.com/otsedom/ENCARA2/. (accedido: 18.06.2020).

[86] F. Pedregosa y col. «Scikit-learn: Machine Learning in Python». En: *Journal of Machine Learning Research* 12 (2011), págs. 2825-2830.

[87] Wes McKinney. *Página oficial de Pandas*. URL: https://pandas.pydata.org/. (accedido: 13.06.2020).

[88] Laurent LAPORTE. *Biblioteca 'deprecated'*. URL: https://github.com/tantale/deprecated. (accedido: 13.06.2020).

[89] PSF. *Biblioteca 'itertools'*. URL: https://docs.python.org/3/library/itertools.html. (accedido: 14.06.2020).

[90] Bob Lockhart. *In Biometrics, Which Error Rate Matters?* URL: https://tractica.omdia.com/biometrics/in-biometrics-which-error-rate-matters/. (accedido: 14.06.2020).

[91] François Chollet. *The Sequential model*. URL: https://keras.io/guides/sequential_model/. (accedido: 16.06.2020).

[92] François Chollet. *The Functional API*. URL: https://keras.io/guides/functional_api/. (accedido: 16.06.2020).

[93] Zenon Waszczysznk. *Neural Networks in the Analysis and Design of Structures*. Springer-Verlag Wien, 1999.

[94] Universidad Nacional de Córdoba. *¿Qué entendemos por sistema?* URL: http://aotgu.eco.catedras.unc.edu.ar/sistemas-de-organizacion-y-contexto/un-sistema-abierto/que-entendemos-por-sistema/. (accedido: 17.06.2020).