ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA INFORMÁTICA
INGENIERÍA DEL SOFTWARE

Diseño y desarrollo de una aplicación web para alojar algoritmos de reconocimiento de enfermedades dermatológicas

Design and development of a web application to host dermatological disease recognition algorithms

Realizado por
Pablo Gordillo Sánchez

Tutorizado por
Enrique Domínguez Merino
Miguel Ángel Molina Cabello

Departamento
Lenguajes y Ciencias de la Computación

UNIVERSIDAD DE MÁLAGA
MÁLAGA, SEPTIEMBRE DE 2022

Fecha defensa:

Resumen

Los modelos de aprendizaje automático son ampliamente utilizados por la comunidad informática desde hace años para realizar tareas de clasificación de imágenes. Una de sus grandes ventajas frente a otros tipos de algoritmos de clasificación es su capacidad para resolver problemas muy distintos sin cambiar su arquitectura, mediante entrenamientos en los que se procesan miles de imágenes. En el ámbito de las imágenes clínicas, su uso es cada vez más extenso y los resultados más precisos. Poner estos algoritmos a disposición de cualquier persona o profesional médico puede ser muy beneficioso para la población.

En este trabajo se ha creado una aplicación web que permite a sus usuarios publicar, entrenar y usar sus propios modelos de *deep learning* de clasificación de imágenes de una manera sencilla y segura, centrándose en el ámbito de la clasificación de enfermedades dermatológicas. La aplicación pone a disposición de los usuarios *datasets* de imágenes para que puedan realizar el entrenamiento de sus modelos aunque no tengan el acceso a las propias imágenes que lo componen.

Además, se han diseñado una serie de modelos de *deep learning* para su uso público en la aplicación. Para su diseño se ha realizado un estudio sobre la diferenciación de tumores malignos en la piel, atendiendo a las recomendaciones del ISIC (International Skin Imaging Collaboration) y a los paradigmas del aprendizaje automático.

Palabras clave: Aplicación web, *Deep learning*, Clasificación de imágenes, ISIC

Abstract

Automatic learning model are widely used by the computer science community for years to accomplish image classification tasks. One of their big advantages over other kinds of image classification algorithm is their ability to solve very diverse problems without changing their architecture, by trainings in which thousands of images are processed. In the field of clinical images their use is becoming more common and are achieving more precise results. Making these algorithms available for anyone or even medical professionals could be very beneficial for all.

In this project it has been created a web application that allows their users to publish, train and use their own image classification deep learning models in an easy and secure way, focusing on the classification of dermatological diseases field. The application gives users access to image datasets to train their models without giving access to the proper images that compose it.

Besides, there have been design some deep learning models for their public use within the application. To design them, it has been researched the differentiation of malignant tumors on the skin, following the ISIC (International Skin Imaging Collaboration) recommendations and the automatic learning paradigms.

Keywords: Web application, Deep learning, Image classification, ISIC

Índice

Resumen 1

Abstract 2

Índice 3

Introducción 5

1.1 Motivación 5

1.2 Objetivos 8

1.3 Estado del arte 8

1.4 Estructura de la memoria 9

1.5 Metodología 11

1.5.1 Cross Industry Standard Process for Data Mining (CRISP-DM) 11

1.5.2 Metodologías Ágiles 13

1.6 Tecnologías utilizadas 14

1.6.1 Flask 14

1.6.2 React.JS 15

1.6.3 Tensorflow 15

1.5.4 Firebase 16

1.5.5 Postman 17

1.5.6 GitLab 17

Modelos de clasificación 19

2.1 Algoritmos de clasificación de imágenes 19

2.1.1 Árboles de decisión 19

2.1.2 Máquinas de vectores de soporte 20

2.1.3 Redes neuronales y *deep learning* 21

2.2 Requisitos de los modelos de clasificación de enfermedades 23

2.2.1 Estudio inicial del dataset 24

2.2.2 Preparación del dataset para el entrenamiento 28

2.2.3 Función de coste. *Binary Cross-Entropy* 29

2.2.4 Métricas de evaluación. Curva ROC y PR 30

2.3 Desarrollo de los modelos de clasificación 39

2.3.1 Arquitectura del modelo 39

2.3.2 Proceso de entrenamiento 41

2.3.3 Técnicas de mejora del *dataset* 43

2.4 Selección de umbral 47

Aplicación web 49

3.1 Requisitos de la aplicación **49**
3.1.1 Requisitos funcionales 50
3.1.2 Requisitos no funcionales 51
3.1.3 Casos de uso 51
3.2 Diseño de la aplicación **58**
3.2.1 Arquitectura 58
3.2.2 Protocolo para la creación y uso de los clasificadores 62
3.2.3 Diseño de base de datos 64
3.2.4 Navegación 66
3.3 Implementación y pruebas **68**
3.3.1 Comunicación cliente-servidor 68
3.3.2 Autenticación 70
3.3.3 Envío, almacenamiento y uso de clasificadores 72
3.3.4 Pruebas 76
Conclusiones 81
4.1 Conceptos desarrollados 82
4.2 Líneas futuras 83
Referencias 85
Manual de Instalación 87
Instalar servidor web con Flask **87**
Instalar cliente web con React **87**
Manual de Usuario 89
1. Registrarse o hacer log in/log out **89**
2. Visualizar información de un clasificador **90**
3. Publicar un clasificador **91**
4. Clasificar una imagen **91**
5. Entrenar clasificador **92**

1

Introducción

1.1 Motivación

La dermatología es la rama de la medicina que se encarga del tratamiento de enfermedades en la piel. Este tipo de enfermedades son muy comunes, afectando a aproximadamente un cuarto de la población mundial. Sin embargo, el 80% de los afectados por este tipo de enfermedades no busca atención médica. Se cree que en los próximos años la conciencia pública sobre estas enfermedades aumentará, incrementando a su vez la demanda de tratamientos en un sistema médico que actualmente no está preparado para asumir ese volumen de trabajo.

Realizar esta generalización sobre la demanda de los servicios dermatológicos es complicado, ya que existe una gran variedad de enfermedades que pueden afectar a la piel. No obstante, según datos de Reino Unido, el 70% de la carga de trabajo para la atención primaria y secundaria se centra en 9 categorías de problemas dermatológicos: cáncer de piel, acné, eccema, psoriasis, verrugas virales, enfermedades infecciosas, tumores benignos y lesiones vasculares, ulceras y dermatitis. Hoy en día, este diagnóstico de la enfermedad se basa en un estudio de los síntomas (principalmente, cómo es la enfermedad visualmente), distribución anatómica (dónde se encuentra) y otros factores como un examen histórico, causas genéticas, etc.

Por tanto, vemos claro que, para asegurar un buen tratamiento de la enfermedad es muy importante realizar un buen diagnóstico de la misma, ya que existe una gran cantidad de variantes. Además, muchas veces no se tiene acceso a datos históricos o genéticos del paciente, por lo que se debe realizar el diagnóstico únicamente viendo los síntomas de la enfermedad (su imagen). Así, se ha planteado el uso de distintos algoritmos informáticos para realizar el diagnóstico (la clasificación) de estas enfermedades a partir de únicamente imágenes de ellas. Entre estos algoritmos destacan los algoritmos de *deep learning* (o aprendizaje profundo), que basan su correcto funcionamiento en una fase de entrenamiento previa, para la cual se necesitan grandes cantidades de imágenes ya clasificadas. Una solución aparente a este problema es poner a disposición del usuario una aplicación web que permita utilizar este tipo de algoritmos ya entrenados para que clasifique sus propias imágenes, o incluso permitir a los algoritmos de los usuarios entrenarse utilizando estos grandes *datasets* (o conjuntos de datos) de imágenes médicas.

Aun contando con estos grandes volúmenes de imágenes, la resolución de este problema utilizando modelos de *deep learning* no es trivial. Hay multitud de factores a tener en cuenta a la hora de diseñar uno de estos modelos para realizar una correcta clasificación de todas estas enfermedades, y se necesita una capacidad de cómputo muy grande para entrenar estos modelos. Por ello, este trabajo se centra en desarrollar modelos que permitan distinguir una de estas enfermedades del resto, el melanoma.

El melanoma es una enfermedad por la que se forman células cancerosas en los melanocitos (células que dan color a la piel). Esta enfermedad es el tipo de cáncer de piel más letal que existe, y su incidencia va en aumento durante las últimas décadas. Si se diagnostica el melanoma antes de que traspase la capa exterior de la piel, su extracción quirúrgica es muy sencilla y la tasa de supervivencia es del 98% durante los próximos 5 años. Aun así, debido a su parecido con el *novus* (lunar común) en sus primeras etapas de crecimiento, se siguen diagnosticando casos avanzados de melanoma. En 2020 se estima que murieron 57.000 personas a causa de esta enfermedad en todo el mundo.

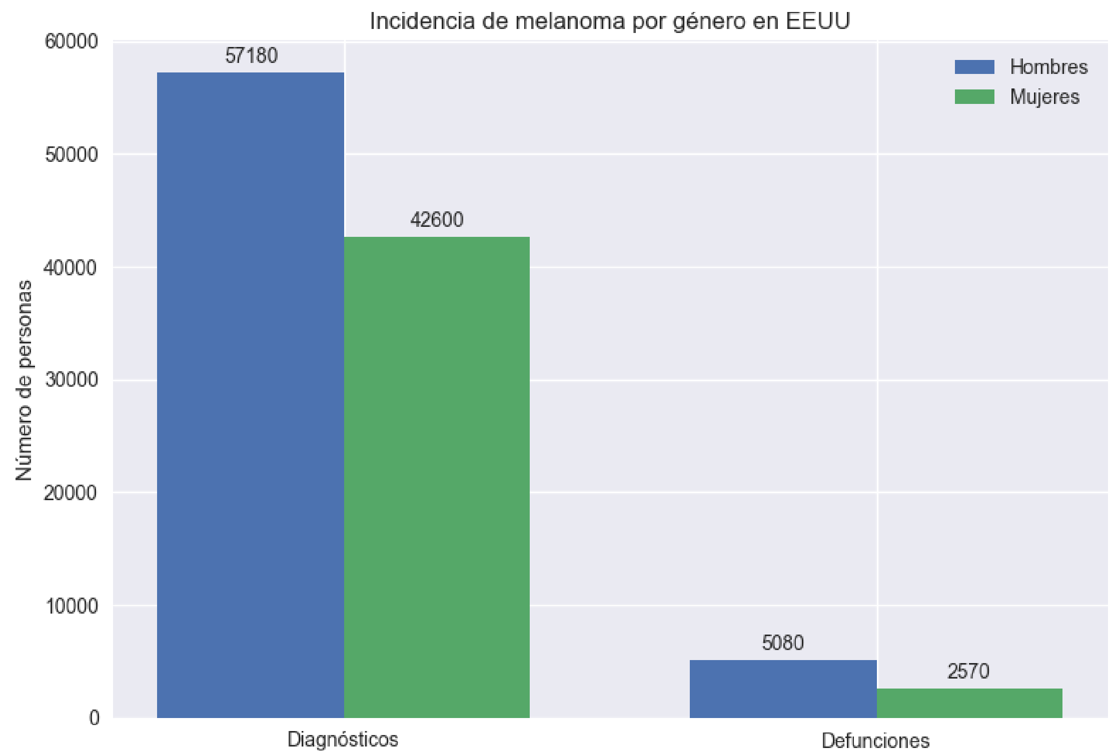


Figura 1.1 Previsión de la incidencia de melanoma en EE. UU. por género para el año 2022.

Por desgracia, realizar un diagnóstico de esta enfermedad a simple vista, incluso para un profesional, es muy difícil. En niños, donde la enfermedad es menos propensa a aparecer, se estima que se realizan 500.000 biopsias para diagnosticar 400 melanomas al año únicamente en Estados Unidos. Este ratio tan descompensado entre casos de melanoma y otras enfermedades más benignas suponen un reto a la hora de diseñar métodos precisos de diagnóstico.

Ante estos datos, la comunidad científica y médica se ha puesto el objetivo de diseñar herramientas capaces de hacer un primer diagnóstico con la mayor precisión posible. Además, estas herramientas deben estar disponibles para la población, permitiendo realizar un autodiagnóstico rápido, que pueda llegar a ser rutinario en el día a día de cualquiera. Para ello, la herramienta predilecta han sido los modelos de *deep learning*, debido a su uso fácil y rápido una vez entrenados.

Por tanto, hemos decidido desarrollar una herramienta que pueda llegar a poner a disposición de la población estos modelos de clasificación.

1.2 Objetivos

Este TFG consiste en el diseño y desarrollo de una aplicación web que permita el uso de distintos algoritmos de reconocimiento de enfermedades dermatológicas. El objetivo de esta aplicación es poner a disposición de cualquier usuario distintas herramientas que le permita clasificar correctamente imágenes de enfermedades de la piel de una manera sencilla. La aplicación implementa un algoritmo de *deep learning* que cualquier usuario podrá utilizar para clasificar sus propias imágenes.

Para lograr estos objetivos, es muy importante realizar una labor de investigación previa de las metodologías de diagnóstico de estas enfermedades. Estas metodologías serán aplicadas para la elaboración de distintos modelos de redes neuronales, los cuales podrán ser utilizados desde la aplicación por cualquier usuario, aunque no cuente con conocimientos técnicos de la materia.

Para la implementación de la aplicación web, deberán recogerse los requisitos funcionales principales a implementar. A partir de estos requisitos se deberán definir la arquitectura de la aplicación y el protocolo a utilizar para poder conectar los modelos implementados por los usuarios con la aplicación.

1.3 Estado del arte

La clasificación de enfermedades dermatológicas es un campo que está actualmente en auge debido a los avances, tanto médicos como tecnológicos que se han realizado en la última década. La posibilidad de que cualquier persona pueda utilizar una aplicación móvil o web desde su *smartphone* ha hecho que se creen multitud de herramientas para realizar un diagnóstico rápido. Realizando una búsqueda en cualquier *store* de aplicaciones móviles, encontraremos numerosas aplicaciones que permiten realizar una tarea de diagnóstico de las enfermedades parecida a la que se pretende realizar con este TFG. Normalmente las empresas que desarrollan estas aplicaciones desarrollan sus propios algoritmos de clasificación, ya sea realizando su propia labor de investigación o subcontratando a empresas especializadas. Es el caso por ejemplo de IDerma con su aplicación Model Dermatology -Skin Disease.

Sin embargo, no es tan común encontrar aplicaciones enfocadas a que cualquier usuario pueda publicar y utilizar su propio clasificador o el de otros usuarios sin salir de ella. La primera referencia que tenemos de un sitio web que permita compartir algoritmos de cualquier tipo es GitHub. En esta web, un usuario puede descargar un proyecto que otro haya publicado, para poder ejecutarlo en su propia máquina. Además, GitHub ha terminado teniendo muchas funcionalidades de red social, que permiten seguir a otros usuarios, mandar mensajes o personalizar un perfil de usuario.

Por el otro lado, existen numerosas herramientas de computación en la nube, las cuales permiten a cualquier usuario subir un archivo en un lenguaje concreto y ejecutarlo en servidores ajenos. Un claro ejemplo de esto es paiza.io, una web que permite escribir en la propia interfaz del navegador el código que queramos ejecutar en los lenguajes que se permite.

Nuestra idea es realizar una combinación, a un nivel básico, de estas 3 aplicaciones de referencia en sus campos. Permitir que un usuario pueda publicar un modelo de clasificación y se mantenga guardado en el tiempo, como si fuese GitHub, pero permitiendo su ejecución sin que se requiera al usuario conocimiento técnicos. Esta idea no es fortuita. Es necesario implementar la ejecución de los modelos en un servidor web para asegurar que el entrenamiento de los modelos se realiza sin que se obtenga la información de las imágenes médicas que se utilizan.

Por tanto, podemos decir que a nivel comercial aún no existen aplicaciones que brinden todas estas funcionalidades al mismo tiempo.

1.4 Estructura de la memoria

La memoria se dividirá en 4 partes principales:

1. **Metodología y tecnologías utilizadas:** Esta primera parte estará compuesta por 2 subapartados, los cuales servirán como introducción previa al cuerpo de este TFG. Se detallarán las metodologías de desarrollo elegidas para la elaboración de

este trabajo (tanto de desarrollo web, como de minería de datos) y las tecnologías utilizadas para desarrollar cada parte de la aplicación.

2. **Modelos de clasificación:** En este capítulo se detalla todo el proceso realizado para crear los algoritmos de clasificación utilizables desde la aplicación. Se empezará detallando las alternativas de algoritmos de clasificación usadas en diagnóstico de enfermedades. Se continuará estudiando el conjunto de datos con el que se ha contado para realizar el entrenamiento de los modelos. Finalmente se describe la arquitectura de los modelos desarrollados y se realizará una evaluación de su rendimiento mediante el estudio de varias métricas.

3. **Aplicación web:** Este capítulo se centra en el diseño y desarrollo de la aplicación web en sí. Se comenzará recopilando los requisitos funcionales y no funcionales a implementar. Se continuará diseñando y detallando la arquitectura de la aplicación, tanto cliente como servidor web. Luego se desarrollará el protocolo para el uso de los métodos de clasificación de los usuarios, junto con la información a almacenar necesaria para su uso. También se diseñará el flujo de navegación entre las vistas de la aplicación. Tras esto se detallará la implementación de los requisitos más importantes de la aplicación, como la comunicación cliente-servidor y el proceso de autenticación de los usuarios. Por último, se mostrarán algunas pruebas desarrolladas.

4. **Conclusiones:** Este último capítulo expone un resumen final de la aplicación finalmente desarrollada, así como el conocimiento que el alumno ha obtenido durante la elaboración de este TFG. Este conocimiento es muy diverso, ya que se juntan campos muy distintos durante el desarrollo. Por último, se dan posibles líneas futuras que deberían realizarse para que la aplicación estuviese finalmente preparada para su uso a gran escala.

1.5 Metodología

Este TFG consta de dos partes claramente diferenciadas que requieren metodologías de desarrollo distintas. Para realizar el desarrollo de los modelos de clasificación se ha utilizado una metodología CRISP-DM, una metodología estándar de *data mining*. Para desarrollar la aplicación web se ha utilizado una metodología ágil, realizando distintas iteraciones sobre el trabajo. A continuación se detallan estas dos metodologías y cómo se han aplicado:

1.5.1 Cross Industry Standard Process for Data Mining (CRISP-DM)

CRISP-DM es una metodología descrita en 1999 como una guía a aplicar a las técnicas de *data* m*ining*. Describe un proceso análogo a las metodologías de desarrollo de software, dividiendo el proceso de minería de datos en fases. CRISP-DM entiende los procesos de minería de datos dentro de un contexto profesional, incluidos en un proyecto mayor del que son parte. Por tanto, incluye fases relacionadas con las necesidades del cliente y con el despliegue de los resultados obtenidos junto con el resto del proyecto. Estas son las 6 fases de las que consta esta metodología:

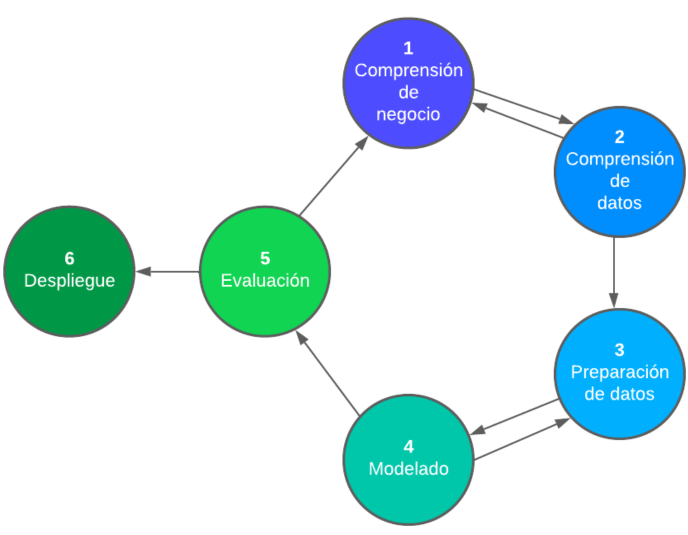


Figura 1.2 Fases de la metodología CRISP-DM.

1. **Comprensión del negocio**: Es una fase inicial, análoga al análisis de requisitos del desarrollo de software. Incluye tanto el proceso de identificación de requisitos (ya sea estudiando el contexto del problema o mediante el diálogo con un

cliente) y el proceso de comprensión y especificación de los requisitos. Para este trabajo, esta fase consiste en un proceso de investigación del problema a resolver.

2. **Estudio y comprensión de los datos**: En esta fase se estudian los datos de los que se dispone sobre el problema a resolver. Este estudio permite conocer la calidad de los mismo o determinar si serán suficientes para el problema a resolver. Para este trabajo, se han utilizado los datos disponibles del ISIC Challenge de 2020, el cual recaba imágenes de lesiones dermatologías. Este estudio ha permitido ajustar el alcance del problema a resolver para ajustarse a los datos de los que se dispone.
3. **Preparación de los datos**: Esta fase incluye realizar distintos procesos que preparen la información de la que se dispone para que puedan utilizarse con los modelos que se definan posteriormente. En esta fase, se han realizado distintas transformaciones, como ajustar las dimensiones de las imágenes o crear subconjuntos de imágenes para aplicarlos en distintas tareas.
4. **Modelado:** Esta es la fase principal del proceso, donde se recogen toda la información obtenida de las fases anteriores y se crean los modelos de minería de datos que puedan resolver el problema descrito. Para este trabajo, este modelado pasa por diseñar modelos de d*eep learning* entrenados con los datos ajustados del *dataset* de ISIC Challenge. De esta fase salen varios modelos que serán comparados más adelante.
5. **Evaluación**: Esta fase se centra en analizar los resultados obtenidos tras el modelado, comparando el rendimiento de los distintos modelos obtenidos y asegurándose de que se cumplen los requisitos del cliente. Muchas veces, los modelos desarrollados son solo una herramienta para obtener unos resultados sobre los datos con los que se cuenta. En ese caso se realizaría un estudio de la información nueva obtenida de los datos. Para este trabajo, el resultado del proceso de modelado son los propios modelos desarrollados, por lo que se compararán utilizando distintas métricas de evaluación.
6. **Despliegue**: Esta fase consiste en integrar los modelos desarrollados o el conocimiento obtenidos de estos en el resto del proyecto. En este trabajo, esta

fase consiste en definir cómo será la entrada y salida de los modelos para que puedan ser fácilmente utilizados desde la aplicación web.

Este proceso no es totalmente cíclico. Las fases de comprensión del negocio y de los datos suelen realizarse al mismo tiempo para asegurar que los datos de los que se dispone permiten realizar los requisitos del cliente o del problema a resolver. Además, muchas veces los propios modelos definidos en la fase 4 son utilizados para realizar procesados de los datos más complejos, los cuáles servirán en futuros modelos. Finalmente, la evaluación de los resultados puede concluir tanto en un resultado exitoso, que lleva a la fase 6 de este proceso, o en la necesidad de ajustar los requisitos iniciales, llevando a la fase 1.

1.5.2 Metodologías Ágiles

Estas metodologías se basan principalmente en iteraciones que intercalan fases de análisis de requisitos, diseño de software, implementación y pruebas. Estas iteraciones suelen comprender, cada una, una parte distinta del software a desarrollar, avanzando desde las funcionalidades más básicas o importantes a las más específicas. Estas metodologías están principalmente pensadas para trabajar en equipos de múltiples personas. Por tanto, dan mucha importancia a realizar un buen proceso de documentación del trabajo realizado por cada integrante del equipo, ya sea mediante reuniones diarias o mediante softwares específicos de organización de proyectos. Otra parte importante de estas metodologías es el cliente, el cual es introducido en el proceso de desarrollo y debe actuar hasta la finalización del mismo. Algunos ejemplos de estas metodologías son la programación extrema (*extreme programming*) o el SCRUM.

Para realizar este trabajo se aplicarán todos estos principios, aunque no se cuente con un equipo de desarrollo al uso o con un cliente que conozca en profundidad el campo de estudio que se está trabajando (como puede ser la dermatología). Así, se ha divido el proceso de desarrollo de la aplicación web en 3 iteraciones:

1. **Primera iteración**: Una primera iteración consiste en desarrollar todas las funcionalidades necesarias para utilizar los modelos de clasificación mediante la

aplicación web, tanto los creados durante el desarrollo del trabajo como los de otros usuarios. Esto incluye subir y almacenar clasificadores, obtener el diagnóstico de una imagen con un clasificador, y entrenar un modelo con un *dataset*.

2. **Segunda iteración**: La siguiente iteración desarrolla el método de autenticación OAuth de usuarios, permitiendo así crear permisos de acceso a las distintas funcionalidades de la aplicación.
3. **Tercera iteración**: La tercera iteración se centra en desarrollar las funcionalidades de los administradores, permitiéndoles validar los modelos de los usuarios y permitir el uso de los *datasets*.
4. **Cuarta iteración**: Esta iteración comienza cuando todas las funcionalidades hayan sido implementadas, y se centra en arreglar posibles errores que puedan aparecer al añadir los datos con los que contará la aplicación inicialmente y en añadir elementos de interfaz de usuario que ayuden en el uso de la web.

Cabe destacar que, tras cada iteración, los requisitos de la aplicación han ido cambiando, tal y como era de esperar en una aplicación de estas características. Los requisitos que se creían necesarios en una primera iteración no han sido los mismos al llegar a las iteraciones posteriores, lo que ha permitido ajustar el proceso de desarrollo a las dificultades que el problema planteaba.

1.6 Tecnologías utilizadas

Aquí se listan las principales tecnologías utilizadas para el desarrollo de este TFG:

1.6.1 Flask

Flask se trata de un *framework* de aplicaciones web de Python, basado en la especificación WSGI de Wekzeug y el motor de *templates* de Jinja2. Un WSGI (*Web Server Gateway Interface*) describe como un servidor web se comunica con otras aplicaciones web mediante peticiones. Es el *framework* de este tipo más utilizado, debido a lo rápido y escalable que puede llegar a ser. Además, trae una configuración por defecto que permite crear un servidor web funcional con pocas líneas de código.

Para utilizarse, el desarrollador define una serie de *endpoints*, los cuales sirven las distintas funcionalidades del servidor.

Flask se ha utilizado para realizar la implementación del servidor web de nuestra aplicación, el cual sirve mediante estos *endpoints* la información que el cliente web necesita para su funcionamiento.

1.6.2 React.JS

React es una biblioteca de JavaScript para construir interfaces de usuario, basándose en el desarrollo de componentes débilmente acoplados. Estos componentes se suponen independientes y reutilizables, lo que permite una gran flexibilidad y modularización de la interfaz. Cada componente React cuenta con una interfaz HTML para que el usuario pueda interactuar con el componente, además de una serie de funciones definidas en JavaScript que describen su comportamiento ante las acciones del usuario. A diferencia de otros *framework* de interfaces web, React no espera que se definan múltiples páginas en HTML que mostrar al usuario, ya que eso forzaría a un comportamiento muy estático de la aplicación. En cambio, los componentes React son cargados y descargados en un único archivo HTML de manera dinámica mediante el acceso a distintas direcciones.

El cliente web de esta aplicación se ha desarrollado utilizando React. Este cliente web lanzará peticiones HTTP al servidor implementado en Flask con la información que obtenga de las interacciones del usuario. Una vez que se obtenga la información del servidor, React carga los componentes necesarios, mostrando al usuario de una manera amable y limpia la respuesta.

1.6.3 Tensorflow

Tensorflow es una plataforma centrada en el desarrollo de modelos de aprendizaje automático. Es compatible con múltiples lenguajes de programación, entre los que se encuentran Python y JavaScript, y tiene versiones especializadas para sistemas empotrados. La arquitectura de los modelos implementados con esta plataforma se basa en los llamados tensores, una matriz de n-dimensiones que permite almacenar

cualquier dato que discurra por la red. Esto permite que cualquier tipo de dato sea normalizado, haciéndolos idénticos en tipo.

Tensorflow nos brinda bastantes herramientas útiles para el desarrollo de nuestros modelos de clasificación, como bibliotecas y métodos de entrenamiento. Entre estas bibliotecas se encuentra Keras, la cual permite un desarrollo y uso de nuevos modelos muy rápido. Gracias a esta biblioteca, se cuenta con modelos ya entrenados, métricas de evaluación u optimizadores disponibles para utilizar. Por tanto, se utilizarán las tecnologías de Tensorflow y Keras para el desarrollo de los modelos de diagnóstico de la aplicación.

1.5.4 Firebase

Firebase se trata de una plataforma creada por Google enfocada al desarrollo de funcionalidades auxiliares para aplicaciones. Firebase sirve un gran número de funcionalidades, entre las que destacan bases de datos en tiempo real, autenticación OAuth, servicio de hosting, análisis de acceso y uso de la aplicación, computación en la nube, etc. Estas funcionalidades son muy útiles para cualquier aplicación web, por lo que se ha hecho uso de dos de ellas para el desarrollo de este trabajo:

- **Firestore Database**: Se trata de una base de datos no relacional, la cual permite un acceso rápido a grandes volúmenes de datos sin atenerse a un esquema fijo. Los datos de las entidades típicas de una base de datos relacional están recogidos en documentos, lo cuales no tienen campos fijos y se pueden componer de tipos de datos complejos (como listas o diccionarios) y otros documentos. Estos documentos se agrupan en colecciones, lo que facilita su agrupación a la hora de realizar consultas. Nuestra aplicación utiliza Firestore Database para almacenar la información de los usuarios que han iniciado sesión en la aplicación, así como los datos necesarios para la ejecución de los modelos de clasificación.
- **Authentication**: La autenticación con Firebase proporciona una serie de servicios ya preparados para poder realizar todo el proceso de registro e inicio de sesión de una manera rápida. Brinda, entre otras cosas, bibliotecas de interfaz de

usuario, que permiten guiarlo en el proceso de autenticación mediante múltiples métodos. Para este trabajo se ha utilizado el inicio de sesión con Google, ya que es el más ampliamente usado por cualquier usuario, pero su implementación para otros servicios, como Facebook, Twitter o únicamente con correo y contraseña, es análogo al de Google. Firebase genera un token de acceso a partir de estos métodos de autenticación, el cual puede ser usado para acceder a los datos de un usuario y brindarle los permisos que necesite para acceder a las funcionalidades de la aplicación.

1.5.5 Postman

Postman es una herramienta que permite realizar de manera sencilla pruebas a APIs de tipo REST. Principalmente, Postman ha sido utilizado para definir las URIs que el servidor web brindará al cliente, y comprobar que el resultado obtenido es el esperado. Postman permite definir fácilmente en formato JSON la información de la petición de entrada y de salida, lo que agiliza el proceso de *testing* de una API REST como la que implementa nuestro servidor web.

1.5.6 GitLab

GitLab comenzó siendo un sistema de control de versiones, el cual se utilizaba para monitorear los cambios realizados en el código de un proyecto software. Sin embargo, con el paso del tiempo se ha convertido en una herramienta de Dev-Ops que permite hacer un seguimiento exhaustivo del proceso de desarrollo de software. Durante este trabajo, GitLab ha servido para centralizar la información sobre los requisitos funcionales que era necesario implementar, y realizar una monitorización del avance de los mismos. Este proceso es muy importante cuando se cuenta con un equipo de trabajo, por lo que hemos visto oportuno realizar este seguimiento. Con esta herramienta, los requisitos se han subdividido en sus iteraciones correspondientes, y se ha realizado el control de versiones del código que se ha implementado, tanto del cliente como del servidor web.

2

Modelos de clasificación

2.1 Algoritmos de clasificación de imágenes

Comenzamos este capítulo realizando un resumen de los principales algoritmos de clasificación automática de imágenes. Se describen los árboles de decisión, las máquinas de vectores de soporte y las redes neuronales.

2.1.1 Árboles de decisión

Los árboles de decisión han sido utilizados desde 1940 para emular el comportamiento humano ante situaciones en las que se debe decidir la solución a un problema. Están compuestos en una estructura de árbol, en la cual cada pregunta que se debe responder (nodo) genera varios caminos (flechas), según la decisión tomada. Al final del árbol se encuentran los resultados que el algoritmo debería devolver (vector). Es importante destacar que las decisiones en estos árboles son excluyentes, es decir, no hay dos caminos distintos que lleven a la misma solución.

Para aplicar este tipo de algoritmo a la clasificación de imágenes se utiliza la siguiente función de coste a optimizar:

$$J(a, l_a) = \frac{m_{izquierdo}}{m} Gini_{izquierdo} + \frac{m_{derecho}}{m} Gini_{derecho}$$

donde *a* es la abreviación del atributo que se quiera comprobar, l_a el límite del atributo (también llamado umbral), y *m* el número de muestras. La función *Gini* utilizada en la función de coste es una medida de la impureza del nodo al que se le van a asignar los valores *a* y l_a. Esta función se calcula como:

$$gini = 1 - \sum_{k=1}^{n} p_c^2$$

donde pc es la probabilidad de cada clase, calculada dividiendo el número de muestras de cada clase en cada nodo por el número de muestras totales.

Estas dos funciones, por tanto, no dan la solución óptima sin conseguir un conocimiento previo de los datos con los que se está trabajando. Este conocimiento se obtiene mediante un proceso de entrenamiento supervisado, por el cual se deben etiquetar las imágenes con las clases a las que pertenecen, y luego el algoritmo optimizará la función de coste para cada nodo, aprendiendo así las características de la imagen que debe reconocer para realizar la clasificación. Este entrenamiento supervisado es muy común y es sobre el que se sustenta la mayoría de algoritmo de clasificación que no trabajen con un gran número de clases.

2.1.2 Máquinas de vectores de soporte

Las máquinas de vectores de soporte (SVM de sus siglas en inglés), son algoritmos de aprendizaje supervisado, utilizados en problemas de regresión y clasificación binaria lineal desde 1990. Además de la clasificación binaria lineal, la potencia de estos algoritmos reside en construir espacios de dimensionalidad muy alta que permitan separar las muestras en dos grupos mediante un hiperplano. El algoritmo trata de encontrar el hiperplano óptimo de separación. Este espacio de dimensionalidad alta permite aplicar estos algoritmo a muestras muy complejas, como conjuntos de imágenes, textos o gráficas.

Estos algoritmos requieren de una fase de entrenamiento previa a su uso, ya que el cálculo del hiperplano óptimo no es directo. Este tipo de algoritmos también necesita en muchos casos un tratamiento de los datos previo, que permita que la división encontrada por el algoritmo sea la correcta. Si no se etiquetaran los datos de entrenamiento (aprendizaje no supervisado) se podría llegar a divisiones de las muestras por características no esperadas, que pueden no ser importantes para la tarea que se quiera realizar.

2.1.3 Redes neuronales y *deep learning*

Las redes neuronales son el método más extendido actualmente para realizar la clasificación de imágenes. La idea de estas redes es conectar nodos llamados neuronas que se transmiten información. Esta información consiste en valores de números reales, calculados mediante funciones no lineales aplicadas a sus entradas. Estas funciones se denominan funciones de activación. Además, la neurona cuenta con un valor de "peso", el cual es calculado para cada neurona mediante un entrenamiento, lo que permite que la red aprenda.

Las neuronas se estructuran en capas. La primera capa se denomina capa de entrada y es la que normalmente adapta la información para poder ser procesada por el resto de neuronas. Las capas intermedias se denominan capas ocultas y son las que calculan los pesos que va a aprender la red mediante el entrenamiento. Las posibles configuraciones de estas capas son muy numerosas, ya que pueden variarse las funciones de activación que utiliza, las conexiones que tienen, el número de las mismas, etc. La última capa, denominada capa de salida, obtiene toda la información procesada por las capas ocultas y le da un formato concreto. El número de valores de salida vendrá dado por el número de neuronas que tenga la capa de salida.

Si la capa de salida cuenta con una sola neurona, el modelo, ante una entrada como puede ser una imagen, dará un único valor de salida. Esto es útil, por ejemplo, para realizar una clasificación binaria, distinguiendo entre positivo y negativo. El perceptrón es el ejemplo más sencillo de red neuronal, donde se tienen una capa de entrada, un número pequeño de capas ocultas y una capa de salida.

Para que el modelo sea capaz de aprender, es decir, calcular los valores de peso para cada neurona, se utilizan una función de coste y el algoritmo de Descenso del Gradiente. La función de coste será la función que nos diga el error cometido por el modelo durante su entrenamiento para la combinación de pesos que esté usando en ese momento. Por tanto, para reducir el error del modelo, será necesario encontrar el punto mínimo de esa función. Para localizar este punto mínimo se utiliza el algoritmo de Descenso del Gradiente, donde se calculan derivadas parciales para calcular el gradiente. Cuando este gradiente tenga valor 0, se habrá encontrado un mínimo en la función de coste, lo que nos dirá que el modelo ha aprendido y realiza su tarea correctamente.

Sin embargo, para modelos mayores que el perceptrón, con más capas ocultas, el cálculo de las derivadas parciales de manera directa es imposible. Para poder obtener el valor del gradiente se utiliza el algoritmo de *Backpropagation*. Este algoritmo calcula el error que han generado las neuronas de la red, ponderando más aquellas que tuvieran un valor de peso mayor. Estos errores son los que se utilizan finalmente para calcular el gradiente.

Dependiendo de los datos que se utilicen a la hora de realizar el entrenamiento de los modelos diferenciamos dos grandes categorías:

- Aprendizaje supervisado: en él, las entradas que utilizaremos para entrenar el modelo ya han sido previamente clasificadas. El modelo dará una respuesta inicial para esas entradas y se comparará con los valores esperados asociados. Dependiendo de cuál sea la tasa de acierto obtendremos el valor de entrada de nuestra función de coste. Vemos que el principal problema de estos modelos es la gran cantidad de valores de entrada que deberán ser correctamente clasificados previamente a comenzar el entrenamiento, por lo que solo serán útiles en situaciones en las que conozcamos las clases existentes en nuestro dataset y a cuál corresponde cada entrada.

- Aprendizaje no supervisado: en él, las entradas utilizadas en el entrenamiento no están previamente clasificadas. Este tipo de entrenamiento permite que el modelo aprenda estructuras y patrones de los datos de entrada, lo que permite, por ejemplo, decir si dos entradas son similares entre sí. Las clases en las que predicen estos modelos no están dadas por sus diseñadores. Los modelos entrenados así están diseñados para reconocer grandes cantidades de clases en volúmenes de *datasets* diferentes muy altos. También son muy utilizados para generar *datasets* de entrenamiento para otros modelos.

Los modelos de *deep learning* no son más que modelos de redes neuronales con una gran cantidad de capas ocultas. Esto permite crear clasificadores universales, mucho más potentes que los del perceptrón. Gracias a esto, los modelos de *deep learning* son muy frecuentemente utilizados para clasificación de imágenes, procesamiento del lenguaje natural, bioinformática, etc. Estos modelos son los que estudiaremos durante el trascurso de este TFG.

2.2 Requisitos de los modelos de clasificación de enfermedades

Para saber qué tipo de red debemos diseñar y cómo entrenarla hemos estudiad el problema que queremos solucionar y los datos de los que se disponemos. La idea principal es construir una red que clasifique correctamente imágenes de distintas enfermedades capilares, para lo cual necesitamos un *dataset* que representa correctamente la naturaleza del problema.

Como hemos mencionado anteriormente, la proporción de alteraciones en la piel benignas, como los nevus o el acné, con respecto a enfermedades malignas (como los melanomas) está muy descompensada. Por tanto, si entrenáramos los modelos con *datasets* balanceados (en los que hay una proporción similar de entradas de todas las clases) el modelo podría no desenvolverse correctamente en un entorno real. Los *datasets* que he utilizamos para el entrenamiento deben ser acordes a la situación real en la que se van a utilizar.

Por otro lado, no debemos olvidar que la clase más infrecuente es también la más importante de clasificar correctamente, ya que representa enfermedades de mayor peligro para la persona. Debemos asegurarnos de que todas las clases se clasifican correctamente, da igual cuál sea su frecuencia de aparición en el dataset.

De ahora en adelante, el dataset que se ha utilizado para realizar el entrenamiento de los modelos es el *dataset* ISIC 2020 del ISIC Challenge. El ISIC Challenge es una competición anual de clasificación de enfermedades dermatológicas, la cual tiene miles de participantes y durante la que se desarrollan miles de modelos especializados en resolver estos problemas. Los *datasets* utilizados para estas competiciones son públicos. En concreto, el de 2020 está constituido por 33.126 imágenes ya clasificadas, por lo que es el perfecto punto de partida para el diseño de los modelos.

2.2.1 Estudio inicial del dataset

Se comienza realizando un estudio de los datos de los que disponemos en el *dataset* del ISIC. El dataset está formado las propias imágenes y una tabla donde se recogen, para cada imagen, los siguientes datos con las distribuciones mostradas en las tablas:

- image_name: Nombre de la imagen a la que corresponde esta entrada de la tabla.
- patient_id: Identificador del paciente al que se le tomó la foto. Puede ser útil para realizar un estudio continuado de un mismo paciente.
- lession_id: Identificador del tipo de lesión que se ha diagnosticado.
- sex: Género del paciente. Únicamente con valores masculino (*male*) y femenino (*female*).
- age_approx: Edad aproximada del paciente en el momento en el que se realizó fotografía. A continuación, se puede ver la distribución de valores.

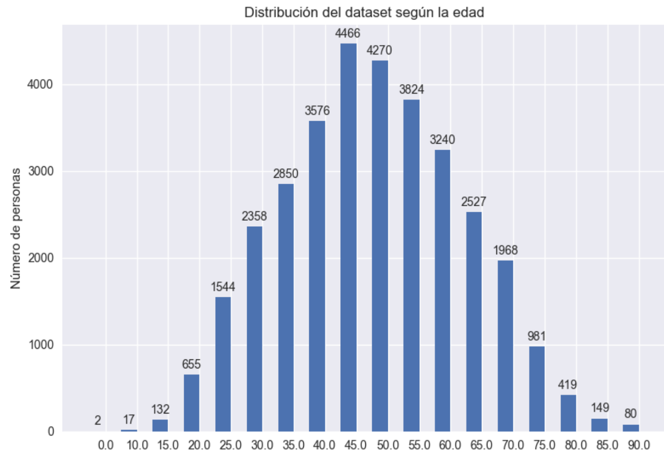


Figura 2.1 Distribución por edad del dataset ISIC 2020.

- anatom_site_general_challenge: posición del cuerpo en la cual se encuentra la lesión.

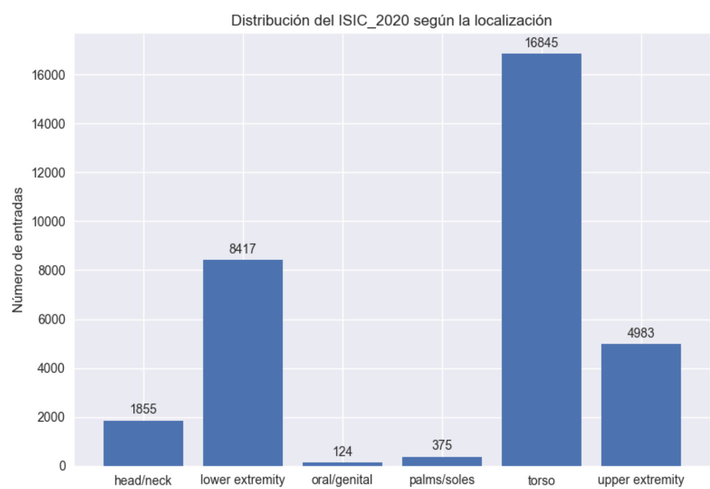


Figura 2.2 Distribución por parte del cuerpo del dataset ISIC 2020.

- diagnosis: Nombre de la enfermedad diagnosticada en la imagen.

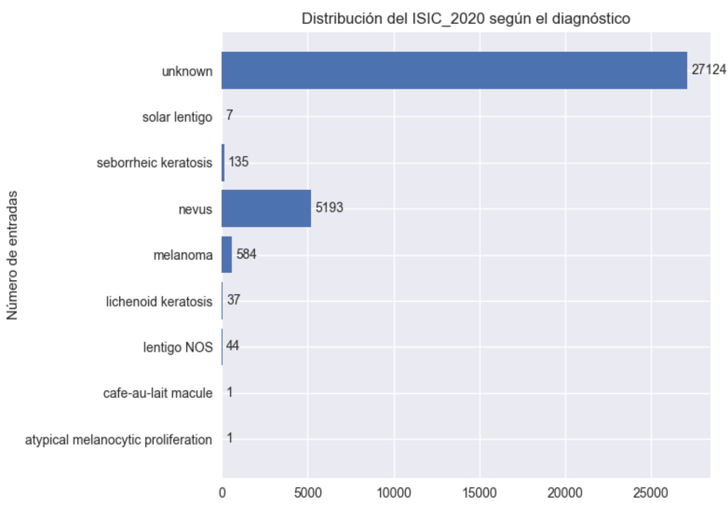


Figura 2.3 Distribución por diagnóstico del dataset ISIC 2020.

- benign_malignant: Clasificación de la enfermedad diagnosticada en benigna o maligna.

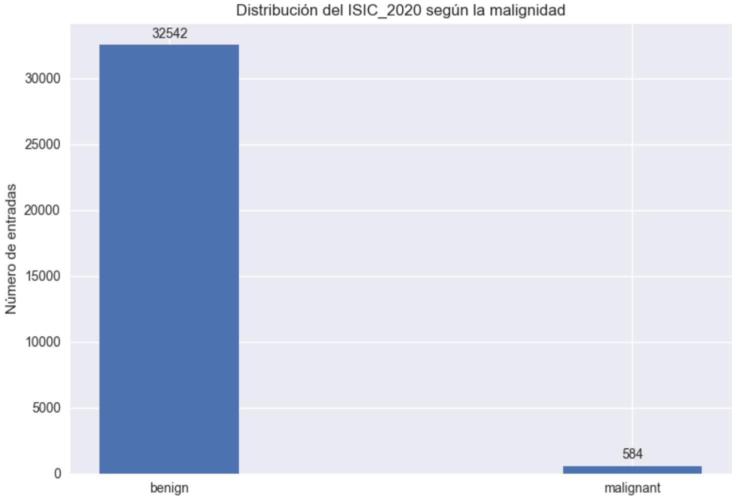
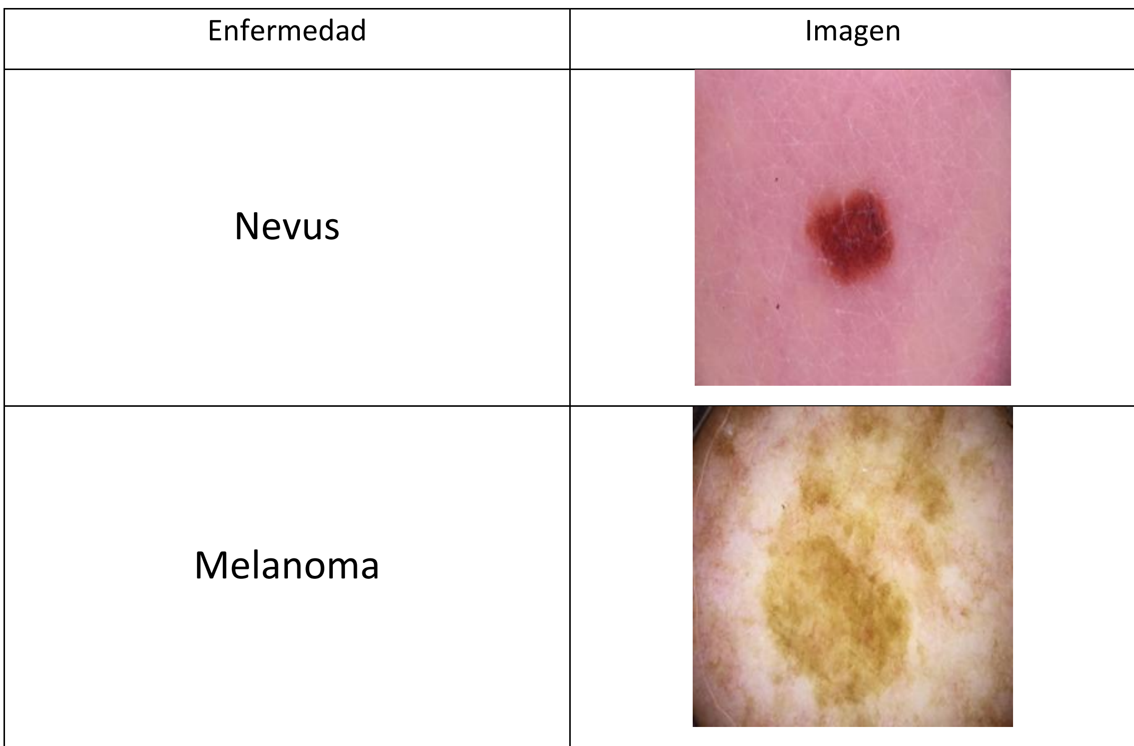


Figura 2.4 Distribución por malignidad del dataset ISIC 2020.

- target: Valor numérico 0 o 1 que representa si la enfermedad es benigna o maligna. Es equivalente a la columna benign_malignant.

A continuación, se muestran algunos ejemplos de las imágenes de algunas de las enfermedades más frecuentes del *dataset*:



Enfermedad	Imagen
Nevus	
Melanoma	

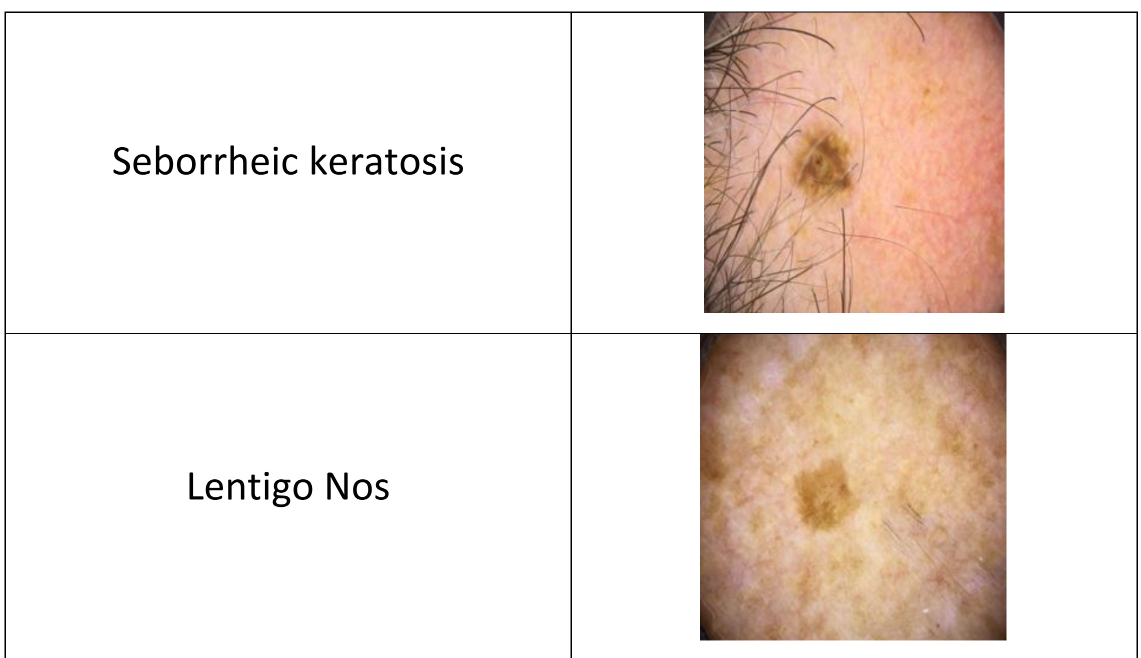


Tabla 2.1 Ejemplos de imágenes del dataset ISIC_2020

Podemos apreciar que las lesiones a simple vista son muy similares entre sí. Además, también vemos que las lesiones no están aisladas, por lo que pueden aparecer en las imágenes junto con otros elementos, como pelos u otras manchas. La iluminación de la imagen también puede llegar a ser muy distinta de una imagen a otra.

En la columna de diagnóstico vemos que las imágenes están clasificadas en 8 enfermedades diferentes, con una gran disparidad en el número de entradas. Mientras que de la clase nevus tenemos 5.193 imágenes, de *solar lentigo*, *lichenoid keratosis*, *lentigo NOS*, *cafe-au-lait macule* y *atypical melanocytic proliferation* tenemos menos de 50. Con tan pocas imágenes sería muy complicado que un modelo pueda llegar a aprender a distinguir estas enfermedades del resto. Además, puede ser que estas pocas imágenes no sean representativas de la lesión para la que están diagnosticadas, por lo que nuestros modelos podrían no estar aprendiendo correctamente.

Por otro lado, de las 33.126 imágenes de las que disponemos, 27.124 están clasificadas como "desconocidas". Para realizar un entrenamiento supervisado deberíamos descartar estas imágenes, ya que no están clasificadas con una categoría que queramos distinguir del resto. Por tanto, perderíamos gran parte del potencial de este dataset si quisiéramos realizar el diagnóstico de las enfermedades concretas.

Aunque la mayoría de las imágenes no tienen un diagnóstico concreto, sí que están clasificadas en enfermedades benignas y malignas. Según la Figura 2.4, 32.542 de las imágenes son benignas, mientras que solo 584 son malignas. Las enfermedades malignas se corresponden directamente con las diagnosticadas como melanoma. Por tanto, vemos que este dataset está totalmente orientado a distinguir los melanomas del resto de alteraciones de la piel.

Por tanto, para poder aprovechar todas las imágenes del dataset y obtener unos resultados exitosos con nuestros modelos, nos hemos centrado en clasificar correctamente las imágenes como "enfermedad benigna" o "melanoma". Además, ya que queremos realizar la clasificación únicamente pidiéndole al usuario una imagen de la lesión, no se ha utilizado el resto de datos recogidos de los pacientes, como la edad o el género. Sin embargo, es muy posible que estos datos pudieran mejorar el resultado de los modelos ya que, como hemos visto anteriormente, los melanomas son mucho más infrecuentes en niños que en adultos y en mujeres que en hombres.

2.2.2 Preparación del dataset para el entrenamiento

Para poder evaluar correctamente el aprendizaje de un modelo de *deep learning* durante el entrenamiento es necesario dividir nuestro dataset en dos subconjuntos, un subconjunto de entrenamiento y otro de evaluación. El modelo aprende a partir de las imágenes del subconjunto de entrenamiento a clasificarlas en las dos clases etiquetadas. Después de cada iteración (o época) del entrenamiento, se utiliza el subconjunto de evaluación para calcular la mejora del modelo, con imágenes que son distintas a las del entrenamiento. Así, se realiza esta división, manteniendo siempre que ambos subconjuntos sean representativos del problema. Como el *dataset* está muy desbalanceado, ya que la naturaleza de nuestro problema es así, deberemos mantener este desbalanceo en ambos subconjuntos.

Del total de imágenes, se ha usado el 70% para el subconjunto de entrenamiento y un 15% para el subconjunto de evaluación, manteniendo en ambos, como en el *dataset* original, un 98% de entradas benignas y un 2% de entradas malignas. El 15% restante se ha mantenido fuera del entrenamiento de los modelos para poder comparar los

resultados finales tras el entrenamiento entre modelos. Esto es importante, ya que no debemos utilizar ninguna imagen que haya participado en el entrenamiento, o los resultados quedarán sesgados. A este subconjunto lo llamaremos subconjunto de test. Este subconjunto también mantiene la proporción original de entradas benignas y malignas.

2.2.3 Función de coste. *Binary Cross-Entropy*

Antes de definir el propio modelo hemos necesitado tener en cuenta qué función de coste utilizar para evaluar el error cometido por el modelo durante el entrenamiento. Ya que estamos trabajando con clasificadores binarios, lo más recomendable es usar la función de Entropía cruzada binaria (*Binary Cross-Entropy*).

Idealmente, teniendo dos clases que distinguir (Clase 0 y Clase 1), los modelos de clasificación binaria devuelven un valor real comprendido entre 0 y 1, que representa la probabilidad de que la entrada sea de la clase 1. Ante una evaluación del modelo con múltiples entradas, se recoge una probabilidad para cada una de ellas y se compara con los resultados esperados, los cuales son valores enteros {0, 1}. La función de pérdida *Binary Cross-Entropy* asigna valores altos a predicciones que estén muy alejadas de los resultados esperados, y valores bajos cuando las predicciones sean acertadas. La función de pérdida se define de la siguiente forma:

$$H_p(q) = -\frac{1}{N}\sum_{i=1}^{N} y_i \cdot log\left(p(y_i)\right) + (1 - y_i) \cdot \log\left(1 - p(y_i)\right)$$

donde *y* es la clase a la que pertenece la entrada (0 o 1), y *p(y)* es la probabilidad predicha por el modelo. Ya que esta función calcula el error cometido por el modelo, durante el entrenamiento se busca que su valor se aproxime a 0.

Sin embargo, definir únicamente una función de coste no es suficiente. Es necesario definir un método de evaluación del modelo que sea acorde al contexto del problema que queremos resolver, una métrica del rendimiento del modelo.

2.2.4 Métricas de evaluación. Curva ROC y PR.

Existen muchas métricas posibles que nos dan información útil sobre el rendimiento de nuestro modelo y que necesitamos saber interpretar correctamente para poder aplicarlo con seguridad. A continuación, vamos a comparar distintas métricas de rendimiento utilizadas comúnmente en problemas de clasificación.

Exactitud (*Accuracy*):

La exactitud de un modelo se define como el número de aciertos en la clasificación de un conjunto de evaluación dividido entre el número de clasificaciones realizadas. Es la métrica de evaluación más sencilla y extendida entre los clasificadores, tanto binarios como multicategóricos. Utilizando esta métrica, se busca que nuestro modelo acierte con la mayor frecuencia posible. Sin embargo, esta métrica no es muy útil en nuestro caso, ya que supone que los *datasets* con los que se entrene el modelo estarán balanceados. Si entrenásemos nuestro modelo únicamente fijándonos en la exactitud siempre conseguiríamos una *accuracy* del 98%. El modelo aprendería rápidamente que clasificar cualquier imagen como la clase más numerosa del *dataset* le daría la mayor exactitud posible, algo que queremos evitar a toda costa.

Diagnóstico clínico:

Por tanto, ha sido necesario buscar métricas que funcionen con *datasets* desbalanceados. Para resolver esto, es muy común recurrir a métricas ampliamente reconocidas y utilizadas en ámbitos médicos. La mayoría de las siguientes métricas tienen su origen en la medicina clínica y se basan en relacionar el resultado de la prueba diagnóstica con el valor real que se debería haber predicho. Mostraremos ahora cómo se definen los términos en los que se basan las siguientes métricas:

<table>
<tr><td colspan="2" rowspan="2">Tipo de diagnóstico</td><td colspan="2">Enfermedad</td></tr>
<tr><td>Ausente</td><td>Presente</td></tr>
<tr><td rowspan="2">Prueba diagnóstica</td><td>Negativo</td><td>Verdadero negativo</td><td>Falso negativo</td></tr>
<tr><td>Positivo</td><td>Falso positivo</td><td>Verdadero positivo</td></tr>
</table>

Figura 2.5 Clasificación de los diagnósticos clínicos.

- Verdadero negativo (VN): Un diagnóstico se considerará VN cuando el resultado de la prueba diagnóstica es NEGATIVO y el paciente NO padece la enfermedad.
- Falso negativo (FN): Un diagnóstico se considerará FN cuando el resultado de la prueba diagnóstica es NEGATIVO y el paciente SÍ padece la enfermedad.
- Falso positivo (FP): Un diagnóstico se considerará VN cuando el resultado de la prueba diagnóstica es POSITIVO y el paciente NO padece la enfermedad.
- Verdadero positivo (VP): Un diagnóstico se considerará VN cuando el resultado de la prueba diagnóstica es POSITIVO y el paciente SÍ padece la enfermedad.

Como se puede observar, estos valores solo hacen referencia a casos de clasificación binaria, en los cuáles un paciente solo puede dar positivo o negativo y tener o no la enfermedad. Ya que este es el caso de nuestro problema, cualquier métrica definida a partir de estos términos puede ser aplicada para diferenciar las imágenes benignas de las malignas. Para ello, a partir de ahora hablaremos de las imágenes clasificadas como benignas de casos negativos, y de las imágenes clasificadas como malignas de casos positivos. Los diagnósticos o clasificaciones se recogen en una matriz de confusión. Esta matriz muestra el número de veces que se ha realizado cada tipo de diagnóstico (VN, FP, FN o VP) en una muestra, resultando así en nuestro caso en una matriz de 2 filas y 2 columnas. Estas matrices se utilizarán de ahora en adelante para ilustrar los resultados de una clasificación.

Sensibilidad (Recall), Especificidad (Specificity) y Precisión (Precision):

Las siguientes métricas se definen a partir de los términos de diagnóstico clínico y nos son muy útiles a la hora de predecir el rendimiento de nuestro modelo en un entorno real. Para ilustrar mejor cómo afectara el uso de cada una al funcionamiento de nuestro modelo de clasificación utilizaremos matrices de confusión con un hipotético modelo de diagnóstico. Las matrices están formadas por cuatro celdas, las cuales tendrán un único valor que representará los valores de VN, FP, FN y VP respectivamente.

Para todos los casos utilizaremos un conjunto de 1000 diagnósticos, donde 100 corresponden a casos positivos y 900 casos negativos.

Sensibilidad: Se define como la tasa de verdaderos positivos, es decir, el porcentaje de pacientes que padecen la enfermedad y se les ha diagnosticado como tal correctamente. En nuestro caso, mide la capacidad que tiene nuestro modelo de detectar casos positivos (imágenes clasificadas como malignas). Esta métrica es muy importante, ya que nuestro principal objetivo es conseguir reconocer los casos positivos (los cuales solo constituían un 2% de nuestro *dataset*) de los casos negativos.

$$Sensibilidad = \frac{VP}{VP + FN}$$

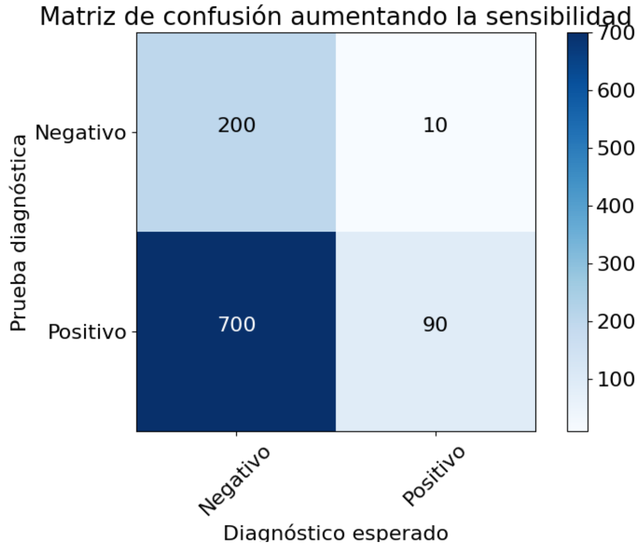


Figura 2.6 Matriz de confusión aumentando la sensibilidad.

Como vemos en la matriz de confusión, si entrenamos nuestro modelo utilizando como guía esta métrica, se volverá muy sensible, permitiendo detectar la mayoría de casos positivos a costa de tener muchos casos de falsos positivos.

Especificidad: Se define como la tasa de verdaderos negativos, es decir, el porcentaje de pacientes que no padecen la enfermedad y se les ha diagnosticado como negativos. En nuestro caso, mide la capacidad que tiene nuestro modelo de detectar los casos negativos (imágenes clasificadas como benignas).

$$Especificidad = \frac{VN}{VN + FP}$$

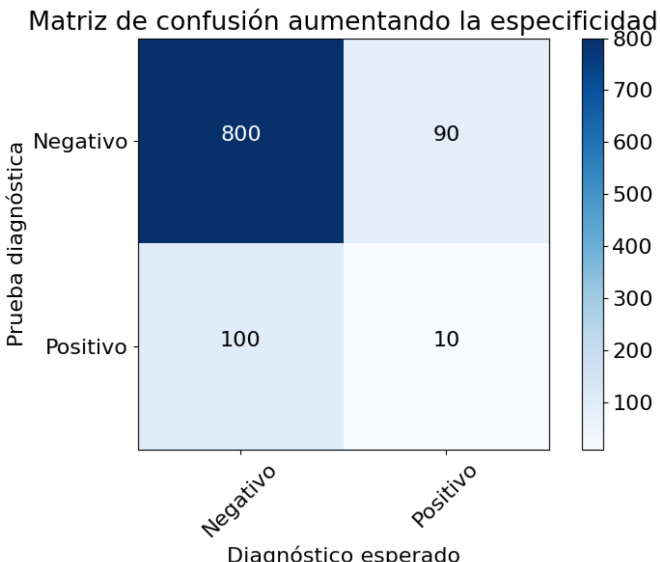


Figura 2.7 Matriz de confusión aumentando la especificidad.

Vemos que esta métrica, por si sola, podría llevarnos a casos como el de guiarnos por la exactitud, donde se clasifiquen el mayor número de imágenes como negativas. Aun así, puede sernos muy útil en conjunción con la sensibilidad, y así reconocer que nuestro modelo no sea demasiado sensible.

Precisión: Se define como el valor de las predicciones positivas, es decir, dado el total de predicciones que han sido positivas (VP + FP), qué porcentaje de éstas son verdaderos positivos. En nuestro caso, esta métrica mediría lo fiables que son las predicciones positivas que dé nuestro modelo. Si nos damos cuenta, en la definición de esta métrica solo participan los diagnósticos positivos. La precisión da mucha importancia a detectar los casos positivos sin aumentar los falsos positivos. Por lo tanto, es una métrica realmente útil para casos como el nuestro, el cual tiene 2 clases muy desbalanceadas, donde aumentar el número de FP para aumentar los VP podría parecer la única solución.

$$Precisión = \frac{VP}{VP + FP}$$

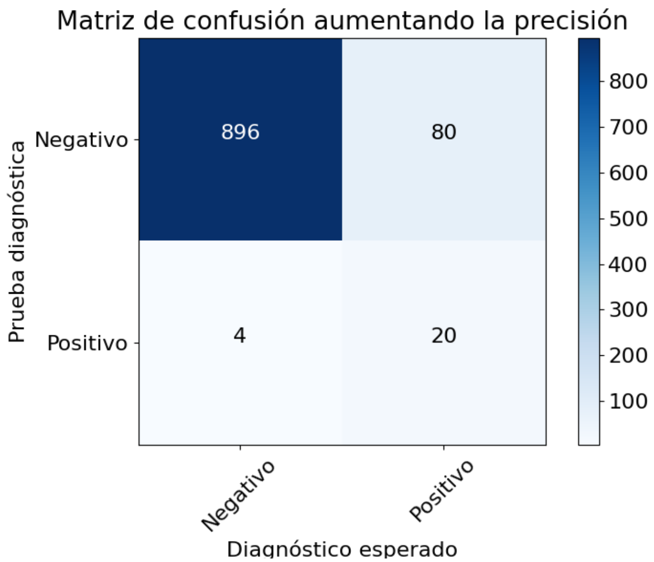


Figura 2.8 Matriz de confusión aumentando la precisión.

Sin embargo, como podemos apreciar en la matriz, esta métrica no tiene en cuenta de ninguna forma cuantos de los casos positivos totales se han clasificado correctamente, sino cuantos de los diagnósticos positivos son acertados. Para *datasets* muy grandes, esto puede llevar a sesgos como el que se ilustra, donde la mayoría de casos se diagnostiquen negativos. El resultado obtenido puede ser parecido al de maximizar la especificidad, aunque con la precisión minimizamos el número de FP, lo cual puede ser muy útil.

Podemos concluir entonces que ninguna de estas métricas nos sirve por si sola para reconocer si nuestro modelo funciona correctamente. Maximizar alguno de estos valores sólo nos llevaría a sesgos e imprecisiones. Sin embargo, sí vemos que estas métricas se complementan entre sí. Por ejemplo, un modelo que fuese muy sensible y a la vez muy específico nos podría permitir detectar correctamente los casos positivos sin incluir falsos positivos.

El siguiente paso entonces ha sido encontrar una manera de equilibrar varias métricas al mismo tiempo, encontrando el punto óptimo entre ellas.

Curvas ROC y PR

Las curvas ROC y PR son métricas que se basan en relacionar las métricas definidas anteriormente para distintos valores de un umbral de discriminación. Veamos en qué se basa esta idea mediante el siguiente diagrama:

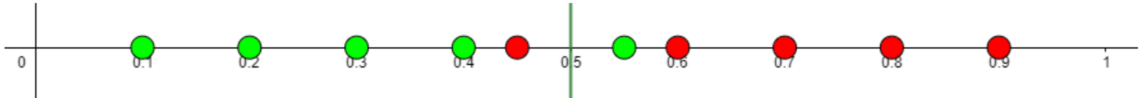


Figura 2.9 Diagrama de distribución de diagnósticos.

El diagrama representa una serie de diagnósticos mediante círculos en un eje, los casos negativos de color verde y los positivos de color rojo. La posición del círculo en el eje representa la probabilidad estimada por el clasificador de que el diagnóstico sea positivo, comprendido entre 0 y 1. Ante esto se define un umbral, un valor a comparar con el resultado del clasificador para cada diagnóstico (línea vertical verde). Si el valor del clasificador es inferior al umbral se diagnosticará negativamente, y si es superior o igual, positivamente. Ya que normalmente un clasificador devuelve probabilidades, lo más común es que este umbral tenga un valor de 0,5. Así, si la probabilidad de que un caso sea positivo es mayor del 50%, se clasificará como tal. Sin embargo, vemos en el diagrama que el umbral de 0,5 crea diagnósticos falsos. Moviendo el umbral a un valor, por ejemplo, de 0,4, conseguiríamos reducir los falsos negativos a 0 a costa de aumentar los falsos positivos. Estaríamos aumentando la sensibilidad a costa de reducir la especificidad y la precisión.

La curva ROC se basa en representar cómo varían la sensibilidad y la tasa de falsos positivos de nuestra clasificación en función del umbral escogido. La tasa de falsos positivos (FPR) se puede expresar como

$$FPR = 1 - Especificidad$$

por lo que, indirectamente, la curva ROC relaciona la sensibilidad y la especificidad.

Una curva ROC (*Receiver Operating Characteristic*) se representa típicamente de la siguiente forma. Primero se obtiene un clasificador y se utiliza para clasificar un conjunto

de datos de los cuáles ya conocemos su diagnóstico. Tras obtener el valor del clasificador para cada punto, se prueba un número determinado de umbrales (en el caso de la imagen siguiente 200 umbrales) y se calcula la sensibilidad y la especificidad para cada uno de ellos. Estos pares de valores generan la curva que vemos a continuación:

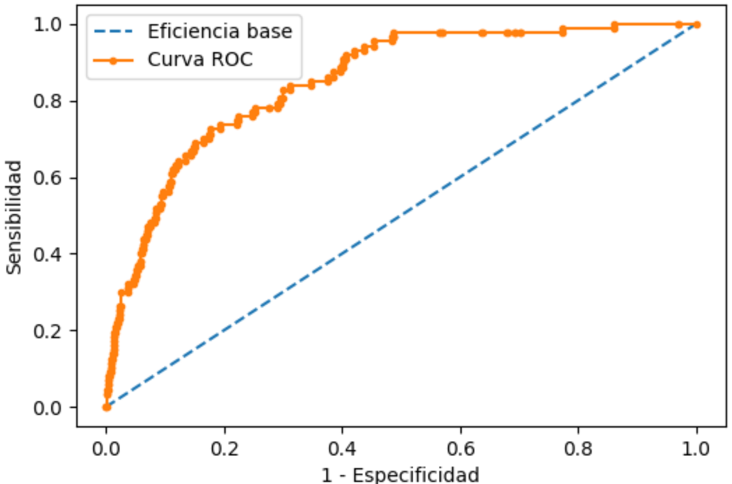


Figura 2.10 Ejemplo de curva ROC.

La línea discontinua azul muestra la eficiencia base de cualquier clasificador. Esta eficiencia base se da cuando un clasificador siempre da el mismo resultado como diagnóstico. Cualquier punto de la curva que se aleje de la diagonal principal representa "conocimiento" aprendido por el clasificador para realizar el diagnóstico. Se ilustra a continuación la curva ROC de un clasificador perfecto, para el cual se puede definir un valor umbral que separe los casos positivos de los negativos.

Respuesta del clasificador para 8 diagnósticos:

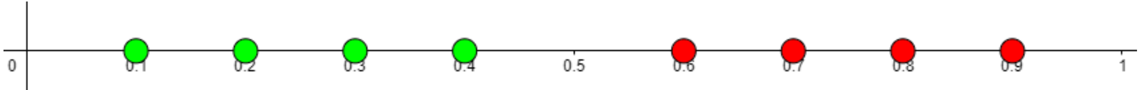


Figura 2.11 Diagrama de distribución de diagnósticos perfecto.

Curva ROC para los valores umbrales {0, 0.25, 0.5, 0.75, 1}:

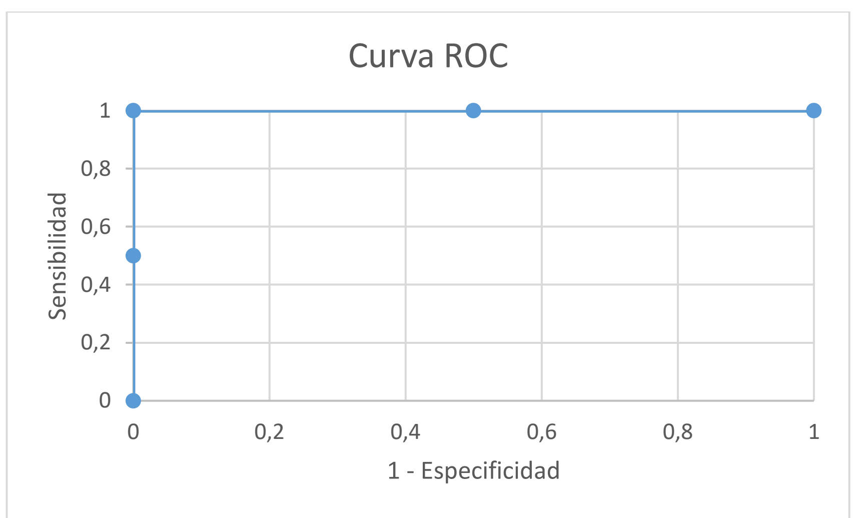


Figura 2.12 Curva ROC para un clasificador perfecto.

Vemos que la curva que se dibuja para un clasificador perfecto pasa por el punto (0,1), donde la sensibilidad con valor 1 permite diagnosticar correctamente los casos positivos y la especificidad con valor 1 permite diagnosticar los casos negativos.

La curva PR (*Precission-Recall*) es similar a la curva ROC. Esta curva relaciona la precisión con la sensibilidad para cada umbral que se pruebe. Su proceso de elaboración es el mismo que el de la curva ROC, obteniendo el siguiente resultado:

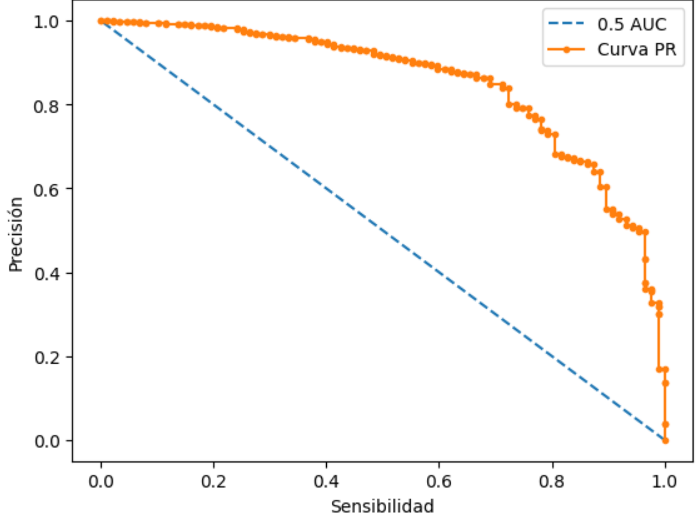


Figura 2.13 Ejemplo de curva PR.

Esta curva es mucho más sensible a los errores de clasificación de los casos positivos, por lo que en *datasets* balanceados es muy difícil obtener valores altos como los que se muestra en esta gráfica. Sin embargo, es un muy buen indicativo de la mejora del

modelo en conjunto con la curva ROC. La curva PR para un clasificador perfecto sería simétrica a la curva ROC:

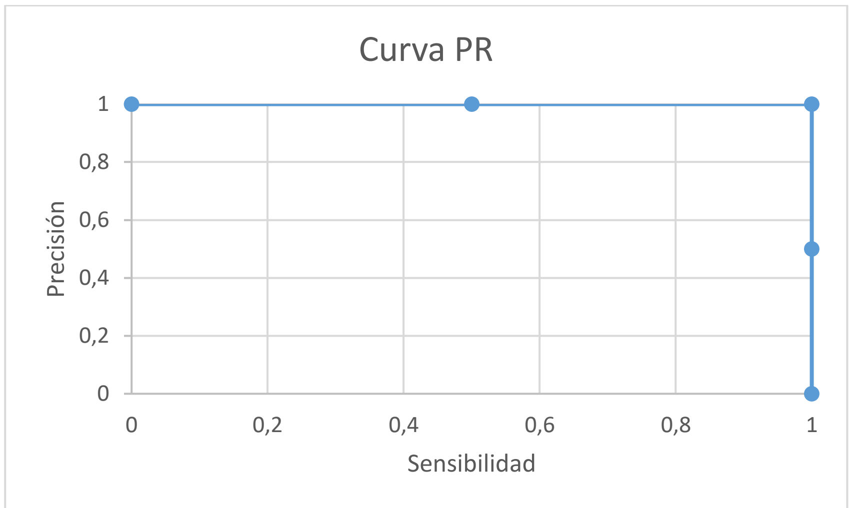


Figura 2.13 Curva PR para un clasificador perfecto.

Si calculamos el área bajo las curvas ROC y PR, obtendremos un área de 1. Por tanto, podemos concluir que cuanto más se acerque el valor del área bajo estas curvas a 1 para un clasificador, más eficiente será. Esta área se denomina AUC (Area Under Curve), y es la métrica más ampliamente utilizada derivada de estas curvas, tanto en ámbitos médicos como en ciencias de la computación. Ésta es finalmente la métrica utilizada para evaluar los modelos construidos durante su entrenamiento. Además, también nos ha permitido comparar el rendimiento de distintos modelos tras el entrenamiento, por lo que podremos monitorizar si los cambios que hemos aplicado han afectado satisfactoriamente a su eficiencia.

Una vez que hemos entendido la naturaleza del problema que queremos resolver, hemos estudiado los datos de los que disponemos, y hemos decidido qué criterios vamos a utilizar para medir el rendimiento de los modelos que construyamos, se comienza a definir los modelos de *deep learning* que podrán ser utilizados desde la aplicación web.

2.3 Desarrollo de los modelos de clasificación

Los siguientes modelos se definen utilizando Tensorflow para Python. Aun así, la arquitectura que vamos a definir con Tensorflow es un estándar del *deep learning* y las distintas técnicas que utilicemos para mejorar su rendimiento son transversales para cualquier tecnología y modelo de *machine learning*.

2.3.1 Arquitectura del modelo

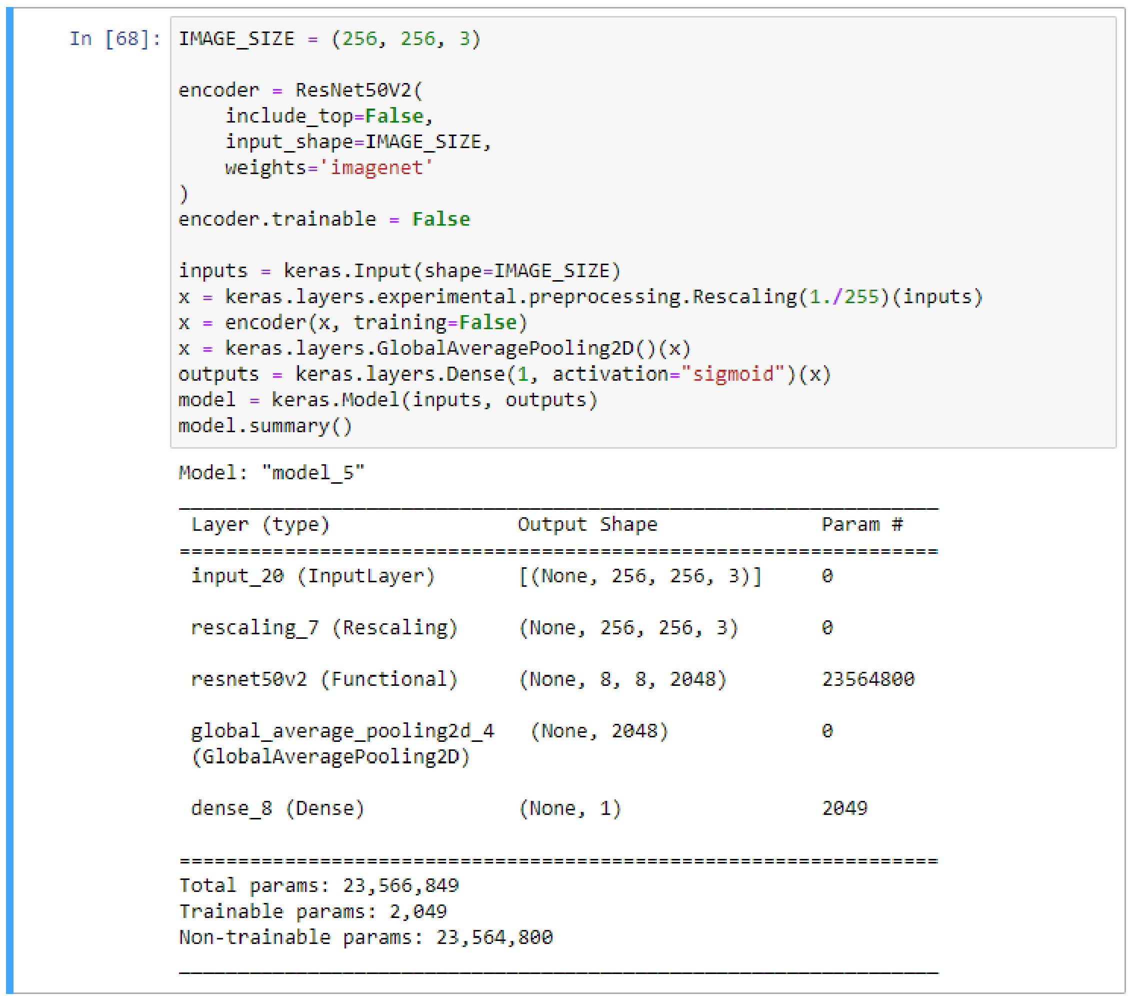
En lugar de partir de un modelo totalmente nuevo, se ha hecho uso de Keras Applications. Keras pone a disposición de cualquier usuario modelos de *deep learning* que ya vienen con un entrenamiento genérico. En nuestro caso, hemos utilizado ResNet50V2, una red convolucional 2D. ResNet50V2 está formada por 23.564.800 parámetros entrenables. Aun así, utilizaremos los valores que traen por defecto, ya que la red ha sido entrenada previamente utilizando el dataset ImageNet, un *dataset* que cuenta con miles de clases distintas.

Al ser un *dataset* muy amplio, con tantas imágenes que representan conceptos muy distintos, ResNet50V2 es capaz de reconocer patrones básicos en las imágenes. Esto es muy útil, ya que la red construida a partir de aquí no tiene que volver a aprender a hacer las transformaciones más básicas. ResNet50V2 conforma las capas ocultas de nuestra red.

La capa de entrada a la red debe tener un tamaño concreto. Estudiando el *dataset* de ISIC_2020, se observa que cada imagen tiene unas dimensiones distintas y son de muy alta resolución. Ya que la capacidad de cómputo con la que se cuenta para realizar este trabajo no permite trabajar con las imágenes en su tamaño original, se han redimensionado todas las imágenes a un tamaño de 256 por 256 píxeles. Así se consigue normalizar la entrada de la red y agilizar el proceso de entrenamiento. Además, las imágenes están en formato RGB, por lo que cuentan con 3 valores de 0 a 255 por cada píxel. Ya que nuestra red trabaja con valores comprendidos entre 0 y 1, se han normalizado los valores de las imágenes para que se ajusten a ese intervalo. Esto conformará nuestras capas de entrada.

Por último, añadiremos tras las capas de ResNet50V2 capas "densas". De esta forma, cada una de las neuronas de una capa está conectada con todas las neuronas de la siguiente capa. Este tipo de red permite una gran flexibilidad. En nuestro caso se ha utilizado para procesar la información de salida de ResNet50V2. Éstas son las neuronas que finalmente han sido entrenadas y especializadas en el diagnóstico de los melanomas. Necesitamos añadir una capa intermedia de *Average Pooling*, que permita adaptar el formato de ResNet a nuestra red densa. Finalmente, toda esta capa acaba en una única neurona, que aplica una función sigmoide para obtener un valor comprendido entren 0 y 1. Ésta es nuestra capa de salida.

A continuación, se muestra el código que crea esta red y el resumen que muestra Tensorflow sobre su arquitectura:



```
In [68]: IMAGE_SIZE = (256, 256, 3)

encoder = ResNet50V2(
    include_top=False,
    input_shape=IMAGE_SIZE,
    weights='imagenet'
)
encoder.trainable = False

inputs = keras.Input(shape=IMAGE_SIZE)
x = keras.layers.experimental.preprocessing.Rescaling(1./255)(inputs)
x = encoder(x, training=False)
x = keras.layers.GlobalAveragePooling2D()(x)
outputs = keras.layers.Dense(1, activation="sigmoid")(x)
model = keras.Model(inputs, outputs)
model.summary()
```

```
Model: "model_5"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_20 (InputLayer)       [(None, 256, 256, 3)]     0

 rescaling_7 (Rescaling)     (None, 256, 256, 3)       0

 resnet50v2 (Functional)     (None, 8, 8, 2048)        23564800

 global_average_pooling2d_4   (None, 2048)             0
 (GlobalAveragePooling2D)

 dense_8 (Dense)             (None, 1)                 2049

=================================================================
Total params: 23,566,849
Trainable params: 2,049
Non-trainable params: 23,564,800
_________________________________________________________________
```

Figura 2.14 Código para generar la red y resumen.

Finalmente, nos ha quedado una red con 23.566.849 parámetros, de los cuales 23.564.800 pertenecen a ResNet50V2 y no hace falta que sean entrenados. La capa densa que sí es entrenada cuenta con 2049 neuronas, las 2048 que conectan con la información obtenida de las capas ocultas y 1 neurona final que aplica la función sigmoide. Este número de parámetros es suficiente para hacer la clasificación con unos resultados correctos.

2.3.2 Proceso de entrenamiento

Para comenzar el proceso de entrenamiento, asignamos el optimizador que queremos que aplique el algoritmo de *Backpropagation*, la función de coste, y las métricas que queremos que guarde durante el entrenamiento para su posterior estudio.

Como optimizador se ha utilizado Adam, el optimizador por defecto de Keras, que aplica *Backpropagation*. Como función de coste se asigna *Binary Cross-Entropy* y como métricas de evaluación, calculamos los valores de AUC para las curvas ROC y PR para cada época de entrenamiento. El siguiente código realiza la configuración mencionada:

```
model.compile(
    optimizer=keras.optimizers.Adam(),
    loss=keras.losses.BinaryCrossentropy(),
    metrics=[keras.metrics.AUC(name="auc-ROC", curve='ROC'), keras.metrics.AUC(name="auc-PR", curve='PR')]
)
```

Figura 2.15 Código de configuración del entrenamiento.

El siguiente código comienza el entrenamiento del modelo:

```
history = model.fit(train_ds,
                    epochs=10,
                    validation_data=val_ds,
                    #validation_steps=10,
                    callbacks=[cb]
                   )
```

Figura 2.16 Código de comienzo del entrenamiento

Estos parámetros de entrenamiento se han mantenido constantes para los distintos modelos desarrollados, lo que permite comparaciones más justas. Mantenemos el mismo subconjunto de entrenamiento (train_ds) con el 70% de las imágenes de ISIC_2020 redimensionadas a 256 por 256 píxeles, 10 épocas de entrenamiento y el

subconjunto de evaluación (val_ds) con el 15% de las imágenes redimensionadas. Por último, el parámetro *callback* está configurado para que, tras el entrenamiento, se guarde la versión del modelo que haya obtenido el valor más alto de ROC-AUC. Estos son los datos obtenidos del entrenamiento:

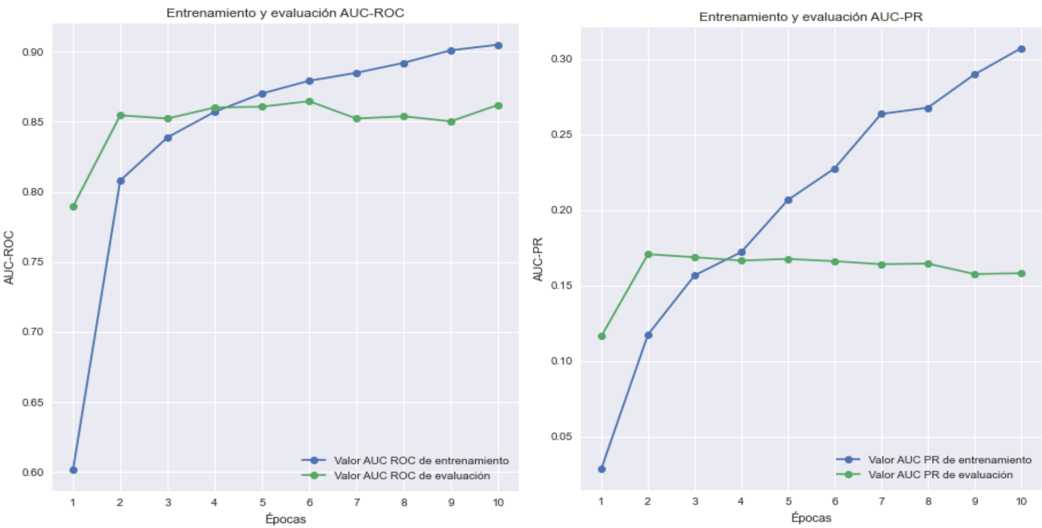


Figura 2.17 Valores de las métricas de evaluación. Modelo inicial.

Estas gráficas muestran en azul el resultado obtenido para su respectiva métrica aplicado al subconjunto de entrenamiento para cada época, y en verde esa misma métrica para el subconjunto de evaluación. Vemos que, con el paso de las épocas, las métricas de entrenamiento siempre mejoran, mientras que las de evaluación, llegan a un máximo a partir del cual no aumentan más o decaen. Esto se debe a un sobreajuste del modelo a los datos de entrenamiento. Este suceso se denomina *overfitting* y es por lo que debemos almacenar la versión del modelo que mejores resultados haya obtenido y no la versión más entrenada.

Vemos que este modelo inicial llega a su capacidad máxima de aprendizaje muy rápido. El valor máximo de AUC-PR se obtiene en la segunda época de entrenamiento, y a partir de ahí empieza a descender. El valor de AUC-ROC máximo se obtiene en la época 6, pero aun así no es mucho mayor que el valor en la época 2.

Esta situación, por la cual tras pocas épocas se llega al rendimiento máximo, se da por el *dataset* que estamos utilizando y el desbalanceo que sufre. Ya que el *dataset* cuenta con muchas imágenes, una sola época de entrenamiento supone al modelo procesar

mucha información y, por tanto, aprender muy rápido. Sin embargo, solo un 2% del *dataset* corresponde a la clase que queremos reconocer. En el subconjunto de evaluación, de 4969 imágenes, las entradas correspondientes a esta clase solo son 100. Esto puede hacer que el conocimiento adquirido con el subconjunto de entrenamiento no sea lo suficientemente variado como para generalizar a otras imágenes fuera de su subconjunto. Concluimos así que necesitamos generalizar los datos del *dataset*, permitiendo que los modelos sean capaces de reconocer un rango más amplio de imágenes.

2.3.3 Técnicas de mejora del *dataset*

La primera mejora que se ha aplicado al entrenamiento de nuestro modelo ha sido aplicar la técnica de *Data Augmentation.* La técnica de *Data Augmentation* consiste en, a partir de los datos con los que ya cuenta el *dataset* utilizado, generar más entrenadas que añadan variedad para el proceso de entrenamiento. Es la técnica más utiliza en procesos de entrenamiento de *machine learning*, sobre todo cuando se trabaja con conjuntos de datos donde no se cuenta con suficientes muestras de alguna de las clases que se quiere distinguir.

Estas técnicas pueden llegar a ser muy complejas, creando modelos de *deep learning* completos para que generen estos datos. En este trabajo nos limitamos a aplicar transformaciones a las imágenes con las que ya contamos. En concreto, al cargar los subconjuntos de entrenamiento y evaluación, aplicamos a las imágenes volteos verticales y horizontales, lo cual nos permitirá que el modelo no relaciona la posición de la lesión en la imagen con el diagnóstico que necesita. También aplicamos ajustes de brillo y contraste aleatorios, para que la clasificación sea independiente de estos parámetros. Las transformaciones aplicadas deben no dañar la propia integridad de la imagen. Por ello, no se aplican transformaciones de corrección de color ni de recorte, ya que son dos factores que podrían ser importantes para la correcta clasificación.

Se repite a continuación el entrenamiento y se comprueba si se ha obtenido alguna mejora con respecto al modelo anterior:

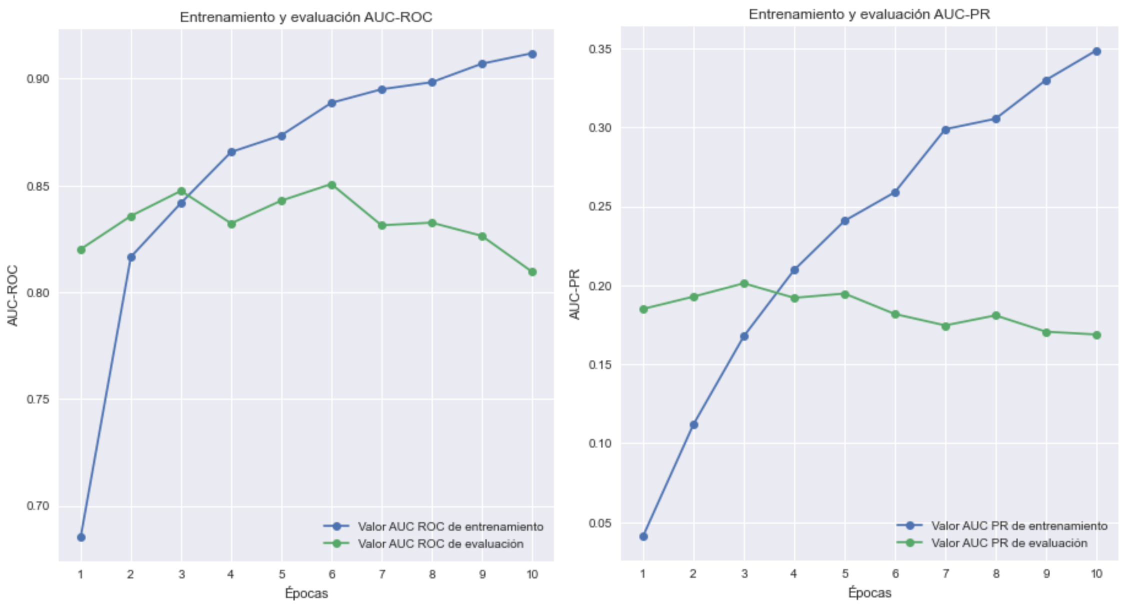


Figura 2.18 Valores de las métricas de evaluación. Modelo con data augmentation.

Comparando los resultados con el anterior modelo, vemos que ahora el mejor resultado para AUC-PR se ha obtenido en la época 3 y ha alcanzado el valor de 0,2. El valor de AUC-ROC ha ido aumentando hasta la época 7, en la cual ha empezado a decaer por el *overfitting,* y no han sido mejores que en el modelo base. Esto ocurre porque el objetivo principal de aplicar las transformaciones era crear más instancias de la clase menos numerosa. AUC-ROC no representa tan bien la mejora en la distinción de la clase menos poblada, por lo que no refleja la mejora. Por tanto, podemos concluir que aplicar *data augmentation,* aún con transformaciones simples, permite mejorar los resultados, ya que hemos aumentado el valor de AUC-PR un 15% aproximadamente. La siguiente tabla resume los resultados obtenidos:

Resultados de entrenamiento	Mejor AUC-ROC	Mejor AUC-PR
Modelo inicial	0,86	0,175
Modelo con data augmentation	0,85	0,2

Tabla 2.2 Comparación resultados de entrenamiento con data augmentation.

La segunda técnica que se aplica para generalizar el aprendizaje de nuestro modelo es la Validación Cruzada (*Cross-Validation*). Esta técnica consiste en, en vez de realizar una solo partición del *dataset* en un conjunto de entrenamiento y otro de prueba, realizar múltiples particiones con las que se entrenarán múltiples modelos. El diagnóstico final consistirá en una media de los diagnósticos que habrán dado todos los modelos, ya que

cada uno se habrá entrenado con un subconjunto distinto. Este proceso de partición del *dataset* se realiza de la siguiente manera:

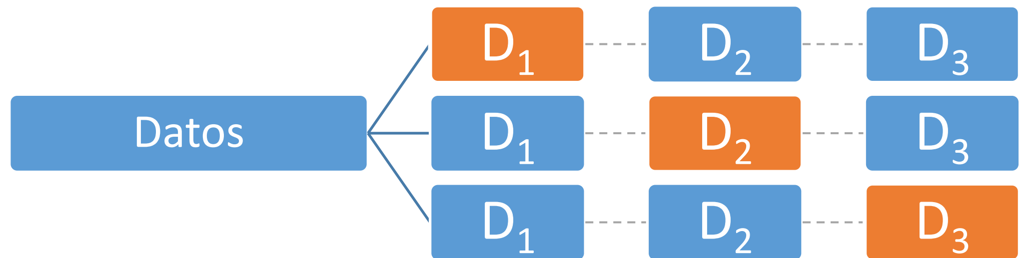


Figura 2.19 Diagrama del proceso de Cross-Validation.

Se parte de un conjunto de datos inicial, en este caso la unión de los subconjuntos de entrenamiento y evaluación (recordemos que además tenemos un subconjunto de test con otro 15% de las entradas del *dataset*). Esta unión la dividiremos en un número N de partes, que se corresponderá con el número de modelos que queramos crear. Una vez hecho esto, para cada uno de los modelos, se asigna 1 parte como subconjunto de evaluación y el resto (N-1) como subconjunto de entrenamiento. En el diagrama superior se representa en naranja la parte de los datos que se asigna como subconjunto de evaluación para cada modelo en el caso de N = 3. Vemos que estos subconjuntos seleccionados son disjuntos. Este proceso permite al modelo ser entrenado con la totalidad de los datos, y por tanto incluir más variedad de imágenes de todas las clases.

Así hemos aplicado esta técnica al modelo original, sin usar d*ata augmentation* para poder realizar comparaciones con el modelo base. En las siguientes gráficas se muestra la media de los resultados de los 3 modelos entrenados.

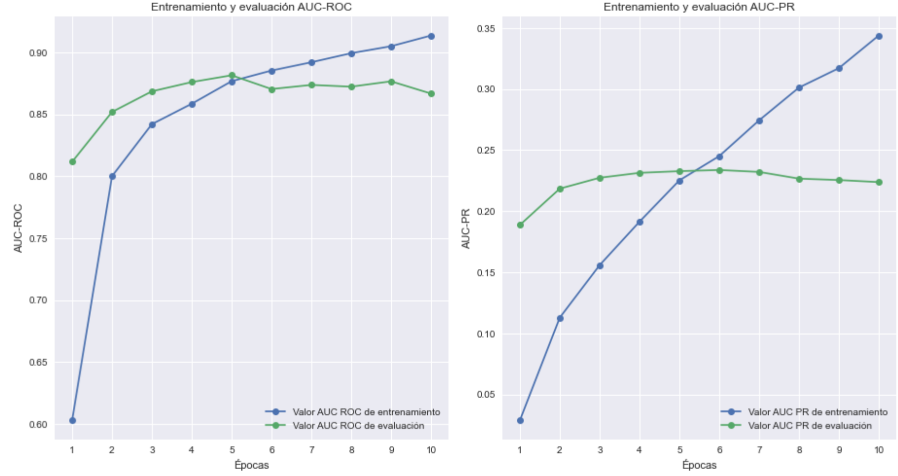


Figura 2.19 Valores de las métricas de evaluación. Modelo Cross-Validation.

A primera vista, vemos que los resultados del entrenamiento son mucho más estables que los de los modelos anteriores. Los valores óptimos se han obtenido más tarde en el entrenamiento (en torno a la época 5), y se han mejorado considerablemente los resultados en las métricas, tanto de AUC-ROC como AUC-PR. Para simplificar el uso posterior de este modelo, se guarda un único modelo que funcione con la media de los pesos de las neuronas de los 3 modelos entrenados. La siguiente tabla resume los resultados obtenidos:

Resultados de entrenamiento	Mejor AUC-ROC	Mejor AUC-PR
Modelo inicial	0,86	0,175
Modelo con data augmentation	0,85	0,2
Modelo con cross-validation	0,875	0,235

Tabla 2.3 Comparación resultados de entrenamiento con cross-validation.

Finalmente, aplicamos tanto d*ata augmentation* como *cross-validation* para obtener el modelo que usaremos finalmente en la aplicación web. Estos son los resultados del entrenamiento:

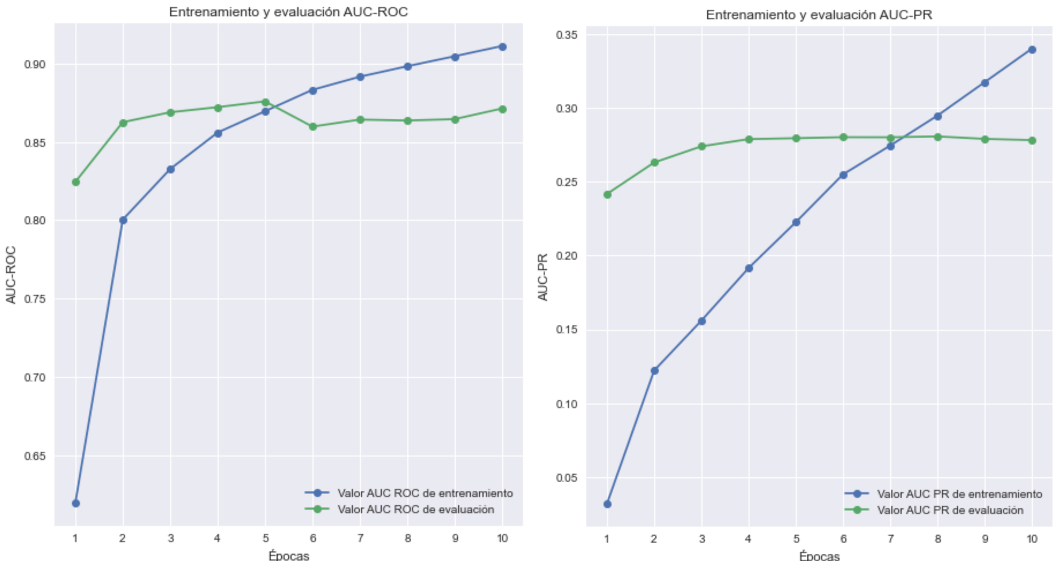


Figura 2.20 Valores de las métricas de evaluación. Modelo final.

Podemos ver que el uso de ambas técnicas ha mejorado sustancialmente el rendimiento del modelo con respecto a la primera versión. La generalización del aprendizaje se puede apreciar en que, aunque la curva PR tenga una forma similar a las otras versiones, comienza con un valor de AUC mucho mejor en la primera época. La siguiente tabla resume los resultados obtenidos:

Resultados de entrenamiento	Mejor AUC-ROC	Mejor AUC-PR
Modelo inicial	0,86	0,175
Modelo con data augmentation	0,85	0,2
Modelo con cross-validation	0,875	0,235
Modelo con data augmentation y cross validation	0,875	0,28

Tabla 2.3 Comparación resultados de entrenamiento del modelo final.

Concluimos así la elaboración de los modelos que estarán disponibles en la aplicación. Ahora, terminaremos de adaptarlos para su uso en la clasificación de imágenes individuales.

2.4 Selección de umbral

Lo último que debemos hacer para poder aplicar nuestro modelo para la predicción de imágenes es seleccionar qué umbral queremos que distinga los diagnósticos positivos de los negativos. Por defecto, se suele utilizar el valor 0.5, pero ya hemos visto que las curvas ROC y PR nos dan información sobre el rendimiento y los resultados del modelo según el umbral utilizado. El proceso de selección de umbral se denomina *Fine-Tuning* o *Threshold-Moving*, y es crucial para la correcta aplicación del modelo. En nuestro caso, nos hemos quedado con el valor que mejor mantenga los valores de precisión y sensibilidad altos, sin dar preferencia a ninguno de ellos. Este proceso en un contexto profesional requiere de expertos en la materia que trate el modelo, en este caso la dermatología, para que estudien las repercusiones médicas que puede tener aumentar o disminuir el umbral. Ese estudio queda fuera del alcance de este trabajo.

Comenzamos este proceso construyendo la curva ROC para el subconjunto de test, el cual no ha participado en ningún momento en el proceso de entrenamiento. Para ello clasificamos todas las imágenes del subconjunto utilizando el modelo y creamos una tabla con los valores predichos y esperados. Luego, se prueban una serie de umbrales y se calcula la especificidad y la sensibilidad para cada uno de ellos, obteniendo así las coordenadas de cada punto de la curva ROC. La librería *sklearn* de Python cuenta con funciones que realizan automáticamente este proceso. Una vez calculadas las coordenadas de la curva, este es el resultado:

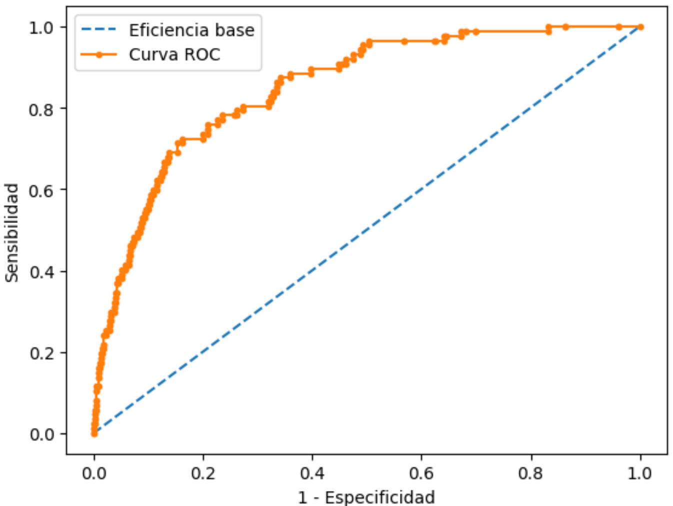


Figura 2.21 Curva ROC del modelo final.

Ahora se elige uno de estos puntos que conforman la curva para que sea el umbral utilizado. Para ello, utilizaremos como criterio la media geométrica de la sensibilidad y la especificidad. La fórmula matemática de la media geométrica es la siguiente:

$$\bar{x} = \sqrt[n]{\prod_{i=1}^{n} x_i} = \sqrt[n]{x_1 \cdot x_2 \cdots x_n}$$

Esta media es menos susceptible a valores extremos que la media aritmética y permite comparar mejor valores que puedan tener distintas magnitudes, por lo que es la que usaremos en este caso. Seleccionamos así el umbral que obtenga un mayor valor para la media geométrica.

Tras realizar el cálculo, obtenemos el valor 0,021065. Por tanto, si al clasificar una imagen utilizando este modelo, la salida es mayor que este umbral, se clasificará como perteneciente a la clase 1, un melanoma. Si es menor, se clasificará como perteneciente a la clase 0, una lesión benigna.

3

Aplicación web

En este capítulo se explicará el proceso de desarrollo de la aplicación web, basándonos en las fases de desarrollo del software que definen las metodologías ágiles. Esta metodología pasa por realizar distintas iteraciones de toma de requisitos, diseño, implementación y realización de pruebas del proyecto. Por ello, estructuraremos este capítulo en base a estas fases de desarrollo, dejando claro a qué iteración pertenecen los requisitos desarrollados.

3.1 Requisitos de la aplicación

Normalmente, trabajando con un cliente, la obtención de requisitos se realizaría dialogando con él directamente. Al no ser este el caso, vamos a repasar brevemente los objetivos que debería cumplir la aplicación y agruparlos según su propósito e importancia. A partir de este razonamiento inicial generaremos el resto de documentación necesaria para definir formalmente los requisitos.

Para empezar, el requisito fundamental de la aplicación es que permita clasificar imágenes de lesiones dermatológicas para cualquier usuario con la mayor precisión posible. Para obtener esta precisión optima, que los modelos desarrollados en este TFG no pueden alcanzar, se decide que la aplicación permita subir a los usuarios sus propios modelos de clasificación, los cuáles pueden ser muy variados. De esto se deriva que las clases que diferencie el clasificador sean también propias del usuario, ya que no todos los modelos tienen por qué distinguir entre lesiones benignas y malignas. Además, para

ayudar a los usuarios a obtener mejores modelos de clasificación, se debe permitir el acceso exclusivo para entrenamiento a *datasets* de imágenes. Éstas son las funcionalidades principales de la aplicación y correspondientes a la primera iteración de desarrollo.

Los siguientes requisitos se derivan de la idea de que existan usuarios en la aplicación. Los usuarios deben contar con un método de autenticación al entrar en la web, lo que les permite utilizar los modelos que ellos mismos hayan creado y subido, además de los propios de la web. Por tanto, se debe crear además una distinción entre modelos públicos (accesibles por cualquier usuario) y privados (solo accesibles por el usuario que los creó). Estos requisitos serán los implementados en la segunda iteración.

Por último, quedan los requisitos derivados de la gestión del contenido de la aplicación. Debido a la gran variabilidad que puede suponer que el usuario pueda subir sus propios modelos, se crea el perfil de usuario administrador, el cual puede descargar los modelos publicados por el usuario. El administrador debe verificar el correcto funcionamiento del modelo y selecciona qué *datasets* puede utilizar para su entrenamiento. De aquí también se deriva la funcionalidad de listar los modelos existentes en la web y ver las especificaciones que introdujo el usuario durante su creación. Éstos serán los requisitos implementados en la tercera iteración.

3.1.1 Requisitos funcionales

A continuación, se detallan los requisitos funcionales a desarrollar:

- **RF1**: La aplicación deberá permitir al usuario clasificar una imagen almacenada en su sistema mediante un clasificador al que tenga acceso.
- **RF2**: La aplicación deberá permitir al usuario publicar un clasificador almacenado en su sistema.
- **RF3**: La aplicación deberá permitir al usuario personalizar las clases de los clasificadores que publique.
- **RF4**: La aplicación deberá permitir al usuario entrenar sus clasificadores publicados.
- **RF5**: La aplicación deberá permitir al usuario iniciar sesión.

- **RF6**: La aplicación deberá permitir al usuario cerrar sesión.
- **RF7**: La aplicación deberá permitir al usuario visualizar la información de los clasificadores a los que tenga acceso.
- **RF8**: La aplicación deberá permitir al administrador descargar los modelos publicados por los usuarios.
- **RF9**: La aplicación deberá permitir al administrador validar los modelos de los usuarios para su uso.
- **RF10**: La aplicación deberá permitir al administrador invalidar los modelos de los usuarios para su uso.
- **RF11**: La aplicación deberá permitir al administrador asignar los *datasets* permitidos al clasificador.

3.1.2 Requisitos no funcionales

A continuación, se detallan los requisitos no funcionales que debe cumplir la aplicación:

- **RNF1**: Se deberá preservar la integridad de los *datasets* de imágenes almacenados en la aplicación.
- **RNF2**: Se deberá asegurar que los clasificadores publicados por los usuarios funcionen correctamente junto con la aplicación.
- **RNF3**: La clasificación de imágenes debe ser compatible con imágenes de distintos formatos de color (escala de grises, RGB y BGR) y dimensiones.
- **RNF4**: La aplicación deberá admitir distintas tecnologías de modelos de clasificación.
- **RNF5**: La aplicación deberá admitir clasificadores con múltiples formatos de salida.
- **RNF6**: La aplicación será sencilla y accesible para el uso de un usuario estándar.
- **RNF7**: La aplicación deberá contar con tiempos de respuesta acordes a los de una aplicación web estándar.

3.1.3 Casos de uso

A continuación, se muestra un diagrama de casos de uso con los 2 tipos de actores que diferenciamos, usuarios y administradores:

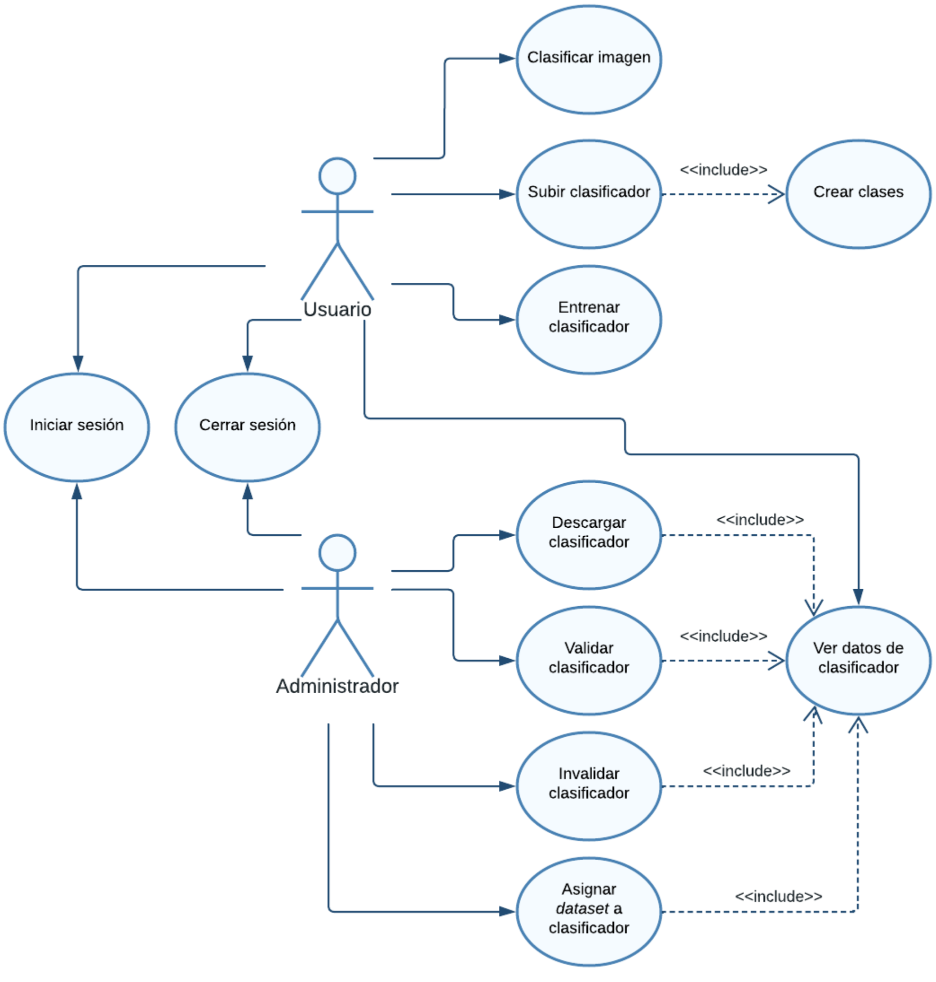


Figura 3.1 Diagrama de casos de uso para usuarios y administradores.

Se realiza un análisis de cada caso de uso:

Caso de Uso	Clasificar imagen (RF1)
Descripción	El usuario clasifica una imagen de su sistema.
Iteración	Primera iteración.
Precondición	1. El usuario se encuentra en la vista de inicio.
Escenario principal	1. El usuario accede a la vista de clasificación. 2. El sistema muestra la vista de clasificación. 3. El usuario selecciona la opción 'Select an image'. 4. El sistema muestra el selector de archivos. 5. El usuario selecciona la imagen de su sistema. 6. El sistema muestra la lista de clasificadores a los que tiene acceso el usuario.

	7. El usuario selecciona el clasificador que quiera. 8. El sistema habilita la opción 'Classify'. 9. El usuario selecciona la opción 'Classify'. 10. El sistema realiza la clasificación de la imagen seleccionada con el clasificador seleccionado. 11. El sistema muestra el resultado de diagnóstico obtenido por el clasificador para la imagen seleccionada.
Postcondición	1. El usuario se encuentra en la vista de resultados de la clasificación.
Escenario alternativo	10.b. La imagen seleccionada no es compatible con el clasificador seleccionado. El sistema muestra un mensaje de error al usuario. Volver a paso 2.

Tabla 3.1 Caso de uso de RF1.

Caso de Uso	**Subir clasificador (RF2)**
Descripción	El usuario sube a la aplicación un clasificador que tenga almacenado en su sistema.
Iteración	Primera iteración.
Precondición	1. El usuario ha iniciado sesión. 2. El usuario se encuentra en la vista de inicio.
Escenario principal	1. El usuario accede a la vista de subir clasificador. 2. El sistema muestra la vista de subir clasificador. 3. El usuario rellena el formulario con los datos de su clasificador y selecciona la opción 'Next'. 4. Se realiza el caso de uso RF3 (Crear clases). 5. El sistema guarda el clasificador. 6. El sistema muestra la vista correspondiente al clasificador creado.
Postcondición	1. El clasificador se ha creado con sus clases asociadas. 2. El clasificador no está validado para su uso.

	3. El usuario se encuentra en la vista de información sobre el clasificador creado.
Escenario alternativo	3.b El usuario ha seleccionado la opción 'Model not trained'. Saltar a paso 5. 5.b. Alguno de los datos introducidos por el usuario no es válido. La aplicación muestra un mensaje de error al usuario. Volver a paso 2.

Tabla 3.2 Caso de uso de RF2.

Caso de Uso	Crear clases (RF3)
Descripción	El usuario crea las clases que distinguirá su clasificador
Iteración	Primera iteración
Precondición	1. El usuario ha iniciado sesión. 2. El usuario se encuentra en la vista de subir clasificador. 3. El usuario ha seleccionado la opción 'Model trained'.
Escenario principal	1. El usuario selecciona la opción 'Next'. 2. El sistema muestra la vista de crear clases. 3. El usuario rellena el formulario con los datos de las clases que distingue su clasificador y selecciona 'Upload'. 4. El sistema guarda las clases.
Postcondición	1. Las clases se han creado.
Escenario alternativo	4.b. Alguno de los datos introducidos por el usuario no es válido. La aplicación muestra un mensaje de error al usuario. Volver a paso 3.

Tabla 3.3 Caso de uso de RF3.

Caso de Uso	Entrenar clasificador (RF4)
Descripción	El usuario sube el clasificador para realizar su entrenamiento.
Iteración	Primera iteración
Precondición	1. El usuario ha iniciado sesión. 2. El usuario se encuentra en la vista de inicio.

Escenario principal	1. El usuario accede a la vista de entrenar clasificador. 2. El sistema muestra la vista de entrenar clasificador con la lista de clasificadores permitidos. 3. El usuario selecciona el clasificador que quiera. 4. El sistema muestra la lista de *datasets* permitidos para el clasificador seleccionado. 5. El usuario selecciona el dataset que quiera y selecciona la opción 'Train'. 6. El sistema muestra un mensaje de tiempo estimado de entrenamiento. 7. El sistema realiza el entrenamiento del clasificador.
Postcondición	1. El sistema crea un nuevo clasificador entrenado. 2. El sistema asocia las clases del *dataset* seleccionado al nuevo clasificador.

Tabla 3.4 Caso de uso de RF4.

Caso de Uso	Iniciar sesión (RF5)
Descripción	El usuario inicia sesión
Iteración	Segunda iteración
Precondición	1. El usuario se encuentra en la vista de inicio.
Escenario principal	1. El usuario selecciona la opción 'Log in' 2. El sistema muestra la vista de inicio de sesión. 3. El usuario introduce sus datos de acceso. 4. El sistema inicia la sesión con los datos introducidos. 5. El usuario muestra la vista de inicio.
Postcondición	1. El usuario ha iniciado sesión. 2. El usuario se encuentra en la vista de inicio.
Escenario alternativo	4.b. Alguno de los datos introducidos por el usuario no es válido. La aplicación muestra un mensaje de error al usuario. Volver a paso 3.

Tabla 3.5 Caso de uso de RF5.

Caso de Uso	Cerrar sesión (RF6)
Descripción	El usuario cierra sesión
Iteración	Segunda iteración
Precondición	1. El usuario ha iniciado sesión. 2. El usuario se encuentra en la vista de inicio.
Escenario principal	1. El usuario selecciona la opción 'Log out' 2. El sistema cierra la sesión del usuario.
Postcondición	1. El usuario ha cerrado sesión. 2. El usuario se encuentra en la vista de inicio.

Tabla 3.6 Caso de uso de RF6.

Caso de Uso	Visualizar información de clasificador (RF7)
Descripción	El usuario accede a la vista de uno de los clasificadores a los que tiene acceso.
Iteración	Segunda iteración
Precondición	1. El usuario ha iniciado sesión. 1. El usuario se encuentra en la vista de inicio.
Escenario principal	1. El usuario accede a la vista de búsqueda de clasificador. 2. El sistema muestra la vista de búsqueda de clasificador. 3. El sistema muestra los clasificadores a los que el usuario tiene acceso. 4. El usuario selecciona el clasificador que quiera. 5. El sistema muestra la vista de detalles del clasificador.
Postcondición	1. El usuario se encuentra en la vista de detalles del clasificador seleccionado.

Tabla 3.7 Caso de uso de RF7.

Caso de Uso	Descargar clasificador (RF8)
Descripción	El administrador descarga el clasificador en su sistema.
Iteración	Tercera iteración
Precondición	1. El administrador se encuentra en la vista de inicio.

Escenario principal	1. Se realiza el caso de uso RF7. 2. El administrador selecciona la opción 'Download' 3. El sistema descargará el clasificador seleccionado en el sistema del administrador.
Postcondición	1. El administrador tiene descargado el clasificador en su sistema.

Tabla 3.8 Caso de uso de RF8.

Caso de Uso	Validar clasificador (RF9)
Descripción	El administrador valida un clasificador publicado por un usuario.
Iteración	Tercera iteración
Precondición	1. El clasificador se encuentra invalidado. 1. El administrador se encuentra en la vista de inicio.
Escenario principal	1. Se realiza el caso de uso RF7. 2. El administrador selecciona la opción 'Validate'. 3. El sistema valida el modelo para su uso. 4. El sistema oculta la opción 'Validate'. 5. El sistema muestra la opción 'Unvalidate'.
Postcondición	1. El clasificador está validado para su uso.

Tabla 3.9 Caso de uso de RF9.

Caso de Uso	Invalidar clasificador (RF10)
Descripción	El administrador invalida un clasificador publicado por un usuario.
Iteración	Tercera iteración
Precondición	1. El clasificador se encuentra validado. 1. El administrador se encuentra en la vista de inicio.
Escenario principal	1. Se realiza el caso de uso RF7. 2. El administrador selecciona la opción 'Unvalidate'. 3. El sistema valida el modelo para su uso.

	4. El sistema oculta la opción 'Unvalidate'. 5. El sistema muestra la opción 'Validate'.
Postcondición	1. El clasificador está invalidado para su uso.

Tabla 3.10 Caso de uso de RF10.

Caso de Uso	Asignar *datasets* a un clasificador (RF11)
Descripción	El administrador asociado los *dataset* que quiera a un clasificador
Iteración	Tercera iteración
Precondición	1. El administrador se encuentra en la vista de inicio.
Escenario principal	1. Se realiza el caso de uso RF7. 2. El sistema muestra la lista de *datasets* disponibles. 3. El administrador selecciona los *datasets* compatibles con el clasificador. 4. El sistema asocia los *datasets* al clasificador.
Postcondición	1. El clasificador tiene asociados los *datasets* seleccionados.

Tabla 3.11 Caso de uso de RF11.

3.2 Diseño de la aplicación

A continuación, se recogen las decisiones de diseño tomadas para satisfacer los requisitos especificados de la aplicación.

3.2.1 Arquitectura

Arquitectura de la aplicación web

La arquitectura utilizada para implementar la aplicación web es una arquitectura REST (Representational State Transfer). Esta arquitectura se caracteriza por identificar y manipular los recursos de la aplicación mediante URIs (Uniform resource identifier). Las URis siguen un formato uniforme, que permite identificar fácilmente el propósito de la misma. La manipulación de los recursos de la aplicación es indirecta, ya que se trabaja con representaciones de ellos. Estas representaciones se envían al cliente mediante mensajes.

Una arquitectura REST está formada por un cliente y un servidor independientes, los cuáles se comunican mediante mensajes HTTP con un conjunto conocido de operaciones. El contexto del cliente no se almacena en el servidor, por lo que toda la información necesaria para realizar las operaciones debe estar contenida en la propia petición, formada por la URL, los parámetros, las cabeceras y el cuerpo.

Arquitectura del servidor

El servidor está estructurado en 3 capas diferenciadas:

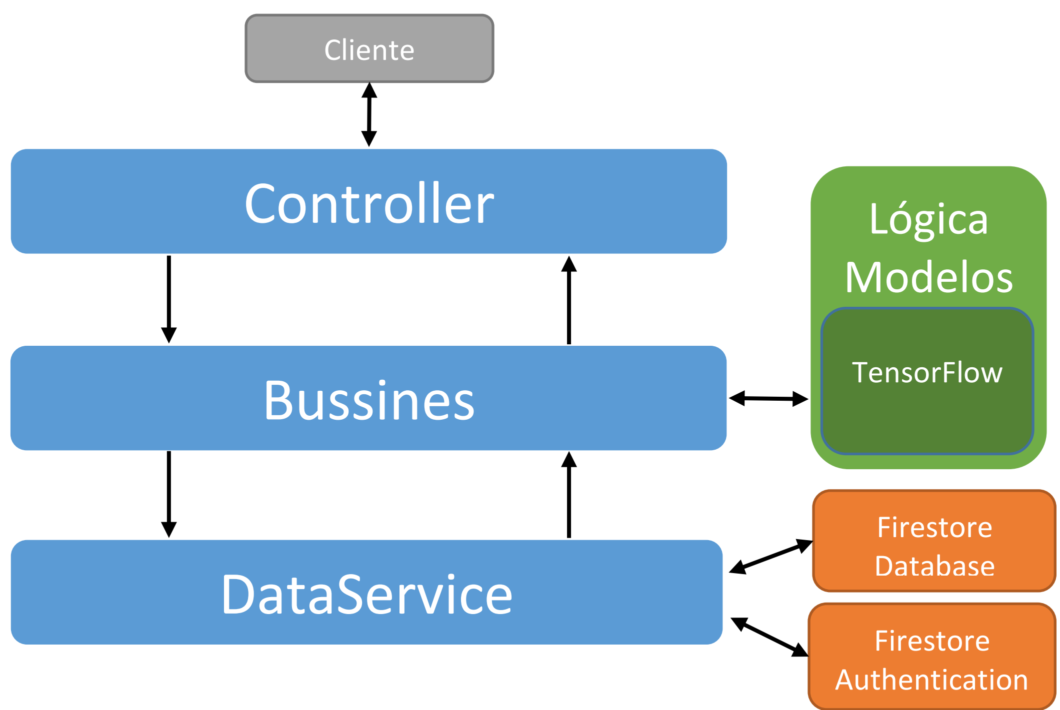


Figura 3.2 Arquitectura del servidor web.

- ***Controller***: Se comunica únicamente con el cliente, mediante las peticiones HTTP, y con la capa *Bussines*, mediante llamadas a las funciones que implementa. Esta capa recoge las URIs a las que se pueden realizar peticiones desde el cliente. Hace una primera transformación de los datos de formato JSON a los tipos de datos utilizados por Python. *Controller* también manda los mensajes de respuesta al cliente, tanto en caso de un funcionamiento exitoso como en caso de error.

- ***Bussines***: Implementa la transformación de los datos, recibidos de la capa *Controller*, y realiza peticiones a la capa *DataService* para obtener datos almacenados en la base de datos y realizar la autenticación mediante OAuth Firebase. También realiza llamadas a los módulos donde se implementan los métodos relacionados con el uso de los modelos de clasificación (Clasificar una imagen y entrenar un modelo). Tras realizar las peticiones a la capa *DataService* y a los módulos externos, crea el contenido de la respuesta para el cliente y la manda a *Controller* para que comunique el resultado.
- ***DataService***: Implementa las peticiones a la base de datos, tanto de acceso a la información almacenada como su modificación, y la autenticación del usuario. Esta capa es importante que esté separada de la capa *Business* y su interfaz debe ser lo más estándar posible, facilitando así posibles futuros cambios de las tecnologías utilizadas. Esta capa está implementada únicamente para su funcionamiento con una base de datos Firestore Database y una autenticación mediante Firestore Authentication. Ante un cambio a otra tecnología, como por ejemplo *MySQL*, únicamente sería necesario modificar la implementación de sus funciones, pero no su interfaz, por lo que la capa Bussines no tendría que modificarse.

Como hemos mencionado, además de esta estructura de capas, el servidor cuenta con módulos extra que implementen las funciones más complejas que corresponde al uso de los modelos de clasificación. Estos módulos están ligados a las tecnologías utilizadas para implementar los modelos de clasificación. En el caso de este TFG, se ha implementado un único módulo que implementa estas funciones utilizando *TensorFlow*. Sin embargo, este módulo puede ser implementado, por ejemplo, utilizando librerías como *skLearn* o *PyTorch*, ampliando así la compatibilidad de la aplicación con distintas tecnologías.

Arquitectura del cliente

El cliente web implementa la vista de nuestra aplicación web. Ya que hemos utilizado el *framework* React.js para su implementación, se utilizará una arquitectura basada en componentes React débilmente asociados (*loosely coupled components*). Esta

arquitectura se basa en identificar componentes individuales que conforman la interfaz de usuario e implementarlos evitando las dependencias entre ellos. Por ejemplo, como veremos más adelante, la vista de información de un clasificador contará con un apartado con los datos cargados del servidor del clasificador y otro apartado, solo disponible para usuarios administradores, con un listado de los *datasets* compatibles con ese clasificador. Como vemos, aunque ambos componentes se encuentran en la misma vista, su información y funcionamiento es totalmente distinto. Por tanto, se implementa un componente individual para cada uno. Así, podrán ser reutilizados en otros apartados de la aplicación, lo que permite agilizar los procesos de implementación y corrección de errores.

Por tanto, la arquitectura del cliente React se basa en componentes anidados, que pueden contenerse unos a otros sin depender entre ellos. El cliente cuenta con un componente React por cada vista, que cargará componentes más simples. Los componentes que forman las vistas de la aplicación (y los más altos en la jerarquía de componentes), realizarán las peticiones al servidor empleado las URIs definidas por este. Una vez se obtenga la información del servidor, se distribuirá a los componentes de menor jerarquía que se utilicen.

Sin embargo, el flujo de información en la vista de la aplicación no solo fluye desde los componentes más altos a los más bajos. Cuando un usuario interactúa con cualquier componente, por ejemplo haciendo click en un elemento de una lista, los componentes más bajos podrán mandar la información a los más altos para que estos se actualicen o realicen una petición.

A continuación, se muestra un diagrama de secuencia que ilustra el comportamiento del cliente y sus componentes React durante la ejecución del caso de uso 7 (Visualizar la información de un clasificador):

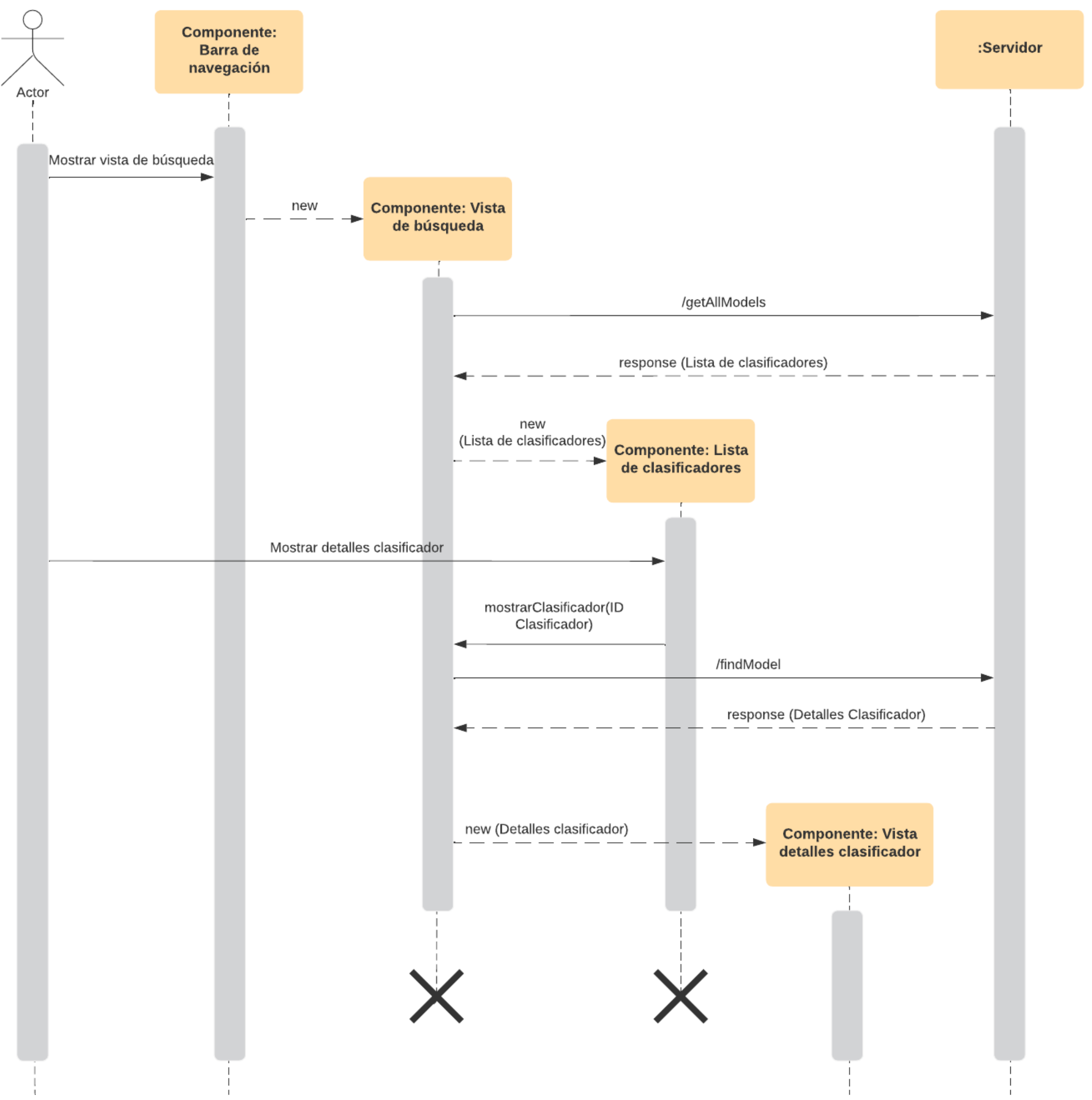


Figura 3.3 Diagrama de secuencia del caso de uso 7.

3.2.2 Protocolo para la creación y uso de los clasificadores

Los clasificadores que publiquen los usuarios de la aplicación pueden estar implementados de distintas formas, tener distintos formatos de entrada y de salida, y trabajar sobre distinto número de clases. Por tanto, hemos necesitado definir un protocolo lo más amplio posible que permita ejecutar correctamente estos clasificadores. Para ello se han tomado las siguientes decisiones, cumpliendo así los requisitos no funcionales 3, 4 y 5:

- El clasificador se publica en un único archivo .h5, un archivo estándar para almacenar modelos de *machine learning* de distintas tecnologías, incluyendo TensorFlow, skLearn y PyTorch.
- Para cada modelo se almacena el tipo de canal de color con el que trabaja (escala de grises, RGB o BGR) y las dimensiones de la imagen de entrada. Con esta información, al realizar una clasificación, el servidor realiza las transformaciones necesarias a la imagen para que se ajuste al formato de entrada del clasificador, permitiendo así su uso con cualquier imagen.
- Para cada modelo se almacena la tecnología necesaria para su ejecución. Con esta información, el servidor puede utilizar el módulo correcto que implemente las funciones para esa tecnología. Para este TFG solo se ha diseñado el módulo para Tensorflow, pero se dejará abierto a la implementación para más tecnologías.
- Para cada modelo se almacenan las clases que distingue y el formato de salida que utilice. En problemas de clasificación existen 4 tipos básicos de salida, para los cuales la aplicación será compatible. Éstos son:
    - *Binary Probability*: Devuelve un único valor que típicamente representa la probabilidad de que la entrada pertenezca a la clase 1. Normalmente se utiliza el valor 0,5 para distinguir entre la clase 0 y la clase 1. Sin embargo, admite un valor umbral distinto.
    - *Multicategorical Probability*: Devuelve un *array* de valores correspondientes a la probabilidad de que la entrada pertenezca a cada una de las clases. Se asume que la entrada pertenece a la clase con mayor probabilidad de pertenencia. Debe aceptar un número variable de clases.
    - *Integer Encoding*: Devuelve un valor entero que indica la clase a la que pertenece la entrada. Debe aceptar un número variable de clases.
    - *One-Hot Encoding*: Devuelve un *array* de ceros, con un valor 1 en el índice de la clase a la que pertenece la entrada. Debe admitir un número variable de clases.
- Las clases que distinga el clasificador se definen en el momento de publicar el mismo. Al realizar el entrenamiento de un clasificador mediante un *dataset*, las clases que distingue serán reemplazadas por las clases que contiene el *dataset*.

Es necesario hacer esto para asegurar un correcto funcionamiento de los modelos durante las clasificaciones.

Toda esta información es necesaria que el usuario la introduzca durante la publicación del clasificador para asegurar su funcionamiento. Tras esto, un administrador deberá validar que el modelo que se publique cumple con las especificaciones introducidas, ya que no es posible derivar todas estas características desde el propio archivo del modelo. Con esto se cumplirían los requisitos no funcionales 1 a 5, permitiendo un uso seguro de los clasificadores.

3.2.3 Diseño de base de datos

La base de datos utilizada es Firestore Database, una herramienta de Firebase. Firestore no es tan potente como otras opciones de bases de datos relacionales como MySQL. Sin embargo, ya que nuestra aplicación no hace un uso intensivo de accesos a base de datos ni necesita aplicar peticiones muy complejas, Firestore es más que suficiente para realizar este trabajo.

Aunque Firestore sea una base de datos no relacional, se debe realizar una especificación previa de las colecciones con las que contará y de los campos que almacenarán los documentos. Éstas son las colecciones definidas con los campos de sus documentos:

- **Users**: recoge los datos de un usuario.
  - Name (String): Nombre.
  - Email (String): Correo electrónico
  - Admin (Bool): Indica si el usuario es administrador.

- **Models**: recoge los datos de los clasificadores publicados en la aplicación.
  - Name (String): Nombre público mostrado en los listados.
  - Description (String): Descripción del clasificador.
  - UserID (String): ID en la base de datos del usuario que lo creó.

- Validated (Bool): Indica si el clasificador ha sido validado por un administrador.
- Trained (Bool): Indica si el clasificador ya ha sido entrenado.
- Public (Bool): Indica si el clasificador es accesible por otros usuarios.
- X, Y (Integer): Indican las dimensiones de las imágenes aceptadas por el clasificador.
- Channel (Integer): Representa el formato de color de las imágenes aceptados por el clasificador (Escala de grises, RGB o BGR).
- NumberClasses (Integer): Indica el número de clases que distingue el clasificador.
- PredictionFormat (Integer): Representa el formato de salida utilizado por el clasificador (binary probability, multicategorical probability, integer encoding, one-hot encoding).
- Technology (Integer): Representa la tecnología que utiliza el clasificador. Solo admite el valor 1, correspondiente a TensorFlow.
- Threshold (Integer): Si el clasificador es binario, indica el valor de salida a partir del cual una imagen se clasificará como clase 1. Se utiliza en caso de realizar clasificación binaria.
- ClassesIDs (*Array*): Contiene los IDs de la base de datos de las clases que distingue el clasificador.
- DatasetsIDs (Array): Contiene los IDs de la base de datos de los *datasets* que puede utilizar el modelo para entrenarse.

- **Classes**: recoge los datos de las clases distinguidas por algún clasificador.
  - Name (String): Nombre de la clase.
  - Description (String): Descripción de la clase.

- **Datasets**: recoge los datos de los *datasets* utilizables por los clasificadores.
  - Name (String): Nombre del *dataset*.
  - Description (String): Descripción del *dataset*.
  - ClassesIDs (*Array*): Contiene los IDs de la base de datos de las clases que distingue el *dataset*.

- ImagePath (String): Indica la dirección donde se encuentran las imágenes del *dataset*.
- TrainPath (String): Indica la dirección del documento con la información necesaria para el entrenamiento (Datos de entrenamiento, evaluación y prueba).

3.2.4 Navegación

Como ya hemos mencionado, la interfaz de la aplicación se ha realizado mediante componentes React en el cliente. A continuación se muestra un diagrama de flujo de interacción, donde se muestra cómo se realiza la navegación a través de estos componentes:

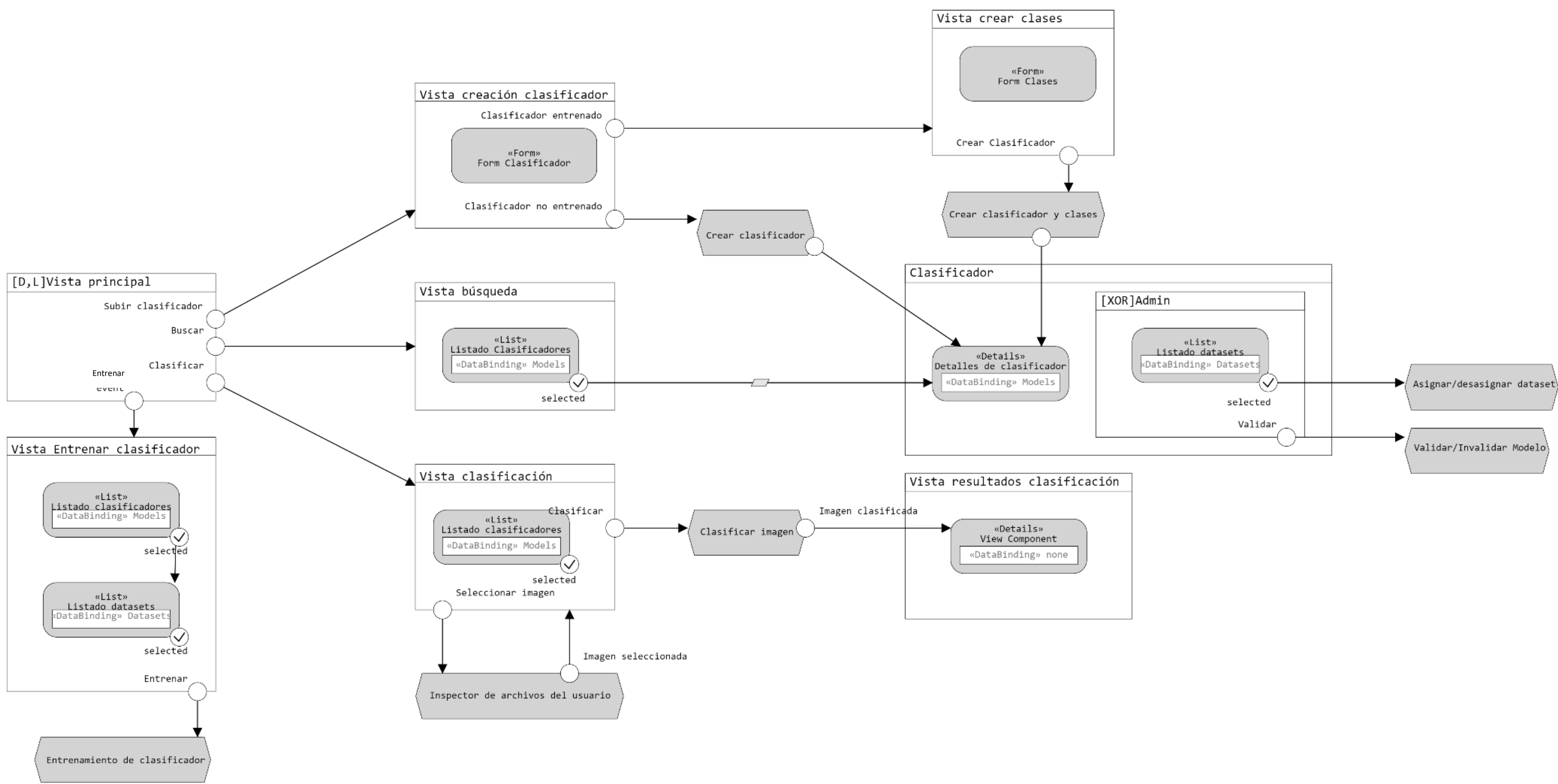


Figura 3.4 Diagrama de flujo de interacción del cliente web.

3.3 Implementación y pruebas

En este apartado se explica cómo se han implementado las funcionalidades mencionadas anteriormente haciendo uso de sus respectivas tecnologías. Se centra en la implementación de la comunicación cliente-servidor mediante *endpoints*, el método de autenticación del usuario mediante Firebase y el almacenamiento y ejecución de modelos mediante Tensorflow, ya que son los apartados de mayor relevancia para este trabajo y sirven como referencia para el resto de funcionalidades.

3.3.1 Comunicación cliente-servidor

En el cliente implementado en React, cada componente define la vista en formato HTML y la lógica asociada a la vista en lenguaje JavaScript. Para realizar las peticiones al servidor se utiliza la función *fetch*, la cual permite definir una dirección, una operación HTTP y el cuerpo y cabeceras de la petición. A continuación se muestra un ejemplo:

```
function handleSubmit(datasetID){
    const formData = new FormData();
    formData.append('datasetID', datasetID)
    formData.append('modelID', selectedModel.id)

    fetch(
        "http://localhost:5000/trainModel",
        {
        method: 'POST',
        body: formData,
        }
    )
    .then(response => response.json())
    .then(data => console.log(data))
}
```

Figura 3.5 Petición para entrenar modelo en el cliente.

En concreto, esta es la función definida para realizar la petición al servidor de entrenamiento de un clasificador. Para ello primero se define el cuerpo de la petición, el cual guardará como un diccionario el ID del *dataset* de entrenamiento y el ID del clasificador. Luego se llama a la función *fetch* con la dirección del servidor en el puerto local, se define como una petición POST y se asigna el cuerpo definido. El cliente en este momento espera a recibir una respuesta del servidor en formato JSON. En este caso, por simplicidad, únicamente muestra los datos obtenidos por consola.

Esta función *handleSubmit* se asigna a un botón el cual, ante un click del usuario, lanza la función con los datos almacenados en el componente (selectedModel) y los parámetros de entrada definidos en el botón (datasetID).

En el lado del servidor se ha utilizado Flask, un framework de Python para la creación de aplicaciones web. Flask basa su funcionamiento en definir *endpoints* para ejecutar funciones de Python. Estos *endpoints* han sido definidos en la capa *Controller* de nuestra aplicación, y sirven de URIs para formar nuestra API REST. Vemos como se define el *endpoint* al que se accede desde el cliente en el ejemplo anterior:

```
72  @app.route("/trainModel", methods= ['POST'])
73  def trainModel():
74      if request.method == 'POST':
75          return business.trainModel(request.form)
76
```

Figura 3.6 Endpoint del servidor para entrenar modelo.

Para empezar, hace falta establecer una ruta, compuesta por la URI y el método HTTP que acepta, en este caso POST. Esta ruta se asigna a una función de Python, la cual recibirá a través *request* la petición que ha llegado del cliente. En este caso vemos que, si la petición es POST, se llama a la función *trainModel* de la capa *Business*, pasándole como parámetros la información almacenada en *request*. Esta información es almacenada en *body* en el cliente. Siguiendo esta estructura, se ha creado una ruta para cada funcionalidad que queremos que nuestro servidor preste.

La capa *Business* utiliza lógica básica de Python para hacer transformaciones a los datos y llamar a las funciones auxiliares necesarias de la capa *DataService* y el módulo de TensorFlow. Para crear el mensaje de respuesta, solo es necesario crear un diccionario con la información que se quiera mandar de vuelta.

```
132  def trainModel(data):
133      model = dataService.findModel(data['modelID'])
134      dataset = dataService.findDataset(data['datasetID'])
135      modelPath = "./modelos/"+data['modelID']+"/"+os.listdir("./modelos/"+data['modelID'])[0]
136  
137  
138      modelTrained, history, threshold= tensorflowFunctions.trainModel(data['modelID'],
139                                                                      modelPath,
140                                                                      dataset['imagesPath'],
141                                                                      dataset['trainPath'],
142                                                                      dataset['valPath'],
143                                                                      dataset['testPath'])
144      model['classesIDs'] = dataset['classesIDs']
145      model['trained'] = 1
146      model['count'] = model['count'] + 1
147      model['name'] = model['name']+"_v" + str(model['count'])
148      newModelID = dataService.saveModelDB(model, model['count'], model['datasets'], threshold, dataset['classesIDs'])
149      savePath = "./modelos/"+newModelID+"/"+os.listdir("./modelos/"+data['modelID'])[0]
150      modelTrained.save(savePath)
151      content = {
152          'history' : history
153      }
154      dataService.updateDB(newModelID, content, 'modelos')
155      return history
```

Figura 3.7 Lógica de la capa business para entrenar modelo.

En esta función de *Business* se toma la información recibida de *Controller*. Con ella, se busca en la base de datos mediante *DataService* el modelo y el *dataset* requeridos. Luego se define el *path* donde se encuentra almacenado el modelo y se llama a la función de entrenamiento utilizando TensorFlow. Tras esto, se modifican algunos parámetros en la base de datos, como las clases que distingue el modelo, se guarda el modelo almacenado y se crea la respuesta para el servidor. En este caso se crea un diccionario con el historial del entrenamiento del modelo. Por último, se almacena el nuevo modelo en la base de datos y se devuelve la respuesta al cliente.

Este proceso se repite para cada *checkpoint* definido, sirviendo así la información requerida por el cliente.

3.3.2 Autenticación

Firebase Authentication permite fácilmente realizar autenticación de usuarios mediante vinculación de cuentas como Google, Facebook o GitHub. Para este caso se ha implementado la autenticación con una cuenta de Google, ya que es la más común para cualquier usuario. Firebase asocia a cada cuenta de Google vinculada un UID, el cual podremos utilizar, por ejemplo, para identificar al usuario en la base de datos.

Para acceder a este UID, el inicio de sesión con Google o cualquier otro método genera un token de acceso, válido solo durante un corto periodo de tiempo. Para iniciar la

sesión del usuario, se manda este token desde el cliente al servidor, donde se valida utilizando métodos propios de Firebase. Estos métodos permiten obtener el UID del usuario que ha iniciado sesión, concediendo así los permisos de acceso que ese usuario tenga. Así, por ejemplo, se cargan correctamente en el cliente los clasificadores que el usuario pueda utilizar.

```
42    function signWithGoogle(){
43        const auth = getAuth()
44        const provider = new GoogleAuthProvider()
45
46        if(auth.currentUser) {
47            signOut(auth)
48        }
49
50        signInWithPopup(auth, provider)
51        .then((result) => {
52            const name = result.user.displayName
53            const email = result.user.email
54            const isNew = getAdditionalUserInfo(result).isNewUser
55
56
57            auth.currentUser.getIdToken()
58            .then(data => {
59                loginRequest(data, name, email, isNew)
60            })
61
62        })
```

Figura 3.8 Obtención y envío del token de verificación en el cliente.

La función de la figura 3.8 inicia sesión con Google en el cliente. De este inicio de sesión, se obtiene el token de acceso, el nombre del usuario en su cuenta de Google, su correo electrónico y si es la primera vez que se utiliza esa cuenta en la aplicación. Mandando estos datos al servidor mediante una petición, como la vista en el apartado, se puede tanto crear un nuevo usuario como iniciar sesión de un usuario ya existente.

```
211  def loginUser(token, data):
212      decodedToken = dataService.verifyFirebaseToken(token)
213      uid = decodedToken['user_id']
214
215      doc = dataService.findUser(uid)
216      if(doc.exists):
217          content = doc.to_dict()
218          content['id'] = doc.id
219          return jsonify(content)
220      else:
221          content = {
222              'name' : data['name'],
223              'email' : data['email'],
224              'admin' : False
225          }
226          savedID = dataService.saveUserDB(content, uid)
227          content['id'] = savedID
228          return jsonify(content)
```

Figura 3.9 Inicio de sesión de un usuario en el servidor.

En el servidor, con esta información, primero se decodifica el token de Firebase y se obtiene el UID del usuario. Luego se busca el usuario de la base de datos que tenga ese UID. Si existe, se obtiene su información y se devuelve al cliente como un diccionario JSON. Si no existe, se toman los datos de nombre y correo obtenidos de Google, se almacenan en la base de datos como un usuario no administrador, y se devuelve al cliente.

3.3.3 Envío, almacenamiento y uso de clasificadores

Para poder implementar todas las funciones descritas por los requisitos de apartados anteriores, hemos tenido que implementar métodos de envío de clasificadores mediante peticiones HTTP, métodos de almacenamiento de los clasificadores en local y módulos especializados en ejecutar clasificadores implementados con una tecnología específica. Estas tres son las funcionalidades principales necesarias para poder implementar todos los requisitos funcionales. A continuación, explicamos cómo se ha abordado cada uno de estos apartados:

Envío y almacenamiento de clasificadores

Los clasificadores implementados con una tecnología de d*eep learning* no son más que una especificación de las capas que tiene el modelo, las funciones de activación que utiliza y los pesos que ha calculado durante entrenamientos previos. Esta información es transversal a todas las tecnologías de m*achine learning* y por tanto se han definido formatos de archivo que permiten guardar y cargar esta información. El más común de estos es .h5, y es el formato utilizado para el almacenamiento de los modelos en este TFG. Utilizar este formato de archivo nos permite comprimir toda la información para el uso del clasificador en un único archivo, lo que hace mucho más sencillo su envío.

```
84  @app.route('/uploadModel', methods=['POST'])
85  def uploadModel():
86      if request.method == 'POST':
87          return business.uploadModel(request.files, request.form, PATH_MODELS)
88
```

Figura 3.10 Endpoint del servidor para publicar un modelo.

Una petición HTTP permite añadir archivos a su *body* al igual que hacemos con el resto de datos. Al realizar el envío de un archivo desde el cliente al servidor, en la variable *request* se almacena en el parámetro *files.*

```
178     for file in MultiDict(files):
179         fileDataStructure = MultiDict(files).getlist(file)[0]
180         fileDataStructure.filename = str(savedID) + "/" + fileDataStructure.filename
181         dataService.saveFile(fileDataStructure, folderPath)
```

Figura 3.11 Implementación de guardado de un archivo en el servidor.

Para obtener los archivos adjuntos, podemos iterar sobre ellos y obtener para cada uno un *fileDataStructure*, el cual nos permite acceder a su nombre (*filename*). Con el nombre del archivo y una ruta podemos iniciar el proceso de guardado.

Para guardar el archivo, hemos decidido crear una carpeta que tenga como nombre el identificador utilizado en la base de datos de Firebase para identificar a ese modelo. Dentro de esa carpeta se almacena finalmente el modelo publicado. Esto puede permitir en el futuro implementar compatibilidad con clasificadores compuestos por varios modelos que clasifiquen al mismo tiempo. Sin embargo, no es totalmente necesario en un inicio. Ya vimos que existen alternativos para emular este comportamiento con un único modelo con el clasificador implementado utilizando *Cross-Validation*. Por último, nos hemos asegurado de que el nombre del archivo subido no sea peligroso. Esto puede deberse al uso de caracteres ilegales en directorios, que puedan tener resultados inesperados al leerlos en nuestro sistema. Para ello, Flask nos brinda la función, *secure_filename*, la cual nos devuelve el nombre del archivo quitando posibles caracteres que puedan llevar a errores. Finalmente, la siguiente función almacena el clasificador:

```
55  def saveFile(file, folderPath):
56      app.config['UPLOAD_FOLDER'] = folderPath
57      path = os.path.dirname(file.filename)
58      path2 = os.path.join(app.config['UPLOAD_FOLDER'], path)
59      if not os.path.exists(path2):
60          os.mkdir(path2)
61      filename = os.path.join(path, secure_filename(os.path.basename(file.filename)))
62      file.save(os.path.join(app.config['UPLOAD_FOLDER'], filename))
```

Figura 3.12 Implementación de guardado de un clasificador en el servidor.

Al realizar el entrenamiento de un clasificador, el nuevo modelo entrenado se guardará con un nuevo identificador de la misma forma que se guardó su original, ya que la información de sus pesos se ha actualizado. Por lo tanto, se le crea una nueva carpeta con su ID y se cambia el nombre público que tendrá en la aplicación para no ser confundidos.

Cuando un administrador quiera descargar un modelo para poder validarlo, el servidor crea un archivo comprimido de la carpeta donde se almacena el modelo. En el cliente, al recibir el archivo comprimido de la respuesta HTTP, se crea un *blob* de JavaScript, el cual permite descargar el archivo únicamente asignándole un nombre y el tipo de archivo del que se trata, en este caso un .zip:

```
19    function downloadZip(){
20        var fetchURL = "http://localhost:5000/downloadModel/"+id
21        fetch(fetchURL,
22            {
23            method: 'GET'
24            }
25        ).then((res) => {return res.blob()})
26        .then(blob =>{
27          download(blob, 'model_'+ id +'.zip', 'application/zip');
28        })
29    }
```

Figura 3.13 Descarga de un clasificador en el cliente.

Uso de clasificadores

Los modelos publicados por los usuarios tienen dos operaciones principales, clasificar una imagen y entrenarse con un *dataset*. En ambos casos, se utilizan los parámetros introducidos por el usuario durante el proceso de publicación del modelo (descritos en el apartado de base de datos).

Para la clasificación de una imagen, primero debemos realizarle 2 transformaciones para que sea compatible con el modelo elegido. Se cambian las dimensiones de la imagen para que se ajusten a la altura y anchura definidas por el usuario, y los canales de color utilizados, por ejemplo, a RGB o escala de grises. Una vez que se obtiene la imagen compatible, se manda un resultado de clasificación del modelo. Este resultado depende del formato de predicción definido, por lo que se debe ajustar la respuesta para que siga

un formato constante que el cliente pueda procesar. Para ello, se envía al cliente un diccionario con dos claves, el valor obtenido del modelo y el id de la clase a la que pertenece la imagen. A continuación podemos ver cómo se trata el resultado para los 4 tipos de formato, de la manera definida anteriormente:

```
115     #Clasificacion
116     predictions_single = model(image).numpy()
117     if(modDic['predictionFormat'] == 1): #Binary probability
118         result = 0 if predictions_single[0][0] < modDic['threshold'] else 1
119         dict = {"result" : result,
120                 "classID" : modDic['classesIDs'][result]
121                 }
122     elif(modDic['predictionFormat'] == 2 or modDic['predictionFormat'] == 4): #Multicategorical probability y One-Hot Encoded
123         dict = {"result" : str(np.argmax(predictions_single[0])),
124                 "classID" : str(np.argmax(predictions_single[0]))
125                 }
126     elif(modDic['predictionFormat'] == 3): #Integer Encoded
127         dict = {"result" : str(predictions_single[0][0]),
128                 "classID" : str(predictions_single[0][0])
129                 }
130     return jsonify(dict)
```

Figura 3.15 Implementación de los formatos de salida de una clasificación en el servidor.

Para realizar el entrenamiento, debemos utilizar los parámetros ya definidos en el capítulo 2. Asignamos una función de pérdida y las métricas pertinentes y se empieza el entrenamiento con el *dataset* seleccionado. Si el modelo es binario, tras este proceso, se realiza el cálculo del umbral óptimo, tal y como se explicó anteriormente. Este entrenamiento tendrá dos resultados: el modelo propiamente dicho ya entrenado, el cual se almacenará como se ha definido en el apartado anterior, y un histórico del entrenamiento, el cual será un diccionario compuesto por los valores obtenidos para cada métrica y cada época de entrenamiento. Este histórico es accesible desde el cliente donde se mostrará para que el usuario pueda analizar los resultados:

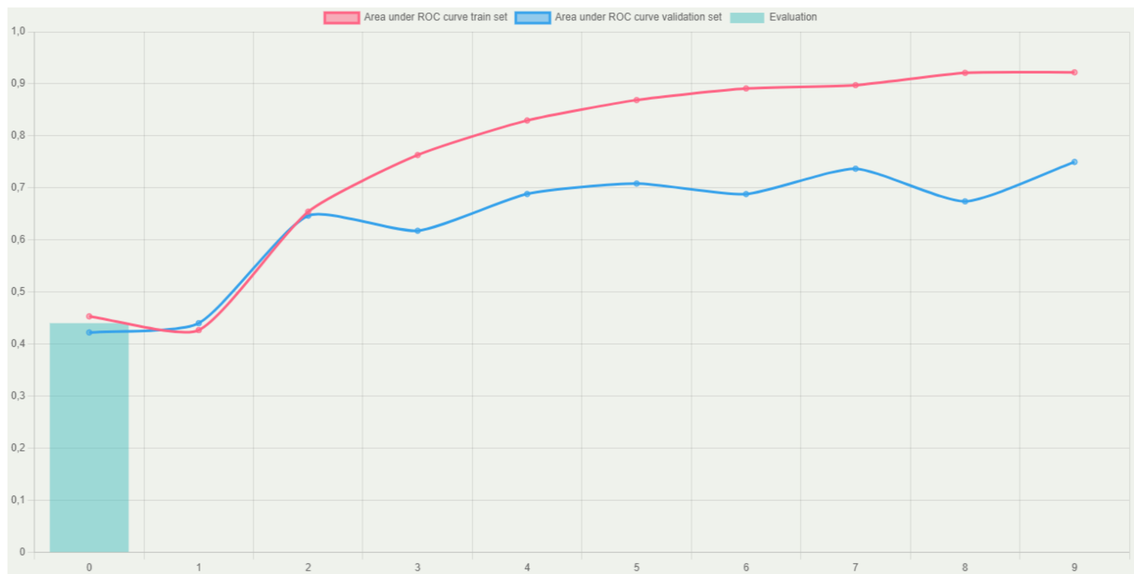


Figura 3.16 Gráfica mostrada al usuario en el cliente web del entrenamiento.

3.3.4 Pruebas

Durante la implementación de la aplicación se han realizado una serie de pruebas para asegurar su correcto funcionamiento. Principalmente se han hecho pruebas de dos tipos:

Pruebas Postman del servidor

Tras la implementación de cada funcionalidad que brinda el servidor mediante un *endpoint*, se ha diseñado una prueba con Postman. Postman permite definir peticiones HTTP y obtener el resultado devuelto por el servidor. Estas peticiones permiten definir la URI a la que se acceden, la operación HTTP, el cuerpo de la petición y las cabeceras. La respuesta, en este caso del servidor, se muestra en formato JSON para poder compararse con el resultado esperado. Por tanto, cualquiera de las funcionalidades implementadas puede probarse fácilmente usando esta herramienta. A continuación, vemos un par de ejemplos:

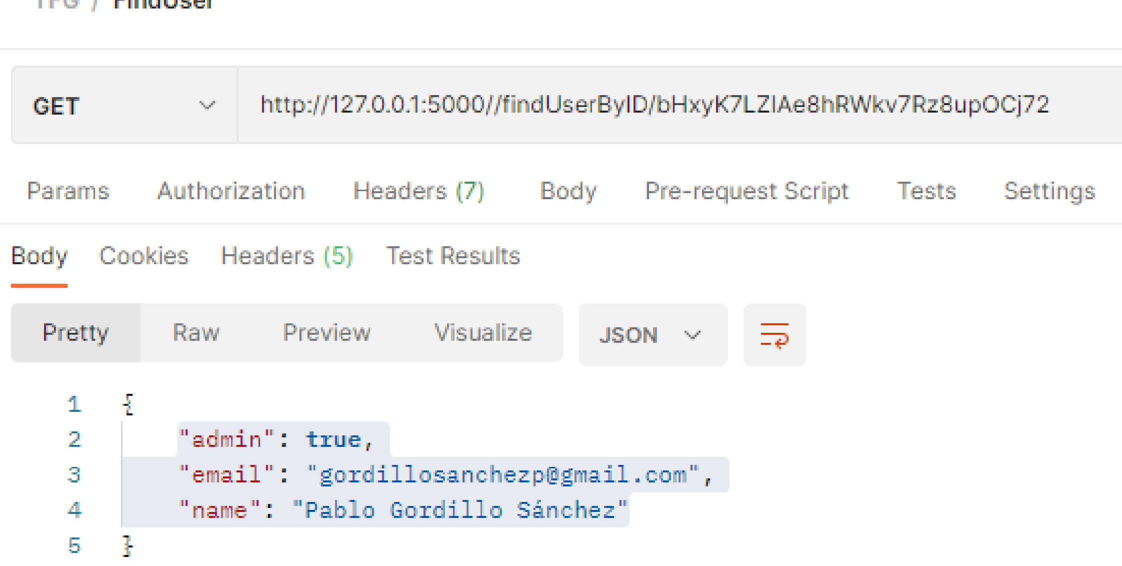


Figura 3.17 Prueba Postman para obtener un usuario por ID.

La anterior petición permite obtener un usuario almacenado en la base de datos a partir de su ID. Vemos cómo el servidor devuelve el resultado en formato JSON con 3 parámetros: nombre, correo y los permisos con los que cuenta el usuario.

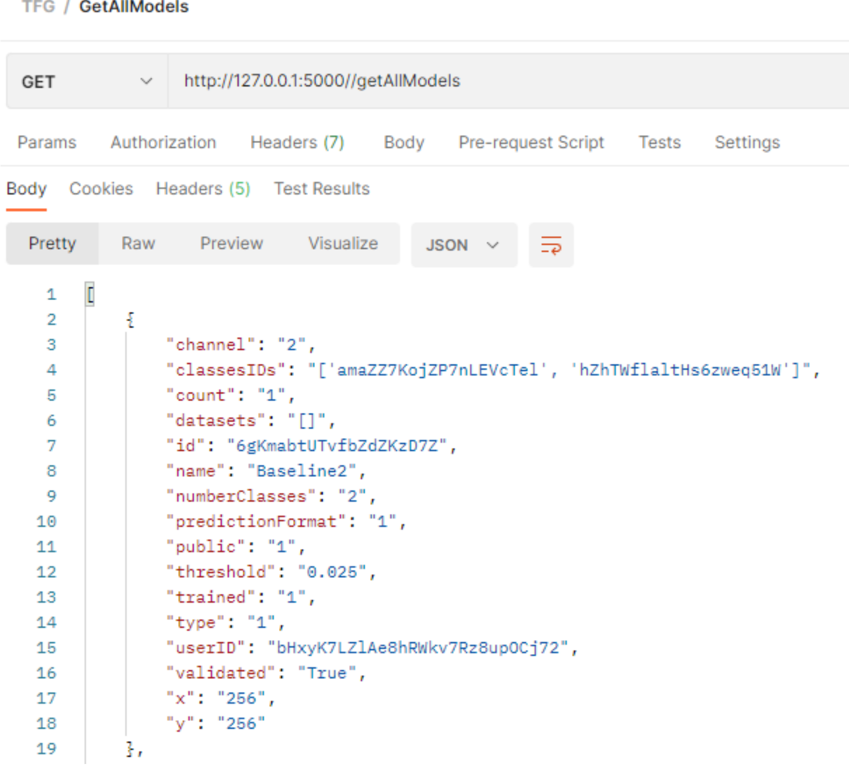


Figura 3.18 Prueba Postman para obtener todos los clasificadores almacenados.

Esta petición obtiene todos los clasificadores almacenados en la base de datos con los parámetros descritos anteriormente. El resultado comprende una lista de varios modelos, pero se ha acortado en la imagen para su más sencilla comprensión.

Pruebas de simulación del cliente

Una vez que nos hemos asegurado de que las respuestas del servidor son correctas, se han realizado pruebas de simulación real con el cliente. Esta simulación consiste en imitar los escenarios descritos en el apartado de requisitos e imitarlos utilizando las herramientas del cliente. Este método permite probar el correcto funcionamiento de los componentes con los que el usuario interactúa y que las peticiones al servidor se envían y leen correctamente. Al ser la arquitectura de React tan modular, este método permite encontrar rápidamente el componente exacto que no realiza bien su labor. Vamos a ver un ejemplo con el escenario del RF1 (clasificar una imagen):

1. Accedemos a la vista de clasificación y seleccionamos una imagen.

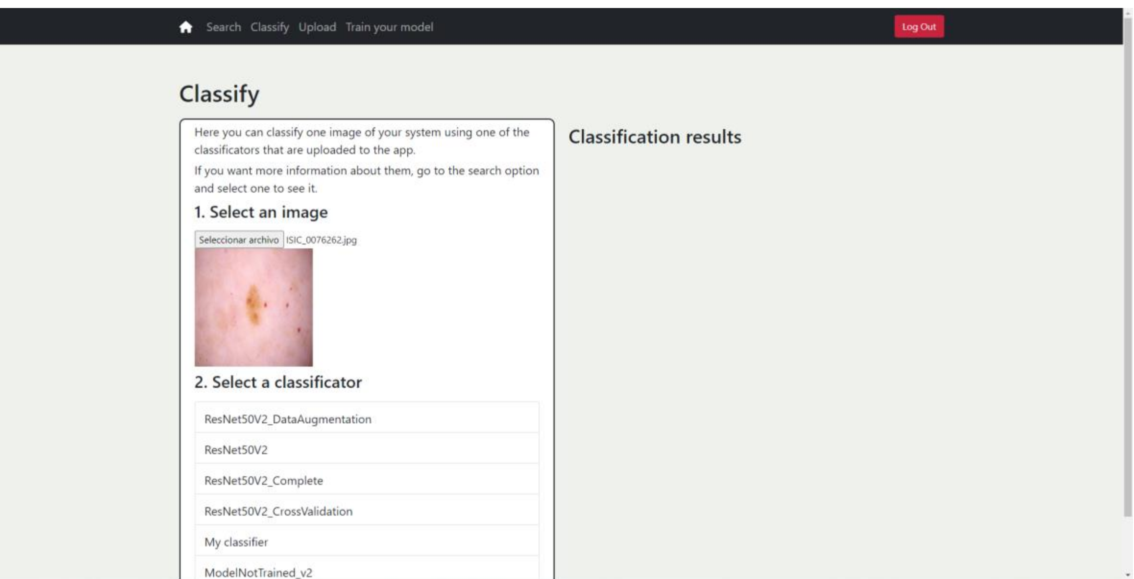


Figura 3.19 Captura 1 para la prueba de RF1.

2. Seleccionamos el clasificador que queramos utilizar.

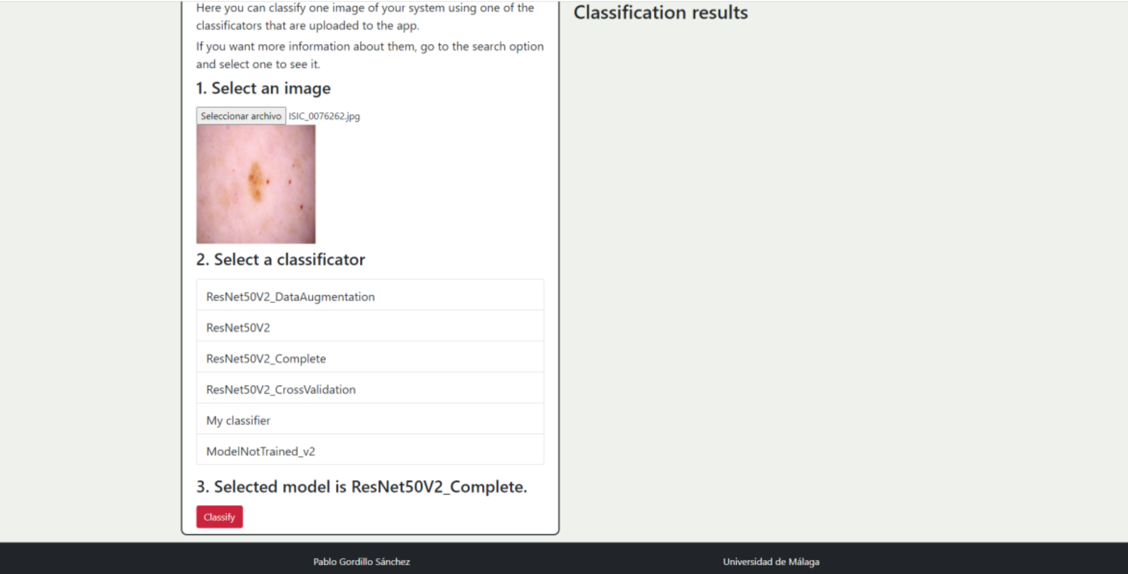


Figura 3.20 Captura 2 para la prueba de RF1.

3. Obtenemos un resultado mostrado por pantalla.

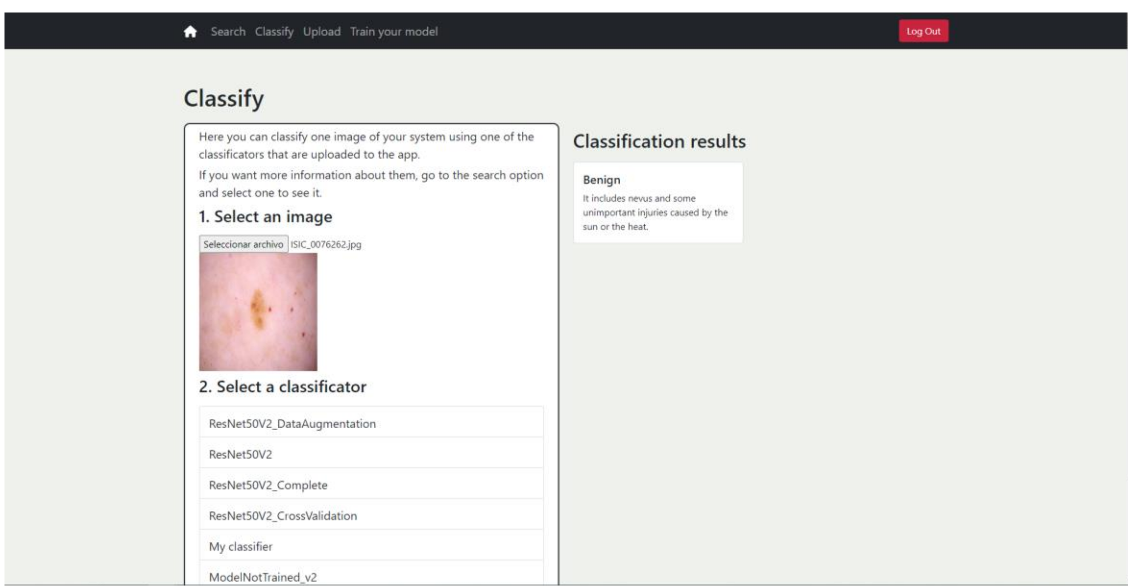


Figura 3.21 Captura 3 para la prueba de RF1.

4

Conclusiones

En esta memoria se ha expuesto el trabajo realizado para el desarrollo de una aplicación web para la publicación de modelos de clasificación de imágenes, centrándose en los métodos de diagnóstico dermatológicos. Finalmente, podemos decir que se han cumplido los principales requisitos que se habían recogido a partir del análisis del problema. Contamos con una aplicación que permite clasificar imágenes de cualquier tipo, contando con modelos especializados en la clasificación de melanomas; se permite a cualquier usuario, que haga previo registro, que suba sus propios modelos tanto para su propio uso como para el resto de usuarios; se pueden utilizar *datasets* de imágenes para el entrenamiento de los modelos, previa autorización de un administrador. Este trabajo ha unido muchos campos de la informática y la ingeniería del software, permitiendo desarrollar una aplicación donde se pueden utilizar modelos de *deep learning* mediante una interfaz web sencilla. Además, la aplicación ha quedado abierta para posibles futuras mejoras, ya que la arquitectura de la misma permite fácilmente añadir modelos más complejos, compatibilidad con otras tecnologías de inteligencia artificial y componentes del cliente React para añadir funcionalidades para los usuarios.

A continuación se detallarán los conceptos principales aprendidos y aplicados durante la elaboración de este TFG, así como líneas futuras que podrían realizarse sobre la aplicación web y los modelos de clasificación que han sido desarrollados.

4.1 Conceptos desarrollados

Este trabajo ha servido como punto de entrada para el alumno a la aplicación de algoritmos de redes neuronales en un problema real, utilizando grandes cantidades de datos representativos del problema a resolver. En este sentido, el cambio con respecto a los trabajos desarrollados durante la carrera ha supuesto la necesidad de aprender sobre muchos campos muy diversos y profundizar en herramientas con las que ya se estaba familiarizado.

El desarrollo de los modelos de clasificación ha necesitado de la adquisición de conocimientos especializados en clasificación de imágenes para diagnóstico de enfermedades. Además, este conocimiento es puramente teórico y transversal a cualquier algoritmo o tecnología de clasificación utilizada, por lo que puede ser aplicable en un futuro en problemas distintos. En concreto, sobre redes neuronales y *deep learning*, se ha profundizado en la arquitectura de las mismas, y se ha aprendido sobre cómo afectan distintos parámetros y técnicas de entrenamiento al rendimiento de los modelos. Además, ha sido muy importante el mantener y ejecutar una metodología propia del campo en el que se ha trabajado, como es CRISP-DM, más alejado de las metodologías ágiles de desarrollo de software.

Para el desarrollo de la aplicación web ya se contaba con un mayor conocimiento de la metodología ágil e iterativa aplicada. Sin embargo, al ser una aplicación de una escala mayor en comparación con las elaboradas en otras asignaturas, ha sido muy necesario realizar las fases de la metodología de desarrollo ágil. Definir el alcance del proyecto ha sido un proceso que se ha repetido durante gran parte del desarrollo, ya que era muy complicado hacer una previsión del esfuerzo necesario para la implementación de toda la aplicación desde el comienzo. Los procesos de diseño y pruebas han permitido asegurar que los requisitos se implementaban correctamente, asegurando así el progreso a siguientes iteraciones.

En cuanto a las tecnologías web empleadas ha sido necesario un conocimiento mucho mayor sobre su funcionamiento para realizar esa aplicación. Aprender sobre conceptos como el funcionamiento de los componentes React o la ejecución de algoritmos de

clasificación en Python sobre imágenes han permitido ampliar el funcionamiento de la aplicación para cumplir con los objetivos requeridos. Aunque estos conocimientos sean más concretos a las tecnologías utilizadas, cabe destacar que son tecnologías que están ahora mismo en auge, con una gran proyección de futuro en sus respectivos campos.

Por último, queremos destacar que el proceso de definir cómo se iba a realizar la unión de todas estas tecnologías y metodologías en una sola aplicación ha sido de los procesos más interesantes durante el desarrollo. Para ello se ha necesitado razonar sobre cada paradigma de programación utilizado, qué tipo de información se necesitaba para trabajar en cada uno de ellos, y cómo debía ser su comunicación. Aquí se incluye, entre otras muchas cosas, el proceso de investigación de los formatos estándar de salida de los modelos de clasificación, el proceso de envío y almacenamiento de los modelos en cliente y servidor y el desarrollo de módulos especializados en la ejecución de estos modelos con una tecnología concreta (como se ha realizado con TensorFlow).

Por tanto, concluimos que este TFG ha servido tanto para profundizar en campos de investigación y desarrollo muy distintos entre sí, como para afianzar las bases metodológicas de un proceso de desarrollo web e investigación.

4.2 Líneas futuras

Todavía se puede y se debe realizar trabajo en la aplicación web desarrollada durante este TFG para que pueda ser publicada y utilizada por cualquier usuario. Muchas de estas características no requieren para su implementación de conocimientos distintos a los ya demostrados en otras partes del trabajo, por lo que se les ha dado menor prioridad ante otras funcionalidades más interesantes para el desarrollo de un TFG. A continuación, se numeran las principales características sobre las que se puede profundizar en un futuro:

1. Los usuarios administradores deberían tener un mayor número de funciones en una aplicación real. Actualmente su única labor es validar el correcto funcionamiento de los modelos publicados por los usuarios. Así, se deberían añadir funcionalidades CRUD tanto sobre usuarios como sobre los modelos que

se publiquen. Además, deberían tener un proceso de registro en la aplicación propio y análogo al de los usuarios estándar.

2. El proceso de entrenamiento de un modelo debería ser más personalizable. No todos los modelos tienen un índice de entrenamiento igual, por lo que poder elegir qué métricas de evaluación utilizar o el número de épocas de entrenamiento puede suponer una gran mejora. Además, también se podría ampliar la información devuelta por el servidor al cliente de este proceso, mostrando al usuario más gráficas y métricas que las actuales.
3. Ya que la aplicación está preparada para ello, sería recomendable implementar más módulos que permitan la compatibilidad de la aplicación con más tecnologías, como PyTorch o skLearn. Este proceso es sencillo, ya que únicamente se debería parametrizar las funciones de clasificación y entrenamiento desarrolladas con estas librerías para que sean compatibles con la información enviada por el cliente, al igual que se ha hecho con TensorFlow. También está la posibilidad de ampliar su compatibilidad a otros tipos de algoritmos, como los descritos en el apartado 2.1.
4. Por último, con todas estas funcionalidades ya desarrolladas, se debería plantear el proceso de despliegue de la aplicación. Para ello será importante evaluar el rendimiento de la aplicación ante grandes volúmenes de datos y cómo gestionar el almacenamiento de los modelos a una escala mayor. Sin embargo, ya que esta funcionalidad esta encapsulada en la capa *DataService* de la aplicación, si fuera necesario realizar algún cambio, éste no afectaría al funcionamiento de la mayor parte de la aplicación.

Referencias

[1] *ISIC Archive*. (s. f.). ISIC Archive. https://www.isic-archive.com/#!/topWithHeader/tightContentTop/about/aboutIsicBackground

[2] *Tratamiento del melanoma (PDQ®)–Versión para pacientes*. (s. f.). Instituto Nacional del Cáncer. https://www.cancer.gov/espanol/tipos/piel/paciente/tratamiento-melanoma-pdq

[3] *CRISP-DM: La metodología para poner orden en los proyectos - Sngular*. (s. f.). Sngular. https://www.sngular.com/es/data-science-crisp-dm-metodologia/

[4] Colaboradores de los proyectos Wikimedia. (2013, 13 de julio). *Cross Industry Standard Process for Data Mining - Wikipedia, la enciclopedia libre*. Wikipedia, la enciclopedia libre. https://es.wikipedia.org/wiki/Cross_Industry_Standard_Process_for_Data_Mining

[5] *Procesos software*. (s. f.). Introducción a la Ingeniería del software.

[6] *Árboles de Decisión con ejemplos en Python - IArtificial.net*. (s. f.). IArtificial.net. https://www.iartificial.net/arboles-de-decision-con-ejemplos-en-python/

[7] Facultad de Comunicación, Universidad de Murcia. (2019, 10 de junio). *Técnicas y usos en la clasificación automática de imágenes*. http://eprints.rclis.org/38798/1/Clasificacion%20imagenes%20ISKO2019.pdf

[8] Contributors to Wikimedia projects. (2005, 4 de enero). *Backpropagation - Wikipedia*. Wikipedia, the free encyclopedia. https://en.wikipedia.org/wiki/Backpropagation

[9] *tf.keras.applications.resnet50.ResNet50 | TensorFlow v2.9.1*. (s. f.). TensorFlow. https://www.tensorflow.org/api_docs/python/tf/keras/applications/resnet50/ResNet50

[10] *ImageNet*. (s. f.). ImageNet. https://www.image-net.org/about.php

[11] *Welcome to Flask — Flask Documentation (2.2.x)*. (s. f.). Welcome to Flask — Flask Documentation (2.2.x). https://flask.palletsprojects.com/en/2.2.x/

[12] *React – Una biblioteca de JavaScript para construir interfaces de usuario*. (s. f.). React – Una biblioteca de JavaScript para construir interfaces de usuario. https://es.reactjs.org/

[13] *React-Bootstrap*. (s. f.). React-Bootstrap Documentation. https://react-bootstrap.github.io/

[14] *Firebase Documentation*. (s. f.). Firebase. https://firebase.google.com/docs

[15] Narkhede, S. (2018, 26 de junio). *Understanding AUC - ROC Curve*. Medium. https://towardsdatascience.com/understanding-auc-roc-curve-68b2303cc9c5

[16] *A Gentle Introduction to Threshold-Moving for Imbalanced Classification*. (s. f.). Machine Learning Mastery. https://machinelearningmastery.com/threshold-moving-for-imbalanced-classification/

Apéndice A

Manual de Instalación

Instalar servidor web con Flask

Para poder ejecutar el servidor en tu sistema sigue los siguientes pasos:

1. Instala Python 3.9.7 en el sistema desde https://www.python.org/downloads/.
2. Descarga el proyecto de Python proporcionado.
3. Crea un entorno virtual de Python dentro del proyecto utilizando el comando:

```
python3 -m venv venv
```

4. Activa el entorno virtual con el comando:

```
venv\Scripts\activate.bat
```

5. Descargua el archivo *requirements.txt* y ejecuta el siguiente comando para instalar las dependencias:

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

6. Una vez haya terminado ejecuta el siguiente comando para lanzar el servidor en un puerto del sistema:

```
flask run
```

Mientras esté funcionando, el servidor recibirá las peticiones del cliente en el puerto http://127.0.0.1:5000. Si se cambiara la dirección del puerto debe cambiarse la dirección a la que el cliente mande sus peticiones.

Instalar cliente web con React

Para poder ejecutar el cliente web en tu sistema sigue los siguientes pasos:

1. Instala Node.Js versión 16 o superior y npm utilizando el siguiente comando:

```
npm install -g npm
```

2. Descarga el proyecto de React proporcionado e instala las dependencias listadas en el archivo *requirements.txt.*
3. Una vez se hayan instalado todas las dependencias ejecuta el siguiente comando:

```
npm start
```

El cliente web se abrirá en el puerto *localhost:3000* y se lanzará una ventana del navegador con la página principal de la aplicación.

Para que la aplicación funcione correctamente es necesario que tanto el cliente como el servidor estén en ejecución al mismo tiempo, ya sea en la misma máquina o desplegando alguno de ellos.

Apéndice B

Manual de Usuario

A continuación se detallan los pasos a seguir para ejecutar cada una de las funcionalidades de la aplicación utilizando la interfaz del cliente web.

1. Registrarse o hacer log in/log out

1. Estando en la página principal, pulsa el botón "Log In".

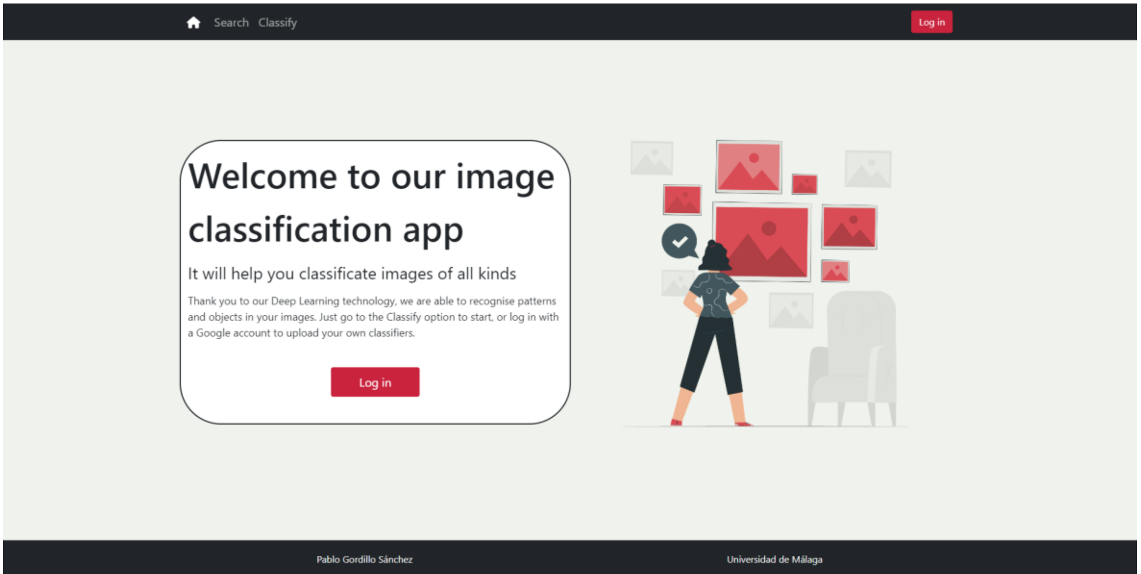


Figura B.1 Captura pantalla de inicio.

2. Aparecerá una pantalla para seleccionar la cuenta de Google con la que se quiera iniciar sesión. Pulsa en la cuenta que prefieras.
    a. Si no se había utilizado antes la cuenta seleccionada, se creará un usuario nuevo en la aplicación y se iniciará sesión.
    b. Si ya se había utilizado antes la cuenta seleccionada, se iniciará sesión cargando los datos almacenados de el usuario.
3. Tras esto te llevará a la pantalla de búsqueda de clasificadores. Mientras la sesión esté iniciada, el botón de "Log Out" de la barra de navegación superior permitirá cerrar la sesión de usuario.

2. Visualizar información de un clasificador

1. Estando en la página principal, pulsa en el botón "Search" de la barra de navegación superior.
2. En pantalla se mostrarán los clasificadores a los que tienes acceso, listado por nombre. Pulsa en el modelo que prefieras para visualizar su información.

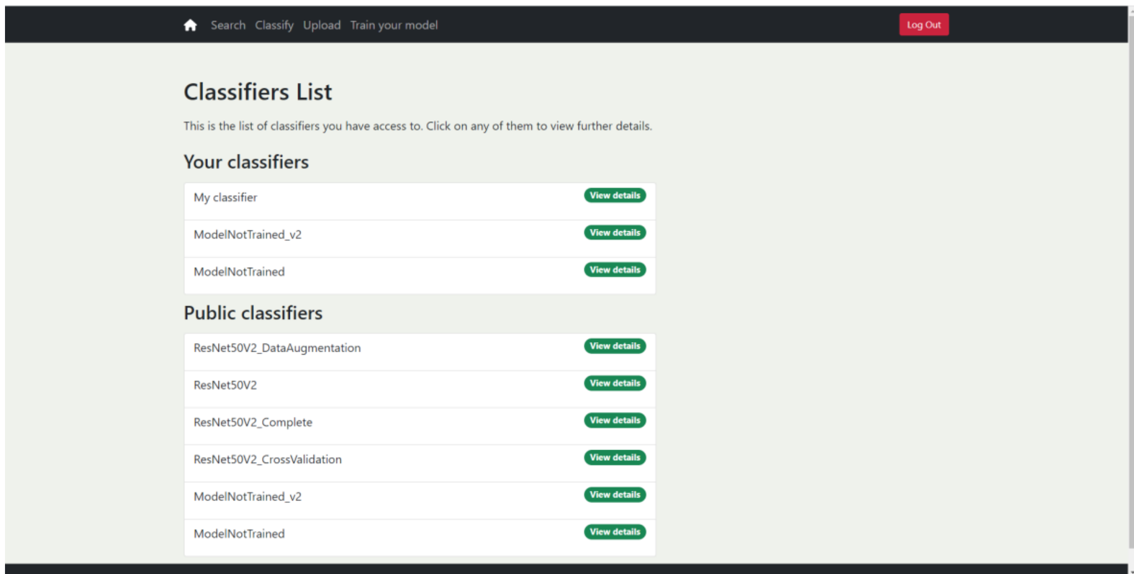


Figura B.2 Captura ventana Search.

3. En la pantalla aparecerá la información del clasificador: su nombre, canal de color que utiliza, tecnología, el tamaño de las imágenes que acepta, su historial de entrenamiento si lo tiene, etc.

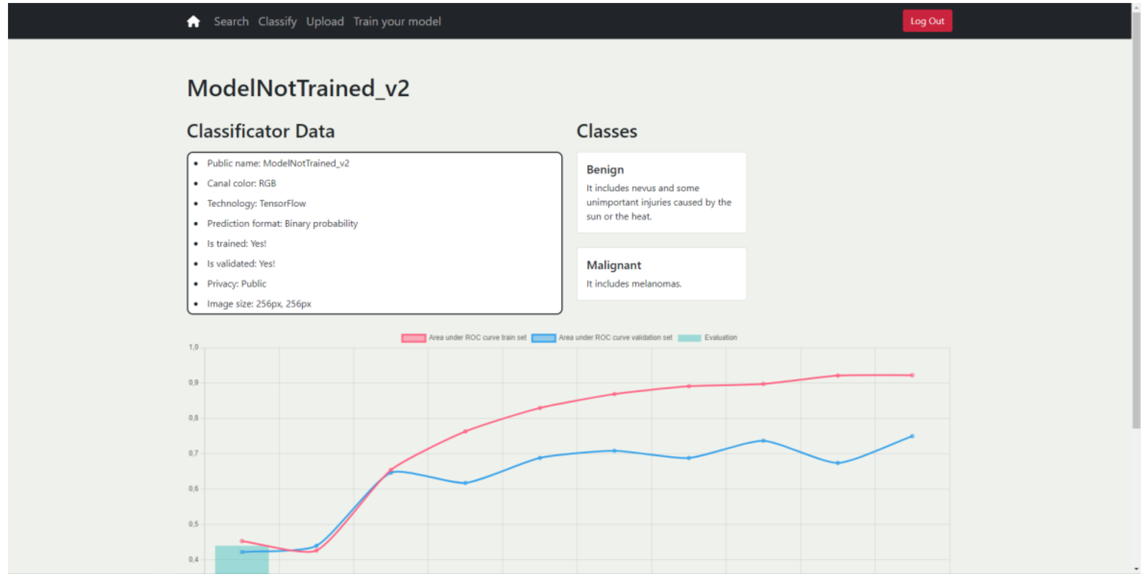


Figura B.3 Captura ventana información clasificador.

3. Publicar un clasificador

1. Comienza iniciando sesión con un usuario (Apartado 1).
2. Con la sesión iniciada pulsa en el botón "Upload" de la barra superior de navegación.
3. Aparecerá la pantalla para publicar un nuevo modelo. Rellene todos los datos del formulario sobre tu clasificador y pulsa el botón "Next".
    a. Si tu modelo ya está entrenado, aparecerá una pantalla para que introduzcas los datos de las clases que el clasificador diferencia.
    b. Si tu modelo no ha sido aún entrenado, el modelo se publicará automáticamente.

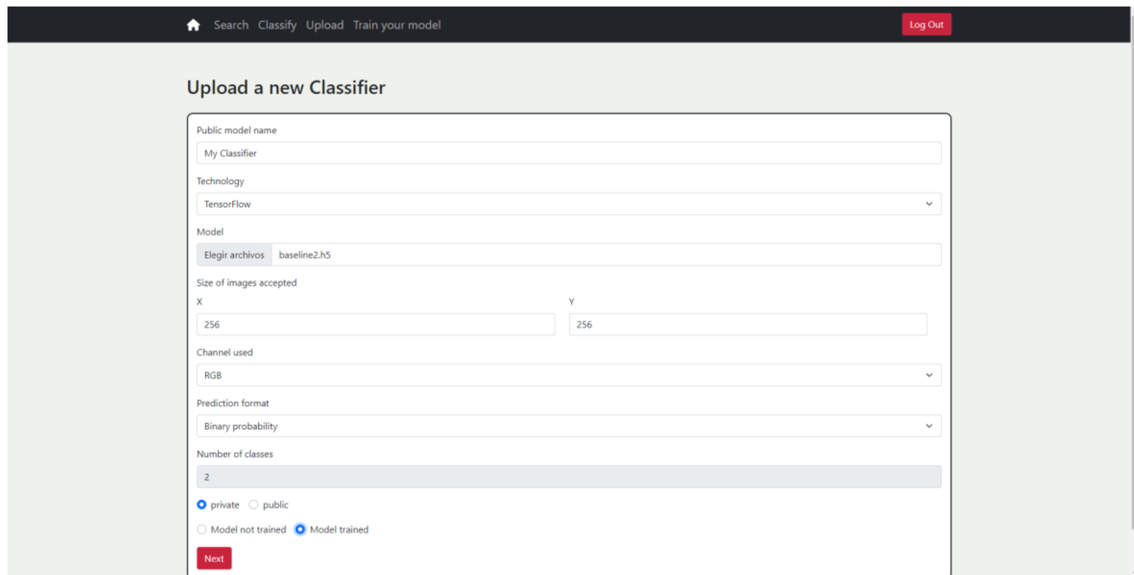


Figura B.4 Captura ventana Upload.

4. Podrá ver cómo se ha publicado su modelo correctamente siguiendo el Apartado 2 de esta guía. Deberás esperar a que el clasificador sea validado por un administrador para utilizarlo.

4. Clasificar una imagen

1. Pulsa el botón "Classify" de la barra de navegación superior.
2. Se mostrará la pantalla de clasificación. Selecciona la imagen que quieras clasificar pulsando en "Seleccionar archivo".

3. La imagen seleccionada se previsualizará y aparecerán los clasificadores validados a los que tienes acceso. Si has iniciado sesión podrás utilizar tus modelos publicados y validados. Selecciona el clasificador que quieras utilizar.
4. Pulsa botón "Classify" que ha aparecido.
5. Se mostrará el resultado de diagnóstico obtenido por el clasificador para la imagen seleccionada.

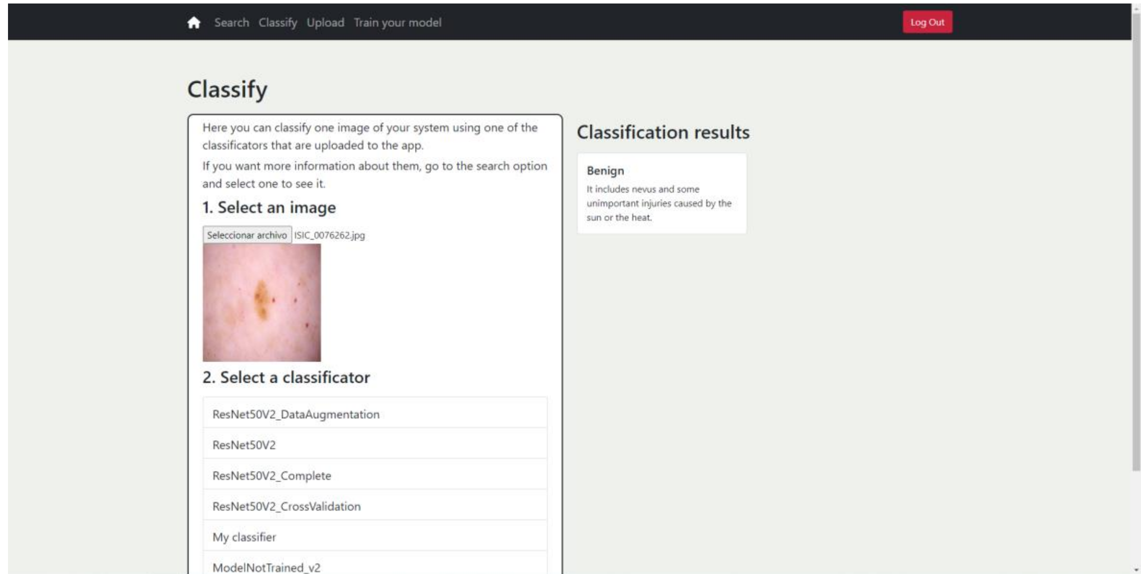


Figura B.6 Captura ventana Classify.

5. Entrenar clasificador

1. Pulsa el botón "Train your model" de la barra de navegación superior.
2. Se mostrará la pantalla de entrenamiento de un clasificador.
3. Elija el clasificador que desee entrenar de la lista.
4. Aparecerá la lista de *datasets* compatibles con el clasificador. Ten en cuenta que algunos de los *datasets* pueden no estar disponibles para todos los clasificadores. Pulsa el botón "Train" del *dataset* deseado.

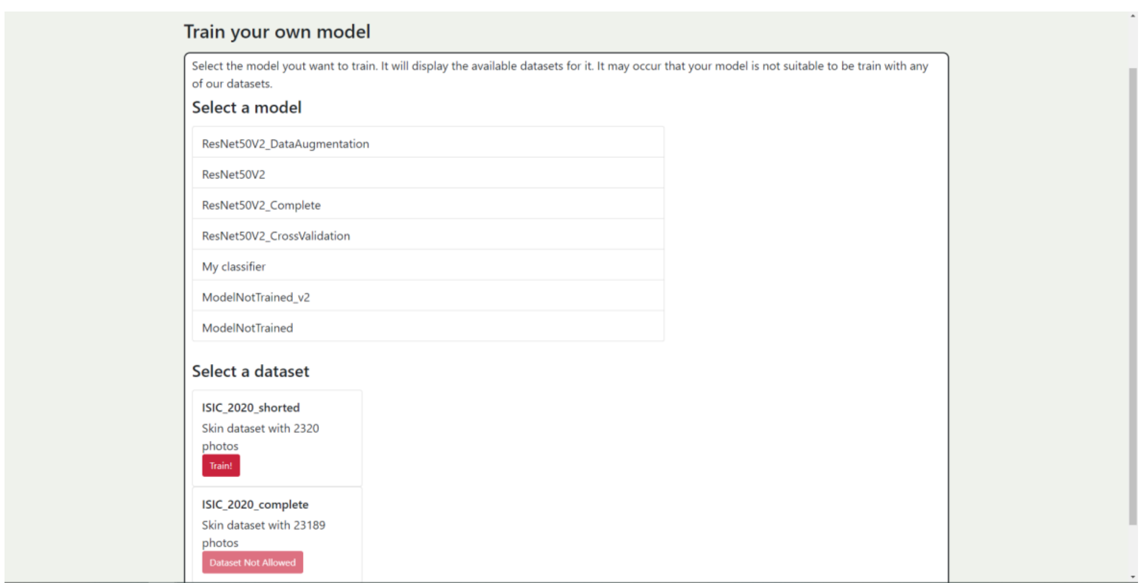


Figura B.7 Captura ventana Train your model.

5. Pasado el tiempo de entrenamiento se creará el nuevo clasificador directamente validado para su uso.

E.T.S. DE INGENIERÍA INFORMÁTICA

E.T.S de Ingeniería Informática
Bulevar Louis Pasteur, 35
Campus de Teatinos
29071 Málaga