



**Universidad
Rey Juan Carlos**

INGENIERÍA EN TECNOLOGÍAS DE LA
TELECOMUNICACIÓN

Curso Académico 2020/2021

Trabajo Fin de Grado

**AUTENTICACIÓN Y GESTIÓN DE
MODELOS DE MACHINE LEARNING PARA
ENTORNOS CORPORATIVOS**

Autor : Jaime Solsona Álvarez

Tutor : Dr. José Felipe Ortega Soto

Trabajo Fin de Grado

Autenticación y Gestión de Modelos de Machine Learning para Entornos
Corporativos

Autor : Jaime Solsona Álvarez

Tutor : Dr. José Felipe Ortega Soto

La defensa del presente Proyecto Fin de Grado se realizó el día de
de 2021, siendo calificada por el siguiente tribunal:

Presidente:

Secretario:

Vocal:

y habiendo obtenido la siguiente calificación:

Calificación:

Fuenlabrada, a de de 2021

|

*Dedicado a mis amigos
y a mi familia*

Agradecimientos

En primer lugar, me gustaría agradecer a mi tutor, Felipe Ortega, por la toda la gran ayuda que me ha prestado para hacer este Trabajo de Fin de Grado. A pesar de todos los compromisos e imprevistos que ha tenido durante los últimos meses, siempre ha podido sacar un hueco para contestar a mis correos o mantener reuniones para comprobar el avance del proyecto, y por ello le estoy muy agradecido.

También me gustaría agradecer a mi familia, a mis amigos, de dentro y fuera de la universidad, y especialmente a mis compañeros del doble grado, por acompañarme en estos 6 años de carrera. Sin ellos, el camino habría sido muchísimo más duro.

Resumen

Este proyecto consiste en el desarrollo e implementación de un sistema de control de integridad y autenticidad en modelos de Machine Learning, y además, un servidor HTTP con una API REST que centralice la gestión de modelos de Machine Learning dentro de un entorno empresarial, permitiendo así que distintos grupos de trabajo puedan compartir sus modelos de Machine Learning.

En este proyecto nos hemos centrado en el paquete de Machine Learning *Scikit-learn*, para el lenguaje de programación Python. Para el control de integridad y autenticidad añadimos unas funciones de firma y verificación a los modelos de *Scikit-learn*. La función de firma calcula un *hash* sobre el modelo serializado y lo encripta con una clave, consiguiendo así una firma del modelo. La función de verificación calcula el *hash* del modelo, desencripta la firma y comprueba que ambos coinciden.

Por otra parte, hemos utilizado *Flask* para desarrollar el servidor HTTP. Dicho servidor tiene una API REST que permite descargar, subir, modificar y eliminar modelos de *Scikit-learn*, así como ver un listado de los mismos. El servidor puede ser accedido tanto a través de un navegador web como desde Python, mediante una interfaz Python que hemos desarrollado. Para acceder a los modelos del servidor es necesario autenticarse previamente mediante *OpenID*.

Summary

The main goal of this project is to develop and implement an integrity and authenticity control system for Machine Learning models, and, additionally, a HTTP server with an API REST that centralizes the management of said models in an enterprise environment, letting work groups share their Machine Learning models with each others.

In this project we have focused on *Scikit-learn*, a Machine Learning package for Python. For the integrity and authenticity control we have added some sign and verify functions to *Scikit-learn* models. The sign function generates a hash of the serialized model, and then encrypts it with a key, getting a signature of the model. The verify function generates the model hash again, decrypts the signature and checks that both are identical.

On the other hand, we have used *Flask* to develop the HTTP server. The server has an API REST that allows to download, upload, modify and delete *Scikit-learn* models. It allows to get a list of all models too. The server can be accessed through either a browser or an Python interface we have developed. In order to access any model on the server, the user has to authenticate with *OpenID*.

Índice general

1	Introducción	1
1.1	Objetivo general	3
1.2	Objetivos específicos	3
1.2.1	Control de verificación de integridad y autenticidad sobre Scikit-learn .	3
1.2.2	API REST	3
1.3	Planificación temporal	4
1.4	Estructura de la memoria	4
1.5	Enlaces de interés	5
2	Estado del arte	7
2.1	Python	7
2.2	Miniconda	8
2.3	Visual Studio Code	9
2.4	GitHub	10
2.5	Scikit-learn	10
2.6	PyCryptodome	12
2.7	orjson	12
2.8	Flask	13
2.9	OpenAPI	14
2.10	Autenticación de servidor	14
2.10.1	Kerberos	14
2.10.2	OAuth2/OpenID	17

3 Diseño e implementación	21
3.1 Arquitectura general	21
3.2 Arquitectura de la API REST	22
3.3 Arquitectura de la autenticación OpenID	28
3.4 Arquitectura de la clase <code>RemoteServer</code>	29
3.5 Arquitectura del módulo <i>ml-fingerprint</i>	31
4 Casos de uso	37
4.1 Autenticación con el servidor y obtención de la <i>API Key</i>	37
4.2 Integración con modelos de <i>Scikit-learn</i>	39
4.2.1 Modelo de clasificación	39
4.2.2 Modelo de clustering	47
4.2.3 Modelo de regresión	54
5 Conclusiones	61
5.1 Consecución de objetivos	61
5.2 Aplicación de lo aprendido	61
5.3 Lecciones aprendidas	62
5.4 Trabajos futuros	63

Índice de figuras

2.1	Interfaz de <i>Visual Studio Code</i>	9
2.2	Comparativa entre clasificadores de <i>Scikit-learn</i>	11
2.3	Ejemplo de estructura de clases en <i>Scikit-learn</i>	11
2.4	Comparativa entre serializadores de JSON	13
2.5	Esquema de diálogo durante un proceso de autenticación con <i>Kerberos</i>	16
2.6	Esquema de diálogo de autenticación mediante OpenID.	18
3.1	Estructura general del proyecto.	22
3.2	Esquema de diálogo cliente-servidor a través de la web.	23
3.3	Página HTML de lista de modelos.	25
3.4	Esquema de diálogo cliente-servidor a través de Python.	25
3.5	Ejemplo de objeto JSON de un modelo.	28
3.6	Diagrama general de conexión con el servidor mediante la web.	29
3.7	Arquitectura de la clase <i>RemoteServer</i> y sus conexiones con la API REST del servidor.	30
3.8	Diagrama de las fases de utilización del paquete <i>ml-fingerprint</i>	32
4.1	Página principal del servidor.	38
4.2	Página de inicio de sesión en <i>Google</i>	38
4.3	Página de perfil de usuario.	39
4.4	Ejemplo de limitación de outliers a una distancia de 3 veces el IQR.	42
4.5	Matriz de confusión para el modelo de clasificación.	44
4.6	Importancia de las variables en el modelo de clasificación.	45
4.7	Ejemplo de integración del modelo de clasificación en el servicio.	46

4.8 Ejemplo de un <i>Pokémon</i> y sus estadísticas dentro del juego.	49
4.9 Histograma de la suma total de estadísticas para cada grupo.	50
4.10 Histograma de la estadística de Defensa para los grupos 3 y 4.	51
4.11 Ejemplo de integración del modelo de clustering en el servicio.	52
4.12 Ejemplo de salida al ejecutar la función <code>get_list_models</code> de <code>RemoteServer</code> . .	53
4.13 Representación de distintas variables del <i>dataset</i> frente al precio de la vivienda. .	55
4.14 Matriz de correlación entre las distintas variables.	56
4.15 Ejemplo de integración del modelo de regresión en el servicio.	58
4.16 Captura de pantalla de la web mostrando las tres versiones del modelo.	59

Índice de tablas

4.1	Diccionario de datos para el modelo de clasificación.	40
4.2	Conversión de datos categóricos a numéricos mediante <i>One Hot Encoding</i> . . .	41
4.3	Diccionario de datos para el modelo de clustering.	49
4.4	Diccionario de datos para el modelo de regresión.	54

Capítulo 1

Introducción

El Machine Learning (en castellano, *Aprendizaje Automático*) es una subcategoría de inteligencia artificial que genera modelos matemáticos en base a una serie de datos conocidos, de manera que la máquina “aprende” de éstos, y usa este modelo para predecir resultados a partir de nuevos datos [5]. Como el aprendizaje es automático, se puede escalar para tamaños de datos muy grandes, lo que ha provocado que el Machine Learning sea cada vez más importante y tenga multitud de usos en todos los sectores [8, 9].

Muchos de los entornos empresariales donde trabajan con modelos de Machine Learning no están optimizados para ello. En numerosas ocasiones se da el caso de que distintos equipos de una misma empresa trabajan en ámbitos similares, y por tanto, muy probablemente acaben desarrollando modelos muy parecidos. Ambos podrían beneficiarse mutuamente de los modelos entrenados por el otro equipo, pero esto no sucede por no tener un sistema centralizado al que poder acceder desde cualquier punto de la empresa [6].

En este proyecto planteamos una posible solución a este problema, consistente en una API REST que permita a todos los trabajadores de una empresa subir, ver, comparar y descargar modelos de Machine Learning, para que así futuros grupos de trabajo puedan aprovechar el esfuerzo ya realizado por otros compañeros.

No obstante, surge otro problema. De forma general, las bibliotecas de Machine Learning no disponen de ningún control de integridad sobre sus modelos, lo que significa que no se puede garantizar que el modelo no haya sido manipulado por un atacante externo [1]. Esto significa que si las empresas quieren verificar la integridad de sus modelos, necesitan acudir a soluciones externas. En la mayoría de casos, esto acaba no sucediendo, dada la complejidad y coste adicional

que les supondría.

En consecuencia, este problema latente se va haciendo más grande a medida que el Machine Learning va ganando importancia en el mundo. Vamos hacia un futuro donde modelos de Machine Learning controlarán aspectos tan importantes como la bolsa, precios de pisos o de billetes de avión, detección de enfermedades, reconocimiento facial o conducción autónoma. Permitir a atacantes modificar estos modelos para su propio beneficio sin que nadie se dé cuenta podría tener consecuencias desastrosas para la sociedad.

En el caso habitual, las empresas guardan cada modelo en el ordenador del grupo de trabajo que lo ha desarrollado. Normalmente, este ordenador estará conectado a Internet, siendo así un posible blanco para un ataque a distancia, por lo que cualquier atacante podría acceder a esos modelos si consigue acceder al ordenador de manera remota. En el caso propuesto de centralizar todos los modelos en un servidor, el problema se acentúa aún más, ya que con tener acceso al servidor podrían modificar todos los modelos de la empresa, no sólo los de un grupo de trabajo.

Así pues, se plantea la posibilidad de añadir un mecanismo de verificación que permita comprobar la integridad y autenticidad del modelo de manera sencilla, rápida y eficiente, con el objetivo adicional de que esta implementación sea lo más transparente posible, para ofrecer a las empresas la posibilidad de verificar la integridad y autenticidad de sus modelos sin apenas modificar su flujo de trabajo.

Evidentemente, es imposible modificar todas las bibliotecas de Machine Learning existentes, ya que existen miles, abarcando multitud de lenguajes de programación. Es necesario centrarse un lenguaje de programación y una biblioteca de Machine Learning en concreto para llevar a cabo una prueba práctica en este proyecto. Dado que el objetivo es añadir un control de verificación de integridad y autenticidad a una biblioteca de Machine Learning, lo apropiado sería trabajar con una de las más utilizadas, para que esta solución sea aplicable al mayor número posible de modelos.

El lenguaje de programación dominante en el mundo del Machine Learning es Python, y dentro de Python, la biblioteca de referencia para implementar Machine Learning es *Scikit-learn*¹. Así pues, usaremos Python para el desarrollo del proyecto, y *Scikit-learn* será la biblioteca a la que añadimos el control de verificación de integridad y autenticidad que resuelva este problema.

¹<https://scikit-learn.org/>

1.1 Objetivo general

Este trabajo fin de grado tiene dos objetivos principales:

- Por una parte, añadir las herramientas necesarias a la biblioteca de Python *Scikit-learn* que permitan **verificar la integridad y autenticidad de los modelos** de Machine Learning. Es importante implementarlas de la manera más transparente posible, de manera que se pueda aplicar a modelos ya existentes y éstos sigan funcionando igual que antes, consiguiendo así que sean compatibles con todos los entornos de trabajo ya existentes.
- Por otra, **crear una API REST** en un servidor web que permita trabajar con modelos de *Scikit-learn*, ofreciendo así una alternativa para el entorno de trabajo de las empresas que centralice los modelos y optimice el trabajo de todos los desarrolladores, y hacerlo de manera limpia y sencilla, para demostrar que las modificaciones necesarias en el flujo de trabajo para implementar la capa de verificación de integridad y autenticidad son mínimas.

1.2 Objetivos específicos

1.2.1 Control de verificación de integridad y autenticidad sobre Scikit-learn

1. Ampliar el código de las clases base de *Scikit-learn*, añadiendo un **método de firma** y otro de **verificación**, para dotar a los modelos de **control de integridad** básico.
2. Procurar que la adición de estos métodos de firma y verificación sea **transparente** para el resto de funcionalidades de los modelos de *Scikit-learn* y permita el acceso a toda la API estándar de entrenamiento y predicción de datos.

1.2.2 API REST

1. Crear un servidor web y una base de datos para guardar los modelos de *Scikit-learn* con los que vamos a trabajar.
2. Desarrollar e implementar una API REST que permita, entre otras cosas, acceder, crear, modificar y borrar modelos de *Scikit-learn* del servidor.

3. Asegurar la seguridad de acceso al servidor mediante un control de acceso.
4. Crear un flujo de trabajo de Machine Learning que haga uso de esta API REST para coger los modelos.
5. Hacer las modificaciones pertinentes para implementar en el flujo de trabajo la verificación de integridad y autenticidad de los modelos a los que se les haya implementado las herramientas para verificar su integridad y autenticidad.

1.3 Planificación temporal

Entre el desarrollo del proyecto, acopio de información, reuniones de control con el tutor y escritura de la memoria, este TFG ha llevado aproximadamente unas 120 horas, repartidas entre octubre de 2020 y marzo de 2021.

1.4 Estructura de la memoria

El resto de capítulos de la memoria se organizan de la siguiente forma:

- En el capítulo 2 se describen las principales tecnologías involucradas en el desarrollo de este proyecto.
- En el capítulo 3 se explica en detalle la arquitectura e implementación de las soluciones tecnológicas propuestas para añadir control de integridad a los modelos de Machine Learning en *Scikit-learn*.
- Después, en el capítulo 4 se ilustran las funcionalidades de la solución implementada en el proyecto mediante distintos casos de uso.
- Por último, en el capítulo 5 se resumen las conclusiones finales del proyecto, así como posibles vías de trabajo futuro.

1.5 Enlaces de interés

El servidor web con la API REST que se menciona a lo largo de la memoria está desplegado y es accesible desde la siguiente URL: <https://dslab01.etsit.urjc.es/mlfingerprint/>

En la página web, además de las funcionalidades descritas en el proyecto, se encuentran disponibles enlaces tanto al repositorio de *GitHub* del proyecto como a la documentación del mismo.

Capítulo 2

Estado del arte

En este capítulo se describen las diferentes tecnologías que han sido usadas en el desarrollo del proyecto, así como las razones por las que fueron escogidas.

2.1 Python

Python es un lenguaje de programación interpretado y dinámico, de código abierto, que soporta tanto el paradigma de la programación orientada a objetos (OOP) como el de la programación funcional.

Se ha elegido este lenguaje de programación por múltiples motivos. En primer lugar, es uno de los lenguajes de programación más usados en todo el mundo. Python es el segundo lenguaje de programación en número de proyectos alojados en *GitHub*, y sigue una tendencia ascendente ¹. Por otra parte, es el tercer lenguaje más querido por los programadores, según la encuesta anual realizada por Stack Overflow en 2020 ².

No obstante, la principal razón por la que hemos escogido Python es porque es el más utilizado en el mundo del Machine Learning. Las facilidades que ofrece para trabajar con grandes cantidades de datos (con paquetes como *NumPy* o *Pandas*) y después visualizarlos (con *Matplotlib*), sumado a la extensísima colección de librerías y paquetes que hay disponibles para hacer modelos de Machine Learning para todo tipo de datos (numéricos, texto, imágenes, sonidos, etc.) han convertido a Python en la opción por defecto para la mayoría de analistas de datos.

¹<https://www.benfrederickson.com/ranking-programming-languages-by-github-users>

²<https://stackoverflow.blog/2020/05/27/2020-stack-overflow-developer-survey-results>

2.2 Miniconda

Para el desarrollo del proyecto ha sido necesario un entorno virtual para poder instalar las bibliotecas necesarias para el funcionamiento del proyecto, y a su vez tener una lista de dichas dependencias para poder incluirlas en la especificación del paquete. Para ello, hay dos alternativas claras: *Virtualenv* y *Anaconda*.

*Virtualenv*³ es la herramienta de entornos virtuales que viene incluida con Python. Su uso está muy extendido, y permite de una manera fácil y ligera instalar los paquetes necesarios para el desarrollo de un proyecto sin afectar otros proyectos.

*Anaconda*⁴ es otra herramienta de entornos virtuales, gratuita, de código abierto, que no es exclusiva para Python, y que además incluye su propio gestor de paquetes (llamado *conda*). Está muy estandarizado dentro del mundo de la ciencia de datos, e incluye por defecto una gran cantidad de paquetes relacionados con dicho sector, lo que permite crear un entorno de desarrollo de manera veloz. No obstante, la gran cantidad de paquetes incluidos puede suponer una desventaja si el proyecto no requiere tantas librerías, ya que se convierte en un entorno muy pesado de manera innecesaria.

Dado que nuestro proyecto va a trabajar con Machine Learning, decidimos escoger *Anaconda*, ya que está más orientado para Machine Learning. No obstante, nuestro proyecto es relativamente ligero, y no requiere muchas librerías, por lo que usar *Anaconda* resultaría en un entorno demasiado pesado para nuestras necesidades.

Así pues, decidimos utilizar *Miniconda*⁵, una versión de *Anaconda* que tan sólo incluye el mínimo número de paquetes imprescindibles para un entorno básico de Python. Esto significa que tenemos que instalar todos los paquetes adicionales manualmente, pero a cambio obtenemos un entorno de trabajo ligero, que sólo contiene lo necesario para nuestro proyecto. Además, *Miniconda* utiliza el mismo gestor de paquetes de *Anaconda*, así que el proceso de instalación de paquetes resulta muy sencillo.

³<https://virtualenv.pypa.io/en/stable/>

⁴<https://www.anaconda.com/>

⁵<https://docs.conda.io/en/latest/index.html>

2.3 Visual Studio Code

*Visual Studio Code*⁶ es un editor de código desarrollado por *Microsoft*⁷. Es gratuito, multiplataforma, de código abierto e incluye funcionalidades como depuración, un terminal, integración de *git*, resultado de sintaxis o autocompletado, entre otras. En la figura 2.1 se muestra su interfaz principal.

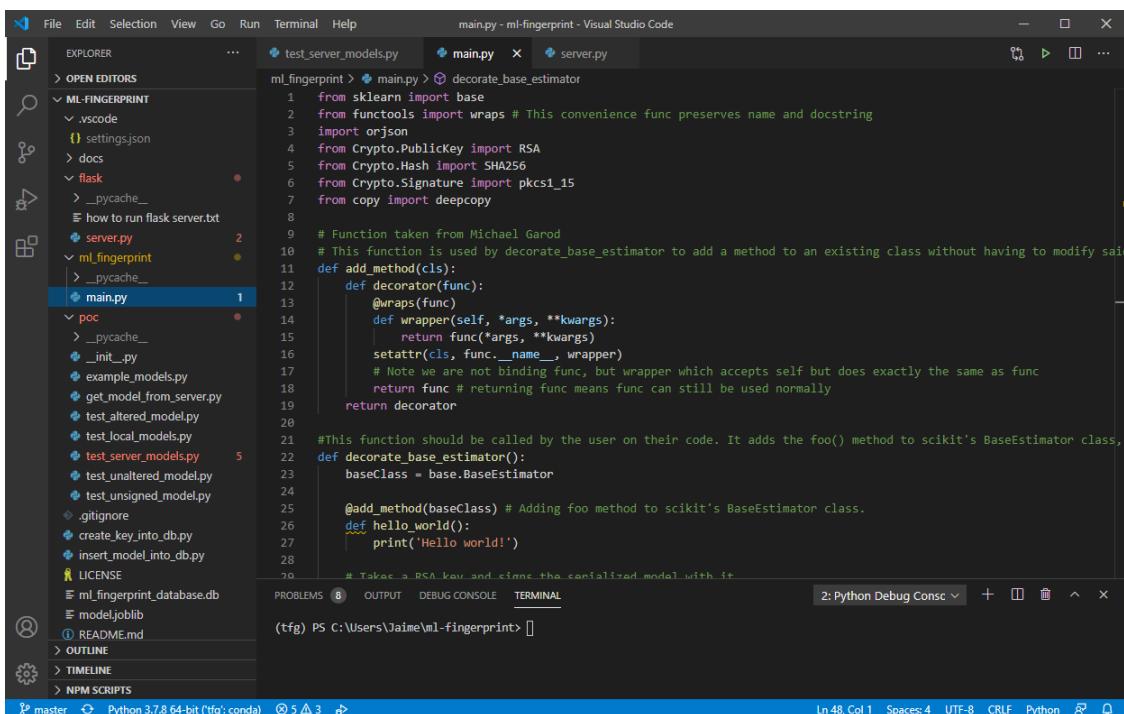


Figura 2.1: Interfaz de Visual Studio Code

Para el desarrollo de código existen infinitud de editores, como por ejemplo *Atom* o *Sublime Text*, y varios *IDE* (Entorno de desarrollo integrado) específicos para Python, como *PyCharm*. A título personal, he optado por Visual Studio Code porque es el editor que suelo utilizar para todos mis proyectos, con el que más cómodo me encuentro y el que mejor manejo. Es el editor más utilizado a nivel mundial, y al ser de código abierto, dispone de multitud de *plugins* de terceros para añadir más funcionalidades aún. Adicionalmente, a pesar de ser sólo un editor de código, al tener el terminal integrado, se pueden ejecutar y depurar los programas como si de un *IDE* se tratase.

⁶<https://code.visualstudio.com/>

⁷<https://www.microsoft.com/es-es>

2.4 GitHub

Git es un software de control de versiones que permite mantener un historial de modificaciones y facilita el trabajo entre varias personas en un mismo proyecto [2].

Por otra parte, *GitHub* es una plataforma gratuita para subir a la red proyectos que usen *Git*. La combinación de ambos es la manera más extendida para llevar un control del proyecto y poder acceder a él desde cualquier parte. Para este proyecto no se ha aprovechado todo su potencial, ya que ha sido desarrollado por sólo una persona y *Git* tiene muchas funcionalidades de cara al trabajo en equipo, pero ha sido igualmente útil para llevar un control de modificaciones y que el tutor, así como el resto de personas interesadas, tengan acceso al código a través de *GitHub*.

Además de *GitHub* había otra gran alternativa: *GitLab*. *GitLab* es muy similar a *GitHub*, y además la Escuela Técnica Superior de Ingeniería de Telecomunicación (ETSIT) de la Universidad Rey Juan Carlos tiene habilitado un servidor Ultimate para uso de los alumnos. No obstante, hemos decidido utilizar *GitHub* porque estamos más familiarizados con su interfaz, está más extendido a nivel mundial y además ofrecen a los estudiantes de forma gratuita su servicio premium *GitHub Pro*, que habilita funcionalidades adicionales.

2.5 Scikit-learn

Scikit-learn es una biblioteca de Machine Learning para Python. Nació en 2010, es software libre (licencia BSD) e incluye muchos algoritmos de clasificación, regresión y análisis de grupos. Está basado en los algoritmos matemáticos de *SciPy*, que a su vez se apoya en las matrices de *NumPy*, dos bibliotecas que son el estándar para trabajos científicos y matemáticos en Python.

Hemos elegido *Scikit-learn* porque es la biblioteca de Machine Learning más utilizada en Python. Es muy completa, dispone de algoritmos para resolver todo tipo de problemas, y al estar tan extendida entre la comunidad científica, existe una amplia documentación. Además, es de código abierto, requisito fundamental para poder añadir las herramientas de verificación de integridad y autenticidad en las que consiste el proyecto. En la figura 2.2 se pueden ver algunos de los tipos de clasificadores que incluye *Scikit-learn*.

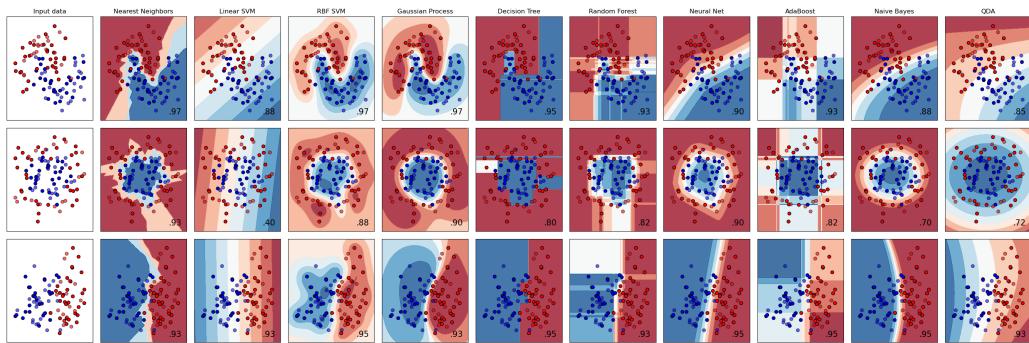


Figura 2.2: Comparativa entre clasificadores de Scikit-learn

Scikit-learn está subdividido en módulos, y cada módulo contiene las clases de los estimadores de cada tipo. Por ejemplo, en `sklearn.linear_model` están todos los estimadores lineales, tanto de regresión como de clasificación. Existe un módulo `sklearn.base` que es donde están las clases base de las que heredan el resto de módulos. Concretamente, dentro de este módulo está la clase `BaseEstimator`, a partir de la cual, clases intermedias mediante, acaban heredando todos los estimadores de *Scikit-learn*, independientemente del tipo que sean. En la Figura 2.3 se puede ver sobre 4 estimadores de ejemplo cómo es la estructura interna de clases. Esto es muy importante de cara al proyecto, ya que se pueden modificar todos los modelos de *Scikit-learn* con tan sólo hacer una simple modificación a la clase `BaseEstimator`.

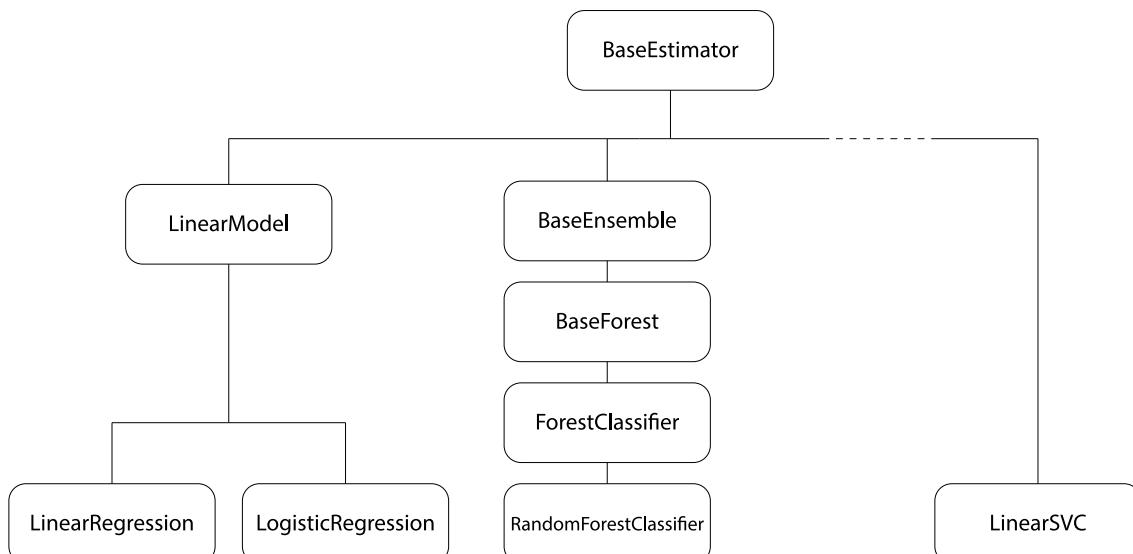


Figura 2.3: Ejemplo de estructura de clases en Scikit-learn

2.6 PyCryptodome

*PyCryptodome*⁸ es una biblioteca de criptografía para Python. Está construida sobre la base de PyCrypto, otra biblioteca de criptografía que era muy utilizada, pero que dejó de recibir actualizaciones en 2014. Contiene multitud de primitivas criptográficas de bajo nivel, entre ellas *hashes* criptográficos, generación de claves asimétricas y cífrados asimétricos, las tres funciones indispensables para este proyecto.

Hemos elegido esta biblioteca en lugar de otras como *cryptography*⁹ o *PyNaCl*¹⁰ por la buena reputación que tenía PyCrypto y por la organización clara y concisa de sus módulos. En términos de eficiencia, todas las librerías mencionadas están a la par, así que la decisión ha sido bastante personal.

2.7 orjson

*orjson*¹¹ es una librería para Python que permite serializar y deserializar objetos en JSON. Se necesitaba un serializador para poder calcular el *hash* del modelo. Las dos opciones principales eran serializar en binario con *pickle*¹² o con JSON. Dado que la eficiencia es un factor importante, se decidió utilizar JSON, ya que es más rápido¹³ que *pickle*.

Dentro de JSON hay varias alternativas, como el serializador incluido en Python *json*¹⁴, o alternativas más rápidas como *simplejson*¹⁵ o *ujson*¹⁶. Hemos tomado la decisión de usar *orjson* no sólo por ser el más rápido de todos¹⁷, tal y como se puede observar en la figura 2.4, sino también por ser el único que admitía arrays de *numpy* de manera nativa, lo cual es importante ya que *Scikit-learn* guarda los coeficientes de sus modelos en arrays de *numpy*.

⁸<https://pycryptodome.readthedocs.io/>

⁹<https://cryptography.io/>

¹⁰<https://pynacl.readthedocs.io/>

¹¹<https://github.com/ijl/orjson>

¹²<https://docs.python.org/3/library/pickle.html>

¹³<https://konstantin.blog/2010/pickle-vs-json-which-is-faster/>

¹⁴<https://docs.python.org/3/library/json.html>

¹⁵<https://simplejson.readthedocs.io/>

¹⁶<https://github.com/ultrajson/ultrajson>

¹⁷<https://github.com/ijl/orjson#performance>

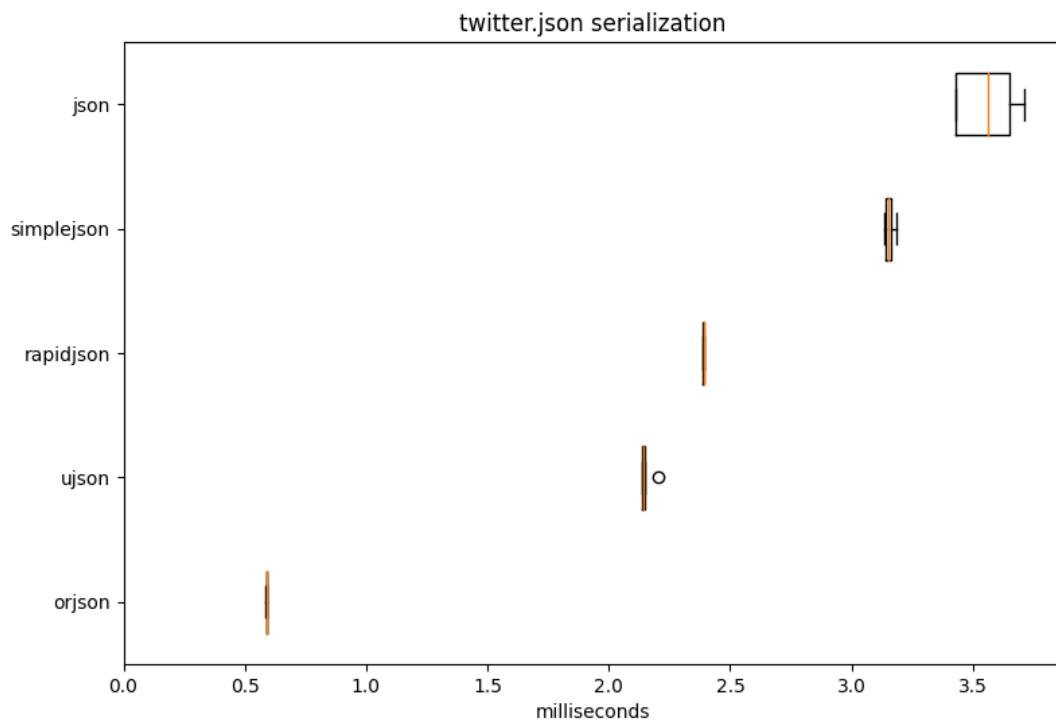


Figura 2.4: Comparativa entre serializadores de JSON

2.8 Flask

*Flask*¹⁸ es un *micro framework* para la elaboración de aplicaciones web en Python [4]. Su uso está bastante extendido, llegando a ser usado en páginas tan conocidas como Pinterest o Linkedin.

Hemos escogido Flask para el desarrollo del servidor web por su simplicidad, ya que tan sólo incluye las librerías mínimas imprescindibles para su funcionamiento, a diferencia de otras alternativas como *Django*. En *Django* vienen incluidas más herramientas adicionales que pueden facilitar algunas tareas, pero que en este caso son innecesarias dado que la API REST a desarrollar es bastante sencilla.

Otro *micro framework* bastante extendido, quizás incluso más que *Flask*, es *Express*. No obstante, hemos decidido usar *Flask* para poder basar el proyecto entero en Python, ya que *Express* está basado en JavaScript.

¹⁸<https://flask.palletsprojects.com/>

2.9 OpenAPI

La especificación *OpenAPI*¹⁹ consiste en una serie de reglas creadas para estandarizar el desarrollo de API REST [7]. Utilizando herramientas como el *Swagger Editor*²⁰ se puede escribir un archivo .yml que siga la especificación *OpenAPI* donde se describa de manera universal y unívoca una API REST. Posteriormente se puede utilizar *Swagger Codegen*²¹ con ese archivo para generar automáticamente una estructura de código que siga la API REST descrita, para cualquier *framework* web.

Hemos implementado la API REST de manera tradicional, escribiendo todo el código de *Flask* a mano, ya que utilizar el *Swagger Codegen* mientras la API aún está en desarrollo resulta poco eficiente. Sin embargo, al describir la API en un archivo .yml que siga la especificación *OpenAPI*, damos la posibilidad a cualquier desarrollador de generar la base del código del servidor web que hemos desarrollado para cualquier lenguaje de programación y *framework* (por ejemplo, *Express*, en *JavaScript*).

2.10 Autenticación de servidor

2.10.1 Kerberos

La aplicación web encargada de servir modelos de *Scikit-learn* necesita un sistema de autenticación de usuarios para evitar que cualquiera pueda ver, modificar o borrar los modelos del servidor. Para ello, existe un gran abanico de opciones.

La primera opción que viene a la mente es un sistema de registro propio con usuario y contraseña. Si bien esta es la opción más sencilla, también es la más insegura, ya que implicaría guardar las contraseñas de los usuarios en una base de datos. Aunque hoy en día existen maneras de guardar contraseñas de manera segura (aplicando un *hash*, por ejemplo), hay sistemas de autenticación alternativos más seguros y modernos.

Una de esas alternativas es *Kerberos*. *Kerberos* es un protocolo de autenticación creado por el MIT que usa un “tercero de confianza”, es decir, un agente externo al cliente y servidor que sirva para autenticar la identidad del cliente frente al servidor. Este “tercero de confianza” es el

¹⁹<https://swagger.io/specification/>

²⁰<https://swagger.io/tools/swagger-editor/>

²¹<https://swagger.io/tools/swagger-codegen/>

KDC (Key Distribution Center, Centro de Distribución de Llaves, en castellano), que consta de dos partes independientes: el AS (Authentication Server, Servidor de Autenticación) y el TGS (Ticket Granting Server, Servidor Emisor de Tickets).

Una de sus grandes fortalezas es que para autenticar la contraseña del usuario no viaja en ningún momento por la red, ni siquiera encriptada. Esto se consigue gracias al modelo de encriptación simétrica, que permite encriptar y desencriptar una información con la misma clave. Como veremos a continuación, si ambos lados de una comunicación conocen de antemano la misma clave, uno de los lados puede comprobar la autenticidad del otro lado enviando una información encriptada con dicha clave; si el otro lado consigue desencriptarla, significa que la clave es la misma, y que por tanto es quien dice ser.

Su funcionamiento se puede ver gráficamente en la figura 2.5. A continuación, se explicará en detalle:

Primero, el cliente envía el nombre del usuario en texto plano al AS. Éste lo recibe, comprueba que existe en su base de datos, y envía de vuelta dos mensajes: un *TGT* (*Ticket-Granting Ticket, Ticket Emisor de Tickets*), encriptado primero con la clave privada del TGS (que conocen tanto el TGS como el AS), y después con la clave privada del usuario (generada a partir de su contraseña), y la clave de sesión del TGS, encriptada con la clave privada del usuario. El cliente recibe estos dos mensajes, genera su propia clave privada con la contraseña del usuario (si la contraseña es correcta, la clave privada será la misma que ha usado el AS), y desencripta ambos mensajes con ella. Nótese que el *TGT* sólo será parcialmente desencriptado, ya que se encriptó adicionalmente con la clave privada del TGS, que el cliente desconoce.

En siguiente lugar, el cliente envía dos mensajes al TGS: uno compuesto por el *TGT* parcialmente desencriptado y el identificador del servidor al que quiere acceder, y otro compuesto por el identificador del cliente y la marca de tiempo actual, cifrado con la clave de sesión del TGS que ha recibido antes del AS. El TGS recibe estos dos mensajes, termina de desencriptar el *TGT* con su clave privada y comprueba su validez, desencripta el segundo mensaje con su clave de sesión, y en caso de estar todo en orden, devuelve al cliente dos mensajes: por una parte, un *Service Ticket*, que contiene información del cliente, un periodo de validez y una clave cliente/servidor, encriptado con la clave privada del servidor, por otra parte, la misma clave cliente/servidor, encriptada con la clave de sesión del TGS. El cliente recibe ambos mensajes, y desencripta el segundo, obteniendo así la clave cliente/servidor que usará para comunicarse con el servidor. El

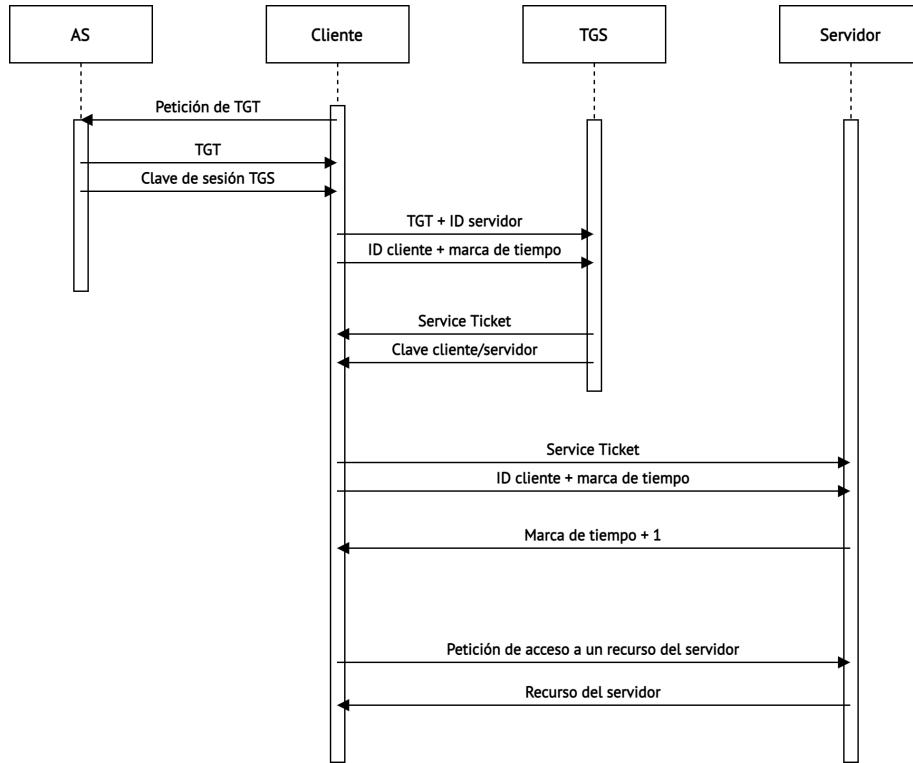


Figura 2.5: Esquema de diálogo durante un proceso de autenticación con Kerberos.

primer mensaje no lo puede descifrar, ya que no conoce la clave privada del servidor.

Por último, el cliente envía al servidor el *Service Ticket* tal cual le llegó del TGS, y otro adicional consistente en su identificador de cliente y una marca de tiempo, encriptado con la clave cliente/servidor que acaba de obtener del TGS. El servidor recibe ambos mensajes, desencripta el *Service Ticket* con su propia clave privada, y obtiene la clave cliente/servidor que iba incluida dentro. Con esa clave desencripta el segundo mensaje y obtiene tanto el identificador del cliente como la marca de tiempo. El servidor sumará uno a dicha marca de tiempo y la enviará de vuelta al cliente, encriptandola con la clave cliente/servidor que ahora conoce. El cliente lo recibirá, lo desencriptará, y si la marca de tiempo coincide con la esperada. De ser así, comenzará la comunicación con el servidor por tanto tiempo como el TGS haya indicado en el periodo de validez que incluyó en el *Service Ticket*. Cuando pase ese periodo, el cliente tendrá que repetir todo el proceso.

Gracias a la división entre AS y TGS, *Kerberos* ofrece mucha más robustez en la autenticación que un servicio de autenticación habitual de usuario/contraseña. No obstante, el coste a pagar es una mayor lentitud en el proceso de autenticación, más puntos de fallo en el sistema (si se cae el

KDC o una de sus partes, el cliente no podrá comunicarse con el servidor, a pesar de que ambos estén operativos), y sobre todo, una mayor complejidad para implementar y mantener el sistema.

2.10.2 OAuth2/OpenID

Otra alternativa es *OAuth2* y *OpenID*. *OAuth2*²² es un protocolo estándar abierto de autorización que permite a una aplicación obtener un acceso limitado a una cuenta de usuario de otro servicio. Es usado por muchos servicios web de renombre como *GitHub*, *Twitter* o *Facebook*. El ejemplo habitual de uso es el de dar permiso a aplicaciones de terceros para tener acceso a una cuenta de usuario de un servicio sin tener que compartir usuario y contraseña con dicha aplicación de terceros, ya que con el protocolo *OAuth2*, es el propio servicio el que autoriza a la aplicación.

*OpenID*²³ es un protocolo de autenticación basado en *OAuth2* que permite a un usuario identificarse en un servicio a través de otro servicio en el que ya está previamente registrado. Esto da comodidad al usuario, que evita tener que registrarse en el servicio, y además añade seguridad, ya que el servicio autenticador suele ser más fiable. *OpenID* es un protocolo que está muy extendido hoy en día, principalmente en los botones de “Iniciar sesión con *Google*” (y muchos otros, como los de *Facebook* o *Twitter*).

El funcionamiento de *OpenID*, resumido gráficamente en la figura 2.6, es el siguiente:

Supongamos que el cliente quiere identificarse en el servicio A (de ahora en adelante, el “servidor”) a través de su cuenta de usuario en el servicio B (de ahora en adelante, el “autenticador”). El servidor tendrá que tener habilitada la integración con el autenticador, habiendo previamente activado y configurado en el autenticador el servicio de autenticación para su página.

Cuando el usuario hace click en el enlace de iniciar sesión, se envía una petición GET a un *endpoint* específico del autenticador, que devuelve una página de inicio de sesión. En ocasiones, el servidor implementa esta fase enlazando el botón de inicio de sesión a un *endpoint* propio (por ejemplo, `/login`), y que éste *endpoint* redirija automáticamente al *endpoint* del autenticador anteriormente mencionado. En cualquiera de los dos casos, el cliente acaba en la misma página.

Allí, el autenticador pedirá al usuario que inicie sesión con su cuenta, y posteriormente le pedirá confirmación acerca de los datos personales que va a compartir con el servidor. Una vez

²²<https://oauth.net/2/>

²³<https://openid.net/connect/>

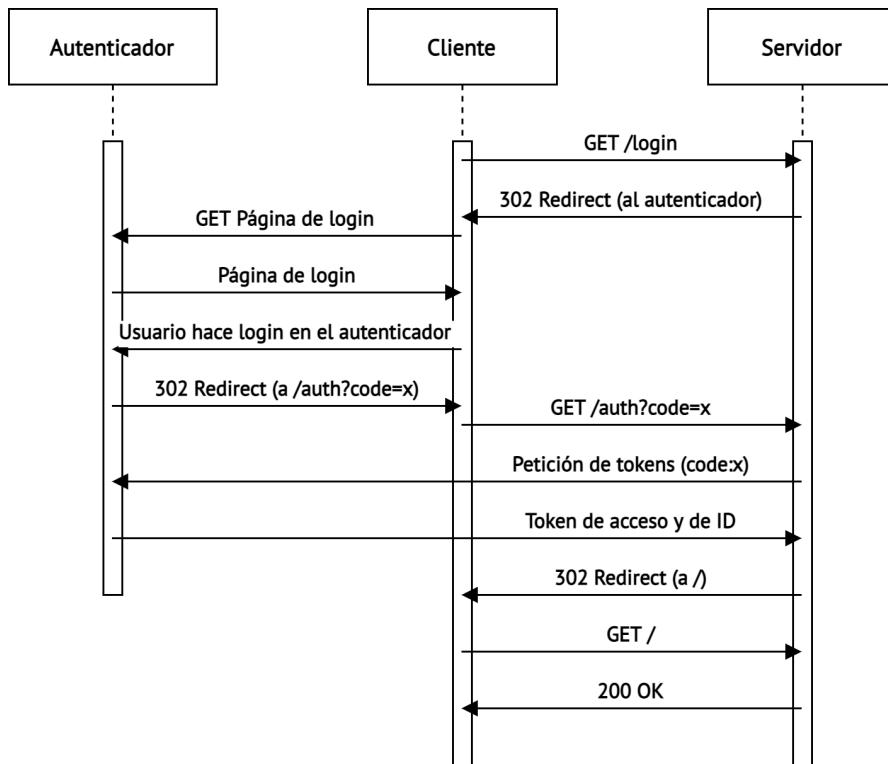


Figura 2.6: Esquema de diálogo de autenticación mediante OpenID.

aceptado, el autenticador envía una respuesta de redirección hacia el *endpoint* que haya configurado el servidor para manejar la autenticación OpenID (por ejemplo, /auth). Esta petición GET cuenta con una serie de parámetros en la *query*, entre los que se encuentra un código de autorización de un sólo uso.

Tras la redirección al *endpoint* de autenticación (/auth), el servidor recibe los parámetros de la *query*, y envía ese código a otro *endpoint* específico del autenticador en una petición POST. Si el código es válido, el autenticador enviará de vuelta un array JSON con información, incluyendo el token de acceso y el token de ID. Este último es un JWT (JSON Web Token), un objeto JSON firmado digitalmente por el propio autenticador, codificado en *base64*. Tras decodificarlo y comprobar la validez de su firma, podemos extraer de él los datos del usuario brindados por el autenticador.

Una vez terminado el intercambio de datos entre el autenticador y el servidor, el servidor redirige al usuario a otra página de su dominio (normalmente, la página principal). Con estos datos, el servidor puede crear un perfil y abrir una sesión al cliente, sin haber tenido éste que registrarse ni haber compartido ninguna contraseña con el servidor.

De entre todas las alternativas, hemos decidido usar *OpenID*. Tanto Kerberos como *OpenID* se basan en el mismo principio: confiar en una tercera parte la autenticación del usuario. Ambos son estándares muy usados, y ambos ofrecen un nivel de seguridad adecuado. No obstante, la principal diferencia reside en que *OpenID* permite integrar un autenticador fiable ya existente como *Google*, *Facebook*, *Twitter* o *GitHub*, además de ofrecerte la posibilidad de crear un servidor autenticador propio, mientras que *Kerberos* sólo ofrece esta segunda opción (montar un *KDC* propio). Así pues, y dado que nuestro servidor no deja de ser una prueba de concepto que sirva de referencia para entornos empresariales que quieran implementar una solución similar, hemos decidido optar por la opción más sencilla: *OpenID*. Concretamente, utilizando a *Google* como autenticador externo, ya que su servicio es gratuito, simplifica mucho la tarea de autenticación en el lado del servidor, y está extendido a nivel mundial.

Capítulo 3

Diseño e implementación

A continuación se explicará todo el diseño e implementación del proyecto, empezando por una visión general, y después incidiendo en las partes fundamentales del proyecto: la API REST del servidor, la autenticación en el servidor y el paquete de verificación de integridad y autenticidad.

3.1 Arquitectura general

En la figura 3.1 se puede ver la arquitectura general del proyecto. Vemos que nuestro sistema incluye un cliente y un servidor, y que el cliente puede comunicarse con nuestro servidor tanto a través de un navegador web como a través de la clase `RemoteServer` de nuestro paquete `ml-fingerprint`.

En la interacción mediante el navegador web, el cliente primero tendrá que identificarse con su cuenta de *Google*, ya que la autenticación de nuestro servidor implementa el servicio *OpenID* de *Google*. Veremos con más detalle cómo funciona la autenticación en el capítulo 3.3.

Una vez iniciado sesión, el cliente tendrá acceso a la lista de modelos disponibles, aunque no podrá descargar, subir, modificar ni eliminar ninguno desde la interfaz web. Adicionalmente, el cliente tiene acceso a una página de perfil, donde podrá generar una *API Key*. Esta *API Key* es un código aleatorio de 16 bytes codificado en *base64*, generado con la función `token_urlsafe` del paquete `secrets` de la biblioteca estándar de Python. Tiene una validez de un día y queda asociado en la base de datos *SQLite3* del *backend* al correo electrónico del usuario (obtenido de la información que *Google* nos brindó a través del protocolo *OpenID*). La *API Key* sirve para poder

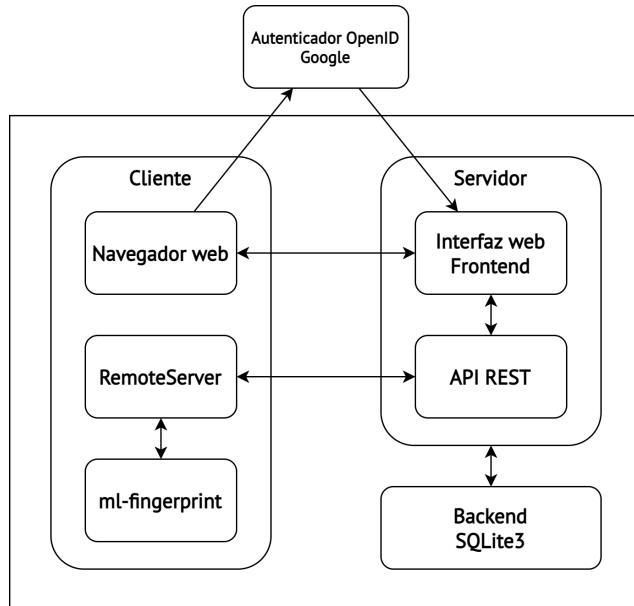


Figura 3.1: Estructura general del proyecto.

acceder al servidor mediante Python, como veremos a continuación.

En la interacción mediante Python, el cliente accede al servidor a través de la clase `RemoteServer`, perteneciente a nuestro paquete `ml-fingerprint`. Esta clase tiene implementados unos métodos para servir de interfaz ante todos los *endpoints* que ofrece la API REST del servidor, permitiendo así ver la lista de modelos y descargar, subir, modificar o eliminar un modelo en concreto. Al descargar un modelo, automáticamente se verifica la firma del mismo, en caso de tenerla, comprobando así su integridad y autenticidad. Entraremos en detalle en el capítulo 3.4.

Cabe destacar que se ha utilizado `SQLite3` como plataforma para alojar la base de datos por su simplicidad, ya que tan sólo consiste en un archivo. Otras alternativas como `MySQL` o `PostgreSQL` pueden ser más eficientes, pero son mucho mas complejas de implementar.

3.2 Arquitectura de la API REST

En la figura 3.2 se muestran los *endpoints* que ofrece nuestra API REST para ser usados a través de la interfaz web. A continuación se describirá cada uno de ellos:

- **GET /:** Muestra la página principal del proyecto, que incluye, entre otras cosas, enlaces a la documentación del paquete `ml-fingerprint`, así como un botón para iniciar sesión, en caso de no tener una sesión activa, o un enlace para acceder a la página de perfil, otro

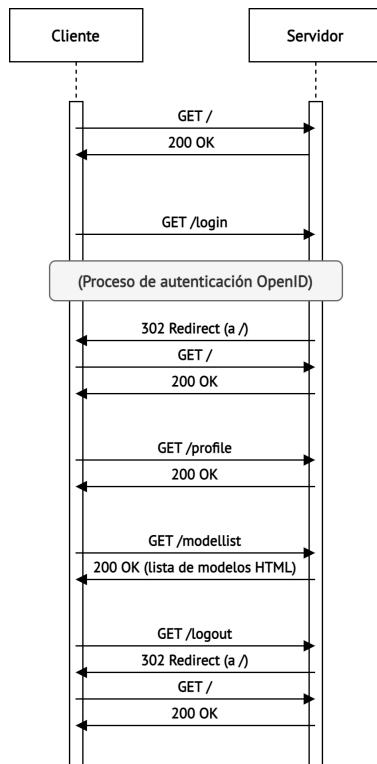


Figura 3.2: Esquema de diálogo cliente-servidor a través de la web.

para ver la lista de modelos disponibles y otro para cerrar sesión, en caso de sí tener una sesión activa.

- **GET /login:** Inicia sesión mediante el protocolo OpenID, usando Google como autenticador externo. El proceso de autenticación se explicará más adelante en profundidad. Una vez finalice el proceso de autenticación, se redirige al usuario a la página principal, con la sesión iniciada.
- **GET /profile:** Página de perfil de usuario, sólo accesible cuando el cliente tenga una sesión iniciada. De no ser así, intentar acceder a /profile devolverá un código de error 403. En la página de perfil se muestra el nombre completo del usuario, una caja de texto con la *API Key*, el periodo de validez de dicha clave, y un botón para generar una nueva *API Key*. Pulsar el botón de generar una *API Key* enlaza a la misma página /profile, pero con un parámetro adicional en la *query*: *generatekey=true*. Al recargar la página con este parámetro adicional, el servidor genera una nueva *API Key*, que como ya se ha mencionado, es una clave de 16 bytes codificada en *base64*. El servidor guarda en la base de datos la clave junto con el correo electrónico del usuario, la fecha de creación (la fecha

actual) y la fecha de expiración (la fecha actual, más un día), y devuelve la página HTML con la nueva *API Key* en la caja de texto.

- **GET /modellist:** Muestra los nombres y las características de todos los modelos disponibles en la base de datos del servidor. Admite varios parámetros adicionales en la *query* de la petición GET: *type*, *allversions* y *format*. El parámetro *type* permite filtrar la lista por tipo de clasificador (por ejemplo, *classifier* o *clustering*). Adicionalmente, se puede filtrar por aprendizaje supervisado o no supervisado dando los valores *supervised* o *unsupervised* (ejemplo: `/modellist?type=supervised`). El parámetro *allversions*, acompañado del valor *true* (`/modellist?allversions=true`) permite mostrar todas las versiones de los modelos. Si no se especifica, por defecto sólo muestra la última versión de cada modelo. El parámetro *format* permite elegir el formato de salida de los datos. Sólo hemos implementado el formato JSON, al cual se accede con `/modellist?format=json`, y está pensado para ser usado desde Python, como veremos más adelante. Si no se especifica un formato, devuelve la página HTML de la lista de modelos.
- **GET /modellist/<modelname>:** Este *endpoint* es una extensión del anterior, con la diferencia de que muestra todos las versiones de un modelo en concreto, para permitir comparaciones entre versiones.
- **GET /logout:** Cierra la sesión actual. Redirige al *endpoint* /.

En la figura 3.4 se muestran las cinco operaciones que ofrece nuestra API REST para ser usadas a través de Python (mediante cualquier paquete que permita hacer peticiones HTTP, como *Requests* o *urllib2*). A continuación se describirá cada uno de los *endpoints*.

- **GET /model/<modelname>:** Obtiene de la base de datos el modelo con el nombre indicado. Requiere el parámetro *api_key* en la *query*, cuyo valor será la *API Key* del usuario. Además, admite un parámetro adicional, *version*, que permite indicar la versión concreta que queremos obtener del modelo. Si no se especifica, se obtiene la última versión disponible. Si el modelo ha sido encontrado, envía una respuesta HTTP con el código 200 y el modelo serializado junto con todos sus metadatos en un objeto JSON en el cuerpo. Si no se ha encontrado, devuelve un código de error 404. Si la *API Key* no es válida, está caducada o no se ha incluido en la *query*, devuelve un error 403.

The screenshot shows a web interface for an API named 'ml-fingerprint'. At the top, there's a navigation bar with links for INDEX, DOCUMENTATION, MODEL LIST (which is highlighted in green), JAIME SOLSONA, and LOGOUT. Below the navigation, it says '2 MODELS'.

australia_rain_predict
/model/australia_rain_predict

Linear classifier that predicts if it will rain tomorrow. The dataset is taken from Kaggle and can be found here: <https://www.kaggle.com/jphyy/weather-dataset-rattle-package>. It consists of daily weather data in Australian cities, and whether it rained the next day or not.

Owner: Jaime Solsona | Type: Supervised (Classification) | Model: LogisticRegression | Created: 14 Jan. 2021 00:53 | Version: 1.0.0
 Accuracy: 0.8491 | Serialized to bytes with pickle and then to ASCII with base64.

vanderplas_regression_example
/model/vanderplas_regression_example

Example linear regressor taken from "Python Data Science Handbook", by Jake VanderPlas. The training points follow the function $y = -2x + z$, with an added noise. The regressor then tries to predict the value of y based on x and z values.

Owner: Jaime Solsona | Type: Supervised (Regression) | Model: LinearRegression | Created: 14 Jan. 2021 01:02 | Version: 1.0.0
 MSE: 0.0003 | R2: 0.9999 | Serialized to bytes with pickle and then to ASCII with base64.

At the bottom of the page, there's a footer bar with the text '2021 © All Rights Reserved.'

Figura 3.3: Página HTML de lista de modelos.

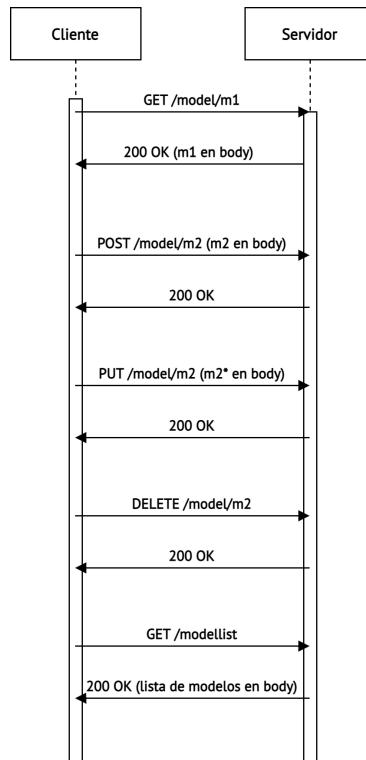


Figura 3.4: Esquema de diálogo cliente-servidor a través de Python.

- **POST /model/<modelname>**: Sube a la base de datos un modelo. En el cuerpo de la petición POST tiene que ir, en formato JSON, el modelo serializado y sus metadatos. Los campos requeridos para este objeto JSON se describirán más adelante. De manera adicional a los campos correspondientes al modelo, el objeto JSON del cuerpo tendrá que incluir el campo `api_key`, con la *API Key* del usuario. Si el modelo ha sido insertado correctamente en la base de datos, envía una respuesta HTTP con el código 200. Si ya existe un modelo con ese mismo nombre y versión en la base de datos, no lo inserta, y devuelve un código de error 400. Al igual que en la petición GET, si la API Key no es válida o no ha sido incluida, devolverá un error 403.
- **PUT /model/<modelname>**: Actualiza un modelo de la base de datos. En el cuerpo de la petición PUT tiene que ir el modelo y sus metadatos en mismo objeto JSON de las petición POST, incluyendo el campo extra para la *API Key*. Si se ha encontrado un modelo con ese mismo nombre y versión, actualiza el modelo en la base de datos y envía una respuesta HTTP con el código 200. Si no existe un modelo con ese mismo nombre y versión en la base de datos, devuelve un código de error 404. Al igual que en el resto de *endpoints*, los errores referentes a la *API Key* devolverán un error 403.
- **DELETE /model/<modelname>**: Borra de la base de datos el modelo con el nombre indicado. De forma similar al *endpoint* GET, el DELETE también requiere el parámetro `api_key` en la *query*, además de admitir el parámetro adicional `version`, que permite indicar la versión concreta que queremos borrar del modelo. Si no se especifica, se borra la última versión disponible. Si el modelo ha sido encontrado y borrado correctamente, envía una respuesta HTTP con el código 200. Si no se ha encontrado, devuelve un código de error 404. Al igual que en el resto de *endpoints*, los errores referentes a la *API Key* devolverán un error 403.
- **GET /modellist**: Devuelve la lista de modelos. Es el mismo *endpoint* definido en el apartado anterior. Aquí está pensado para ser usado acompañado del parámetro `format=json`, para poder acceder a la información de los modelos desde el código Python.

Se puede ver un ejemplo del objeto JSON que requieren los métodos POST y PUT en su cuerpo, y que devuelve el GET, en la figura 3.5. El objeto contiene los siguientes campos:

- **name:** El nombre del modelo. En los métodos POST y PUT, deberá coincidir con el introducido en el *endpoint*.
- **serialized_model:** El modelo de *Scikit-learn*, serializado primero a bytes con el serializador de bytes indicado, y después a texto con el conversor de bytes a texto indicado.
- **serializer_bytes:** El serializador utilizado para convertir el modelo de un objeto Python a una cadena de bytes. De manera general, se suele utilizar *pickle*. Tiene que indicarse para saber con qué paquete deserializar el modelo.
- **serializer_text:** El conversor utilizado para convertir la cadena de bytes a texto compatible con JSON y HTTP. Habitualmente, se suele utilizar *base64*, que asigna los bytes a caracteres ASCII (mayúsculas A-Z, minúsculas a-z, números 0-9 y dos caracteres adicionales, normalmente + y /). Tiene que indicarse para poder reconvertir la cadena de texto en una cadena de bytes.
- **supervised:** Entero, 1 si el modelo pertenece a la categoría de aprendizaje supervisado, y 0 si pertenece al aprendizaje no supervisado.
- **type:** El tipo del modelo. Por ejemplo, clasificador, regresión o *clustering*.
- **estimator:** La clase de *Scikit-learn* del modelo. Por ejemplo, *LinearRegression* o *RandomForestClassifier*.
- **scores:** Un objeto en formato JSON con distintos indicadores de precisión del modelo. Sigue el formato de un diccionario clave - valor, donde la clave será el nombre del indicador, y el valor será el propio valor del indicador.
- **version:** Una cadena de texto con la versión del modelo.
- **metadata:** Un objeto en formato JSON que permite guardar cualquier información adicional sobre el modelo, siguiendo el formato de diccionario clave - valor ya descrito.
- **date:** Fecha de creación del modelo.
- **description:** Descripción del modelo.

```
{
  "id": 1,
  "name": "australia_rain_predict",
  "serialized_model": "gASV2QUAAAAAAACMHnNrbGVhcm4",
  "serializer_bytes": "pickle",
  "serializer_text": "base64",
  "supervised": 0,
  "type": "Classification",
  "estimator": "LogisticRegression",
  "scores": {
    "Accuracy": 0.8491021613765296
  },
  "version": "1.0.0",
  "metadata": {},
  "date": "2021-01-14T00:53:14.548148",
  "description": "Linear classifier that predicts",
  "owner": "Jaime Solsona",
  "email": null
},
```

Figura 3.5: Ejemplo de objeto JSON de un modelo.

Al margen de estos campos, el objeto JSON devuelto por el método GET contiene dos campos adicionales:

- **owner**: El nombre del usuario que subió el modelo al servidor, obtenido de la información de inicio de sesión obtenida de *Google*.
- **email**: El correo electrónico del usuario que subió el modelo al servidor, obtenido también de *Google*.

Todos estos campos, incluidos los dos últimos, están guardados en la base de datos *SQLite3* dentro de la tabla `models`.

3.3 Arquitectura de la autenticación OpenID

En el capítulo 2.10.2 se explicó el diálogo que comparten cliente, servidor y autenticador para autenticar al usuario. En la figura 3.6 se muestra un resumen gráfico de la conexión entre usuario, servidor y autenticador. A continuación, se expondrán los detalles concretos de nuestra implementación.

En primer lugar, hubo que encontrar y habilitar un autorizador OpenID externo para que dé servicio a nuestra página. En este caso, decidimos usar *Google*, ya que es gratuito, fácil de implementar y está muy extendido en todo el mundo. Para ello, creamos un proyecto nuevo dentro de *Google Cloud Platform* y activamos un ID de cliente OAuth 2.0, en el cual especificamos la URL de nuestro servidor hacia la cual redirigirá tras iniciar sesión, así como los datos que pediremos

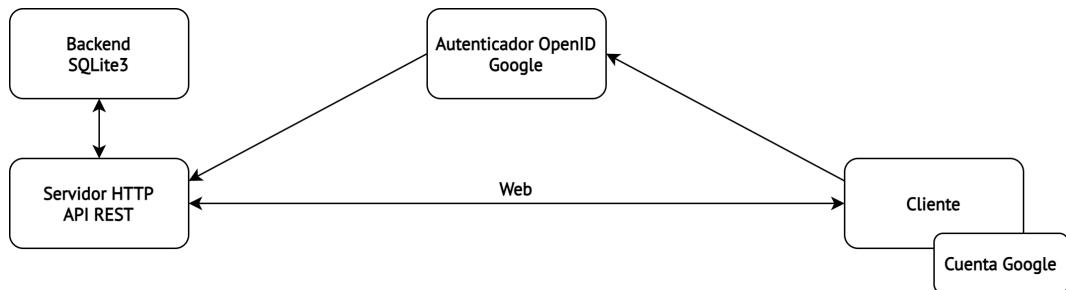


Figura 3.6: Diagrama general de conexión con el servidor mediante la web.

al usuario compartir con nuestro servidor. Después, configuraremos nuestro servidor para enlazar a la página de inicio de sesión de *Google* y recoger los datos del usuario posteriormente. Para ello, nos ayudamos de la biblioteca *Authlib*, que proporciona un conector OpenID para *Flask*. Establecemos el *endpoint* `/auth` como punto al cual redirigir cuando se haya iniciado sesión en *Google*, en el cual se establecerá la conexión entre el servidor y *Google* para el intercambio de datos.

Una vez configurado todo, el funcionamiento es el explicado en el capítulo 2.10.2. El *endpoint* intermedio al que enlaza el botón de iniciar sesión es `/login` (tras hacer click, redirige automáticamente a la página de inicio de sesión de *Google*), y el *endpoint* de autenticación al cual redirige *Google* es `/auth`, como ya hemos mencionado.

En nuestro caso, de todos los datos que nos brinda *Google* sobre el usuario, sólo nos interesan el nombre completo y la dirección de correo electrónico, para poder asociarlos tanto a la *API Key* del usuario como a los modelos que el usuario suba o modifique.

Una vez el servidor ha extraído estos datos del usuario, abre una sesión para el cliente (mediante *cookies* de sesión), y le redirige a la página principal, donde aparecerá ya con la sesión iniciada.

3.4 Arquitectura de la clase RemoteServer

La clase `RemoteServer` es una interfaz diseñada para que el usuario pueda subir, bajar, modificar o borrar modelos del servidor, además de ver la lista de modelos, de la manera más sencilla posible.

En la figura 3.7 se muestran todas las funciones que contiene la clase `RemoteServer`, y a qué *endpoint* del servidor se conecta cada una. A continuación, se detallará su funcionamiento.

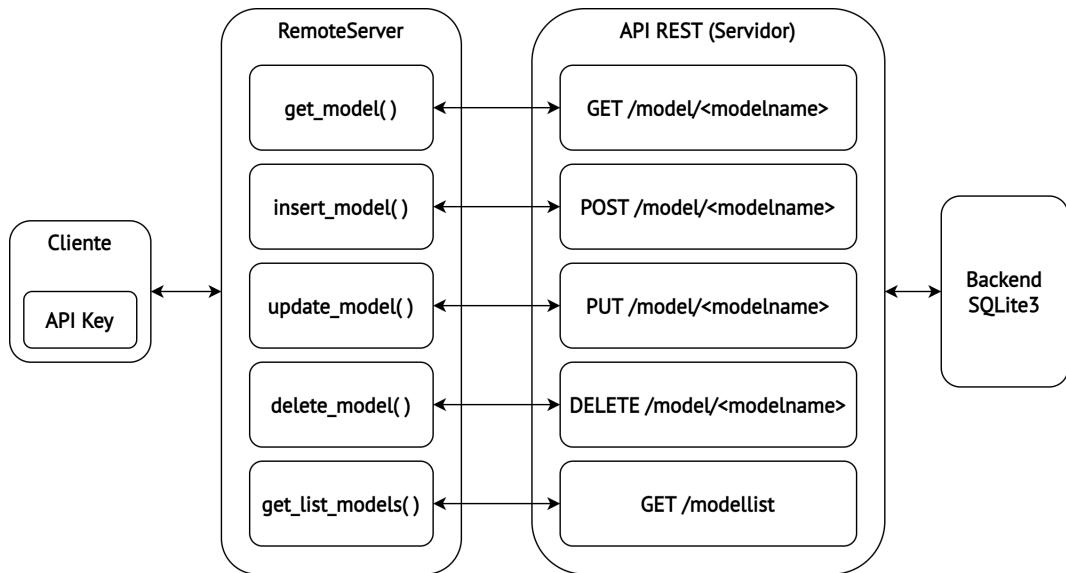


Figura 3.7: Arquitectura de la clase `RemoteServer` y sus conexiones con la API REST del servidor.

`RemoteServer` es una clase, lo que significa que el usuario deberá instanciar un objeto de esta clase para poder utilizar sus funciones. En el momento de instanciarlo, el constructor de la clase requiere dos parámetros: la URL del servidor al cual el usuario quiere conectarse, y la *API Key* del usuario, sin la cual el servidor rechazaría todas sus peticiones.

Una vez construido el objeto de clase `RemoteServer`, se puede llamar a sus cinco funciones para comunicarse con el servidor:

- **`get_model()`:** Descarga un modelo del servidor. Requiere como parámetros el nombre del modelo y la clave pública correspondiente a la clave privada con la que se firmó dicho modelo. La función hace una petición GET al endpoint `/model/<nombre de modelo>`, deserializa el modelo recibido, usa el módulo `ml-fingerprint` para comprobar la integridad y autenticidad del modelo (como explicaremos en el capítulo 3.5), y, en caso de que la firma sea válida, devuelve el modelo de *Scikit-learn*.
- **`insert_model()`:** Sube un modelo al servidor. Requiere como parámetros el modelo a subir y los metadatos enumerados en la parte donde se describe el objeto JSON del capítulo 3.2, a excepción de `serialized_bytes`, `serializer_text`, `estimator`, `owner` y `email`, cuyo valor tomará valor automáticamente. La función hace una petición POST al endpoint `/model/<nombre de modelo>` con dichos parámetros en el cuerpo de la petición.

- **update_model()**: Actualiza un modelo del servidor. Requiere como parámetros el modelo a subir y los mismos metadatos descritos anteriormente. La función hace una petición PUT al endpoint /model/<nombre de modelo> con dichos parámetros en el body de la petición.
- **delete_model()**: Borra un modelo del servidor. Requiere como parámetros el nombre del modelo y, de manera opcional, la versión del modelo a borrar. La función hace una petición DELETE al endpoint /model/<nombre de modelo>.
- **get_list_models()**: Devuelve la lista de modelos del servidor. Admite como parámetros opcionales el nombre del modelo, type y allversions, que permiten filtrar la lista tal y como se describió en la definición de este endpoint en el capítulo 3.2. Adicionalmente, permite el parámetro booleano doprint, que permite imprimir por pantalla la lista de modelos antes de retornarla. La función hace una petición GET al endpoint /modellist/ (o a /modellist/<nombre de modelo>, si se ha especificado el nombre de modelo a filtrar), y devuelve una lista con objetos que contienen los metadatos de cada modelo. Es importante clarificar que la lista no contiene los modelos como tal, sólo sus metadatos. La única manera de descargar un modelo del servidor es mediante la petición GET al endpoint /model/<nombre de modelo>.

3.5 Arquitectura del módulo ml-fingerprint

En la figura 3.8 se muestra el proceso que se ha de seguir para verificar la integridad y autenticidad de un modelo de *Scikit-learn*. A continuación, se detallarán los pasos.

En primer lugar, tras instalar e importar el paquete `ml-fingerprint`, hay que llamar a la función `decorate_base_estimator`. Esta función inserta las funciones de firma (`sign`) y verificación (`verify`) en la clase base de *Scikit-learn*, `BaseEstimator`. Todos los estimadores extienden de esta clase base, por lo que ambas funciones estarán disponibles en todos los modelos de *Scikit-learn*, sea cual sea su tipo.

Esto es posible gracias a que, en Python, las funciones son *objetos de primera clase*, lo que significa que se pueden tratar como una variable más. Por tanto, al igual que se puede utilizar la función `setattr` para añadir una variable a una clase (convirtiéndola en un atributo más de

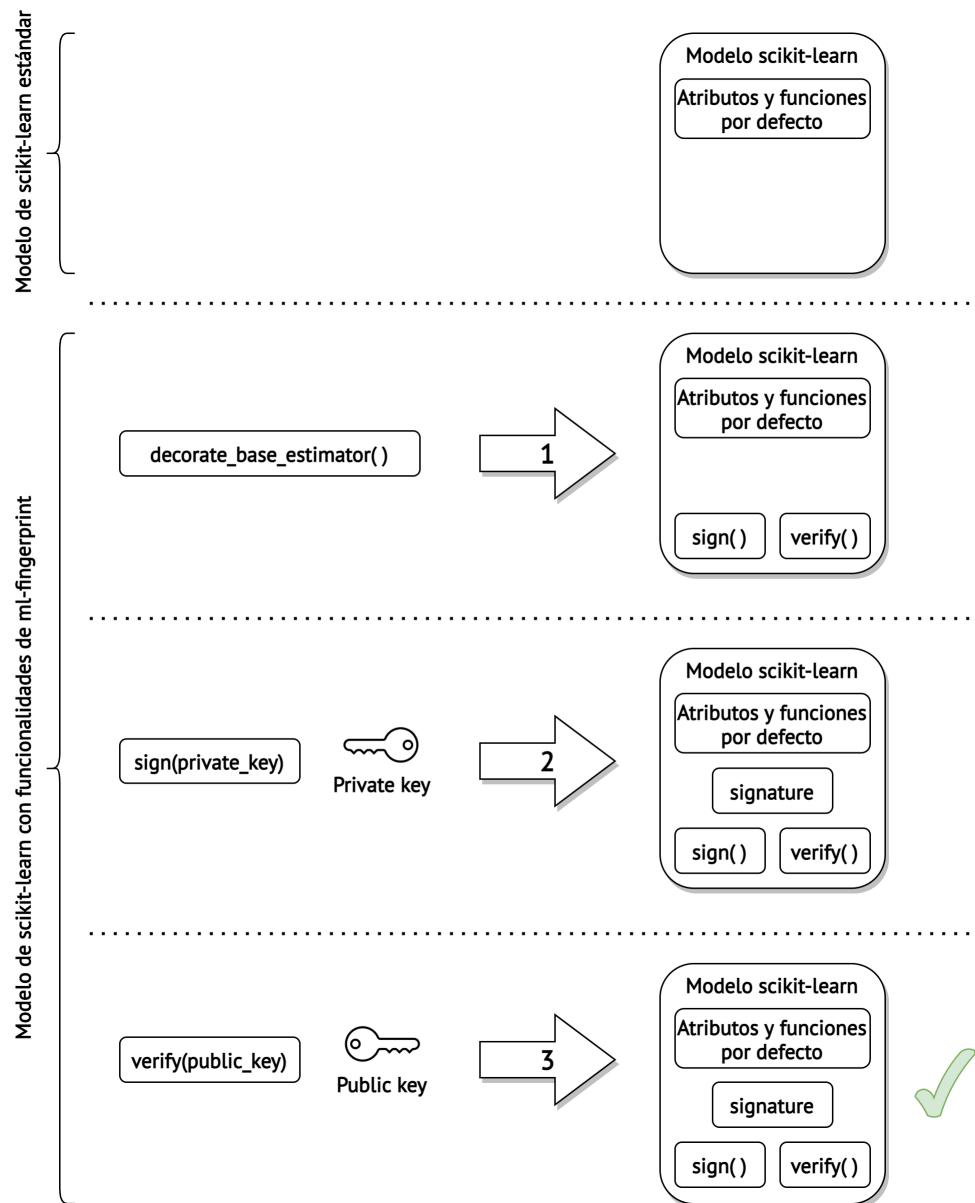


Figura 3.8: Diagrama de las fases de utilización del paquete ml-fingerprint

la misma), se puede utilizar para añadir una función, convirtiéndola así en un método más de la clase. Nótese que estamos añadiendo las funciones a la clase, no a un objeto instanciado de la clase. Por tanto, en el momento en que se llame a `decorate_base_estimator`, tanto todos los modelos ya existentes como los que se creen a partir de este momento tendrán ya integradas ambas funciones.

El siguiente paso es firmar el modelo. Para ello, necesitaremos una clave. Dado que estamos usando criptografía asimétrica, es necesario que dicha clave tenga dos partes: una privada y otra pública. En nuestro caso, estas claves se generarán con el paquete *PyCryptodome*, y serán claves RSA de 2048 bits. La clave privada generada se pasa como argumento a la función `sign`, la función generará un *hash* del modelo y encriptará dicho *hash* con la clave privada, generando así la firma del modelo. El proceso para generar el *hash* del modelo tiene sus peculiaridades, así que a continuación lo vamos a detallar:

Para calcular el *hash* del modelo, la primera opción fue serializar el modelo a bytes usando el serializador estándar de Python, *pickle*, y calcular el *hash* sobre esa cadena de bytes resultante. No obstante, nos encontramos con un problema bastante grave al implementar esta opción: por cómo funciona internamente *pickle*, si se calcula el *hash* de un modelo serializado usando *pickle*, se serializa y deserializa dicho modelo (utilizando *pickle*, o cualquier otro serializador) y se vuelve a calcular el *hash* sobre el modelo serializado, los *hashes* no coinciden. Esto supone que, por ejemplo, si el modelo es guardado en una base de datos, o es enviado por red, la firma del modelo no va a ser válida, ni aunque el modelo no haya sido modificado. Evidentemente, esto impide el propósito principal del proyecto, que es comprobar la integridad y autenticidad de un modelo que hayamos obtenido de otro lugar, así que nos vimos obligados a buscar otra alternativa.

La siguiente opción fue utilizar un serializador JSON. El problema que surgió es que ningún serializador JSON permite serializar un objeto de Python entero, sólo pueden serializar tipos más primitivos como arrays o diccionarios. Ante este inconveniente nos dimos cuenta de que no hace falta serializar el objeto entero, sino que con sólo serializar todos los atributos del objeto bastaría. Por suerte, Python incluye de manera nativa en todos los objetos un diccionario `__dict__` que contiene todos los atributos del objeto. Este diccionario sí podría ser serializado con un serializador JSON, consiguiendo así una representación en texto del modelo sobre la cual poder calcular el *hash*.

No acabaron aquí los problemas. Como ya se ha mencionado, los serializadores JSON sólo

pueden serializar tipos primitivos, por lo que si dentro del diccionario hay un algún elemento que no sea de un tipo primitivo, no podrá ser serializado. El ejemplo más notable son los arrays de *NumPy*, comúnmente usados por *Scikit-learn*, y que además suelen contener información vital, como los coeficientes del modelo. Este problema fue parcialmente resuelto eligiendo *orjson* como serializador, ya que admite de forma nativa arrays de *NumPy*. Aun así, no se puede asegurar que *orjson* pueda serializar todos los atributos de un modelo, ya que existen cientos de modelos en *Scikit-learn*, y resulta inviable comprobar uno a uno los atributos de cada modelo.

Así pues, se tomó la decisión de ignorar los atributos no serializables. Al llamar a la función `sign`, se genera una copia del modelo (mediante `deepcopy`, de la biblioteca estándar), y se recorre todo el diccionario `__dict__` en busca de atributos problemáticos. En caso de haber alguno, se guardará su nombre en una lista llamada `excluded_data` y se borrará del diccionario. Esta lista tiene que ser guardada, ya que el proceso de verificación tiene que repetir este proceso, y necesita saber qué elementos se borraron del diccionario al generar la firma para poder hacer lo propio antes de serializar. Para guardar esta lista, decidimos crear un diccionario llamado `ml_fingerprint_data` y añadirlo al modelo como un atributo más. Dentro de este diccionario se guardará esta lista (bajo la clave “`excluded_data`”), pero también la firma del modelo, como veremos más adelante. Este diccionario también tendrá que ser borrado de `__dict__` antes de serializar, ya que si se tiene en cuenta para calcular la *firma* y después se le añade la propia firma, estamos modificando el diccionario, por lo que la firma dejará de ser válida y la verificación fallará.

Una vez borrados todos los elementos problemáticos, se serializa el diccionario `__dict__` con *orjson*, y se calcula un *hash* utilizando el algoritmo SHA256, uno de los estándares más utilizados a nivel mundial. Este *hash* es encriptado con la clave privada que se ha pasado como argumento a la función, utilizando el esquema de firmado digital *PKCS#1 v1.5*, uno de los sistemas de encriptación más utilizados en todo el mundo, y que, si bien es viejo, sigue siendo seguro a día de hoy. Tanto el *hash* como el firmado se hacen mediante funciones de la librería *PyCryptodome*. El *hash* encriptado es la firma del modelo, que añadiremos al diccionario `ml_fingerprint_data` bajo la clave “`signature`”.

Con el modelo ya firmado, se puede proceder a su verificación mediante la función `verify`. Esta función requiere como parámetro la clave pública que sea pareja de la clave privada con la que se firmó el modelo. Es idéntica a `sign` hasta el punto donde se calcula el *hash*, con única la

diferencia de que para borrar los elementos del diccionario `__dict__` utiliza directamente la lista `excluded_data` de nuestro diccionario `ml_fingerprint_data`. Una vez calculado el *hash* del modelo, utiliza la función `verify` del módulo de *PKCS#1 v1.5* de *PyCryptodome*. Internamente, esta función desencripta la firma del modelo con la clave pública, obteniendo así el *hash* original, y compara ese *hash* con el que acaba de generar. Si ambos coinciden, se puede asegurar la integridad del modelo. Si adicionalmente se puede asegurar que la clave pública pertenece al dueño del modelo (por ejemplo, al haber sido verificada por un tercero de confianza), se puede asegurar también la autenticidad del modelo. En caso de que no coincidan, se elevará una excepción personalizada, perteneciente también al paquete `ml-fingerprint`, llamada `VerificationError`. En caso de llamar a la función `verify` sobre un modelo que no ha sido firmado, se elevará otra excepción personalizada, llamada `ModelNotSigned`. Estas excepciones están declaradas dentro del módulo `exceptions.py` de nuestro paquete.

Capítulo 4

Casos de uso

En este capítulo se explicará paso a paso el proceso a seguir para utilizar nuestro proyecto, desde la autenticación en el servidor hasta el firmado y autenticación de modelos. También se detallarán distintos modelos de *Scikit-learn*, y se demostrará su integración con distintas funcionalidades del servidor.

4.1 Autenticación con el servidor y obtención de la API Key

Para poder subir y bajar modelos del servidor es necesaria una *API Key* y, para generar una, hay que autenticarse en el servidor a través de la interfaz web. Para ello, como ya se ha explicado en los capítulos anteriores, se utilizará una autenticación *OpenID* usando *Google* como tercero de confianza.

El usuario tendrá que entrar con un navegador en la página principal del servidor, mostrada en la figura 4.1. Allí, tendrá que hacer click en el enlace “Sign in with Google” de la parte superior derecha, el cual le redirigirá a la página mostrada en la figura 4.2.

Una vez iniciada la sesión y dado permiso a *Google* para que comparta la información personal del usuario con el servidor, el usuario es redirigido a la página principal, donde encontrará un enlace con su nombre en el lugar del enlace de iniciar sesión. Pinchando en ese enlace, el usuario puede acceder a su página de perfil, mostrada en la figura 4.3. En esta página es donde el usuario puede generar una *API Key* para poder acceder al servidor desde el código, a través de *RemoteServer*. La página muestra la *API Key* anterior, en caso de haberla, y la fecha en la que caduca. En el momento en que el usuario haga click en el botón de generar una nueva *API Key*

What does it do?
This package adds integrity verification to all `scikit-learn` models.

How it works?
`ml-fingerprint` adds two methods into all `scikit-learn` models: `sign(private_key)` and `verify(public_key)`.

`sign(private_key)` takes the current state of the model and generates a signature signed with the private key given. `verify(public_key)` decrypts the existing signature of the model, compares it to the current state of the model and returns `True` if the signature is valid (and therefore the model hasn't been changed since it was signed).

To add both methods into all existing scikit-learn models, the user has to call `decorate_base_estimator()` before using either `sign()` or `verify()`.

Source code GitHub repository of the project.	Sample link This is a sample link that points to Google	Sample link This is a sample link that points to Google	Sample link This is a sample link that points to Google

2021 © All Rights Reserved.

Figura 4.1: Página principal del servidor.

Iniciar sesión con Google

Iniciar sesión
Ir a [ml-fingerprint](#)

Correo electrónico o teléfono

¿Has olvidado tu correo electrónico?

Para continuar, Google compartirá tu nombre, tu dirección de correo electrónico, tu preferencia de idioma y tu foto de perfil con ml-fingerprint.

[Crear cuenta](#) [Siguiente](#)

Figura 4.2: Página de inicio de sesión en Google.

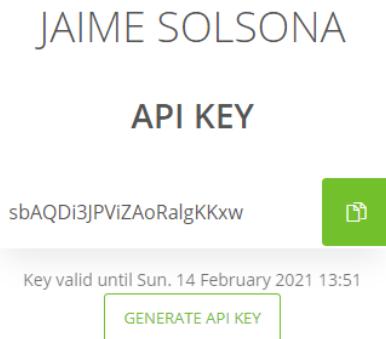


Figura 4.3: Página de perfil de usuario.

esta se mostrará en la caja de texto correspondiente. La *API Key* se podrá utilizar a partir de ese mismo momento y tendrá una vigencia de 24 horas.

4.2 Integración con modelos de Scikit-learn

Una vez tenemos una *API Key*, ya podemos acceder al servidor a través de la interfaz `RemoteServer`. Para ilustrar las distintas funcionalidades de esta clase y su interconexión con el servidor, se han desarrollado tres modelos de *Scikit-learn*:

- Un modelo de clasificación, que a partir de datos meteorológicos de Australia predice si va a llover al día siguiente.
- Un modelo de *clustering*, que a partir de estadísticas de distintos personajes del universo *Pokémon* los agrupa en k distintos grupos.
- Un modelo de regresión, que a partir de datos de viviendas en Boston, predice su precio.

4.2.1 Modelo de clasificación

Este modelo consiste en un estimador de la clase `LogisticRegression` (*Regresión Logística*), que lee un conjunto de datos meteorológicos diarios de distintas ciudades de Australia e intenta predecir, para cada día, si al día siguiente va a llover o no.

El *dataset* fue obtenido de *Kaggle*, una página web que ofrece decenas de miles de *datasets* gratuitos para el desarrollo de modelos de *Machine Learning*. En concreto, el *dataset* utilizado es

Dato	Tipo	Descripción
Date	Categórico	Fecha del dato, en formato YYYY-MM-DD
Location	Categórico	Ciudad correspondiente al dato
Rainfall	Numérico	Cantidad de lluvia caída en el día, en mm
WindGustSpeed	Numérico	La velocidad más alta del viento en el día, en km/h
WindSpeed3pm	Numérico	Velocidad media del viento a las 15:00, en km/h
Humidity3pm	Numérico	Humedad a las 15:00, en porcentaje
Pressure9am	Numérico	Presión atmosférica a las 09:00, en hPa
Pressure3pm	Numérico	Presión atmosférica a las 15:00, en hPa
Cloud3pm	Numérico	Nubosidad del cielo, en octas (valores del 0 al 8)
RainTomorrow	Categórico	Yes o No según si llovió al día siguiente

Tabla 4.1: Diccionario de datos para el modelo de clasificación.

*Rain in Australia*¹, elaborado por Joe Young. El *dataset* ofrece, para un gran número de ciudades australianas, distintos datos: temperaturas, cantidad de precipitación, dirección y sentido del viento, humedad o presión atmosférica, entre muchos otros. Estos datos abarcan un intervalo de casi 10 años (desde 2008 hasta 2017). Junto a los datos diarios se incluye un dato adicional, que indica si llovió al día siguiente o no. Éste será el valor que intentaremos predecir. En la tabla 4.1 se incluye un listado de todas las variables usadas en el modelo, así como una breve descripción de lo que representan.

Antes de empezar a detallar el proceso de preparación de datos y entrenamiento del modelo, hay que destacar que el *dataset* original incluye 23 variables, pero en la tabla sólo hemos indicado 10. Esto se debe a que hemos aplicado el principio de parsimonia: “en igualdad de condiciones, la explicación más sencilla es la más probable” [5, 10]. Primero se entrenó el modelo con todas las variables, y después fuimos eliminando de manera recursiva aquellas que no tenían un gran impacto en el porcentaje de acierto del modelo. Así, conseguimos eliminar más de la mitad de las variables a costa de un mero 0.03% de pérdida de precisión, simplificando de manera considerable el modelo y quedándonos sólo con aquellas que realmente ayudan a predecir la lluvia.

Los estimadores de *Scikit-learn*, por complejos que sean, no dejan de ser funciones matemá-

¹<https://www.kaggle.com/jsphyg/weather-dataset-rattle-package>

ID	Location	ID	Canberra	Sydney	Melbourne
1	Canberra	1	1	0	0
2	Sydney	2	0	1	0
3	Melbourne	3	0	0	1
4	Sydney	4	0	1	0

Tabla 4.2: Ejemplo de conversión de datos categóricos a numéricos mediante valores dummy.

ticas. Por tanto, todos los valores que le introduzcamos para entrenar dicho modelo tienen que ser numéricos, lo que significa que tenemos que convertir todas nuestras variables categóricas a numéricas. Para ello, convertimos cada columna de variables categóricas en “valores dummy” mediante la codificación *One Hot Encoding* [11]. Esta codificación crea columnas de valores binarios, una por cada valor distinto que tenga la columna original. Dichas columnas tendrán todas el valor 0 excepto en la columna que corresponda al valor original, que tomará el valor 1. En la tabla 4.2 se visualiza un ejemplo de *One Hot Encoding*.

Cabe destacar que esta conversión sólo se ha aplicado en las columnas categóricas genéricas (en la práctica, la de `Location`), ya que para convertir `Date` y `RainTomorrow` se han utilizado dos conversiones especiales. En primer lugar, para convertir la de `Date`, simplemente creamos tres nuevas columnas para el día, mes y año, y asignamos los valores correspondientes, convirtiendo así la fecha en tres valores numéricos. Por otro lado, la variable `RainTomorrow` sólo puede tomar los valores `Yes` o `No`, por lo que una simple conversión binaria asignando el 1 a `Yes` y 0 a `No` fue suficiente.

Adicionalmente, aunque no es estrictamente necesario, también se suelen aplicar transformaciones a los valores numéricos para adaptarlos mejor al modelo, consiguiendo así que el modelo predictivo haga un mejor trabajo.

La primera transformación que hemos aplicado es la limitación de valores extremos. Los llamados *outliers* (valores atípicos) son unos valores muy extremos, que se alejan mucho del resto de valores de la distribución de una variable, y que pueden provocar una distorsión en el modelo. Para detectarlos y limitarlos, se establece un umbral máximo, y se da el valor del umbral a todo dato que sobrepase dicho umbral, limitando así su valor. En nuestro caso, y asumiendo que la distribución de los valores sigue una distribución normal, se establece el umbral a tres rangos intercuartílicos (IQR) a partir del tercer cuartil. En la figura 4.4 se explica visualmente

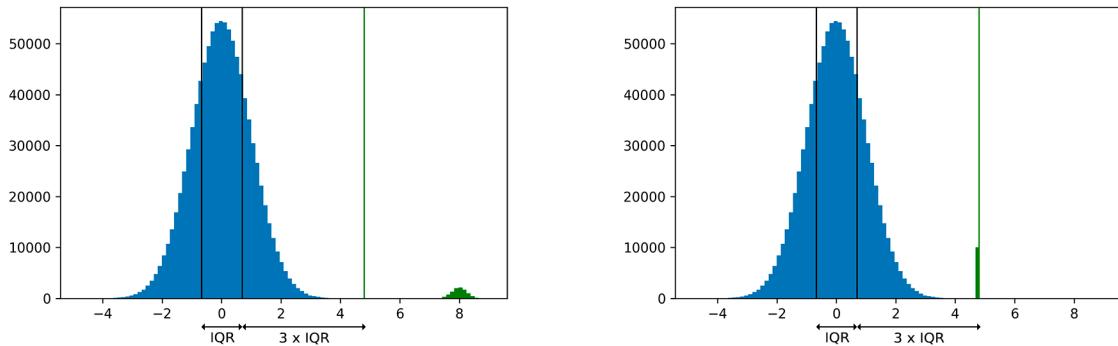


Figura 4.4: Ejemplo de limitación de outliers a una distancia de 3 veces el IQR.

este proceso. En la primera gráfica vemos como la mayoría de datos siguen una distribución normal con media $\mu = 0$, pero además tenemos unos valores extremos, pintados de verde, que de dejarse así, distorsionarían nuestro modelo. Se calcula el rango intercuartílico (entre ambas líneas negras, identificado como IQR), y se calcula el umbral (línea verde) multiplicando por 3 esta distancia a partir del tercer cuartil (la segunda línea negra). Tras ello, en la segunda gráfica vemos cómo todos los valores que sobrepasaron el umbral (pintados de verde para facilitar su distinción) están ahora apilados en el propio umbral.

Otra transformación de datos que es muy habitual para entrenar modelos de *Machine Learning* es el escalado de valores, denominado a veces normalización [11] (aunque este último término corresponde, más bien, a un tipo particular de transformación). Al escalar los datos, hacemos que valores numéricos cuya magnitud es varios órdenes superior a la del resto de variables no enmascaren a estas últimas. En nuestro caso, esto podría ocurrir al enfrentar valores como la presión atmosférica (valores del orden de 1.000) frente a la precipitación acumulada (valores del orden de 1). Para escalar los datos, hemos utilizado un normalizador `MinMaxScaler` (en terminología de la biblioteca *Scikit-learn*), que para cada columna de datos asigna 1 al valor máximo del rango, asigna 0 al valor mínimo, e interpola todos los valores intermedios para que queden en el nuevo rango de 0 a 1:

$$\tilde{x} = \frac{x - \min(x)}{\max(x) - \min(x)} \quad (4.1)$$

Por último, hay que comprobar si en nuestro *dataset* faltan datos, y de ser así, cómo gestionar estas faltas. En nuestro *dataset* hay muchas filas de datos a los que les falta algún dato, por lo que

nos vemos en la obligación de gestionar estos casos. Una opción es eliminar todas las filas en dónde falte algún dato. No obstante, en nuestro *dataset*, faltan datos hasta en un 40% de las filas, por lo que eliminar casi la mitad de los datos no es la mejor opción. La alternativa es llenar estos datos con valores que influyan lo mínimo posible en el entrenamiento del modelo. En nuestro caso, decidimos llenar todos los valores restantes con la mediana de cada columna, para los valores numéricos, y con la moda, para los valores categóricos.

Una vez transformados los datos a un formato adecuado, hay que fraccionar los datos para entrenar y después comprobar la eficacia del modelo. Para ello, se separa el *dataset* en un 70% para entrenamiento (*train*) y un 30% para comprobación (*test*). Es importante desordenar el *dataset* antes de separar los datos de *train* de los de *test*, para evitar correlaciones temporales en caso de estar el *dataset* ordenado temporalmente (como sucede en nuestro caso). Con el *dataset* fraccionado, se separan, tanto de los valores de *train* como de los de *test*, los valores X (todas los datos de la tabla 4.1 excepto RainTomorrow) y los valores y (la columna RainTomorrow, convertida a binario, como se ha explicado antes).

Con el *dataset* fraccionado en cuatro variables (*X_train*, *X_test*, *y_train* e *y_test*), es el momento de entrenar el modelo. Se crea un objeto de la clase `LogisticRegression` y se entrena con el método `fit` del modelo, que recibe como argumentos los valores X e y de entrenamiento (*X_train* e *y_train*). Con el modelo entrenado, se usa el modelo para predecir los valores y de *test* mediante el método `predict` del modelo, que recibe como argumento los valores X de *test* (*X_test*). Estos valores y predichos son comparados con los reales (*y_test*) mediante la función `accuracy_score` de *Scikit-learn*, que devuelve el porcentaje de acierto de la predicción. En nuestro caso, este porcentaje es un 84,91% de acierto en la predicción de lluvia, por lo que estamos ante un modelo de predicción bueno.

Con los valores reales y predichos del modelo podemos calcular también la matriz de confusión, presentada en la figura 4.5. La matriz representa el porcentaje de valores predichos frente a los valores reales. Vemos que el caso más común es que nuestro modelo prediga que no va a llover, y no llueve. Tan sólo en un 3,9% de los datos nuestro modelo ha predicho que va a llover y se ha equivocado.

Sin embargo, resultan más interesante los días que va a llover, ya que es la información que más impacto tiene en la sociedad (por ejemplo, para elegir qué ropa ponerse, coger un paraguas o tender una colada). Esos días se ven reflejados en la fila inferior de la matriz de confusión,

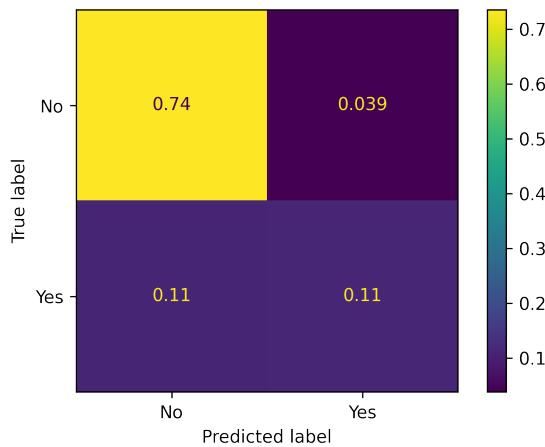


Figura 4.5: Matriz de confusión para el modelo de clasificación.

que corresponde con los días que llovió. Como vemos, a pesar de que nuestro modelo tiene un porcentaje de acierto global del 84,91%, tan sólo acierta el 50% de las veces en los días de lluvia, así que no es tan bueno como pensábamos.

Adicionalmente, vamos a calcular otra métrica: el valor F. El valor F es un parámetro que mide la precisión y exhaustividad del modelo, teniendo en cuenta no sólo la tasa de acierto en la predicción, sino también la proporción de falsos positivos y falsos negativos que hay. En nuestro caso vamos a dar igual importancia a la precisión y la exhaustividad, así que se trata del valor F_1 , que tiene la siguiente fórmula (TP es el número de casos positivos acertados, FN el de falsos negativos y FP el de falsos positivos):

$$F_1 = \frac{2TP}{2TP + FN + FP} \quad (4.2)$$

El valor de ésta métrica, comprendida entre el 0 y el 1 (siendo 0 un modelo muy malo y 1 uno perfecto), tiene un valor de 0.60. Es un valor muy inferior al 0.8491 obtenido en la tasa de acierto, confirmando que la tasa de acierto se ve muy beneficiada y en cierto modo "inflada" por la gran cantidad de días que no llueve, tal y como vimos con la matriz de confusión.

Por último, mediante la librería *Yellowbrick*² vamos a analizar la importancia de cada variable en nuestro modelo. El resultado, acotado para las 5 variables con mayor importancia y mostrado en la figura 4.6, demuestra que las variables con más valor son la humedad, especialmente la correspondiente a la tarde (lógico, pues es un valor más cercano al día siguiente), así como la

²<https://www.scikit-yb.org/en/latest/>

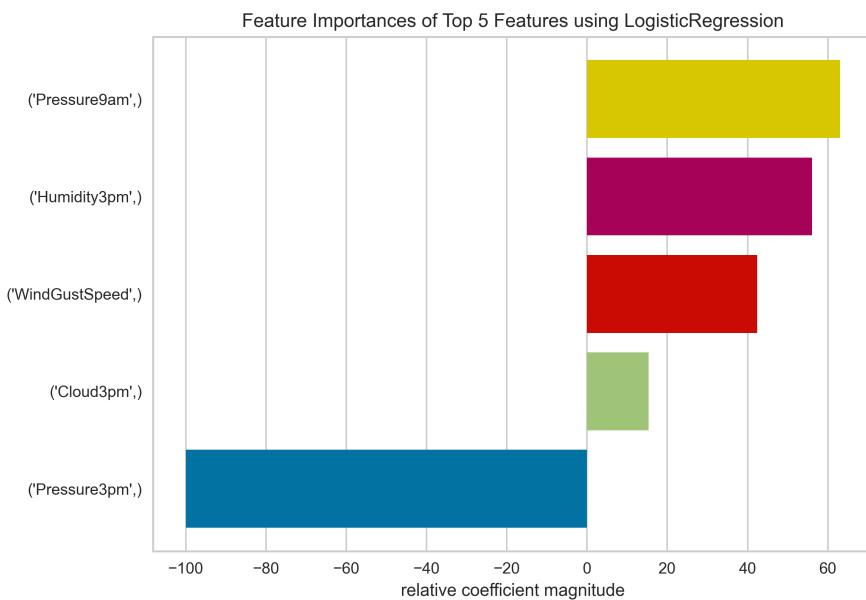


Figura 4.6: Importancia de las variables en el modelo de clasificación.

presión atmosférica, la velocidad del viento y, en menor medida, la nubosidad. Es importante destacar que la importancia de las variables se mide por el valor absoluto, por lo que, a pesar de aparecer el último y con un valor muy negativo, en realidad la humedad a las 3 de la tarde es el valor más importante de todos en nuestro modelo.

Integración del modelo de clasificación en el servicio

A continuación, se explicará la integración de este modelo de clasificación en el servicio, descrita visualmente en la figura 4.7. En este caso, nuestro objetivo será firmar el modelo, subirlo al servidor y posteriormente descargarlo, comprobando la firma al hacerlo. Antes de comenzar, hay que mencionar que este modelo está disponible dentro del módulo `example_models` de nuestro paquete `ml-fingerprint`. La función `rain_classifier` devuelve dicho modelo ya entrenado.

En primer lugar, hay que firmar el modelo. Para ello, tal y como se explicó en el capítulo 3.5, hay que llamar a la función `decorate_base_estimator` para habilitar las funciones de firma y verificación en los modelos de *Scikit-learn*. Tras ello, hay que firmar el modelo con la clave privada. En un entorno habitual real, las claves ya estarán generadas, y además, la clave pública estará verificada por un tercero de confianza. En este caso, dado que estamos en un caso de ejemplo, crearemos sobre la marcha una pareja de claves público-privadas mediante la función

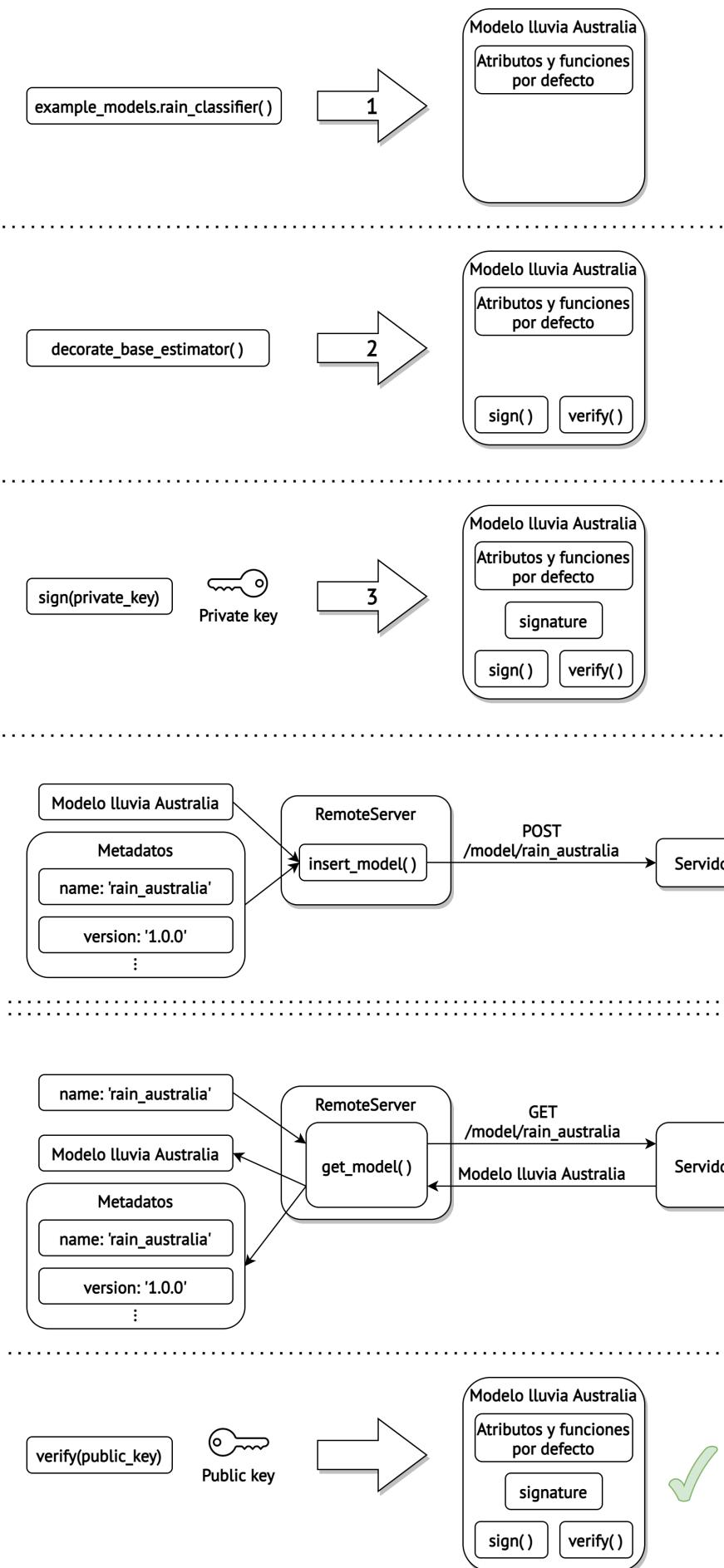


Figura 4.7: Ejemplo de integración del modelo de clasificación en el servicio.

`RSA.generate(2048)` de la librería *PyCryptodome*. Utilizamos la clave privada para firmar el modelo mediante la función `sign(private_key)` que hemos integrado previamente en el modelo. En este momento, el modelo ya está firmado y listo para ser subido al servidor.

Tras firmar el modelo, se instancia un objeto de la clase `RemoteServer`, cuyo constructor requiere la URL del servidor y la *API Key* necesaria para acceder al mismo. Con el objeto ya instanciado, se llama a su función `insert_model`, que recibe como argumentos el modelo y sus metadatos, que rellenaremos de forma manual. Tras la llamada a la función, y si no se ha recibido ningún mensaje de error, el modelo ya estará subido en el servidor. El proceso interno de interconexión entre `RemoteServer` y el servidor ya ha sido explicado en el capítulo 3.4.

Con el modelo ya alojado en el servidor, podemos descargarlo en cualquier otro ordenador para volver a trabajar con él. Para simplificar el ejemplo, lo descargaremos desde esta misma sesión. Para ello, con el mismo objeto de la clase `RemoteServer` que hemos instanciado anteriormente, llamamos a su función `get_model`, pasándole como argumentos el nombre del modelo (que será el que hayamos indicado en los metadatos en el paso anterior) y la clave pública (pareja de la clave privada que generamos anteriormente). La propia función `get_model` obtiene el modelo del servidor y ejecuta la función `verify` del modelo, para comprobar con la clave pública si la firma es correcta o no. En caso de estar todo en orden, la función `get_model` devuelve el objeto del modelo, listo para seguir trabajando con él.

En el código 4.1 se presenta el código utilizado para este caso de ejemplo.

4.2.2 Modelo de clustering

El modelo de clustering se basa en el algoritmo *k-means* [3], construido mediante un estimador de *Scikit-learn* de la clase `KMeans`. El algoritmo lee datos de un *dataset* con información de personajes *Pokémon*. Cada *Pokémon* tiene unas estadísticas numéricas que definen su comportamiento dentro del juego, cómo se puede ver en la figura 4.8. Esto significa que existen, de manera objetiva, *Pokémon* mejores y peores. El objetivo de este modelo es conseguir, de manera no supervisada, dividir todos los *Pokémon* en *k* grupos a partir de sus estadísticas. Posteriormente comprobaremos si estas divisiones tienen sentido o han sido aleatorias.

Este *dataset*, al igual que el anterior, también fue obtenido de *Kaggle*. Su nombre es *The Complete Pokemon Dataset*³, y ha sido elaborado por Rounak Banik. El *dataset* ofrece multitud de

³<https://www.kaggle.com/rounakbanik/pokemon>

información acerca de cada uno de los 801 *Pokémon* que vienen incluidos en él, pero a nosotros sólo nos interesan las estadísticas base, comúnmente llamadas *stats*. Los *stats* son 6 números enteros, descritos en la tabla 4.3, que definen el comportamiento de cada *Pokémon*. Por ejemplo, un *Pokémon* ofensivo tendrá más puntos de Ataque y Ataque Especial que de Defensa y Defensa Especial, mientras que a uno defensivo le pasará justo lo contrario. De igual manera, un *Pokémon* más fuerte tendrá una suma total de *stats* mayor que la de uno débil. Estas relaciones serán las utilizadas posteriormente para el análisis de los grupos.

En esta ocasión no hemos tenido que procesar los datos antes de entrenar el modelo. Los 6 valores de estadísticas son todos numéricos y están en el mismo rango (entre 0 y 255), así que no es necesario escalarlos o normalizarlos. Así pues, entrenamos el modelo *k-means* pasándole los datos tal cual vienen en el *dataset*.

Un aspecto importante a tener en cuenta es que hay que indicar previamente el número de grupos *k* en el que queremos que el modelo divida el *dataset*. Para ello, hicimos varias pruebas con distinto número de grupos, y decidimos quedarnos con *k* = 4, ya que hizo una división de

```
from ml_fingerprint import ml_fingerprint, example_models
from Crypto.PublicKey import RSA
from datetime import datetime

# Generamos el par de claves público-privadas
key = RSA.generate(2048)
private_key = key
public_key = key.publickey()

# Insertamos las funciones de firma y verificación en los modelos de Scikit-learn
ml_fingerprint.decorate_base_estimator()

# Obtenemos el modelo de nuestro módulo de modelos de ejemplo y lo firmamos
model, scores = example_models.rain_classifier()
model.sign(private_key)

# Subimos el modelo al servidor
rem.insert_model(model, "rain_australia", True, "classification", scores,
                 "1.0.0", {}, datetime.now(), "Predictor de lluvia en Australia")

# Descargamos el modelo del servidor
server_model = rem.get_model("rain_australia", public_key)
```

Código 4.1: Código para el caso de ejemplo del modelo de clasificación de lluvia en Australia.



Figura 4.8: Ejemplo de un Pokémon y sus estadísticas dentro del juego.

Dato	Tipo	Descripción
hp	Numérico	Puntos de vida
attack	Numérico	Ataque físico
defense	Numérico	Defensa física
sp_attack	Numérico	Ataque especial
sp_defense	Numérico	Defensa especial
speed	Numérico	Velocidad

Tabla 4.3: Diccionario de datos para el modelo de clustering.

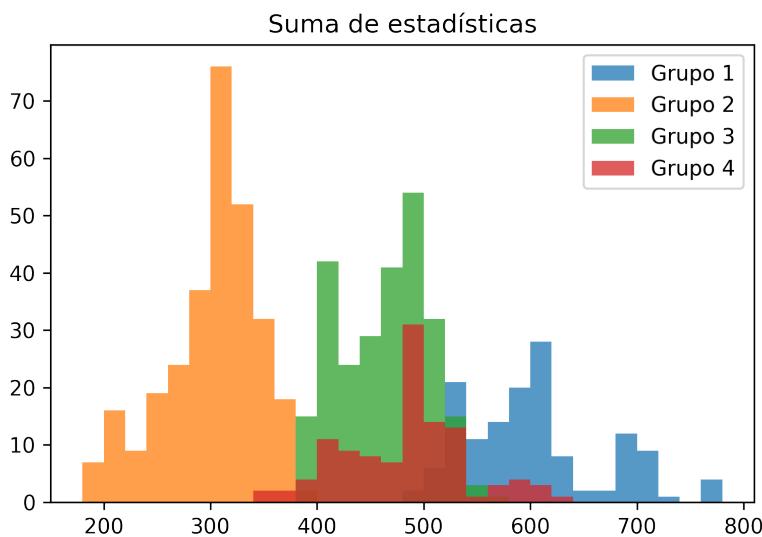


Figura 4.9: Histograma de la suma total de estadísticas para cada grupo.

grupos que, bajo nuestro criterio, tenía bastante sentido.

Con los datos listos y el número de grupos elegido, se procede al entrenamiento del modelo, primero instanciando un objeto de clase `KMeans`, y después entrenándolo con su función `fit`, a la que se le pasa el *dataset* con las 6 estadísticas. Éste nos devuelve 4 grupos de desigual tamaño (140, 292, 256 y 113 *Pokémon*, respectivamente). A continuación haremos un análisis de estos grupos para comprobar si tiene sentido dicha división.

En primer lugar, dado que la mayoría de los *Pokémon* tienen una o varias evoluciones (un *Pokémon* se convierte en otro), y que estos *Pokémon* evolucionados siempre tienen mejores estadísticas que sus antecesores, es fácil pensar que habrá un gran número de *Pokémon* débiles, que aún no han evolucionado, y otro gran número de *Pokémon* fuertes, que sí han evolucionado. Así pues, el primer análisis a hacer será el de la suma total de las 6 estadísticas para cada *Pokémon* de cada grupo. En la figura 4.9 se muestra el histograma de esta suma de estadísticas para cada grupo.

De la figura 4.9 podemos sacar la primera conclusión clara: el grupo 1, representado en azul, ha agrupado a los *Pokémon* más fuertes, y el grupo 2, representado en naranja, ha agrupado a los más débiles. Tanto el grupo 3 como el 4 han agrupado a los *Pokémon* intermedios, los que no son tan fuertes como para ser considerados los mejores, pero tampoco tan débiles como para ser considerados los peores. Queda averiguar qué distinción ha hecho el modelo para separar a los

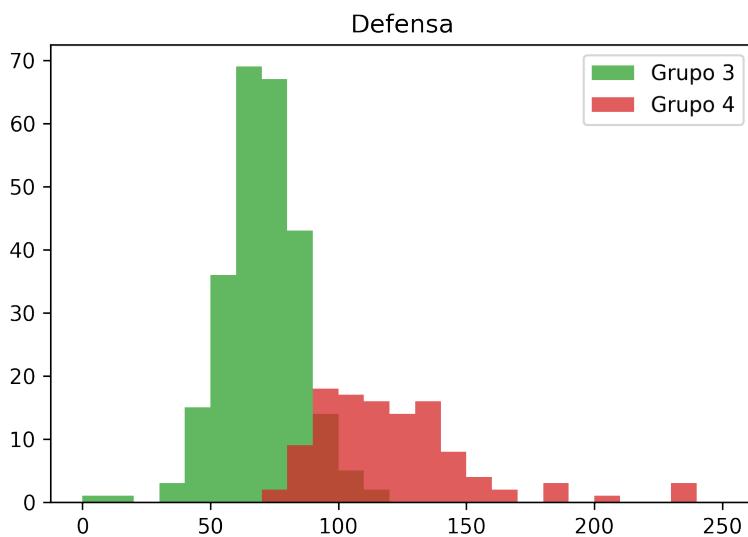


Figura 4.10: Histograma de la estadística de Defensa para los grupos 3 y 4.

Pokémon entre los grupos 3 y 4.

Echando un rápido vistazo a los histogramas de la distribución de cada estadística por grupo no vemos una distinción clara entre ambos grupos, excepto en el de Defensa, mostrado en la figura 4.10. Aquí podemos ver claramente cómo el grupo 4 tiene a albergar *Pokémon* mucho más defensivos que el grupo 3.

Integración del modelo de clustering en el servicio

Una vez explicado el modelo, vamos a poner otro caso de ejemplo de su integración con el servicio. En esta ocasión, supondremos que el modelo ya está alojado en el servidor, y el usuario quiere ver la lista de modelos, descargar este modelo, hacerle una pequeña modificación y reflejar este cambio en el modelo del servidor. Se adjunta una explicación visual en la figura 4.11.

El procedimiento inicial es similar al descrito en el ejemplo del anterior modelo. Al igual que aquella vez, tenemos que instanciar un objeto de la clase `RemoteServer` con la URL del servidor y la *API Key*. Esta vez, dado que el modelo ya está en servidor, en lugar de generar un par de claves privado-pública, vamos a cargar ambas desde un archivo, que serán las mismas utilizadas para firmar el modelo que ya está subido al servidor.

En primer lugar, el usuario desea comprobar la lista de modelos. Puede hacerlo tanto a través de la página web como a través de la clase `RemoteServer`. Desde la web podrá ver la lista

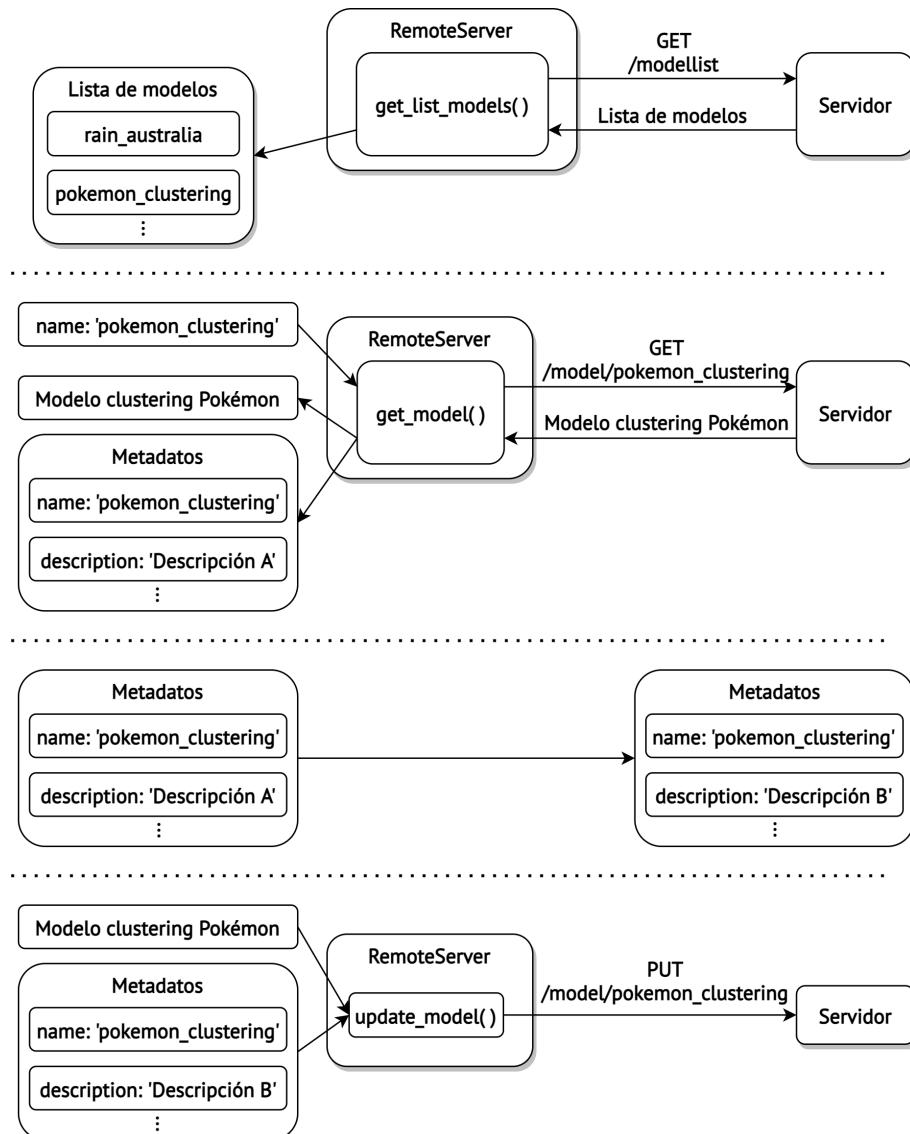


Figura 4.11: Ejemplo de integración del modelo de clustering en el servicio.

```
-- pokemon_clustering --
id: 1
serializer_bytes: pickle
serializer_text: base64
supervised: 0
type: clustering
estimator: KMeans
scores: {}
version: 1.0.0
metadata: {}
date: 2021-02-15T14:24:22.872712
description: Modelo de clustering que agrupa Pokémon en 4 grupos distintos.
owner: Jaime Solsona
email: jaimesolsona7@gmail.com
```

Figura 4.12: Ejemplo de salida al ejecutar la función `get_list_models` de `RemoteServer`.

haciendo click en el enlace de “Model List” de la parte superior derecha. Allí verá una página similar a la mostrada anteriormente en la figura 3.3, que incluirá los datos de nuestro modelo de *clustering*, incluyendo el nombre del modelo, necesario para poder descargarlo posteriormente. Si decide acceder a la lista de modelos a través de `RemoteServer`, tan sólo necesitará llamar a la función `get_list_models` del objeto de clase `RemoteServer`. Esta función devolverá una lista con un diccionario por cada modelo. Cada diccionario tendrá los metadatos del modelo, tal y como se ve en la figura 4.12.

Tras comprobar que el modelo de *clustering* existe, y se llama `pokemon_clustering`, el usuario decide descargarlo. Para ello, tal y como vimos en el anterior ejemplo, tan sólo deberá llamar a la función `get_model` del objeto de clase `RemoteServer`, pasándole como argumentos el nombre del modelo y la clave pública. Tras ello, la función obtendrá el modelo del servidor, verificará su firma con la clave pública y devolverá el modelo al usuario en caso de estar todo en orden.

El usuario entonces decide hacer una pequeña modificación al modelo, como una corrección de errores o algo similar que no suponga una nueva versión del modelo, sino una mera actualización de la versión actual. En nuestro caso, el usuario decide modificar la descripción de los metadatos del modelo. Para subir una modificación de una misma versión del modelo, el usuario tendrá que llamar a la función `update_models` de la clase `RemoteServer`, pasándole como argumentos el modelo y los metadatos recibidos, a excepción de la descripción, que será la modificada por el usuario. Tras finalizar la llamada a la función, el modelo estará actualizado en el servidor, y la modificación podrá verse reflejada en la lista de modelos.

Dato	Tipo	Descripción
RM	Numérico	Número de habitaciones
LSTAT	Numérico	Porcentaje de población de clase baja en el barrio
PTRATIO	Numérico	Número de profesores por alumno del barrio
MEDV	Numérico	Precio de la vivienda

Tabla 4.4: Diccionario de datos para el modelo de regresión.

4.2.3 Modelo de regresión

Este modelo consiste en un estimador de la clase `LinearRegression` (Regresión Lineal), que toma datos de un *dataset* con datos acerca de casas en la zona metropolitana de Boston e intenta predecir el precio de las mismas.

El *dataset* fue obtenido de Kaggle. Se llama *Boston Housing*⁴, y fue creado por Chad Schirmer. Ofrece cuatro datos para cada vivienda, descritos en la tabla 4.4. El objetivo es predecir el precio de la vivienda, correspondiente a la columna MEDV, a partir de las otras tres columnas de datos, mediante un modelo de regresión lineal.

En primer lugar, tenemos que comprobar qué datos tienen más relación con el precio de la vivienda, y qué tipo de relación tienen. Para ello, hemos representado en la figura 4.13 los valores de cada dato frente al precio de la vivienda. Como podemos apreciar, tanto LSTAT como RM muestran una clara correlación con el precio de la vivienda, al contrario que PTRATIO. Además, podemos comprobar que RM tiene una relación lineal frente al precio, mientras que LSTAT aparenta una relación no lineal, si bien la curva no es muy pronunciada y se puede aproximar linealmente.

Por otra parte, para corroborar este primer análisis, hemos dibujado la matriz de correlación de las variables en la figura 4.14. Fijándonos en la última fila, confirmamos que las columnas de LSTAT y RM son las que más correlación tienen con el precio (-0.76 y 0.70, respectivamente). Así pues, nos centraremos en estos dos datos para elaborar nuestro modelo.

Primero vamos a hacer una versión inicial (versión 1) del modelo muy sencilla, que sólo utilice una variable, la que más correlación tenga con el precio (LSTAT). La regresión lineal tendrá

⁴<https://www.kaggle.com/schirmerchad/bostonhousingmlnd>

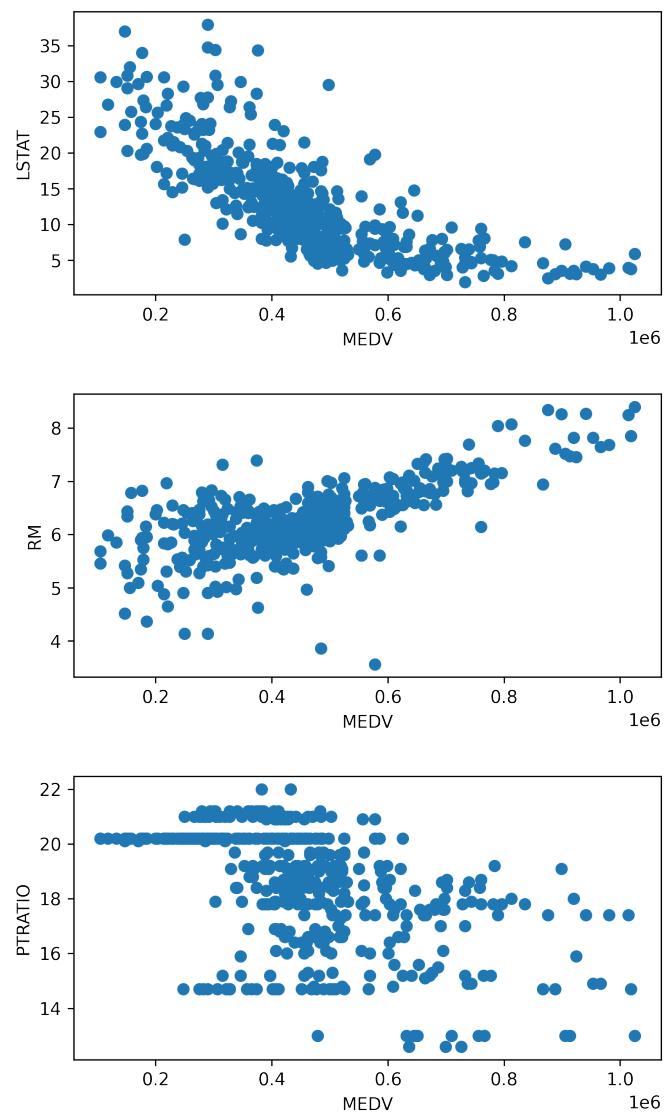


Figura 4.13: Representación de distintas variables del dataset frente al precio de la vivienda.

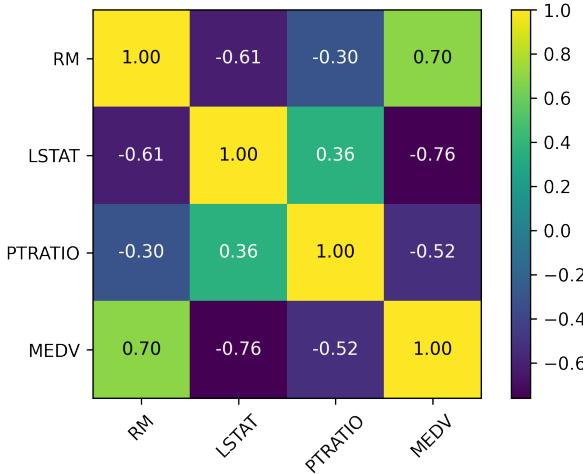


Figura 4.14: Matriz de correlación entre las distintas variables.

la siguiente forma:

$$y = \alpha_0 + \alpha_1(LSTAT) + \epsilon, \quad (4.3)$$

donde α_i son los coeficientes que *Scikit-learn* obtendrá mediante el método de mínimos cuadrados y ϵ denota el error de predicción que comete el modelo. Al igual que en el modelo de lluvia en Australia, sepáramos el dataset en un 70% para entrenamiento y 30% para test, y entremos el modelo pasándole la columna LSTAT como valores de X y la columna MEDV como valores de y . Una vez entrenado el modelo, valoramos su precisión calculando el coeficiente de determinación R^2 con los datos de test.

El coeficiente R^2 tiene la siguiente fórmula:

$$R^2 = \frac{\sum_{i=1}^n (\hat{Y}_i - \bar{Y})^2}{\sum_{i=1}^n (Y_i - \bar{Y})^2} \quad (4.4)$$

Este coeficiente tiene un rango entre 0 y 1, siendo 0 una carencia total de correlación entre los valores predichos y los reales, y 1 una predicción perfecta. El modelo tiene un R^2 de 0.57 para los valores de entrenamiento, y de 0.54 para los de test, los cuales no son valores muy buenos. La paridad entre ambos coeficientes indica que no estamos haciendo un sobreajuste (*overfit*).

Además del R^2 , vamos a calcular otras dos métricas de error: la raíz del error cuadrático medio (RMSE) y el error absoluto medio (MAE). Ambas miden el error del modelo en las mismas unidades que la variable de salida (en nuestro caso, dólares). Sus fórmulas son las siguientes:

$$RMSE = \sqrt{\frac{1}{n} \sum_{i=1}^n (\hat{Y}_i - Y_i)^2} \quad (4.5)$$

$$\text{MAE} = \frac{\sum_{i=1}^n |y_i - x_i|}{n} \quad (4.6)$$

La diferencia entre ambos es que el RMSE, al elevar al cuadrado el valor del error antes de sumarlo a la media y posteriormente calcular su raíz cuadrada, da más peso a los errores grandes. Por tanto, una disparidad entre los valores de RMSE y MAE puede ser indicador de la presencia de algunos errores muy por encima de la media.

En nuestro caso, los valores de RMSE y MAE son de \$114.877,37 y \$84.714,11, respectivamente. Teniendo en cuenta que el valor medio de las viviendas de nuestro *dataset* es de \$454.342,95, estos valores de error son altos, acorde con el pobre valor de R^2 obtenido anteriormente. También podemos observar que no hay una gran disparidad entre ambos valores, por lo que el error está bastante distribuido entre todos los datos.

Para tratar de mejorar el modelo, vamos a incluir en el modelo de regresión la otra variable que mostraba alta correlación, RM. La regresión lineal de la versión 2 de nuestro modelo tendrá la siguiente forma:

$$y = a_0 + a_1(\text{LSTAT}) + a_2(\text{RM}) + \epsilon \quad (4.7)$$

En esta ocasión, el modelo nos devuelve un R^2 de 0.67 para los valores de entrenamiento, y de 0.64 para los de test, lo cual supone una notable mejora sobre la versión anterior, aunque siguen sin ser unos valores brillantes. Los valores de RMSE y MAE son de \$102.232,84 y \$76.714,57, también inferiores a los de la versión anterior, aunque siguen siendo altos.

Como último intento para mejorar nuestro modelo, vamos a considerar la no linealidad de la variable LSTAT, añadiendo un término de segundo grado al modelo. La versión 3 quedaría así:

$$y = a_0 + a_1(\text{LSTAT}) + a_2(\text{RM}) + a_3(\text{LSTAT})^2 + \epsilon \quad (4.8)$$

Esta versión final del modelo devuelve un R^2 de 0.73 para los valores de entrenamiento, y 0.71 para los de test, otra mejora sustancial respecto a las versiones anteriores. Aunque la precisión en la predicción no sea excelente, 0.71 es un valor de R^2 aceptable. Además, hemos conseguido reducir más aún la diferencia entre los coeficientes de entrenamiento y test, por lo que no estamos sobreajustando nuestro modelo. Los valores de RMSE y MAE son de \$91.171,69 y \$70.450,69, también más bajos que los de versiones anteriores.

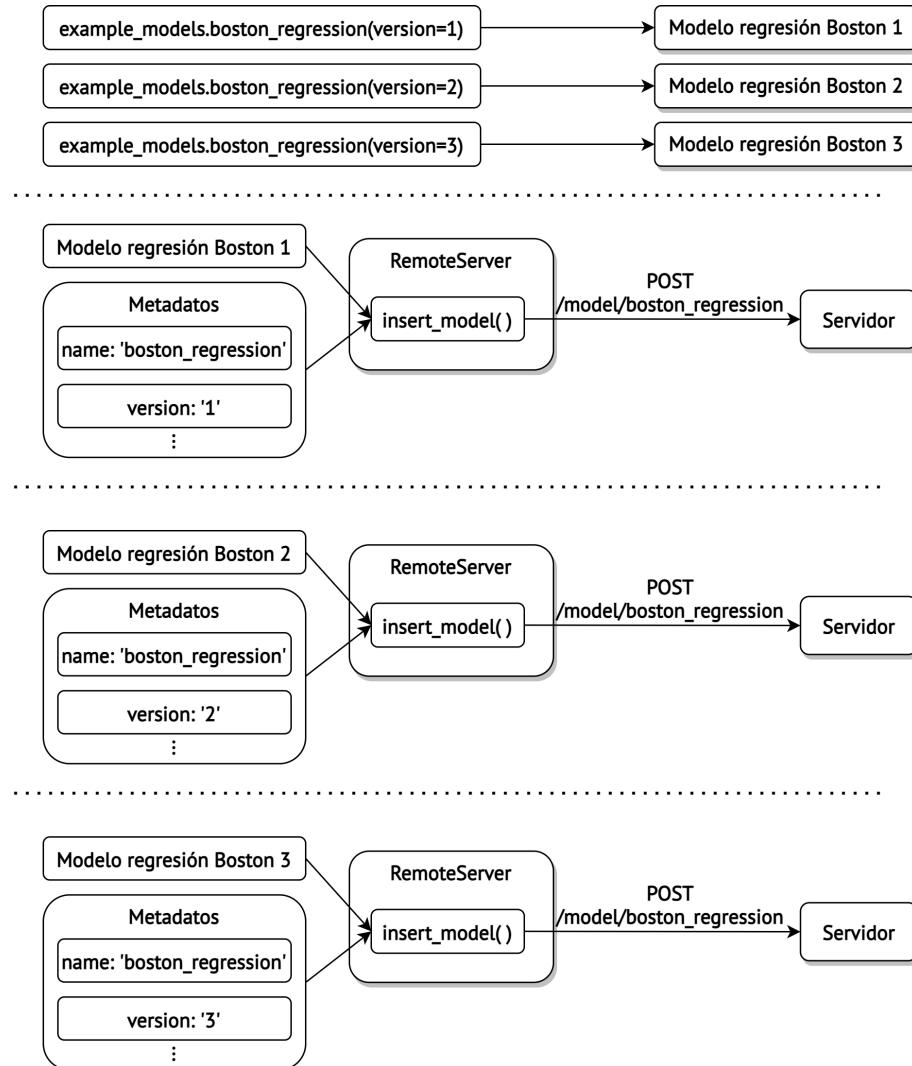


Figura 4.15: Ejemplo de integración del modelo de regresión en el servicio.

Integración del modelo de predicción en el servicio

A continuación, se explicará la integración de estas tres versiones del modelo en el servicio. Partiremos de la base de que el usuario ya tiene una *API Key* válida, un objeto de la clase `RemoteServer` instanciado y el par de claves privado-públicas generadas. Al igual que en los anteriores casos, se adjunta una explicación visual, disponible en la figura 4.15.

En primer lugar, el usuario quiere meter en el servidor la versión 1 del modelo. Para ello, tal y como hemos visto anteriormente, tendrá que llamar a la función `insert_model` del objeto de clase `RemoteServer`, pasándole el modelo y sus metadatos como parámetros, incluyendo un “1” en el campo `version`. Tras ello, el modelo ya estará disponible en el servidor.

En segundo lugar, el usuario quiere subir la segunda versión del modelo, sin modificar ni

boston_regression

/model/boston_regression

Predicción de precios de viviendas en Boston.

👤 Jaime Solsona

⚙️ Supervised (regression)

.LinearRegression

📅 03 Mar. 2021 14:15

ⓘ 3

📊 R2: 0.7135

✉️ Serialized to bytes with pickle and then to ASCII with base64.

boston_regression

/model/boston_regression

Predicción de precios de viviendas en Boston.

👤 Jaime Solsona

⚙️ Supervised (regression)

.LinearRegression

📅 03 Mar. 2021 14:15

ⓘ 2

📊 R2: 0.6398

✉️ Serialized to bytes with pickle and then to ASCII with base64.

boston_regression

/model/boston_regression

Predicción de precios de viviendas en Boston.

👤 Jaime Solsona

⚙️ Supervised (regression)

.LinearRegression

📅 03 Mar. 2021 14:15

ⓘ 1

📊 R2: 0.5452

✉️ Serialized to bytes with pickle and then to ASCII with base64.

Figura 4.16: Captura de pantalla de la web mostrando las tres versiones del modelo.

borrar la anterior que ya está subida. Para ello, simplemente tendrá que insertar el modelo repitiendo los pasos anteriores, llamando a la función `insert_model`, y asegurándose de que pone un “2” en el campo `version`. Tras esto, ambas versiones del modelo estarán disponibles en el servidor.

Por último, el usuario quiere meter el tercer modelo, y para ello, tendrá que repetir los pasos anteriores, poniendo un “3” en el campo `version`. Tras ello, las tres versiones del modelo estarán disponibles en el servidor, tal y como se puede observar en la figura 4.16 . Llegados a este punto, si el usuario ejecuta la función `get_model` del objeto de clase `RemoteServer` indicando sólo el nombre del modelo, obtendrá la última versión. Si quiere obtener alguna de las versiones anteriores, tendrá que indicar la versión concreta como parámetro adicional de la función `get_model`.

Capítulo 5

Conclusiones

En este capítulo se expondrán las conclusiones finales del proyecto, enumerando los objetivos conseguidos y sugiriendo posibles mejoras aplicables en el futuro.

5.1 Consecución de objetivos

Pusimos dos objetivos principales para este trabajo fin de grado:

En primer lugar, hemos conseguido añadir las funciones necesarias a *Scikit-learn* que permiten verificar la integridad y autenticidad de sus modelos de Machine Learning. Además, hemos conseguido hacerlo de una manera totalmente transparente, ya que cualquier ningún modelo de *Scikit-learn* pierde ninguna funcionalidad al añadir las funciones de firma y verificación.

Por otra parte, hemos conseguido desarrollar una API REST en un servidor web que permite trabajar con modelos de *Scikit-learn*, y hemos conseguido desarrollarla con un control de acceso y seguridad suficientes como para ser integrada en un entorno empresarial. Además, nuestro paquete de verificación y autenticación está totalmente integrado con esta API REST, ofreciendo a los usuarios un acceso rápido y sencillo al servidor que apenas interfiere con su flujo de trabajo habitual.

5.2 Aplicación de lo aprendido

Durante el transcurso de mi doble grado, he cursado varias asignaturas que me han servido de gran ayuda para la elaboración de este Trabajo Fin de Grado. Estas son:

1. **Procesamiento Digital de la Información**, impartida por Óscar Barquero y, especialmente, Sergio Muñoz, que fue el profesor que nos explicó con brillantez la base de los estimadores, clasificadores y demás aspectos del Machine Learning, parte fundamental de este proyecto.
2. **Servicios y Aplicaciones Telemáticas**, impartida en mi año por Jesús M. González Barahona, donde aprendí a programar en Python, a desarrollar servidores web con una API REST, y al manejo de sesiones y cuentas de usuario, parte también fundamental en este proyecto.
3. **Ingeniería de Sistemas Telemáticos**, impartida por mi tutor de TFG, Felipe Ortega, donde aprendí el paradigma de la programación orientada a objetos (OOP), así como los patrones de diseño, ambos necesarios para el desarrollo de nuestro paquete *ml-fingerprint*.
4. **Ingeniería de Sistemas de Información**, impartida por Pedro de las Heras, donde aprendí a utilizar bases de datos SQL, necesarias para el *backend* de nuestro servidor, y a manejar repositorios *Git*, utilizado para todo el desarrollo del proyecto.

5.3 Lecciones aprendidas

Durante el transcurso de este Trabajo Fin de Grado, he adquirido los siguientes conocimientos:

1. He aprendido a desarrollar distintos tipos de modelos de Machine Learning con *Scikit-learn*.
2. He mejorado mi conocimiento acerca de servidores HTTP y API REST, y he aprendido a desarrollar servidores con *Flask*.
3. Mis habilidades para desarrollar código con programación orientada a objetos ha mejorado considerablemente, así como mi conocimiento sobre el funcionamiento interno de Python.

5.4 Trabajos futuros

Este proyecto puede dar lugar a muchas extensiones del mismo, como por ejemplo:

- Ampliar las funcionalidades del servidor web, añadiendo más *endpoints*.
- Crear un gestor de versiones de los modelos que permita llevar un control automático, en lugar de poner el número de versión de forma manual en los metadatos.
- Dar soporte para más formatos de cifrado y tipos de *hash* para la firma y verificación.
- Desarrollar un serializador que permita serializar el modelo entero, sin tener que dejar fuera de la firma las variables incompatibles con *orjson*.
- Dar soporte para modelos de Machine Learning de otros paquetes y lenguajes de programación, al margen de *Scikit-learn* y Python.

Bibliografía

- [1] Marco Barreno y col. «The security of machine learning». En: *Machine Learning* 81.2 (2010), págs. 121-148. doi: [10.1007/s10994-010-5188-5](https://doi.org/10.1007/s10994-010-5188-5).
- [2] Scott Chacon y Ben Straub. *Pro Git (Second Edition)*. USA: Apress, nov. de 2014. ISBN: 9781484200773.
- [3] Peter Flach. *Machine Learning. The Art and Science of Algorithms that Make Sense of Data*. Cambridge, UK: Cambridge University Press, sep. de 2012. ISBN: 9781107422223.
- [4] Miguel Grinberg. *Flask Web Development: Developing Web Applications with Python*. USA: O'Reilly Media, mayo de 2014. ISBN: 9781449372620.
- [5] John D. Kelleher, Brian Mac Namee y Aoife D'arcy. *Fundamentals of Machine Learning for Predictive Analytics*. 2nd ed. USA: MIT Press, 2020. ISBN: 9780262044691.
- [6] Valliappa Lakshmanan, Sara Robinson y Michael Munn. *Machine Learning Design Patterns*. Sebastopol, CA, USA: O'Reilly Media, oct. de 2020. ISBN: 9781098115784.
- [7] Arnaud Lauret. *The Design of Web APIs*. USA: Manning Publications, oct. de 2019. ISBN: 9781617295102.
- [8] Foster Provost y Tom Fawcett. *Data Science for Business: What You Need to Know about Data Mining and Data-Analytic Thinking*. Sebastopol, CA, USA: O'Reilly Media, sep. de 2013. ISBN: 9781449361327.
- [9] Matt Taddy. *Business Data Science: Combining Machine Learning and Economics to Optimize, Automate, and Accelerate Business Decisions*. USA: McGraw-Hill Education, 2019. ISBN: 9781260452778.
- [10] Pang-Nin Tan y col. *Introduction to Data Mining*. 2nd ed. NY, NY, USA: Pearson, 2020. ISBN: 9780273769224.

- [11] Alice Zheng y Amanda Casari. *Feature Engineering for Machine Learning*. Sebastopol, CA, USA: O'Reilly Media, abr. de 2018. ISBN: 9781491953242.