Máster en Matemática Industrial

Curso 2022– Curso 2023

Proyecto Fin de Máster

Estudios estadísticos de parámetros de calibrado de boyas M3iGo por Marine Instruments, S.A.

Carlos Vázquez Monzón

Fecha presentación:	19/07/2023
Tutor/a académico/a:	Javier Martínez Torres
Empresa:	Marine Instruments
Tutor/a empresa:	David Vázquez Vázquez

Índice general

Índice de figuras

Resumen

Las boyas satelitales son un tipo de boyas muy útiles para la pesca, sobre todo de atunes. La boya M3iGO, desarrollada por Marine Instruments, es capaz, por una parte, es capaz de estimar la frecuencia del banco de atunes que se encuentra por debajo de la ecosonda y, por otra parte, permite geolocalizar dicho banco. Esta sonda tiene que llevar un proceso de calibración en fábrica. El problema surge cuando, muchas veces, la respuesta de la sonda (los llamados parámetros de salida) no concuerdan con el valor esperado y se salen fuera de la "media" para unos ciertos factores de calibrado (parámetros de entrada).

Nuestro trabajo consistirá en estudiar la discrepancia que existe en los datos experimentales con los datos esperados, en base a datos históricos del calibrado de las boyas. Para ello, utilizaremos técnicas de *machine learning*. Trataremos por separado los problemas de regresión, donde intentaremos aproximar los valores numéricos del calibrado con el mayor grado de precisión posible, y el problema de clasificación, donde buscamos clasificar si el calibrado se sale fuera del rango esperado o no, dependiendo si su desvío de la media teórico concuerda con el esperado.

Nuestros resultados demuestran que, en el problema de regresión, conseguimos predecir los parámetros de calibrado un 97 %, mientras que, en el problema de clasificación, nos encontramos con que un 85 % de las boyas cuentan con un calibrado que caen dentro de los valores esperados, donde en el 15 % restante existen factores que afectan el calibrado que no estamos teniendo en cuenta en el estudio y que desconocemos.

Introducción

La predicción de datos se ha convertido en una herramienta fundamental para la toma de decisiones en la industria moderna. Gracias al creciente volumen de datos disponibles y a la capacidad de procesarlos de manera eficiente, las empresas pueden ahora analizar patrones y tendencias en tiempo real para identificar oportunidades de mejora en sus operaciones y aumentar la eficiencia y la rentabilidad.

En la manufactura, la predicción de la demanda permite a las empresas ajustar sus procesos de producción para evitar la escasez o el exceso de inventario.

En la banca y las finanzas, la predicción de los riesgos de crédito y las tendencias del mercado pueden ser utilizadas para tomar decisiones de inversión más informadas y reducir el riesgo de pérdidas.

En la atención médica, resulta crucial tener información sobre los factores de riesgo que puedan afectar la salud del paciente a futuro, para así identificar a aquellos que tienen un mayor riesgo de desarrollar ciertas enfermedades o complicaciones, lo que les permite intervenir temprano y prevenir problemas graves.

La estadística es una herramienta muy útil para predecir datos en todas estas áreas, ya que nos ayuda a encontrar patrones y tendencias que puedan ser extrapolados al futuro. De hecho, uno de las técnicas más antiguas e importantes sobre la predicción de datos es el **método de mínimos cuadrados**, que fue propuesto independientemente por Gauss y Legendre en el siglo XIX (Gauss 1809; Legendre 1806). Este método se utiliza para ajustar una línea o curva a un conjunto de datos, minimizando la suma de los errores cuadráticos entre los valores observados y los valores predichos por el modelo.

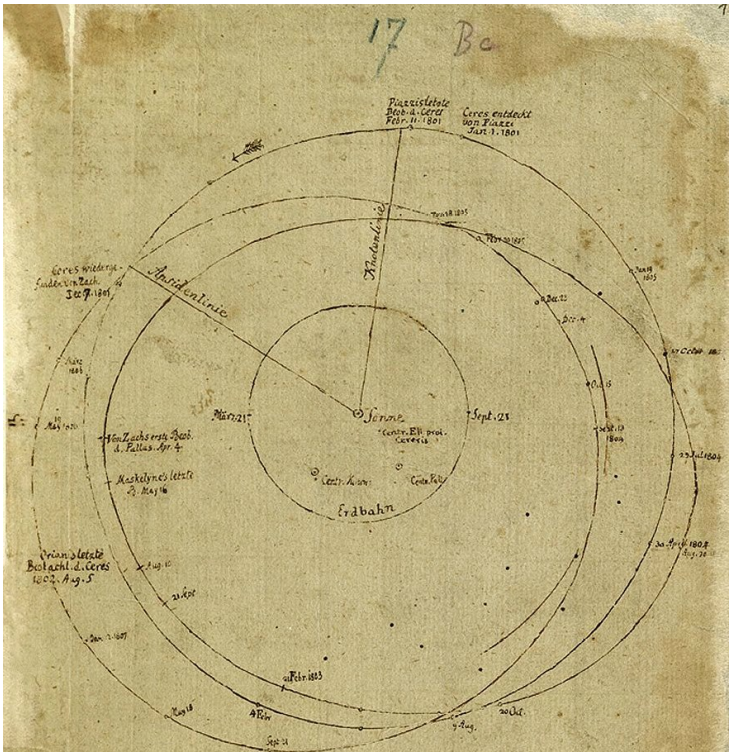


Figura 1: Órbitas de Ceres y Palas, dos objetos del cinturón de asteroides, dibujadas por Gauss en 1809. Estas órbitas fueron ajustadas a los datos de observación mediante el método de mínimos cuadrados (Gauss 1809).

Si bien el método de mínimos cuadrados es una técnica útil para ajustar una línea o curva a un conjunto de datos, tiene sus limitaciones en situaciones donde no existe un modelo matemático conocido y las relaciones entre las variables predictoras y objetivo son demasiado complejas. En estos casos, se necesita un enfoque diferente para modelar patrones complejos en los datos y hacer predicciones precisas.

El *machine learning*, o aprendizaje automático, ofrece una solución a este problema al proporcionar un enfoque flexible y adaptable para modelar patrones en los datos y hacer predicciones precisas sin la necesidad de un modelo matemático preestablecido. En estos casos, a menudo se refiere también

como **aprendizaje estadístico** (*statistical learning*). Estos algoritmos pueden ajustar modelos complejos a los datos, incluso en situaciones donde no se conoce la naturaleza precisa de la relación entre las variables. Por ejemplo, en la detección de fraudes en el sector financiero, los modelos de aprendizaje automático pueden ser entrenados para identificar patrones en los datos que puedan indicar actividades fraudulentas, sin necesidad de conocer su precisa naturaleza.

Concretamente, existe un tipo de modelo de *machine learning* llamado **red neuronal** (véase Sección 3.4) que, teóricamente, permite la construcción de un modelo estadístico todo lo similar que queramos al modelo real. Este hecho está recogido en el *teorema de aproximación universal*, el cual establece que una red neuronal artificial con una sola capa oculta y un número suficientemente grande de neuronas puede aproximar cualquier función continua en un dominio compacto a cualquier nivel de precisión. En otras palabras, la red neuronal puede aprender y modelar la relación entre las variables de entrada y la variable de salida con un grado arbitrario de precisión, incluso en problemas complejos donde la relación es difícil de entender. Este teorema fue demostrado por George Cybenko en 1989, lo que significó un gran avance en la comprensión de las capacidades de las redes neuronales y su aplicación en el mundo real (Cybenko 1989).

Uso histórico del *machine learning* en la industria

Los orígenes del *machine learning* se remontan a los años 50, cuando se comenzó a investigar cómo las máquinas podrían aprender a realizar tareas de manera autónoma. Uno de los primeros antecedentes de aplicación de *machine learning* fue el programa de ajedrez de Claude Shannon, que utilizó técnicas de aprendizaje automático para mejorar la habilidad de la máquina en el juego (Shannon 1950).

En la década de los sesenta, el *machine learning* y la inteligencia artificial eran disciplinas emergentes que comenzaron a atraer la atención de la industria. El interés inicial se centró en el reconocimiento de patrones y la toma de decisiones en entornos complejos. Algunos de los primeros usos de esta tecnología en la industria se dieron en la industria militar, como el programa

DENDRAL, desarrollado por el investigador Joshua Lederberg en la Universidad de Stanford. Este programa fue utilizado para analizar la estructura molecular de los compuestos químicos y se considera uno de los primeros ejemplos de aplicación del *machine learning* en la industria química (Buchanan et al. 1969). Otro ejemplo notable es el programa de lenguaje natural ELIZA, desarrollado por Joseph Weizenbaum en el MIT (Weizenbaum 1966).

En la década de los setenta, el enfoque principal en el campo del *machine learning* se centró en el desarrollo de sistemas de inteligencia artificial que pudieran realizar tareas de razonamiento y toma de decisiones. Otras aplicaciones durante esta década incluyen la identificación de patrones en las señales de radar o la predicción del rendimiento del mercado de valores.

En las décadas de los ochenta y noventa, los sistemas expertos se convirtieron en la principal herramienta de inteligencia artificial en la industria (Buchanan & Smith 1988). Estos programas utilizaban conocimiento explícito para realizar tareas específicas, como el diagnóstico médico o la planificación de rutas de transporte. Aunque los sistemas expertos fueron una importante contribución al campo de la inteligencia artificial, también tenían limitaciones significativas debido a su incapacidad para aprender de los datos y adaptarse a nuevas situaciones.

Cabe decir que, hasta los años noventa, la aplicación de técnicas de *machine learning* en la industria se limitaba en gran medida a aplicaciones especializadas y de nicho. Fue solo con la llegada del siglo XXI que la tecnología comenzó a ser ampliamente adoptada por las empresas en gran variedad de industrias. Por ejemplo, Netflix y Amazon utilizaron algoritmos de *machine learning* para analizar el historial de visualización y compra de sus usuarios, y para recomendar películas y productos que probablemente les gustarían, lo que a menudo se conoce como *sistemas recomendadores*.

En esta y en la última década, el uso del *machine learning* en la industria ha crecido exponencialmente debido al gran aumento en la cantidad de datos disponibles y a los avances en la capacidad de procesamiento de las computadoras, y se ha convertido en una herramienta esencial para analizar grandes cantidades de datos, así como para la optimización de procesos productivos y la mejora de la calidad de los productos.

En definitiva, el *machine learning* es una tecnología en constante evolución desde sus inicios, y su potencial en la industria se ha ido ampliando a medida que las capacidades informáticas han avanzado. Se prevé que su impacto continúe en el futuro, transformando sectores enteros y mejorando la eficiencia y productividad de las empresas. De hecho, según un informe de McKinsey, se espera que su uso y el de otras tecnologías relacionadas con la inteligencia artificial generen entre 3.5 y 5.8 billones de dólares en valor anualmente en 2025 (Bughin et al. 2017).

Marine Instruments

Marine Instruments es una empresa creada en 2004 líder a nivel mundial en desarrollo y fabricación de equipos electrónicos adaptados al medio marino con un enfoque orientado al fomento de océanos inteligentes y pesca sostenible. El centro logístico y de producción está asentado en Nigrán (Pontevedra) y actualmente cuenta con más de 150 empleados. Desde entonces, se ha seguido invirtiendo en I+D como una de las filosofías más sólidas para garantizar el crecimiento de la empresa tanto a nivel humano como tecnológico buscando una mejora continua de nuestros productos a la par que explorando nuevas tecnologías en aras de una mayor diversificación. Es por ello que de los 150 empleados que actualmente trabajan en Marine Instruments, más de 60 están concentrados en el área de I+D, lo que supone alrededor de un 40 % de la plantilla y un pilar fundamental en el ideario de Marine.

Uno de los productos más importantes de la producción de Marine, y en el cual nos centraremos a lo largo de este Proyecto, es la boya satelital M3iGO.

Boya satelital M3iGO

Desde 2005 en Marine Instruments se llevan desarrollando boyas satelitales para la pesca de atún, incorporándose en 2009 las ecosondas, lo cual ha permitido estimar la cantidad de biomasa debajo del FAD (*fish aggregating devices*). La M3iGO es el modelo más reciente de este tipo de boyas fabricada por Marine. Los datos de la ecosonda de la M3iGO permiten a los patrones de los atuneros no sólo geoposicionar el FAD, sino también cuantificar la biomasa e identificar el pescado pequeño o de baja calidad del pescado grande

(véase Sección 1.3.1).

Figura 2: Boya satelital M3iGO

Al igual que todo producto manufacturado, la boya requiere de un proceso de *calibración* para garantizar el correcto funcionamiento de sus funcionalidades. La calibración se lleva a cabo midiendo los parámetros del instrumento y ajustándolo para que los valores medidos coincidan con los valores conocidos y precisos. Los parámetros de calibración pueden ser difíciles de interpretar, y están generalmente influenciados por una variedad de factores, como el uso, el desgaste y la temperatura.

Los productos que no se calibran correctamente pueden proporcionar lecturas erróneas, lo que puede llevar a errores en la toma de decisiones y a la producción de productos defectuosos. Además, una calibración errónea puede resultar en costos adicionales debido a la necesidad de reprocesar productos y en la pérdida de la confianza del cliente en la calidad de los productos. Es aquí donde el *machine learning* nos será de utilidad, ya que nos dará una base

de referencia sobre qué datos de calibrado son los esperados, o "correctos", y nos permitirá compararlos con los datos obtenidos de fábrica. Con esta comparación, no solo podremos calcular cuál es el valor del parámetro de calibrado correcto, sino que podremos descartar aquellas boyas cuyo calibrado se sale del rango esperado.

Objetivos y estructura del PFM

Nuestro objetivo a lo largo de este PFM es el estudio de los parámetros de calibrado de la boya M3iGO para saber, dado unos ciertos datos históricos, si dichos parámetros se corresponden, a la hora de testar una boya, con lo datos esperados. Para ello, establecemos ciertas variables en todo momento conocidas, o de *entrada* que serán las que utilizamos para entrenar los modelos, y variables de *salida*, que son las que buscamos predecir. Analizaremos dos problemas bien conocidos en el campo del *machine learning* supervisado: el problema de regresión y clasificación. En el primero, las variables de salida corresponden a los valores numéricos de los parámetros de calibrado, y buscamos una predicción lo más cercana posible a dichos valores. En el segundo, las variables de salida a predecir son dos clases, "YES" o "NO", dependiendo si cada parámetro de calibrado se desvía de la media menos que una desviación estándar o no.

El PFM se estructura como sigue:

- En el **Capítulo** 1, hablamos de Marine Instruments y de algunos de sus proyectos más importantes.
- En el **Capítulo** 2, detallamos los problemas a resolver y describimos el proceso de calibrado.
- En el **Capítulo** 3, hablamos en detalle de todos los modelos de *machine learning* utilizados a lo largo de este trabajo.
- En el **Capítulo** 4, se detallan todas las técnicas necesarias para analizar los datos de manera efectiva, incluyendo la visualización de patrones y correlaciones, y el procesamiento adecuado de los datos antes de entrenar los modelos.
- En el **Capítulo** 5, se caracteriza el proceso de entrenamiento y evaluación de los modelos de *machine learning*.

- En el **Capítulo** 6, se hace una somera descripción de todo el software utilizado. Nos basamos, principalmente, en la librerías scikit-learn y pandas.
- **Resultados** obtenidos de todo el proceso de entrenamiento y evaluación de los modelos de *machine learning* aplicados a nuestros datos históricos.
- **Conclusiones** de nuestro trabajo
- **Bibliografía**
- **Apéndices** con los códigos de Python utilizados y con un diagrama de flujo indicando la metodología de nuestro trabajo en orden.

Capítulo 1

Descripción de la empresa

Marine Instruments es una sociedad anónima que fue constituida como empresa el 29 de julio de 2003 en Nigrán, Galicia (España). Es una empresa orientada al desarrollo de nuevas tecnologías y fabricación de productos y sistemas electrónicos específicos, con una fuerte especialización en sistemas tecnológicos innovadores de localización y comunicaciones para el medio marino y el sector pesquero.

Desde sus orígenes, la empresa experimentó una rápida y constante expansión dentro del sector marítimo pesquero, situándose en tan sólo unos años como el primer fabricante de boyas a nivel mundial para el sector del atún, gracias, principalmente, a su rigor tecnológico, a su enfoque internacional, a su innovación constante y a los rígidos controles de calidad a los que someten todos sus productos.

La visión de la empresa ha sido siempre internacional debido al nivel de innovación y adaptación de los productos que fabrica. Desde su fundación, Marine Instruments trabaja con clientes internacionales promoviendo la innovación y el desarrollo de productos de la más alta calidad y las mejores soluciones para el mercado internacional.

Marine Instruments ha sido galardonada con el Premio Nacional de Innovación y Diseño del Ministerio de Ciencia e Innovación 2022.

1.1. Estructuración del I + D

Todos los productos desarrollados por Marine Instruments requieren de un importante desarrollo tecnológico en software, hardware y tecnologías de la comunicación. Tanto el software como el hardware están enteramente desarrollados en el departamento de I+D en las oficinas de Marine Instruments.

El departamento de I+D de Marine Instruments está compuesto por tres áreas diferenciadas:

ÁREA HARDWARE: Creación, diseño y prototipado del producto físico.

ÁREA SOFTWARE: Control de comunicaciones vía satélite y desarrollo de software.

ÁREA TÉCNICA: Desarrollo de especificaciones técnicas de producto y elaboración de documentación técnica.

Todos los productos de Marine Instruments son el resultado de proyectos de I+D, por tanto, son productos totalmente nuevos para el mercado. Esto implica un trabajo de desarrollo por parte del departamento de I+D en la fase inicial y la incorporación de otros departamentos en las siguientes fases del producto.

Por tanto, el área de hardware del departamento se encargaría junto con el área técnica del prototipado del producto y el área de software de la creación del software adaptado a ese producto en concreto. Una vez creado el prototipo, éste se somete a varios controles de calidad para garantizar su funcionamiento en condiciones adversas y posteriormente, tras superar todas las pruebas de calidad, se procede a la fabricación del producto para su distribución a nivel internacional.

Una vez introducido el producto en el mercado objetivo, se obtiene todo el *feedback* de cliente y se adapta el producto a sus necesidades específicas con el fin de conseguir un producto totalmente adaptado a clientes y mercados.

En los últimos años, el departamento de I+D ha aplicado su alto conocimiento en comunicaciones y tecnología para desarrollar otros productos para

el sector pesquero que ayuden a mejorar la eficacia y la sostenibilidad.

Además, Marine Instruments, consciente de la importancia de la diversificación, inició en 2018 el desarrollo de nuevos productos innovadores para otros sectores como la Seguridad y Defensa o la Acuicultura, proyectos que tienen continuidad en la actualidad.

1.2. Colaboraciones con Universidades españolas

Desde hace años, Marine Instruments colabora con la Universidad de Vigo y la Universidad de Santiago de Compostela en:

- Máster de Matemática Industrial (M2i) desde 2015. Universidad de Vigo.
- Máster de mecatrónica. Universidad de Vigo.
- Asignatura de Proyectos de Felipe Gil Castiñeira en la facultad de Telecomunicaciones. Universidad de Vigo.
- Departamento de Física no lineal. Universidad de Santiago de Compostela (2019 y 2020).
- Centro Universitario de la Defensa (CUD) en el ámbito de *machine learning* (2019 y 2020).
- Universidad de Santiago de Compostela: convenio de 4 para prácticas de alumnado del Máster universitario de Física y del grado de Ingeniería Informática (2021-2024).
- Fundación Universidad de Vigo: convenio de 4 años para la realización de prácticas académicas con las facultades TIC (2021-2024).
- Universidade de Coruña: convenio de 4 años para la realización de prácticas académicas con las facultades TIC (2021-2024).
- Universidad Autónoma de Barcelona: convenio para desarrollo de trabajo de Máster en Gestión Aeronáutica (2022).

- Fundación Empresa Universidad Gallega (FEUGA): firma de seis convenios para prácticas y trabajos de de Máster en Data Science, Máster Big Data y Máster en Ingeniería de Telecomunicaciones (2022).

1.3. Descripción de los proyectos de I+D

A continuación, se detallan los proyectos de I+D más importantes desarrollados por Marine Instruments en los últimos años que se han lanzado al mercado y han obtenido grandes resultados de ventas.

1.3.1. Proyecto: SISTEMA AVANZADO DE PESCA INTELIGENTE (2007-2008)

Producto comercial: serie boyas satelitales M3i

En el año 2005, Marine Instruments lanzó al mercado la primera **boya satelital**, dando comienzo a la serie M3i que revolucionó el mercado y que ha ido evolucionando hasta nuestros días. Se han vendido 150.000 unidades en todo el mundo desde su lanzamiento y manteniendo el liderazgo mundial hasta la última versión: la **M3iGO**.

La M3iGO es una versión notablemente mejorada con respecto a la anterior, con importantes avances tecnológicos como la introducción por primera vez en el mercado de inteligencia artificial que entrega al patrón, armador y administración las toneladas de pescado comercial puras filtradas de la biomasa y de una sonda bifrequencia para poder mejorar la precisión de las estimas de abundancia y discriminar el pescado, facilitando una pesca selectiva y sostenible que permita optimizar los recursos marinos. Con la introducción de esta boya en el mercado internacional, Marine Instruments consigue avanzar en la obtención de información relativa a los siguientes conceptos:

1. Estimación de **volumen** (toneladas) de pescado comercial.

2. Estimación de **tamaños de pescado** (distribución en cm).

3. Estimación de **especies** (con o sin vejiga natatoria).

La boya M3iGO es la primera que incorpora inteligencia artificial (tras un programa de desarrollo de cuatro años 2017-2021) y la única boya que posee sondas bifrequencia. La inclusión de las dos frecuencias en la boya fue el resultado de un trabajo del equipo de I+D con la ayuda de clientes que probaron y compartieron resultados con Marine Instruments con el fin de obtener los datos más precisos y fiables.

Tras el análisis del sondeo en la doble frecuencia, concluyeron que dado que las distintas especies producen diferentes ecos dependiendo de sus cualidades morfológicas, con esta doble información, el patrón podría interpretar el tipo de especie y su posible distribución de tamaños, a través de la forma, intensidad y degradación del eco que se muestre en pantalla.

Así, las frecuencias más bajas son más indicadas para la detección de especies con vejiga natatoria como es el caso del atún patudo o el atún blanco, y por el contrario, las más altas se adecuan más a las especies sin vejiga natatoria como el bonito. Esta distinción de especies y tamaños supone una gran innovación en el producto y en el sector ya que mejora notablemente la eficacia de la pesca permitiendo una pesca más selectiva y sostenible, a la vez que ahorra combustible.

1.3.2. Proyecto: TUNAMONITOR (2014-2015). Cofinanciado por CDTI

Producto comercial: MarineObserve

El objetivo del proyecto fue desarrollar nuevas tecnologías para la monitorización y control de la pesca de atún para grandes barcos atuneros, mediante:

- El desarrollo de algoritmos de transmisión y almacenamiento cifrado de imágenes (1 imagen por segundo de cada cámara) en tiempo real, asignando metadatos encriptados (posición GPS, velocidad, etc.) de forma a cada fotograma individual.
- Desarrollo de algoritmos de inicio de monitorización y captación de imágenes en base a la velocidad y maniobras del barco y de otros sensores sí es necesario.
- Desarrollo de algoritmos de análisis y tratamiento automático de imágenes, para reducir el tiempo de análisis por parte de los observadores en tierra.
- Desarrollo conceptual de tecnologías de pesaje dinámico de las capturas, y métodos de transmisión e integración de datos.
- Desarrollo de envoltorios inviolables y seguros (*racks* y armarios, etc.)
- El desarrollo de una arquitectura segura e inviolable con hasta siete cámaras IP de alta resolución con objetivos de gran angular.

1.3.3. Proyecto: XDRONE (2016-2017). Co-financiado por CDTI

Producto comercial: Tunadrone

El proyecto consiste en el desarrollo de avión no tripulado, lanzable y recuperable desde un atunero, para la captación y envío de imágenes de alta resolución sobre bancos de túnidos.

Para ello, es necesario el desarrollo de distintos elementos:

- Plataforma aeronáutica estable, robusta y ágil capaz de volar a 30 nudos con un consumo máximo de 60W, con la integración de nuevos materiales y con un peso total menor que 4 kg.
- Electrónica embarcada con una integración de sensores, firmware y un piloto automático. Sistema de captación de imágenes con una cámara de alta resolución y un nuevo concepto de polarizador, algoritmos de compresión y envío de imágenes a larga distancia por radio (más de 20 millas náuticas).

- Desarrollo de un sistema de despegue y sistema de aterrizaje óptico autónomo específico para barcos en movimiento.
- Desarrollo de una estación operativa en el barco con un software para gestión y definición de misiones, gestión de lanzamiento y aterrizaje, visualización de imágenes y gestión del tracking (orientación) de la antena de comunicación con el avión.

1.3.4. Proyecto: ASF (2016-2017). Co-financiado por CDTI

Producto comercial: Marine Acoustic System

Como parte de la estrategia de diversificación de la empresa, el departamento de I+ D de Marine Instruments desarrolló este proyecto para el sector de la acuicultura del camarón.

El proyecto consiste en:

- Desarrollo de un alimentador automático con sistema autónomo de generación de energía, sistema de control, comunicación, geolocalización y sensores activos/pasivos (hidroacústicos, sensores de temperatura y oxígeno)
- Desarrollo de un módulo de energía en forma de paneles solares y con acumuladores de energía.
- Desarrollo de un módulo de comunicaciones, radio ISM 900 MHz, 10 km alcance), con posicionamiento GPS.
- Desarrollo e integración de sensores activos (sonares hidroacústicos) y pasivos (solo escuchar) (sonda hidrófono, sensores de temperatura y oxígeno) sensor acústico en tolva de alimento.
- Desarrollo de un algoritmo de cálculo que define cantidad y momento de alimentación, partiendo de datos de los sensores activos/pasivos.
- Desarrollo de un módulo de control para el sistema de alimentación
- Desarrollo de un software piloto de monitorización, para poder controlar varios sistemas de alimentación.

1.3.5. OCEANVIEW (2017-2018). Co-financiado por CDTI

Producto comercial: MarineView

En el marco de este proyecto se diseñaron nuevas tecnologías complementarias para la pesca, basadas en la visualización e interpretación de datos oceanográficos, con el fin de desarrollar nuevos productos y mantener el liderazgo de Marine en el sector.

El proyecto tenía dos líneas de investigación sobre tecnologías de información oceanográfica claramente definidas:

(a) Desarrollo de un nuevo tipo de interfaz físico HMI, para el control de la visualización de datos.

(b) Desarrollo de un sistema para gestionar y visualizar datos oceanográficos de varias fuentes.

Capítulo 2

Descripción del problema

Es sabido, desde hace años, que los atunes y otros peces tienden a reunirse alrededor de estructuras flotantes, como las plataformas petroleras, los arrecifes artificiales o, que es el caso que nos ocupa, las boyas. Las causas de ello son múltiples: no solo ofrecen un refugio seguro, sino que proporcionan sombra y protección contra los depredadores, y también atraen a pequeños organismos marinos que sirven como alimento para los peces. Concretamente, las boyas pueden ser un lugar de descanso para los peces después de una larga migración o de una búsqueda activa de alimento en aguas abiertas. Son, además, especialmente atractivas para ellos porque su movimiento y vibración pueden simular la presencia de otros peces, lo que aumenta las posibilidades de encontrar una pareja para reproducirse (Hunter & Mitchell 1967, Love et al 2009; Davies et al. 2014; Block et al. 2001; Josse et al. 2000, etc.).

Las boyas satelitales, como la ya mencionada M3iGO (Sección 1.3.1), se utilizan comúnmente en la industria pesquera para monitorizar los océanos e identificar bancos de peces, los cuales tienden a congregarse alrededor de estos objetos. M3iGO cuenta, entre muchas otras funcionalidades, con una *ecosonda*, que es el dispositivo que permite detectar la presencia de objetos en el fondo del océano. Al emitir ondas sonoras y medir el tiempo que tardan en rebotar, la sonda de la M3iGO puede determinar la distancia al fondo del océano e identificar bancos de peces. Esta información es valiosa para los pescadores, ya que les permite localizar la ubicación de estos bancos y determinar la profundidad a la que se encuentran, lo que les ayuda a planificar sus actividades de pesca y maximizar su eficiencia. A este tipo de pesca se le conoce como *pesca con FAD*.

Figura 2.1: *Pesca con FAD*

2.1. Proceso de calibrado

La calibración de la boya M3iGO, es decir, el ajuste de sus parámetros físicos, es un proceso fundamental para que pueda realizar correctamente sus funciones. A continuación, pasamos a describir someramente el proceso de calibración:

1º - **Test de la placa PCB en una mesa de programación**.

Una PCB (*Placa de Circuito Impreso*) es el elemento que dará todo el soporte electrónico a la boya. Esta manda una señal eléctrica, y se testea la potencia que se transmite a una resistencia, que se corresponde con el parámetro *Valor_trans_con_carga_50KHz/200 KHz*, y la potencia transmitida en vacío. Por otra parte, se mide la linealidad de la recepción, a través de la envolvente de una señal con amplitud creciente (parámetros *Linealidad_etapa_recep_50KHz/200KHz*). La linealidad es crítica para asegurar la

transmisión y recepción precisa de señales de comunicación. Finalmente, se analiza la recepción con señal constante, donde se manda un señal constante en el tiempo y se estudia la tensión media recibida. Este valor viene dado por los parámetros *RX 50KHz_cte_media* y *RX 200KHz_cte_media*. También se registra la desviación estándar, que corresponde a los parámetros *RX 50KHz_cte_dev* y *RX 200KHz_cte_dev*.

2º - **Ensamblado del transductor piezoeléctrico.**

El transductor piezoeléctrico es el encargado de convertir las señales eléctricas en acústicas y viceversa. Tiene tres proveedores distintos, los cuales vienen recogidos en el dato *Tipo_sonda*, y a menudo corresponde a un lote de fabricación concreto, que es lo que indica el dato *Lote_Sonda*. En esta fase del proceso, se le reviste de corcho y se suelda al cable.

3º - **Test de esfuerzo del transductor en montaje.**

A continuación, se mide la **frecuencia de resonancia**, tanto radial como axial, del piezo. Estos parámetros corresponden a los datos *Resonancia_baja_frecuencia* y *Resonancia_alta_frecuencia*, respectivamente.

4º - **Resinado y fresado del piezo.**

Para proteger a la sonda de la salinidad del océano, se reviste con una capa de resina epoxi. Esta resina puede ser de dos tipos: SND-CAST(FLEXIBLE) o SND-CAST(P-FLEXIBLE). El tipo de resinado corresponde al dato *Tipo_Resina*. Es necesario mencionar que la sonda lleva un proceso de curación, el cual depende del calendario laboral. El tiempo de curación lo indica el parámetro *Tiempo_curado*, el cual crearemos a partir del dato *Fecha_test_ok* (véase Sección 4.8). Posteriormente, tiene lugar el fresado del transductor.

5º - **Test del transductor en agua (minicuba)**.

El procedimiento consiste en unir la PCB con el piezo, sumergirlo en la cuba, y cuando el transformador de la PCB (el cual es de tres tipos y viene recogido en el dato *Transformador_PCB*) manda un impulso eléctrico y el transductor lo transforma en señal acústica, se coloca un hidrófono (micrófono subacuático) al fondo de la minicuba que es capaz de recoger el voltaje

de transmisión TX recibido por él (a 50 y 200 KHz). De igual manera, el hidrófono manda impulsos acústicos, el transductor los convierte en pulsos eléctricos y así se mide la tensión de recepción Rx (a 50 y 200 KHz). Estos parámetros se corresponden con los datos *Media_TX_50KHz_minicuba, Media_RX_50KHz_minicuba, Media_TX_200KHz_minicuba* y *Media_RX_200KHz_minicuba*. La minicuba está llena de agua con una cierta temperatura, aunque no incorporamos este dato de temperatura ya que al parecer existe un error en los sensores de medición.

6º - **Pegado del transductor en base.**

El transductor se pega a la base de la boya. La base, a su vez, es inyectada sobre un molde, el cual varía según el fabricante. En nuestro caso, existen dos posibles, y son indicados por el dato *Tipo_base*.

7º - **Calibrado en cuba**.

Finalmente, tiene lugar el calibrado en cuba. Es importante recalcar que, dependiendo de la cuba, los resultados pueden variar ya que son tres modelos distintos. El tipo de modelo viene dado por el dato *Equipo_cuba*. En esta fase, se va a testar la placa, la base de la boya y el transductor. El procedimiento es similar al test en minicuba, con la diferencia que, esta vez, la potencia de transmisión TX es un valor objetivo, el cual está dentro de un rango y viene dado por el dato *Vpp_50KHz/200 KHz*. Este valor objetivo se corresponde con una señal cuya longitud de onda es un porcentaje de la longitud de onda de la señal del pulso enviado por el transformador de la PCB. Estos porcentajes vienen dados por los datos *Pn 50/200_KHz*, y son calculados aplicando el método de la bisección hasta alcanzar el Tx objetivo dentro de un intervalo de confianza. El número de iteraciones del método viene recogido por los datos *num_iteraciones_50KHz* y *num_iteraciones_200KHz*.

Por otro lado, se calibra la tensión de la señal de recepción tomando como referencia una ganancia objetivo, la cual es obtenida inyectándole una tensión adicional, llamada *tensión de calibración*, mediante un microcontrolador. La tensión de calibrado está indicada por los datos *tens_calibrado_50KHz* y *tens_calibrado_200KHz*.

En este caso, se ha medido correctamente la temperatura del agua en

cuba, que aparece en el dato *Temperatura*.

Recapitulando, contamos con los siguientes parámetros de calibrado:

Tipo_sonda, Lote_Sonda, Tipo_Resina, Tipo_base, Transformador_PCB, Equipo_cuba, Vpp_50KHz, Vpp_200KHz, num_iteraciones_50KHz, num_iteraciones_200KHz, Temperatura, Resonancia_baja_frecuencia, Resonancia_alta_frecuencia, Valor_trans_con_carga_50KHz, Valor_trans_con_carga_200KHz, Linealidad_etapa_recep_50KHz, Linealidad_etapa_recep_200KHz, Media_TX_50KHz_minicuba, Media_RX_50KHz_minicuba, Media_TX_200KHz_minicuba, Media_RX_200KHz_minicuba, RX 50KHz_cte_media, RX 50KHz_cte_dev, RX 200KHz_cte_media, RX 200KHz_cte_de y Tiempo_curado.

Además, contamos con *Production_Version* (versión de producción del transformador), y con *Fecha_test_ok*, que nos servirá para calcular el dato de *Tiempo_curado*.

Se puede ver un extracto de la tabla donde figuran los datos de las cinco primeras boyas en la Sección 4.1.

El proceso de calibrado descrito presenta un desafío: los factores de calibración no son completamente deterministas y se ven afectados por cierta aleatoriedad. Esto puede generar ocasiones en las que algunos parámetros tomen valores considerados "inesperados" o "demasiado desviados de la media". Para abordar esta problemática, se busca predecir ciertos factores de calibrado, llamados de respuesta o salida, en función de otros parámetros de entrada. Asimismo, se busca clasificar los datos de calibrado como "normales" o "anómalos" en función de si sus desvíos de la media son esperados o no, teniendo en cuenta los datos históricos. Para lograr ambos objetivos debemos, en el primer caso, encontrar una función que pueda hacer predicciones precisas (problema de *regresión*) y, en el segundo, clasificar información en base a patrones detectados en los datos (problema de *clasificación*). Ambos problemas forman parte del campo del **machine learning**.

Capítulo 3

Modelos y Algoritmos

El *machine learning* (aprendizaje automático) es una rama de la inteligencia artificial que se centra en el desarrollo de algoritmos y modelos que permiten a los ordenadores aprender de los datos y realizar tareas específicas sin ser programados explícitamente para ello. En otras palabras, permite a los ordenadores "aprender" a partir de datos y experiencias, y utilizar ese conocimiento para tomar decisiones y realizar predicciones.

En el contexto de un modelo de *machine learning*, los **parámetros de entrada** (*features* o *inputs*) son las variables que se utilizan como entrada para el modelo. Estas variables se seleccionan cuidadosamente en función del problema que se esté abordando y pueden incluir información como valores numéricos, categorías, texto, imágenes, audio, entre otros.

Nuestros *features* corresponden a los parámetros de calibrado siguientes: *Production_Version, Tipo_sonda, Lote_Sonda, Tipo_Resina, Tipo_base, Transformador_PCB, Equipo_cuba, Vpp_50KHz, Vpp_200KHz, num_iteraciones_50KHz, num_iteraciones_200KHz, Temperatura, Resonancia_baja_frecuencia, Resonancia_alta_frecuencia, Valor_trans_con_carga_50KHz, Valor_trans_con_carga_200KHz, Linealidad_etapa_recep_50KHz, Linealidad_etapa_recep_200KHz, Media_TX_50KHz_minicuba, Media_RX_50KHz_minicuba, Media_TX_200KHz_minicuba, Media_RX_200KHz_minicuba, RX 50KHz_cte_media, RX 50KHz_cte_dev, RX 200KHz_cte_media, RX 200KHz_cte_dev y Tiempo_curado*. En total son 27.

Por otro lado, los **parámetros de salida** (*targets* o *outputs*) son las variables que se intentan predecir o clasificar a través del modelo de *machine learning*. Los *targets* pueden ser valores numéricos continuos en el caso de la regresión, o pueden ser etiquetas de clase discretas en el caso de la clasificación.

Nuestros *targets* son: *Pn_50KHz, Pn_200KHz, tens_calibrado_50KHz* y *tens_calibrado_200KHz*. En total son 4.

Durante todo este trabajo, estudiaremos y aplicaremos solo los modelos **supervisados**. Este tipo de modelos se entrenan utilizando un conjunto de datos que incluye tanto las variables de entrada como las de salida esperadas, y se aplican cuando queremos predecir una variable de salida a partir de una o más variables de entrada. El objetivo del entrenamiento es ajustar el modelo para que pueda hacer predicciones precisas basadas en nuevos datos que este no ha visto, y con ello *generalizar* los datos que puede predecir.

Existen otro tipo de modelos, llamados **no supervisados**, que se utilizan para encontrar patrones y relaciones en datos que no tienen variables de salida conocidas, los cuales no utilizaremos.

3.1. Clasificación y regresión

La clasificación y la regresión son dos técnicas fundamentales en el campo del *machine learning* supervisado. La regresión se utiliza para predecir valores continuos, es decir, una variable de salida que puede tomar cualquier valor dentro de un rango determinado. Este será el primer problema que trataremos, en donde buscaremos aproximarnos lo máximo posible a los valores numéricos de los *targets*.

Por otro lado, la clasificación se utiliza para predecir valores discretos, es decir, una variable de salida que toma un número limitado de valores posible. Los modelos de clasificación buscan encontrar patrones en los datos que permitan distinguir entre diferentes categorías o clases de la variable de salida. Este será el segundo problema que trataremos, el cual consistirá en predecir si se espera que los valores de salida se desvíen de su media una desviación estándar (clase YES) o no (clase NO).

En cuanto a las diferencias entre la regresión y la clasificación, además de las diferencias en los tipos de variables de salida que se pueden predecir, se encuentra el hecho de que la regresión busca encontrar una función continua que modele la relación entre las variables de entrada y de salida, mientras que la clasificación busca encontrar límites o fronteras de decisión que permitan separar las diferentes clases de la variable de salida.

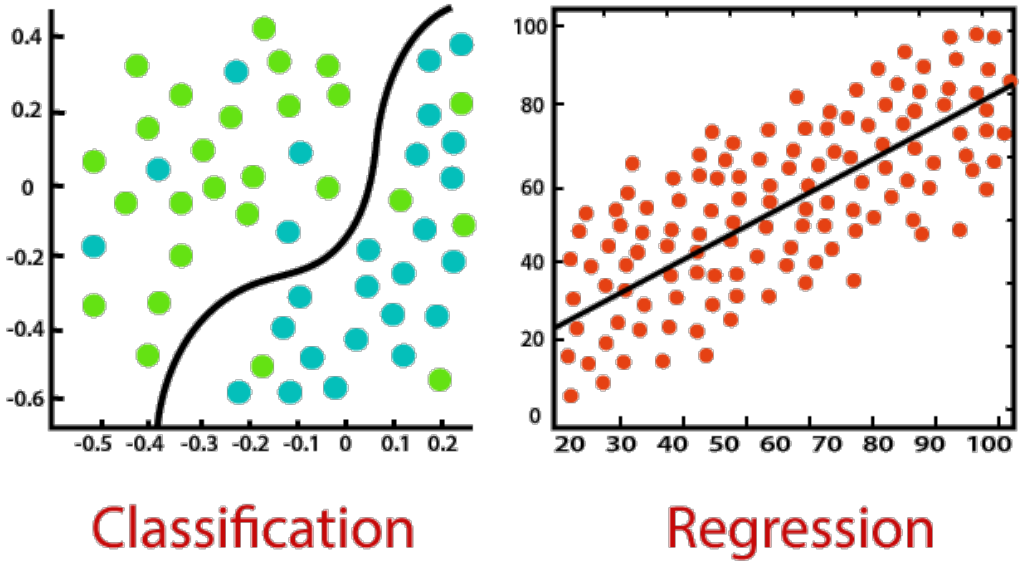


Figura 3.1: Clasificación vs regresión. Fuente: https://blogdatlas.wordpress.com/2020/06/28/algoritmos-supervisados-clasificacionvs-regresion-datlas-research/

3.2. *Under y overfitting*

El *underfitting* y el *overfitting* son dos problemas comunes en el *machine learning* y la estadística que se producen cuando un modelo no se ajusta adecuadamente a los datos de entrenamiento y no puede generalizar bien a nuevos datos.

Underfitting (subajuste) se produce cuando el modelo es demasiado simple para capturar la complejidad de los datos. Como resultado, el modelo

tiene un alto sesgo (bias) y una baja varianza. Es decir, el modelo subestima la verdadera relación entre las variables de entrada y salida, lo que resulta en un ajuste deficiente a los datos de entrenamiento y una pobre capacidad de generalización a nuevos datos. El underfitting puede solucionarse aumentando la complejidad del modelo o añadiendo más variables de entrada.

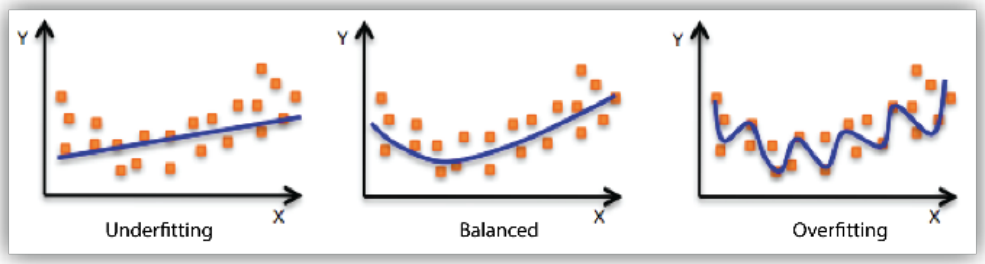


Figura 3.2: Modelo subajustado (izquierda), correcto (centro) y sobreajustado (derecha). Fuente: *Amazon Machine Learning Developer Guide*.

Por otro lado, se produce *overfitting* (sobreajuste) cuando el modelo es demasiado complejo y se ajusta demasiado a los datos de entrenamiento, incluso al ruido o las irregularidades de los datos. Como resultado, el modelo tiene una baja sesgo (bias) y una alta varianza. Es decir, el modelo ajusta demasiado bien a los datos de entrenamiento y no generaliza bien a nuevos datos. El *overfitting* puede solucionarse mediante la reducción de la complejidad del modelo, utilizando técnicas de regularización, o mediante la adición de más datos de entrenamiento.

3.3. *Bias-variance trade-off*

El compromiso (*trade-off*) entre sesgo (*bias*) y varianza (*variance*) es otro concepto clave en el *machine learning* y la estadística. Se refiere al equilibrio que debe encontrarse al construir un modelo que tenga un bajo bias (un bajo error entre los datos predichos y los reales) y una baja varianza (baja variabilidad en la estimación de los parámetros).

El *sesgo* hace referencia a la tendencia del modelo a hacer suposiciones simplificadoras sobre la relación entre las variables de entrada y salida. Un

modelo con un alto sesgo puede subestimar o sobrestimar la verdadera relación entre las variables, lo que resulta en un ajuste deficiente a los datos de entrenamiento. Se calcula como el valor esperado del parámetro estimado de salida menos el parámetro real:

$$bias = E(\hat{\theta}) - \theta. \tag{3.1}$$

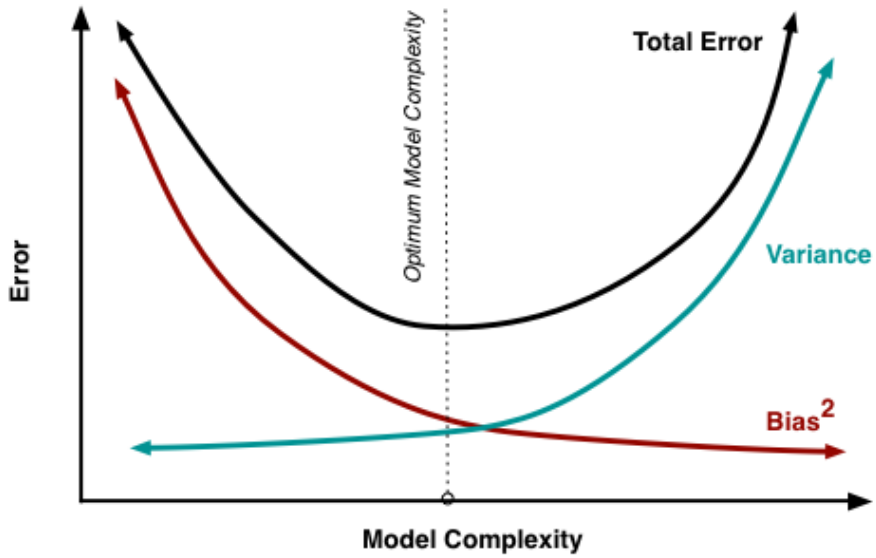


Figura 3.3: Gráfica del bias, la varianza y el MSE (http://scott.fortmann-roe.com/docs/BiasVariance.html).

La *varianza* se refiere a la sensibilidad del modelo a pequeñas variaciones en los datos de entrenamiento. Un modelo con alta varianza puede ajustarse demasiado (*overfitting*) a los datos de entrenamiento y no generalizar bien a nuevos datos. Se calcula como una varianza usual aplicada al parámetro estimado:

$$varianza = E\left[(\hat{\theta} - E(\hat{\theta}))^2\right] \tag{3.2}$$

Con eso, es fácil de ver que el MSE (*Mean Squared Error*)

$$MSE(\theta) = E\left[(\theta - \hat{\theta})^2\right] = E\left[(\theta - \hat{\theta} - E(\hat{\theta}) + E(\hat{\theta}))^2\right] \tag{3.3}$$

es la suma del bias al cuadrado más la varianza. Para ello, expandimos los sumandos del segundo miembro de (3.3), utilizamos que el parámetro real θ es constante y que la esperanza de una esperanza es igual a la esperanza original, con lo que nos queda:

$$MSE(\theta) = E\left[(\hat{\theta} - E(\hat{\theta}))^2\right] + E[(E(\hat{\theta}) - \theta)^2] = varianza(\hat{\theta}) + bias(\theta, \hat{\theta})^2. \tag{3.4}$$

Por lo tanto, el compromiso entre sesgo y varianza implica encontrar un modelo que tenga un bajo sesgo y una baja varianza. Un modelo con un bajo sesgo y alta varianza puede mejorarse aumentando el tamaño del conjunto de entrenamiento o utilizando técnicas de regularización. Por otro lado, un modelo con alto sesgo y baja varianza puede mejorarse utilizando un modelo más complejo o agregando más variables de entrada (por ejemplo, Hastie et al. 2009).

Nuestro objetivo consistirá en encontrar un equilibrio entre el sesgo y la varianza que resulte en un modelo que tenga la menor varianza y sesgo posible.

A continuación, describiremos en detalle los modelos de *machine learning* que utilizaremos a lo largo de este trabajo.

3.4. Red neuronal

Una red neuronal es un modelo inspirado en el funcionamiento del cerebro humano y es muy popular en el campo del aprendizaje automático. Está compuesta por una serie de nodos o neuronas interconectados que procesan información mediante un número de capas, donde cada una de ellas procesa dicha información en diferentes etapas (*epochs*). La primera capa recibe las variables de entrada, y las capas siguientes se encargan de realizar operaciones cada vez más complejas, que se suelen denominar *capas ocultas*. La última se denomina *capa de salida* y se corresponden con los *targets* predichos.

En una red neuronal, cada conexión entre dos neuronas tiene asociado un *peso* que determina la fuerza y dirección de la influencia de la neurona de entrada sobre la neurona de salida. Los pesos son valores numéricos que se

asignan a las conexiones entre las neuronas, y pueden ser positivos o negativos. Un peso positivo indica que la neurona de entrada aumenta la activación de la neurona de salida, mientras que un peso negativo indica que la neurona de entrada disminuye la activación de la neurona de salida. Durante el entrenamiento de la red neuronal, los pesos son ajustados para minimizar una función de pérdida y mejorar la precisión de la red.

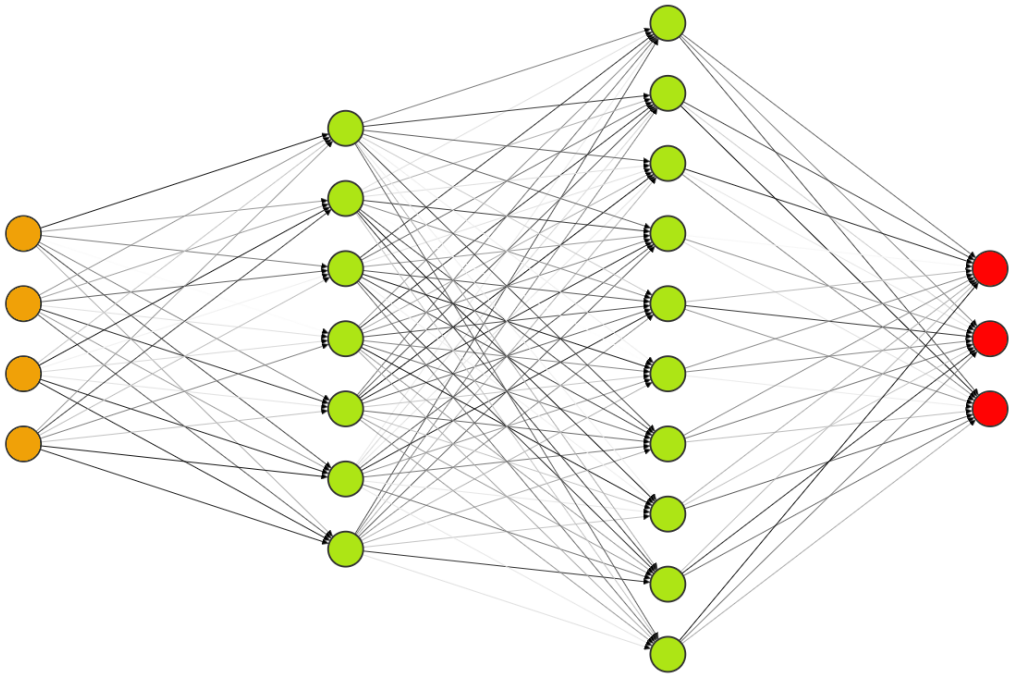


Figura 3.4: Diagrama de una red neuronal con 2 capas ocultas. Los nodos naranja pertenecen a la capa de entrada y los rojos, a la capa de salida. Las líneas más tenues representan pesos más cercanos a 0.

Además de los pesos, cada neurona en una red neuronal tiene un valor asociado llamado *bias*[1] (sesgo). El sesgo es un valor numérico que se suma a la suma ponderada de las entradas de la neurona, antes de aplicar una **función de activación**, y tiene como propósito mejorar la capacidad de la red para ajustarse a los datos.

[1]No confundir con el *bias* de la Sección 3.3.

3.4.1. Arquitectura de la red neuronal

La red neuronal se puede pensar como una función $f(x)$, donde la variable independiente son los *inputs* y el resultado que devuelve, la predicción de los *outputs*. Esta función se construye de manera recursiva:

$$z^{[l]} = W^{[l]}a^{[l-1]} + b^{[l]} \in \mathbb{R}^{n_l},\ l = 2, 3, .., L, \tag{3.5}$$

donde L es en número de capas, $W^{[l]}$ y $b^{[l]}$ son los pesos y bias correspondientes a la capa l, respectivamente, y n_l es el número de neuronas de la capa l. $z^{[l]}$ se denomina la suma ponderada de las neuronas en la capa l. Para la capa de entrada ($l = 1$), las neuronas $a^{[1]}$ son iguales al vector de *inputs* de nuestro modelo.

σ, a su vez, es la función de activación que se aplica a la suma ponderada de las neuronas de la capa $l - 1$ para producir las neuronas de la capa l. Esta función es importante porque le da a la red neuronal la capacidad de modelar relaciones no lineales entre los *inputs* y *outputs*:

$$a^{[l]} = \sigma(z^{[l]}). \tag{3.6}$$

Existen varios tipos de funciones de activación utilizadas en redes neuronales, como la función sigmoide, la tangente hiperbólica, la identidad, o la llamada ReLU (*Rectified Linear Unit*). Esta última es una de las más populares utilizadas en redes neuronales. Se define como $f(x) = \text{máx}(x, 0)$.

La elección de la función de activación depende, en general, del problema que se está resolviendo y del tipo de red neuronal que se está utilizando. En nuestras simulaciones, probaremos con las funciones identidad, sigmoide y ReLU, y seleccionaremos la que mejores resultados proporcione en cada caso.

3.4.2. Algoritmo de retropropagación

El funcionamiento de las redes neuronales se basan en el **algoritmo de retropropagación** (*backpropagation*). Este tipo de algoritmos fueron desarrollados por primera vez en la década de los setenta, y se basaron sobre todo en técnicas de optimización y control (Werbos 1974; Bryson 1975), pero no sería hasta los ochenta cuando verdaderamente se popularizaron (Rumelhart

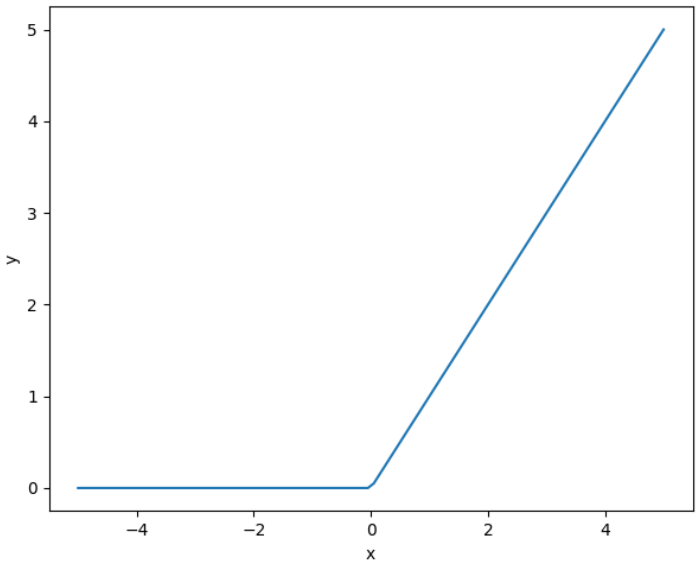


Figura 3.5: Función de activación ReLU

et al. 1985, 1986).

El objetivo primordial de las redes neuronales es ajustar los pesos y bias para minimizar el siguiente funcional:

$$C(W,b) = \frac{1}{N}\sum_{i=1}^{N} C_{x_i}(W,b), \tag{3.7}$$

donde N es el número de datos con los que vamos a entrenar la red y

$$C_{x_i}(W,b) = \frac{1}{2}||y(x^{\{i\}}) - a^{[L]}||_2^2, \tag{3.8}$$

para cada dato de entrenamiento x_i. y hace referencia al *output* real correspondiente a x_i.

Lema 1. *Si definimos*

$$\lambda_j^{[l]} := \frac{\partial C}{\partial z_j^{[l]}},\ 2 \leq l \leq L,\ 1 \leq j \leq n_l,$$

entonces se tiene

$$\lambda^{[L]} = \sigma'(a^{[L]} - y) \tag{3.9a}$$

$$\lambda^{[l]} = \sigma'(W^{[l+1]})\lambda^{[l+1]} \qquad 2 \leq l \leq L-1 \tag{3.9b}$$

$$\frac{\partial C}{\partial b_j^{[l]}} = \lambda_j^{[l]} \qquad 2 \leq l \leq L \tag{3.9c}$$

$$\frac{\partial C}{\partial W_{jk}^{[l]}} = \lambda_j^{[l]} a_k^{[l-1]}, \qquad 2 \leq l \leq L \tag{3.9d}$$

donde j y k hacen referencia a los índices de las filas y columnas de los pesos, respectivamente. C se refiere a C_{x_i}. Esta propiedad recursiva "hacia atrás" es lo que le da el nombre al algoritmo.

Demostración. Por (3.6), está claro que

$$\frac{\partial a_j^{[L]}}{\partial z_j^{[L]}} = \sigma'(z_j^{[L]}). \tag{3.10}$$

Derivando el primer miembro de (3.8) con respecto a a_j:

$$\frac{\partial C}{\partial a_j^{[L]}} = \frac{\partial}{\partial a_j^{[L]}} \frac{1}{2} \sum_{k=1}^{n_L} \left(y_k - a_k^{[L]}\right)^2 = -\left(y_j - a_j^{[L]}\right). \tag{3.11}$$

Usando la definición y la regla de la cadena:

$$\lambda_j^{[l]} = \frac{\partial C}{\partial a_j^{[L]}} \frac{\partial a_j^{[L]}}{\partial z_j^{[L]}} = \sigma'(z_j^{[L]}) \left(y_j - a_j^{[L]}\right), \tag{3.12}$$

con lo que tenemos probado (3.9a). Para la segunda, de nuevo, aplicamos la regla de la cadena y la definición de $\lambda_j^{[l]}$:

$$\lambda_j^{[l]} = \sum_{k=1}^{n_{l+1}} \lambda_k^{[l+1]} \frac{\partial z_k^{[l+1]}}{\partial z_j^{[l]}}. \tag{3.13}$$

Por otra parte, de (3.10) y de la formulación componente a componente de (3.5) se llega a que

$$\lambda_j^{[l]} = \sigma'(z_j^{[l]}) \left(\sum_{k=1}^{n_{l+1}} W_{jk}^{[l+1]} \lambda_k^{[l+1]} \right) = \sigma'(z_j^{[l]}) \left((W^{[l+1]})^T \lambda^{[l+1]} \right)_j, \tag{3.14}$$

lo que prueba (3.9b). Para probar (3.9c), basta con aplicar la regla de la cadena en $\frac{\partial C}{\partial b_j^{[l]}}$ y observar que

$$\frac{\partial z_j^{[l]}}{\partial b_j^{[l]}} = 1, \tag{3.15}$$

puesto que $z^{[l-1]}$ no depende de $b_j^{[l]}$, por definición. Entonces,

$$\frac{\partial C}{\partial b_j^{[l]}} = \frac{\partial C}{\partial z_j^{[l]}}, \tag{3.16}$$

y hemos probado (3.9c). Finalmente, para probar (3.9d), derivamos $z_j^{[l]}$ con respecto a $W_{jk}^{[l]}$ y obtenemos:

$$\begin{aligned} \frac{\partial z_j^{[l]}}{\partial W_{jk}^{[l]}} &= a_k^{[l-1]}, \ \forall j \\ \frac{\partial z_s^{[l]}}{\partial W_{jk}^{[l]}} &= 0, \ s \neq j. \end{aligned} \tag{3.17}$$

Ahora, usando (3.17) y la regla de la cadena:

$$\frac{\partial C}{\partial W_{jk}^{[l]}} = \sum_{s=1}^{n_l} \frac{\partial C}{\partial z_s^{[l]}} \frac{\partial z_s^{[l]}}{\partial W_{jk}^{[l]}} = \frac{\partial C}{\partial z_j^{[l]}} a_k^{[l-1]} = \lambda_j^{[l]} a_k^{[l-1]}, \tag{3.18}$$

lo que completa la demostración (Higham & Higham 2019).

□

Se puede probar, además, que los $\lambda_j^{[l]}$ coinciden con el sistema adjunto del problema de control definido por minimizar el funcional en (3.7), sujeto a las restricciones

$$\begin{aligned} z^{[l]} - W^{[l]} a^{[l-1]} - b^{[l]} &= 0, \\ a^{[l]} - \sigma(z^{[l]}) &= 0, \end{aligned} \tag{3.19}$$

donde $W^{[l]}$ y $b^{[l]}$ son las variables de control y $a^{[l]}$ y $z^{[l]}$, las variables de estado, para $l \in \{2, ..., L\}$.

3.4.3. Algoritmo de gradiente estocástico (SGD)

Una vez hemos calculado los $\lambda_j^{[l]}$, podemos computar el gradiente de C, lo cual nos servirá para encontrar su mínimo utilizando el método de descenso. Para ello, simplemente tomando gradientes en (3.7), sigue trivialmente que

$$\nabla C = \frac{1}{N} \sum_{i=1}^{N} \nabla C_{x^{\{i\}}}. \tag{3.20}$$

Recordamos que N es el número de datos (filas) en nuestro *dataset*. En la práctica, es bien sabido que este gradiente es computacionalmente muy costoso de calcular, sobre todo cuando tenemos una cantidad enorme de datos, y por tanto se busca aproximar dicho gradiente mediante el denominado **algoritmo de gradiente estocástico**.

Este algoritmo se basa en la aproximación de la media de los gradientes en cada dato de entrenamiento por medio de la estimación de dicha media en un cierto subconjunto de datos. Aparece por primera vez en los años cincuenta (Robbins & Monro 1951), pero no sería hasta finales de los años setenta cuando Bradley Efron y sus colegas propondrían una variante del SGD llamada *bootstrapped subsampling*, que utiliza submuestras aleatorias del conjunto de datos para ajustar los parámetros del modelo (Efron 1979). Esta variante del algoritmo resulta mucho más eficiente y efectiva, y será la que describiremos a continuación:

1 - Elegimos m enteros, $\{k_i\}_{i=1}^{m}$, elegidos aleatoriamente del conjunto $\{1, 2..., N\}$.

2 - Actualizamos los pesos y bias de la siguiente manera:

$$\begin{aligned} W_{jk}^{[l]} &= W_{jk}^{[l]} - \frac{1}{m} \sum_{i=1}^{m} \frac{\partial C_{x^{\{k_i\}}}}{\partial W_{jk}^{[l]}}, \\ b_{j}^{[l]} &= b_{j}^{[l]} - \frac{1}{m} \sum_{i=1}^{m} \frac{\partial C_{x^{\{k_i\}}}}{\partial b_{j}^{[l]}}, \end{aligned} \tag{3.21}$$

con $l \in \{2, ..., L\}$, y j y k recorren las dimensiones de los pesos y bias, respectivamente. Nótese que las parciales en (3.21) son conocidas gracias al Lema 1.

El conjunto $\{x^{\{k_i\}}\}_{i=1}^{n}$ se denomina mini-lote (*mini-batch*), y tenemos que recorrer el conjunto de muestras recorriendo cada mini-lote. A su vez, tenemos que recorrer todos los mini-lotes, los cuales debemos generar particionando de manera aleatoria la muestra de datos.

Las ecuaciones de (3.21) se modifican añadiendo la **tasa de aprendizaje** η (*learning rate*), un valor entre 0 y 1 que permite una mejor aproximación de los pesos y bias óptimos:

$$\begin{aligned} W_{jk}^{[l]} &= W_{jk}^{[l]} - \eta \frac{1}{m} \sum_{i=1}^{m} \frac{\partial C_{x^{\{k_i\}}}}{\partial W_{jk}^{[l]}}, \\ b_{j}^{[l]} &= b_{j}^{[l]} - \eta \frac{1}{m} \sum_{i=1}^{m} \frac{\partial C_{x^{\{k_i\}}}}{\partial b_{j}^{[l]}}. \end{aligned} \tag{3.22}$$

3.4.4. Algoritmo de Kingma-Diedrik-Ba (Adam)

El algoritmo Adam modifica el SGD agregando dos elementos nuevos: los momentos de primer y segundo orden. Estos momentos se calculan a partir de los gradientes que ya hemos visto y se utilizan para actualizar la tasa de aprendizaje. Es decir, en lugar de utilizar una tasa de aprendizaje fija, como en el SGD, Adam utiliza una tasa de aprendizaje adaptativa que varía para cada parámetro del modelo. Esto ayuda a reducir la sensibilidad del algoritmo a la elección de una tasa de aprendizaje fija y a evitar que se atasque en mínimos locales. Para más detalles sobre la implementación del algoritmo, véase Kingma & Ba (2014).

3.5. Árbol de decisión

Los árboles de decisión son otro tipo de modelo que se utilizan para realizar predicciones basadas en una serie de reglas o criterios. En un árbol de decisión, cada nodo representa una pregunta o una condición sobre un *input*, y cada rama representa una posible respuesta a esa pregunta. Al llegar a una hoja, se toma una decisión basada en la respuesta final.

Los árboles de decisión tienen varias ventajas en comparación con otros modelos de *machine learning*. Una de ellas es que son fáciles de entender e

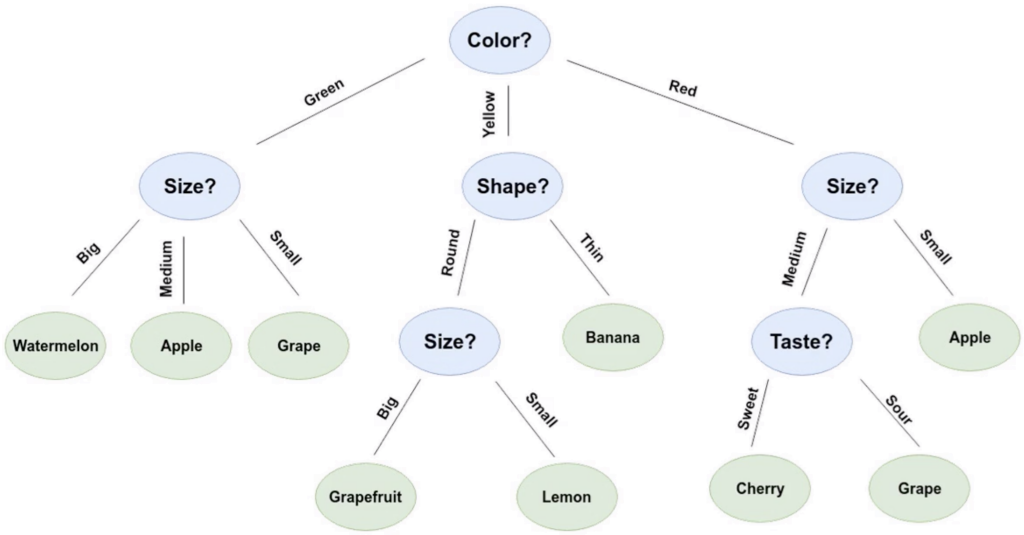


Figura 3.6: Ejemplo de árbol de decisión para un problema de clasificación. Se clasifica una categoría de fruta (variable de salida) en base a otras de entrada (color, tamaño, forma y sabor). Crédito a LinkedIn.

interpretar, lo que los hace útiles para explicar y comunicar los resultados a los usuarios finales (Breiman et al. 1984). Además, suelen ser especialmente rápidos en el entrenamiento y la predicción, lo que los hace ideales para problemas en tiempo real o grandes conjuntos de datos (Loh 2011).

No obstante, los árboles de decisión también cuentan con grandes desventajas que deben tenerse en cuenta. Las dos más importantes son, probablemente, la **tendencia al *overfitting***, la cual suele ser consustancial al método y generalmente causa una peor generalización en los datos de test, y **la sensibilidad a pequeñas variaciones** en los datos de entrenamiento, que puede conducir a la obtención de árboles muy diferentes y a veces inconsistentes para conjuntos de datos similares (Breiman et al. 1984).

3.5.1. Algoritmo CART

El algoritmo CART (*Classification and Regression Trees*) es un algoritmo de clasificación y regresión basado en árboles de decisión. Fue desarrollado por Breiman y sus colegas en los años ochenta y es uno de los algoritmos

más populares y ampliamente utilizados en la construcción de modelos de árboles de decisión. Se basa en la división (*split*) recursiva de los datos de entrenamiento en subconjuntos cada vez más pequeños y homogéneos en términos de la variable de salida que se desea predecir. A continuación, procedemos a describir el algoritmo CART en versión optimizada, basándonos en su implementación por la librería de Python sklearn (https://scikit-learn.org/stable/modules/tree.html#mathematical-formulation):

1 - Elige un nodo del árbol. Supongamos que corresponde al nodo m, el cual contiene n_m datos agrupados en dicho nodo. Este grupo de datos lo denotamos por Q_m.

2 - Para cada *feature* j y valor del *feature* t_j^m, se puede particionar Q_m en los siguientes conjuntos:

$$\begin{aligned} Q_m^i(j, t_j^m) &= \{(x, y)/x_j \leq t_j^m\}, \\ Q_m^d(j, t_j^m) &= Q^m \setminus Q_m^i(j, t_j^m), \end{aligned} \tag{3.23}$$

siendo y los *targets* asociados a los *features*.

3 - Encontrar (j, t_j^m) que minimicen el funcional G definido por:

$$G(Q_m, j, t_j^m) = \frac{\#Q_m^i}{n_m} H(Q_m^i) + \frac{\#Q_m^d}{n_m} H(Q_m^d). \tag{3.24}$$

H se conoce como función de pérdida o impureza. En el caso de problemas de clasificación, se puede tomar como la función de impureza de Gini o como la función de entropía. En el caso de problemas de regresión, H suele tomarse como el error cuadrático medio (MSE).

4 - Iterar el proceso sobre los conjuntos Q_m^i y Q_m^d hasta que se llegue a un nodo hoja o hasta que se llegue a un número mínimo predeterminado de datos por nodo. Si se llega a este criterio de parada, se le asigna, en el problema de clasificación, una etiqueta de clase al nodo hoja basándose en la mayoría de las clases contenidas en dicho nodo. En un problema de regresión, se suele asignar el valor promedio de los valores de los *targets*.

Cabe recalcar que los árboles de decisión, *por sí solos*, no nos serán de utilidad puesto que tienden al *overfitting* de manera desmesurada, y por ello lo combinaremos con las técnicas que siguen para mejorarlos.

3.6. Métodos de ensamblado

El *método de ensamblado* es una técnica que combina múltiples modelos para mejorar la precisión de las predicciones. En lugar de depender de un solo modelo, se crean varios independientes, y sus predicciones se combinan para producir una predicción más precisa. El ensamblado de modelos ha demostrado ser efectivo en la mayoría de los problemas de *machine learning*, especialmente cuando los datos son ruidosos o la relación entre los *features* y los *targets* no es lineal (Géron 2017; Zhou 2012).

Los métodos de ensamblado que trataremos son el *bagging* y el *boosting*. El primero se basa en la creación de múltiples muestras aleatorias del conjunto de datos de entrenamiento y entrenar un modelo en cada muestra. Luego, se combinan las predicciones de los modelos para generar una predicción final. El segundo, se enfoca en mejorar el rendimiento del modelo en los datos más difíciles de clasificar. A continuación, pasamos a describir los modelos de ensamblado que utilizaremos en nuestro trabajo.

3.6.1. *Bagging. Random Forests.*

El *bagging* (*Bootstrap Aggregating*) es una técnica de ensamblado en donde cada uno de los modelos independientes que la conforman son entrenados con diferentes subconjuntos aleatorios de los datos de entrenamiento originales.

Los bosques aleatorios (*Random Forests*) son un tipo un algoritmo de ensamblado que utiliza múltiples árboles de decisión para hacer predicciones. Son una combinación de *bagging* y aleatorización, de manera que cada árbol se entrena en un subconjunto aleatorio del conjunto de datos de entrenamiento y del conjunto de *features*.

En el proceso de entrenamiento, se utiliza una técnica llamada "muestreo con reemplazo" para crear múltiples árboles de decisión, lo que permite una mayor diversidad en los modelos y una reducción en la correlación entre

ellos. A través de la combinación de los árboles de decisión, Random Forest es capaz de producir predicciones mucho mejores que los árboles de decisión, evitando el *overfitting* de estos últimos.

Durante la predicción, las predicciones de cada árbol se combinan mediante votación o promedio ponderado para producir una predicción final.

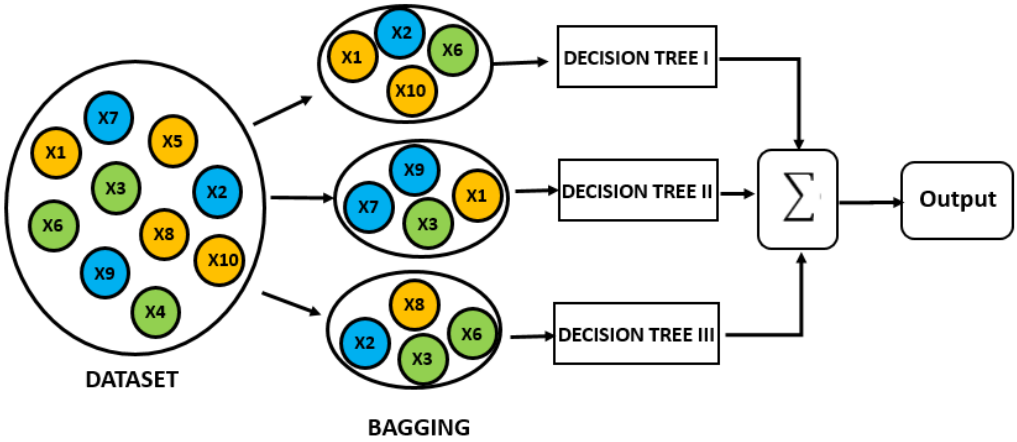


Figura 3.7: Esquema del *bagging* aplicado a los árboles de decisión.

La principal ventaja de los *Random Forests* es su capacidad para manejar conjuntos de datos grandes y complejos con alta dimensionalidad y ruido. Además, son mucho más resistentes al *overfitting* que los árboles de decisón, y tienen una buena capacidad de generalización. De hecho, las primeras publicaciones acerca de los bosques aleatorios sostenían la idea de que no sufren *overfitting* (por ejemplo, Breiman 2001). Sin embargo, la realidad es que no están exentos de este problema y, en algunas ocasiones, pueden sobreajustar los datos de entrenamiento (Hastie et al. 2009).

3.6.2. *Boosting*. Métodos de Gradient Boosting y *XGBoost*

Boosting es otra técnica de ensamblado de modelos. El objetivo del *boosting* es combinar varios modelos débiles, como árboles de decisión, para crear un modelo fuerte que tenga una precisión más alta y pueda generalizar mejor

a datos nuevos. Esto se hace de forma iterativa; de manera que, en cada iteración, se da más peso a los datos que fueron incorrectamente predichos por los modelos anteriores, lo que permite que los modelos posteriores se centren en los casos más difíciles. Con ello, se mejora la precisión general del modelo final.

El modelo resultante final $f(x)$ puede escribirse como

$$f(x) = f^{(M)}(x) = \sum_{m=1}^{M} \hat{\theta}_m(x), \tag{3.25}$$

donde $\hat{\theta}_m(x)$ representa la estimación del modelo débil que se implementa en la iteración m. $f^{(M)}$ denota el modelo fuerte tras M iteraciones, el cual coincide con el modelo final. Concretamente, en la iteración m, se actualiza el modelo fuerte de la siguiente manera:

$$f^{(m)}(x) = f^{(m-1)}(x) + \theta_m(x). \tag{3.26}$$

$f^{(0)}(x)$ viene dado por el modelo inicial.

El *boosting* se utiliza comúnmente en problemas de clasificación y regresión, y hay varios algoritmos de *boosting* populares, como AdaBoost, Gradient Boosting, y XGBoost. Nosotros utilizaremos los dos últimos en ambos problemas.

Gradient Boosting

El método de *Gradient Boosting* utiliza árboles de decisión como modelo débil, y funciona agregando iterativamente nuevos árboles al modelo para corregir los errores de la previa iteración. El método recibe su nombre debido a que utiliza el método de descenso de gradiente para minimizar la función de pérdida en cada iteración.

Algoritmo

Como es costumbre, el algoritmo se basa en la minimización de una función de pérdida $L(y_i, f(x_i))$, con $i \in \{1, ..., N\}$, siendo N el número de datos. En la iteración m, la mejor estimación del modelo fuerte viene dado por $f^{(m-1)}(x)$. Por tanto, para minimizar L mediante el método de descenso en

esta iteración, necesitamos calcular el gradiente de L con respecto a esta variable y evaluarlo en su estimación, es decir, en $f^{(m-1)}(x)$. Llamamos a este gradiente el pseudo-residuo $\hat{g}_m(x)$:

$$\hat{g}_m(x_i) = -\left[\frac{\partial L(y_i, f(x_i))}{\partial f(x_i)}\right]_{f(x)=f^{(m-1)}(x)} \tag{3.27}$$

En principio, se podría pensar en estimar θ_m(x) como el pseudo-residuo en la iteración m. No obstante, surge un problema: con este procedimiento, estamos minimizando el funcional en los puntos de entrenamiento x_i, por lo que nuestro algoritmo no tiene capacidad de generalización y corremos el riesgo de que el modelo sufra *overfitting*. Para ello, lo que se hace es entrenar el modelo débil de la iteración m con los *inputs* como variable de entrada y, los pseudo-residuos, como variable de salida. Así nuestra capacidad de generalizar los datos mediante el método de descenso aumentará.

El siguiente paso consiste en encontrar el paso óptimo para nuestra dirección de descenso. Para ello, necesitamos resolver el problema de optimización unidimensional siguiente:

$$\rho_m = \arg\min_{\rho} \sum_{i=1}^{N} L(y_i, f^{(m-1)}(x) + \rho h_m(x)), \tag{3.28}$$

siendo $h_m(x)$ la función predictora del modelo débil. Así pues, nuestro $\hat{\theta}_m(x)$ quedaría de la siguiente manera:

$$\hat{\theta}_m(x) = \eta \rho_m h_m, \tag{3.29}$$

donde hemos añadido el parámetro η, análogo a la tasa de aprendizaje en las redes neuronales.

Finalmente, calculamos nuestro modelo fuerte $f^{(m)}(x)$ con (3.26). Para $m = 0$, podemos suponer que la función predictora es una constante γ y tomar $f^{(0)}$ como

$$f^{(0)}(x) = \arg\min_{\gamma} \sum_{i=1}^{N} L(y_i, \gamma). \tag{3.30}$$

XGBoost

XGBoost (*Extreme Gradient Boosting*) es otro método de *boosting* que, igual que el *Gradient Boosting*, utiliza los árboles de decisión como modelo débil. La diferencia reside en que, en lugar del método de gradiente, este algoritmo implementa el **método de Newton-Raphson**, por lo que a veces se le conoce como Newton *boosting*. Fue creado por Tianqi Chen en 2014, y se ha vuelto muy popular en los últimos años debido a su alta precisión y velocidad de entrenamiento (Chen et al. 2015; Chen & Guestrin 2016).

Algoritmo

El algoritmo de XGBoost sigue un esquema análogo al de *Gradient Boosting*, aunque existen diferencias importantes. Por ejemplo, $\hat{\theta}_m(x)$ se calcula entrenando el árbol de decisión pero, esta vez, minimizando la siguiente función de pérdida:

$$\hat{\theta}_m(x) = \arg\min_{\phi\in\Phi} \sum_{i=1}^{N} \left[\hat{g}_m(x_i)\phi(x_i) + \frac{1}{2}\hat{h}_m(x_i)\phi(x_i)^2\right], \tag{3.31}$$

siendo

$$\hat{h}_m(x) = -\left[\frac{\partial^2 L(y_i, f(x_i))}{\partial f(x_i)}\right]_{f(x)=f^{(m-1)}(x)}. \tag{3.32}$$

Nótese que (3.31) surge de la expansión en serie de Taylor de segundo orden de la función de pérdida, centrada en el punto $f^{(m-1)}(x)$. En efecto:

$$\begin{aligned} &L(y_i, f^{(m-1)}(x_i) + \theta_m(x_i)) \approx L(y_i, f^{(m-1)}(x_i)) \\ &- \left[\frac{\partial L(y_i, f(x_i))}{\partial f(x_i)}\right]_{f(x)=f^{(m-1)}(x)} \theta_m - \frac{1}{2}\left[\frac{\partial^2 L(y_i, f(x_i))}{\partial f(x_i)}\right]_{f(x)=f^{(m-1)}(x)} \theta_m^2 = \\ &L(y_i, f^{(m-1)}(x_i)) + \hat{g}_m(x_i)\theta_m(x_i) + \frac{1}{2}\hat{h}_m(x_i)\theta_m(x_i)^2. \end{aligned}$$

Luego, haciendo el sumatorio de todos los i, y despreciando los términos que no dependen de θ_m, tenemos que

$$\hat{\theta}_m(x) = \arg\min_{\theta} \sum_{i=1}^{N} L(y_i, f^{(m-1)}(x_i) + \theta_m(x_i)) =$$
$$\arg\min_{\theta} \sum_{i=1}^{N} \left[\hat{g}_m(x_i)\theta_m(x_i) + \frac{1}{2}\hat{h}_m(x_i)\theta_m(x_i)^2 \right],$$

y obtenemos (3.31), tal y como queríamos probar.

XGBoost también utiliza una técnica llamada "regularización" para evitar el sobreajuste y mejorar la generalización del modelo.

3.7. Hiperparámetros de los modelos

Los hiperparámetros son variables que controlan el comportamiento de un determinado modelo y se establecen antes de entrenarlo. A diferencia de los parámetros, que se aprenden automáticamente a partir de los datos de entrenamiento, los hiperparámetros se establecen manualmente por el usuario y afectan directamente a su calidad de predicción.

Los hiperparámetros varían según el tipo de modelo y, por ejemplo, en el caso de las redes neuronales, se incluyen los siguientes:

1 - Tasa de aprendizaje (*learning rate*): Controla la velocidad a la que el modelo ajusta los pesos durante el entrenamiento.

2 - Tamaño del *minilote* (*minibatch size*): El número de ejemplos de entrenamiento que se utilizarán en cada iteración del entrenamiento.

3 - Número de capas: El número de capas de la red neuronal.

4 - Número de neuronas por capa: El número de neuronas en cada capa de la red neuronal.

5 - Función de activación: La función matemática utilizada para calcular la salida de cada neurona, como por ejemplo, la ya mencionada ReLU.

6 - Parámetro de regularización α: un parámetro utilizado para evitar el *overfitting* y mejorar la generalización del modelo.

7 - Optimizador. El algoritmo utilizado para ajustar los pesos de la red neuronal durante el entrenamiento. En nuestro caso, será el Adam.

Cabe decir que los hiperparámetros que podemos ajustar dependerán de la librería de *machine learning* que utilicemos. En la Sección 5.3.3, hablaremos del conjunto de hiperparámetros seleccionados para cada modelo.

Es importante ajustar adecuadamente los hiperparámetros para obtener el mejor rendimiento posible. Sin embargo, encontrar los valores óptimos puede requerir mucho tiempo y recursos de computación. En nuestro caso, buscaremos el valor óptimo de los hiperparámetros determinando un array de posibles valores para cada hiperparámetro y evaluando la precisión del modelo para toda combinación de estos arrays, lo que se conoce como "búsqueda de mallado" (*Grid Search*). Hablaremos más de esta técnica en los Capítulos siguientes.

Capítulo 4

Análisis de los datos

Analizar los datos que disponemos es esencial para hallar modelos de *machine learning* predictivos, puesto que ayuda a comprender mejor los patrones y las relaciones presentes en ellos y, por tanto, a crear modelos más precisos y confiables.

El análisis exploratorio de datos (AED) es un enfoque para analizar y resumir un conjunto de datos de manera descriptiva y visual. El objetivo principal del AED es descubrir patrones, relaciones, anomalías y tendencias en los datos, así como identificar cualquier problema o error en ellos. Suele tener las siguientes fases (por ejemplo, Tukey 1977):

1 - **Recopilación de datos**: El primer paso en el AED es recopilar los datos relevantes para el análisis.

2 - **Limpieza de datos**. Se eliminan las filas de datos que contengan algún *input* que falte o que tenga un valor atípico.

3 - **Descripción de datos**. A menudo suele ser necesario realizar estadísticas descriptivas y gráficos para comprender mejor la distribución de los datos y las relaciones entre las variables. Esto puede incluir la creación de histogramas o *boxplots*.

4 - **Identificación de patrones**: En esta fase, se utilizan técnicas de análisis estadístico y visualización para identificar patrones y relaciones en los datos. Esto puede incluir el análisis de correlación o la visualización de datos.

5 - **Transformación de datos**: Se convierten los datos originales en formatos más adecuados para su uso en un modelo de *machine learning*.

6 - **Creación de datos**: se crean nuevos *inputs* a partir de los que ya disponemos para aportar nueva información al modelo.

En el problema que nos concierne, empezaremos por describir la tabla, explicaremos la diferencia entre los distintos tipos de variables, hablaremos de la limpieza de datos, y procederemos a visualizar los datos y sus posibles patrones mediante la creación de histogramas, *boxplots*, *scatter plots*, y tablas de correlaciones. Finalmente, transformaremos los datos pertinentes mediante la codificación y normalización, y hablaremos de crear nuevos *inputs* a partir de los que ya tenemos por medio de la ingeniería de *features* o *feature engineering*.

4.1. Descripción de la tabla

La tabla de datos cuenta con filas, que indican todos los factores de calibrado para cada boya (muestras), y columnas, que corresponden a cada factor de calibrado, los cuales pueden ser *features* o *targets*. En total son 31308 muestras, que corresponden a las boyas fabricadas desde enero de 2022 hasta febrero de 2023.

A continuación, escribimos el diccionario que asocia el número de columna al parámetro de calibración:

1: Production_Version
2: Tipo_sonda
3: Lote_Sonda
4: Tipo_Resina
5: Tipo_base
6: Transformador_PCB
7: Equipo_cuba
8: Vpp_50KHz
9: Vpp_200KHz
10: Pn_50KHz
17: Resonancia_baja_frecuencia
18: Resonancia_alta_frecuencia
19: Valor_trans_con_carga_50KHz
20: Valor_trans_con_carga_200KHz
21: Linealidad_etapa_recep_50KHz
22: Linealidad_etapa_recep_200KHz
23: Media_TX_50KHz_minicuba
24: Media_RX_50KHz_minicuba
25: Media_TX_200KHz_minicuba
26: Media_RX_200KHz_minicuba

11: Pn_200KHz
12: tens_calibrado_50KHz
13: tens_calibrado_200KHz
14: num_iteraciones_50KHz
15: num_iteraciones_200KHz
16: Temperatura
27: RX 50KHz_cte_media
28: RX 50KHz_cte_dev
29: RX 200KHz_cte_media
30: RX 200KHz_cte_dev
31: Tiempo_curado

	1	2	3	4	5
0	2001.0	Disco Noliac	N48	SND-CAST(FLEXIBLE)	PLAST-YECT
1	2001.0	Disco Noliac	N48	SND-CAST(FLEXIBLE)	PLAST-YECT
2	2001.0	Disco Noliac	N48	SND-CAST(FLEXIBLE)	PLAST-YECT
3	2001.0	Disco Noliac	N48	SND-CAST(FLEXIBLE)	PLAST-YECT
4	2001.0	Disco Noliac	N48	SND-CAST(FLEXIBLE)	PLAST-YECT

6	7	8	9	10	11
Suesa	PCMARINE184	1425	1979	46.9	38.2
Suesa	PCMARINE190	1472	2007	46.0	37.0
Suesa	PCMARINE190	1457	1988	45.8	37.4
Suesa	PCMARINE184	1425	1979	43.8	38.2
Suesa	PCMARINE190	1422	1988	45.9	36.4

12	13	14	15	16	17
463.2	136.8	1	1	22.5	50.002
450.0	135.4	1	2	22.1	49.947
396.5	121.9	1	1	22.1	50.107
463.6	140.6	1	1	22.3	49.967
473.6	152.6	1	2	22.0	50.067

18	19	20	21	22	23
199.090	98.28	57.96	0.9996	0.9990	1880
198.607	99.96	58.80	0.9996	0.9990	1960
199.140	98.28	56.28	0.9996	0.9971	1810
199.174	100.80	58.80	0.9995	0.9964	1910
199.908	99.96	57.96	0.9995	0.9982	1880

24	25	26	27	28	29	30	31
703	1755	716	2.27	0.0	0.0	0.0	11
714	1750	750	2.38	0.0	0.0	0.0	11
706	1782	735	167.08	0.0	0.0	0.0	11
696	1760	729	228.27	0.0	0.0	0.0	11
675	1755	649	335.80	0.0	0.0	0.0	11

Tabla 4.2: Tabla de datos para las primeras cinco filas. El número de columna corresponde al *feature* del diccionario anterior.

4.2. Variables numéricas continuas, numéricas discretas y categóricas

Se dice **variables numéricas continuas** a aquellas que pueden tomar cualquier valor en un rango continuo. Estas variables tienen una escala de medición cuantitativa y se representan en términos de números decimales.

En los parámetros de calibrado, este tipo de variables corresponden a los *features* y *targets Vpp_50KHz, Vpp_200KHz, Pn_50KHz, Pn_200KHz, tens_calibrado_50KHz, tens_calibrado_200KHz, Temperatura, Resonancia_baja_frecuencia, Resonancia_alta_frecuencia, Valor_trans_con_carga_50KHz, Valor_trans_con_carga_200KHz, Linealidad_etapa_recep_50KHz, Linealidad_etapa_recep_200KHz, Media_TX_50KHz_minicuba, Media_RX_50KHz_minicuba, Media_TX_200KHz_minicuba* y *Media_RX_200KHz_minicuba*.

Por otra parte, las variables **numéricas discretas** son aquellas que solo pueden tomar valores enteros y se pueden contar. Nuestras variables numéricas discretas son: *Production_Version*, *num_iteraciones_50KHz*,

num_iteraciones_200KHz y *Tiempo_curado*.

Las **variables categóricas** son aquellas que representan categorías o grupos, y no se pueden medir en una escala numérica. Estas variables se dividen, a su vez, en dos tipos:

Variables **nominales**. Son aquellas que no tienen un orden específico, es decir, que sus clases no se pueden ordenar de manera jerárquica. En nuestro caso, las variables categóricas nominales son: *Tipo_sonda*, *Lote_sonda*, *Tipo_Resina*, *Tipo_base*, *Transformador_PCB* y *Equipo_cuba*.

Variables **ordinales**. Se dice de aquellas variables que siguen un orden específico o jerarquía. Con respecto a los parámetros de calibrado, no existe ninguna variable de este tipo ya que todas las categorías son nominales, pero algunos ejemplos de variables ordinales pueden ser el nivel de educación (primaria, secundaria, universidad) o la escala de calificación de un examen (*Sobresaliente*, *Notable*, o *Deficiente*).

4.3. Limpieza de datos

La limpieza de datos es un paso crucial en cualquier modelo de *machine learning*. Si nuestra tabla de datos contienen errores o datos irrelevantes, el modelo no podrá aprender patrones precisos y, por lo tanto, la precisión del modelo se verá afectada. Esta fase del AED ayuda a garantizar que los datos utilizados para entrenar el modelo sean precisos y relevantes.

Además, la limpieza de datos puede reducir el tiempo y los costos de desarrollo de modelos. Si los datos no están limpios, el modelo tardará más tiempo en entrenarse y los costes pueden aumentar debido a la necesidad de recopilar y procesar más datos.

Nosotros abordaremos la limpieza de datos siguiendo tres puntos: filtrado por valores faltantes, filtrado por rango y filtrado por *outliers*.

4.3.1. Filtrado por valores faltantes

Eliminar datos sobrantes es un proceso de limpieza importante en el análisis exploratorio de datos ya que debemos apartar del modelo aquellos in-

completos, innecesarios o irrelevantes.

	1	10	11	12	13
29311	NULL	57.7	37.0	442.5	142.7
29312	NULL	54.5	38.9	462.8	155.1
29313	NULL	54.5	38.9	462.8	155.1
29314	NULL	54.5	38.9	462.8	155.1
29315	NULL	54.5	38.9	462.8	155.1
29407	NULL	55.4	37.8	454.3	162.5

Tabla 4.3: Filas con valores faltantes (NULL) en nuestra tabla de datos. Se muestran las columnas con los *targets* y los *features* que contienen algún valor faltante.

4.3.2. Filtro por rango

Además, filtramos por rangos proporcionados por la empresa, en los *inputs* donde dispongamos de dichos rangos.

Los rangos proporcionados por la empresa son:

- Pn_50 KHz: $[30 - 65]$
- Pn_200 KHz: $[30 - 50]$
- tens_calibrado_50KHz: $[167 - 900]$
- tens_calibrado_200KHz: $[84 - 350]$
- Valor_trans_con_carga_50KHz: $[61.4 - 113.8]$
- Valor_trans_con_carga_200KHz: $[41.6 - 77.3]$
- Linealidad_etapa_recep_50KHz: $[0.985 - 1]$
- Linealidad_etapa_recep_200KHz: $[0.985 - 1]$
- Transformador_PCB: [Suesa, Corgan, Pomceg]

- Tipo_sonda: [Disco Fuji, Noliac, Zibo]
- Resonancia_baja_frecuencia: $[48.5 - 51.5]$
- Resonancia_alta_frecuencia: $[196 - 204]$
- Tipo_Resina: [SND-CAST(FLEXIBLE), SND-CAST(P-FLEXIBLE)]
- Media_TX_50KHz: $[1000 - 2500]$
- Media_TX_200KHz: $[400 - 1200]$
- Media_RX_50KHz: $[900 - 2000]$
- Media_RX_200KHz: $[350 - 1200]$
- Temperatura: $[18 - 28]$
- Equipo_cuba: [PCMARINE184, PCMARINE187, PCMARINE190]
- Tipo_base: [Plast-Yect, Valiant]

4.3.3. Filtro por *outliers*. *Scatter plots*

Los *scatter plots* (gráficos de dispersión) se utilizan a menudo en el análisis exploratorio de datos para visualizar la relación entre dos variables. Cada punto en el gráfico representa una observación en los datos, y la posición de cada punto en los ejes x e y corresponde al valor de las dos variables en esa observación. Son especialmente útiles para identificar patrones, tendencias y relaciones entre variables. En particular, los scatter plots pueden ayudar a identificar patrones de correlación entre los *inputs* del conjunto de datos. Lógicamente, si dos variables están altamente correladas, el valor de una variable tiende a estar estrechamente relacionado con el valor de la otra variable.

Además, los *scatter plots* son una herramienta muy útil para detectar *outliers* (valores atípicos) en un conjunto de datos. Los *outliers* son valores que se encuentran muy lejos del resto de los datos y pueden tener un impacto significativo en los resultados del análisis estadístico, por lo que es necesario filtrarlos para entrenar nuestros modelos correctamente. Si hay valores atípicos presentes, estos se destacarán claramente al encontrarse alejados del resto de la "nube" de puntos. Por ello, los *scatter plots* pueden ayudar a identificar

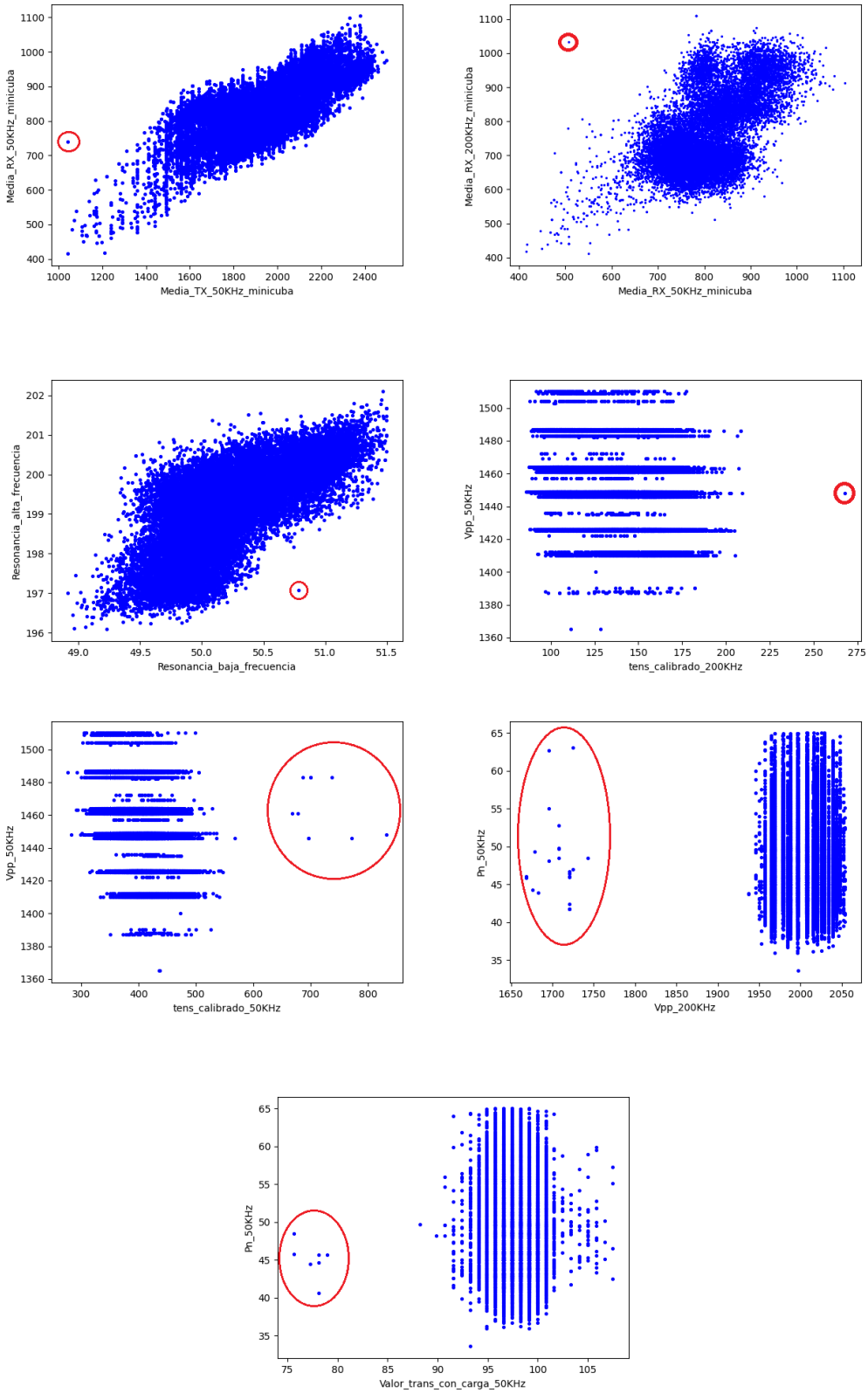


Figura 4.1: Outliers que filtramos debido a que están alejados de la nube de puntos de los *scatter plots*.

y visualizar la presencia de *outliers* de una manera muy intuitiva.

En la Figura 4.1, mostramos algunos de los *outliers* (rodeados en un círculo rojo) que aparecen cuando representamos en un *scatter plot* los pares de factores de calibrado de nuestra tabla de datos.

Existe otra técnica de limpieza de datos, que es el filtrado por *rango intercuartílico* (Véase Sección 4.5 y Figura 4.4), pero en nuestro problema hemos considerado que, debido a la gran cantidad de datos que se salen de este rango para casi todo *input*, no parece recomendable eliminarlos ya que ello implicaría una pérdida de información muy importante.

4.4. Histogramas

Los histogramas resultan útiles para detectar la forma de la distribución de los datos, incluyendo la simetría, la asimetría y la presencia de valores atípicos o extremos. Incluimos en la Figura 4.2 un histograma para cada factor de calibrado, siguiendo el diccionario en la Sección 4.1.

4.5. *Boxplots*

Un *boxplot* es una herramienta gráfica utilizada para representar la distribución de un conjunto de datos numéricos. El *boxplot* muestra la mediana del conjunto de datos, así como el rango intercuartílico (IQR), que es la distancia entre el primer cuartil (Q1) y el tercer cuartil (Q3). También muestra los valores mínimo y máximo del conjunto de datos, y los llamados "bigotes" que se extienden desde el borde del cuadro hasta los valores mínimo y máximo.

Incluimos en la Figura 4.4 un *boxplot* para cada factor de calibrado cuya variable es numérica continua, siguiendo el diccionario en la Sección 4.1.

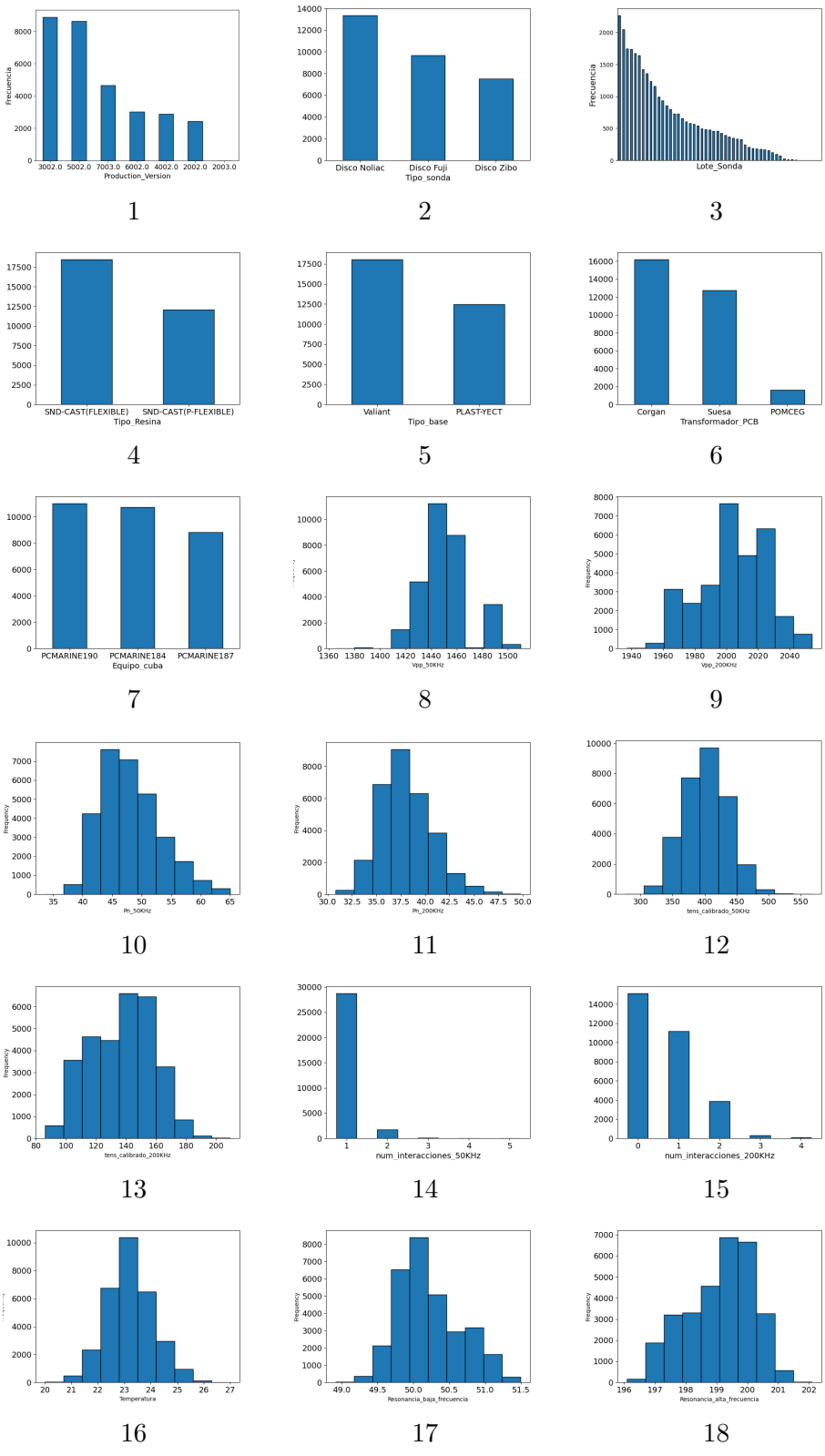
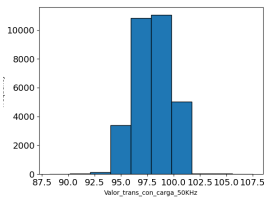
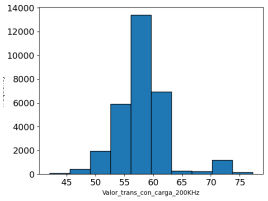


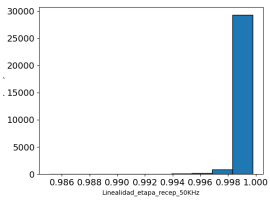
Figura 4.2: Histogramas para los factores de calibrado.



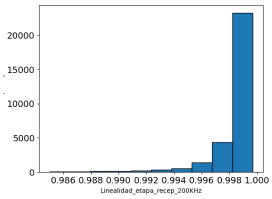
19



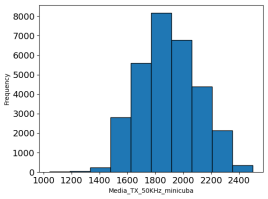
20



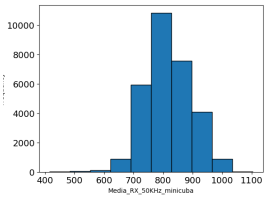
21



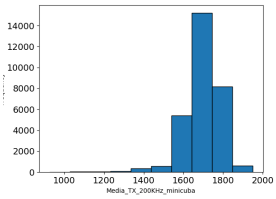
22



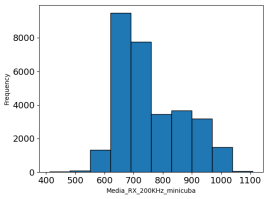
23



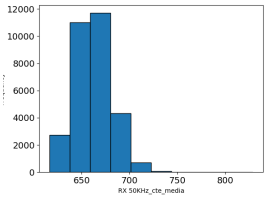
24



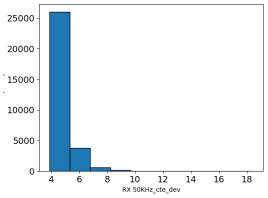
25



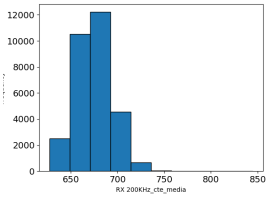
26



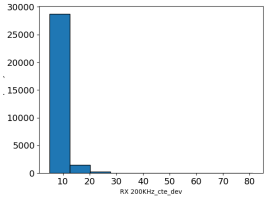
27



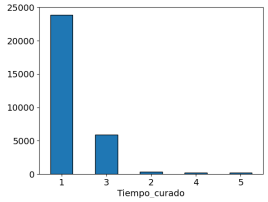
28



29



30



31

Histogramas para los factores de calibrado (continuación).

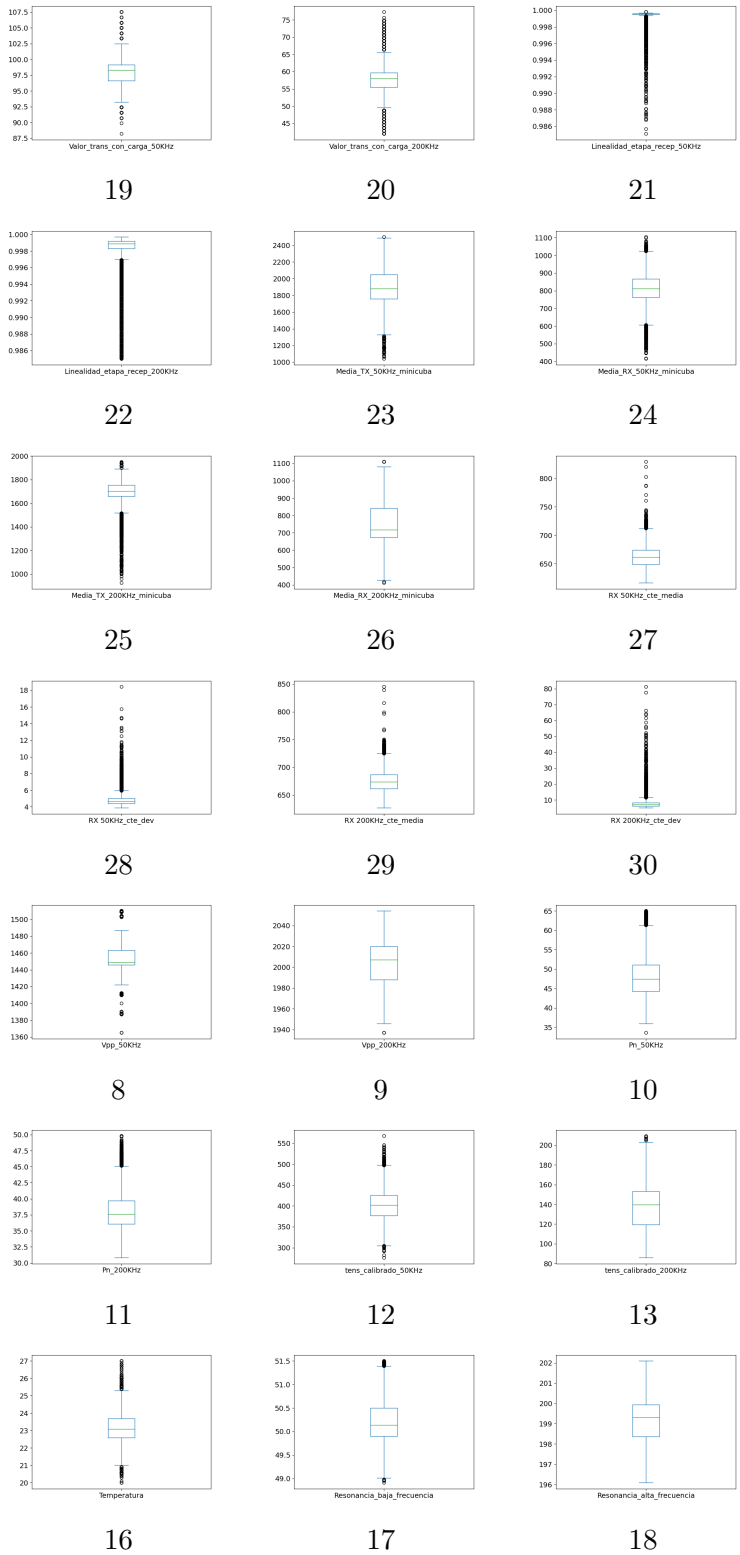


Figura 4.4: *Boxplots* para los factores de calibrado con variables numéricas continuas.

4.6. Correlaciones entre cada tipo de dato. ANOVA, Cramer y Pearson

El estudio de las correlaciones es importante en un problema de *machine learning* porque nos puede ayudar, por un lado, a identificar las variables que tienen una relación más fuerte con la variable objetivo y, por otro lado, a interpretar los resultados de un modelo, ya que las correlaciones fuertes entre una variable explicativa y la variable objetivo pueden indicar una fuerte influencia de la primera sobre la segunda.

Los coeficientes que hemos utilizado son los siguientes:

Correlación de **Pearson**. Es una medida de la relación lineal entre dos variables numéricas continuas. La correlación de Pearson se utiliza para medir la fuerza y la dirección de la relación entre dos variables continuas. Los valores de la correlación de Pearson van desde -1 (correlación negativa perfecta) hasta 1 (correlación positiva perfecta), con 0 que indica ausencia de correlación.

Correlación de **Cramer**. Es una medida de la relación entre dos variables categóricas, y se utiliza para medir la fuerza y la dirección de la relación entre dichas variables. Los valores de la correlación de Cramer van desde 0 (ausencia de correlación) hasta 1 (correlación perfecta).

ANOVA. ANOVA significa análisis de varianza y se utiliza para determinar si hay una diferencia significativa entre las medias de dos o más grupos de datos. Se utiliza para datos numéricos continuos y categóricos. Por ejemplo, si tenemos una variable numérica continua que queremos comparar entre diferentes grupos de una variable categórica, se puede utilizar ANOVA para determinar si hay una diferencia significativa entre las medias de los grupos.

Vemos que existen ciertas variables perfectamente correladas, sobre todo de tipo categórico (Figuras 4.5 − 4.7). Otras variables, como las que se refieren a la Tensión de la señal cuba-transductor del test en minicuba (*Media TX 50KHz*, *Media RX 50KHz*, *Media TX 200KHz* y *Media RX 200KHz*)

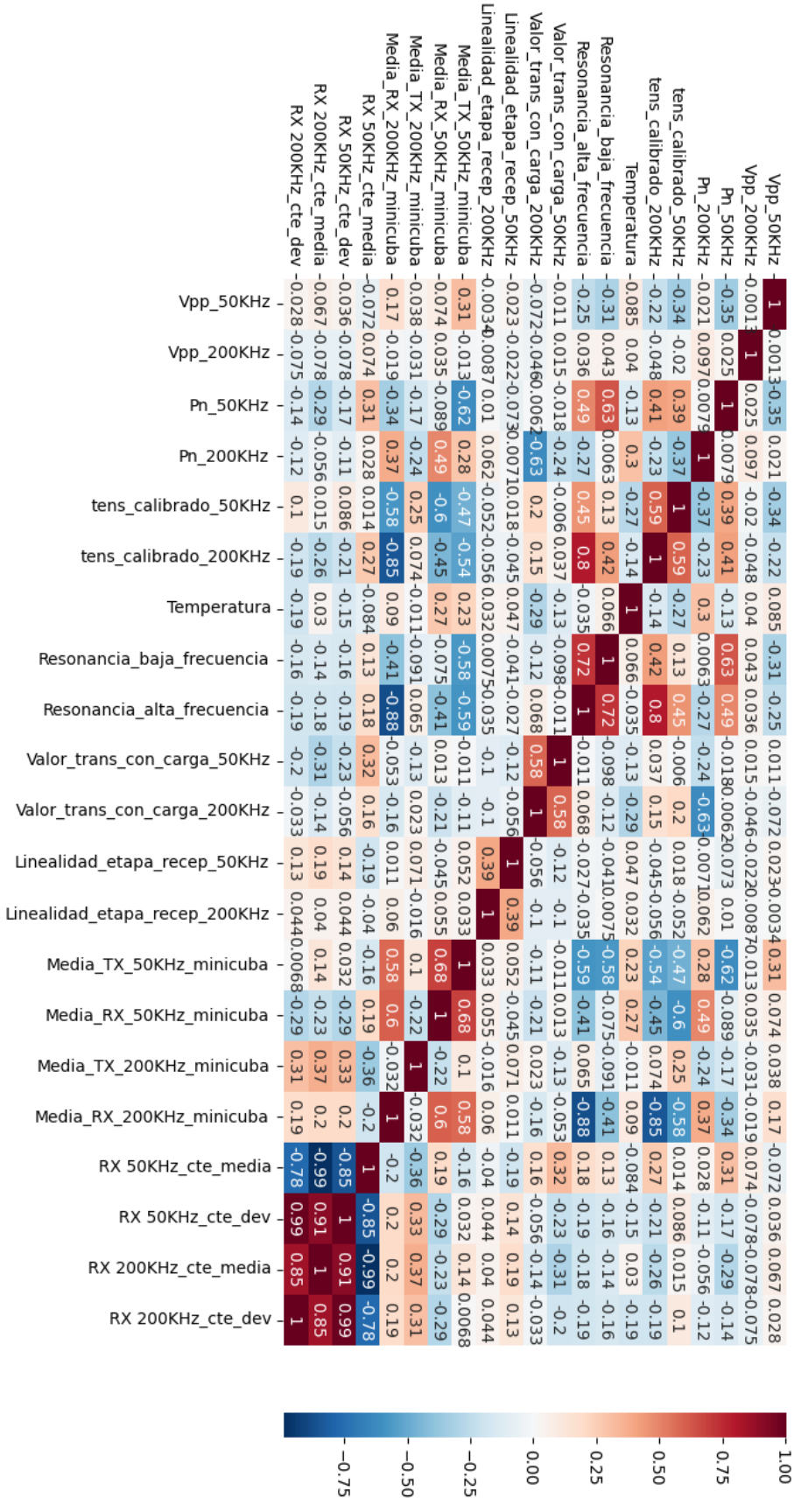


Figura 4.5: Tabla corr. Pearson entre variables numéricas continuas.

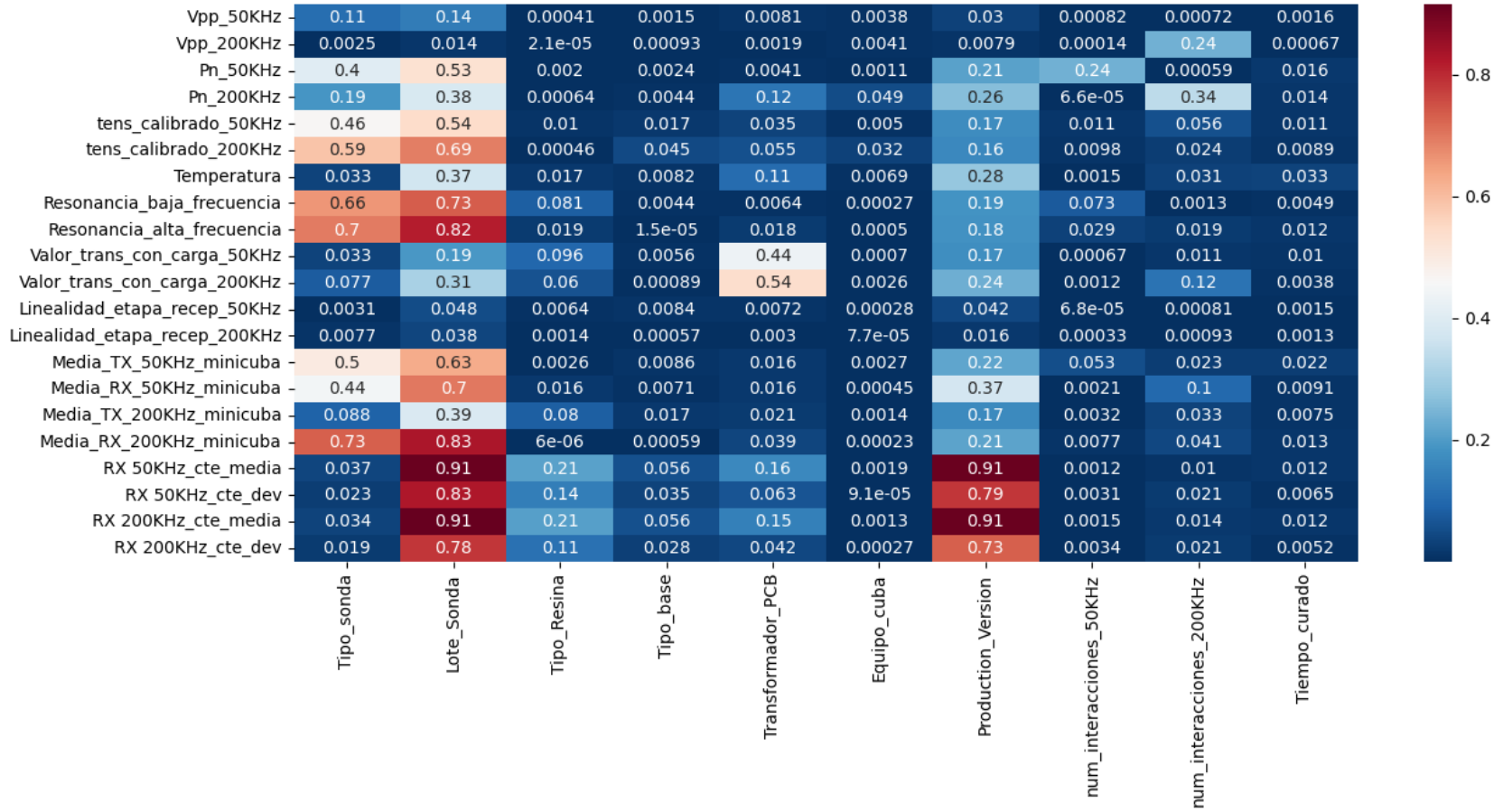


Figura 4.6: Tabla corr. ANOVA de variables numéricas continuas vs variables categóricas/num. discretas.

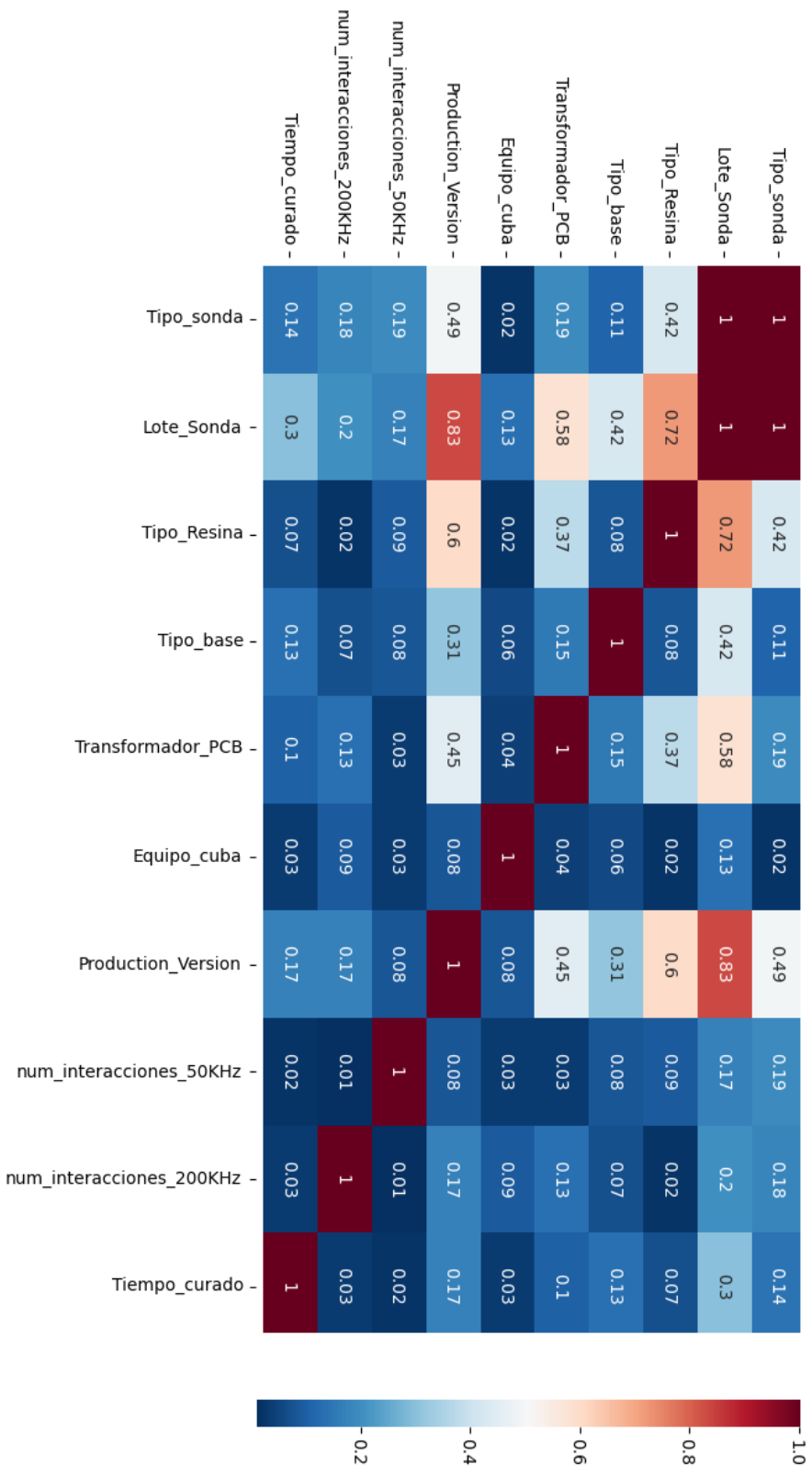


Figura 4.7: Tabla corr. V de Cramer entre variables categóricas o numéricas discretas.

o a las frecuencias de resonancia, están fuertemente correlacionadas con los *targets*, lo que nos puede dar una idea de la influencia que pueden llegar a tener en las variables de salida (Figura 4.5). Existe, por otro lado, variables como la linealidad en etapa de recepción (a 50 y 200 KHz) cuya influencia es muy pequeña o nula sobre el resto de parámetros, por lo que es de esperar que no influya casi nada en el rendimiento del modelo.

En Resultados, demostraremos que el filtrado de *inputs* poco correlados con los *targets* mejora en muchos casos nuestros modelos.

4.7. Transformación de datos

La transformación de datos es una parte crítica del proceso de entrenamiento de modelos de *machine learning*. Esta etapa implica preparar los datos para que puedan ser procesados y utilizados de manera efectiva por el modelo.

Las transformaciones de datos que llevaremos a cabo en nuestros modelos son las siguientes:

1 - Codificación de variables categóricas. Cuando las variables categóricas (nominales u ordinales) están presentes en los datos, se deben convertir en valores numéricos antes de utilizarlos en un modelo de *machine learning*. Esto se puede lograr mediante la codificación, que implica asignar un valor numérico a cada categoría única. Las dos técnicas más comunes para la codificación de variables categóricas nominales son:

Label Encoder. Se utiliza para transformar las categorías de una variable categórica en valores numéricos enteros únicos, donde cada valor entero representa una categoría diferente. Es importante tener en cuenta que no hay relación de orden entre las categorías y que los valores numéricos asignados son arbitrarios, por lo que no suele ser recomendado para variables nominales.

One Hot Encoder. Se utiliza para crear variables ficticias o "dummy" para cada categoría única de una variable categórica. Cada variable ficticia representa una categoría y puede tomar valores binarios (1 o 0), donde 1 indica que la observación pertenece a esa categoría y 0 indica que no lo hace. Este método, en general, es preferible al anterior cuando trabajamos con varia-

bles nominales, puesto que, por un lado, no se asume una relación de orden en categorías que no lo tienen y, por otro lado, evita la sobreestimación del efecto de las categorías al evitar la asignación de pesos numéricos arbitrarios a las categorías (por ejemplo, Potdar et al. 2017). Será el que aplicaremos a los *inputs* en todos los modelos.

Por ejemplo, para la columna Tipo_sonda, tenemos que crear tres columnas (*features*), una por cada clase, con valores binarios asignados a cada una (Tabla 4.4).

	Tipo_sonda	Tipo_sonda_cat1	Tipo_sonda_cat2	Tipo_sonda_cat3
9967	Disco Noliac	0	1	0
31069	Disco Noliac	0	1	0
24035	Disco Fuji	1	0	0
13694	Disco Fuji	1	0	0
22506	Disco Zibo	0	0	1
26316	Disco Noliac	0	1	0
9635	Disco Noliac	0	1	0
19158	Disco Zibo	0	0	1
6678	Disco Noliac	0	1	0
7439	Disco Fuji	1	0	0

Tabla 4.4: Ejemplo de codificación *One Hot Encoder* de la variable *Tipo_sonda*, para 10 filas aleatorias. *cat1*, *cat2* y *cat3* hacen referencia a Disco Fuji, Disco Noliac y Disco Zibo, respectivamente.

Resulta fácil deducir una gran desventaja de este método: el **aumento considerable de la dimensionalidad**, sobre todo en *features* que cuentan con muchas clases, puesto que cada una de ellas implica la creación de un nuevo *input*.

2 - Normalización de *inputs*: Las variables numéricas de los *inputs* pueden estar en diferentes escalas y rangos, lo que puede afectar el rendimiento del modelo. La normalización es una técnica que se utiliza para ajustar todos los parámetros de entrada para que tengan una escala común.

Existen varios tipos de normalización de datos, pero los más importantes son:

El *escalado min-max*. Es un proceso que tiene como objetivo transformar los datos de una variable para que se ajusten a un rango determinado, generalmente entre 0 y 1. Sigue la fórmula

$$X_{trans} = \frac{X - \text{mín}(X)}{\text{máx}(X) - \text{mín}(X)}. \tag{4.1}$$

Si queremos que ese rango sea uno específico, y no [0,1], debemos aplicar otra transformación más:

$$X_{trans} = X_{trans} \cdot (max - min) + min, \tag{4.2}$$

siendo max y min los extremos superior e inferior del intervalo, respectivamente.

La *estandarización*. Esta normalización implica escalar los datos de entrada para tener una media de cero y una desviación estándar de uno. No utilizaremos esta transformación, ya que la mayoría de nuestros *inputs* no siguen una distribución normal (véase Figura 4.2).

En nuestro problema, la única técnica de normalización que nos ha sido realmente útil es la del escalado *min-max*. Por ejemplo, para la columna *tens_calibrado_50KHz*, hemos obtenido los valores siguientes:

	tens_calibrado_50KHz	tens_calibrado_50KHz (normalizado)
0	463.2	0.642441
1	450.0	0.597189
2	396.5	0.413781
3	463.6	0.643812
4	473.6	0.678094
5	441.7	0.568735
6	492.3	0.742201
7	418.6	0.489544
8	453.8	0.610216
9	420.7	0.496743

10 primeros valores de la tensión de calibrado a 50 KHz. La primera columna es el dato sin procesar y, la segunda, el dato normalizado.

3 - Reducción de dimensionalidad. Se refiere a la transformación de datos para reducir el número de variables o dimensiones que se utilizan en un análisis, sin perder demasiada información. Tiene que ver con la **selección de mejores *features***.

La selección de *features* es un proceso importante que implica identificar y seleccionar un subconjunto relevante de *features* durante el proceso de entrenamiento del modelo. Con ello, buscamos reducir su complejidad y, potencialmente, mejorar su rendimiento.

Para seleccionar *features*, emplearemos la clase `SelectKBest`, la cual está disponible en el módulo `sklearn.feature_selection` de la librería scikit-learn (Sección 6.3). Esta función consiste en la selección de los k mejores *inputs*, donde k es un número que se especifica por adelantado. El criterio de selección se basa en una prueba estadística univariante entre las variables de entrada y de salida; en nuestro caso, la correlación.

4.8. *Feature Engineering*

La ingeniería de *features* (*feature engineering*) es el proceso de crear nuevas variables a partir de las ya existentes en un conjunto de datos. Tiene como objetivo mejorar el modelo al proporcionar más información relevante sobre los datos.

Se puede llevar a cabo de las siguientes formas:

Extracción de *features*. Consiste en extraer *inputs* de las variables existentes. Por ejemplo, si se tiene una variable de fecha, se puede extraer la hora del día, el día de la semana o el mes como nuevos *features*. En nuestro caso, esto será lo que haremos con la columna de *Fecha_test_ok*, la cual separaremos en un dato de días[2] y en otro de horas:

[2]El dato resultante *Fecha_dia* no es un *input* de nuestro modelo como tal, sino que es utilizado para crear el *feature* de *Tiempo_curado*.

	Fecha_test_ok	Fecha_dia	Fecha_hora
10248	2022-05-23 07:34:13.523	2022-05-23	07:34:13.523
18052	2022-09-28 12:44:56.240	2022-09-28	12:44:56.240
20221	2022-10-17 09:22:16.350	2022-10-17	09:22:16.350
479	2022-01-24 11:20:28.563	2022-01-24	11:20:28.563
24705	2022-11-18 09:50:43.607	2022-11-18	09:50:43.607
17695	2022-09-21 10:50:26.907	2022-09-21	10:50:26.907
24115	2022-11-15 08:35:40.013	2022-11-15	08:35:40.013
6377	2022-03-31 09:38:36.620	2022-03-31	09:38:36.620
4010	2022-03-08 08:34:09.443	2022-03-08	08:34:09.443
24357	2022-11-16 10:32:24.360	2022-11-16	10:32:24.360

Tabla 4.6: Fecha_test_ok, y las dos nuevas columnas que añadimos a partir de ella tras separar el dato de día y hora, para 10 filas aleatorias.

Combinación de *features*. Consiste en combinar dos o más variables existentes para crear un nuevo *feature*. En nuestro problema, no hemos incorporado esta técnica.

Capítulo 5

Ajuste de los datos a los modelos establecidos

En este capítulo, describiremos las principales técnicas de *machine learning* para encontrar la mejor manera de representar los datos a través de los modelos definidos en el Capítulo 3, de manera que puedan ser utilizados para realizar predicciones sobre los datos. El proceso de ajuste incluye varias etapas, como la selección del modelo apropiado, la optimización de los hiperparámetros, la división de los datos, el ajuste del modelo, la validación cruzada y la evaluación final del modelo (Raschka 2015; Hastie et al. 2009; etc.) A continuación, pasamos a describir cada etapa en detalle.

5.1. Selección del modelo

Existen diferentes tipos de modelos, cada uno con sus propias fortalezas y debilidades (véase Capítulo 3). La elección del modelo adecuado depende del problema que se está abordando, la naturaleza de los datos y la cantidad de datos disponibles. Es importante elegir el modelo que mejor se adapte al problema y los datos en cuestión.

Para nuestro problema de regresión, nos centraremos principalmente en tres métodos: las redes neuronales, el *Gradient Boosting* y el *XGBoost*. Para el segundo, utilizaremos una versión optimizada y mucho más rápida para conjuntos de datos grandes: el *Gradient Boosting* **basado en histogramas**. Este algoritmo se diferencia del *Gradient Boosting* usual en que, en vez de crear un nodo de decisión en base a un valor concreto del *feature*, utiliza

histogramas para agrupar valores similares y crear *bins* (consúltese Ke et al. 2017 y Wen et al. 2019 para más información). Veremos su implementación en Python en el siguiente capítulo.

Para nuestro problema de clasificación, los modelos que hemos considerado son el *Random Forest*, el *XGBoost* y el *Gradient Boosting*. Discutiremos los grados de precisión obtenidos para cada uno de los modelos de los dos problemas en Resultados.

5.2. *Splitting* de los datos

El *splitting* o "división" de datos es una práctica fundamental para obtener un modelo que sea capaz de generalizar bien a nuevos datos.

Cuando se entrena un modelo, se utiliza un cierto conjunto de datos para ajustar sus parámetros. Sin embargo, si utilizamos el 100 % de ese conjunto para el entrenamiento, no tendremos certeza si es capaz de generalizar bien a nuevos datos que le pasemos, y con ello corremos el riesgo de que exista sobreajuste. Para evitar esto, se divide el conjunto de datos en tres subconjuntos, que se utilizan en diferentes etapas del entrenamiento y evaluación del modelo: **entrenamiento**, **validación** y **prueba**. El conjunto de entrenamiento se utiliza para ajustar sus parámetros a los datos. Por otro lado, el conjunto de validación sirve como referencia para saber si este está ajustándose bien a los datos en cada una de las fases del entrenamiento. Finalmente, el conjunto de prueba se utiliza para evaluar su precisión una vez el entrenamiento ha finalizado.

Es importante asegurarse de que la división de los datos sea aleatoria y que todos los conjuntos contengan una muestra representativa de los datos.

En nuestro caso, hemos asignado el 10 % de todos los datos a nuestro conjunto de prueba (3049 muestras en total). El resto de datos ($31308 - 3049 = 28259$ muestras) se utilizarán como entrenamiento y validación en cada una de las fases de entrenamiento. El conjunto de validación es, por un lado, el 10 % del tamaño del conjunto de datos resultante al substraer el subconjunto de prueba (es decir, $0.1 \cdot 28259 = 2826$ muestras), mientras que el conjunto de entrenamiento son las muestras restantes ($28259 - 2826 = 25433$).

5.3. Ajuste del modelo

Una vez que se han seleccionado los modelos y se ha hecho el *splitting* correctamente, se puede proceder a ajustar el modelo a los datos, lo que se conoce como **entrenamiento**[3]. Durante el proceso de ajuste, el modelo recibe los datos de entrenamiento y se ajustan sus hiperparámetros para minimizar el error en las predicciones. Este proceso es iterativo y cuenta con varias fases, y es lo que se conoce como validación cruzada (*cross validation*).

5.3.1. Validación cruzada (*cross validation*)

En esta fase, el conjunto de entrenamiento se divide en k subconjuntos o *folds*. Seguidamente, se entrena el modelo en k iteraciones, utilizando en cada iteración uno de los subconjuntos como conjunto de validación y los $k-1$ restantes como conjunto de entrenamiento. El resultado final es el promedio de los resultados obtenidos en cada iteración.

La principal ventaja de la validación cruzada es que permite evaluar el rendimiento del modelo de manera más precisa y robusta que la simple división en subconjuntos de entrenamiento, validación y prueba, ya que no depende de una sola división aleatoria de los datos.

5.3.2. Precisión del modelo

Para tener constancia del rendimiento del modelo, necesitamos evaluar su precisión en cada una de las etapas de entrenamiento. Esto se hace comparando las predicciones con los valores reales del conjunto de validación. El rendimiento del modelo se puede medir utilizando diferentes métricas, como el coeficiente de determinación (R^2) o el error cuadrático medio (MSE).

En el problema de regresión, lo que nos interesa es saber cuánto nos desviamos de media del parámetro de calibrado real, por lo que hemos con-

[3]Es imprescindible realizar cualquier transformación de los datos, es decir, la normalización, codificación y reducción de dimensionalidad (Sección 4.7) **después** de haberse producido el *splitting* (véase Apéndice B), ya que de lo contrario estamos condicionando a nuestro conjunto de prueba para que retenga cierta información acerca del conjunto de entrenamiento, lo que se conoce como fuga de datos o *data leakage*.

Figura 5.1: Validación cruzada para un conjunto de datos con 5 *folds*. Nótese que, donde se indica *Test Set*, en realidad corresponde al *Validation Set*, ya que muchas referencias utilizan esta terminología de manera intercambiada.

siderado adecuado utilizar el *acierto relativo medio* (ARM) y el R^2.

R^2 es una medida de la proporción de la varianza total en la variable de respuesta que es explicada por el modelo. El valor de R^2 varía de 0 a 1, donde 0 indica que el modelo no explica la variabilidad en los datos y 1 indica que el modelo explica toda la variabilidad en los datos:

$$R^2 = 1 - \frac{\sum_{i=1}^{N_t} (y_i - \hat{y}_i)^2}{\sum_{i=1}^{N_t} (y_i - \bar{y})^2}, \tag{5.1}$$

donde N_t es el número de datos del conjunto de prueba, y_i es el valor real del *target* i, $\hat{y}_i$ es el valor predicho del *target* i, e $\bar{y}$ es el valor medio de los *targets*.

El ARM, por otro lado, es una medida de la precisión relativa de las predicciones del modelo. Se calcula como 100 menos el promedio de los errores porcentuales absolutos multiplicados por cien para cada punto de datos en la muestra:

$$ARM = 100 - \left(\frac{1}{N_t} \sum_{i=1}^{N_t} \frac{|y - \hat{y}|}{|y|} \right) \times 100. \tag{5.2}$$

Para el problema de clasificación, utilizaremos la función *accuracy*:

$$accuracy(y, \hat{y}) = \frac{1}{N} \sum_{i=0}^{N-1} 1_{\{\hat{y}_i = y_i\}}, \tag{5.3}$$

donde N es el número total de muestras y 1(x) es la *función indicador*, que vale 1 cuando, para una muestra i, el valor predicho coincide con el valor real, y 0 cuando no lo hace.

5.3.3. Optimización de los hiperparámetros

A la hora de aplicar nuestros modelos, necesitamos ajustar sus hiperparámetros (Sección 3.7) para encontrar el que mayor rendimiento ofrece y maximizar su precisión. Para ello utilizaremos el comando `GridSearchCV` de Python.

`GridSearchCV` es una herramienta de optimización de hiperparámetros en la librería de Python, scikit-learn (véase Sección 6.3). Esta herramienta nos ayuda a encontrar la combinación óptima de valores de hiperparámetros para un modelo dado. En lugar de probar manualmente diferentes combinaciones de valores, `GridSearchCV` automatiza el proceso y lo hace de manera exhaustiva, buscando entre todas las combinaciones posibles de valores dentro de un array definido por el usuario.

A continuación describimos el array de hiperparámetros considerados para cada modelo:

Redes neuronales

- *Número de neuronas por cada capa.* Valores posibles: (20,20), (30,30), (50,50) y (100,100).

- *Parámetro de regularización* α. Valores posibles: [1e-2, 1e-3, 1e-4].

- *Función de activación.* Valores posibles: ['relu', 'logistic', 'identity'],

- *Máximo número de iteraciones.* Este hiperparámetro hace referencia al número de *epochs* máximo. Valores posibles: [300, 500].

- *Learning rate.* Valores posibles: [0.005, 0.01, 0.03, 0.05].

- *Tipo de learning rate.* Este parámetro puede tener tres valores: constante (*constant*), con el valor igual al parámetro anterior en todo momento, decreciente (*invscaling*) o adaptativo (*adaptive*), el cual se mantiene constante hasta que la función de pérdida se estanca, en ese caso se divide entre 5. Valores posibles: ['constant', 'invscaling', 'adaptive'].

Gradient Boosting

Learning rate. Valores posibles: [0.05, 0.1, 0.2]

Profundidad máxima. Este hiperparámetro hace referencia a la máxima profundidad permitida del árbol de decisión, desde el nodo raíz hasta el nodo hoja más profundo. Valores posibles: [10, 20, 30, 50, 70, 100, None]. Cuando es igual a "None", no existe un límite de profundidad.

Número máximo de iteraciones. Valores posibles: [100, 200, 300].

Parámetro de regularización. Valores posibles: [0.1, 0.2, 0.3, 0.5, 0.7, 1.5].

Random Forest

Número de árboles. Valores posibles: [10, 50, 100, 250].

Profundidad máxima. Valores posibles: [30, 50, 70, 100, None].

Número mínimo de muestras por cada nodo hoja. Valores posibles: [2, 4, 6, 8, 10].

Número mínimo de muestras requerido para hacer un split. Valores posibles: [2, 4, 6, 8, 10].

XGBoost

Número de árboles. Valores posibles: [100, 200, 300].

Learning rate. Valores posibles: [0.1, 0.2, 0.25].

Profundidad máxima. Valores posibles: [20, 30, 50, 70, 100, None].

5.3.4. Selección exhaustiva de *features* con `GridSearchCV`

La selección de *features* basada en correlaciones (Sección 4.7) puede no ser adecuada para todos los conjuntos de datos. En algunos casos, los *features* con una correlación baja con los *targets* pueden ser importantes para la precisión del modelo. Por ello, lo que hacemos es tratar `SelectKBest` como si fuera un hiperparámetro más en el modelo, de manera que `GridSearchCV` busque el parámetro k que maximice la precisión del mismo junto al resto de hiperparámetros, a través de la implementación del Pipeline que proporciona sklearn (véase Sección 6.3).

5.4. Evaluación del modelo

Una vez el ajuste del modelo ha concluido, debemos evaluar su precisión en el conjunto de prueba. Esto se hace de manera análoga a lo que ya hemos hecho en el procedimiento de la validación cruzada, solo que aquí tenemos que aplicarlo una sola vez y sobre el conjunto de prueba, no sobre el de validación.

Capítulo 6

Software utilizado

En este Capítulo, explicamos brevemente cuales son las librerías, módulos, clases y métodos utilizados a lo largo de este PFM, todos basados en lenguaje Python.

6.1. IDE: Visual Studio Code

Visual Studio Code (VSCode) es un editor de código fuente gratuito, de código abierto, multiplataforma y altamente personalizable. Compilaremos todo el código Python con VSCode.

6.2. pandas

pandas es una librería de Python utilizada para el análisis y la manipulación de datos. Proporciona estructuras de alto nivel para manejar datos de forma eficiente, como DataFrames y Series. Pandas es especialmente útil para la limpieza, transformación, agregación y visualización de datos. Además, es compatible con la importación y exportación de datos desde y hacia varios formatos de archivo, como CSV, Excel y SQL.

Nuestra base de datos se carga a Python desde un archivo Excel, convirtiéndose en un objeto de la clase DataFrame de la siguiente manera:

```
import pandas as pd
df = pd.read_excel("datos_marine_M3iGO.xlsx")
```

Nótese que el método `read_excel()` es el que permite convertir el archivo Excel en un objeto DataFrame.

6.3. scikit-learn

scikit-learn (también conocido como sklearn) es una de las librerías de Python más populares para la implementación de algoritmos de *machine learning*. Contiene una amplia variedad de herramientas para la clasificación, la regresión, el clustering y la reducción de dimensionalidad, entre otros. Asimismo, proporciona herramientas para la validación de modelos, la selección de modelos y la preprocesamiento de datos. sklearn es una librería muy completa y es utilizada por muchos profesionales y científicos de datos en todo el mundo.

Concretamente, utilizaremos los módulos siguientes:

`sklearn.neural_network`. Módulo diseñado para modelos de redes neuronales. Importaremos la `MLPRegressor` para el problema de regresión.

`sklearn.preprocessing`. Este módulo es el encargado del preprocesamiento de datos, e incluye escalado, normalización y estandarización de datos. Importaremos las clases `MinMaxScaler` (escalado *min-max*), `OneHotEncoder` (codificación de los *inputs*) y `LabelEncoder` (codificación para convertir las categorías YES y NO del problema de clasificación en 1 y 0, respectivamente).

`sklearn.multioutput`. Módulo diseñado para modelos multirrespuesta. Importaremos la clase `MultiOutputRegressor` (modelos de regresión).

`sklearn.ensemble`. Módulo diseñado para modelos basados en métodos de ensamblado. Importaremos las clases `HistGradientBoostingRegressor`, `RandomForestClassifier` e `HistGradientBoostingClassifier`. La primera corresponde al problema de regresión y las dos últimas, al problema de clasificación.

`sklearn.metrics`. Módulo diseñado para implementar las métricas de evaluación de los modelos. Utilizaremos la clase `ConfusionMatrixDisplay` (confecciona el plot de la matriz de confusión), y los métodos `confusion_matrix()`

(calcula la matriz de confusión), `mean_absolute_percentage_error()` (calcula el error relativo), `r2_score()` (calcula coef. de determinación R^2) y `make_scorer()`, que convierte una función de nuestra elección en una métrica de evaluación. Este último método nos servirá para convertir el error relativo en acierto **relativo por cien**.

`sklearn.model_selection`. Este módulo se encargará de ajustar el modelo. Utilizaremos los métodos `train_test_split()`, que hace el *splitting*, y `cross_validate()`, que implementa la validación cruzada. Utilizaremos, asimismo, la clase `GridSearchCV` para buscar el óptimo de los hiperparámetros para todos los modelos considerados.

`sklearn.feature_selection`. Este módulo implementa algoritmos de selección de *features*. Utilizaremos la clase `SelectKBest` para seleccionar los k *features* con mayor coeficiente de correlación con el target (véase Sección 4.7).

Para el problema de clasificación, le pasaremos como argumento a una instancia de `SelectKBest` la función `f_classif`, que mide el coeficiente de correlación entre variables numéricas y categóricas, y corresponde al test ANOVA de la Sección 4.6. En el problema de regresión, emplearemos la función `f_regression`, que mide la correlación entre variables numéricas y se basa en el coeficiente de Pearson. Así, `SelectKBest` seleccionará los k *features* con mayor coef. de correlación entre las variables de entrada y de salida, y filtrará el resto.

`sklearn.pipeline`. Este módulo nos resultará de gran importancia. Implementa funcionalidades para construir un modelo compuesto que permite la inclusión de múltiples modelos y de distintas transformaciones de datos, a través de la clase `Pipeline`. Esta clase es tremendamente útil ya que, por un lado, implementa automáticamente la transformación de datos una vez el *splitting* tiene lugar mediante la instancia de un objeto `GridSearchCV` y, por otro lado, permite la optimización de hiperparámetros sobre múltiples modelos.

Incluimos aquí un ejemplo de Pipeline para un modelo compuesto en el problema de regresión, incluyendo normalización, codificación, y selección de *features*:

```
    '''Transformación de las columnas numéricas mediante el
        escalado min-max'''
    numeric_transformer = Pipeline(steps=[('scaler',
       MinMaxScaler())])
    '''Transformación de las columnas categóricas mediante
       la codificación OneHotEncoder'''
    categorical_transformer = Pipeline(steps=[('onehot',
       OneHotEncoder(handle_unknown = 'ignore'))])
    preprocessor =  ColumnTransformer(transformers=[
    ('cat', categorical_transformer,
       col_interest_categorical),
    ('num', numeric_transformer, col_interest_numeric)
    ])
    '''Modelo compuesto de una red neuronal, un Gradient
       Boosting y un XGBoost para el problema de regresión
       '''
    mlp1 = MLPRegressor()
    mlp2 = HistGradientBoostingRegressor()
    mlp3 = XGBRegressor()
    ''' Declaración del Pipeline definido por la
       transformación de datos (preprocesamiento), la
       selección de features y el modelo compuesto
    '''
    pipe = Pipeline(steps=[('preprocessor', preprocessor),
    ('kbest', SelectKBest(f_regression)),
    ('regressor',PipelineHelper([('NN', mlp1), ('hist',
       mlp2), ('xgb',mlp3)]))])
```

`sklearn.compose`. Este módulo permite la transformación de datos a la hora de definir el Pipeline para el entrenamiento de los modelos mediante la clase `ColumnTransformer`.

6.4. seaborn

Seaborn es una librería de visualización de datos en Python basada en Matplotlib. Seaborn proporciona una interfaz de alto nivel para crear gráficos estadísticos atractivos y informativos con menos líneas de código que con Matplotlib. Además, cuenta con una amplia gama de gráficos estadísticos para explorar las relaciones entre variables en los datos, como gráficos de dispersión, gráficos de barras, gráficos de cajas y bigotes, gráficos de violín y mapas de calor.

Nosotros utilizaremos esta librería solo para calcular el mapa de calor de las tablas de correlaciones, a través de la función `heatmap()`:

```
import seaborn as sn
import matplotlib.pyplot as plt
table = pd.DataFrame()
"""Aplicamos el mapa de calor al objeto table,
que es un DataFrame"""
sn.heatmap(table,annot=True)
plt.show()
```

Esta función es útil puesto que ayuda visualmente a localizar las correlaciones grandes, particularmente cuando la dimensionalidad es muy alta.

6.5. pingouin

Pingouin es una librería de Python para análisis estadístico que cuenta con una amplia gama de funciones estadísticas, como correlación, análisis factorial y regresión. Además, Pingouin proporciona una interfaz clara y coherente para realizar estas funciones, lo que hace que el análisis de datos sea más fácil y eficiente.

Utilizaremos pingouin básicamente para calcular la tabla ANOVA (Sección 4.6), ya que esta librería proporciona el método `anova()` para su cálculo a partir de un DataFrame.

6.6. xgboost

xgboost es la librería que implementa el modelo *boosting* del mismo nombre. Utilizamos las clases `XGBClassifier` y `XGBRegressor` para clasificación y regresión, respectivamente.

6.7. Otras librerías

Además de las ya mencionadas, también utilizaremos otras librerías como Matplotlib, Scipy, Numpy, sys, pipelinehelper...

Capítulo 7

Resultados

En este Capítulo, presentamos los resultados obtenidos al aplicar las técnicas de ajuste de datos descritas en el Capítulo 5 a los modelos presentados en el Capítulo 3.

Primero, determinamos los mejores hiperparámetros para cada modelo de los problemas de clasificación y regresión. Los valores posibles de cada parámetro son los establecidos en la Sección 5.3.3. A continuación, calculamos la media de precisión del conjunto de entrenamiento y validación. Después, evaluamos el modelo con dichos parámetros en el conjunto de prueba y calculamos su precisión. Finalmente, para el problema de clasificación, obtenemos los *features* más importantes en el mejor modelos, los *features* con mayor importancia y las matrices de confusión.

7.1. Problema de regresión

Como ya se ha comentado, nuestro objetivo, en este problema, es predecir los valores numéricos de los *targets* a partir de los *features*. Debido a que contamos con cuatro *targets* (*Pn_50KHz, Pn_200KHz, tens_calibrado_50KHz* y *tens_calibrado_200KHz*), podemos tratar nuestros problemas tomando los targets individualmente o conjuntamente, lo que llamaremos regresión multirespuesta. Para el problema de regresión, trataremos estos dos casos por separado.

7.1.1. Regresión con una sola variable de respuesta

En esta Sección, presentamos los resultados obtenidos para cada uno de los cuatro problemas de regresión donde, en cada uno de ellos, la variable a predecir corresponde a un *output*.

Pn_50KHz

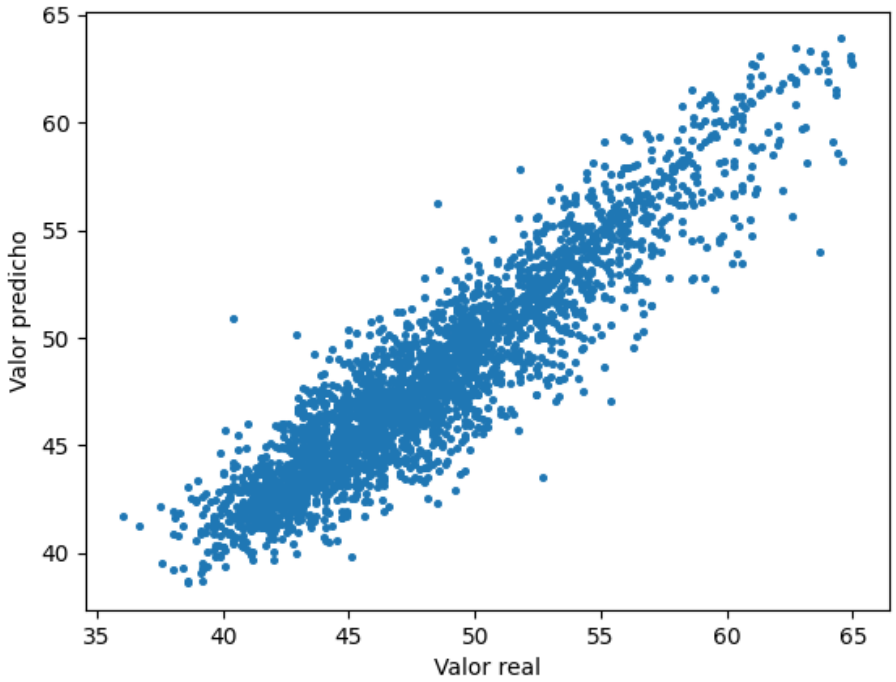


Figura 7.1: *Scatter plot* del target real vs la predicción para el mejor modelo de *Pn_50KHz* para el conjunto de prueba.

Mejores hiperparámetros para XGBoost: *learning rate* 0.1, profundidad máxima 30, número de árboles 200. Selección de *features*: 60. ARM = 97.06 %, $R^2 = 0.86$ (conjunto validación).

Mejores hiperparámetros para *Gradient Boosting*: parámetro de regularización 0.3, *learning rate* 0.1, profundidad máxima 20, número máximo de iteraciones 300. Selección de *features*: 60. ARM = 96.9 %, $R^2 = 0.85$ (con-

junto validación).

Mejores hiperparámetros para la red neuronal: función de activación sigmoide, parámetro de regularización α 0.0001, tamaño de las capas (100,100), *learning rate* decreciente con valor inicial 0.005, número máximo de iteraciones 300. Selección de *features*: 60. ARM = 96.74 %, $R^2 = 0.83$ (conjunto validación).

Mejor modelo: XGBoost, con un ARM de 97.06 % y $R^2 = 0.86$ para el conjunto de validación y un ARM de 99.99 % y $R^2 = 0.99$ para el conjunto de entrenamiento tras 10 iteraciones de la validación cruzada.

Evaluación en conjunto de prueba: ARM = 97.15 %, $R^2 = 0.84$.

Pn_200KHz

Mejores hiperparámetros para XGBoost: *learning rate* 0.1, profundidad máxima 20, número de árboles 300. Selección de *features*: 40. ARM = 98.04 %, $R^2 = 0.86$ (conjunto validación).

Mejores hiperparámetros para *Gradient Boosting*: parámetro de regularización 0.7, *learning rate* 0.1, profundidad máxima ninguna, número máximo de iteraciones 300. Selección de *features*: todos. ARM = 98.02 %, $R^2 = 0.86$ (conjunto validación).

Mejores hiperparámetros para la red neuronal: función de activación sigmoide, parámetro de regularización α 0.001, tamaño de las capas (100,100), *learning rate* decreciente con valor inicial 0.005, número máximo de iteraciones 300. Selección de *features*: todos. ARM = 97.95 %, $R^2 = 0.85$ (conjunto validación).

Mejor modelo: XGBoost, con un ARM de 98.04 % y $R^2 = 0.86$ para el conjunto de validación y un ARM de 99.99 % y $R^2 = 0.99$ para el conjunto de entrenamiento tras 10 iteraciones de la validación cruzada.

Evaluación en conjunto de prueba: ARM = 98.02 %, $R^2 = 0.83$.

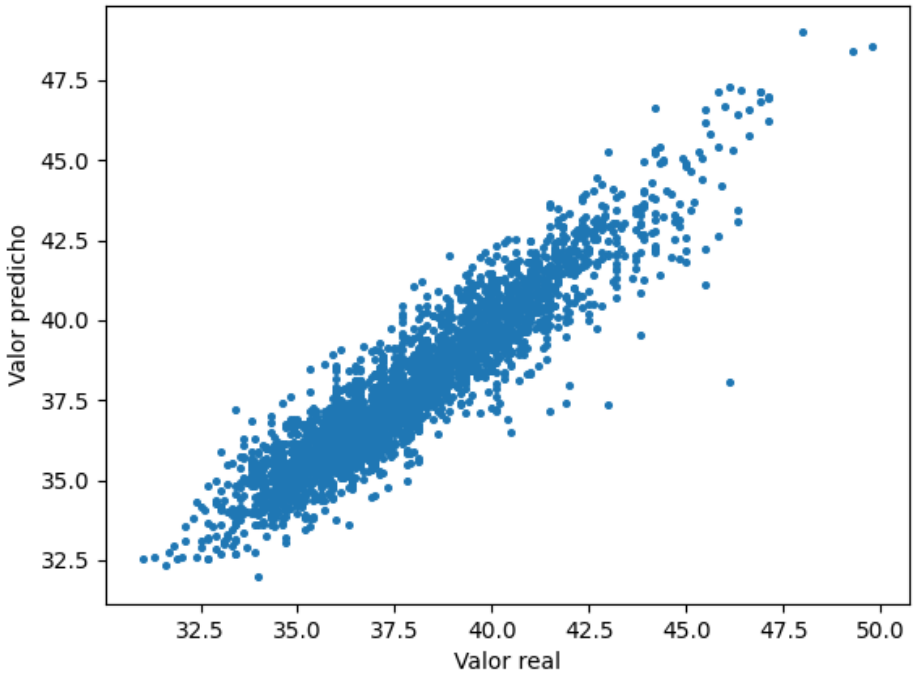


Figura 7.2: *Scatter plot* del target real vs la predicción para el mejor modelo de *Pn_200KHz* en el conjunto de prueba.

tens_calibrado_50KHz

Mejores hiperparámetros para XGBoost: *learning rate* 0.1, profundidad máxima 20, número de árboles 300. Selección de *features*: 60. ARM = 96.78 %, $R^2 = 0.75$ (conjunto validación).

Mejores hiperparámetros para *Gradient Boosting*: parámetro de regularización 0.7, *learning rate* 0.1, profundidad máxima 30, número máximo de iteraciones 300. Selección de *features*: todos. ARM = 96.68 %, $R^2 = 0.75$ (conjunto validación).

Mejores hiperparámetros para la red neuronal: función de activación ReLU, parámetro de regularización α 0.0001, tamaño de las capas (100,100), *learning rate* adaptativo con valor inicial 0.01, número máximo de iteraciones 500. Selección de *features*: 60. ARM = 96.38 %, $R^2 = 0.69$ (conjunto validación).

Mejor modelo: XGBoost, con un ARM de 96.78 % y $R^2 = 0.75$ para el conjunto de validación y un ARM de 99.99 % y $R^2 = 0.99$ para el conjunto de entrenamiento tras 10 iteraciones de la validación cruzada.

Evaluación en conjunto de prueba: ARM = 96.70 %, $R^2 = 0.69$.

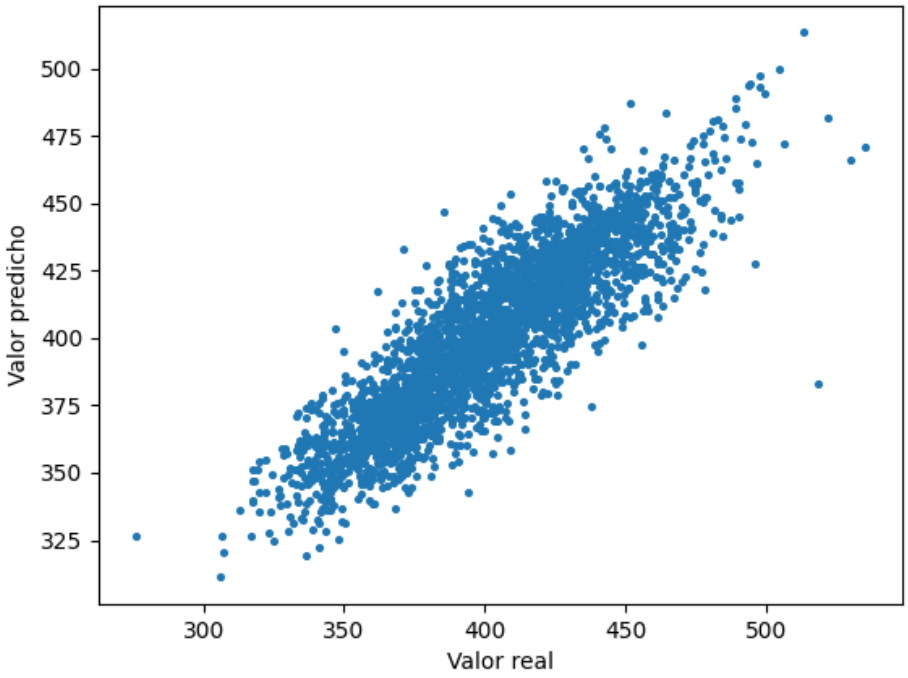


Figura 7.3: *Scatter plot* del target real vs la predicción para el mejor modelo de *tens_calibrado_50KHz* para el conjunto de prueba.

tens_calibrado_200KHz

Mejores hiperparámetros para XGBoost: *learning rate* 0.1, profundidad máxima 20, número de árboles 300. Selección de *features*: todos. ARM = 96.99 %, $R^2 = 0.92$ (conjunto validación).

Mejores hiperparámetros para *Gradient Boosting*: parámetro de regularización 0.5, *learning rate* 0.1, profundidad máxima 20, número máximo de ite-

raciones 300. Selección de *features*: 50. *Accuracy*: ARM = 96.82 %, $R^2 = 0.92$ (conjunto validación).

Mejores hiperparámetros para red neuronal: función de activación sigmoide, parámetro de regularización α 0.001, tamaño de las capas (100,100), *learning rate* constante con valor inicial 0.01, número máximo de iteraciones 300. Selección de *features*: todos. ARM = 96.62 %, $R^2 = 0.91$ (conjunto validación).

Mejor modelo: XGBoost, con un ARM de 96.99 % y $R^2 = 0.92$ para el conjunto de validación y un ARM de 99.99 % y $R^2 = 0.99$ para el conjunto de entrenamiento tras 10 iteraciones de la validación cruzada.

Evaluación en conjunto de prueba: ARM = 96.97 %, $R^2 = 0.92$.

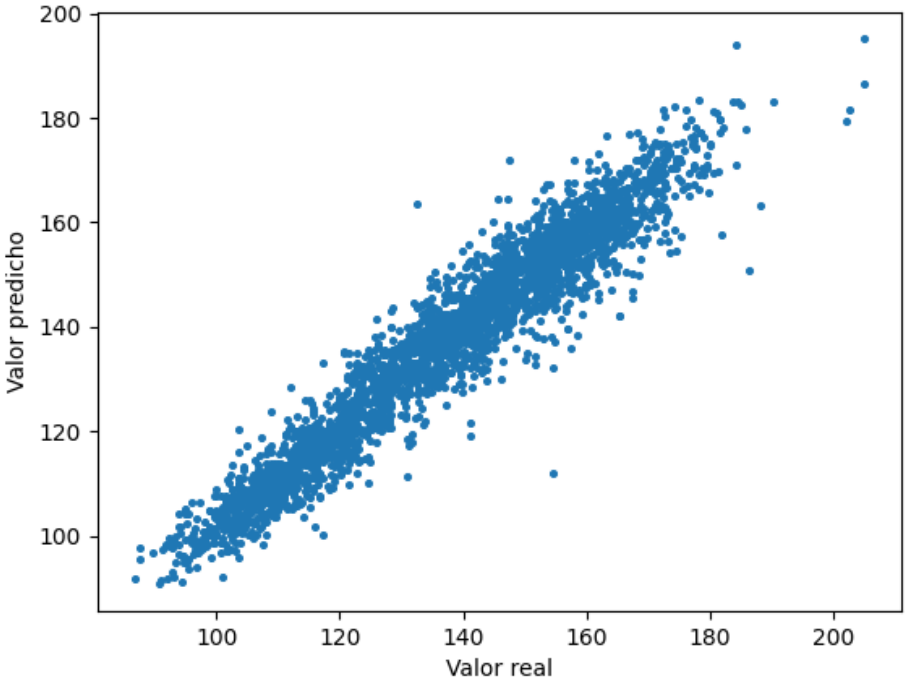


Figura 7.4: Scatter plot del target real vs la predicción para el mejor modelo de *tens_calibrado_200KHz* para el conjunto de prueba.

7.1.2. Regresión multi-respuesta

Mejores hiperparámetros para XGBoost: *learning rate* 0.1, profundidad máxima 20, número de árboles 300. Selección de *features*: todos. ARM = 97.2 %, $R^2 = 0.85$ para el conjunto de validación y ARM = 99.99 %, $R^2 = 0.99$ para el conjunto de entrenamiento.

Evaluación en conjunto de prueba: ARM = 97.34 %, $R^2 = 0.82$.

Extracto del conjunto de prueba de las primeras 7 boyas:

$$\begin{bmatrix} 47.3 & 39.8 & 373.2 & 164.7 \\ 42.7 & 36.7 & 383.9 & 124.7 \\ 44.9 & 34.7 & 464.4 & 145.2 \\ 45.9 & 37.9 & 452.4 & 172.3 \\ 46.3 & 36.3 & 382.8 & 167.2 \\ 43.2 & 40.5 & 379.5 & 142.3 \\ 45.2 & 38.6 & 372.8 & 126.0 \end{bmatrix}$$

Predicciones:

$$\begin{bmatrix} 48.479435 & 40.101635 & 381.36447 & 163.62585 \\ 43.62341 & 35.81528 & 367.27335 & 117.71173 \\ 43.240444 & 33.21628 & 443.3773 & 135.78763 \\ 47.074097 & 37.15727 & 431.89502 & 160.67905 \\ 48.30173 & 37.581818 & 410.49164 & 172.69499 \\ 45.607063 & 41.173794 & 395.79562 & 145.73602 \\ 45.241505 & 38.0498 & 374.96942 & 125.737206 \end{bmatrix}$$

7.2. Problema de clasificación

En el problema de clasificación, transformamos los valores numéricos de los *targets* en dos clases, YES o NO, según si su diferencia con respecto a la media es menor o mayor que una desviación estándar, respectivamente. El objetivo es predecir, para un cierto conjunto de *features*, a cual de estas dos categorías pertenecen. Si la predicción es correcta, significa que el calibrado es correcto y, si no lo es, existe información no recogida en los inputs que

no estamos considerando y, tomando como referencia estos, decimos que el calibrado es "incorrecto".

Además, en esta Sección, obtenemos los cinco *inputs* más importantes para cada *output*, es decir, aquellos que resultan más útiles para predecir el *output*. Como el mejor modelo para todos los casos es el XGBoost, utilizamos el atributo feature_importances_ de la clase `XGBoostClassifier`. Este atributo se calcula sumando la ganancia o reducción total de la función de pérdida atribuida a dicho *feature* en todos los árboles de decisión. Dicha ganancia o reducción se obtiene, a su vez, como sigue: para cada nodo en un árbol de decisión, el programa computa la diferencia de la función de pérdida entre antes y después de hacer el *split* basado en el *feature* en dicho nodo. Esta ganancia o reducción es, posteriormente, ponderada por el número de muestras en el nodo (véase la documentación de XGBoost en https://xgboost.readthedocs.io/en/latest/python/python_api.html#xgboost.Booster.feature_importances).

Finalmente, se procede a calcular las "matrices de confusión" para cada uno de los *targets* en el conjunto de prueba. Estas matrices proporcionan una representación visual útil para evaluar el rendimiento del modelo en un problema de clasificación.

Pn_50KHz

Mejores hiperparámetros para XGBoost: *learning rate* 0.1, profundidad máxima 50, número de árboles 300. Selección de *features*: todos. *Accuracy*: 86.34 %

Mejores hiperparámetros para *Gradient Boosting*: parámetro de regularización 0.2, *learning rate* 0.1, profundidad máxima 10, número máximo de iteraciones 300. Selección de *features*: todos. *Accuracy*: 85 %

Mejores hiperparámetros para *Random Forest*: profundidad máxima 30, número mínimo de muestras por cada nodo hoja 2, número mínimo de muestras requeridas para hacer el *split* 2, número de árboles 250. Selección de *features*: todos. *Accuracy*: 85.9 %

Mejor modelo: XGBoost, con *accuracy* 86.34 %.

Evaluación en conjunto de prueba. *Accuracy*: 87 %.

5 *features* más importantes: Tipo_sonda_Disco Fuji (0.24), *num_iteraciones_50KHz* (0.22), *Tipo_Resina_SND-CAST(FLEXIBLE)* (0.14), *Lote_Sonda_47169* (0.06) y *Lote_Sonda_46436* (0.02).

Pn_200KHz

Mejores hiperparámetros para XGBoost: *learning rate* 0.1, profundidad máxima 70, número de árboles 300. Selección de *features*: todos. *Accuracy*: 85.19 %

Mejores hiperparámetros para *Gradient Boosting*: parámetro de regularización 0.7, *learning rate* 0.1, profundidad máxima 10, número máximo de iteraciones 300. Selección de *features*: todos. *Accuracy*: 84.4 %

Mejores hiperparámetros para *Random Forest*: profundidad máxima 50, número mínimo de muestras por cada nodo hoja 2, número mínimo de muestras requeridas para hacer el *split* 2, número de árboles 250. Selección de *features*: todos. *Accuracy*: 84.44 %

Mejor modelo: XGBoost, con *accuracy* 85.19 %.

Evaluación en conjunto de prueba. *Accuracy*: 85 %.

5 features más importantes: *Lote_Sonda_46436* (0.22), *Tipo_sonda_Disco Noliac* (0.06), *Lote_Sonda_RW3* (0.05), *RX 200KHz_cte_dev* (0.049) y *Valor_trans_con_carga_200KHz* (0.03).

tens_calibrado_50KHz

Mejores hiperparámetros para XGBoost: *learning rate* 0.1, profundidad máxima 20, número de árboles 300. Selección de *features*: todos. *Accuracy*: 80.9 %

Mejores hiperparámetros para *Gradient Boosting*: parámetro de regularización 1.5, *learning rate* 0.1, profundidad máxima 20, número máximo de

iteraciones 300. Selección de *features*: 60. *Accuracy*: 79.12 %

Mejores hiperparámetros para *Random Forest*: profundidad máxima 50, número mínimo de muestras por cada nodo hoja 2, número mínimo de muestras requeridas para hacer el *split* 2, número de árboles 250. Selección de *features*: todos. *Accuracy*: 80.84 %

Mejor modelo: XGBoost, con *accuracy* 80.9 %.

Evaluación en conjunto de prueba. *Accuracy*: 81 %

5 features más importantes: *Tipo_sonda_Disco Fuji* (0.29), *Tipo_sonda_Disco Zibo* (0.21), *Lote_Sonda_47100* (0.04), *Lote_Sonda_N80* (0.02) y *Lote_Sonda_RW-M1* (0.016).

tens_calibrado_200KHz

Mejores hiperparámetros para XGBoost: *learning rate* 0.2, profundidad máxima 30, número de árboles 300. Selección de *features*: todos. *Accuracy*: 87.98 %

Mejores hiperparámetros para *Gradient Boosting*: parámetro de regularización 0.5, *learning rate* 0.1, profundidad máxima 20, número máximo de iteraciones 300. Selección de *features*: todos. *Accuracy*: 87.51 %

Mejores hiperparámetros para *Random Forest*: profundidad máxima 30, número mínimo de muestras por cada nodo hoja 2, número mínimo de muestras requeridas para hacer el *split* 8, número de árboles 250. Selección de *features*: todos. *Accuracy*: 87.37 %

Mejor modelo: XGBoost, con *accuracy* 87.98 %.

Evaluación en conjunto de prueba. *Accuracy*: 81 %

5 features más importantes: *Tipo_sonda_Disco Fuji* (0.23), *Lote_Sonda_16033QR* (0.1), Tipo_base_PLAST-YECT (0.08), *Equipo_cuba_PCMARINE190* (0.07) y *Lote_Sonda_N77MIX* (0.05).

7.2.1. Matrices de confusión

Las matrices de confusión son matrices que muestran el número de predicciones correctas e incorrectas hechas por un modelo de clasificación en un conjunto de datos. Las filas indican las clases verdaderas (0=NO y 1=YES) y las columnas, las clases predichas por el modelo. Cada celda de la matriz cuenta con el número de muestras en el conjunto de prueba que pertenecen a la clase de la fila correspondiente y que fueron predichas por el modelo como pertenecientes a la clase de la columna consiguiente. La diagonal principal, por tanto, nos dice la cantidad de predicciones correctas, mientras que las demás celdas representan las predicciones incorrectas. A continuación, presentamos las cuatro matrices de confusión obtenidas:

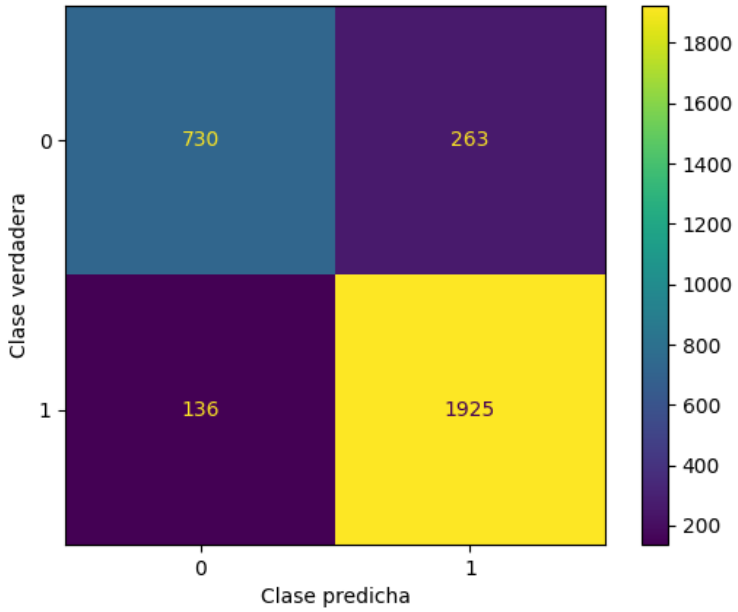


Figura 7.5: Matriz de confusión para el conjunto de prueba del *output* Pn_50KHz.

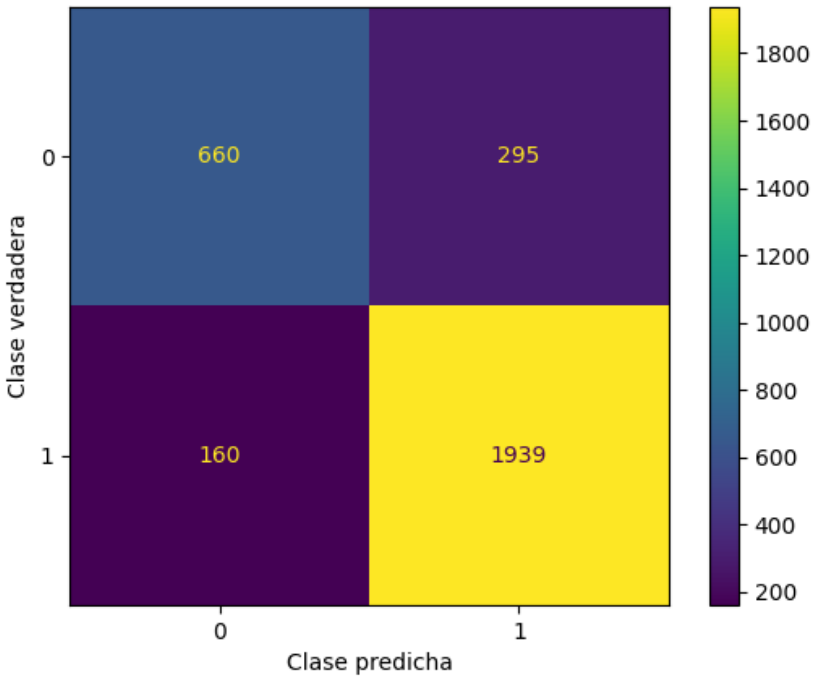


Figura 7.6: Matriz de confusión para el conjunto de prueba del *output* Pn_200KHz.

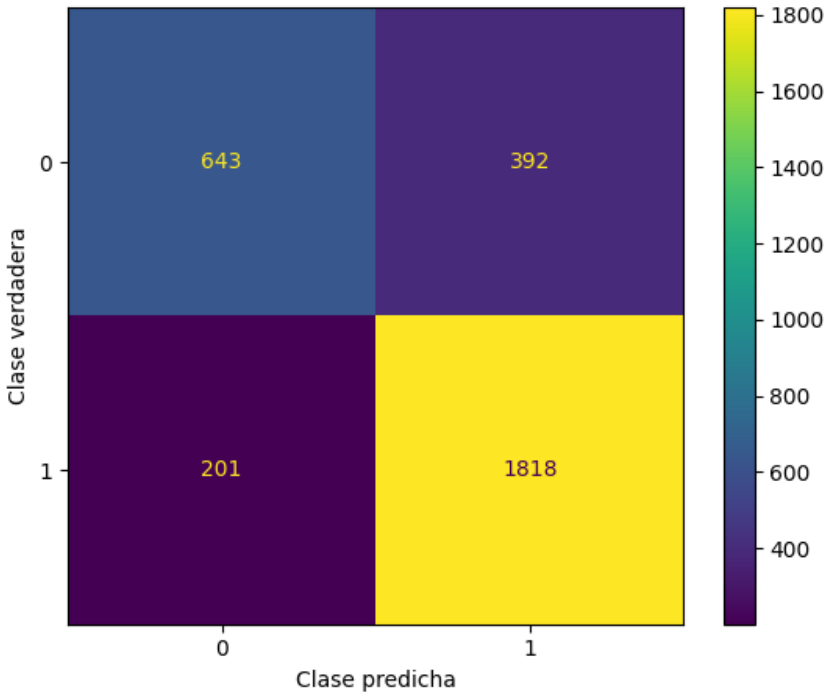


Figura 7.7: Matriz de confusión para el conjunto de prueba del *output* tens_calibrado_50KHz.

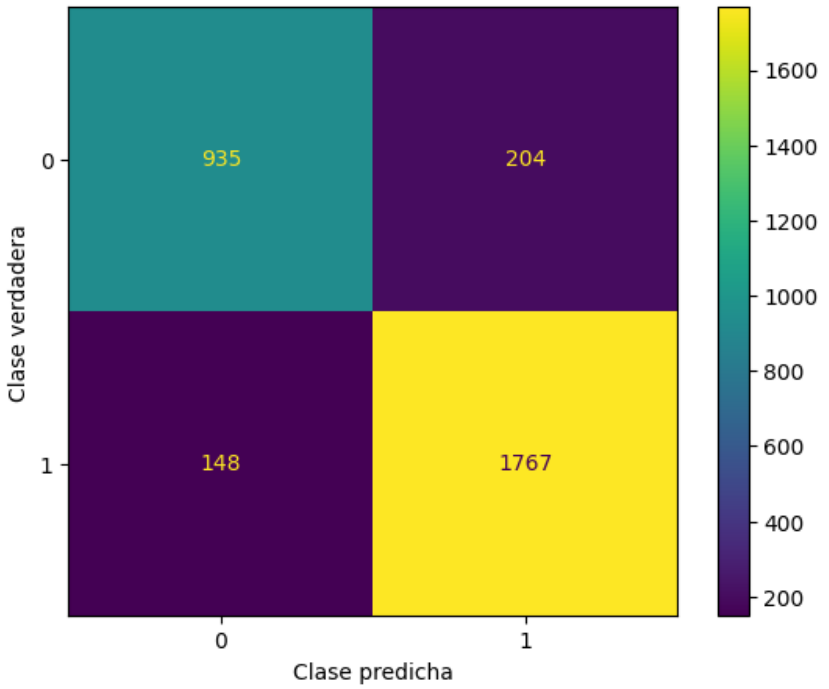


Figura 7.8: Matriz de confusión para el conjunto de prueba del *output* tens_calibrado_200KHz.

7.3. Resumen e interpretación de los resultados

En las Tablas 7.1 y 7.2 recopilamos la precisión de los modelos, sobre el conjunto de validación, en el problema de regresión y clasificación (Secciones 7.1 y 7.2, respectivamente).

	Red neuronal	Gradient Boosting	XGBoost
Pn 50KHz (ARM)	96.74 %(60)	96.90 %(60)	97.06 %(60)
Pn 200KHz (ARM)	97.95 %(todos)	98.02 %(todos)	98.04 %(40)
tens_calibrado_50KHz (ARM)	96.38 %(60)	96.68 %(todos)	96.78 %(60)
tens_calibrado_200KHz (ARM)	96.62 %(todos)	96.82 %(todos)	96.99 %(todos)
Pn 50KHz (R^2)	0.83(60)	0.85(60)	0.86(60)
Pn 200KHz (R^2)	0.85(todos)	0.86(todos)	0.86(40)
tens_calibrado_50KHz (R^2)	0.69(60)	0.75(todos)	0.75(60)
tens_calibrado_200KHz (R^2)	0.91(todos)	0.92(todos)	0.92(todos)
Multirespuesta (ARM)	-	-	97.2 %
Multirespuesta (R^2)	-	-	0.85

Tabla 7.1: Precisión (ARM y R^2) del conjunto de validación para cada uno de los modelos con mejores hiperparámetros (problema de regresión). Los números en paréntesis indican el número de *features* seleccionados para el mejor modelo.

	Random Forest	Gradient Boosting	XGBoost
Pn 50KHz	85.9 %(todos)	85 %(todos)	86.34 %(todos)
Pn 200KHz	84.44 %(todos)	84.4 %(todos)	85.19 %(todos)
tens_calibrado_50KHz	80.84 %(todos)	79.12 %(60)	80.9 %(todos)
tens_calibrado_200KHz	87.37 %(todos)	87.51 %(todos)	87.98 %(todos)

Tabla 7.2: Precisión (*accuracy*) del conjunto de validación para cada uno de los modelos con mejores hiperparámetros (problema de clasificación).

De las Tablas podemos apreciar que el mejor modelo es, en ambos problemas, el XGBoost. La precisión de este modelo en el conjunto de entrenamiento es, en todos los *outputs*, cercano a 100 % (*accuracy* y ARM) o a 1

(R^2), lo que indica que XGBoost sobreajusta los datos de entrenamiento de tal manera que se obtiene una predicción casi perfecta para los datos vistos por el modelo.

Por otro lado, vemos que el problema multirrespuesta se comporta de manera similar a los problemas con una sola variable de respuesta, puesto que obtenemos, para el primero, un ARM de 97.2 % y un R^2 de 0.85, mientras que, para los segundos, la media de los ARM es 97.22 % y la media de los coeficientes R^2, 0.8475.

Con respecto a los *features* seleccionados, en el problema de regresión, a menudo el mejor modelo lleva asociado el filtrado de *features* poco correlados con el *target*, de hecho es frecuente la selección de los 60 mejores (son 85 en total, una vez codificadas las variables categóricas). Por el contrario, en el problema de clasificación, en todos los casos excepto uno el mejor modelo corresponde a la inclusión de todos los *features*. Esto nos indica que la adición de todas estas variables generan ruido o errores que se traducen, por un lado, en peores predicciones de los valores de los parámetros de calibrado y, por otro lado, en una ganancia de información sobre la predictibilidad del calibrado.

Pn_50KHz	Pn_200KHz	tens_calibrado_50KHz	tens_calibrado 200KHz
Tipo_sonda Disco Fuji	Lote_Sonda 46436	Tipo_sonda Disco Fuji	Tipo_sonda Disco Fuji
num_iteraciones 50KHz	Tipo_sonda Disco Noliac	Tipo_sonda Disco Zibo	Lote_Sonda 16033QR
Tipo_Resina_SND-CAST(FLEXIBLE)	Lote_Sonda RW3	Lote_Sonda 47100	Tipo_base_PLAST-YECT
Lote_Sonda 47169	RX_200KHz cte_dev	Lote_Sonda N80	Equipo_cuba PCMARI-NE190
Lote_Sonda 46436	Valor_trans con_carga_200KHz	Lote_Sonda RW-M1	Lote_Sonda N77MIX

Tabla 7.3: 5 *features* con mayor importancia para cada *target*.

Recordemos que, en el problema de clasificación, también hemos estudiado cuáles son los *features* que mayor importancia tienen sobre los mo-

delos, con el objetivo de tener una clara referencia sobre aquellos *features* que pueden tener mayor poder de decisión sobre la posibilidad de fallo del calibrado (Sección 7.2). Los resultados vienen recogidos en la Tabla 7.3. No es de extrañar que los cuatro *outputs* se vean tremendamente influenciados por Tipo_sonda y Lote_Sonda, ya que estos parámetros de calibrado tienen una fuerte correlación con los cuatro *outputs* (Véase Figura 4.6). Sin embargo, vemos ciertas discrepancias entre las correlaciones y las importancias. Por ejemplo, las frecuencias de resonancia (*Resonancia_baja_frecuencia* y *Resonancia_alta_frecuencia*) y los parámetros medidos en el test en minicuba (*Media_TX_50KHz_minicuba, Media_RX_50KHz_minicuba, Media_TX_200KHz_minicuba* y *Media_RX_200KHz_minicuba*) cuentan con una fuerte correlación con todos los *targets* (Figura 4.5), pero ninguno de ellos se encuentra entre los *features* con mayor importancia. De igual manera, existen *features* con una gran importancia sobre el *target* débilmente correlados con él, por ejemplo *RX_200KHz cte_dev*, el cual es el cuarto *feature* con mayor importancia sobre el *target Pn_200KHz*, pero cuya correlación es de solo -0.12. Entonces, ¿a qué se debe esta discrepancia?

Como ya se ha comentado, las importancias en un árbol de decisión se miden como la reducción de la función de pérdida tras hacer los *splits* basados en los nodos representados por un determinado *feature*. Esto quiere decir que una variable de entrada puede estar poco correlada con la variable de salida pero puede ser importante si ayuda a mejorar la precisión del modelo. Por ejemplo, puede darse la posibilidad que, por si solo, un *input* no esté significativamente correlado con un *output* pero, combinado con otros *features*, puede proporcionar gran información sobre él. Asimismo, es posible que un *input* se encuentre fuertemente correlado con un *output*, pero que existan otros *features* igualmente correlados con ese *input* y que estos enmascaren su importancia en la predictibilidad del modelo. Tal puede ser el caso de las frecuencias de resonancia y los parámetros de calibrado medidos en el test de minicuba, ya que dichos parámetros cuentan con una correlación considerable sobre casi todos los demás *features* (consúltense Figuras 4.5 y 4.6)[4].

Finalmente, para el problema de clasificación, hemos obtenidos las matrices de confusión para cada *output* (Figuras 7.5 − 7.8). De ellas podemos sacar

[4]Cabe mencionar también que las importancias se calculan en base al conjunto de entrenamiento que consideremos, por lo que pueden variar ligeramente de un conjunto a otro.

una conclusión clara: es más fácil predecir el calibrado "cercano" a la media (clase 1) que el calibrado que se desvía de la media (clase 0). Por ejemplo, para *Pn_50KHz*, la predicción correcta de la clase 1 se produce para 1925 boyas y, de la clase 0, para 730 boyas, un número claramente inferior. De hecho, para la clase 0, se producen 263 predicciones incorrectas (un 26.5 % del total) y, para la clase 1, 136 (un 6.6 % del total). A continuación, presentamos una tabla con las predicciones incorrectas para cada clase y cada *output*, junto al error de los *targets* para el mejor modelo (XGBoost):

	Clase 0	**Clase 1**	Error $(100 - accuracy)$
Pn 50KHz	263(26.5 %)	136(6.6 %)	13.66 %
Pn 200KHz	295(30.89 %)	160(7.62 %)	14.81 %
tens_calibrado_50KHz	392(37.87 %)	201(9.96 %)	19.1 %
tens_calibrado_200KHz	204(17.91 %)	148(7.72 %)	11.96 %

Tabla 7.4: Predicciones incorrectas para cada clase y cada *output*, junto al error de cada *target*.

De la Tabla 7.4 podemos deducir que, efectivamente, la clase 0 es más difícil de predecir, sobre todo para tens_calibrado_50KHz, el cual cuenta con 392 boyas con una predicción incorrecta, un número mucho mayor que los demás *targets*. De hecho, este parámetro de calibrado es el que mayor error cuenta de los cuatro, con un 19.1 %. Esto resultado está en concordancia, curiosamente, con el R^2 para este output en el problema de regresión, en donde veíamos que tenía el peor resultado de entre los cuatro targets, 0.75, frente a 0.86, 0.86 y 0.92 (Tabla 7.1). Definitivamente, la información de este parámetro es más difícil de capturar, al menos para las variables de entrada con las que contamos.

7.4. ¿Se ha cumplido el objetivo de la empresa?

El objetivo de la empresa requerido al inicio de este PFM es "predecir el resultado de los cuatro parámetros del proceso de calibrado de nuestras boyas con un error menor al 15 % con los datos obtenidos a lo largo del proceso productivo".

Podemos decir que, por un lado, el objetivo se ha cumplido con creces, ya que, en el problema de regresión, hemos podido predecir los valores de los *outputs* con una precisión del 97 %, un valor muy superior al 85 % requerido. Por otro lado, a la empresa también le interesa saber si, cuando el calibrado se desvía de la media, dicha desviación es esperada o no, y por ello hemos establecido el problema de clasificación. En dicho problema, el umbral del 85 % es superado para todos los parámetros de salida excepto para *tens_calibrado_50KHz*, donde sólo podemos predecir correctamente el 80.9 % de las boyas del conjunto de prueba. Esto significa que existen ciertos factores sobre el calibrado de la boya que influencia la tensión de calibrado a 50 KHz y que no afectan o, al menos, que afectan en menor medida al resto de los *targets*. En suma, concluimos que el objetivo de la empresa ha sido superado, pero nuestro estudio cuenta todavía con ciertas limitaciones a la hora de clasificar el calibrado de las boyas, sobre todo a lo que concierne el parámetro *tens_calibrado_50KHz*.

Conclusiones

En la actualidad, las empresas cuentan con grandes cantidades de datos que no son aprovechados al máximo. La boya M3iGO, desarrollada por Marine Instruments (Capítulo 1), es una boya satelital diseñada principalmente para la pesca de atún, la cual cuenta con un proceso de calibrado complejo donde influyen muchos factores (Capítulo 2). Por ello, el manejo de los datos históricos de la M3iGO es un recurso importante que puede proporcionar valor a la empresa.

Este trabajo ha consistido en la aplicación de técnicas estadísticas y de *machine learning* para la predicción de los datos de calibrado de la M3iGO. En él, hemos explorado y analizado diversas técnicas y algoritmos con el objetivo de mejorar la toma de decisiones y optimizar los procesos empresariales. A lo largo del estudio, hemos obtenido resultados significativos y hemos realizado varias conclusiones clave que resumiremos a continuación:

1 - **Los datos son muy importantes**. Los datos son el activo más valioso en el proceso de predicción que ofrece el *machine learning*. Los modelos de *machine learning* requieren una gran cantidad de datos de entrenamiento para poder aprender y generalizar correctamente. Hemos observado que la calidad, la disponibilidad y la relevancia de los datos son cruciales para obtener modelos precisos y confiables, y que datos incompletos, inconsistentes o erróneos deben ser eliminados. Por ello, es esencial invertir en la recopilación y limpieza de datos para maximizar la efectividad del modelo.

Durante el proceso de transformación de datos, hemos utilizado técnicas como la normalización, la codificación de variables categóricas y la selección de k mejores *features*. Estas técnicas nos han permitido mejorar la calidad de los datos y reducir el ruido, lo que ha tenido un impacto positivo en la precisión de nuestros modelos. Por ejemplo, para el problema de regresión,

muchos de nuestros modelos han requerido del filtrado de variables poco correladas con los *targets* para encontrar los hiperparámetros óptimos (Tabla 7.1).

2 - Tanto el análisis de datos como las técnicas de *machine learning* nos ha ayudado a descubrir patrones y relaciones ocultas en los datos. En muchas ocasiones, la información valiosa se encuentra oculta en ellos y es difícil de detectar a simple vista. Por ejemplo, ¿qué influencia tiene el tipo de sonda sobre la tensión de calibrado en 50 KHz? Para responder a esta pregunta, hemos llevado a cabo, por un lado, un estudio de las correlaciones entre variables (Sección 4.6) y, por otro lado, un *ranking* de los *features* con mayor importancia sobre cada *target* (Tabla 7.4). Hemos encontrado diversas discrepancias. Por ejemplo, las frecuencias de resonancia se encuentran fuertemente correlacionados con todos los *outputs* pero, a la hora de contribuir al modelo predictivo, no tienen demasiada influencia. Deducimos, por tanto, que existe una distinguida diferencia entre la significancia estadística de las variables y su importancia en el modelo de *machine learning*.

3 - La **selección del modelo adecuado** es un aspecto crítico en nuestro trabajo. Existen diferentes algoritmos y modelos de *machine learning* disponibles, y por ello es fundamental evaluar y comparar varios de ellos para determinar cuál ofrece el mejor rendimiento en términos de precisión, tiempo de entrenamiento y capacidad de generalización. En este trabajo, se aplicaron diversos modelos de aprendizaje supervisado para el problema de regresión y el de clasificación. Aun cuando la elección del modelo depende, a menudo, del tipo de problema y de la naturaleza de los datos, para nuestros problemas de regresión y clasificación, el modelo XGBoost (Sección 3.6.2) es el que mejores resultados proporciona (véase Tablas 7.1 y 7.2), por lo que pensamos que la naturaleza de los datos (27 *inputs* y 31308 muestras) tiene mayor peso. La red neuronal (Sección 3.4), pese a su popularidad, nos ha proporcionado peores predicciones y en mucho mayor tiempo de computación, debido al gran número de hiperparámetros con los que cuenta.

4 - La **evaluación**, **validación** y **entrenamiento** *rigurosos* de los modelos es esencial. Para garantizar la confiabilidad de nuestras predicciones, hemos dividido los datos en conjuntos de entrenamiento, validación y prueba. Nuestra división de datos ha sido 90 % conjunto entrenamiento/validación, 10 % conjunto de prueba. Para comprobar que el modelo funciona y que los

resultados no dependen de una división de datos puntual, hemos procedido a aplicar la **validación cruzada** (Sección 5.3.1) a lo largo de 10 iteraciones. Mediante dicho proceso, hemos encontrado resultados consistentes, tanto en el conjunto de validación como en el de prueba, aunque el conjunto de entrenamiento se ve muy sobreajustado (véase Secciones 7.1 y 7.2).

5 - El *machine learning* es una herramienta muy útil para **la toma de decisiones** en tiempo real. Los modelos de *machine learning* pueden ser implementados en sistemas informáticos para realizar predicciones *in situ*. Esto permite a la empresa testar un calibrado en tiempo real y determinar, en un tiempo menor al tiempo de calibrado de cada boya ($\sim 2-3$ min), los outputs predichos y si dicho calibrado se sale fuera de lo esperado.

6 - **Resultados positivos**. El enfoque de este trabajo ha sido el problema de regresión y el problema de clasificación. En el primero, buscamos aproximarnos lo máximo posible a los valores numéricos de los *targets*, lo cual hemos logrado con una precisión media del 97 % (Tabla 7.1). En el segundo, buscamos clasificar los calibrados de las boyas como "normales" o "anómalos", y hemos logrado superar el umbral del 85 % propuesto por la empresa en *accuracy* para todos los *targets*, excepto para *tens_calibrado_50KHz*, el cual hemos obtenido un 80.9 % con el mejor modelo.

Este estudio ha proporcionado información valiosa a la empresa para tomar decisiones informadas sobre el calibrado de una cierta boya, como puede ser la presencia de ciertos factores externos que no se estén tomando en cuenta. Un ejemplo de ello es, de nuevo, el *output tens_calibrado_50KHz*, el cual cuenta, en el problema de regresión, con un $R^2 = 0.75$, lo que nos dice que una considerable parte de su variabilidad (un 25 %) no puede explicarse por la variabilidad de los *inputs*. Esto, a su vez, nos indica que algo más debe causar dicha variabilidad, y no se está teniendo en cuenta con los *features* considerados.

En resumen, este proyecto sobre la estimación de los parámetros de calibrado de la boya M3iGO ha demostrado ser valioso y prometedor, permitiendo extraer información de datos muy complejos y sin contar con un modelo matemático explícito, subyacente a todos ellos. Los modelos construidos han logrado un buen rendimiento predictivo y proporcionan información relevante para la toma de decisiones. Sin embargo, es necesario continuar mejorando

y actualizando los modelos a medida que se disponga de nuevos datos.

Bibliografía

[1] C. M. Bishop and N. M. Nasrabadi, *Pattern recognition and machine learning*, Springer, 2006.

[2] B. A. Block, H. Dewar, S. B. Blackwell, T. D. Williams, E. D. Prince, C. J. Farwell, A. Boustany, S. L. Teo, A. Seitz, A. Walli, et al., *Migratory movements, depth preferences, and thermal biology of Atlantic bluefin tuna*, Science, 293 (2001), pp. 1310–1314.

[3] L. Breiman, *Random Forest, vol. 45*, Mach Learn, 1 (2001).

[4] L. Breiman, J. Friedman, R. Olshen, and C. Stone, *Classification and regression trees–crc press*, Boca Raton, Florida, (1984).

[5] P. Bruce, A. Bruce, and P. Gedeck, *Practical statistics for data scientists: 50+ essential concepts using R and Python*, O'Reilly Media, 2020.

[6] A. E. Bryson, *Applied optimal control: optimization, estimation and control*, CRC Press, 1975.

[7] B. Buchanan, G. Sutherland, and E. A. Feigenbaum, *Heuristic DENDRAL: A program for generating explanatory hypotheses*, Organic Chemistry, (1969), p. 30.

[8] B. G. Buchanan and R. G. Smith, *Fundamentals of expert systems*, Annual review of computer science, 3 (1988), pp. 23–58.

[9] J. Bughin, E. Hazan, S. Ramaswamy, M. Chui, T. Allas, P. Dahlström, N. Henke, and M. Trench, *Artificial intelligence: the next digital frontier? McKinsey Global Institute*, 2017.

[10] T. Chen and C. Guestrin, *XGBoost: A scalable tree boosting system*, in Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining, 2016, pp. 785–794.

[11] T. Chen, T. He, M. Benesty, V. Khotilovich, Y. Tang, H. Cho, K. Chen, R. Mitchell, I. Cano, T. Zhou, et al., *Xgboost: extreme gradient boosting*, R package version 0.4-2, 1 (2015), pp. 1–4.

[12] G. Cybenko, *Approximation by superpositions of a sigmoidal function*, Mathematics of control, signals and systems, 2 (1989), pp. 303–314.

[13] T. K. Davies, C. C. Mees, and E. Milner-Gulland, *The past, present and future use of drifting fish aggregating devices (FADs) in the Indian Ocean*, Marine policy, 45 (2014), pp. 163–170.

[14] B. Efron, *Bootstrap Methods: Another Look at the Jackknife*, Ann. Statist., 7 (1979), pp. 1–26.

[15] T. Fawcett and F. Provost, *Data Science for Business*, O'Reilly, 2013.

[16] J. H. Friedman, *Greedy function approximation: a gradient boosting machine*, Annals of statistics, (2001), pp. 1189–1232.

[17] J. R. García, *Notas de Optimización y Control*.

[18] C. F. Gauss, *Theoria motus corporum coelestium in sectionibus conicis solem ambientium auctore Carolo Friderico Gauss*, sumtibus Frid. Perthes et IH Besser, 1809.

[19] A. Géron, *Hands-on machine learning with scikit-learn and tensorflow: Concepts*, Tools, and Techniques to build intelligent systems, (2017).

[20] I. Goodfellow, Y. Bengio, and A. Courville, *Deep learning*, MIT press, 2016.

[21] J. Grus, *Data science from scratch: first principles with Python*, O'Reilly Media, 2019.

[22] T. Hastie, R. Tibshirani, J. H. Friedman, and J. H. Friedman, *The elements of statistical learning: data mining, inference, and prediction*, vol. 2, Springer, 2009.

[23] C. F. Higham and D. J. Higham, *Deep learning: An introduction for applied mathematicians*, Siam review, 61 (2019), pp. 860–891.

[24] J. R. Hunter and C. T. Mitchell, *Association of fishes with flotsam in the offshore waters of Central America*, Fish. Bull, 66 (1967), pp. 13–29.

[25] G. James, D. Witten, T. Hastie, and R. Tibshirani, *An introduction to statistical learning*, vol. 112, Springer, 2013.

[26] E. Josse, L. Dagorn, and A. Bertrand, *Typology and behaviour of tuna aggregations around fish aggregating devices from acoustic surveys in French Polynesia*, Aquatic Living Resources, 13 (2000), pp. 183–192.

[27] G. Ke, Q. Meng, T. Finley, T. Wang, W. Chen, W. Ma, Q. Ye, and T.-Y. Liu, *Lightgbm: A highly efficient gradient boosting decision tree*, Advances in neural information processing systems, 30 (2017).

[28] D. P. Kingma and J. Ba, *Adam: A method for stochastic optimization*, arXiv preprint arXiv:1412.6980, (2014).

[29] A. M. Legendre, *Nouvelles Methodes pour la determination des Orbites des Cometes*, Paris: Courcier, (1806).

[30] W.-Y. Loh, *Classification and regression trees*, Wiley interdisciplinary reviews: data mining and knowledge discovery, 1 (2011), pp. 14–23.

[31] M. S. Love, M. Yoklavich, and D. M. Schroeder, *Demersal fish assemblages in the Southern California Bight based on visual surveys in deep water*, Environmental Biology of Fishes, 84 (2009), pp. 55–68.

[32] W. McKinney, *Python for Data Analysis*, .º'Reilly Media, Inc.", 2022.

[33] V. Mirjalili and S. Raschka, *Python machine learning*, Marcombo, 2020.

[34] H. Motoda and H. Liu, *Data reduction: feature selection*, in Handbook of data mining and knowledge discovery, Springer, 2002, pp. 208–213.

[35] K. P. Murphy, *Machine learning: a probabilistic perspective*, MIT press, 2012.

[36] D. Nielsen, *Tree boosting with XGBoost*, Norwegian University of Science and Technology, (2016).

[37] L. Pierson, *Data science for dummies*, John Wiley & Sons, 2021.

[38] K. Potdar, T. S. Pardawala, and C. D. Pai, *A comparative study of categorical variable encoding techniques for neural network classifiers*, International journal of computer applications, 175 (2017), pp. 7–9.

[39] S. Raschka, *Python machine learning*, Packt publishing ltd, 2015.

[40] H. Robbins and S. Monro, *A stochastic approximation method*, The annals of mathematical statistics, (1951), pp. 400–407.

[41] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, *Learning internal representations by error propagation*, tech. rep., California Univ San Diego La Jolla Inst for Cognitive Science, 1985.

[42] ——, *Learning representations by back-propagating errors*, nature, 323 (1986), pp. 533–536.

[43] C. E. Shannon, *A chess-playing machine*, Scientific American, 182 (1950), pp. 48–51.

[44] J. W. Tukey et al., *Exploratory data analysis*, vol. 2, Reading, MA, 1977.

[45] J. VanderPlas, *Python data science handbook: Essential tools for working with data*, .º'Reilly Media, Inc.", 2016.

[46] J. Weizenbaum, *ELIZA: a computer program for the study of natural language communication between man and machine*, Communications of the ACM, 9 (1966), pp. 36–45.

[47] Z. Wen, J. Shi, B. He, J. Chen, K. Ramamohanarao, and Q. Li, *Exploiting GPUs for efficient gradient boosting decision tree training*, IEEE Transactions on Parallel and Distributed Systems, 30 (2019), pp. 2706–2717.

[48] P. Werbos, *Beyond regression: New tools for prediction and analysis in the behavioral sciences (Doctoral dissertation, Harvard University)*, 1974.

[49] A. Zheng and A. Casari, *Feature engineering for machine learning: principles and techniques for data scientists*, .º'Reilly Media, Inc.", 2018.

[50] Z.-H. Zhou, *Ensemble Methods: Foundations and Algorithms*, 2012.

Apéndice A

Códigos de Python

aed.py

```
import pandas as pd
pd.options.mode.chained_assignment = None
import matplotlib.pyplot as plt
import numpy as np
from scipy import stats
import seaborn as sn
import pingouin as pg  # Pingouin is an open-source
   statistical
# package written in Python 3 and based mostly on
   Pandas and NumPy. Some of its main features are
   listed below.

""" En este script se procede al análisis exploratorio
    de los datos """
def cramers_stat(contingency_table):
""" Tabla de contigencia entre variables categóricas
   """
  chi2 = stats.chi2_contingency(contingency_table)
  n = contingency_table.values.sum()
  return np.round(np.sqrt(chi2[0] /(n*(np.min(
     contingency_table.shape)-1))), 2)

def filter_range_missing_data(col_interest, col_num,
   min_max, df):
""" Filtramos las filas por rango y eliminamos
   aquellas que contienen datos que faltan"""
df.dropna(axis=0, how='any', inplace=True)
""" for x in col_num:
```

```
        q1 = df[x].quantile(0.25)
        q3 = df[x].quantile(0.75)
        iqr = q3 - q1
        # Apply filter with respect to IQR, including
            optional whiskers
        filter = (df[x]>= q1 - 1.5*iqr) & (df[x]<= q3 + 1.5*
            iqr)
        df.loc[~filter,:].to_excel('C:/Users/carko/OneDrive/
            Documentos/Academico/M2i/TFM/tabla_outliers_IQR_'
            + x  + '.xlsx') """

        print('Tamaño tras eliminar filas con valores que
            faltan: ', np.array(df).shape)
        for x, interv in zip(col_interest, min_max):
        if (x in col_num):
        df.drop(df.loc[~df[x].between(interv[0],interv[1])
            ,:].index,inplace = True, axis = 0)
        else:
        df.drop(df.loc[~(df[x].isin(interv)),:].index,inplace
             = True, axis = 0)
        print('Tamaño tras filtrar por rango: ', np.array(df)
            .shape)

        def corr_function(df,row,col,func):
        """ Define la función de correlación que vayamos a
            utilizar,
        dependiendo si tratamos con datos de tipo numérico o
            categórico"""
        if func == 'Cramer':
        return cramers_stat(pd.crosstab(df[row],df[col]))
        elif func == 'ANOVA':
        aov =  pg.anova(dv=row, between=col, data=df,
            detailed=True)
        return aov.SS[0]/(aov.SS[0] + aov.SS[1])
        elif func == 'Pearson':
        if ((stats.pearsonr(df[row], df[col])[1]>0.1)&(stats.
            pearsonr(df[row], df[col])[0]>0.1)):
        print('Cuidado! Grande p-value para (' + row + ', '+
            col + ')')
        return stats.pearsonr(df[row], df[col])[0]

        def visualize_corr(df, row_type, col_type, func, args
            =False):
        """ Calcula y hace el plot de las tablas de
            correlación:
        una para las variables categóricas, otra para las
```

```
        numéricas con las categóricas, y otra
    para las numéricas."""
    table = pd.DataFrame([[corr_function(df,row,col,func)
        for col in col_type] for row in row_type],
    columns=col_type, index=row_type)
    if args:
    sn.heatmap(table,annot=True, cmap = "RdBu_r")
    plt.show()

    def corr(col_num, col_cat, df,args=False):
    """ Visualiza tabla de correlaciones """
    """ Cramer: >0.25 significativo """
    visualize_corr(df,col_cat, col_cat, 'Cramer',args)
    """ Anova: >0.16 significativo """
    visualize_corr(df, col_num , col_cat, 'ANOVA', args)
    visualize_corr(df, col_num, col_num, 'Pearson',args)

    def delete_outliers(df):
    """ Borra outliers de los scatter plots. ATENCIÓN!
        Esta función deberá ser modificada con un dataset
        diferente  """
    """ pd.concat([df.loc[(df['Resonancia_baja_frecuencia
        '].between(50.5, 51)) &
    (df['Resonancia_alta_frecuencia'].between(197, 198)),
        :],
    df.loc[(df['Media_TX_50KHz_minicuba'].between(1000,
        1200)) & (df['Media_RX_50KHz_minicuba'].between
        (700, 800)),
    :],
    df.loc[(df['Media_RX_200KHz_minicuba'].between(1000,
        1100)) & (df['Media_RX_50KHz_minicuba'].between
        (500, 600)),
    :],
    df.loc[(df['Valor_trans_con_carga_50KHz'].between
        (0.75e2, 0.8e2)),:],
    df.loc[(df['Valor_trans_con_carga_50KHz'].between
        (1.050e2, 1.075e2)) &
    (df['Media_TX_200KHz_minicuba'].between(1300, 1400)),
        :],
    df.loc[(df['tens_calibrado_50KHz'].between(600, 900))
        ,:],
    df.loc[(df['tens_calibrado_200KHz'].between(250, 275)
        ),:],
    df.loc[(df['Vpp_200KHz'].between(1650, 1800)), :]
    ],axis=0).to_excel('C:/Users/carko/OneDrive/
        Documentos/Academico/M2i/TFM/
```

```
        tabla_outliers_extremos.xlsx') """
    df.loc[(df['Resonancia_baja_frecuencia'].between
        (50.5, 51)) &
    (df['Resonancia_alta_frecuencia'].between(197, 198)),
         'Resonancia_baja_frecuencia'] = np.nan
    df.loc[(df['Media_TX_50KHz_minicuba'].between(1000,
        1200)) & (df['Media_RX_50KHz_minicuba'].between
        (700, 800)),
    'Media_TX_50KHz_minicuba'] = np.nan
    df.loc[(df['Media_RX_200KHz_minicuba'].between(1000,
        1100)) & (df['Media_RX_50KHz_minicuba'].between
        (500, 600)),
    'Media_RX_50KHz_minicuba'] = np.nan
    df.loc[(df['Valor_trans_con_carga_50KHz'].between
        (0.75e2, 0.8e2)),'Valor_trans_con_carga_50KHz'] =
        np.nan
    df.loc[(df['Valor_trans_con_carga_50KHz'].between
        (1.050e2, 1.075e2)) &
    (df['Media_TX_200KHz_minicuba'].between(1300, 1400)),
         'Valor_trans_con_carga_50KHz'] = np.nan
    df.loc[(df['tens_calibrado_50KHz'].between(600, 900))
        ,'tens_calibrado_50KHz'] = np.nan
    df.loc[(df['tens_calibrado_200KHz'].between(250, 275)
        ),'tens_calibrado_200KHz'] = np.nan
    df.loc[(df['Vpp_200KHz'].between(1650, 1800)), '
        Vpp_200KHz'] = np.nan
    df.dropna(axis=0, how='any', inplace=True)
    print('Tamaño tras filtrar por outlier:', np.array(df
        ).shape)
    df['tens_calibrado_50KHz'].loc[df['
        tens_calibrado_50KHz'] == 350]+=2*np.random.normal
        (
    size=np.array(df['tens_calibrado_50KHz'].loc[df['
        tens_calibrado_50KHz'] == 350]).shape)
    df['tens_calibrado_50KHz'].loc[df['
        tens_calibrado_50KHz'] == 450]+=2*np.random.normal
        (
    size=np.array(df['tens_calibrado_50KHz'].loc[df['
        tens_calibrado_50KHz'] == 450]).shape)
    df['tens_calibrado_200KHz'].loc[df['
        tens_calibrado_200KHz'] == 150]+=2*np.random.
        normal(
    size=np.array(df['tens_calibrado_200KHz'].loc[df['
        tens_calibrado_200KHz'] == 150]).shape)

    def visualize_data(col_interest, col_interest_numeric
```

```
    , df, arg1 = False, arg2 = False, args3 = False):
"""  Visualizamos los histogramas/boxplots/scatter
    plots """
for x in col_interest:
if (x in col_interest_numeric):
""" df.boxplot(column = x).figure
plt.xticks(fontsize=14)
plt.yticks(fontsize=14)
plt.savefig(x + '.png')
plt.close() """
""" df.hist(column = [x],ec = 'black')
plt.savefig(x + '_hist.png')
plt.xticks(fontsize=14)
plt.yticks(fontsize=14)
plt.close() """
if arg1:
df[x].plot.hist(ec = 'black')
plt.xlabel(x)
plt.xticks(fontsize = 14)
plt.yticks(fontsize = 14)
plt.show()

if arg2:
df.plot.box(column= x)
plt.xticks(fontsize = 14)
plt.yticks(fontsize = 14)
plt.show()

else:

if arg1:
df[x].value_counts().plot(kind = 'bar',ec='black')
if len(df[x].unique())>20:
plt.xticks([])

else:
plt.xticks(fontsize = 14, rotation=0)
plt.yticks(fontsize = 14, rotation=0)
plt.ylabel(ylabel='Frecuencia',fontsize=14)
plt.xlabel(x,fontsize=14)
""" plt.savefig(x + '_hist.png')
plt.close() """
plt.show()

if (args3):
for i in col_interest_numeric:
```

```
        for j in col_interest_numeric:
        if (i>j):
        df.plot.scatter(x=i, y=j, s=7, c='blue')
        plt.xticks(fontsize=14)
        plt.yticks(fontsize=14)
        plt.show()

        def get_inputs(df,targets):
        """ Función que devuelve la tabla de inputs de los
            datos
        """
        X = df.loc[:, ~df.columns.isin(targets)]
        print('Tamaño de los inputs', X.shape)
        return X
```

func_M3iGO_clasif.py

```
    import numpy as np
    import pandas as pd
    import matplotlib.pyplot as plt
    from sklearn.preprocessing import MinMaxScaler,
        LabelEncoder, OneHotEncoder
    from sklearn.model_selection import train_test_split,
        cross_validate, GridSearchCV
    from sklearn.metrics import ConfusionMatrixDisplay,
        confusion_matrix
    from sklearn.ensemble import RandomForestClassifier,
        HistGradientBoostingClassifier
    from sklearn.pipeline import Pipeline
    from sklearn.compose import ColumnTransformer
    from sklearn.feature_selection import SelectKBest,
        f_classif
    from xgboost import XGBClassifier, to_graphviz
    from pipelinehelper import PipelineHelper


    """ Módulo del problema del problema de clasificación
        """

    def hyperparameter_tuning_transform(
        col_interest_categorical, col_interest_numeric):
    """ Optimización de los hiperparámetros considerando
        los 3 modelos seleccionados
```

```
    para el problema de clasificación: Random Forest,
       Gradient Boosting y XGBoost
    """
    mlp1 = RandomForestClassifier(random_state=0)
    mlp2 = HistGradientBoostingClassifier(random_state=0)
    mlp3 = XGBClassifier(random_state=0)
    numeric_transformer = Pipeline(steps=[('scaler',
       MinMaxScaler())])
    categorical_transformer = Pipeline(steps=[('onehot',
       OneHotEncoder(handle_unknown = 'ignore'))])
    preprocessor =  ColumnTransformer(transformers=[
    ('cat', categorical_transformer,
       col_interest_categorical),
    ('num', numeric_transformer, col_interest_numeric)
    ])

    pipe = Pipeline(steps=[('preprocessor', preprocessor),
    ('kbest', SelectKBest(f_classif)),
    ('classifier',PipelineHelper([('hist', mlp2),('rf',
       mlp1), ('xgb',mlp3)]))
    ])
    params = {
      'classifier__selected_model': pipe.named_steps['
         classifier'].generate({
        'rf__n_estimators' : [10, 50, 100, 250],
        'rf__max_depth' : [30, 50, 70, 100, None],
        'rf__min_samples_leaf':  [2, 4, 6, 8, 10],
        'rf__min_samples_split':  [2, 4, 6, 8, 10],
        'rf__bootstrap':  [True],
        'hist__learning_rate' : [0.05,0.1,0.2],
        'hist__max_iter' : [30,50,70,100],
        'hist__max_depth' : [10, 20, 30, 50, 70, 100, None
           ],
        'hist__l2_regularization':
           [0.1,0.2,0.3,0.5,0.7,1.5],
        'xgb__n_estimators': [100,200,300],
        'xgb__learning_rate': [0.1,0.2,0.25],
        'xgb__max_depth': [20, 30, 50, 70, 100, None]
      }),
      'kbest__k': ['all', 5, 10, 20, 40, 60]
    }

    return pipe, params

    def training_data(X,y,col_interest_categorical,
       col_num_cat, col_interest_numeric):
```

```
    """ Entrenamiento de los modelos """
    col_interest_categorical = [item for item in
       col_interest_categorical if item not in col_num_cat]
    col_interest_numeric = col_interest_numeric +
       col_num_cat
    X[col_interest_categorical]=X[col_interest_categorical
       ].astype(str)  #Convertimos a String las var.
       categóricas
    y = LabelEncoder().fit_transform(y)
    X_spl, X_test, y_spl, y_test = train_test_split(X, y,
       test_size=0.1, shuffle=True)
    pipe, params = hyperparameter_tuning_transform(
       col_interest_categorical, col_interest_numeric)
    gridsearch = GridSearchCV(pipe,param_grid=params, cv =
       10, n_jobs=-1,verbose=5, scoring='accuracy',
    return_train_score=True,error_score='raise')
    gridsearch.fit(X_spl,y_spl)
    print(gridsearch.best_score_)
    print(gridsearch.best_params_)
    print('Inputs tras haber codificado las variables
       categóricas: ',
    gridsearch.best_estimator_.named_steps['kbest'].
       n_features_in_)
    scores = cross_validate(gridsearch.best_estimator_,
       X_spl, y_spl, cv=10,scoring=['accuracy'],
    verbose = 5, return_train_score=True, n_jobs=-1)
    print('Media para val. set: ', round(scores['
       test_accuracy'].mean(),2),
    ', con desv. estándar: ', round(scores['test_accuracy'
       ].std(),2))
    print('Media para training set: ', round(scores['
       train_accuracy'].mean(),2),
    ', con desv. estándar: ', round(scores['test_accuracy'
       ].std(),2))
    feature_importances(gridsearch,col_interest_categorical
       ,col_interest_numeric)
    #save_tree_model(gridsearch.best_estimator_.named_steps
       ['classifier'].selected_model)
    return X_test, y_test, gridsearch

    def test_validation(X_test, y_test, gridsearch):
    """ Calcula la precisión del modelo óptimo en el
       conjunto de test
    y calcula la matriz de confusión """
    X_test = gridsearch.best_estimator_.named_steps['
       preprocessor'].transform(X_test)
```

```
    X_test = gridsearch.best_estimator_.named_steps['kbest'
        ].transform(X_test)
    print('Accuracy for test set: %.2f' %(gridsearch.
        best_estimator_.named_steps['classifier'].score(
        X_test, y_test)))
    print('#####################################')
    cm = confusion_matrix(y_true=y_test, y_pred= gridsearch
        .best_estimator_.named_steps['classifier'].predict(
        X_test))
    #np.savetxt('pred.csv',np.vstack([y_test,gridsearch.
        best_estimator_.named_steps['classifier'].predict(
        X_test)]))
    disp  = ConfusionMatrixDisplay(confusion_matrix=cm,
        display_labels=gridsearch.classes_)
    disp.plot()
    plt.show()

    def main_clasif(targets, df, col_interest_categorical,
        col_num_cat,col_interest_numeric, X):
    """ Función principal del entrenamiento de los modelos
        """
    print('Empezando entrenamiento...')
    mean_std = [[48.078286,5.023674],[37.965892,2.661017],
    [401.77392,34.277406],[137.142216,20.983973]]
    for x, interv in zip(targets,mean_std):
    mask1 =  (np.abs(df[x]-interv[0]) < interv[1])
    mask2 =  (np.abs(df[x]-interv[0]) >= interv[1])
    df.loc[mask1, x] = 'YES'
    df.loc[mask2, x] = 'NO'
    #print(df[x].value_counts())
    print('Parámetro de salida: ', x)
    y = df[x]
    X_test, y_test, gridsearch = training_data(
    X,y,col_interest_categorical, col_num_cat,
        col_interest_numeric)
    test_validation(X_test, y_test, gridsearch)


    def save_tree_model(model):
    """ Guarda el primer árbol del mejor modelo XGBoost """
    graph = to_graphviz(model, num_trees=0, rankdir='LR')
    dot = graph.pipe(format='dot').decode('utf-8')
    with open('tree.dot', 'w') as f:
    f.write(dot)

    def feature_importances(gridsearch,
```

```
    col_interest_categorical,col_interest_numeric):
"""" Escribe por pantalla las importancias de cada
    feature sobre el modelo en orden descendente"""
preprocessor = gridsearch.best_estimator_.named_steps['
    preprocessor']
cat_cols = preprocessor.named_transformers_['cat'].
    named_steps['onehot'].get_feature_names_out(
    col_interest_categorical)
num_cols = preprocessor.named_transformers_['num'].
    named_steps['scaler'].get_feature_names_out(
    col_interest_numeric)
feature_names = np.concatenate([cat_cols, num_cols])
print(pd.DataFrame(gridsearch.best_estimator_.
    named_steps['classifier'].selected_model.
    feature_importances_,
index=feature_names,columns=['importance']).sort_values
    ('importance', ascending=False))
```

func_M3iGO_regr.py

```
from sklearn.preprocessing import MinMaxScaler,
    OneHotEncoder
from sklearn.model_selection import train_test_split,
    cross_validate, GridSearchCV
from sklearn.metrics import make_scorer,
    mean_absolute_percentage_error, r2_score
from sklearn.neural_network import MLPRegressor
from sklearn.ensemble import
    HistGradientBoostingRegressor
from sklearn.pipeline import Pipeline
from sklearn.compose import ColumnTransformer
from sklearn.feature_selection import SelectKBest,
    f_regression
from sklearn.multioutput import MultiOutputRegressor
from xgboost import XGBRegressor
from pipelinehelper import PipelineHelper
import os
import pickle

""" Módulo del problema del problema de regresión """


def acc_func(y_true, y_pred):
""" Función de porcentaje de acierto relativo """
```

```
    return 100-mean_absolute_percentage_error(y_true=y_true
        ,y_pred=y_pred)*100

    def hyperparameter_tuning_transform(
        col_interest_categorical, col_interest_numeric,
        multi_output):
    """ Optimización de los hiperparámetros considerando
        los 3 modelos seleccionados
    para el problema de regresión: Red Neuronal, Gradient
        Boosting y XGBoost. Utilizamos un
    Pipeline para implementar múltiples modelos y la
        transformación de datos pertinente en cada
    uno de ellos (codificación, normalización y reducción
        de la dimensionalidad)
    """
    mlp1 = MLPRegressor()
    mlp2 = HistGradientBoostingRegressor()
    mlp3 = XGBRegressor()

    numeric_transformer = Pipeline(steps=[('scaler',
        MinMaxScaler())])
    categorical_transformer = Pipeline(steps=[('onehot',
        OneHotEncoder(handle_unknown = 'ignore'))])
    preprocessor =  ColumnTransformer(transformers=[
    ('cat', categorical_transformer,
        col_interest_categorical),
    ('num', numeric_transformer, col_interest_numeric)
    ])
    if multi_output:

    pipe = Pipeline(steps=[('preprocessor', preprocessor),
    ('regressor', MultiOutputRegressor(estimator=mlp1))
    ])
    params = [
    {

      'regressor__estimator': [mlp3],
      'regressor__estimator__learning_rate':
          [0.1,0.2,0.25],
      'regressor__estimator__max_depth': [20, 30, 50, 70,
          100, None],
      'regressor__estimator__n_estimators': [100,200,300]
    }
    ]
    else:
    pipe = Pipeline(steps=[('preprocessor', preprocessor),
```

```
    ('kbest', SelectKBest(f_regression)),
    ('regressor',PipelineHelper([('hist', mlp2),('NN', mlp1
       ), ('xgb',mlp3)]))
    ])
    params = {
      'regressor__selected_model': pipe.named_steps['
         regressor'].generate({
        'NN__hidden_layer_sizes' : [(20,20), (30,30),
           (50,50), (100,100)],
        'NN__alpha' : [1e-2,1e-3,1e-4],
        'NN__activation':  ['relu', 'logistic', 'identity'
           ],
        'NN__learning_rate_init':  [0.005, 0.01, 0.03,
           0.05],
        'NN__learning_rate':  ['constant', 'invscaling', '
           adaptive'],
        'NN__max_iter':  [300, 500],
        'hist__learning_rate' : [0.05,0.1,0.2],
        'hist__max_iter' : [30,50,70,100],
        'hist__max_depth' : [10, 20, 30, 50, 70, 100, None
           ],
        'hist__l2_regularization':
           [0.1,0.2,0.3,0.5,0.7,1.5],
        'xgb__n_estimators': [100,200,300],
        'xgb__learning_rate': [0.1,0.2,0.25],
        'xgb__max_depth': [20, 30, 50, 70, 100, None]
      }),
      'kbest__k': ['all', 5, 10, 20, 40, 60]
    }


    return pipe, params


    def training_data(X,y,col_interest_categorical,
       col_num_cat, col_interest_numeric, multi_output):
    """ Entrenamos los modelos """
    col_interest_categorical = [item for item in
       col_interest_categorical if item not in col_num_cat]
    col_interest_numeric = col_interest_numeric +
       col_num_cat
    X[col_interest_categorical]=X[col_interest_categorical
       ].astype(str) #Convertimos a String las var.
       categóricas
    X_spl, X_test, y_spl, y_test = train_test_split(X, y,
       test_size=0.1, shuffle=True)
```

```
    custom_scorer = make_scorer(score_func=acc_func,
        greater_is_better=True)
    pipe, params = hyperparameter_tuning_transform(
        col_interest_categorical, col_interest_numeric,
        multi_output)

    gridsearch = GridSearchCV(pipe,param_grid=params, cv =
        10, n_jobs=-1,verbose=5, scoring=custom_scorer,
    return_train_score=True,error_score='raise')
    gridsearch.fit(X_spl,y_spl)
    """ Mejores hiperparámetros y mejor predicción"""
    print(gridsearch.best_score_)
    print(gridsearch.best_params_)
    best_estimator = gridsearch.best_estimator_
    scores = cross_validate(best_estimator, X_spl, y_spl,
        cv=10,scoring='r2', return_train_score=True,
    verbose = 5, n_jobs=-1)
    print('Media: ', round(scores['test_score'].mean(),2),
    '\n Desv. estándar: ', round(scores['test_score'].std()
        ,2))
    print('Media: ', round(scores['train_score'].mean(),2),
    '\n Desv. estándar: ', round(scores['train_score'].std
        (),2))

    return X_test, y_test, best_estimator

    def test_validation(X_test, y_test, estimator,
        multi_output):
    """ Calcula la precisión del modelo óptimo en el
        conjunto de test """
    X_test = estimator.named_steps['preprocessor'].
        transform(X_test)
    if multi_output==False:
    X_test = estimator.named_steps['kbest'].transform(
        X_test)
    y_pred = estimator.named_steps['regressor'].predict(
        X_test)
    print('R^2 para conjunto test: %.2f' %(r2_score(y_pred,
        y_test)))
    print('Acierto relativo para conjunto test: %.2f' %(
        acc_func(y_test,y_pred)))
    print('##############################')



    def main_regr(targets, df, col_interest_categorical,
```

```
    col_num_cat,col_interest_numeric, X, multi_output):
"""  Función principal del problema de regresión """
print('Empezando entrenamiento...')
if multi_output==True:
y = df[targets]
X_test, y_test, gridsearch = training_data(
X,y,col_interest_categorical, col_num_cat,
    col_interest_numeric, multi_output)
print('Entrenamiento finalizado!')
test_validation(X_test, y_test, gridsearch,
    multi_output)
else:
for x in targets:
print('Parámetro de salida: ', x)
y = df[x]
X_test, y_test, gridsearch = training_data(
X,y,col_interest_categorical, col_num_cat,
    col_interest_numeric, multi_output)
print('Entrenamiento finalizado!')
test_validation(X_test, y_test, gridsearch,
    multi_output)
```

main_M3iGO.py

```
import numpy as np
import pandas as pd
pd.options.mode.chained_assignment = None
import  func_M3iGO_clasif
import func_M3iGO_regr
import aed
import sys
def df_unidades_tipos(filename,targets, col_others,
    col_num_cat):
""" Crea el dataframe, declara el tipo de variables, y
    transforma los features en sus correctas unidades
    """
df = pd.read_excel(filename)
df[['Fecha_dia', 'Fecha_hora']] = df['Fecha_test_Ok'].
    str.split(pat=' ', n=1, expand=True)
df.drop(columns=['Fecha_hora', 'Fecha_test_Ok'],
    inplace=True)
col_interest = [item for item in df.columns if item not
     in col_others]
```

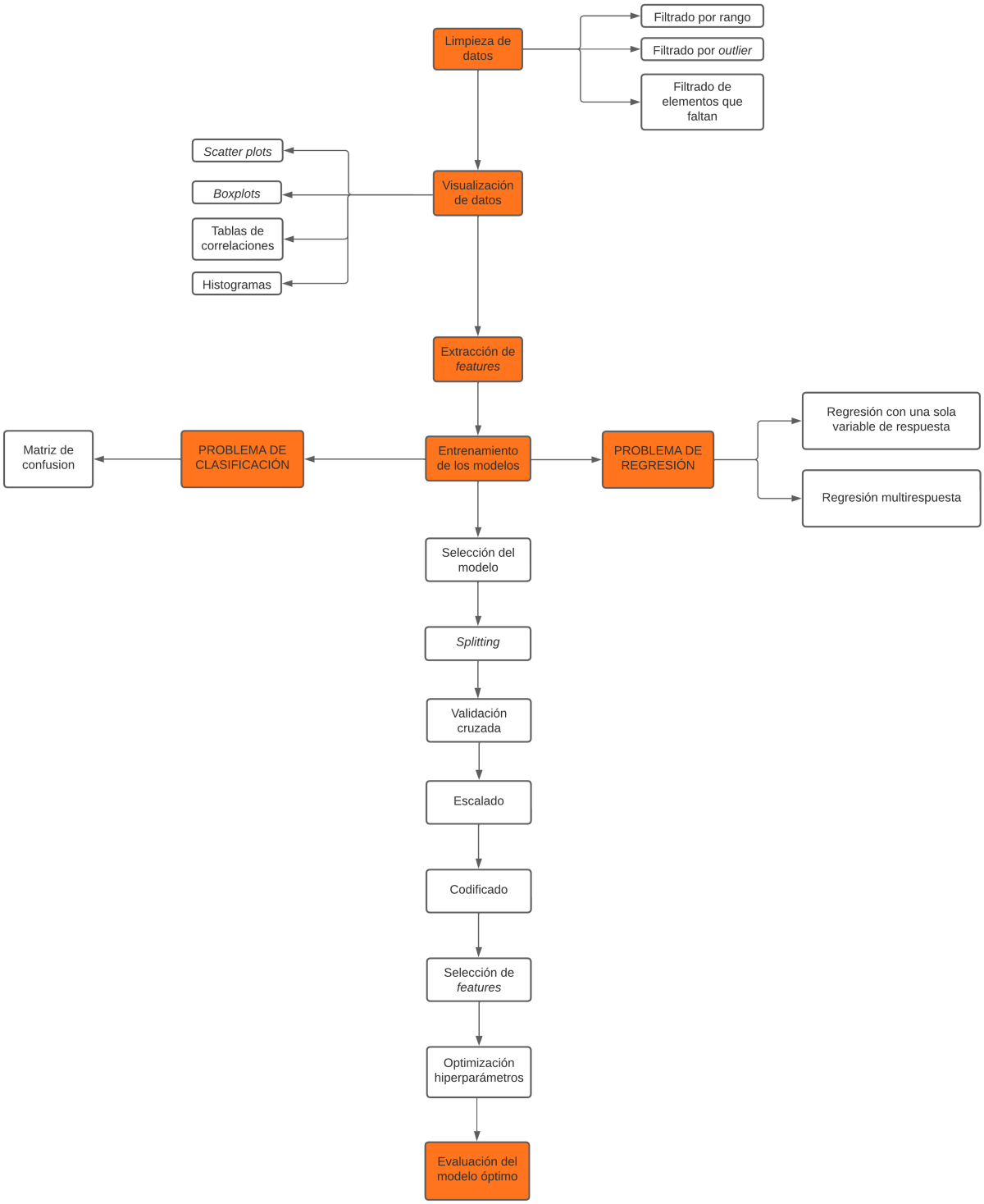
```
    col_interest_numeric = [item for item in col_interest
        if item in list(df.select_dtypes(include=np.number).
        columns)]
    col_interest_categorical = [item for item in
        col_interest if item not in col_interest_numeric]
    col_interest_categorical = col_interest_categorical +
        col_num_cat #Hacer las variables num. discretas
        categóricas
    col_interest_numeric = [item for item in
        col_interest_numeric if item not in col_num_cat]
    """ Transformamos las variables en sus correctas
        unidades """
    df[targets] = df[targets]/100000
    df[[ 'Resonancia_baja_frecuencia', '
        Resonancia_alta_frecuencia']]/=1000
    df[['RX 50KHz_cte_media', 'RX 50KHz_cte_dev', 'RX 200
        KHz_cte_media','RX 200KHz_cte_dev']] /=100000
    df[['Valor_trans_con_carga_50KHz','
        Valor_trans_con_carga_200KHz']] /=10000
    df['Temperatura']/=10
    return df, col_interest, col_interest_numeric,
        col_interest_categorical

    """ Programa principal """

    def main():
    """ targets,col_others,col_num_cat y filename tienen
        que introducirse por el usuario """
    targets = ['Pn_50KHz', 'Pn_200KHz', '
        tens_calibrado_50KHz', 'tens_calibrado_200KHz'] #
        Parámetros de salida
    col_others = ['FactoryCode','Fabricante_Resina','
        Rx_50KHz', 'Rx_200KHz','Baja_frecuencia',
    'Alta_frecuencia', 'Max_convolution','Temp_minicuba', '
        Canal', 'Fecha_dia']  # Variables que no nos
        interesan
    col_num_cat = ['Production_Version','
        num_interacciones_50KHz', 'num_interacciones_200KHz'
        , 'Tiempo_curado']  # Variables numéricas discretas
    filename = 'datos_marine_M3iGO.xlsx' #Nombre de la
        tabla desde la que queremos importar los datos

    df, col_interest, col_interest_numeric,
        col_interest_categorical = df_unidades_tipos(
        filename, targets, col_others, col_num_cat)
```

```
    """ Rangos de las variables. Se define por el usuario,
        aunque se recomienda no modificar los rangos,
    sino añadir o quitar elementos del array según las
        variables que se desee introducir"""
    min_max = [list(df['Production_Version'].unique()),['
        Disco Fuji', 'Disco Noliac', 'Disco Zibo'], list(df[
        'Lote_Sonda'].unique()),
    list(df['Tipo_Resina'].unique()), ['PLAST-YECT', '
        Valiant'], ['Suesa', 'Corgan','POMCEG'],
    ['PCMARINE184', 'PCMARINE187', 'PCMARINE190'],
    [1365, 1510], [1668, 2071], [30, 65], [30, 50], [167,
        900], [84, 350],
    [*range(0, 6)], [*range(0, 5)], [18, 28], [48.5, 51.5],
         [196, 204], [61.4, 113.8], [41.6, 77.3],
    [0.98500, 1.0000], [0.98500, 1.0000], [1000, 2500],
        [400, 1200], [900, 2000], [350, 1200],
    [50,900], [1.2,50], [60,900], [1.4,90], list(df['
        Tiempo_curado'].unique())]

    df = df[col_interest]
    print('Tamaño inicial: ', np.array(df).shape)
    print('Columnas en nuestro estudio: \n', col_interest)
    aed.filter_range_missing_data(col_interest,
        col_interest_numeric, min_max, df)
    aed.delete_outliers(df)
    """ Estas variables declaran si queremos que se
        visualicen los gráficos """
    visualiza_scatter_plots, visualiza_hist,
        visualiza_boxplots, visualiza_tablas_corr= False,
        False,False,False
    aed.visualize_data(col_interest, col_interest_numeric,
        df, visualiza_hist, visualiza_boxplots,
        visualiza_scatter_plots)
    aed.corr(col_interest_numeric, col_interest_categorical
        , df,visualiza_tablas_corr)
    X = aed.get_inputs(df,targets)
    col_interest_numeric = list(set(col_interest_numeric)-
        set(targets))
    #print(df[targets].describe())

    #sys.exit()

    regr, clf = False, True #Declarar como True si se
        quiere resolver el problema de regresión (regr) o
        clasificación (clf)
    multi_output = False #Esta variable declara si se
```

```
        quiere resolver el problema de regresión
        multirrespuesta

    """ Resuelve el problema de regresión y/o clasificación
        """
    if regr:
    func_M3iGO_regr.main_regr(targets, df,
        col_interest_categorical,col_num_cat,
        col_interest_numeric, X, multi_output)

    if clf:
    func_M3iGO_clasif.main_clasif(targets, df,
        col_interest_categorical,col_num_cat,
        col_interest_numeric, X)


    if __name__=='__main__':
    main()
```

Apéndice B

Workflow del trabajo


Limpieza de datos
Filtrado por rango
Filtrado por *outlier*
Filtrado de elementos que faltan
Visualización de datos
Scatter plots
Boxplots
Tablas de correlaciones
Histogramas
Extracción de *features*
Entrenamiento de los modelos
PROBLEMA DE CLASIFICACIÓN
Matriz de confusion
PROBLEMA DE REGRESIÓN
Regresión con una sola variable de respuesta
Regresión multirespuesta
Selección del modelo
Splitting
Validación cruzada
Escalado
Codificado
Selección de *features*
Optimización hiperparámetros
Evaluación del modelo óptimo