



Escuela Técnica Superior de
Ingeniería Informática

TRABAJO FIN DE GRADO

Autoencoders y Error de Reconstrucción para Detectar Muestras de Cáncer

Realizado por
Marco Tenorio Cortés

Para la obtención del título de
Grado en Ingeniería Informática - Tecnologías Informáticas

Dirigido por
Juan Antonio Nepomuceno Chamorro

En el departamento de
Lenguajes y Sistemas Informáticos

Convocatoria de junio, curso 2023/2024

Agradecimientos

Quiero agradecer especialmente:

*a mi pareja y mis amigos, por su incondicional apoyo a lo largo de todo este camino.
a Juan Antonio Nepomuceno, por todos y cada uno de los valiosos consejos que me ha dado
además de haberme transmitido su pasión por este campo.
y, por supuesto, a mi querida madre, por su eterna confianza en mí hasta el último de sus
días.*

Resumen

Incluya aquí un resumen de los aspectos generales de su trabajo, en español. Este Trabajo de Fin de Grado se centra en el desarrollo y evaluación de un modelo avanzado de autoencoder para la detección de anomalías en muestras de cáncer de mama. Utilizando el conjunto de datos Breast Cancer Wisconsin, se ha implementado un autoencoder profundo para identificar características distintivas de las muestras malignas a partir de sus patrones de reconstrucción.

La metodología aplicada abarca desde la preparación y normalización de los datos hasta la implementación de arquitecturas de redes neuronales profundas, haciendo uso de técnicas como la validación cruzada y ajuste de hiperparámetros para optimizar el modelo. El enfoque principal ha sido examinar cómo el error de reconstrucción del autoencoder puede servir como un indicador fiable de anomalías, potencialmente indicando presencia de tejido canceroso.

Los resultados obtenidos demuestran que los autoencoders son capaces de distinguir con una precisión significativa entre muestras normales y patológicas, superando en algunos casos a métodos tradicionales de aprendizaje automático. Además, se ha desarrollado una interfaz web para facilitar la visualización de los resultados del modelo, permitiendo una interpretación más intuitiva de los datos.

Este trabajo no solo proporciona una herramienta útil para la detección temprana de cáncer de mama, sino que también abre nuevas vías para la investigación futura, especialmente en la aplicación de autoencoders en otros tipos de datasets médicos y en la mejora de la interpretación de los modelos de aprendizaje profundo.

Palabras clave: Aprendizaje Automático, Aprendizaje Profundo, Cáncer de mama, Autoencoder, Redes Neuronales.

Abstract

This Final Degree Project focuses on the development and evaluation of an advanced autoencoder model for anomaly detection in breast cancer samples. Using the Breast Cancer Wisconsin dataset, a deep autoencoder has been implemented to identify distinctive features of malignant samples through their reconstruction patterns.

The methodology applied ranges from data preparation and normalization to the implementation of deep neural network architectures, utilizing techniques such as cross-validation and hyperparameter tuning to optimize the model. The primary focus has been to examine how the reconstruction error of the autoencoder can serve as a reliable indicator of anomalies, potentially signaling the presence of cancerous tissue.

The results demonstrate that autoencoders are capable of distinguishing with significant accuracy between normal and pathological samples, surpassing some traditional machine learning methods in certain instances. Additionally, a web interface has been developed to facilitate the visualization of the model's results, allowing for a more intuitive interpretation of the data.

This work not only provides a useful tool for early breast cancer detection but also opens new avenues for future research, particularly in applying autoencoders to other types of medical datasets and in improving the interpretation of deep learning models.

Keywords: Machine Learning, Deep Learning, Breast Cancer, Autoencoder, Neural Networks.

Índice general

1 Planificación	1
1.1. El reto planteado	1
1.2. Objetivos	1
1.2.1. Objetivos Docentes	1
1.2.2. Objetivos Técnicos	1
1.3. Metodología	2
1.3.1. Diseño del Estudio	2
1.3.2. Herramientas y Tecnologías	2
1.3.3. Metodología de Desarrollo	3
1.3.4. Demostración de Aplicación Práctica	3
1.3.5. Documentación y Reporte	3
1.4. Planificación	4
1.5. Presupuesto	4
1.5.1. Costes de personal	5
1.5.2. Costes de Software	6
1.5.3. Costes totales	6
2 Introducción	7
3 Estado del arte	9
3.1. Introducción	9
3.2. Clasificación con poca información	9
3.3. Detección de anomalías	10
3.3.1. Autoencoders para la detección de anomalías	11
3.4. Error de reconstrucción con autoencoders	11
3.5. Avances en el uso de autoencoders para la clasificación de tipos de cáncer	11
4 Metodos y Herramientas	13
4.1. Introducción	13
4.2. Introducción al Machine Learning	13
4.2.1. Árboles de decisión	13
4.2.2. Exploración de datos mediante Pandas	14
4.2.3. Filtrado de datos para el Modelo	14
4.2.4. Validación del modelo	15
4.2.5. Sobreajuste y Subajuste en Machine Learning	15
4.2.6. Random Forests	16
4.3. Introducción al Deep Learning	17
4.3.1. Perceptron, una única neurona	17
4.3.2. Redes Neuronales	19
4.3.3. Entrenamiento de la red	21
4.3.4. Sobreajuste y Subajuste en aprendizaje de Redes Neuronales .	22

4.3.5. Dropout and Batch Normalization	24
4.3.6. Clasificación Binaria	25
4.4. Autoencoders	26
4.4.1. Aplicaciones del autoencoder	27
4.4.2. Programación de Autoencoder	28
4.5. Medidas de evaluación	29
4.5.1. Curva de Entrenamiento y Evaluación en Función de la Pérdida	29
4.5.2. Matriz de Confusión	29
4.5.3. Gráfico de Precisión/Recall en función de los Umbrales	30
4.5.4. Curva ROC	31
5 Experimentos	33
5.1. Introducción	33
5.2. Configuración del escenario	33
5.2.1. Carga y Análisis de los datos	33
5.2.2. Recreación de escenario Anómalo	37
5.2.3. División del conjunto de datos	39
5.2.4. Normalización del conjunto de datos	40
5.2.5. Creación del Autoencoder	41
5.3. Marco experimental	45
5.3.1. Variación la cantidad de muestras anómalas	45
5.3.2. Variación del umbral ante el error de reconstrucción	46
5.3.3. Comparación con modelos de Machine Learning	46
5.3.4. Comparación con Entrenamiento exclusivo con muestras de clase Maligna	47
6 Resultados	49
6.1. Introducción	49
6.2. Resultados de los Experimentos	49
6.2.1. Variación la cantidad de muestras anómalas	49
6.2.2. Variación del umbral ante el error de reconstrucción	57
6.2.3. Comparación con modelos de Machine Learning	57
6.2.4. Comparación con Entrenamiento exclusivo con muestras de clase Maligna	60
7 Discusion	65
7.1. Análisis de los Resultados	65
7.1.1. Desempeño del Autoencoder	65
7.1.2. Comparación con Otros Modelos de Machine Learning	65
7.1.3. Impacto de la Selección del Umbral en la Detección de Anomalías	65
7.1.4. Variaciones el en modelo de aprendizaje	66
7.2. Limitaciones del Estudio	66
7.3. Futuras Direcciones de Investigación	66

8 Demo - Herramienta clínica para la detección de cáncer de mama	67
8.1. Introducción	67
8.2. Elección y Despliegue del Servidor	67
8.3. Instalación de Flask y Tensorflow	68
8.4. Configuración de Flask	69
8.5. Configuración del dominio	70
8.5.1. Alojamiento del dominio	70
8.5.2. Configuración de certificado SSL para el dominio	71
8.5.3. Configuración NGinx	71
8.6. Visualización de la página	72
8.7. Conclusiones	74
9 Conclusiones	77
9.1. Resumen de Objetivos Alcanzados	77
9.2. Principales Hallazgos y Resultados	77
9.3. Análisis y Discusión de Resultados	77
9.4. Limitaciones y Desafíos	78
9.5. Implicaciones Prácticas	78
9.6. Recomendaciones para Trabajos Futuros	78
9.7. Reflexión Personal	78
10 Apéndice	79
10.1. Código del fichero Main Flask	79
Bibliografía	81

Índice de figuras

1.1. Diagrama de Gantt del proyecto.	4
3.1. Estructura de una Red Siamesa.	9
3.2. Comparación entre fotos de personas mediante Redes Siamesas.	10
4.1. Árbol de decisión para la predicción del precio de una vivienda	13
4.2. Resumen de las estadísticas sobre diferentes tipos de casas.	14
4.3. Gráfico sobre el Sobreajuste y el Subajuste utilizando MAE como medida	16
4.4. Ejemplo de Random Forests con decisión de elegir frutas.	17
4.5. Estructura lógica del perceptrón	18
4.6. Diferencia de función lineal a no lineal.	19
4.7. Estructura lógica del perceptrón aplicando Sigmoide	19
4.8. Estructura de una Red Neuronal por capas	20
4.9. Red Neuronal utilizando ReLU.	20
4.10. Entrenamiento de una red neuronal	22
4.11. Sobreajuste y Subajuste respecto a los Epochs	23
4.12. Representación gráfica del Early Stopping	23
4.13. Red neuronal con capas Dropout	24
4.14. Evolución de la entropía y el fallo a lo largo de las iteraciones.	25
4.15. Estructura de una red neuronal encoder	27
4.16. Uso de autoencoders para eliminar ruido en una imagen	28
4.17. Ejemplo de matriz de confusión en el Modelo Iris de Scikit-learn.	30
4.18. Ejemplo de curva ROC	31
5.1. Propiedades del dataset Breast Cancer Wisconsin	33
5.2. Datos del dataset Breast Cancer Wisconsin	34
5.3. Asignación de variables X e y	34
5.4. Mapa de calor de los diferentes atributos	35
5.5. Eliminación del atributo <i>mean fractial dimension</i>	36
5.6. Cantidad de elementos por clase	37
5.7. Modificación del conjunto de datos	38
5.8. Cantidad de elementos por clase tras la modificación	38
5.9. Separación del conjunto de datos	39
5.10. Separación en 3 conjuntos de datos	39
5.11. División de los conjuntos según las clases de sus elementos.	40
5.12. Normalización de los datos.	41
5.13. Visualización detallada de la estructura de la red neuronal autoencoder	42
5.14. Enter Caption	42
5.15. Codificación de la red neuronal	43
5.16. Compilación y ajuste del modelo	44
5.17. Clasificación utilizando el error de reconstrucción.	45

6.1.	Variable auxiliar para especificar la cantidad de anomalías.	49
6.2.	Distribuciones de los datos en función del porcentaje de anomalías.	50
6.3.	Curva de aprendizaje con 0.1 % de clases Malignas.	51
6.4.	Curva de aprendizaje con 0.2 % de clases Malignas.	51
6.5.	Curva de aprendizaje con 0.3 % de clases Malignas.	52
6.6.	Matriz de confusión 10 % de muestras malignas.	53
6.7.	Matriz de confusión 20 % de muestras malignas.	53
6.8.	Matriz de confusión 30 % de muestras malignas.	54
6.9.	Precision y Recall según los diferentes umbrales.	55
6.10.	Precision y Recall según los diferentes umbrales.	56
6.11.	Enter Caption	58
6.12.	Matrices de confusión de los diferentes modelos.	60
6.13.	Curva de aprendizaje del modelo A.	61
6.14.	Curva de aprendizaje del modelo B.	61
6.15.	Matriz de confusión del modelo A.	62
6.16.	Matriz de confusión del modelo B.	62
6.17.	Curva ROC del modelo A.	63
6.18.	Curva ROC del modelo B.	63
8.1.	Servicio Cloud Oracle	67
8.2.	Proveedor de dominios OVH	70
8.3.	Registro del subdominio en Zona DNS	70
8.4.	Página principal con formulario para predicción de una muestra . . .	73
8.5.	Página de resultado de la predicción	73
8.6.	Página web para clasificar muestras contenidas en un fichero Excel. .	74
8.7.	Resultados de la predicción del fichero excel.	74

Índice de tablas

1.1.	Planificación temporal del proyecto	4
1.2.	Presupuesto por horas	6
6.1.	Desempeño del modelo para distintos porcentajes de muestras de clase maligna	56
6.2.	Desempeño del modelo para diferentes umbrales	57
6.3.	Comparación del desempeño de los modelos de detección de anomalías	59
6.4.	Comparación del desempeño de los modelos de clasificación	64
8.1.	Especificaciones técnicas del servidor en Oracle Cloud	68

Índice de extractos de código

4.1.	Lectura de fichero csv con Pandas.	14
4.2.	Recolección de atributos para el modelo	14
4.3.	División del conjunto de datos y ajuste del modelo para su predicción y validación	15
4.4.	Poda del árbol de decisión	16
4.5.	Modelo de Random Forests	17
4.6.	Modelado del perceptrón con keras	18
4.7.	Red Neuronal stackeando diferentes capas	21
4.8.	Aplicación de optimizador y función de perdida	22
4.9.	Entrenamiento y validación del modelo	22
4.10.	Codificación del callback EarlyStopping	23
4.11.	Codificación del callback EarlyStopping	24
4.12.	Especificación del tipo de cada capa	25
4.13.	Red Neuronal para clasificación binaria	26
4.14.	Estructura de Autoencoder	28
8.1.	Instalación de Python	68
8.2.	Instalación de Flask	68
8.3.	Instalación de Tensorflow	69
8.4.	Configuración fichero Flask	69
8.5.	Eliminación de datos redundantes para el modelo	69
8.6.	Instalación de certbot	71
8.7.	Configuración de SSL para el dominio	71
8.8.	Instalación de NGinx	71
8.9.	Redirección al servicio Flask	71
10.1.	Configuración del servicio Flask	79

1. Planificación

1.1. El reto planteado

El reto de este proyecto de fin de grado es construir un modelo avanzado de autoencoder que pueda identificar con precisión ejemplos muy extraños. Vamos a trabajar con el conjunto de datos del Breast Cancer Wisconsin, que se encuentra en el repositorio UCI. Los autoencoders son una herramienta de aprendizaje automático que opera de forma no supervisada, lo cual los hace excelentes para descubrir patrones ocultos en los datos por sí mismos. Esto nos permite usarlos sin la necesidad de etiquetar cada muestra como sana o cancerosa.

1.2. Objetivos

1.2.1. Objetivos Docentes

Los objetivos docentes del presente TFG están orientados hacia el aprendizaje y la adquisición de competencias en el ámbito de la inteligencia artificial y su aplicación en la salud, específicamente en la detección de cáncer mediante autoencoders.

- Comprender en profundidad la teoría y la práctica de los autoencoders y su aplicación en el aprendizaje automático.
- Desarrollar habilidades en el análisis y procesamiento de datos médicos, con especial énfasis en el conjunto de datos Breast Cancer Wisconsin.
- Mejorar las competencias técnicas en programación utilizando librerías especializadas como Scikit Learn y Keras.
- Desarrollar la habilidad para abordar y resolver problemas complejos en el contexto de la inteligencia artificial aplicada a la salud.
- Prepararse para la integración profesional en campos emergentes relacionados con la inteligencia artificial y la salud.

1.2.2. Objetivos Técnicos

Los objetivos técnicos se centran en la aplicación práctica y el desarrollo de soluciones concretas en el ámbito de la detección de cáncer mediante el uso de autoencoders.

- Implementar y evaluar diversas arquitecturas de autoencoders, examinando su eficacia en el procesamiento del conjunto de datos Breast Cancer Wisconsin.
- Analizar el papel del error de reconstrucción en la identificación de muestras anómalas cancerígenas.
- Realizar experimentos prácticos que validen la utilidad de los autoencoders en escenarios reales y clínicos.
- Explorar la integración de los autoencoders en los procesos clínicos para la mejora del diagnóstico y tratamiento del cáncer.

1.3. Metodología

La metodología adoptada en este Trabajo de Fin de Grado

1.3.1. Diseño del Estudio

El estudio se enfoca en un enfoque experimental, implementando y evaluando diferentes arquitecturas de autoencoders para determinar su eficacia en la detección de muestras anómalas de cáncer. El estudio se organiza en las siguientes fases:

1. **Revisión Bibliográfica:** Amplia revisión de literatura para comprender los fundamentos y las aplicaciones más recientes de autoencoders.
2. **Preparación mediante diferentes cursos:** Para poder adquirir la capacidad de desarrollo de los diferentes modelos aplicables se mantendrá un enfoque constante en la realización de diferentes cursos de la plataforma Kaggle, así como al documentación mediante diferentes artículos y blogs científicos.
3. **Preparación de Datos:** Los datos del Breast Cancer Wisconsin son preprocesados para su normalización y partición, facilitando un entrenamiento eficaz del modelo.
4. **Desarrollo y Pruebas:** Implementación de diversas arquitecturas de autoencoders, ajustando hiperparámetros y evaluando su rendimiento mediante métricas específicas como el error de reconstrucción.

1.3.2. Herramientas y Tecnologías

Las herramientas y tecnologías utilizadas en el desarrollo e implementación de los autoencoders incluyen:

- **Python:** Lenguaje de programación principal por su amplia adopción en ciencia de datos y aprendizaje automático.
- **Keras y TensorFlow:** Bibliotecas de aprendizaje profundo que facilitan la construcción y el entrenamiento de modelos de redes neuronales.

- **Scikit-Learn:** Utilizado para manejo de datos y aplicaciones de aprendizaje automático más tradicionales.

1.3.3. Metodología de Desarrollo

El desarrollo del TFG sigue una metodología ágil donde se van realizando diferentes cambios constantemente con el objetivo de mejorar y rectificar el contenido ya realizado hasta la fecha.

La realización de este TFG ha llevado una alta cantidad de horas en la escritura de la misma memoria puesto que al ser un estudio científico gran parte del tiempo se ha requerido para la interpretación de la información y la exposición de la misma.

1. **Prototipado Rápido:** Construcción inicial de modelos básicos de autoencoders para establecer un punto de referencia.
2. **Evaluación y Optimización:** Los modelos se evalúan y optimizan basándose en su precisión de reconstrucción y capacidad para detectar anomalías.

Análisis de Datos

El análisis de datos es esencial, empleando técnicas estadísticas y de visualización para interpretar los resultados del modelo.

1.3.4. Demostración de Aplicación Práctica

Para ilustrar la aplicabilidad práctica de los autoencoders en la detección de cáncer, se desarrolló una demostración que permite visualizar cómo el modelo identifica y clasifica las muestras anómalas en tiempo real. La demostración fue implementada utilizando las siguientes tecnologías y procedimientos:

- **Desarrollo de la Interfaz:** Se utilizó Flask, un microframework de Python, para crear una interfaz web sencilla que permite a los usuarios interactuar con el modelo de autoencoder. La interfaz facilita la carga de datos y muestra los resultados de la detección de manera intuitiva.
- **Integración con TensorFlow:** El modelo de autoencoder, desarrollado con Keras y TensorFlow, fue integrado en la aplicación Flask. Esto permite que la detección de anomalías se realice en tiempo real, procesando los datos cargados por los usuarios a través de la interfaz web.

1.3.5. Documentación y Reporte

Todos los hallazgos y metodologías son documentados detalladamente en la memoria del TFG, asegurando que el proceso de investigación sea reproducible y transparente para futuras referencias.

Supervisión y Retroalimentación

A lo largo del desarrollo del TFG, se han establecido reuniones periódicas con el director del proyecto. Estas sesiones han sido esenciales para el seguimiento y evaluación continua del progreso del estudio. En cada reunión, se han discutido los avances, enfrentado desafíos y reajustado estrategias según fuera necesario, asegurando así que el proyecto mantuviera una dirección clara y objetiva.

1.4. Planificación

Este TFG estipulado en 300 horas ha sido dividido y planificado a lo largo de los meses de la forma representada en la tabla, dejando márgenes realmente amplios considerando que el autor de este TFG requiere de compaginar los estudios con la realización de este mismo TFG.

Tabla 1.1: Planificación temporal del proyecto

Fase	Inicio	Fin	Tiempo
Planificación	10/10/2023	15/10/2023	20h
Estudios Previos	16/10/2023	05/02/2024	100h
Desarrollo del Notebook	06/02/2024	15/03/2024	40h
Experimentación	16/03/2024	15/04/2024	85h
Despliegue del Servidor	16/04/2024	05/05/2024	15h
Escritura de la memoria	06/05/2024	20/05/2024	35h
Finalización del proyecto	21/05/2024	24/05/2024	5h
Total			300h

A continuación se muestra la información representada en el diagrama de gantt del proyecto.

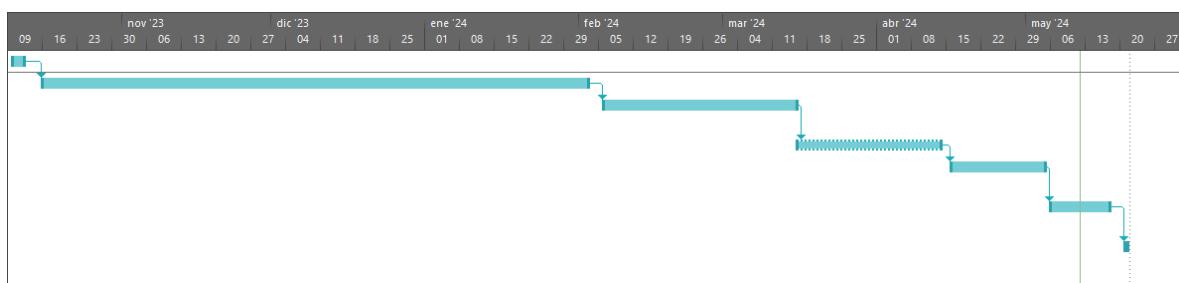


Figura 1.1: Diagrama de Gantt del proyecto.

1.5. Presupuesto

Este capítulo detalla los costes asociados al desarrollo del TFG, dividido en costes de personal y costes de software y herramientas, asegurando la cobertura de

todos los aspectos financieros del proyecto.

El proyecto se realizará exclusivamente por una sola persona, que deberá dividirse las tareas en 2 puestos principales, **Analista Programador** con un salario anual de 18.024,55 , y **Arquitecto de Sistemas** con 28.163,37 de salario anual [1].

1.5.1. Costes de personal

Teniendo en cuenta la planificación temporal consta que el Analista Programador dispondrá del mayor tiempo, llegando a las 280 horas donde se incluyen todas las fases a excepción de *Despliegue del servidor*. Contando con pagas prorrataeadas podemos contar el salario por hora del siguiente modo.

$$\left(\frac{\text{Salario Anual}}{12} \right) \div (\text{Horas semanales} \times 4)$$

Por lo cual el salario por horas de un Analista Programador sería de

$$\left(\frac{18,024,55}{12} \right) \div (40 \times 4) = 9,39$$

Lo que nos dejaría un sueldo bruto total de $9,39\text{€} * 280 = 2.629,2\text{€}$.

Mientras que el salario por horas de un Arquitecto de Sistemas constaría de la siguiente forma:

$$\left(\frac{28,163,37}{12} \right) \div (40 \times 4) = 14,67$$

Teniendo en cuenta que se requerirá dichas habilidades para las 15 horas del Despliegue del Servidor así como 5 horas de escritura de la memoria en el proceso relativo su sueldo bruto constaría de $14,67\text{€} * 20 = 293,4\text{€}$.

A estas cifras hay que sumarle un 30 % adicional debido al coste de la seguridad social que se desglosa del siguiente modo:

- Contingencias comunes (23,60 %)
- Desempleo para contratos indefinidos (5,50 %)
- FOGASA o Fondo de Garantía Salarial (0,20 %)
- Formación profesional (0,70 %)

Este coste adicional constaría de $(2629,2 + 293,4) * 30 \% = 876,78\text{€}$.

Con lo que nos quedaría un coste de personal total de $876,78\text{€} + 2692,4 + 293,4 = 3.799,38$.

1.5.2. Costes de Software

Todas las tecnologías así como los servidores utilizados han sido de coste gratuitos sin necesidad de elegir ningún plan de pago en ningun momento. A la hora de realizar algun escalado vertical u horizontal en el Servidor podría ser necesario a futuro implementar un plan superior con costes por tiempo de uso.

1.5.3. Costes totales

Como se ha comentado, los Costes de Software son nulos por lo que el coste total de proyecto reside en el precio integral de Costes del Personal, que consta de **3.862,58€**.

Tabla 1.2: Presupuesto por horas

Rol	Actividad	Horas	Salario por Hora (€)	Coste Total (€)
Analista Programador	Planificación	20	9.39	187.80
	Estudios Previos	100	9.39	939.00
	Desarrollo del Notebook	40	9.39	375.60
	Experimentación	85	9.39	798.15
	Escritura de la memoria	35	9.39	328.65
Arquitecto de Sistemas	Despliegue del Servidor	15	14.67	220.05
	Escritura de la memoria	5	14.67	73.35
Total Horas		300	-	2,922.60
Coste Seguridad Social (30 %)		-	-	876.78
Coste Total de Personal		-	-	3,799.38

2. Introducción

En la actualidad, la inteligencia artificial y el aprendizaje automático están revolucionando numerosos campos de la ciencia y la tecnología, proporcionando herramientas poderosas para resolver problemas complejos y mejorar los procesos existentes. En el ámbito de la salud, estas tecnologías ofrecen un potencial significativo para el diagnóstico y tratamiento de enfermedades, particularmente en la detección y análisis de diferentes tipos de cáncer. Este Trabajo de Fin de Grado (TFG) se centra en la aplicación de autoencoders, una clase de redes neuronales, en la detección de muestras de cáncer, específicamente utilizando el conjunto de datos Breast Cancer Wisconsin del UCI Repository [2].

La motivación inicial para este proyecto surgió del creciente interés en el uso de técnicas avanzadas de aprendizaje profundo para mejorar la precisión en la detección de cáncer. En el campo clínico, se dan muchas situaciones de falta de información que no se observan en otros campos donde se aplica el Machine Learning. En particular, los autoencoders han demostrado ser efectivos para identificar patrones ocultos en datos complejos, lo cual es crucial para la detección temprana de cáncer. Un estudio reciente publicado en *Briefings in Bioinformatics* destaca los avances en el uso de autoencoders para la clasificación de diferentes tipos de cáncer, demostrando cómo estas redes pueden discriminar entre diversos subtipos con alta precisión y proponiendo mejoras continuas en estas técnicas para facilitar diagnósticos más efectivos y tempranos [3].

El objetivo principal de este trabajo es explorar la capacidad de los autoencoders para identificar patrones ocultos en datos complejos y su aplicación en la detección del cáncer. Para ello, se estudiará en profundidad la arquitectura y el funcionamiento de los autoencoders, analizando cómo estas redes pueden aprender representaciones eficientes de los datos en un espacio latente. Esta capacidad se explora a través del análisis del error de reconstrucción, una métrica clave que indica cómo de bien un autoencoder puede reconstruir la entrada original a partir de su representación codificada.

El trabajo comienza con un análisis exhaustivo del conjunto de datos Breast Cancer Wisconsin, una referencia en la investigación del cáncer de mama. Se contemplarán diferentes modelos y casos de uso de aprendizaje automático, empleando la librería Scikit Learn, para establecer una base comparativa con los autoencoders. Seguidamente, se profundizará en el estudio de los autoencoders utilizando la librería Keras, explorando diferentes arquitecturas y su impacto en el error de reconstrucción.

Finalmente, el TFG incluirá una serie de casos de estudio experimentales donde se aplicarán los conocimientos adquiridos. Estos estudios buscarán no solo validar la eficacia de los autoencoders en la detección de muestras de cáncer, sino también ofrecer una perspectiva sobre cómo estas técnicas pueden integrarse en los flujos de trabajo clínicos para mejorar el diagnóstico y tratamiento del cáncer, para el cuál se

proveerá de una breve demo técnica para el caso de uso.

Este TFG representa una oportunidad para contribuir al campo emergente de la inteligencia artificial en la salud, con un enfoque específico en la oncología. A través de este estudio, buscamos no solo avanzar en el conocimiento técnico de las redes neuronales y el aprendizaje automático, sino también en su aplicación práctica en un contexto que tiene un impacto directo en la vida de las personas.

Se ha creado un repositorio en GitHub al que se puede acceder mediante el siguiente enlace <https://github.com/MarcoTenCortes/TFG-Autoencoder>.

3. Estado del arte

3.1. Introducción

El estado del arte abordará dos temas centrales: La clasificación con poca información y el error de reconstrucción con autoencoders, comentando diferentes estudios que avalan las diferentes herramientas utilizadas.

3.2. Clasificación con poca información

Existen una gran cantidad de casos en los que dispondremos de una cantidad muy limitada de datos, por lo que nos veremos obligados a utilizar técnicas especializadas como pueden ser las siguientes:

- **Few-shot learning:** Se trata de una técnica que permite a los modelos aprender a partir de muy pocos datos, utilizando diferentes técnicas de aprendizaje como el aprendizaje por meta-ejemplo [4], una técnica que permite a nuestro modelo “aprender a aprender”, contemplando como varían los patrones al modificar un conjunto de datos. Un estudio de Oxford realizado por Xi Chen y otros colaboradores, utilizaron esta técnica para la predicción de subtipos moleculares de meduloblastoma, un tipo de cáncer cerebral [5].
- **One Shot Learning:** El modelo se entrena con únicamente un ejemplo por clase, usando por ejemplo redes Siamesas [6], donde 2 sub-redes idénticas son entrenadas con diferentes datos de entrada y se comparan sus resultados.

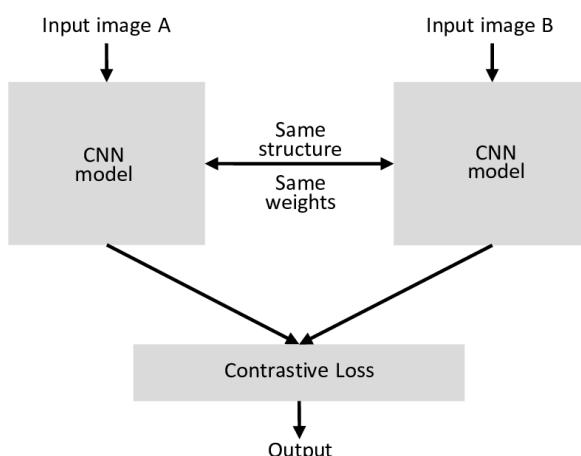


Figura 3.1: Estructura de una Red Siamesa.

Un claro ejemplo del One Shot Learning utilizando redes siamesas es la semejanza entre 2 imágenes sobre una misma persona.

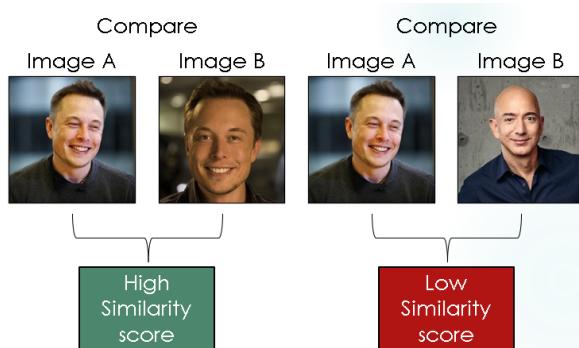


Figura 3.2: Comparación entre fotos de personas mediante Redes Siamesas.

Lake et al. (2015) exploraron la emulación de los modelos cognitivos humanos a través de un único ejemplo, utilizando la base lógica de cómo hasta un niño a través de un patinete podía relacionarlo con otros automóviles con características similares, mientras que generalmente los diferentes algoritmos de machine learning requieren de cantidades muy superiores como ejemplos para poder realizar una clasificación efectiva [7]. Para realizar la emulación realizaron diferentes modelos generativos.

- **Zero Shot Learning:** Esta técnica presenta un desafío único ya que los modelos se entrena sin contar con ejemplos directos para algunas de sus clases. En Zero Shot Learning, es común emplear información auxiliar para complementar y enriquecer el proceso de clasificación. Entre las estrategias más utilizadas destacan *Transfer Learning* [8], que prepara al modelo con conjuntos de datos similares previamente adquiridos, y *Sentence Embedding*, que transforma la información textual en vectores numéricos para facilitar comparaciones y clasificaciones por similitud. Además, una faceta interesante de Zero Shot Learning es su capacidad de manejar clases que no estaban presentes durante el entrenamiento del modelo, utilizando descripciones o atributos semánticos para hacer inferencias sobre nuevos objetos [9]. En el ámbito práctico, Rezael y Shaidi (2020) demostraron la aplicabilidad de esta técnica en diversos contextos, incluyendo la conducción autónoma y el diagnóstico médico, como en el caso del COVID-19 [10]. Zero Shot Learning no solo es innovador por su enfoque de aprendizaje sin ejemplos, sino también por cómo se adapta a nuevos desafíos y escenarios, potenciando su utilidad en campos donde la obtención de datos etiquetados es complicada o imposible.

3.3. Detección de anomalías

La detección de anomalías consiste en identificar los diferentes patrones obtenidos cuando el comportamiento de los resultados no es el esperado, lo que

nos ayuda a poder realizar una clasificación en función de la misma. En medicina, la detección de anomalías se utiliza para identificar casos raros que pueden ser difíciles de diagnosticar mediante métodos convencionales. Los autoencoders, en particular, se aplican en la reconstrucción de imágenes médicas para identificar desviaciones respecto a lo normal que podrían sugerir la presencia de patologías. Schlegl et al. (2017) demostraron cómo los autoencoders pueden detectar anomalías en imágenes de fondo de ojo, identificando lesiones diabéticas y otros signos patológicos con precisión [11].

3.3.1. Autoencoders para la detección de anomalías

Los autoencoders son eficaces para aprender representaciones compactas de los datos, lo que los hace útiles para identificar anomalías al reconstruir los datos y evaluar los errores de reconstrucción. Por ejemplo, Baur et al. (2018) utilizaron autoencoders convolucionales para detectar anomalías en imágenes de resonancia magnética del cerebro, destacando efectivamente las áreas con posibles lesiones cerebrales [12].

3.4. Error de reconstrucción con autoencoders

Un autoencoder es un tipo de red neuronal el cuál está diseñado para comprimir los datos de entrada en un espacio latente para posteriormente ser reconstruido. Dicha diferencia entre la entrada y la salida se mide mediante el error de reconstrucción. El error de reconstrucción se calcula típicamente como la pérdida entre los datos de entrada y su reconstrucción. En muchos casos, esta pérdida se mide mediante la función de pérdida del error cuadrático medio (MSE). Pumsirirat et al. (2018) utilizó dicha técnica para implementar un sistema de detección de fraude para tarjetas de crédito[13] donde se reconstruyen diferentes transacciones con el objetivo de encontrar anomalías.

Torabi et al. (2023) recientemente ha presentado un modelo eficiente utilizando autoencoders para la detección de anomalías en redes de conexión[14], enfocándose en la perspectiva del *cloud computing*, donde se utiliza el error de reconstrucción como métrica para identificar y clasificar anomalías. A diferencia de en otros métodos, se interpreta el error de reconstrucción como un vector, lo que le ayuda a obtener una clasificación más detallada.

3.5. Avances en el uso de autoencoders para la clasificación de tipos de cáncer

En el artículo *Advancements in Autoencoder-Based Classification for Cancer Typing* Zheng et al. (2024) [3] publicado en la revista *Briefings in Bioinformatics*, se discuten los últimos desarrollos en el uso de autoencoders para mejorar la clasificación de

diferentes tipos de cáncer. Los autores examinan cómo la representación en el espacio latente generada por autoencoders puede ser utilizada para discriminar entre diversos subtipos de cáncer con alta precisión. Este enfoque destaca la importancia del diseño de la arquitectura de la red y la selección de características en la mejora de la capacidad de clasificación. El estudio propone también que la mejora continua en las técnicas de autoencoders puede facilitar la detección precoz y precisa de cánceres, ofreciendo así un potencial significativo en la aplicación clínica para diagnósticos más efectivos.

4. Metodos y Herramientas

4.1. Introducción

A lo largo de este capítulo se estudiarán y expondrán los diferentes enfoques y tecnologías que se han requerido aprender para la realización de los diferentes experimentos a posteriori.

4.2. Introducción al Machine Learning

Este primer curso introductorio a los modelos de aprendizajes explora diferentes áreas del *Machine Learning* con el propósito de poder diseñar modelos de aprendizaje eficientes.

4.2.1. Árboles de decisión

Un árbol de decisión es una forma de representar la información, dividiéndola en subconjuntos más pequeños y manejables mediante preguntas o atributos, con el objetivo de realizar una predicción factible.

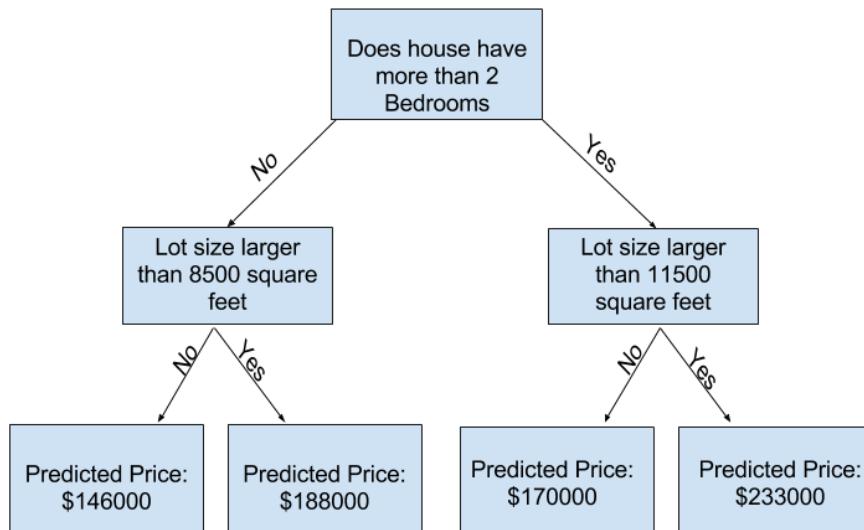


Figura 4.1: Árbol de decisión para la predicción del precio de una vivienda
Fuente: Imagen perteneciente al curso de Intro to Machine Learning

4.2.2. Exploración de datos mediante Pandas

La librería pandas nos permite explorar y modificar los datos. DataFrames, una de las más importantes partes de la librería pandas, es una gran herramienta mediante la cual podremos representar la información estructurada en una tabla similar a las hojas de Excel.

```
1 import pandas as pd
2 # Ruta del fichero
3 melbourne_file_path = '../input/melbourne-housing-snapshot/melb_data.csv'
4
5 # Lectura del fichero mediante Pandas
6 melbourne_data = pd.read_csv(melbourne_file_path)
7
8 # Resumen de la informacion representada en forma de tabla
9 melbourne_data.describe()
```

Extracto de código 4.1: Lectura de fichero csv con Pandas.

	Rooms	Price	Distance	Postcode	Bedroom2	Bathroom	Car	Lan
count	13580.000000	1.358000e+04	13580.000000	13580.000000	13580.000000	13580.000000	13518.000000	135
mean	2.937997	1.075684e+06	10.137776	3105.301915	2.914728	1.534242	1.610075	558
std	0.955748	6.393107e+05	5.868725	90.676964	0.965921	0.691712	0.962634	399
min	1.000000	8.500000e+04	0.000000	3000.000000	0.000000	0.000000	0.000000	0.00
25%	2.000000	6.500000e+05	6.100000	3044.000000	2.000000	1.000000	1.000000	177
50%	3.000000	9.030000e+05	9.200000	3084.000000	3.000000	1.000000	2.000000	440
75%	3.000000	1.330000e+06	13.000000	3148.000000	3.000000	2.000000	2.000000	651
max	10.000000	9.000000e+06	48.100000	3977.000000	20.000000	8.000000	10.000000	433

Figura 4.2: Resumen de las estadísticas sobre diferentes tipos de casas.

Fuente: Imagen perteneciente al curso de Intro to Machine Learning

4.2.3. Filtrado de datos para el Modelo

Como hemos visto en el ejemplo que utiliza el conjunto de datos sobre diferentes casas, podemos encontrarnos con una cantidad excesiva de atributos. Al implementar un modelo de aprendizaje, debemos escoger los atributos que consideremos más relevantes al tomar decisiones, así como elegir un atributo objetivo para clasificar los diversos resultados.

```
1 # Atributos de decision
2 melbourne_features = ['Rooms', 'Bathroom', 'Landsize', 'Latitude', 'Longitude']
3 X = melbourne_data[melbourne_features]
4
5 # Atributo de clase objetivo
6 y = melbourne_data.Price
```

Extracto de código 4.2: Recolección de atributos para el modelo

4.2.4. Validación del modelo

La elección de los atributos para la toma de la decisión es fundamental para que el modelo escogido sea efectivo y realice una predicción lo más exacta posible. Para poder identificar cuan buena ha sido la elección de nuestro modelo utilizaremos **MAE** (Error absoluto medio) como medida cuantificativa del error del modelo.

```
1 from sklearn.model_selection import train_test_split
2
3 # División del conjunto de datos y ajuste del modelo
4 train_X, val_X, train_y, val_y = train_test_split(X, y, random_state = 0)
5 melbourne_model = DecisionTreeRegressor()
6 melbourne_model.fit(train_X, train_y)
7
8 # Predicción y cálculo del error mediante el error absoluto medio
9 val_predictions = melbourne_model.predict(val_X)
10 print(mean_absolute_error(val_y, val_predictions))
```

Extracto de código 4.3: División del conjunto de datos y ajuste del modelo para su predicción y validación

4.2.5. Sobreajuste y Subajuste en Machine Learning

Comúnmente denominados por su terminología inglesa (Underfitting and Overfitting), hablamos de dos errores comunes en los que tienden a caer los modelos de aprendizaje automático. El sobreajuste ocurre cuando un modelo se ajusta excesivamente a los datos de entrenamiento, capturando tanto los patrones reales como el ruido, lo que afecta su capacidad para generalizar a nuevos datos. Por otro lado, el subajuste sucede cuando un modelo es demasiado simple para capturar la estructura completa de los datos, resultando en un rendimiento deficiente tanto en el entrenamiento como en la validación o prueba.

En el siguiente gráfico podemos contemplar cómo al comienzo del árbol de decisión el error es muy alto puesto que ocurre Subajuste al no haber obtenido suficiente información para el modelo, y posteriormente alcanzar el sobreajuste al obtener demasiada información irrelevante sobre el conjunto de datos.

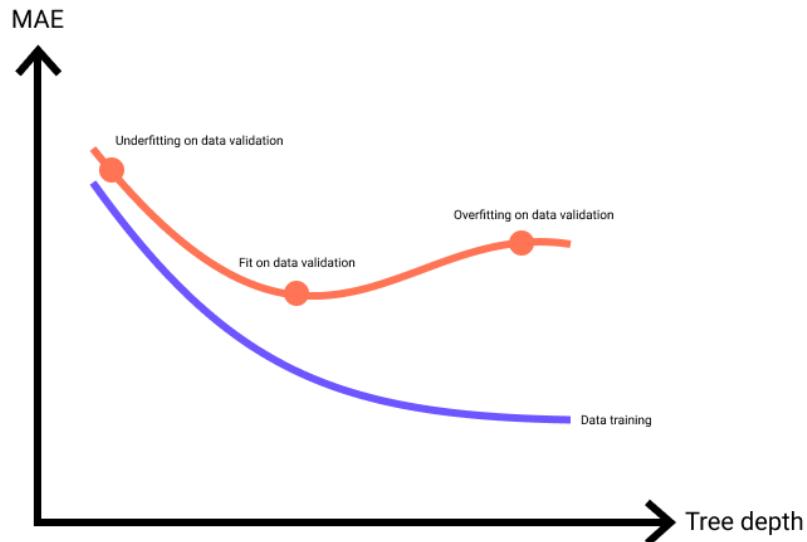


Figura 4.3: Gráfico sobre el Sobreajuste y el Subajuste utilizando MAE como medida

Fuente: <https://medium.com/@kuntikhoirunnisa/machine-learning-model-fit-underfitting-vs-overfitting-9d2716eebcd7>

Una solución a estos problemas consiste en *podar* el árbol de decisión, donde vamos modificando la altura máxima del árbol hasta alcanzar el mejor MAE posible.

```

1 # Se modifica la altura maxima hasta alcanzar el menor error posible.
2 max_leaf_nodes = 10
3 model = DecisionTreeRegressor(max_leaf_nodes=max_leaf_nodes, random_state=0)

```

Extracto de código 4.4: Poda del árbol de decisión

4.2.6. Random Forests

Una forma de solucionar la tendencia que tienen los árboles de decisión a la hora del sobreajuste y del subajuste es utilizar el modelo de aprendizaje Random Forests (o Bosques aleatorios), donde se crean diferentes árboles de decisión y se llega a un consenso mediante una media de los diferentes resultados[15].

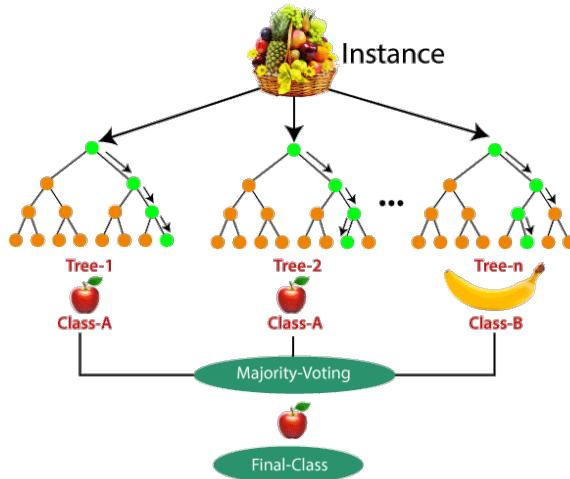


Figura 4.4: Ejemplo de Random Forests con decisión de elegir frutas.
Fuente: Javapoint.com

Mediante scikit-learn podemos utilizar este modelo mediante la librería RandomForestRegressor.

```

1 from sklearn.ensemble import RandomForestRegressor
2 from sklearn.metrics import mean_absolute_error
3
4 # Creacion y ajuste del modelo
5 forest_model = RandomForestRegressor(random_state=1)
6 forest_model.fit(train_X, train_y)
7
8 # Prediccion y error medio
9 melb_preds = forest_model.predict(val_X)
10 print(mean_absolute_error(val_y, melb_preds))

```

Extracto de código 4.5: Modelo de Random Forests

4.3. Introducción al Deep Learning

El siguiente curso de Kaggle se muestran diferentes campos de modelos de aprendizaje en Deep Learning, que a diferencia del Machine Learning se basan en redes neuronales, las cuales son más idoneas para el tratamiento de cantidades excesivas de datos. Para su desarrollo se usarán diferentes librerías como Keras y Tensorflow.

4.3.1. Perceptron, una única neurona

La menor unidad posible a la hora de crear una Red Neuronal se trata del perceptrón, en la que una neurona recibe diferentes parámetros de entrada a las cuales se les asigna un **peso** representando la importancia atributo para el modelo

de aprendizaje, así como el peso **bias** o sesgo, el cuál se añade con el propósito de ajustar el valor de salida del perceptrón.

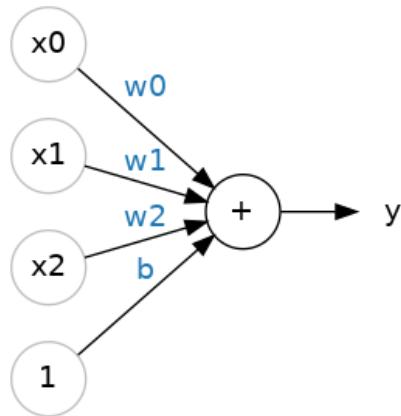


Figura 4.5: Estructura lógica del perceptrón

La salida esperada del perceptrón se conoce como la suma de los valores de entradas multiplicada por sus pesos

$$y = \sum_{i=1}^n w_i x_i + b \quad (4.1)$$

Utilizando la librería Keras podemos representar un perceptrón mediante la función Sequential, escogiendo una densidad de 1 (cantidad de neuronas) y escogiendo un el tamaño de entrada.

```

1 from tensorflow import keras
2 from tensorflow.keras import layers
3
4 # Creacion del perceptron con 3 entradas
5 model = keras.Sequential([
6     layers.Dense(units=1, input_shape=[3])
7 ])
  
```

Extracto de código 4.6: Modelado del perceptrón con keras

Cuándo modelamos un modelo de aprendizaje necesitamos que aprenda a identificar los **patrones no lineales** de la mayoría de los datos reales, por lo que el cálculo de la suma ponderada de las entradas no es viable puesto que esto no es más que una operación lineal. Para ello aplicaremos **funciones de activación** sobre el cálculo de los valores de entrada.

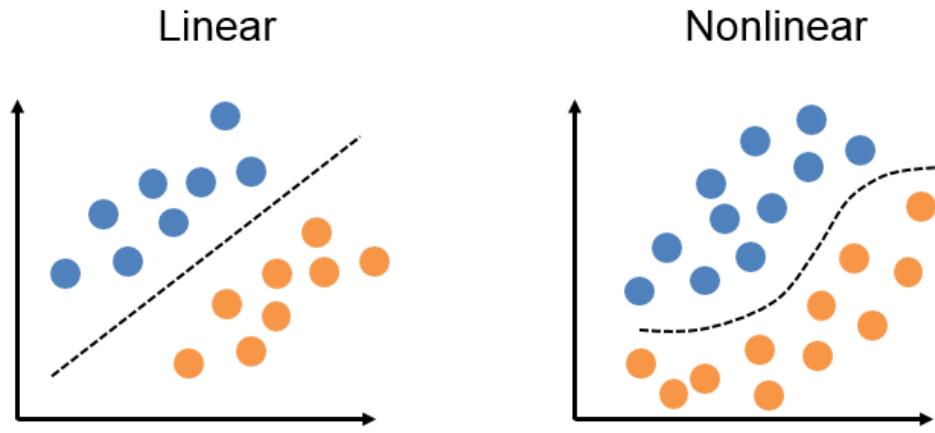


Figura 4.6: Diferencia de función lineal a no lineal.

Fuente:

<https://github.com/jtsulliv/ML-from-scratch/blob/master/Neural-Networks/perceptron.ipynb>

En los siguientes ejemplos optaremos por utilizar la función **ReLU** (Rectifier Function), una de las funciones más usadas en los últimos años donde rectificamos la parte negativa de la función convirtiéndola en 0.

$$\text{ReLU}(x) = \max(0, w * x + b)$$

La función de activación ReLU puede ser representada en el perceptrón del siguiente modo.

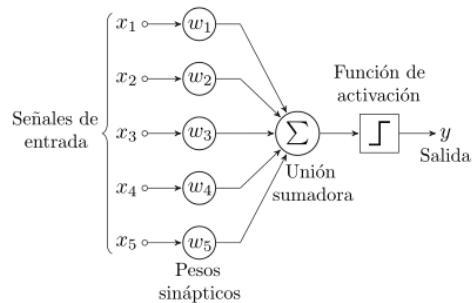


Figura 4.7: Estructura lógica del perceptrón aplicando Síntomeide

Fuente: Wikipedia

4.3.2. Redes Neuronales

Una red neuronal típica se compone de una serie de capas que incluyen capas de entrada, ocultas y de salida. Cada capa está formada por unidades, o neuronas, que están interconectadas y transmiten señales entre sí. Existen 3 tipos diferentes de capas.

1. La **capa de entrada** recibe los datos iniciales y no realizan ninguna operación.
2. Las **capas ocultas** procesan la información mediante conexiones ponderadas. No vemos los resultados de sus operaciones.
3. La **capa de salida** produce el resultado final.

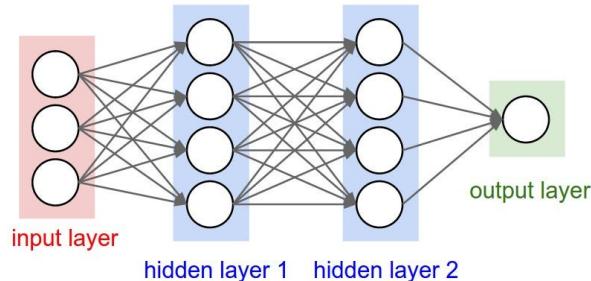


Figura 4.8: Estructura de una Red Neuronal por capas
Fuente: <https://bootcampai.medium.com/redes-neuronales-13349dd1a5bb>

En el siguiente caso podemos ver cómo las capas ocultas realizan la función ReLU sobre los datos entrantes en su neurona correspondiente, para luego en la capa de salida producir una salida ejecutando el sumatorio de los resultados entrantes en dicha neurona.

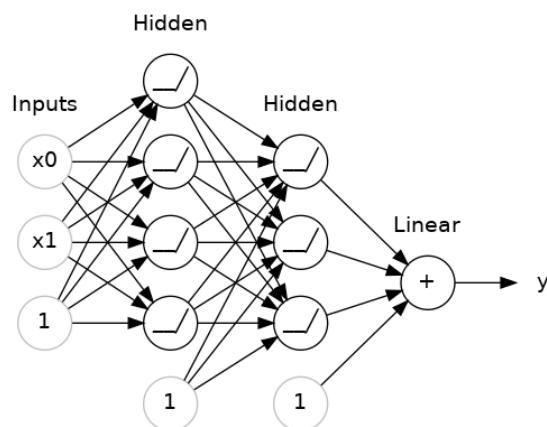


Figura 4.9: Red Neuronal utilizando ReLU.

El motivo tras el que la capa de salida no realiza ninguna función de activación es debido a que esta red se trata de un modelo para una tarea de regresión, con el objetivo de predecir un valor numérico arbitrario (como por ejemplo el precio de la vivienda en el ejemplo utilizado en la sección 4.2). Sin embargo, si el objetivo de la red es realizar una clasificación, deberíamos utilizar alguna función de activación en la capa de salida.

Para codificar una Red Neuronal basta con stackear¹ diferentes capas de neuronas dentro de la secuencia

¹El término *stackear* se refiere a apilar o acumular.

```

1 model = keras.Sequential([
2     # Capas ocultas
3     layers.Dense(units=4, activation='relu', input_shape=[2]),
4     layers.Dense(units=3, activation='relu'),
5     # Capa de salida
6     layers.Dense(units=1),
7 ])

```

Extracto de código 4.7: Red Neuronal stackeando diferentes capas

4.3.3. Entrenamiento de la red

El aprendizaje de una red neuronal reside en su capacidad para ajustar sus pesos de manera que pueda mapear entradas complejas a salidas deseadas. Este ajuste se realiza mediante algoritmos de optimización, que iterativamente mejoran los pesos basándose en la diferencia entre las salidas producidas por la red y las salidas esperada Para entrenar necesitamos también:

- Una **función de pérdida**, que mide la calidad de las predicciones de la red.
- Un **optimizador**, que le indica a la red cómo cambiar sus pesos.

Para guiar este proceso de ajuste, se utilizan funciones de pérdida, que proporcionan una medida cuantitativa de cuán lejos están las predicciones del modelo de los resultados reales. Al igual que en [4.2.4](#) utilizaremos **MAE** como medida para cuantificar cuan válido es nuestro modelo.

El optimizador es un algoritmo que ajusta los pesos para minimizar la pérdida. La mayoría de los algoritmos de optimización utilizados en el aprendizaje profundo pertenecen a la familia del descenso de gradiente estocástico (SGD). Este algoritmo ajusta los pesos de la red en pasos iterativos a través del siguiente proceso:

- Tomar una muestra del conjunto de entrenamiento y hacer predicciones.
- Medir la pérdida entre las predicciones y los valores reales.
- Ajustar los pesos en una dirección que reduzca la pérdida.

Cada vez que recorremos un ejemplo del conjunto de entrenamiento realizamos un **batch**, mientras que cuando recorremos todo el conjunto de entrenamiento realizamos un **epoch**.

La tasa de aprendizaje determina el tamaño de los ajustes en los pesos en cada actualización, o dicho de otro modo. Una tasa de aprendizaje menor significa que la red necesita una cantidad mayor de epochs para obtener unos valores óptimos.

A través de un proceso iterativo de prueba y error, la red va refinando sus predicciones hasta alcanzar un nivel de precisión aceptable o hasta que los ajustes adicionales ya no resulten en mejoras significativas.

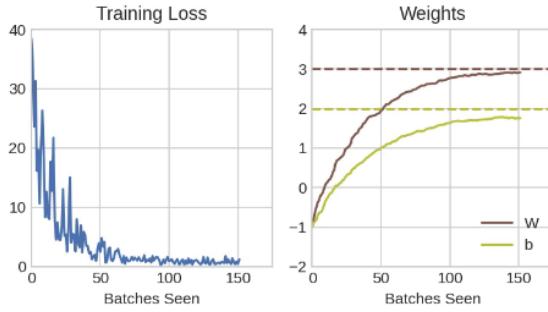


Figura 4.10: Entrenamiento de una red neuronal

Utilizaremos Adam, un algoritmo SGD muy eficiente, como optimizador para nuestras redes neuronales. Para poder codificar que funciones de optimización y perdida utilizaremos basta con especificar los parámetros adecuados en el método `compile` de nuestro modelo.

```

1 model.compile(
2     optimizer="adam",
3     loss="mae",
4 )

```

Extracto de código 4.8: Aplicación de optimizador y función de perdida

Para el ajuste del modelo elegiremos una cantidad de epochs y de batch, que podemos ir modificando hasta encontrar un error menor.

```

1 history = model.fit(
2     X_train, y_train,
3     validation_data=(X_valid, y_valid),
4     batch_size=256,
5     epochs=10,
6 )

```

Extracto de código 4.9: Entrenamiento y validación del modelo

4.3.4. Sobreajuste y Subajuste en aprendizaje de Redes Neuronales

Al igual que en modelos de aprendizaje de Machine Learning 4.2.5, podemos encontrarnos con estos famosos problemas en modelos de aprendizaje de Deep Learning.

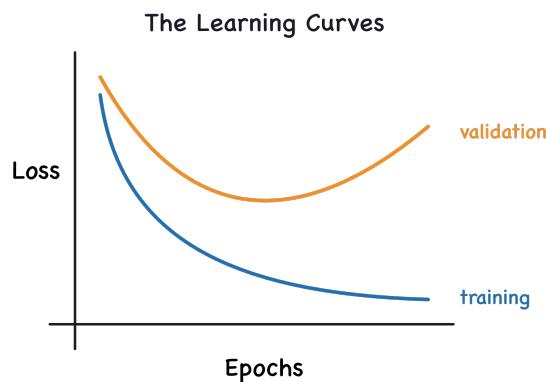


Figura 4.11: Sobreajuste y Subajuste respecto a los Epochs

En este caso la profundidad del árbol de decisiones se extrapola a la cantidad de epochs realizado para el aprendizaje del modelo.

Así mismo, la solución utilizada en este caso se trata del *Early Stopping*. Si en los árboles de decisiones ajustabamos una altura máxima de profundidad, en este caso el modelo de aprendizaje modifica el valor de los pesos a un estado previo a la subida del error.

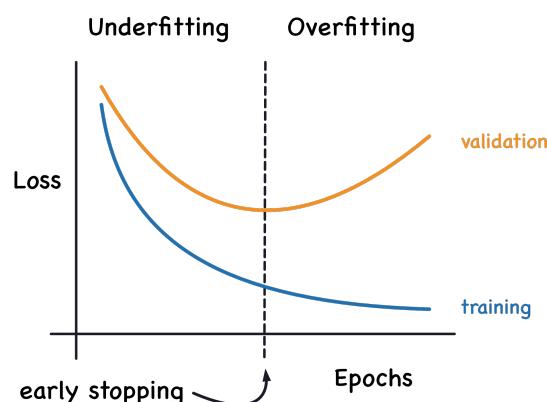


Figura 4.12: Representación gráfica del Early Stopping

```

1 from tensorflow.keras import layers, callbacks
2
3 early_stopping = callbacks.EarlyStopping(
4     # Cantidad mínima para considerarse una mejora
5     min_delta=0.001,
6
7     # Cantidad de epochs que se esperan si no se ve una mejora
8     patience=20, # how many epochs to wait before stopping
9     restore_best_weights=True,
10)

```

Extracto de código 4.10: Codificación del callback EarlyStopping

Una vez creado basta con añadirlo como parámetro dentro del ajuste del modelo.

```
1 history = model.fit(  
2     X_train, y_train,  
3     validation_data=(X_valid, y_valid),  
4     batch_size=256,  
5     epochs=500,  
6     callbacks=[early_stopping]  
7 )
```

Extracto de código 4.11: Codificación del callback EarlyStopping

4.3.5. Dropout and Batch Normalization

En el Deep Learning existe una variedad de capas que se pueden añadir a un modelo más allá de las tradicionales capas densas. Algunas de estas capas, como las capas densas, definen conexiones entre neuronas, mientras que otras realizan preprocesamiento o transformaciones de otros tipos.

- Capas de tipo Dropout: Un tipo de capa que tiende a ayudar a corregir el sobreajuste. Al dejar caer aleatoriamente una fracción de las unidades de entrada de una capa en cada paso del entrenamiento, se hace mucho más difícil para la red aprender esos patrones específicos en los datos de entrenamiento. En cambio, la red debe buscar patrones amplios y generales, cuyos patrones de peso tienden a ser más robustos.

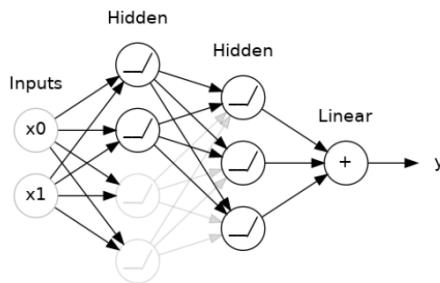


Figura 4.13: Red neuronal con capas Dropout

- Capas de tipo Batch Normalization: Se trata de capas que pueden ayudar a corregir un entrenamiento que es lento o inestable. Generalmente solemos normalizar² los datos antes de introducirlos al modelo, pero no podemos normalizar manualmente los datos en el interior de la red neuronal. Las capas de este tipo normalizan la información que se procesa en cada una de las capas, facilitando el procesamiento de los datos, facilitando el modelo de aprendizaje y de este modo necesitando una cantidad menor de epochs.

Para codificar este tipo de capas basta con especificarlo en su secuencia.

²El término *normalizar* se refiere a ajustar los valores a una escala común.

```

1 model = keras.Sequential([
2     layers.Dense(1024, activation='relu', input_shape=[11]),
3     layers.Dropout(0.3),
4     layers.BatchNormalization(),
5     layers.Dense(1024, activation='relu'),
6     layers.Dropout(0.3),
7     layers.BatchNormalization(),
8     layers.Dense(1024, activation='relu'),
9     layers.Dropout(0.3),
10    layers.BatchNormalization(),
11    layers.Dense(1),
12 ])

```

Extracto de código 4.12: Especificación del tipo de cada capa

4.3.6. Clasificación Binaria

La clasificación binaria es una forma de problema de aprendizaje supervisado en el que el objetivo es predecir cuál de dos clases posibles corresponde a cada muestra de entrada. Esta tarea es común en campos como la medicina, las finanzas y el reconocimiento de patrones, donde las preguntas suelen formularse como "sí/no", "verdadero/falso"... Ejemplos típicos incluyen determinar si un email es spam o no, si una transacción es fraudulenta, o si un paciente tiene una determinada enfermedad basándose en sus síntomas...

A diferencia de en los anteriores modelos deberemos de usar funciones diferentes para el error y la evaluación.

- **Accuracy:** Se calcula como el número de predicciones correctas dividido por el número total de predicciones hechas.
- **Entropía Cruzada:** Se trata de una medida que calcula cuán lejos está la probabilidad de ser 1.0 en base a la Accuracy.

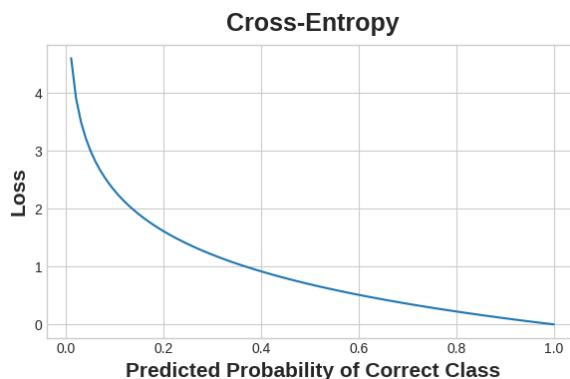


Figura 4.14: Evolución de la entropía y el fallo a lo largo de las iteraciones.

Cómo se comentó en 4.3.1 los modelos de clasificación binario deben tener en la capa de salida una función de activación. La entropía cruzada y Accuracy requieren

tener valores de entradas entre 0 y 1, por lo que utilizamos la función ReLU en las capas intermedias, y luego para convertir en probabilidades los datos de salida utilizaremos la función sigmoide σ .

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

Para codificarlo este tipo de modelo basta con añadirlo como métrica en su función compile así cómo especificar la función de activación en la secuencia.

```

1 model = keras.Sequential([
2     layers.Dense(4, activation='relu', input_shape=[33]),
3     layers.Dense(4, activation='relu'),
4     layers.Dense(1, activation='sigmoid'),
5 ])
6
7 model.compile(
8     optimizer='adam',
9     loss='binary_crossentropy',
10    metrics=['binary_accuracy'],
11 )

```

Extracto de código 4.13: Red Neuronal para clasificación binaria

4.4. Autoencoders

Un autoencoder es un tipo de red neuronal utilizada para el aprendizaje no supervisado, ya que los datos no se encuentran etiquetados. Se encarga de comprimir la información de entrada en un "cuello de botella" denominado espacio latente, el cual contiene la "esencia" de la información de los datos de entrada. Posteriormente se intenta reconstruir los datos de entrada originales a partir del espacio latente.

Su estructura se divide en 3 partes principalmente:

1. **Codificador (Encoder):** Es la primera parte de la red donde la entrada se procesa para reducir su dimensión. El codificador transforma los datos de entrada en un conjunto más pequeño y denso de neuronas que representan las características esenciales de esos datos. Este paso implica generalmente varias capas que van reduciendo su tamaño hasta llegar al espacio latente.
2. **Espacio Latente o Bottleneck:** Esta es la parte central de la red donde la entrada ha sido comprimida. El espacio latente contiene la representación comprimida de los datos de entrada y es desde donde el decodificador reconstruirá la entrada original. Este espacio suele ser de menor dimensión que la entrada y la salida, y es el "cuello de botella"^a través del cual deben pasar todas las características esenciales de los datos.
3. **Decodificador (Decoder):** La tercera parte de la red recibe la representación comprimida del espacio latente y trabaja para reconstruir la entrada original a

su tamaño y forma original. El proceso de decodificación a menudo es simétrico al de codificación, utilizando una estructura de capas que incrementa gradualmente su tamaño hasta alcanzar la dimensión original de los datos.

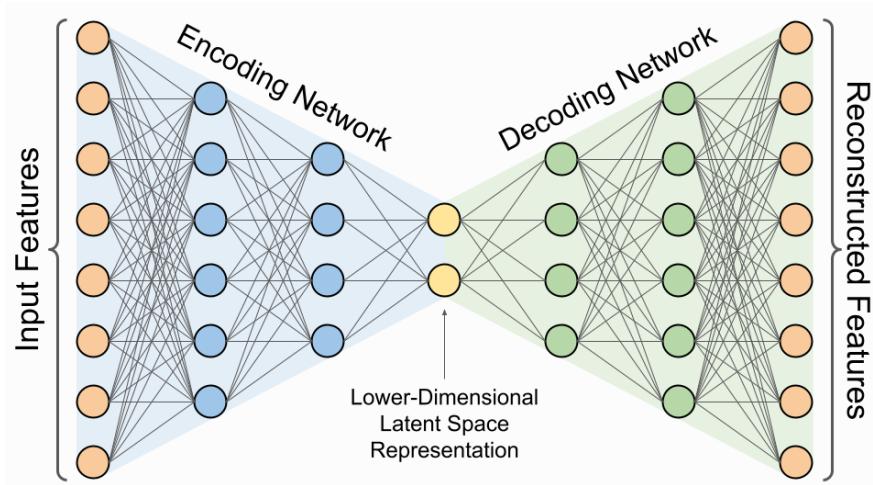


Figura 4.15: Estructura de una red neuronal encoder.

Fuente:

<https://www.assemblyai.com/blog/introduction-to-variational-autoencoders-using-keras/>

4.4.1. Aplicaciones del autoencoder

El espacio latente obtenido con la compresión de la información entrante puede ser utilizada para diferentes usos, como eliminar ruido en imágenes o detectar anomalías. Por ejemplo, Theis et al. (2017)[16] desarrolló diferentes mejoras para eliminar ruido en las imágenes con formato JPEG 2000.

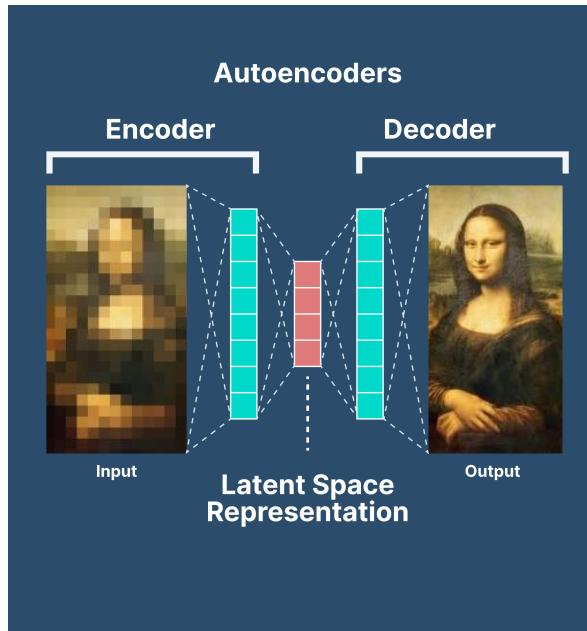


Figura 4.16: Uso de autoencoders para eliminar ruido en una imagen

4.4.2. Programación de Autoencoder

Para implementar un autoencoder en Keras, se definen el codificador y el decodificador utilizando capas densas (Dense layers). A continuación se presenta un ejemplo básico de cómo codificar un autoencoder simple.

```

1 from keras.layers import Input, Dense
2 from keras.models import Model
3
4 # Definicion de las dimensiones de la entrada y el espacio latente
5 input_dim = X_train.shape[1] # numero de caracteristicas en los datos de entrada
6 latent_dim = 32 # tamano deseado del espacio latente
7
8 # Capa de entrada
9 input_layer = Input(shape=(input_dim,))
10
11 # Codificador
12 # Reducción progresiva hasta alcanzar la dimensión del espacio latente
13 encoder_layer1 = Dense(128, activation='relu')(input_layer)
14 encoder_layer2 = Dense(64, activation='relu')(encoder_layer1)
15 encoder_output = Dense(latent_dim, activation='relu')(encoder_layer2)
16
17 # Decodificador
18 # Reconstrucción progresiva hasta alcanzar la dimensión original de la entrada
19 decoder_layer1 = Dense(64, activation='relu')(encoder_output)
20 decoder_layer2 = Dense(128, activation='relu')(decoder_layer1)
21 decoder_output = Dense(input_dim, activation='sigmoid')(decoder_layer2)
22
23 # Construcción del modelo de autoencoder
24 autoencoder = Model(inputs=input_layer, outputs=decoder_output)
25 autoencoder.compile(optimizer='adam', loss='mean_squared_error')
26

```

```

27 # Entrenamiento del modelo
28 autoencoder.fit(X_train, X_train,
29                 epochs=50,
30                 batch_size=256,
31                 shuffle=True,
32                 validation_data=(X_test, X_test))

```

Extracto de código 4.14: Estructura de Autoencoder

4.5. Medidas de evaluación

Este capítulo detalla cómo se evalúan los modelos de autoencoder diseñados para la detección de muestras de cáncer, utilizando un conjunto de técnicas de visualización y análisis estadístico para comprender la eficacia y el comportamiento del modelo bajo diversas condiciones de entrenamiento y clasificación.

4.5.1. Curva de Entrenamiento y Evaluación en Función de la Pérdida

La curva de pérdida es esencial para monitorear el progreso del modelo durante su entrenamiento y validación, proporcionando una representación visual clara de cómo el modelo aprende y se adapta al problema en cuestión.

- **Descripción:** Durante el entrenamiento del autoencoder, se registra la pérdida de reconstrucción en cada epoch , tanto para el conjunto de entrenamiento como para el de validación. La curva de entrenamiento muestra cómo la pérdida disminuye a medida que el modelo se ajusta a los datos de entrenamiento, mientras que la curva de validación indica cómo el modelo generaliza a nuevos datos.
- **Interpretación:** Idealmente, ambas curvas deberían mostrar una tendencia decreciente, indicando que el modelo está aprendiendo correctamente y mejorando su precisión. Si la curva de validación comienza a aumentar mientras la de entrenamiento sigue disminuyendo, esto puede indicar un sobreajuste, sugiriendo que el modelo está memorizando los datos de entrenamiento en lugar de aprender a generalizar.

4.5.2. Matriz de Confusión

La matriz de confusión es una herramienta analítica fundamental en clasificación, que permite una evaluación detallada de la precisión del modelo en la clasificación de las muestras en sus categorías correctas.

- **Descripción:** En el contexto de la detección de cáncer, la matriz incluirá cuatro cuadrantes que representan los verdaderos positivos (TP), falsos

positivos (FP), verdaderos negativos (TN), y falsos negativos (FN). Los TP y TN corresponden a las clasificaciones correctas de muestras malignas y benignas, respectivamente, mientras que los FP y FN representan errores en la clasificación.

- **Interpretación:** La matriz ofrece una perspectiva del rendimiento del modelo, mostrando no solo cuántas muestras fueron correctamente identificadas, sino también los tipos de errores cometidos. Esta es una muy útil herramienta para ajustar el modelo y mejorar su precisión, especialmente en aplicaciones médicas donde los FN (casos malignos no detectados) pueden tener consecuencias graves. Un buen modelo tiene la mayoría de sus valores concentrados en su diagonal, donde se encuentran todos los valores correctamente clasificados.

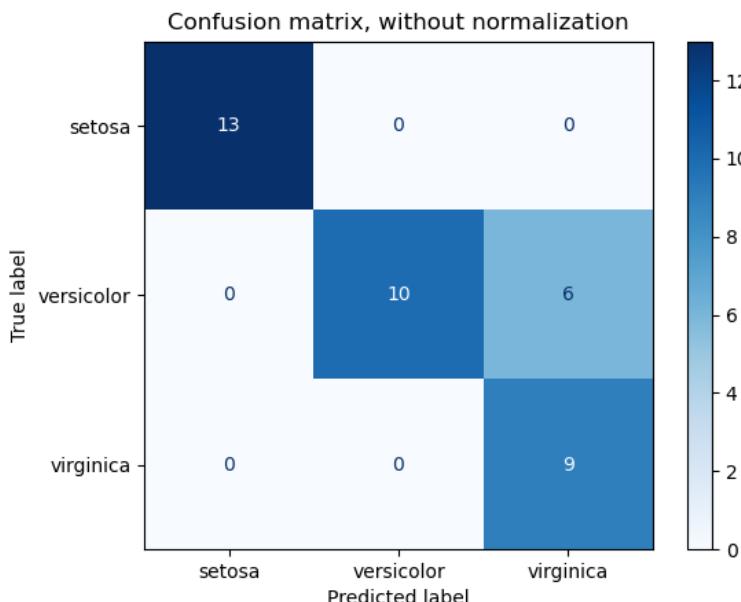


Figura 4.17: Ejemplo de matriz de confusión en el Modelo Iris de Scikit-learn.

Fuente: https://scikit-learn.org/stable/_auto_examples/model_selection/plot_confusion_matrix.html

4.5.3. Gráfico de Precisión/Recall en función de los Umbrales

Este gráfico proporciona una perspectiva valiosa sobre cómo la selección de diferentes umbrales de decisión afecta la precisión y el recall del modelo, dos métricas críticas en la evaluación del rendimiento del clasificador.

- **Descripción:** Se traza la precisión y el recall para una serie de valores de umbral. La precisión se define como la proporción de identificaciones positivas correctas (TP) entre todas las identificaciones positivas (TP + FP), y el recall es la proporción de positivos correctos identificados entre todos los positivos reales (TP + FN).

- **Interpretación:** Al ajustar el umbral de decisión, se puede observar cómo cambian estas métricas. Un gráfico bien balanceado indicará un umbral donde tanto la precisión como el recall son óptimos, ayudando a elegir el mejor punto de compromiso para la aplicación específica, especialmente importante en el diagnóstico médico donde se busca maximizar el recall sin sacrificar demasiada precisión.

4.5.4. Curva ROC

La Curva ROC es una representación gráfica que evalúa la calidad de los modelos de clasificación binaria a diferentes umbrales de clasificación, siendo un indicador clave de su capacidad para discriminar entre clases [17].

- **Descripción:** La curva ROC traza la tasa de verdaderos positivos (sensibilidad) contra la tasa de falsos positivos (1-especificidad) para diferentes umbrales. Este gráfico ayuda a visualizar la relación entre sensibilidad y especificidad a lo largo de varios umbrales.
- **Interpretación:** Un área bajo la curva (AUC) grande (cercana a 1) indica una excelente capacidad del modelo para distinguir entre clases benignas y malignas. Un AUC bajo podría señalar que el modelo no es mejor que un clasificador aleatorio y puede necesitar revisión o ajuste en su enfoque.

Modelo bueno

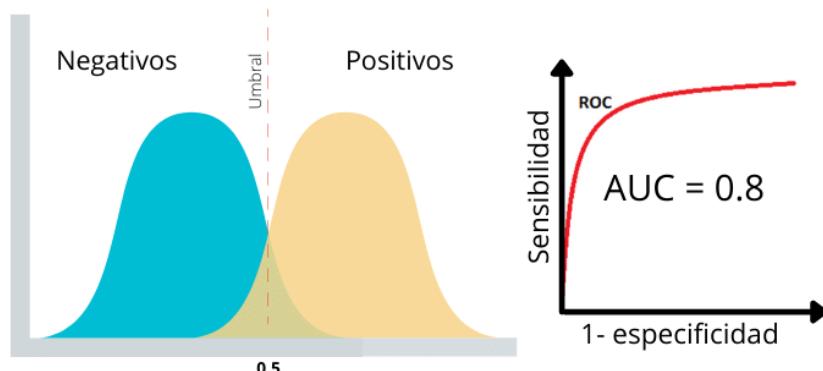


Figura 4.18: Ejemplo de curva ROC
Fuente: <https://abdatum.com/ciencia/curvas-roc>

5. Experimentos

5.1. Introducción

Tras haber adquiridos los conocimientos necesarios en el capítulo previo nos dispondremos a crear un escenario adecuado para la realización de los experimentos relacionados con un caso desigual de clases.

5.2. Configuración del escenario

Para el desarrollo de los diferentes experimentos, se desarrollará un autoencoder para la detección de muestras de un conjunto de cáncer de mama, utilizado el famoso dataset *Breast Cancer Wisconsin*, modificando dicho conjunto para crear datos anómalos, desproporcionando la cantidad de datos disponibles para su resultado. Se ha desarrollado un Notebook para poder facilitar el entendimiento y seguimiento del proceso.

5.2.1. Carga y Análisis de los datos

Para este caso como se ha comentado previamente hemos escogido el conjunto de datos *Breast Cancer Wisconsin*, el cuál podemos importar directamente de la librería datasets de la clase sklearn. Además, utilizaremos las librerías pandas y numpy para manipular los datos y poder tratar con los datos.

Carga y Análisis de los datos

Afortunadamente el repositorio datasets de sklearn incorpora el `load_breast_cancer`, por lo que podremos cargarlo directamente

```
import numpy as np
import pandas as pd
from sklearn.datasets import load_breast_cancer
dataset = load_breast_cancer(as_frame=True)
data = dataset.frame
```

Podemos cómo en este caso las clases posibles para los ejemplos son "malignant" o "benign".

```
dataset.target_names
array(['malignant', 'benign'], dtype='|U9')
```

Usando la función `feature_names` seguida de la función `size` podemos ver la cantidad de atributos o características cada ejemplo, que en este caso consta de 30.

```
dataset.feature_names.size
30
```

Figura 5.1: Propiedades del dataset Breast Cancer Wisconsin

Como podemos ver el conjunto de datos se divide en un problema de clasificación binaria, donde un ejemplo del conjunto, que contiene un total de 30 atributos, puede ser clasificado como *malignant* o *benign*.

Podemos ver los diferentes valores posibles que pueden tomar los diferentes atributos para poder entender más como se compone nuestro conjunto de datos, donde podemos ver el radio, la textura, el area etc... de la muestra a analizar. Adicionalmente se nos muestra las estadísticas más relevantes de cada uno de los atributos, como la media, desviación típica (std), el valor máximo...

Podemos ver los datos de nuestro conjunto de datos para poder entenderlos mejor.

```
dataset.data.head()
```

	mean radius	mean texture	mean perimeter	mean area	mean smoothness	mean compactness	mean concavity	mean concave points	mean symmetry	mean fractal dimension	...	worst radius	worst texture	worst perimeter	worst area	worst smoothness
0	17.99	10.38	122.80	1001.0	0.11840	0.27760	0.3001	0.14710	0.2419	0.07871	...	25.38	17.33	184.60	2019.0	0.1622
1	20.57	17.77	132.90	1326.0	0.08474	0.07864	0.0869	0.07017	0.1812	0.05667	...	24.99	23.41	158.80	1956.0	0.1238
2	19.69	21.25	130.00	1203.0	0.10960	0.15990	0.1974	0.12790	0.2069	0.05999	...	23.57	25.53	152.50	1709.0	0.1444
3	11.42	20.38	77.58	386.1	0.14250	0.28390	0.2414	0.10520	0.2597	0.09744	...	14.91	26.50	98.87	567.7	0.2098
4	20.29	14.34	135.10	1297.0	0.10030	0.13280	0.1980	0.10430	0.1809	0.05883	...	22.54	16.67	152.20	1575.0	0.1374

5 rows × 30 columns

Con la función describe vemos las estadísticas más relevantes del conjunto de datos (media, desviación típica, valor máximo...)

```
data.describe()
```

	mean radius	mean texture	mean perimeter	mean area	mean smoothness	mean compactness	mean concavity	mean concave points	mean symmetry	mean fractal dimension	...	worst texture	worst perimeter
count	569.000000	569.000000	569.000000	569.000000	569.000000	569.000000	569.000000	569.000000	569.000000	569.000000	...	569.000000	569.000000
mean	14.127292	19.289649	91.969033	654.889104	0.096360	0.104341	0.088799	0.048919	0.181162	0.062798	...	25.677223	107.261213
std	3.524049	4.301036	24.298981	351.914129	0.014064	0.052813	0.079720	0.038803	0.027414	0.007060	...	6.146258	33.602542
min	6.981000	9.710000	43.790000	143.500000	0.052630	0.019380	0.000000	0.000000	0.106000	0.049960	...	12.020000	50.410000
25%	11.700000	16.170000	75.170000	420.300000	0.086370	0.064920	0.029560	0.020310	0.161900	0.057700	...	21.080000	84.110000
50%	13.370000	18.840000	86.240000	551.100000	0.095870	0.092630	0.061540	0.033500	0.179200	0.061540	...	25.410000	97.660000
75%	15.780000	21.800000	104.100000	782.700000	0.105300	0.130400	0.130700	0.074000	0.195700	0.066120	...	29.720000	125.400000
max	28.110000	39.280000	188.500000	2501.000000	0.163400	0.345400	0.426800	0.201200	0.304000	0.097440	...	49.540000	251.200000

8 rows × 31 columns

Figura 5.2: Datos del dataset Breast Cancer Wisconsin

A continuación se guardarán los datos del conjunto (sin contar las clases) en la variable X, y las clases de los diferentes ejemplos en la variable y.

Guardamos en X los datos de los atributos así como en y las clase de cada ejemplo.

```
X = dataset.data
y = dataset.target
```

Figura 5.3: Asignación de variables X e y

Una vez hemos contemplado los datos que contendrá nuestro modelo, es

importante ver que atributos son prescindibles y por lo cuál podremos borrar.

En la siguiente figura se contempla el **mapa de calor** que contiene la correlación entre los diferentes atributos de nuestro modelo, dónde entre 2 atributos se contempla una correlación en un rango entre [-1,1].

- **Correlación cercana a 1** significa que tienen una correlación fuerte esos 2 atributos.
- **Correlación cercana a -1** significa que tienen una correlación inversa esos 2 atributos.
- **Correlación cercana a 0** significa que no tienen correlación alguna esos 2 atributos.

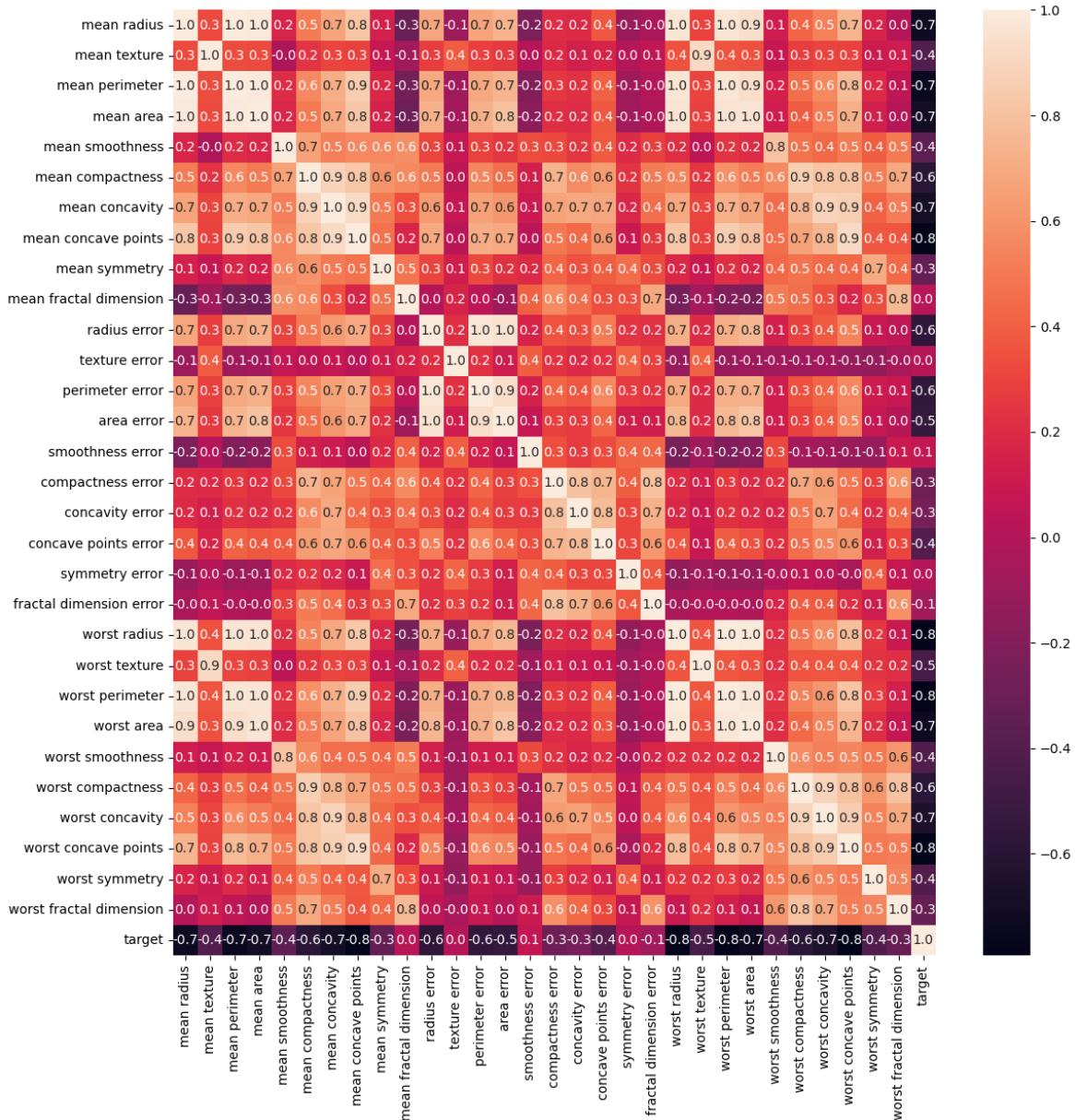


Figura 5.4: Mapa de calor de los diferentes atributos

Utilizando dicho mapa de calor podemos deducir que variables son irrelevantes. Para ello nos vamos a basar en 2 criterios.

1. Dicho atributo se convertirá en candidato para su eliminación si tiene una correlación cercana a 0 con la clase.
2. Si además dicho atributo tiene una correlación fuerte o inversa con otro atributo que no sea él mismo ni la clase podemos deducir que su información es redundante y será candidato para su eliminación.

En caso de que se cumplan estas 2 propiedades eliminaremos dicho atributo. En nuestro conjunto de datos podemos ver que este criterio se cumple con el atributo *worst fractal dimension* donde tiene 0 de correlación con el la clase objetivo y una alta correlación con otros atributos, como con *worst fractal dimension*, por lo que procederemos a eliminar el primero de estos atributos.

```
X = dataset.data.drop("mean fractal dimension", axis=1)
```

Figura 5.5: Eliminación del atributo *mean fractal dimension*

Hay que tener en cuenta que el criterio anteriormente mencionado como motivo de eliminación no asegura que el modelo vaya a ser mejor, pero intuitivamente se estima que probablemente pueda serlo, por lo que este modelo esta sujeto a errores y habrá que modificarlo futuramente para ver si varía.

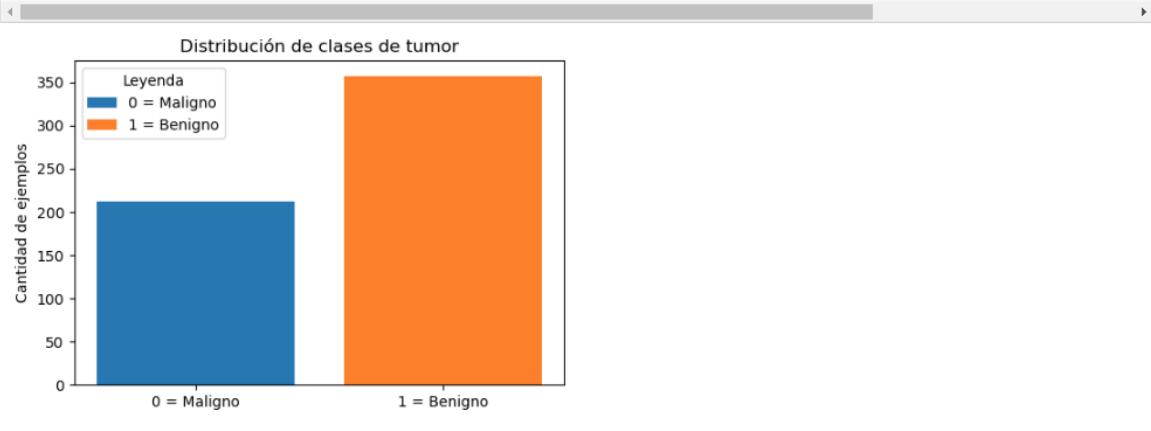
Al contemplar las cantidades de datos disponibles de cada una de las clases podemos contemplar que este escenario no cumple las expectativas, ya que mantiene unas proporciones ideales de cantidad de elementos por clase, donde hay un total de 357 datos de clase benigna y 212 de clase maligna.

Podemos visualizar la cantidad de elementos que hay por clase

```
data['target'].value_counts().sort_index().get(1,0)  
357  
  
data['target'].value_counts().sort_index().get(0,0)  
212
```

A continuación se muestra de manera gráfica la distribución de los datos del dataset, dividiéndolos por clase

```
# Calculamos los conteos de cada clase  
conteos = data['target'].value_counts().sort_index() # Asegurar el orden correcto de las clases  
  
# Crear una figura y un eje  
plt.figure(figsize=(6, 4))  
# Dibujar un gráfico de barras para cada clase con una etiqueta  
plt.bar(0, conteos.get(0, 0), color="#1f77b4", label='0 = Maligno') # Asumimos que existe la posibilidad de qu  
plt.bar(1, conteos.get(1, 0), color="#ff7f0e", label='1 = Benigno') # Asumimos que existe la posibilidad de qu  
  
plt.bar(conteos.index, conteos.values, color=['#1f77b4', '#ff7f0e']) # Colores azul y naranja  
  
# Agregar título y etiquetas  
plt.title('Distribución de clases de tumor')  
plt.xlabel('Clase (0 = Maligno, 1 = Benigno)')  
plt.ylabel('Cantidad de ejemplos')  
  
# Configurar las etiquetas del eje x para que muestren claramente qué representa cada barra  
plt.xticks([0, 1], ['0 = Maligno', '1 = Benigno'])  
plt.legend(title='Leyenda')  
# Mostrar el gráfico  
plt.show()
```



Como podemos ver en la distribución de los datos, no nos encontramos con un caso **anómalo**, así que deberemos recrearlo.

Figura 5.6: Cantidad de elementos por clase.

5.2.2. Recreación de escenario Anómalo

Como hemos mencionado anteriormente el escenario de este conjunto de datos no es el idóneo para el escenario deseado, por lo que modificaremos el conjunto de datos perteneciente a la clase *Malign* con el propósito de desbalancear la cantidad de elementos por clase.

Para ello vamos a utilizar una variable auxiliar **porcentaje_anomalias** donde podremos modificar el porcentaje de instancias malignas existentes en el conjunto de datos. Aprovechándonos de dicha variable modificaremos el tamaño del conjunto de datos, modificando el tamaño total tanto de la clase maligna como el

Modificación del conjunto de datos

Cómo queremos detectar casos anómalos y en este caso la distribución de los ejemplos no cumple con esta condición deberemos modificar la cantidad de datos. Para ello escogeremos un porcentaje de casos malignos que sea anómalo.

```
porcentaje_anomalias = 0.3 # % de instancias malignas
```

Creamos máscaras para los datos benignos y los datos malignos, modificamos la proporcionalidades y mezclamos los conjuntos.

```
mask = y == 1 # Instancias benignas
X_benignas = X[mask]
X_malignas = X[~mask]
num_malignas = int(porcentaje_anomalias * len(X_benignas))
X_nuevo = np.concatenate([X_malignas[:num_malignas], X_benignas], axis=0)
y_nuevo = np.concatenate([np.zeros(num_malignas), np.ones(len(X_benignas))], axis=0)

pd.Series(y_nuevo).value_counts()

1.0    357
0.0    107
Name: count, dtype: int64
```

Figura 5.7: Modificación del conjunto de datos

de la clase benigna.

Tras la modificación de los elementos vemos gráficamente como quedaría la distribución del conjunto de datos:

Podemos comprobar que ahora hay un caso anómalo de ejemplos con la clase maligno.

```
# Calculamos los conteos de cada clase
conteos = pd.Series(y_nuevo).value_counts().sort_index() # Asegurar el orden correcto de las clases

# Creación de la figura
plt.figure(figsize=(6, 4))
# Dibujar un gráfico de barras para cada clase con una etiqueta
plt.bar(0, conteos.get(0, 0), color="#1f77b4", label='0 = Maligno') # Asumimos que existe la posibilidad de que no existan datos de clase 0
plt.bar(1, conteos.get(1, 0), color="#ff7f0e", label='1 = Benigno') # Asumimos que existe la posibilidad de que no existan datos de clase 1

plt.bar(conteos.index, conteos.values, colors=['#1f77b4', '#ff7f0e']) # Colores azul y naranja

# Agregar título y etiquetas
plt.title('Distribución de clases de cáncer')
# plt.xlabel('Clase (0 = Maligno, 1 = Benigno)')
plt.ylabel('Cantidad de ejemplos')

# Configurar las etiquetas del eje x para que muestren claramente qué representa cada barra
plt.xticks([0, 1], ['0 = Maligno', '1 = Benigno'])
plt.legend(title='Leyenda')
# Mostrar el gráfico
plt.show()
```

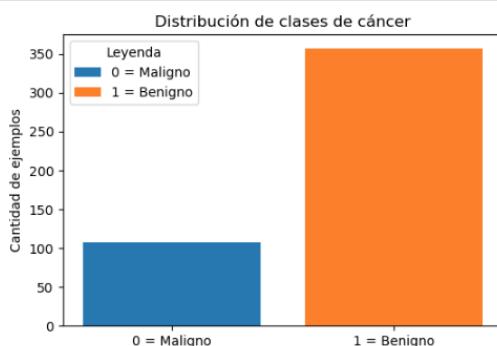


Figura 5.8: Cantidad de elementos por clase tras la modificación

5.2.3. División del conjunto de datos

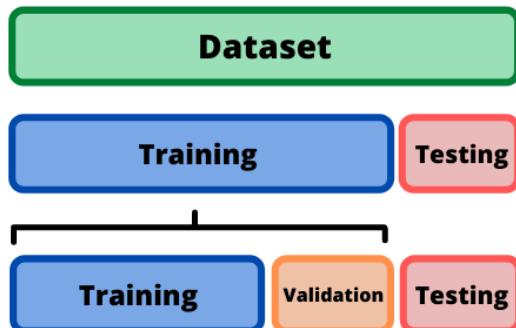


Figura 5.9: Separación del conjunto de datos

Fuente: <https://dhavalpatel2101992.wordpress.com/2021/05/21/kaggle-titanic-dataset-cleaning-split-data-into-train-validation-and-test-set/>

Deberemos separar el conjunto de datos en diferentes partes para entrenar nuestro modelo adecuadamente. Para ello, se realizan dos divisiones:

1. Se divide el conjunto de datos en dos subconjuntos:
 - Un subconjunto de **entrenamiento** con el 80 % de los datos
 - Un subconjunto de **prueba** (Test) con el 20 % de los datos.
2. Se extrae del subconjunto de entrenamiento un subconjunto de **validación** con el 20 % de su tamaño.

Preparación del conjunto de entrenamiento

Dividiremos el conjunto de datos en un conjunto de entrenamiento y otro de validación (o test) para poder entrenar nuestro modelo

```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X_nuevo, y_nuevo, test_size=0.2, random_state=42)
X_train, X_valid, y_train, y_valid = train_test_split(X_train, y_train, test_size=0.2, random_state=42)
```

Figura 5.10: Separación en 3 conjuntos de datos

El objetivo de esta división es tener un subconjunto para entrenar el modelo buscando los pesos adecuados, un subconjunto para corregir los parámetros del modelo, y un último modelo para poder evaluar nuestro modelo.

Además vamos dividir cada uno de los 3 subconjuntos divididos en 2 subconjuntos en función de las clases de los datos. De este modo queda la siguiente organización:

- Conjunto de entrenamiento
 - Elementos de clase 0 del Conjunto de entrenamiento.

- Elementos de clase 1 del Conjunto de entrenamiento.
- Conjunto de validación
 - Elementos de clase 0 del Conjunto de validación.
 - Elementos de clase 1 del Conjunto de validación.
- Conjunto de test
 - Elementos de clase 0 del Conjunto de test.
 - Elementos de clase 1 del Conjunto de test.

Vamos a crear máscaras para dividir los conjuntos de datos según las clases de cada uno de sus ejemplos.

```
# Mascaras para identificar las clases en los conjuntos
mask_clase_0_tr = (y_train == 0)
mask_clase_0_v = (y_valid == 0)
mask_clase_0_test = (y_test == 0)
mask_clase_1_tr = (y_train == 1)
mask_clase_1_v = (y_valid == 1)
mask_clase_1_test = (y_test == 1)

# Filtra X para quedarte solo con las instancias de la clase deseada
X_train_clase_0_S = X_train[mask_clase_0_tr]
X_valid_clase_0_S = X_valid[mask_clase_0_v]
X_test_clase_0_S = X_test[mask_clase_0_test]
X_train_clase_1_S = X_train[mask_clase_1_tr]
X_valid_clase_1_S = X_valid[mask_clase_1_v]
X_test_clase_1_S = X_test[mask_clase_1_test]
```

Figura 5.11: División de los conjuntos según las clases de sus elementos.

5.2.4. Normalización del conjunto de datos

La normalización de los datos es un paso crítico en el preprocesamiento de datos para algoritmos de aprendizaje automático, especialmente en tareas relacionadas con el análisis de datos complejos como los biomédicos.

En la subsección *Dropout and Batch Normalization* 4.3.5 utilizamos normalización entre cada capa de una red neuronal, pero es recomendable realizar una normalización de los datos previamente antes de entrenar con estos.

En este proyecto, se ha utilizado el método StandardScaler para normalizar el conjunto de datos.

Existen diferentes modos de normalizar nuestros datos, en nuestro caso utilizaremos StandardScaler para poder normalizar los datos de entrada al modelo. Estoy tendré como resultado que la desviación típica sea 1.

```
import pickle
from sklearn.preprocessing import StandardScaler
from sklearn.preprocessing import MinMaxScaler
normalizador=StandardScaler().fit(X_train_clase_1_S)
X_train_clase_0=normalizador.transform(X_train_clase_0_S)
X_valid_clase_0=normalizador.transform(X_valid_clase_0_S)
X_test_clase_0=normalizador.transform(X_test_clase_0_S)
X_train_clase_1=normalizador.transform(X_train_clase_1_S)
X_valid_clase_1=normalizador.transform(X_valid_clase_1_S)
X_test_clase_1=normalizador.transform(X_test_clase_1_S)

X_train_norm=normalizador.transform(X_train)
X_valid_norm=normalizador.transform(X_valid)
X_test_norm=normalizador.transform(X_test)

with open('normalizador.pkl', 'wb') as file:
    pickle.dump(normalizador, file) # Se guarda un fichero con la configuración de normalización
```

Figura 5.12: Normalización de los datos.

El StandardScaler transforma cada característica del conjunto de datos de modo que tenga una media (promedio) de cero y una desviación estándar de uno. Este escalado asegura que las características tengan la misma escala y, por lo tanto, contribuyan equitativamente al proceso de aprendizaje del modelo, evitando que características con rangos más amplios dominen el entrenamiento.

La importancia de este proceso radica en su capacidad para mejorar la convergencia de los algoritmos de optimización durante el entrenamiento, facilitando un aprendizaje más rápido y estable.

Adicionalmente se ha guardado en un fichero de configuración los parámetros de la normalización.

5.2.5. Creación del Autoencoder

Una vez ya tenemos los datos listos procederemos a crear nuestro modelo de Red Neuronal, que en este caso se trata de un tipo llamado Autoencoder. En la sección 4.4 *Autoencoders* ya hablamos más en detalle sobre cómo funcionaba esta red. La lógica seguida para crear esta red consta de una compresión de los diferentes atributos en un **espacio latente** del ejemplo para luego volver a descomprimir los datos. El resultado que buscamos es el **error de reconstrucción**, que consta de la diferencia que haya entre los datos de entrada y los datos de salida.

Nosotros utilizaremos esta red para introducir aquellas muestras de clase Maligna y el modelo aprenda a reconstruirla tras comprimir dichos datos.

Vamos a intentar seguir el modelo de la siguiente estructura:

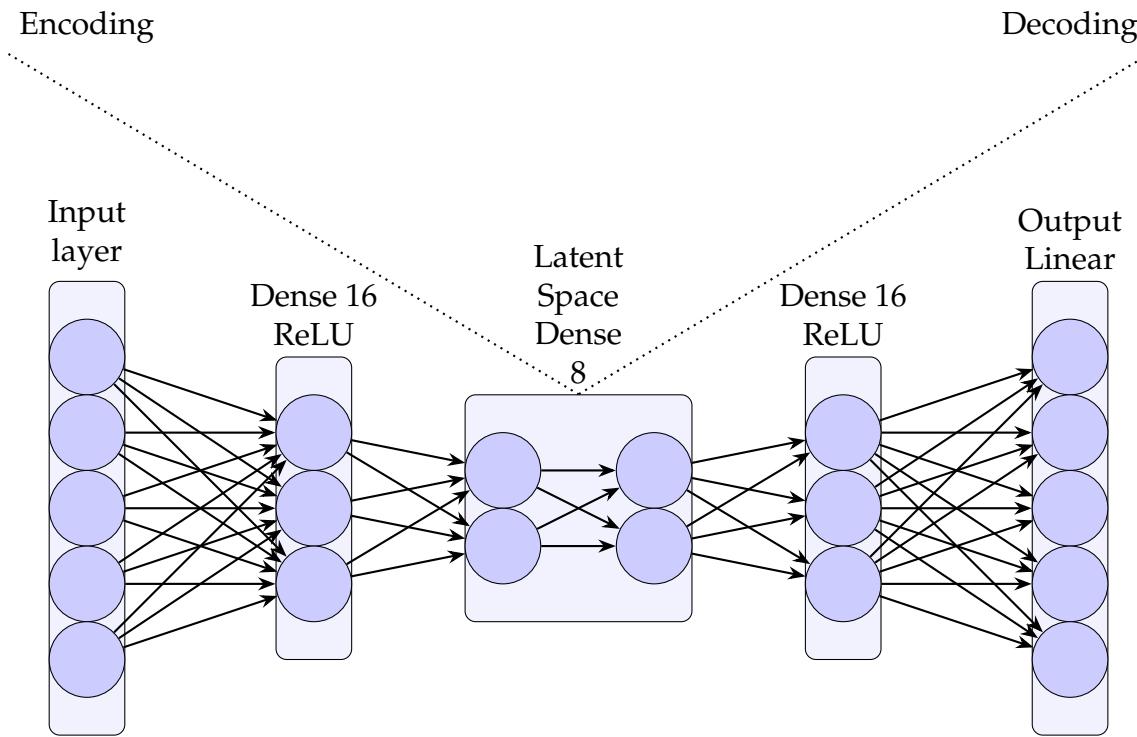


Figura 5.13: Visualización detallada de la estructura de la red neuronal autoencoder

Podemos ver que consta de 6 capas divididas en la siguiente estructura:

1. Codificación:

- 1.1 Capa de entrada de 29 neuronas con los datos de los atributos.
- 2.2 Capa de compresión con 16 neuronas.
- 3.3 Capa de compresión con 8 neuronas perteneciente al espacio latente.

2. Decodificación:

- 1.1 Capa de descompresión con 8 perteneciente al espacio latente.
- 2.2 Capa de descompresión con 16 neuronas.
- 3.3 Capa de salida con 29 neuronas con la salida similar a la entrada.

Para ello primeramente se definen el tamaño de las diferentes capas.

Escogemos el tamaño de neuronas de cada capa.

```
input_dim = X_train_norm.shape[1]
encoding_dim = 16 # Dimensión de la codificación
hidden_dim = 8
```

Figura 5.14: Enter Caption

Se añadirá regularización l1 en la capa de entrada para prevenir el sobreajuste al eliminar parte de la complejidad que pueda tener.

La red neuronal empleará funciones de activación ReLU (Rectified Linear Unit) para todas las capas intermedias, debido a su eficacia y simplicidad en la promoción de la no-linealidad sin afectar significativamente la velocidad de convergencia durante el entrenamiento. La función ReLU se caracteriza por su habilidad para activar una neurona sólo si la entrada es positiva, lo que ayuda a reducir el problema del decaimiento del gradiente durante las fases de retropropagación, permitiendo que la red aprenda más eficientemente y se mantenga estable. Por otro lado, la capa de salida utilizará una función de activación lineal, que es esencial para este tipo de autoencoder, ya que permite reconstruir la salida en la misma escala que la entrada original.

Para construir la estructura usaremos **regularización l1**, para de este modo poder prevenir el sobreajuste. La estructura se dividirá en capas encoder y decoder (codificación y decodificación).

```
from keras import regularizers

from tensorflow.keras.models import Model
from tensorflow.keras.layers import Input, Dense, BatchNormalization
from tensorflow.keras.optimizers import Adam, SGD

learning_rate = 1e-2

input_layer = Input(shape=(input_dim, ))
encoder = Dense(encoding_dim, activation="relu", activity_regularizer=regularizers.l1(learning_rate))(input_layer)
encoder = Dense(hidden_dim, activation="relu")(encoder)
decoder = Dense(hidden_dim, activation="relu")(encoder)
decoder = Dense(encoding_dim, activation="relu")(decoder)
decoder = Dense(input_dim, activation="linear")(decoder)

autoencoder = Model(inputs=input_layer, outputs=decoder)
autoencoder.summary()

Model: "model"
-----  

Layer (type)          Output Shape         Param #
-----  

input_1 (InputLayer)   [(None, 29)]        0  

dense (Dense)          (None, 16)           480  

dense_1 (Dense)        (None, 8)            136  

dense_2 (Dense)        (None, 8)            72  

dense_3 (Dense)        (None, 16)           144  

dense_4 (Dense)        (None, 29)           493  

-----  

Total params: 1325 (5.18 KB)  

Trainable params: 1325 (5.18 KB)  

Non-trainable params: 0 (0.00 Byte)
```

Figura 5.15: Codificación de la red neuronal

El modelo se compila utilizando el optimizador Adam, reconocido por su eficiencia en diversos problemas de aprendizaje automático gracias a su manejo adaptativo de las tasas de aprendizaje. Se utiliza la métrica 'accuracy' para monitorear la precisión durante el entrenamiento, aunque para tareas de reconstrucción como esta, es más indicativo observar la pérdida directamente. El 'mse' (error cuadrático medio) se emplea como función de pérdida, cuantificando la diferencia entre los valores originales y las reconstrucciones generadas por el

autoencoder. Previamente se utilizó otras métrica como 'mae' (error absoluto medio) que también representa la perdida en la reconstrucción, pero tras diferentes pruebas se ve una pequeña mejoría usando la primera de estas opciones.

Se utiliza ModelCheckpoint para guardar el modelo después de cada época solo si es el mejor hasta el momento en términos de pérdida en el conjunto de validación, lo que ayuda a prevenir la pérdida de un modelo bien entrenado en caso de un entrenamiento posterior subóptimo.

Se inicia el entrenamiento del modelo usando los datos de entrenamiento con de la clase Maligna, donde las entradas y las etiquetas son las mismas, reflejando el enfoque no supervisado del autoencoder. Además, se utiliza un conjunto de validación para monitorear el desempeño del modelo en datos no vistos durante el entrenamiento.

El callback EarlyStopping es crucial para evitar el sobreajuste, deteniendo el entrenamiento si la pérdida en el conjunto de validación no mejora durante 20 épocas consecutivas. Este enfoque asegura que el entrenamiento se detenga en un punto óptimo, preservando recursos y evitando la degradación del modelo.

Aprendizaje del modelo

Una vez construido el autoencoder los ajustaremos para que nos de un aprendizaje correcto a nuestro caso, para ello hay que probar con diferentes optimizadores y funciones de perdida. En este caso el optimizador Adam y la función de perdida mse (error cuadrático medio).

```
autoencoder.compile(metrics=['accuracy'],optimizer=Adam(), loss='mse')

from tensorflow.keras import layers, callbacks
from keras.callbacks import ModelCheckpoint, TensorBoard
cp = ModelCheckpoint(filepath="autoencoder.h5",
                      save_best_only=True,
                      verbose=0)
history = autoencoder.fit(X_train_clase_1, X_train_clase_1, epochs=300, batch_size=128, shuffle=True, validation_data=(X_val_clase_1, X_val_clase_1), callbacks=[cp],)

2/2 [=====] - 0s 31ms/step - loss: 0.9065 - accuracy: 0.1212 - val_loss: 0.8064 - val_accuracy: 0.15
52
Epoch 43/300
2/2 [=====] - 0s 30ms/step - loss: 0.8998 - accuracy: 0.1169 - val_loss: 0.7980 - val_accuracy: 0.15
52
Epoch 44/300
2/2 [=====] - 0s 31ms/step - loss: 0.8924 - accuracy: 0.1169 - val_loss: 0.7898 - val_accuracy: 0.15
52
Epoch 45/300
2/2 [=====] - 0s 32ms/step - loss: 0.8853 - accuracy: 0.1255 - val_loss: 0.7819 - val_accuracy: 0.15
52
Epoch 46/300
2/2 [=====] - 0s 32ms/step - loss: 0.8782 - accuracy: 0.1212 - val_loss: 0.7741 - val_accuracy: 0.17
24
Epoch 47/300
2/2 [=====] - 0s 31ms/step - loss: 0.8706 - accuracy: 0.1255 - val_loss: 0.7664 - val_accuracy: 0.17
24
Epoch 48/300
2/2 [=====] - 0s 32ms/step - loss: 0.8637 - accuracy: 0.1255 - val_loss: 0.7590 - val_accuracy: 0.15
```

Figura 5.16: Compilación y ajuste del modelo

Una vez nuestro modelo ya es completamente funcional utilizaremos el error de reconstrucción al realizar una predicción para clasificar los conjuntos. Si ahora realizamos una predicción con la clase Maligna el error de reconstrucción será muy alto ya que la configuración de los pesos establecidos no será conveniente para reconstruirlo tras comprimirlo, mientras que si en la predicción se introduce una muestra de la clase Benigna los pesos serán óptimos para su comprensión y

posterior reconstrucción, por lo que el dato saliente del modelo será muy parecido al inicial, lo que conlleva a un error de reconstrucción muy bajo. Nos aprovecharemos de dicho error para clasificarlos las diferentes muestras, en función de lo pequeño que sea el error. Para ello se utilizará un umbral fijo establecido previamente, donde si el error es menor al umbral será reconocida la muestra como Benigno, y en caso contrario será Maligno.

Aciertos y fallos

Una vez el modelo ha aprendido correctamente los datos es momento de ver los fallos y aciertos en el modelo, para de este modo asegurarnos que el aprendizaje ha sido correcto.

```
# Calcular el error de reconstrucción en el conjunto de prueba
X_test_pred = autoencoder.predict(X_test_norm)
test_reconstruction_errors = np.mean(np.power(X_test_norm - X_test_pred, 2), axis=1)
umbral_fijo = 0.4
y_pred = [1 if e < umbral_fijo else 0 for e in test_reconstruction_errors]

3/3 [=====] - 0s 1ms/step
```

Figura 5.17: Clasificación utilizando el error de reconstrucción.

5.3. Marco experimental

Para poder afirmar que tanto el Autoencoder creado cómo la forma en la que hemos abordado el problema planteado es óptimo y es completamente funcional se realizarán diferentes experimentos con el propósito de corroborar la resolución del problema en diferentes escenarios posibles. Se utilizarán diferentes herramientas para la representación de información cómo puede ser la **matriz de confusión**, la **curva del error del modelo** y la **curva ROC**.

5.3.1. Variación la cantidad de muestras anómalas

En este **experimento 1**, el objetivo principal es evaluar la capacidad del autoencoder para identificar muestras anómalas en un conjunto de datos donde se manipula activamente la proporción de muestras malignas. Dado que el conjunto de datos de cáncer de mama de Wisconsin originalmente presenta una distribución balanceada de muestras benignas y malignas, la alteración en el número de muestras malignas crea un escenario que simula condiciones de rareza o anomalía.

Metodología

- 1. Preparación de los datos:** Se parte del conjunto de datos original, seleccionando una fracción de las muestras malignas para incluir en el conjunto de datos. Esto se realiza varias veces para crear múltiples conjuntos de datos con diferentes proporciones de muestras malignas, por ejemplo, 5 %, 10 %, 20 %, y 30 % del total de las muestras.

2. **Configuración del autoencoder:** Se utilizará el autoencoder cuya arquitectura ha sido definida previamente y se utilizará indistintivamente con cada una de las proporciones de las muestras, dónde este se entrenará exclusivamente con las muestras malignas del conjunto de datos.
3. **Entrenamiento del modelo:** Cada versión del autoencoder se entrena con su respectivo conjunto de datos. El entrenamiento se realiza registrando la pérdida de reconstrucción durante varias epoch, utilizando una tasa de aprendizaje y parámetros de regularización previamente especificados.
4. **Evaluación del modelo:** La evaluación se centra en medir la efectividad del autoencoder para clasificar correctamente las muestras malignas como anómalas, viendo como varían la cantidad de muestras clasificadas en función de la cantidad de muestras Malignas usadas en el entrenamiento del modelo.

5.3.2. Variación del umbral ante el error de reconstrucción

En el **experimento 2** nos enfocaremos en analizar cómo la variación del umbral de error de reconstrucción afecta la capacidad del autoencoder para clasificar muestras como anómalas o normales. Este umbral determina el nivel de error de reconstrucción a partir del cual una muestra es considerada anómala. Ajustar este umbral es crucial para optimizar el equilibrio entre la sensibilidad y especificidad del modelo.

Metodología

1. **Configuración inicial:** Utilizando el conjunto de datos y el modelo de autoencoder ya entrenado en experimentos previos, este experimento se centra exclusivamente en la post-evaluación del modelo, sin requerir entrenamiento adicional.
2. **Determinación del umbral:** Se inicia con un umbral basado en el error de reconstrucción medio de las muestras benignas del conjunto de entrenamiento. A partir de este punto de referencia, se varía el umbral en un rango definido, variando entre el 0.1 y 0.5.
3. **Evaluación del modelo:** Se medirá la posibilidad de clasificar las diferentes muestras hasta encontrar el umbral óptimo donde se clasifica el mayor porcentaje de muestras en su clase correspondiente.

5.3.3. Comparación con modelos de Machine Learning

El **experimento 3** tiene como objetivo comparar el desempeño del autoencoder en la detección de muestras anómalas con otros modelos de machine learning establecidos y anteriormente explayados en [4.2 Intro to Machine Learning](#). Esta comparación ayudará a determinar la viabilidad y eficacia del autoencoder frente a

métodos alternativos en la tarea de detectar cáncer en un contexto de desequilibrio de clases.

Para la realización de la comparación se utilizará el caso más anómalo, en el que solo existe un 10% de muestras malignas.

Metodología

1. **Selección de modelos:** Se eligen varios modelos de Machine Learning comúnmente utilizados en tareas de clasificación y detección de anomalías, como el Random Forest, Máquinas de Soporte Vectorial (SVM) y KNN. Cada modelo será configurado y entrenado para manejar la misma tarea que el autoencoder.
2. **Preparación de datos:** Utilizando el mismo conjunto de datos utilizado para el autoencoder, cada modelo se entrena y evalúa. Se asegura que todos los modelos se enfrenten a la misma distribución de clases, y se aplica la misma técnica de preprocesamiento para mantener la consistencia.
3. **Evaluación y métricas:** Cada modelo se evalúa en términos de varias métricas clave, incluyendo precisión, recall y F1-score. Específicamente, el interés está en su capacidad para identificar correctamente las muestras malignas (anomalías) dentro del conjunto de datos.
4. **Análisis estadístico:** Se realiza un análisis estadístico para determinar si las diferencias en el desempeño entre el autoencoder y otros modelos son significativas.

5.3.4. Comparación con Entrenamiento exclusivo con muestras de clase Maligna

Este **experimento 4** compara dos modelos de autoencoder, uno entrenado exclusivamente con muestras malignas y otro entrenado únicamente con muestras benignas. El objetivo es evaluar cómo cada modelo especializado se desempeña en la identificación de muestras anómalas, respectivamente, y determinar cuál enfoque es más efectivo para la detección de anomalías en el contexto del cáncer de mama.

La idea de utilizar un Modelo con solo muestras Malignas, las cuales son una cantidad muy inferior ante las muestras Benignas, puede parecer obviamente que va a resultar en un modo Negativo, pero hay que tener en cuenta la naturaleza de cada una de las clases. El consiguiente experimento se realiza tras la idea del patrón de valores que tiene una muestra Maligna es significativamente más específico que el que tiene una muestra Benigna, puesto que los valores para que se considere como Maligna están más concretados en un rango de valores cerrados, mientras que en el caso de las Benigna solo tiene que estar fuera de dicho rango. Experimentaremos para ver si dicho rango de valores compensa la ínfima cantidad de muestras disponibles a la hora de realizar el aprendizaje.

Se usará el error de construcción de manera inversa, ya que ahora el error de reconstrucción será significativamente inferior cuando se intente predecir una muestra de clase benigna.

El porcentaje de muestras Malignas se mantendrá al 10 % a lo largo de todo el experimento, manteniendo en todo momento la anomalía en la distribución de muestras por clases.

Metodología

1. Configuración de los modelos:

- **Modelo A:** Autoencoder entrenado exclusivamente con muestras malignas.
- **Modelo B:** Autoencoder entrenado exclusivamente con muestras benignas.

2. **Preparación de datos:** Cada modelo utiliza el conjunto de datos correspondiente a su entrenamiento: Modelo A solo con muestras malignas y Modelo B solo con muestras benignas.

3. **Evaluación y métricas:** Cada modelo se evalúa sobre el conjunto completo, incluyendo ambos tipos de muestras, utilizando las siguientes métricas:

- **Tasa de detección de anomalías:** Capacidad del modelo para identificar correctamente las muestras del tipo opuesto al que fue entrenado.
- **Tasa de falsos positivos:** Incidencia de muestras del mismo tipo que el entrenamiento, incorrectamente clasificadas como anómalas.
- **Precisión, recall y F1-score.**

4. **Análisis comparativo:** Se comparan los resultados de ambos modelos para determinar cuál presenta una mejor capacidad de generalización y detección de muestras anómalas.

6. Resultados

6.1. Introducción

En el recorrido de este capítulo se expondrán los diferentes resultados obtenidos tras realizar los experimentos estipulados en [5.3 Marco Experimental](#), manteniendo la estructura previamente especificada en sus respectivos apartados.

El objetivo de los resultados es poder verificar si nuestro modelo se puede amoldar ante diferentes situaciones, viendo cómo podemos optimizarlo modificando diferentes parámetros, y además poder cuantificar cuan bueno es nuestro modelo ante otras opciones.

6.2. Resultados de los Experimentos

6.2.1. Variación la cantidad de muestras anómalas

Preparación de los datos

Para la configuración del entorno del **experimento 1** en el que la cantidad de anomalías respecto al conjunto total de datos es cada una de las posibilidades se utilizará la variable auxiliar previamente declarada porcentaje de anomalías.

Modificación del conjunto de datos

Cómo queremos detectar casos anómalos y en este caso la distribución de los ejemplos no cumple con esta condición deberemos modificar la cantidad de datos. Para ello escogeremos un porcentaje de casos malignos que sea anómalo.

```
porcentaje_anomalias = 0.3 # % de instancias malignas
```

Figura 6.1: Variable auxiliar para especificar la cantidad de anomalías.

La proporción de datos en función del porcentaje de muestras como malignas queda con las siguientes distribuciones:

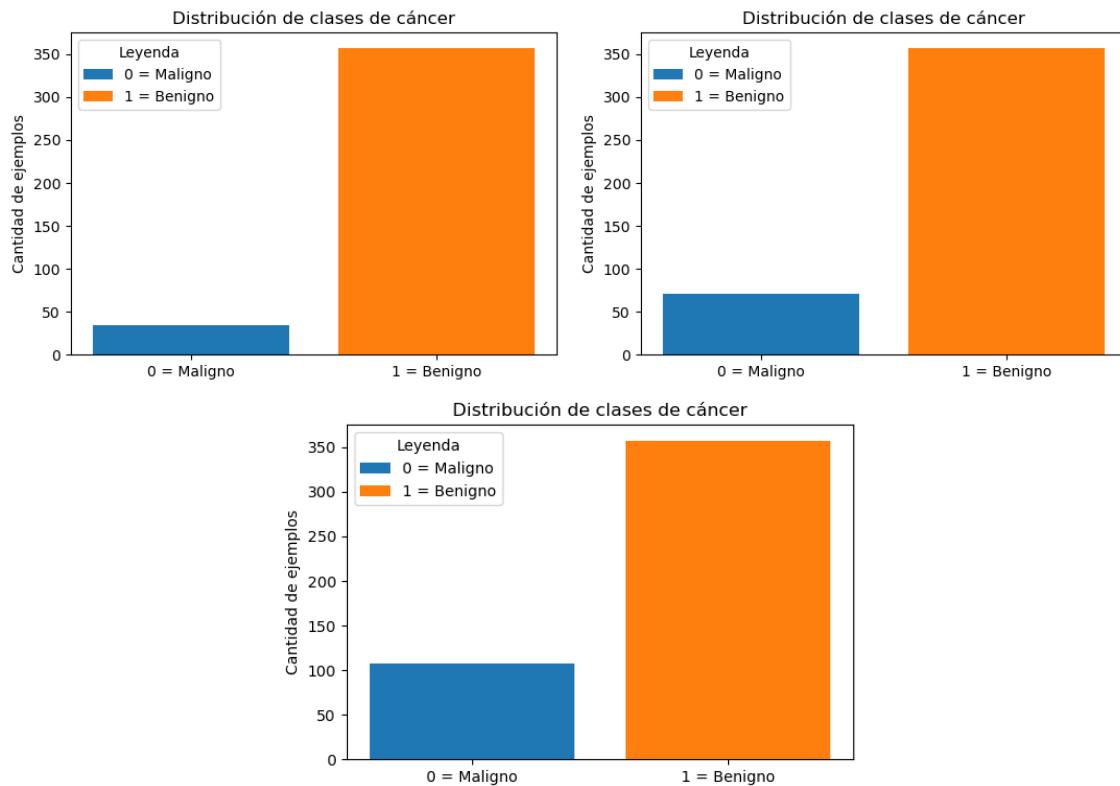


Figura 6.2: Distribuciones de los datos en función del porcentaje de anomalías.

Entrenamiento del modelo

A pesar de tener exactamente los mismos parámetros la curva de aprendizaje varía un poco en cada uno de los casos, llegando a un 0.4 de pérdida en el caso de 0.3 % de muestras de datos de clase benigna.

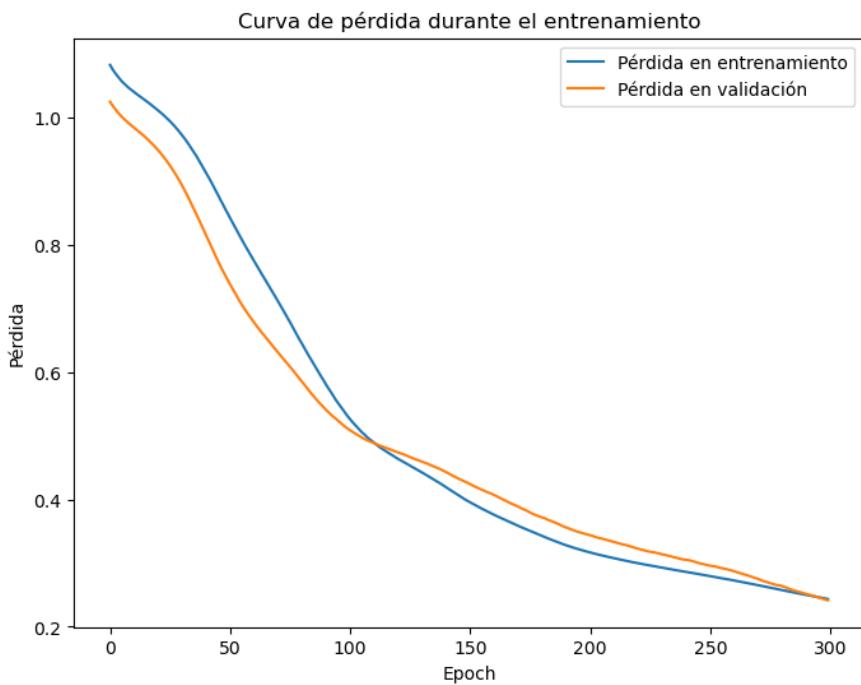


Figura 6.3: Curva de aprendizaje con 0.1 % de clases Malignas.

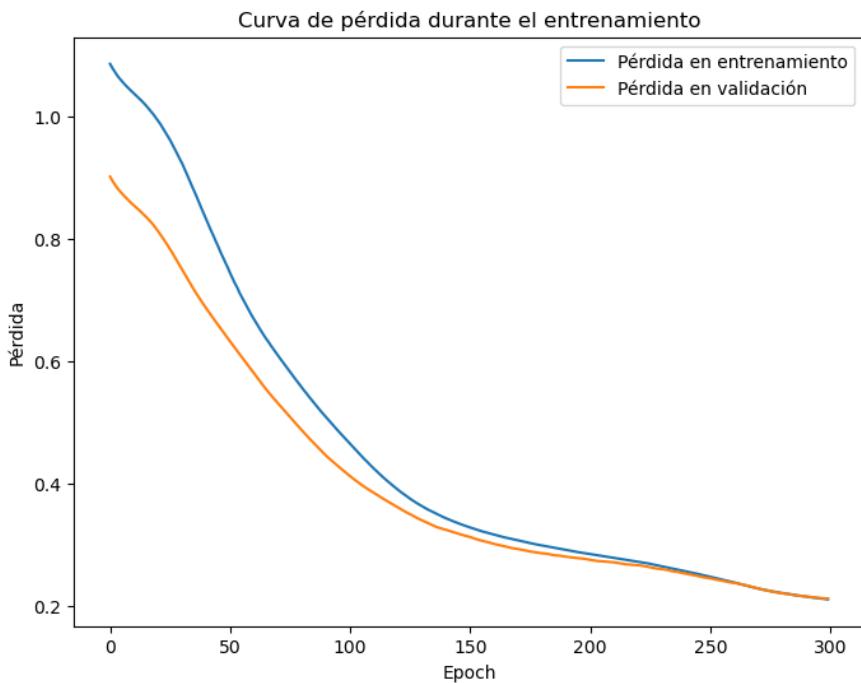


Figura 6.4: Curva de aprendizaje con 0.2 % de clases Malignas.

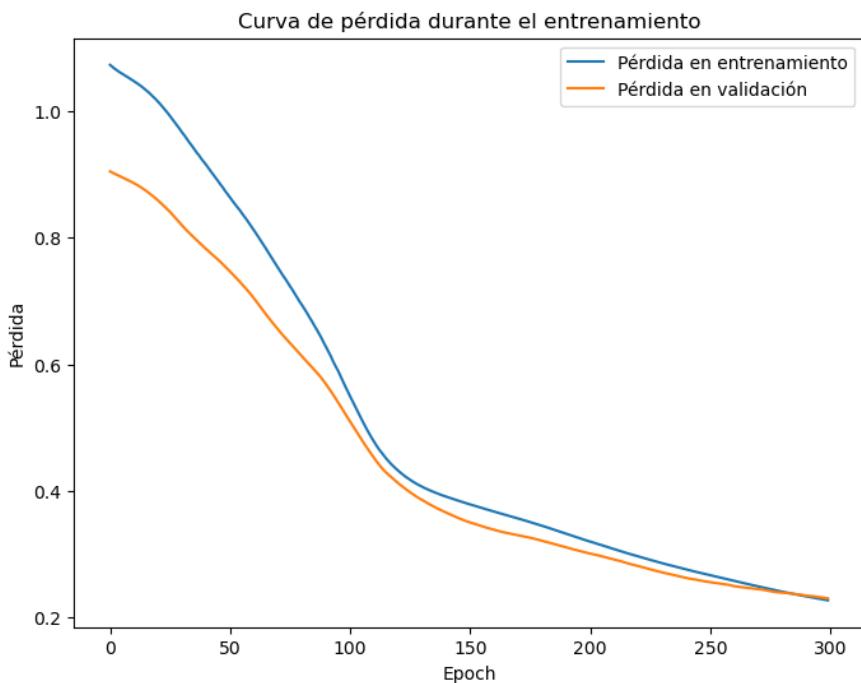


Figura 6.5: Curva de aprendizaje con 0.3 % de clases Malignas.

Como podemos ver en cualquiera de los 3 modelos tenemos un mismo tipo de aprendizaje puesto que el modelo es siempre el mismo debido a que lo entrenamos exclusivamente con muestras benignas.

Evaluación del modelo:

Cómo primera medida de evaluación utilizaremos las diferentes matrices de confusión para ver cuantos aciertos puede realizar al variar la cantidad de muestras de clase Maligna.

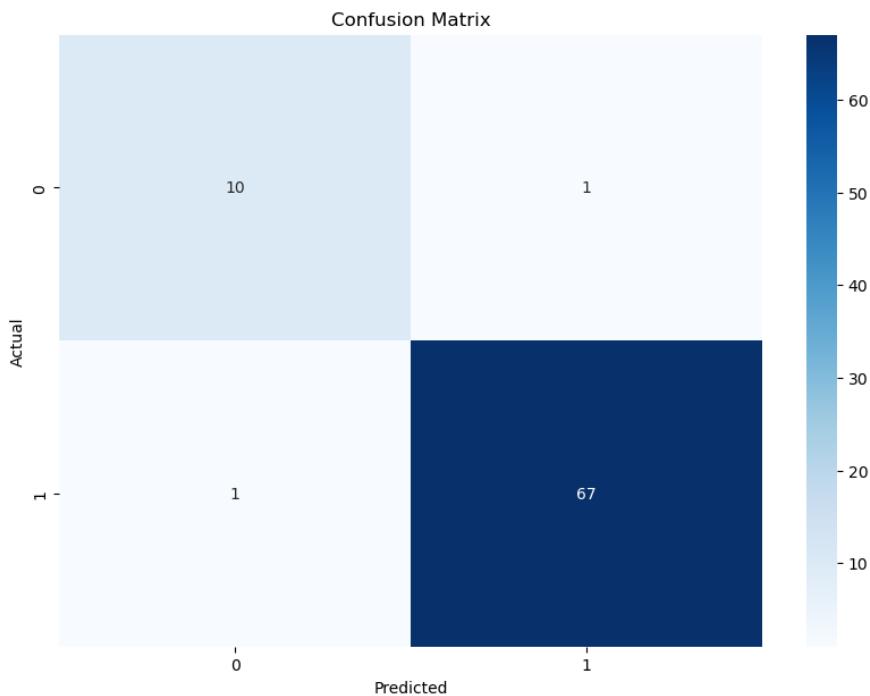


Figura 6.6: Matriz de confusión 10 % de muestras malignas.

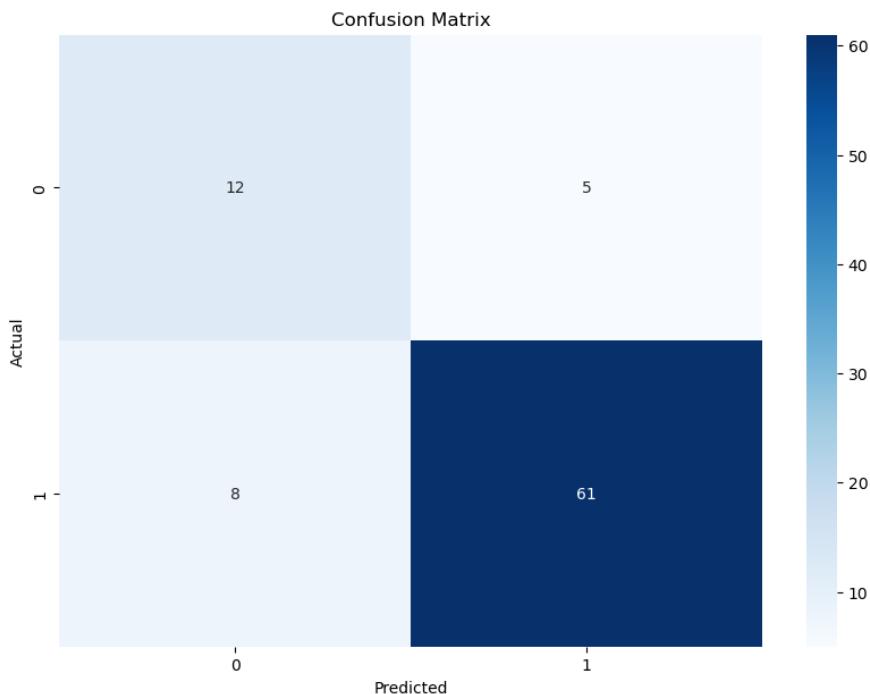


Figura 6.7: Matriz de confusión 20 % de muestras malignas.

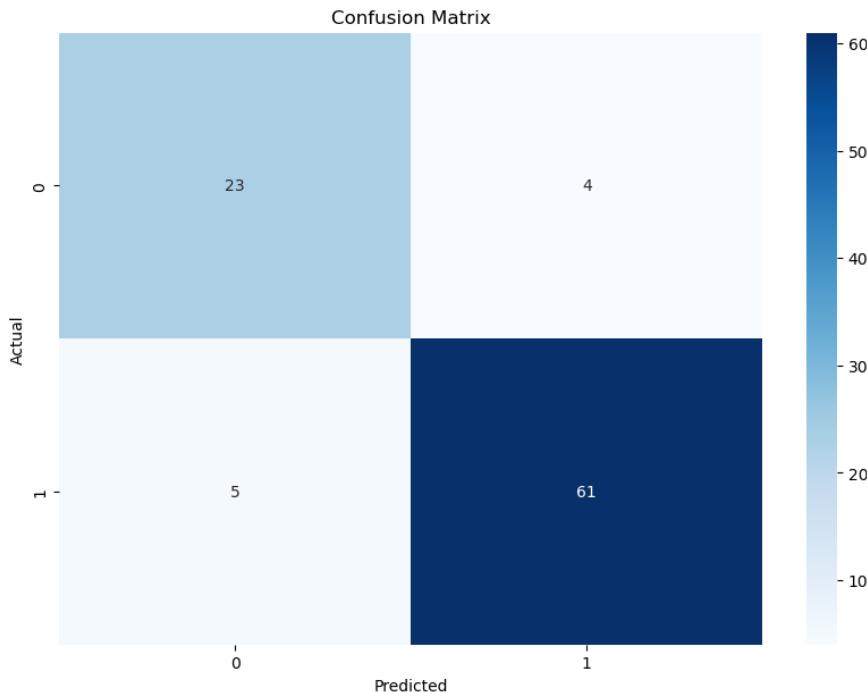


Figura 6.8: Matriz de confusión 30 % de muestras malignas.

1. **Matriz de confusión con 0.1 % de anomalías:** Consigue clasificar casi completamente bien a los casos benignos a excepción de uno mientras que con los malignos acierta un 10/11 de los casos, lo que lo convierte en una opción casi perfecta.
2. **Matriz de confusión con 0.2 % de anomalías:** Al encontrarnos con muchos más casos ahora el modelo no consigue predecir casi todos los benignos, donde ahora acierta un 61/69, lo que es una gran certeza. Mientras tanto la tasa de malignos acertados ha empeorado significativamente a 12/17.
3. **Matriz de confusión con 0.3 % de anomalías:** La matriz en este caso tiene una mayor mejoría 23/27 mientras que la tasa de benignas desciende a 61/66.

A continuación visualizaremos las diferentes métricas de Precisión y Recall a lo largo del aumento del umbral, con el objetivo de determinar que la Precision aumenta ya que el modelo es más conservador al etiquetar una instancia como positiva , al mismo tiempo baja la Recall ya que menos instancias positivas reales son capturadas por el umbral más estricto.

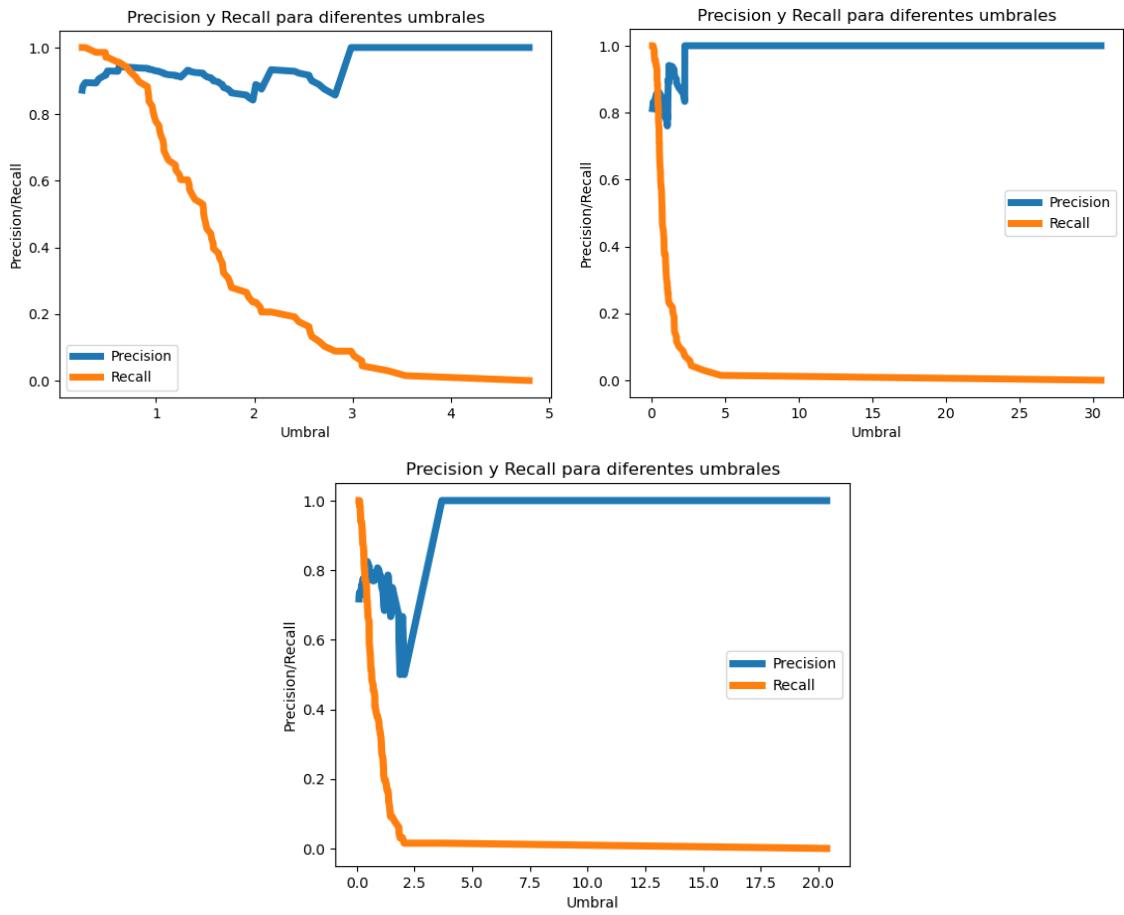


Figura 6.9: Precision y Recall según los diferentes umbrales.

Al visualizar la Precisión y Recall de los diferentes de las diferentes versiones del modelo vemos que cumplen perfectamente, donde la precisión aumenta a lo largo del aumento del umbral, mientras que el Recall disminuye hasta acercarse lo más posible a 0.

Observaremos las diferentes curvas ROC para comprobar el area AUC, y así ver cuanto mejor es nuestro modelo ante el azar.

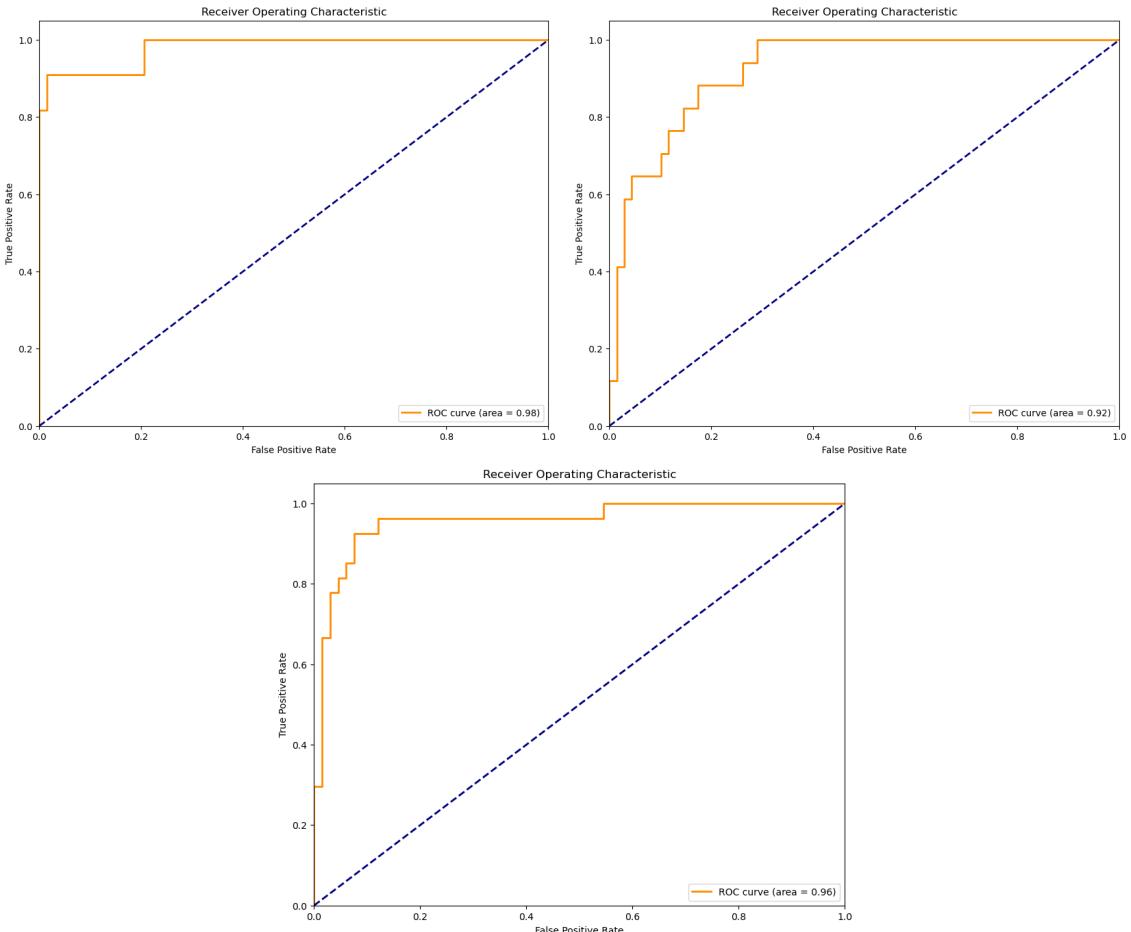


Figura 6.10: Precision y Recall según los diferentes umbrales.

Podemos contemplar cómo el AUC va aumentando progresivamente cuanto más datos tenemos, pasando de tener tan solo 0.73, 0.82 a 0.85, dejando claro que nuestro modelo es mucho mejor que el azar, donde solo habría un 0.5 de posibilidades de acertar.

Por último se mostrará los diferentes valores de Precisión, Exhaustividad, F1-score y Exactitud a lo largo de la variación de muestras Malignas.

Tabla 6.1: Desempeño del modelo para distintos porcentajes de muestras de clase maligna

% de Muestra Maligna	Precisión	Exhaustividad	F1-score	Exactitud
0.1 %	0.9710	0.9853	0.9781	0.9620
0.2 %	0.9242	0.8841	0.9037	0.8488
0.3 %	0.9394	0.9394	0.9394	0.9140

A medida que el porcentaje de muestras malignas aumenta, se observa una ligera variación en las métricas de desempeño del modelo. En general, parece que el modelo mantiene un rendimiento bastante consistente en términos de precisión, exhaustividad, F1-score y exactitud a través de los diferentes porcentajes de muestras de clase maligna.

6.2.2. Variación del umbral ante el error de reconstrucción

Configuración Inicial

Para este **experimento 2** nos basaremos en el escenario de tener un 0.3 % de datos Malignos constantemente, evaluando cual puede ser el umbral óptimo comparandolo con el resto de los umbrales escogidos para este caso.

Determinación del umbral

Tabla 6.2: Desempeño del modelo para diferentes umbrales

Umbral	Precisión	Exhaustividad	F1-score	Exactitud
0.1	1.0000	0.1970	0.3291	0.4301
0.2	0.9778	0.6667	0.7928	0.7527
0.3	0.9818	0.8182	0.8926	0.8602
0.4	0.9683	0.9242	0.9457	0.9247
0.5	0.9538	0.9394	0.9466	0.9247

Esta tabla muestra el desempeño de un modelo para diferentes umbrales fijos establecidos para el error de reconstrucción en un problema de clasificación. Como se puede observar, a medida que aumenta el umbral, las métricas de precisión y exactitud tienden a disminuir, mientras que las métricas de exhaustividad y F1-score tienden a aumentar. Esto se debe a que un umbral más alto significa que el modelo es más conservador al clasificar las instancias como positivas, lo que resulta en menos falsos positivos pero potencialmente más falsos negativos. Por lo tanto, la elección del umbral adecuado es crucial y depende de los requisitos específicos del problema, como la sensibilidad a los falsos positivos y falsos negativos.

Evaluación del modelo

El umbral de 0.3 parece ofrecer un buen equilibrio entre estas dos métricas, ya que tiene una precisión alta (0.9818) y una exhaustividad significativa (0.8182), lo que puede conducir a una mayor cantidad de True Positive y True Negative, que conlleva a unas predicciones más acertadas.

6.2.3. Comparación con modelos de Machine Learning

Selección de modelos

Tras diferentes intentos con diferentes modelos de Machine Learning a lo largo del **experimento 3** se han tenido que descartar diferentes como Regresión Logística o Máquinas de Soporte Vectorial puesto que necesitaban entrenarse con un mínimo de 2 clases al momento del entrenamiento. Los modelos de Machine Learning

escogidos que han permitido el entrenamiento con una única clase son los siguientes:

- K-Nearest Neighbors.
- Árbol de decisión.
- Random Forest.

Preparación de datos

Al momento de la preparación de los datos han habido diferentes errores, puesto que la mayoría de los modelos no pueden ajustar sus parámetros si no hay al menos 2 clases en su fase de entrenamiento.

```
from sklearn.svm import SVC
svm = SVC(gamma=0.001, C=10)
svm.fit(X_train_clase_1, y_check)
print("Tasa de aciertos de SVM: {:.2f}".format(svm.score(Xn_test, y_test)))

-----
ValueError                                 Traceback (most recent call last)
Cell In[71], line 3
  1 from sklearn.svm import SVC
  2 svm = SVC(gamma=0.001, C=10)
----> 3 svm.fit(X_train_clase_1, y_check)
  4 print("Tasa de aciertos de SVM: {:.2f}".format(svm.score(Xn_test, y_test)))

File C:\ProgramData\anaconda3\Lib\site-packages\sklearn\base.py:1151, in _fit_context.<locals>.wrapper(estimator, *args, **kwargs)
 1144     estimator._validate_params()
 1145     with config_context(
 1146         skip_parameter_validation=(
 1147             prefer_skip_nested_validation or global_skip_validation
 1148         )
 1149     ):
--> 1151     return fit_method(estimator, *args, **kwargs)

File C:\ProgramData\anaconda3\Lib\site-packages\sklearn\svm\_base.py:199, in BaseLibSVM.fit(self, X, y, sample_weight)
 198 else:
 199     X, y = self._validate_data(
 200         X,
 201         y,
 202         accept_large_sparse=False,
 203         )
--> 204 y = self._validate_targets(y)
 205 sample_weight = np.asarray(
 206     [] if sample_weight is None else sample_weight, dtype=np.float64
 207 )
 208 solver_type = LIBSVM_IMPL.index(self._impl)

File C:\ProgramData\anaconda3\Lib\site-packages\sklearn\svm\_base.py:747, in BaseSVC._validate_targets(self, y)
 745 self.class_weight_ = compute_class_weight(self.class_weight, classes=cls, y=y_)
 746 if len(cls) < 2:
--> 747     raise ValueError(
 748         "The number of classes has to be greater than one; got %d class"
 749         % len(cls)
 750     )
 751 self.classes_ = cls
 752 return np.asarray(y, dtype=np.float64, order="C")

ValueError: The number of classes has to be greater than one; got 1 class
```

Figura 6.11: Enter Caption

SVC falla puesto que la idea de su modelo de resolución es asignar una frontera de decisión dentro de un conjunto de datos para poder dividirlo en las 2 clases posibles, pero al utilizar un conjunto de datos de una sola misma clase el modelo de aprendizaje falla.

Evaluación y métricas

A continuación se muestra una tabla donde se han realizado diferentes pruebas de medición a los diferentes modelos

Tabla 6.3: Comparación del desempeño de los modelos de detección de anomalías

Modelo	Precisión	Exhaustividad	F1-score	Exactitud
KNN	0.74	0.86	0.80	0.86
Random Forest	0.74	0.86	0.80	0.86
Árbol de decisión	0.74	0.86	0.80	0.86
Autoencoder	0.971	0.985	0.978	0.962

Análisis estadístico

La evaluación se centra en cuatro métricas principales: precisión, exhaustividad, F1-score y exactitud.

- **KNN, Random Forest, y Árbol de Decisión** muestran idénticos resultados en todas las métricas evaluadas. Específicamente, los tres modelos exhiben una precisión del 0.74, una exhaustividad del 0.86, un F1-score del 0.80 y una exactitud del 0.86. Estas cifras indican que, aunque los modelos son bastante buenos para clasificar correctamente la clase mayoritaria (benigna), fallan completamente en detectar la clase minoritaria (maligna), como se evidencia por la falta de reconocimiento de cualquier **muestra** de la clase 0.
- Autoencoder muestra un desempeño superior en todas las métricas en comparación con los otros modelos. Con una precisión de 0.971, una exhaustividad de 0.985, un F1-score de 0.978 y una exactitud de 0.962, el Autoencoder no solo clasifica correctamente casi todas las muestras benignas, sino que también es excepcionalmente eficaz en identificar las muestras malignas, lo cual es crucial en aplicaciones médicas donde la detección temprana es vital.

La superioridad del Autoencoder en este contexto sugiere que puede ser una herramienta más efectiva para la detección de anomalías, especialmente en conjuntos de datos con un desequilibrio significativo de clases, como es común en muchas aplicaciones médicas. Si analizamos las diferentes matrices de confusión de los modelos de Machine Learning podemos ver cómo no han podido averiguar cómo calificar ninguna de las muestras de clase Maligna y solo se han encargado de calificar como Benigna a todas las muestras posibles. Sin embargo, el modelo que ha utilizado el autoencoder ha conseguido calificar correctamente la mayoría de modelos, puesto que ha comprendido un modelo correcto para poder clasificar cada uno de las muestras.

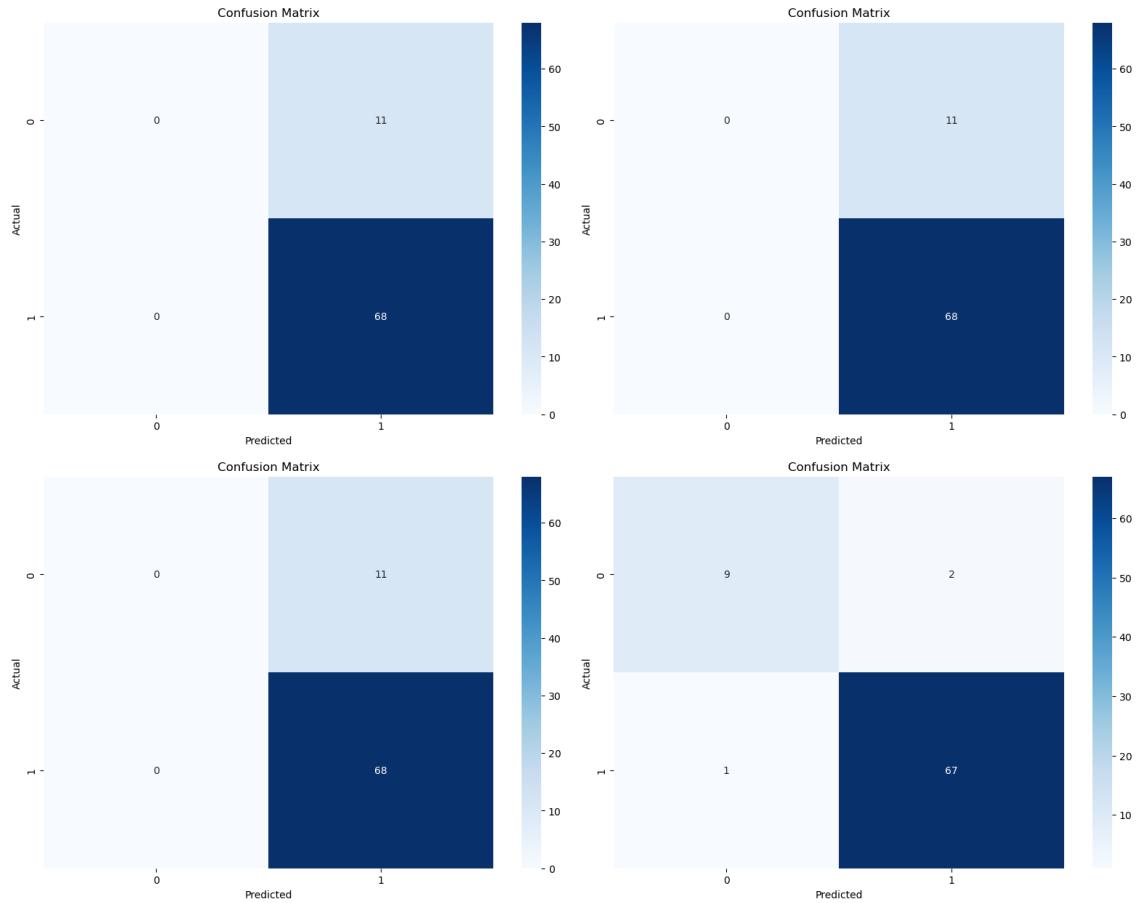


Figura 6.12: Matrices de confusión de los diferentes modelos.

6.2.4. Comparación con Entrenamiento exclusivo con muestras de clase Maligna

En el **experimento 4** se ha procedido a comparar 2 modelos diferentes de Autoencoder para la detección de muestras anómalas, para las cuales se ha contemplado en un mismo escenario donde los valores eran excesivamente anomalos debido al 10 % de cantidades únicamente Malignas.

Configuración de los modelos y Preparación de los datos

Tras configurar el Modelo A exclusivamente con muestras clasificadas como Malignas y el Modelo B exclusivamente con muestras clasificadas como Benignas procedemos diferenciar la curva de aprendizaje de los modelos.

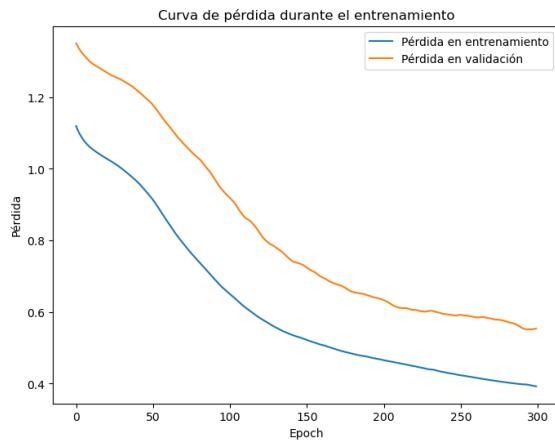


Figura 6.13: Curva de aprendizaje del modelo A.

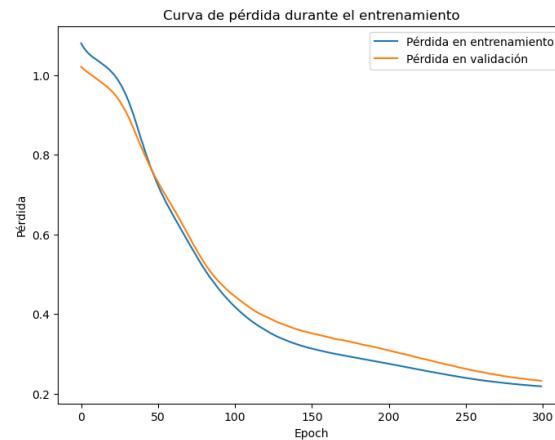


Figura 6.14: Curva de aprendizaje del modelo B.

Modelo A

- La curva de pérdida de entrenamiento comienza en un valor alto y disminuye de forma constante, lo que indica que el modelo está aprendiendo efectivamente de los datos de entrenamiento a lo largo del tiempo.
- La curva de pérdida de validación también desciende, pero tiende a aplanarse a medida que se acerca a 300 épocas, sugiriendo que el modelo podría estar empezando a estabilizarse y no mejora significativamente con más entrenamiento.
- La brecha entre las curvas de entrenamiento y validación es moderada, lo que podría indicar un ajuste razonable sin un sobreajuste severo, aunque existe cierta diferencia que podría sugerir un ligero sobreajuste.

Modelo B

- Tanto la curva de pérdida de entrenamiento como la de validación disminuyen rápidamente al inicio, lo cual es indicativo de un aprendizaje eficiente y rápido al principio de las épocas.
- Ambas curvas convergen a medida que aumentan las épocas, con la curva de validación acercándose mucho a la de entrenamiento hacia el final del proceso de entrenamiento.
- La convergencia de las curvas sugiere que el Modelo B tiene un excelente equilibrio entre aprendizaje y generalización, con un riesgo muy bajo de sobreajuste.

Comparación entre el Modelo A y el Modelo B:

- **Eficiencia del aprendizaje:** El Modelo B parece aprender más eficientemente, ya que sus curvas de pérdida tanto de entrenamiento como de validación convergen más rápidamente y a un nivel más bajo de pérdida comparado con el Modelo A.

- **Riesgo de sobreajuste:** El Modelo B muestra un menor riesgo de sobreajuste comparado con el Modelo A, como se evidencia por la menor brecha entre las curvas de entrenamiento y validación.
- **Estabilidad:** El Modelo B parece ser más estable al final de las épocas dadas, ya que las curvas de pérdida de entrenamiento y validación son casi indistinguibles, lo que sugiere que el modelo generaliza bien a nuevos datos. En contraste, el Modelo A muestra una brecha continua, aunque pequeña, entre entrenamiento y validación.

Estas observaciones sugieren que, aunque ambos modelos están aprendiendo, el Modelo A probablemente necesita diferentes modificaciones a sus parámetros, o bien introducir más datos de ejemplos con el objetivo optimizar la curva tanto en validación como en entrenamiento.

Evaluación y métricas

Al comparar las matrices de confusión de los modelos A y B, observamos que el modelo B supera al modelo A en términos de precisión para clasificar correctamente las muestras tanto de la clase Maligna como de la clase Benigna.

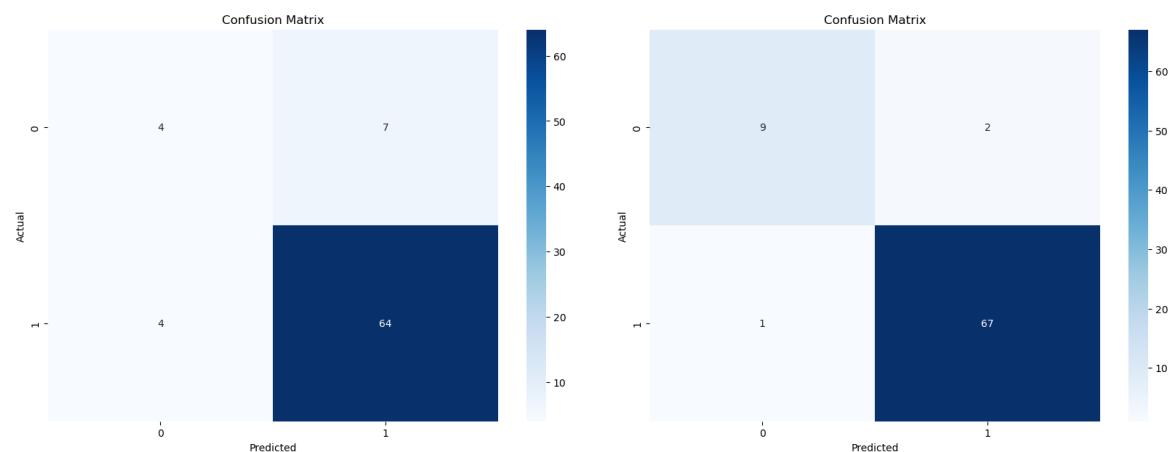


Figura 6.15: Matriz de confusión del modelo A.

Figura 6.16: Matriz de confusión del modelo B.

Específicamente, para la clase Maligna, el modelo B logra un porcentaje de acierto del $\frac{9}{11}$ o aproximadamente 81.82 %, que es considerablemente superior al $\frac{4}{11}$ o aproximadamente 36.36 % de aciertos del modelo A. Esto indica que el modelo B es más eficaz en identificar correctamente los casos malignos en comparación con el modelo A.

En cuanto a la clase Benigna, el modelo B también presenta un mejor desempeño que el modelo A, aunque la diferencia es menos marcada. El modelo B alcanza $\frac{67}{68}$ o aproximadamente 98.53 % de precisión, comparado con $\frac{64}{68}$ o aproximadamente 94.12 % del modelo A. Esto muestra que el modelo B tiene una ligera ventaja también en la identificación correcta de casos benignos.

En resumen, el modelo B demuestra ser superior al modelo A al tener tasas de acierto más altas en la clasificación correcta de muestras tanto malignas como benignas, lo cual es probablemente motivo de la insuficiencia de muestras Malignas en el conjunto de entrenamiento, lo cuál no le ha permitido obtener unos pesos suficientemente ajustados.

En las figuras presentadas, observamos las curvas ROC para dos modelos de clasificación, Modelo A y Modelo B.

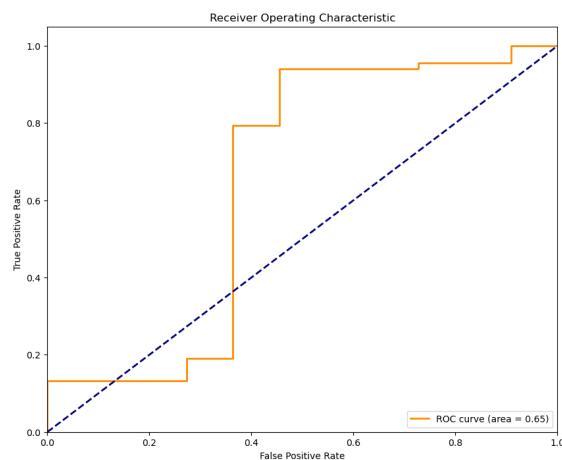


Figura 6.17: Curva ROC del modelo A.

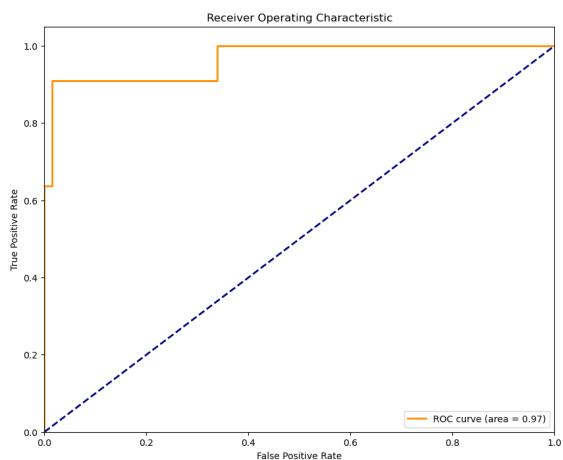


Figura 6.18: Curva ROC del modelo B.

■ Curva ROC del Modelo A:

- La curva muestra un comportamiento escalonado, incrementando la Tasa de Verdaderos Positivos (TPR) sin aumentar la Tasa de Falsos Positivos (FPR) en determinados tramos.
- Se aproxima a la línea diagonal azul, indicando una capacidad discriminativa moderada.
- El área bajo la curva (AUC) es de 0.63, sugiriendo un rendimiento aceptable pero no sobresaliente.

■ Curva ROC del Modelo B:

- La curva asciende rápidamente hacia el borde superior izquierdo, indicando un mejor rendimiento en la clasificación.
- Se mantiene por encima y más cerca del borde izquierdo del gráfico, lo cual es indicativo de una mejor eficiencia en identificar casos positivos con bajos falsos positivos.
- El AUC para el Modelo B es de 0.97, demostrando una excelente capacidad discriminativa del modelo.

El Modelo B supera considerablemente al Modelo A en términos de capacidad de discriminación entre clases, como lo demuestra su AUC más alto (0.97 frente a 0.63). La curva del Modelo B alcanza mayores tasas de verdaderos positivos para menores tasas de falsos positivos más rápidamente que la del Modelo A, indicando

una mayor eficiencia y efectividad en la clasificación de casos positivos sin incrementar los errores de clasificación de casos negativos como positivos.

Mientras que el Modelo B es un modelo que aporta confianza debido a su alta probabilidad de acertar, el modelo A no es más que un 10 % preferible ante el azar.

Finalmente vamos a realizar una comparación entre las estadísticas obtenidas con cada uno de los 2 modelos.

Tabla 6.4: Comparación del desempeño de los modelos de clasificación

Modelo	Precisión	Exhaustividad	F1-score	Exactitud
Modelo A	0.7407	0.2941	0.4211	0.3038
Modelo B	0.9014	0.9412	0.9209	0.8608

La información de esta tabla no hace más que confirmar los datos que hemos visto anteriormente, contemplando que el Modelo B es superior ante el modelo A al momento realizar predicciones con los datos.

Cómo hemos visto a lo largo de este experimento el Modelo A utilizando las muestras clasificadas como Malignas como conjunto exclusivo de entrenamiento es viable a la hora de crear un modelo de aprendizaje, pero muy inferior al que podemos tener al usar el Modelo B, probablemente debido a la inmensa cantidad de muestras que hemos tenido a la hora de realizar el entrenamiento en comparación de utilizando el modelo A.

7. Discussion

7.1. Análisis de los Resultados

7.1.1. Desempeño del Autoencoder

Los resultados obtenidos sugieren que el autoencoder ha demostrado ser superior en la detección de muestras anómalas en comparación con otros modelos evaluados. La principal fortaleza del autoencoder radica en su habilidad para trabajar con un solo tipo de clase durante el entrenamiento, lo cual es especialmente útil en escenarios donde las muestras positivas (como las malignas en contextos médicos) son escasas.

El autoencoder mostró una notable robustez al ajustar su rendimiento ante variaciones significativas en el porcentaje de muestras malignas, lo que demuestra su capacidad para adaptarse a diferentes niveles de desbalance de clase. Esta característica es crucial en aplicaciones médicas, donde la proporción de casos positivos a menudo es baja.

7.1.2. Comparación con Otros Modelos de Machine Learning

A pesar de que modelos como K-Nearest Neighbors, Árbol de Decisión y Random Forest son ampliamente utilizados en clasificación, su rendimiento fue notablemente inferior en este contexto específico. Estos modelos requieren de una distribución de clase más equilibrada para funcionar adecuadamente, y su incapacidad para entrenar con una sola clase limita seriamente su aplicabilidad en detección de anomalías.

7.1.3. Impacto de la Selección del Umbral en la Detección de Anomalías

La elección del umbral de error de reconstrucción mostró tener un impacto significativo en el desempeño del modelo. Un umbral muy bajo generaba un gran número de falsos positivos, mientras que un umbral alto podía omitir casos genuinamente anómalos. Encontrar el balance correcto es clave para maximizar la efectividad del modelo en la práctica clínica.

7.1.4. Variaciones en el modelo de aprendizaje

Originalmente se pensó en la idea de seguir el Modelo A del experimento 4 puesto que la idea de utilizar una clase de elementos cuyo patrón está delimitado en un rango cerrado de números parecía más prometedor que encontrar un patrón a negar dentro de una variación tan alta como tienen los atributos del modelo B. Ante los resultados se puede apreciar como la alta diferencia de elementos contribuye de mejor manera al aprendizaje del modelo que el modelo previamente planeado.

7.2. Limitaciones del Estudio

Aunque los resultados son prometedores, existen varias limitaciones que deben ser abordadas:

- **Diversidad de Datos:** Los modelos fueron evaluados en un conjunto de datos relativamente homogéneo y controlado. La variabilidad real en muestras médicas puede ser mayor, lo que podría afectar el rendimiento del modelo en escenarios reales.
- **Generalización:** Los modelos necesitan ser validados en conjuntos de datos más amplios y diversificados para verificar su capacidad de generalización en diferentes poblaciones y condiciones médicas.

7.3. Futuras Direcciones de Investigación

Para superar las limitaciones identificadas y mejorar la utilidad de los modelos en entornos clínicos, se recomiendan varias direcciones futuras:

- **Incorporación de Técnicas de Aprendizaje Semi-Supervisado:** Esto podría mejorar la eficiencia del modelo al utilizar un número mayor de muestras no etiquetadas.
- **Expansión de la Base de Datos:** Integrar más variedades de datos, incluyendo diferentes tipos de cáncer y subtipos, podría ayudar a mejorar la robustez y generalidad del modelo.
- **Mejora de la Interpretabilidad del Modelo:** Investigar métodos para hacer que los autoencoders sean más transparentes podría facilitar su adopción en la práctica médica, permitiendo a los profesionales entender y confiar más en las decisiones automáticas.
- **Estudios de Validación Clínica:** Realizar pruebas clínicas para validar la efectividad de los modelos en condiciones reales sería un paso crucial para su implementación.

8. Demo - Herramienta clínica para la detección de cáncer de mama

8.1. Introducción

En última instancia se ha decidido añadir al TFG una demo técnica recreando un servicio como página web dedicada al sector sanitario donde cualquier especialista clínico tenga la posibilidad de utilizar el modelo de predicción del modelo de nuestro Autoencoder para poder predecir si las muestras de unas analíticas de tumores en cancer de mama son detectadas como tumores Malignos o Benignos.

Para la realización de dicho servicio se desplegará un Servidor Ubuntu en Oracle, donde se instalarán diferentes servicios como *Flask* para poder alojar la página web, tensorflow para poder utilizar el modelo del Autoencoder, y otros servicios complementarios para el cifrado SSL entre otros.

8.2. Elección y Despliegue del Servidor



Figura 8.1: Servicio Cloud Oracle

Fuente: <https://www.cloverinfotech.com/blog/why-oracle-cloud-infrastructure-oci-is-the-key-growth-driver-for-your-business/>

Se ha escogido la Infraestructura as a Service (IaaS) que nos proporciona Oracle Cloud en conveniencia mia, Marco Tenorio Cortés, que ya poseia una cuenta en dicho servicio, el cuál nos proporciona la infraestructura necesaria para poder desplegar un servidor con la suficiente capacidad y sin limite temporal, ofreciéndonos la capacidad de gestionarlo y realizarle un mantenimiento adecuado para ello a posteriori.

El servidor desplegado consta de las siguientes especificaciones técnicas:

Tabla 8.1: Especificaciones técnicas del servidor en Oracle Cloud

Característica	Especificación
Sistema Operativo	Ubuntu Server
Memoria RAM (GB)	18
Procesador	ARM con 3 núcleos de 2 GHz
Almacenamiento (GB)	200

Dichas especificaciones nos proveen de capacidad más que suficiente para realizar la breve Demo Técnica que nos proponemos. Entre las especificaciones nos encontramos que el Servidor tendrá de un procesador con arquitectura ARM, la cuál es realmente eficiente y cada día tiene más soporte. Una de las principales ventajas de los procesadores ARM es su increíble eficiencia energética. Utilizan una arquitectura RISC (Reduced Instruction Set Computing), que es capaz de realizar operaciones con menos ciclos de reloj y menor consumo de energía.

El usar una arquitectura ARM aunque tiene sus ventajas por su eficiencia es cierto que da otras desventajas como puede ser la compatibilidad a la hora de instalar diferentes servicios.

La aplicación se desplegará en un sistema operativo Ubuntu Server, el cual se accederá por SSH en remoto y no dispone de interfaz gráfica.

8.3. Instalación de Flask y Tensorflow

Una vez desplegado el servidor Ubuntu en Oracle Cloud, el próximo paso es instalar y configurar el entorno necesario para alojar nuestra aplicación web. El core de nuestra demo técnica involucra dos componentes principales: Flask, para manejar la interacción web, y Tensorflow, para ejecutar y procesar nuestro modelo de autoencoder.

Lo primero a realizar en nuestro servidor es instalar Python que nos dará la posibilidad de poder instalarnos Flask y Tensorflow.

```
1 sudo apt update && sudo apt upgrade -y  
2 sudo apt install python3-pip python3-dev -y
```

Extracto de código 8.1: Instalación de Python

Flask, un micro-framework para Python, fue elegido por su simplicidad y eficiencia para desarrollar aplicaciones web:

```
1 pip3 install Flask
```

Extracto de código 8.2: Instalación de Flask

Además se instala Tensorflow para poder realizar la carga del modelo del Autoencoder, así como realizar predicciones con el mismo.

```
1 pip3 install Tensorflow
```

Extracto de código 8.3: Instalación de Tensorflow

Se creó un la siguiente estructura para la configuración de la aplicación.

```
/Aplicación sanitaria
├── app.py
├── /model
├── /normalizador
├── /templates
│   ├── index.html
│   ├── resultado.html
│   └── resultados.html
├── /uploads
└── /static
    ├── style.css
    └── style2.css
```

8.4. Configuración de Flask

Dentro del fichero app.py añadimos la configuración del servicio, donde importamos el Autoencode y fichero de normalización que guardamos previamente cuando configuramos estos datos en el Notebook, los cuales nos servira para realizar las predicciones y normalizar los datos de entrada a predecir. Además como el usuario tendra la posibilidad de subir un fichero Excel con los datos a predecir habrá que configurar una carpeta de subida.

```
1 get_custom_objects().update({'mse': MeanSquaredError()})
2 app = Flask(__name__)
3
4 # Configura un directorio para guardar archivos subidos
5 app.config['UPLOAD_FOLDER'] = 'uploads/'
6
7 # Cargar el modelo y el normalizador
8 autoencoder = load_model('model/autoencoder.h5')
9 normalizador = load('normalizador/normalizador.pkl')
```

Extracto de código 8.4: Configuración fichero Flask

Dicho excel podrá contener todos los atributos del conjunto de datos, por lo que tendremos que eliminar aquellos atributos que hayamos eliminado en el modelo de nuestro autoencoder.

```
1 if 'MEAN FRACTAL DIMENSION' in data.columns:
2     data = data.drop(columns=['MEAN FRACTAL DIMENSION'])
```

Extracto de código 8.5: Eliminación de datos redundantes para el modelo

Además se crearán páginas estáticas HTML así como sus ficheros de estilo CSS en sus respectivas carpetas de la estructura. Si estás más interesado en conocer el código puedes visualizarlo en el Apéndice 10.1.

8.5. Configuración del dominio

Para que la recreación del escenario sea lo más fiel posible se le asignará un dominio a nuestra página web para que se pueda acceder desde forma externa mediante un nombre asociado a la IP del servidor.

Para ello se utilizará un dominio ya adquirido previamente, **mtcor.es**, y se le asignará el subdominio **sanidad.mtc当地.es**.



Figura 8.2: Proveedor de dominios OVH

Fuente: https://es.m.wikipedia.org/wiki/Archivo:Logo_OVH.svg

8.5.1. Alojamiento del dominio

Dicho dominio se aloja en el proveedor de dominios OVH, donde configuramos el espacio de nombres de dominio (Zona DNS) para añadir un nuevo registro de tipo A, apuntando hacia nuestro servidor.

A screenshot of a web interface titled "Editar un registro de la zona DNS" (Step 1 of 2). It shows fields for Subdominio (sanidad), TTL (Por defecto), Destino (144.24.196.205), and a preview box showing "sanidad IN A 144.24.196.205". Buttons for Cancelar and Siguiente are at the bottom.

Figura 8.3: Registro del subdominio en Zona DNS

8.5.2. Configuración de certificado SSL para el dominio

Los certificados SSL protegen la integridad y la privacidad de la información que tus usuarios intercambian con tu sitio web al cifrar los datos transmitidos entre el navegador del usuario y tu servidor web. Esto impide que los hackers puedan leer o modificar la información durante la transmisión, incluyendo datos personales, contraseñas, y detalles de tarjetas de crédito.

Añadiremos un certificado SSL al dominio utilizando certbot con el plugin que incorpora para dominios de OVH.

Primero instalaremos certbot y su plugin para OVH

```
1 sudo apt install certbot  
2 sudo apt install python3-certbot-dns-ovh
```

Extracto de código 8.6: Instalación de certbot

Y ahora mediante un fichero ovh.ini donde tendremos una API Key de OVH configuraremos el subdominio

```
1 sudo certbot certonly --dns-ovh --dns-ovh-credentials /etc/ovh/ovh.ini -d sanidad.mtcor.es -d sanidad.mtcor.es
```

Extracto de código 8.7: Configuración de SSL para el dominio

Tras la configuración se nos creará la clave privada y la clave pública para para nuestro subdominio la cuál incorporaremos en el futuro fichero de configuración NGinx.

8.5.3. Configuración NGinx

En nuestro servidor deberemos configurar NGinx como Proxy Inverso para que cuando nos envíen una solicitud con el subdominio `sanidad.mtcor.es` lo redirijamos al puerto 8086, donde tenemos el servicio Flask funcionando.

Primero instalaremos NGinx

```
1 apt install NGinx
```

Extracto de código 8.8: Instalación de NGinx

Una vez instalado configuraremos su fichero de zonas disponibles para la redirección. Para ello nos dirigimos a la carpeta de configuración de zonas `/etc/nginx/sites-available` y configuraremos la zona `sanidad.mtcor.es`

```
1 server {  
2     listen 80;  
3     server_name sanidad.mtcor.es;  
4     return 301 https://$server_name$request_uri;  
5 }
```

```

6
7 server {
8     listen 443 ssl;
9     server_name sanidad.mtc.cor.es;
10
11    # SSL parameters
12    ssl_certificate /etc/letsencrypt/live/mtcor.es/fullchain.pem;
13    ssl_certificate_key /etc/letsencrypt/live/mtcor.es/privkey.pem;
14    ssl_session_timeout 30m;
15    ssl_protocols TLSv1.2;
16    ssl_ciphers ECDHE-ECDSA-AES128-GCM-SHA256:ECDHE-RSA-AES128-GCM-SHA256:ECDHE-ECDSA-AES256-GCM-SHA384:ECDHE-RSA-AES256-GCM-SHA384:ECDHE-ECDSA-CHACHA20-POLY1305:ECDHE-RSA-CHACHA20-POLY1305:DHE-RSA-AES128-GCM-S>    ssl_prefer_server_ciphers off;
17
18    # CodeServer configuration
19    location / {
20        proxy_pass http://127.0.0.1:8086;
21        proxy_set_header Host $host;
22        proxy_set_header X-Real-IP $remote_addr;
23        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
24
25        # WebSocket proxying
26        proxy_http_version 1.1;
27        proxy_set_header Upgrade $http_upgrade;
28        proxy_set_header Connection "upgrade";
29    }
30}

```

Extracto de código 8.9: Redirección al servicio Flask

8.6. Visualización de la página

Si accedemos desde cualquier navegador a <https://sanidad.mtc.cor.es/> podremos ver un formulario para introducir los datos de una muestra a identificar.

Introduzca los atributos para la predicción de cáncer de mama

[Formulario](#)
[Subir Excel](#)

Radio medio:

Textura media:

Perímetro medio:

Área media:

Suavidad media:

Compacidad media:

Concavidad media:

Puntos de concavidad medios:

Simetria media:

Radio SE:

Textura SE:

Perímetro SE:

Figura 8.4: Página principal con formulario para predicción de una muestra

Si rellenamos el formulario con una muestra, que en este caso se ha escogido la primera del dataset Cancer Breast Wisconsin, podremos mediante el botón *Enviar* redirigirnos a una página donde nos sale el resultado de la predicción tras haber utilizado el modelo de nuestro autoencoder.



Figura 8.5: Página de resultado de la predicción

Podemos ver que nuestro modelo esta funcionando correctamente puesto que es efectivamente maligno.

Desde la página principal podemos acceder *Subir Excel* donde se nos ofrece la posibilidad de enviar un fichero excel para que prediga las muestras que están contenidas en el fichero.

Introduzca los atributos para la predicción de cáncer de mama

Formulario Subir Excel

Seleccionar archivo Ningún archivo seleccionado

Enviar

Figura 8.6: Página web para clasificar muestras contenidas en un fichero Excel.

Si introducimos dicho fichero y utilizamos el botón *Enviar* se nos redirigirá a una página muy similar a la del resultado de la muestra, con la diferencia de que en esta ocasión aparece una clasificación por cada muestra contenida en el fichero Excel.

Resultados de la Predicción

Resultado 1: Maligno
 Resultado 2: Benigno
 Resultado 3: Benigno
 Resultado 4: Maligno
 Resultado 5: Maligno

Volver a la página principal

Figura 8.7: Resultados de la predicción del fichero excel.

8.7. Conclusiones

Se ha podido demostrar como los modelos que desarrollamos para la predicción de datos se pueden utilizar en un entorno controlado y especializado para su uso, con el objetivo de poder contribuir a la ayuda sanitaria con el objetivo

de la detección de Cancer de mama. Esta página es solamente una demo técnica con las mínimas características requisitas para poder desempeñar la función necesaria para la detección del Cancer de Mama, y sirve exclusivamente como demostración visual, y como aliciente para el resto de personas a la contribución a la investigación de diferentes técnicas para la mejora sanitaria.

9. Conclusiones

Este capítulo presenta las conclusiones derivadas del estudio y desarrollo del Trabajo de Fin de Grado, titulado "Autoencoders y Error de Reconstrucción para Detectar Muestras de Cáncer". A continuación se discuten los logros obtenidos en relación con los objetivos planteados, las limitaciones encontradas durante la investigación y las posibles direcciones futuras que podría tomar este trabajo.

9.1. Resumen de Objetivos Alcanzados

Este proyecto se propuso explorar la viabilidad y eficacia de los autoencoders en la detección de muestras anómalas de cáncer, utilizando el conjunto de datos Breast Cancer Wisconsin. Los objetivos específicos incluyeron entender profundamente la teoría y aplicación de los autoencoders, desarrollar competencias técnicas en la manipulación de datos médicos, y evaluar el desempeño de estos modelos contra métodos de clasificación estándares.

9.2. Principales Hallazgos y Resultados

Los experimentos realizados demostraron que los autoencoders, particularmente cuando están bien ajustados y configurados, pueden identificar eficazmente muestras anómalas, con un rendimiento comparable o incluso superior en algunos casos a técnicas de aprendizaje automático más tradicionales. La clave de su éxito reside en su capacidad para aprender representaciones densas y significativas de los datos, lo que permite una distinción clara entre muestras normales y patológicas.

9.3. Análisis y Discusión de Resultados

El análisis de los resultados indica que el error de reconstrucción proporciona una métrica robusta para la detección de anomalías. Comparativamente, este enfoque ofrece ventajas en términos de flexibilidad y adaptabilidad sobre métodos supervisados, especialmente en escenarios donde las etiquetas pueden ser incompletas o imprecisas.

9.4. Limitaciones y Desafíos

Una limitación significativa encontrada fue la dependencia del rendimiento del modelo en la calidad y cantidad de datos disponibles. Las discrepancias en la recolección y etiquetado de datos pueden afectar adversamente la capacidad del modelo para generalizar. Además, la selección de hiperparámetros sigue siendo un desafío considerable que requiere una exploración extensiva y puede ser computacionalmente costosa.

9.5. Implicaciones Prácticas

Los hallazgos de este estudio sugieren que los autoencoders tienen un potencial considerable para su incorporación en sistemas de diagnóstico clínico, mejorando la precisión y la eficiencia de los exámenes médicos. La automatización de la detección preliminar de muestras anómalas podría reducir significativamente las cargas de trabajo de los especialistas y permitir intervenciones más tempranas y precisas.

9.6. Recomendaciones para Trabajos Futuros

Futuras investigaciones podrían enfocarse en mejorar los algoritmos de ajuste de hiperparámetros para autoencoders, explorar conjuntos de datos más grandes y diversificados, y desarrollar modelos híbridos que combinen autoencoders con otras técnicas de aprendizaje profundo para mejorar la robustez y la precisión.

9.7. Reflexión Personal

A través de la realización de este TFG, he adquirido no sólo un conocimiento técnico más profundo sobre el aprendizaje automático y su aplicación en problemas reales, sino también habilidades valiosas en la gestión de proyectos y la resolución de problemas complejos. Este proyecto ha reforzado mi interés en continuar explorando y contribuyendo al campo de la inteligencia artificial en aplicaciones médicas.

10. Apéndice

10.1. Código del fichero Main Flask

A continuación se muestra el fichero de configuración python principal del servicio Flask para la instancia del proceso que ejecuta la herramienta clínica con el Autoencoder.

```
1 from flask import Flask, request, render_template, redirect, url_for
2 import numpy as np
3 import pandas as pd
4 from tensorflow.keras.models import load_model
5 from joblib import load
6 from werkzeug.utils import secure_filename
7 import os
8 from tensorflow.keras.metrics import MeanSquaredError
9 from tensorflow.keras.utils import get_custom_objects
10
11 get_custom_objects().update({'mse': MeanSquaredError()})
12 app = Flask(__name__)
13
14 # Configura un directorio para guardar archivos subidos
15 app.config['UPLOAD_FOLDER'] = 'uploads/'
16
17 # Cargar el modelo y el normalizador
18 autoencoder = load_model('model/autoencoder.h5')
19 normalizador = load('normalizador/normalizador.pkl')
20
21 @app.route('/')
22 def index():
23     return render_template('index.html')
24
25 @app.route('/predict', methods=['POST'])
26 def predict():
27     if 'file' in request.files and request.files['file'].filename != '':
28         return predict_from_excel(request.files['file'])
29
30     try:
31         return predict_from_form(request.form)
32     except Exception as e:
33         # Si hay un error en el procesamiento del formulario, redireccionar a la página
34         # principal
35         print(e) # Es bueno registrar el error para depuración
36         return redirect(url_for('index'))
37
38 def predict_from_excel(file):
39     filename = secure_filename(file.filename)
40     filepath = os.path.join(app.config['UPLOAD_FOLDER'], filename)
41     file.save(filepath)
42     data = pd.read_excel(filepath)
43     return make_prediction(data)
```

```

43
44 def predict_from_form(form):
45     # Asegurarse de que todos los campos requeridos estn presentes
46     feature_names = [
47         'radius_mean', 'texture_mean', 'perimeter_mean', 'area_mean',
48         'smoothness_mean', 'compactness_mean', 'concavity_mean',
49         'concave_points_mean', 'symmetry_mean', 'radius_se', 'texture_se',
50         'perimeter_se', 'area_se', 'smoothness_se', 'compactness_se',
51         'concavity_se', 'concave_points_se', 'symmetry_se', 'fractal_dimension_se',
52         'radius_worst', 'texture_worst', 'perimeter_worst', 'area_worst',
53         'smoothness_worst', 'compactness_worst', 'concavity_worst',
54         'concave_points_worst', 'symmetry_worst', 'fractal_dimension_worst'
55     ]
56     data = {feature: [float(form[feature])] for feature in feature_names if feature in form}
57     df = pd.DataFrame(data)
58     return make_prediction(df)
59
60 def make_prediction(data):
61     if 'MEAN FRACTAL DIMENSION' in data.columns:
62         data = data.drop(columns=['MEAN FRACTAL DIMENSION'])
63
64     features = data[
65         'radius_mean', 'texture_mean', 'perimeter_mean', 'area_mean',
66         'smoothness_mean', 'compactness_mean', 'concavity_mean',
67         'concave_points_mean', 'symmetry_mean', 'radius_se', 'texture_se',
68         'perimeter_se', 'area_se', 'smoothness_se', 'compactness_se',
69         'concavity_se', 'concave_points_se', 'symmetry_se', 'fractal_dimension_se',
70         'radius_worst', 'texture_worst', 'perimeter_worst', 'area_worst',
71         'smoothness_worst', 'compactness_worst', 'concavity_worst',
72         'concave_points_worst', 'symmetry_worst', 'fractal_dimension_worst'
73     ].values
74
75     features_norm = normalizador.transform(features)
76     predictions = autoencoder.predict(features_norm)
77     reconstruction_errors = np.mean(np.power(features_norm - predictions, 2), axis=1)
78
79     umbral_fijo = 0.3
80     y_preds = [1 if e > umbral_fijo else 0 for e in reconstruction_errors]
81     resultados = ['Maligno' if pred == 1 else 'Benigno' for pred in y_preds]
82
83     return render_template('resultados.html', resultados=resultados)
84
85 if __name__ == '__main__':
86     app.run(debug=True, host='0.0.0.0')

```

Extracto de código 10.1: Configuración del servicio Flask

Bibliografía

- [1] <https://www.boe.es>. Boletín oficial del estado. URL https://www.boe.es/diario_boe/txt.php?id=BOE-A-2022-5438.
- [2] Dheeru Dua and Casey Graff. Breast cancer wisconsin (diagnostic) data set. UCI Machine Learning Repository, 2019. URL [https://archive.ics.uci.edu/ml/datasets/breast+cancer+wisconsin+\(diagnostic\)](https://archive.ics.uci.edu/ml/datasets/breast+cancer+wisconsin+(diagnostic)).
- [3] Chuanyang Zheng, Yixuan Wang, Yuqi Cheng, Xuesong Wang, Hongxin Wei, Irwin King, and Yu Li. scNovel: a scalable deep learning-based network for novel rare cell discovery in single-cell transcriptomics, 03 2024. ISSN 1477-4054. URL <https://doi.org/10.1093/bib/bbae112>.
- [4] Deval Shah. Few shot learning guide with meta-learnig, 2022. URL <https://www.v7labs.com/blog/few-shot-learning-guide>.
- [5] Kay Ka-Wai Li Guoqing Wu Zhong Yang Xin Gao Yingchao Liu Haibo Wu Hong Chen Qisheng Tang Liang Chen Yuanyuan Wang Ying Mao Ho-Keung Ng Zhifeng Shi Jinhua Yu Liangfu Zhou Xi Chen, Zhen Fan. Molecular subgrouping of medulloblastoma based on few-shot learning of multitasking using conventional mr images: a retrospective multicenter study, 2020. URL <https://academic.oup.com/noa/article/2/1/vdaa079/5860907>.
- [6] Meet Neri. Siamese network, 2023. URL <https://spotintelligence.com/2023/01/19/siamese-network-in-nlp/>.
- [7] Joshua B. Tenenbaum Brenden M. Lake, Ruslan Salakhutdinov. Human-level concept learning through probabilistic program induction, 2015. URL <https://www.science.org/doi/abs/10.1126/science.aab3050>.
- [8] Meet Neri. How to apply transfer learning to large language models (llms) — detailed explanation tutorial to fine tune a gpt-3 model, 2023. URL <https://spotintelligence.com/2023/03/28/transfer-learning-large-language-models/>.
- [9] <https://www.scribbledata.io/>. Zero shot learning: A complete guide. URL <https://www.scribbledata.io/blog/zero-shot-learning-a-complete-guide/>.
- [10] Mahsa Shahidi Mahdi Rezaei. Zero-shot learning and its applications from autonomous vehicles to covid-19 diagnosis: A review, 2020. URL <https://pubmed.ncbi.nlm.nih.gov/33043311/>.
- [11] Sebastian M. Waldstein Ursula Schmidt-Erfurth Georg Langs Thomas Schlegl, Philipp Seeböck. Unsupervised anomaly detection with generative adversarial networks to guide marker discovery, 2017. URL <https://arxiv.org/abs/1703.05921>.

- [12] Stefan Denner Benedikt Wiestler Nassir Navab Shadi Albarqouni Christoph Baur. Autoencoders for unsupervised anomaly segmentation in brain mr images: A comparative study, 2018. URL <https://www.sciencedirect.com/science/article/abs/pii/S1361841520303169>.
- [13] Apapan Pumsirirat Liu Yan. Credit card fraud detection using deep learning based on auto-encoder and restricted boltzmann machine, 2018. URL <https://www.semanticscholar.org/paper/Credit-Card-Fraud-Detection-using-Deep-Learning-on-Pumsirirat-Yan/01be7624aa0e0251182593350a984411c2e5128a>.
- [14] Mirtaheri S.L. Greco Torabi, H. S. practical autoencoder based anomaly detection by using vector reconstruction error. cybersecurity 6, 1, 2023. URL <https://link.springer.com/article/10.1186/s42400-022-00134-9>.
- [15] www.javatpoint.com. Random forest algorithm. URL <https://www.javatpoint.com/machine-learning-random-forest-algorithm>.
- [16] Andrew Cunningham Ferenc Huszár Lucas Theis, Wenzhe Shi. Lossy image compression with compressive autoencoders, 2017. URL <https://arxiv.org/abs/1703.00395>.
- [17] Canal de Youtube Bioestadística Para No Estadísticos. Curva roc, 2014. URL <https://www.youtube.com/watch?v=fsgDD0pNkZ0>.