Universidad de Valladolid

FACULTAD DE CIENCIAS

GRADO EN ESTADÍSTICA

Aplicación de redes neuronales al análisis de juegos con recompensas discretas

---

Alumno:
Pablo Simón Aparicio

Tutores:
Óscar Arratia García
Cesáreo Jesús González Fernández

Agradecimientos

A todos los profesores que me han formado durante estos cuatro años, por contribuir a construir los conocimientos que han hecho posible este trabajo.

A mis tutores, Cesáreo y Óscar, por guiarme durante la elaboración del proyecto.

A mi familia por estar siempre presente a lo largo de este camino.

Y a mi pareja, por su apoyo constante y por estar a mi lado en todo momento.

Resumen

En este trabajo se realiza un análisis comparativo del rendimiento de distintos algoritmos aplicados a un problema basado en la Teoría de Juegos con recompensas discretas. Concretamente, se genera un conjunto de datos que simula partidas entre dos jugadores, donde cada uno escoge una estrategia y ambos reciben un pago conjunto en función de su combinación. A partir de estos datos, se evalúa la capacidad predictiva de diversos modelos para clasificar correctamente las recompensas obtenidas.

Los algoritmos estudiados incluyen modelos lineales, como el Perceptrón, la Regresión Logística, el Descenso Estocástico del Gradiente (SGD) y Máquina de Vectores de Soporte (SVM) y modelos no lineales, como el Perceptrón Multicapa, k-Vecinos y XGBoost. También se consideran modelos probabilísticos y no supervisados como Naive Bayes y K-Means. La evaluación se realiza mediante validación cruzada estratificada, empleando como métrica principal la precisión, y se complementa con el análisis de la varianza y los tiempos de ejecución.

Los resultados muestran que los algoritmos no lineales superan ampliamente a los lineales en precisión, especialmente en un entorno donde las clases no son linealmente separables. Destacan XGBoost y k-Vecinos por su rendimiento y estabilidad. Asimismo, se profundiza en el comportamiento del algoritmo SVM con distintos kernels, analizando su sensibilidad a los parámetros y la aparición de sobreajuste en algunos escenarios. Este estudio evidencia el potencial de las redes neuronales y otras técnicas avanzadas de clasificación para modelar entornos estratégicos de decisión, y plantea futuras líneas de trabajo orientadas a juegos más complejos y análisis multiclase.

Palabras clave: Teoría de Juegos, redes neuronales, clasificación, aprendizaje automático, SVM, validación cruzada, sobreajuste.

Abstract

This work presents a comparative analysis of the performance of various algorithms applied to a problem based on Game Theory with discrete rewards. Specifically, a dataset is generated that simulates matches between two players, where each one selects a strategy and both receive a joint payoff depending on their combination. Based on this data, the predictive capacity of different models to correctly classify the obtained rewards is evaluated.

The studied algorithms include linear models, such as the Perceptron, Logistic Regression, Stochastic Gradient Descent (SGD), and Support Vector Machines (SVM) and non-linear models, such as the Multilayer Perceptron, k-Nearest Neighbors, and XGBoost. Probabilistic and unsupervised models are also included, such as Naive Bayes and K-Means. Evaluation is carried out using stratified cross-validation, employing accuracy as the main metric, and complemented by analysis of variance and execution times.

The results show that non-linear algorithms clearly outperform linear ones in terms of accuracy, especially in a setting where the classes are not linearly separable. XGBoost and k-Nearest Neighbors stand out for their performance and stability. Furthermore, the behavior of the SVM algorithm is explored with different kernels, analyzing its sensitivity to parameter tuning and the appearance of overfitting in certain scenarios. This study highlights the potential of neural networks and other advanced classification techniques to model strategic decision-making environments and outlines future research directions toward more complex games and multiclass analysis.

Keywords: Game Theory, neural networks, classification, machine learning, SVM, cross-validation, overfitting.

Índice general

Lista de Figuras

Lista de Tablas

Capítulo 1

Introducción

1.1. Contexto

En los últimos años, las redes neuronales se han consolidado como una de las herramientas más potentes dentro del campo del aprendizaje automático, ofreciendo soluciones eficaces a problemas complejos en múltiples disciplinas. Su capacidad para modelar relaciones no lineales y aprender patrones a partir de grandes volúmenes de datos las convierte en un instrumento especialmente valioso en contextos donde los métodos tradicionales resultan insuficientes.

Desde el punto de vista estadístico, las redes neuronales pueden interpretarse como modelos de regresión o clasificación altamente flexibles, capaces de captar relaciones complejas entre variables. Aunque a menudo se asocian con el ámbito de la inteligencia artificial y pueden parecer herramientas "inteligentes" por sí mismas, lo cierto es que su funcionamiento se basa en algo profundamente estadístico: el reconocimiento de patrones a partir de datos. No existe aprendizaje sin datos, y no se pueden extraer conclusiones útiles sin una comprensión adecuada del comportamiento de esos datos. Como estudiante de Estadística, me parece esencial subrayar que, detrás de cada red neuronal, hay una base estadística que permite formular, ajustar y validar los modelos. Este trabajo parte precisamente de esa unión entre estadística e inteligencia artificial, aplicando redes neuronales a un problema enmarcado en la Teoría de Juegos con recompensas discretas, para estudiar su capacidad de aprendizaje en entornos con múltiples agentes y decisiones estratégicas.

1.2. Objetivos

El objetivo principal de este Trabajo de Fin de Grado es comparar el rendimiento de distintos algoritmos de clasificación en un entorno simulado basado en Teoría de Juegos con recompensas discretas. Se pretende analizar la capacidad predictiva de estos modelos y extraer conclusiones sobre su aplicabilidad en contextos similares. Además, se presta especial atención a las redes neuronales, con el fin de estudiar cómo su arquitectura y parámetros afectan al rendimiento final en este tipo de problemas.

1.3. Estructura

Este documento se estructura de la siguiente manera:

Capítulo 2 - Marco teórico: Se describen los conceptos fundamentales para la comprensión del trabajo, incluyendo nociones básicas sobre Teoría de Juegos y redes neuronales.

Capítulo 3 - Metodología: Se explican los pasos seguidos durante el desarrollo del trabajo, incluyendo la generación del conjunto de datos, la selección de algoritmos y las métricas de evaluación empleadas.

Capítulo 4 - Comparación Redes lineales vs Redes no lineales: Aquí se presenta un análisis comparativo entre distintos modelos lineales y no lineales aplicados al problema descrito.

Capítulo 5 - Algoritmo SVM: Se estudia en detalle el comportamiento del algoritmo de Máquinas de Vectores de Soporte utilizando distintos kernels, analizando sus ventajas e inconvenientes en el contexto propuesto.

Capítulo 6 - Redes no lineales: Se profundiza en los modelos de redes neuronales no lineales, evaluando cómo afectan diferentes configuraciones y parámetros (como número de capas ocultas, número de vecinos o tasa de aprendizaje) al rendimiento del modelo.

Capítulo 7 - Conclusiones: Se resumen los principales hallazgos del trabajo y se proponen posibles líneas de mejora o extensión para futuras investigaciones.

Capítulo 2

Marco teórico

2.1. Teoría de Juegos

La Teoría de Juegos es una rama de la Matemática Aplicada y la Estadística que estudia las decisiones estratégicas entre agentes racionales. Se originó a mediados del siglo XX con los estudios de John von Neumann y Oskar Morgenstern, reflejados en su obra Theory of Games and Economic Behavior (1944) [1].

Desde entonces, la Teoría de Juegos se ha aplicado en diversos campos como la economía, la biología evolutiva, la política, la informática o la inteligencia artificial. Una de las contribuciones más influyentes fue la formulación del equilibrio de Nash, desarrollado por John Nash, quien recibió el Premio Nobel de Economía en 1994 por sus avances en el análisis de los juegos no cooperativos [2].

La Teoría de Juegos permite modelar situaciones en las que el resultado para un jugador depende no solo de sus propias decisiones, sino también de las de otros participantes, lo que hace posible analizar interacciones complejas.

2.1.1. Terminología básica

En la obra *Teoría de Juegos* de Emilio Cerdá Tena, Joaquín Pérez Navarro y José Luis Jimeno Pastor [3], se citan los conceptos más esenciales a la hora de hablar de la Teoría de Juegos que definiremos a través del siguiente ejemplo:

Ejemplo 1. Juego de pares o nones

Dos jugadores, a los que llamaremos J1 y J2, deben tomar una decisión simultánea entre las opciones pares (P) o nones (N). Si ambos seleccionan la misma alternativa, J2 deberá entregarle 5 euros a J1. En cambio, si sus elecciones son diferentes, será J1 quien pague 5 euros a J2. Así, cada jugador debe elegir sin conocer la decisión del otro, aunque sabiendo que el desenlace final dependerá de la combinación de ambas elecciones.

Jugadores

Se trata de los agentes que intervienen en el juego, responsables de tomar decisiones estratégicas con el objetivo de maximizar su utilidad. El número de jugadores puede ser igual o superior a dos. En el Ejemplo 1 tendríamos dos jugadores, J1 y J2, por tanto el conjunto de los jugadores es $J = \{1, 2\}$.

Acciones

Representan las distintas alternativas que un jugador puede seleccionar en cada turno o momento de decisión dentro del juego. El conjunto de acciones disponibles puede ser tanto finito como infinito, dependiendo de la naturaleza del juego. En el Ejemplo 1 cada jugador puede decidir entre sacar pares o sacar nones en cada turno. Por tanto el conjunto de las acciones de J1 es $A_1 = \{P, N\}$ y de J2 es $A_2 = \{P, N\}$.

Resultados

Corresponden a las posibles formas en que puede finalizar un juego. Cada resultado implica ciertas consecuencias para los jugadores, las cuales se especificarán posteriormente mediante los llamados pagos. En el Ejemplo 1 los resultados, dependiendo de las acciones que se tomen, serían que J1 gane 5 euros y J2 los pierda o que J1 pierda 5 euros y J2 los gane.

Estrategias. Perfiles de estrategias

Una estrategia consiste en el conjunto de decisiones o acciones que un jugador planifica seguir a lo largo del juego. Un perfil de estrategias se compone de una estrategia seleccionada por cada jugador, formando así una combinación completa de decisiones posibles entre todos los participantes. En el Ejemplo 1 J1 y J2 pueden decidir entre sacar pares o sacar nones, por tanto el conjunto de estrategias para J1 es $S_1 = \{P, N\}$ y el conjunto de estrategias para J2 es $S_2 = \{P, N\}$.

Los cuatro perfiles de estrategias que tendríamos, por tanto, en este juego serían: (P, P), (N, N), (P, N) y (N, P).

Pagos

En función del resultado final del juego, cada jugador recibe un pago asociado. Este pago representa la utilidad que el jugador atribuye al resultado obtenido, es decir, refleja la valoración individual que cada participante asigna a las consecuencias de una determinada situación final del juego.

En el Ejemplo 1 los pagos que obtienen J1 y J2 para cada perfil de estrategias son:

$$\begin{array}{ll} u_1(\text{P, P}) = 5 & u_2(\text{P, P}) = \text{-}5 \\ u_1(\text{P, N}) = \text{-}5 & u_2(\text{P, N}) = 5 \\ u_1(\text{N, P}) = \text{-}5 & u_2(\text{N, P}) = 5 \\ u_1(\text{N, N}) = 5 & u_2(\text{N, N}) = \text{-}5 \end{array}$$

Forma estratégica y forma extensiva

Estas son las dos principales maneras de representar un juego. Ambas describen los jugadores, sus acciones disponibles y los pagos resultantes. La forma estratégica (o forma normal) presenta el juego en una estructura tipo matriz, enfocándose en las estrategias completas de los jugadores, como si decidieran todas sus acciones desde el inicio. Por otro lado, la forma extensiva utiliza un diagrama en forma de árbol para destacar el orden en que se suceden las decisiones, permitiendo visualizar la secuencia de movimientos y sus posibles ramificaciones. A continuación podemos ver el Ejemplo 1 representado de forma estratégica, en la Tabla 2.1, y de forma extensiva, en la Figura 2.1.

		Jugador 2	
		P	N
Jugador 1	P	5, −5	−5, 5
	N	−5, 5	5, −5

Tabla 2.1: Ejemplo 1 expresado en forma estratégica.

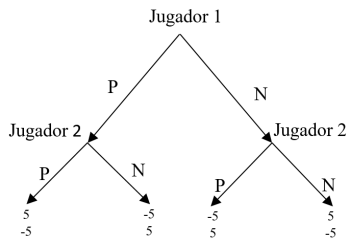

Figura 2.1: Ejemplo 1 expresado en forma extensiva.

2.1.2. Tipos de juegos

Se pueden clasificar los juegos en función de varias características. Distinguimos entre juegos cooperativos, si los jugadores pueden llegar a un acuerdo previo a actuar, y no cooperativos, si no puede existir un pacto previo.

Dentro de los juegos no cooperativos diferenciamos entre los juegos estáticos o dinámicos y entre juegos con o sin información completa. Los juegos estáticos son aquellos en los que los jugadores toman las decisiones sin saber las decisiones del resto. Mientras que en los juegos dinámicos el jugador conoce las decisiones que se han tomado antes de decidir él.

En los juegos con información completa todos los jugadores conocen de antemano los pagos para todos los jugadores. Sin embargo, en los juegos con información incompleta algún jugador desconoce alguno de estos pagos.

En el Ejemplo 1 estaríamos hablando de un juego no cooperativo ya que no pueden pactar nada, estático ya que los dos eligen simultáneamente y con información completa puesto que saben los posibles resultados desde el principio.

2.1.3. El equilibrio de Nash

Una de las contribuciones más significativas a la Teoría de Juegos fue realizada por John Nash, quien formuló el concepto de *equilibrio de Nash* en su artículo de 1950 [2]. Este describe una situación en la que, dado un perfil de estrategias, ningún jugador puede obtener un mayor beneficio modificando su estrategia de forma unilateral, siempre que las estrategias del resto permanezcan constantes.

El equilibrio de Nash proporciona una noción de estabilidad, en juegos no cooperativos, al reflejar un punto en el que todos los jugadores actúan racionalmente y no tienen incentivos para desviarse. Esta formulación generalizó los modelos anteriores al permitir el análisis de interacciones estratégicas sin necesidad de acuerdos previos o cooperación explícita.

Ejemplo 2. Dilema del Prisionero

Dos delincuentes, A y B, han sido condenados por un delito leve pero se les investiga por otro más grave. Cada uno puede optar por confesar (C) o guardar silencio (S). Si ambos guardan silencio, reciben 1 año de prisión cada uno. Si uno confiesa y el otro guarda silencio, el que confiesa queda libre y el otro cumple 10 años. Si ambos confiesan, cada uno recibe 5 años.

	B: S	**B: C**
A: S	$(-1,-1)$	$(-10,0)$
A: C	$(0,-10)$	$(-5,-5)$

Tabla 2.2: Matriz de pagos del Ejemplo 2.

Este ejemplo muestra cómo, aunque ambos jugadores obtendrían un mejor resultado si cooperaran y guardaran silencio, la opción de confesar se impone como la más segura para cada uno si actúan por separado.

Si A elige guardar silencio, B puede guardar silencio y tendría un año de prisión o confesar y no tendría que entrar a prisión. Por tanto la mejor decisión sería confesar.

Si A elige confesar, B puede guardar silencio y tendría 10 años de prisión o confesar y serían la mitad. De nuevo, la mejor decisión para B sería confesar.

Por eso, confesar es la estrategia elegida por ambos, y constituye un equilibrio de Nash: ninguno tiene incentivos para cambiar su decisión de forma unilateral.

2.2. Redes Neuronales

Las redes neuronales se inspiran en el funcionamiento del cerebro humano, intentando emular el modo en que las neuronas procesan y transmiten información. Hoy en día constituyen una de las principales herramientas dentro del campo de la inteligencia artificial y el aprendizaje automático.

El origen de estos modelos se remonta a los trabajos de McCulloch y Pitts [4], quienes en 1943 propusieron una primera aproximación matemática a una neurona artificial. Posteriormente, en los años 1950 y 60, se desarrollaron las primeras redes simples, aunque su uso se vio limitado hasta que en la década de 1980 se redescubrieron y mejoraron gracias al algoritmo de retropropagación (Detallado en el Apéndice A.0.1), lo que permitió entrenar redes multicapa con mayor eficacia. Hoy en día, las redes neuronales se utilizan ampliamente para resolver problemas complejos que requieren la detección de patrones en grandes cantidades de datos, como el reconocimiento de imágenes, la traducción automática, el diagnóstico médico o el análisis de decisiones estratégicas en entornos dinámicos como los juegos.

En el contexto de este trabajo, se explorará cómo estas redes pueden aplicarse a la resolución de juegos formulados desde la Teoría de Juegos, permitiendo modelar comportamientos óptimos sin necesidad de establecer reglas explícitas.

2.2.1. Cómo funcionan

Una red neuronal artificial se compone de unidades básicas, llamadas neuronas, organizadas en capas: una capa de entrada, una o varias capas ocultas y una capa de salida. Cada neurona recibe un conjunto de entradas, las pondera mediante unos valores llamados pesos, y aplica una función de activación para producir una salida. Esta salida se transmite a las neuronas de la siguiente capa, y así sucesivamente, hasta llegar a la capa de salida, donde se obtiene la predicción final del modelo. Durante el entrenamiento, la red ajusta sus pesos internos en función del error cometido para mejorar su rendimiento. Este proceso permite a las redes neuronales aprender patrones complejos en los datos, incluso sin necesidad de una programación explícita de reglas.

En la Figura 2.2 se muestra la estructura de una red neuronal profunda con tres o más capas ocultas, la capa de entrada y la capa de salida.

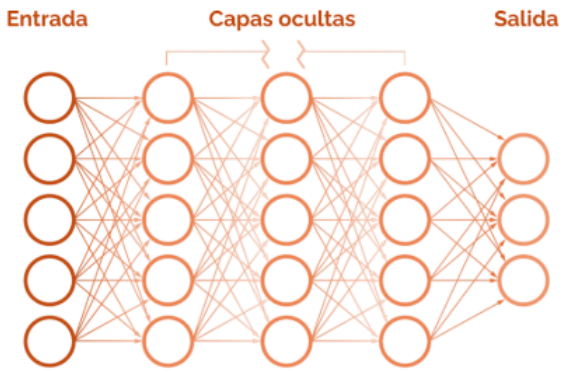

Figura 2.2: Esquema de una red neuronal profunda. *Fuente: Datademia (2021)* [5].

2.2.2. Tipos de redes neuronales

Las redes neuronales artificiales pueden clasificarse atendiendo a distintos criterios: el flujo de información, la arquitectura de la red y el tipo de aprendizaje entre otros.

Clasificación según el flujo de información

Según cómo fluye la información entre las neuronas y las capas, las redes neuronales pueden dividirse en los siguientes tipos:

- **Redes Feedforward:** La información se transmite en una única dirección, desde la capa de entrada hasta la de salida, sin ciclos ni retroalimentación. Son las redes más simples y comunes. Ejemplos representativos son el perceptrón multicapa y las redes convolucionales.

- **Redes Recurrentes (RNN):** Estas redes incorporan conexiones cíclicas, lo que permite que la salida de una neurona en un instante influya en su entrada futura. Son adecuadas para trabajar con datos secuenciales o temporales, como texto, audio o series temporales.

- **Redes Bidireccionales:** Extienden las RNN tradicionales procesando las secuencias tanto hacia adelante como hacia atrás, lo que permite a la red tener en cuenta el contexto completo de una secuencia. Son especialmente útiles en tareas como la traducción automática o el reconocimiento del habla.

Clasificación según la arquitectura

Desde el punto de vista estructural, las redes neuronales pueden clasificarse según el número y tipo de capas que las componen. Los principales tipos son:

- **Redes Monocapa:** Son las más simples, compuestas únicamente por una capa de salida conectada directamente a la entrada. Su capacidad de modelado es limitada, ya que solo pueden resolver problemas linealmente separables. El perceptrón simple es el ejemplo más representativo.

- **Redes Multicapa:** Incorporan una o más capas ocultas entre la entrada y la salida. Gracias al uso de funciones de activación no lineales y al entrenamiento mediante retropropagación, estas redes pueden modelar relaciones complejas y no lineales. Se utilizan ampliamente en tareas de clasificación y regresión.

- **Redes Convolucionales (CNN):** Especializadas en el procesamiento de datos con estructura espacial, como imágenes o señales bidimensionales. Su arquitectura se basa en capas de convolución (Detalladas en el Apéndice A.0.2) que extraen características locales, habitualmente seguidas por capas de pooling (Descritas en el Apéndice A.0.3) y capas densas. Son muy eficaces en tareas de visión por computador.

- **Redes Recurrentes (RNN):** Como hemos visto en la anterior clasificación, están diseñadas para tratar datos secuenciales. Incorporan conexiones recurrentes que les permiten conservar información de pasos anteriores, lo que resulta útil en tareas como el análisis de texto, el reconocimiento de voz o la predicción temporal.

Clasificación según el tipo de aprendizaje

Según el tipo de aprendizaje que emplean, las redes neuronales pueden clasificarse en tres grandes categorías:

- **Aprendizaje supervisado:** La red se entrena utilizando un conjunto de datos etiquetados. Cada dato de entrada tiene una salida deseada asociada, y el objetivo del entrenamiento es ajustar los pesos internos de la red para minimizar el error entre la salida predicha y la real, empleando técnicas como la retropropagación del error.
- **Aprendizaje no supervisado:** En este enfoque, la red se entrena con datos no etiquetados, tratando de identificar de forma autónoma patrones, agrupaciones o estructuras subyacentes sin conocer previamente las salidas esperadas.
- **Aprendizaje por refuerzo:** La red actúa como un agente que interactúa con un entorno dinámico. Toma decisiones con el objetivo de maximizar una señal de recompensa acumulada. A diferencia del aprendizaje supervisado, no se parte de datos etiquetados, sino que el conocimiento se adquiere a través de la experiencia y la retroalimentación de sus acciones.

Según las clasificaciones previamente expuestas, la red representada en la Figura 2.2 puede definirse como una red feedforward, ya que la información fluye únicamente hacia adelante, desde la capa de entrada hasta la de salida, sin ciclos ni retroalimentación. Asimismo, se trata de una red multicapa, al observarse la presencia de varias capas ocultas entre la entrada y la salida. En cuanto al tipo de aprendizaje, no es posible determinarlo a partir de la imagen, dado que esta no proporciona información suficiente al respecto.

2.2.3. Redes neuronales aplicadas en este TFG

Perceptrón Simple

El perceptrón simple es uno de los modelos más básicos de red neuronal. Fue propuesto por Frank Rosenblatt en 1958 [6] y se utiliza para resolver problemas de clasificación binaria, es decir, donde las respuestas posibles son solo dos (por ejemplo, sí o no, 0 o 1).

Este modelo funciona tomando varias entradas numéricas (que pueden representar características como altura, edad, etc.), les asigna un peso a cada una y calcula una combinación ponderada. Luego, ese valor total pasa por una función de activación que decide si la salida será positiva o negativa. Si el resultado supera un umbral, se clasifica en una clase, si no, en la otra.

En la Figura 2.3 se muestra un esquema visual del funcionamiento del perceptrón. Las entradas $x_1, x_2, \ldots, x_n$ son multiplicadas por sus respectivos pesos $w_1, w_2, \ldots, w_n$, se suman y se comparan con un umbral. La función de activación escalón decide el valor final de salida.

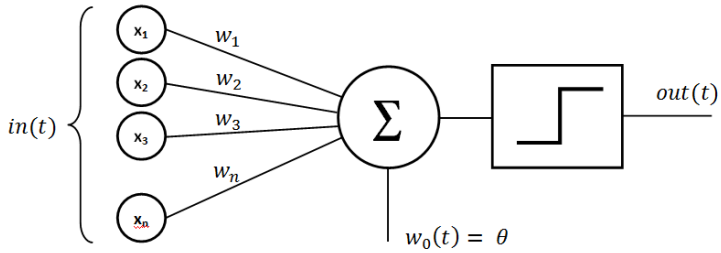

Figura 2.3: Esquema del perceptrón simple. *Fuente: DataScientest (2022)* [7].

Perceptrón Multicapa

El perceptrón multicapa (Multilayer Perceptron, MLP por sus siglas en inglés) es una extensión del perceptrón simple que permite resolver problemas más complejos, como la clasificación no lineal. Fue desarrollado en la década de 1980 como parte del avance en redes neuronales artificiales, y constituye la base de muchas redes neuronales profundas actuales.

A diferencia del perceptrón simple, que solo tiene una capa de neuronas (la capa de salida), el MLP incluye una o más capas ocultas entre las entradas y la salida. Cada neurona en estas capas realiza una operación similar a la del perceptrón simple: combina entradas ponderadas, aplica una función de activación (como la sigmoide, ReLU o tanh), y transmite la salida a la siguiente capa.

Este tipo de red es capaz de aprender representaciones internas más complejas de los datos, lo que permite clasificar correctamente incluso cuando las clases no son separables linealmente. El aprendizaje se realiza mediante un algoritmo llamado retropropagación del error, que ajusta los pesos internos minimizando el error entre la salida esperada y la obtenida.

En la Figura 2.4 se muestra un esquema de un MLP con una capa oculta. Las entradas se propagan hacia adelante y el error se retropropaga para ajustar los pesos.

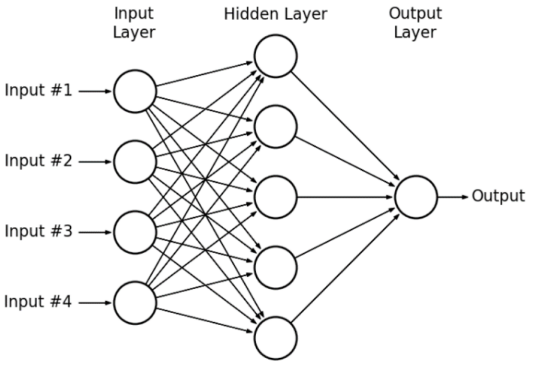

Figura 2.4: Esquema de un MLP con una capa oculta. *Fuente: Sensio, J. (2020)* [8].

2.3. Algoritmos de Aprendizaje Automático utilizados

Además de las redes neuronales, en este trabajo se han evaluado varios algoritmos clásicos de aprendizaje automático. Estos métodos no requieren arquitecturas profundas ni grandes volúmenes de datos para su entrenamiento, y son especialmente eficaces en problemas donde la dimensionalidad y la cantidad de muestras están bien equilibradas. A continuación se describen los modelos utilizados, seleccionados por su simplicidad, robustez y buena capacidad de generalización.

Descenso Estocástico del Gradiente

El descenso estocástico del gradiente (SGD, por sus siglas en inglés) es un algoritmo de optimización ampliamente utilizado en el entrenamiento de modelos de aprendizaje automático, especialmente en redes neuronales. Su objetivo es encontrar los valores óptimos de los parámetros (como los pesos en una red neuronal) que minimicen una función de pérdida, es decir, la diferencia entre las predicciones del modelo y los valores reales.

La idea principal del SGD es ajustar los parámetros del modelo utilizando el gradiente (la derivada) de la función de pérdida con respecto a cada parámetro. A diferencia del descenso del gradiente clásico, que calcula el gradiente usando todos los datos del conjunto de entrenamiento a la vez, el SGD actualiza los parámetros usando un único ejemplo (o un pequeño subconjunto aleatorio) en cada iteración. Esto hace que el algoritmo sea más rápido y eficiente, aunque más ruidoso e inestable, lo que a veces ayuda a escapar de mínimos locales.

Regresión Logistica

La regresión logística es un modelo de clasificación supervisado muy utilizado cuando la variable objetivo es binaria. Aunque su nombre sugiere un modelo de regresión, se utiliza para tareas de clasificación.

El funcionamiento de la regresión logística comienza de forma similar a una regresión lineal, calcula una combinación lineal de las variables de entrada ponderadas por sus respectivos coeficientes. Sin embargo, en lugar de devolver un valor numérico arbitrario, este valor se transforma mediante una función logística (también llamada sigmoide), que comprime la salida al intervalo (0, 1). Este resultado puede interpretarse como una probabilidad de pertenencia a una de las clases, y se asigna la clase más probable según un umbral, normalmente 0.5.

Clasificador Bayesiano

El clasificador Bayesiano, o Naive Bayes, es un modelo de clasificación supervisada basado en el Teorema de Bayes. Parte del supuesto de independencia entre las variables predictoras, lo cual simplifica los cálculos y hace que el algoritmo sea muy eficiente. El modelo estima la probabilidad de que una observación pertenezca a una clase dada, basándose en las probabilidades condicionadas de cada atributo.

A pesar de su simplicidad y del supuesto fuerte de independencia, Naive Bayes ha demostrado ser muy eficaz en tareas como clasificación de texto, detección de spam y análisis de sentimientos.

k-Vecinos

El algoritmo de los *k*-Vecinos más cercanos (*k*-Nearest Neighbors, *k*-NN por sus siglas en inglés) es un método de clasificación no paramétrico basado en instancias. Para clasificar una nueva observación, el algoritmo busca las *k* instancias más cercanas (según una métrica de distancia como la euclídea) en el conjunto de entrenamiento y decide su clase por mayoría de votos.

Este modelo no realiza un proceso de entrenamiento propiamente dicho, sino que guarda todos los datos de entrenamiento y los utiliza directamente al predecir. Por ello, se le denomina un “método perezoso".

k-NN es fácil de implementar y puede ser muy eficaz en problemas con límites de decisión complejos, aunque su rendimiento depende fuertemente del valor de *k* y de la escala de las variables.

Máquinas de vectores soporte

Las máquinas de vectores soporte (Support Vector Machine, SVM por sus siglas en inglés) son modelos de clasificación supervisada que buscan encontrar un hiperplano que separe las clases de manera óptima. La idea clave es maximizar el margen, es decir, la distancia entre el hiperplano y los puntos de datos más cercanos de cada clase, conocidos como vectores soporte. Si los datos son linealmente separables, la SVM encuentra la separación con el mayor margen posible.

En casos más complejos, donde no es posible separar las clases con una línea o plano, se utilizan funciones de transformación conocidas como kernels, que permiten proyectar los datos a un espacio de mayor dimensión donde sí puedan separarse linealmente. Las SVM son especialmente útiles en problemas de clasificación binaria y destacan por su robustez y capacidad de generalización, incluso con conjuntos de datos pequeños o con muchas variables.

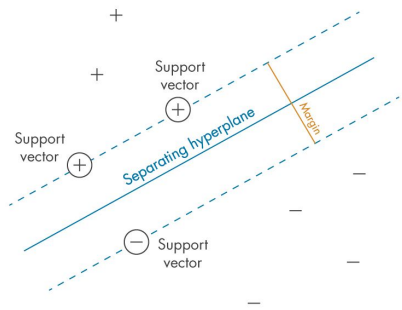

Figura 2.5: Representación gráfica del funcionamiento del algoritmo SVM. *Fuente: MathWorks* [9].

XGBoost

XGBoost (Extreme Gradient Boosting) es un algoritmo de aprendizaje supervisado basado en árboles de decisión y optimización por gradiente. Es una de las técnicas más potentes y utilizadas en la práctica para problemas de clasificación y regresión, debido a su alto rendimiento y flexibilidad.

Este modelo construye un conjunto de árboles de decisión de manera secuencial. Cada nuevo árbol se entrena para corregir los errores del modelo anterior, minimizando una función de pérdida mediante un procedimiento de descenso por gradiente. Además, XGBoost incluye técnicas como regularización, poda y paralelización, lo que mejora la precisión y evita el sobreajuste.

K-Means

K-Means es un algoritmo de agrupamiento (clustering) no supervisado que tiene como objetivo dividir un conjunto de datos en k grupos o clústeres, en función de su similitud. El algoritmo comienza seleccionando k centroides iniciales y luego asigna cada observación al centro más cercano. Posteriormente, se recalculan los centroides como el promedio de los puntos asignados a cada grupo y se repite el proceso hasta que las asignaciones no cambien.

Aunque no es un modelo de clasificación en el sentido tradicional, *K*-Means es útil para identificar patrones o estructuras ocultas en los datos y puede servir como paso previo para otras técnicas de aprendizaje supervisado.

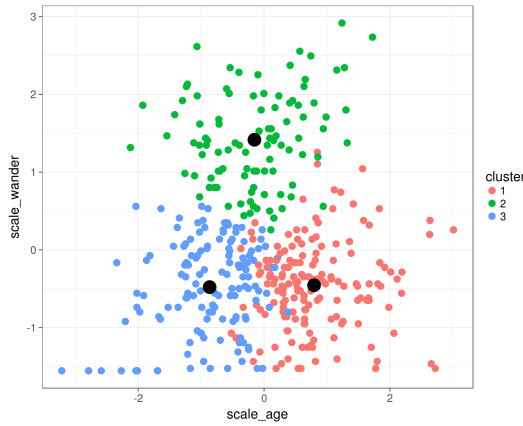

Figura 2.6: Ejemplo de segmentación mediante el algoritmo *K*-Means con tres clústeres. *Fuente: Cyobero (2018)* [10].

Capítulo 3

Metodología

Para estudiar el ajuste de los distintos algoritmos descritos anteriormente a un modelo de la Teoría de Juegos primero hemos de elegir el modelo a utilizar, tendremos que generar muestras de datos de dicho modelo y por último ajustar los algoritmos a nuestro conjunto de datos y valorar los resultados obtenidos.

Para todos estos procesos se ha utilizado el lenguaje de programación Python.

3.0.1. Modelo de Juego elegido

El problema elegido ha sido un juego bipersonal con 3 estrategias por jugador.

Cada jugador ha de elegir una de las tres estrategias, y en función de la distancia entre las estrategias elegidas por ambos jugadores recibirán ambos el mismo pago. Si eligen la misma estrategia ambos recibirán 1 punto. Si eligen una estrategia distinta en 1 unidad no recibirán ningún punto. Y si eligen estrategias completamente dispares perderán un punto.

	B: 0	**B: 1**	**B: 2**
A: 0	$(1,1)$	$(0,0)$	$(-1,-1)$
A: 1	$(0,0)$	$(1,1)$	$(0,0)$
A: 2	$(-1,-1)$	$(0,0)$	$(1,1)$

Tabla 3.1: Matriz de pagos.

El juego elegido es no cooperativo, estático y con información completa. Es no cooperativo porque los jugadores actúan sin posibilidad de formar acuerdos. Estático, al tomarse las decisiones de forma simultánea. Y con información completa, dado que todos los pagos son conocidos por los participantes.

Esta elección también permite observar cómo los modelos de aprendizaje manejan entornos donde la estrategia óptima depende del comportamiento del otro jugador.

Para generar las muestras de datos de este problema he creado una función en Python que toma como parámetros el número de jugadores, el número de estrategias y el número de muestras, y que devuelve como salida un array de la librería NumPy que, al ejecutarlo fijando el número de jugadores en dos, está formado por tres columnas, la estrategia jugada por el primer jugador, la elegida por el segundo y, por último, el pago obtenido.

La elección de las estrategias por parte de cada jugador ha sido aleatoria a través de la función *np.random* de la librería NumPy y, para añadir realidad y complejidad a nuestros conjuntos de datos, he añadido que con una probabilidad del 20 % se cambie aleatoriamente el pago recibido.

3.0.2. Datos empleados

Como hemos descrito en el punto anterior, nuestro conjunto de datos tiene la siguiente forma:

```
array([[ 2,  0, -1],
       [ 0,  1,  0],
       [ 1,  2,  1],
       [ 1,  0,  0],
       [ 0,  2, -1],
       [ 2,  0, -1],
       [ 1,  0,  0],
       [ 2,  0, -1],
       [ 2,  1,  0],
       [ 1,  0,  0]])
```

Figura 3.1: Conjunto de datos obtenido al ejecutar la función con 2 jugadores, 3 estrategias y 10 muestras como parámetros.

Ya en la Figura 3.1 podemos ver cómo hay perfiles de estrategias que no reciben los pagos descritos en la matriz de pagos, estas muestras son consecuencia del ruido introducido.

Dado que algunos clasificadores requieren que las etiquetas sean enteros no negativos se ha aplicado una transformación lineal $y' = y + 1$, de modo que las clases originales -1, 0, 1 pasan a ser 0, 1, 2. Por tanto, en adelante se trabajará con estas nuevas etiquetas transformadas.

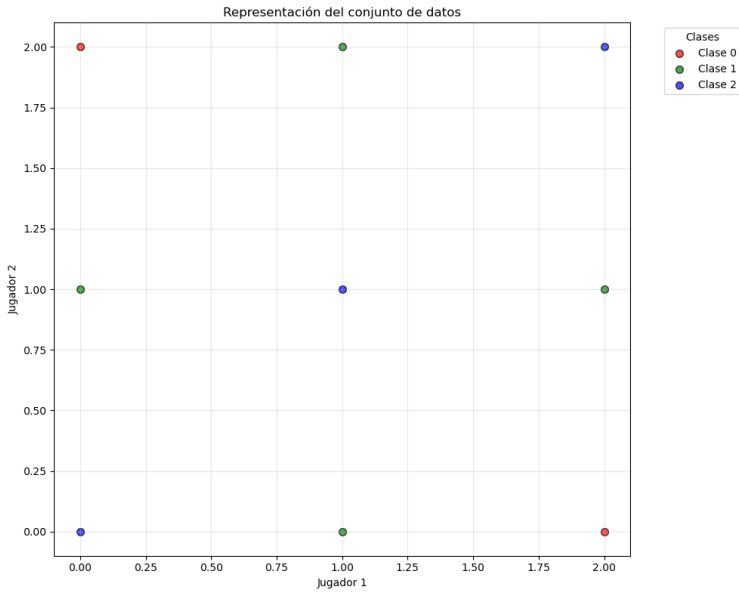

Figura 3.2: Representación gráfica del conjunto de datos sin ruido.

En la Figura 3.2 se representan los 9 perfiles de estrategias posibles, junto con su respectiva clasificación. Dado que tratamos con datos discretos y un mismo perfil de estrategias siempre obtiene el mismo pago (ignorando el posible ruido introducido) es razonable esperar que los algoritmos de clasificación se ajusten adecuadamente a nuestro conjunto de datos. Por otro lado, dado que las clases no son linealmente separables, los algoritmos lineales *a priori* no obtendrán los mejores resultados.

Para separar el conjunto de datos para entrenamiento y prueba he optado por utilizar la validación cruzada de k particiones. Se reparten los datos en k subconjuntos disjuntos del mismo tamaño y, en cada iteración, se utiliza un subconjunto como prueba y el resto de entrenamiento. Para asegurar que todas las clases están representadas en todos los subconjuntos utilizamos la variante estratificada.

Para realizar este procedimiento antes de introducir los datos de entrenamiento en los algoritmos utilizo la función *StratifiedKFold* de la librería Scikit-learn.

Al utilizar la validación cruzada estamos ejecutando los algoritmos varias veces sobre el mismo conjunto de datos por lo que no se puede aproximar a la normal.

La tasa de error es el promedio de las obtenidas en cada partición:

$$\bar{e}(h) = \frac{\sum_{i=1}^{k} e_i(h)}{k}. \tag{3.1}$$

Para estimar la varianza utilizaremos la varianza muestral:

$$S^2_{e(h)} = \frac{1}{k-1} \sum_{i=1}^{k} (e_i(h) - \bar{e}(h))^2. \tag{3.2}$$

Y para los intervalos de confianza, con probabilidad $N\%$ la tasa de error está en el intervalo:

$$\bar{e}(h) \pm t_{N,k-1} \times \frac{S_e(h)}{\sqrt{k}}. \tag{3.3}$$

3.0.3. Aplicación de las redes neuronales

Para la implementación de los distintos algoritmos se empleará el lenguaje de programación Python mediante Jupyter Notebooks, utilizando Visual Studio Code como entorno de desarrollo integrado. Los proyectos Jupyter son muy utilizados en la ciencia de datos, machine learning así como en la investigación y en la educación ya que son interactivos, se puede ejecutar el código en bloques y se puede intercalar con gráficos y texto indistintamente.

Las librerías de Python más relevantes utilizadas son las siguientes:

- **Pandas y NumPy:** Librerías esenciales para la manipulación y análisis de datos. **Pandas** proporciona estructuras de datos como DataFrame que permiten gestionar conjuntos de datos de forma eficiente, mientras que **NumPy** ofrece funcionalidades avanzadas para cálculos numéricos y operaciones matriciales.
- **Scikit-learn:** Librería fundamental en el ámbito del aprendizaje automático. Implementamos todos los algoritmos incluidos en el estudio excepto XGBoost así como para aplicar técnicas de validación cruzada y calcular las métricas de evaluación, como la precisión.
- **XGBoost:** Librería optimizada para algoritmos de potenciación del gradiente (gradient boosting). Utilizada para evaluar el rendimiento del algoritmo XGBoostClassifier.

- **Matplotlib:** Librería especializada en la generación de gráficos. Nos permitirá enriquecer el análisis exploratorio así como la visualización de los resultados obtenidos.

Para evaluar cada algoritmo he definido funciones que toman como parámetros los conjuntos de entrenamiento y prueba y devuelven el porcentaje de precisión obtenido. A continuación se muestra como ejemplo la función correspondiente al Perceptrón Simple:

```
def perceptron(X_train, X_test, y_train, y_test):
    modelo = Perceptron(max_iter=1000)
    modelo.fit(X_train, y_train)
    y_pred = modelo.predict(X_test)
    return accuracy_score(y_test, y_pred) * 100
```

3.0.4. Entorno de ejecución

Todos los experimentos computacionales se han llevado a cabo en un ordenador portátil personal con las siguientes especificaciones técnicas:

- **Procesador**: Intel® Core™ i5-10210U CPU @ 1.60 GHz (4 núcleos, 8 hilos), frecuencia base aproximada de 2.1 GHz.
- **Memoria RAM**: 8 GB DDR4.
- **Disco**: SSD PHISON S11-256G (256 GB).
- **Sistema operativo**: Windows 11 Pro de 64 bits (versión 10.0, compilación 26100).
- **Gráficos**: Intel® UHD Graphics.

Es importante tener en cuenta que los tiempos de ejecución pueden variar significativamente en función del equipo utilizado, por lo que los resultados obtenidos deben interpretarse siempre en relación con el entorno descrito.

Capítulo 4

Comparación redes lineales vs redes no lineales

Uno de los principales motivos por los que un algoritmo obtiene mejores resultados es por cómo modela la relación entre las variables de entrada y salida. Por ello a la hora de comparar los resultados obtenidos hemos dividido los distintos algoritmos en lineales, no lineales y probabilísticos y clustering.

Antes de ver los resultados es importante tener en cuenta que como hemos comentado en el capítulo anterior, nuestras clases no son linealmente separables por lo que *a priori* esperamos mejores resultados de los algoritmos capaces de encontrar relaciones no lineales.

Algoritmos lineales:

Tamaño	SVM (Linear)	Perceptrón	SGD	R. Logística
1000	43.300	36.000	39.400	43.300
5000	38.340	39.680	43.260	43.000
10000	29.380	39.620	34.960	42.150
50000	33.364	31.544	32.358	42.444
100000	33.309	30.682	32.588	42.008

Tabla 4.1: Medias de precisión (%) - Algoritmos lineales (10-fold CV).

Tamaño	SVM (Linear)	Perceptrón	SGD	R. Logística
1000	0.233333	154.888889	119.155556	0.233333
5000	111.813778	84.037333	54.133778	0.000000
10000	82.226222	100.164000	55.442667	0.002778
50000	0.156960	86.709316	114.585462	0.000071
100000	0.000010	85.803618	60.157796	0.000018

Tabla 4.2: Varianza - Algoritmos lineales (10-fold CV).

En las Tablas 4.1 y 4.2 se muestran las medias y las varianzas de precisión obtenidas con los distintos algoritmos lineales. Podemos confirmar lo que esperábamos: al no haber una relación lineal en el problema descrito la capacidad de estos modelos se ve muy limitada.

En general vemos baja precisión en torno al 35 %-50 % y una varianza muy alta en general, lo que indica inestabilidad y que son muy sensibles al conjunto de entrenamiento.

Cabe destacar que la varianza del SVM cae drásticamente conforme aumenta el tamaño lo que indica mayor fiabilidad cuantos más datos tenemos y que el límite de lo que puede aprender SVM con una frontera lineal ya está alcanzado, por lo que probaremos cómo se comporta variando sus parámetros.

Lo más llamativo es la prácticamente nula varianza en el algoritmo de Regresión Logística lo que indica su gran estabilidad.

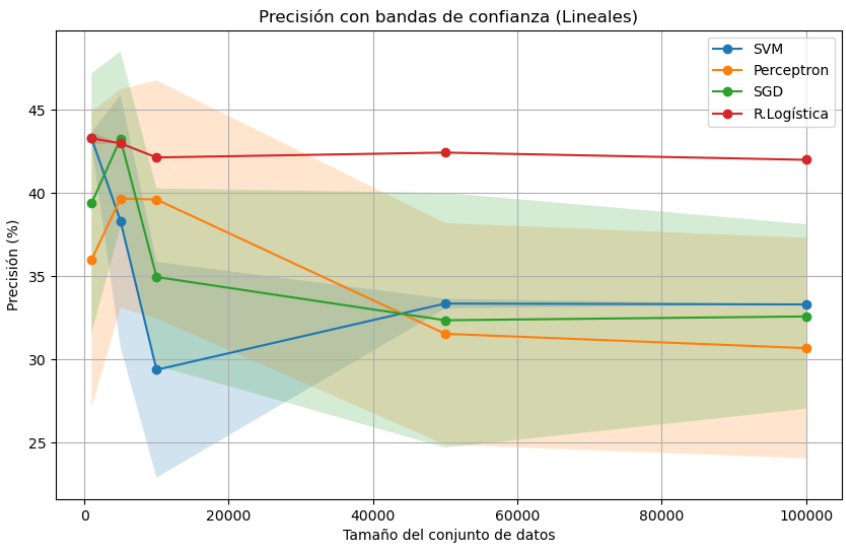

Figura 4.1: Bandas de confianza 95 % para la precisión de los algoritmos lineales.

Podemos observar claramente en la Figura 4.1 cómo llama la atención que la anchura de la banda de confianza de la Regresión Logística es prácticamente nula como hemos comentado anteriormente y que este algoritmo nos da la precisión media más alta, pero aún así no llega al 45 %.

Tamaño	SVM (Linear)	Perceptrón	SGD	R. Logística
1000	0.323491	0.073725	0.085106	0.065361
5000	5.631418	0.093993	0.845602	0.098663
10000	13.314132	0.155992	1.692042	0.156896
50000	53.170660	0.727970	4.344104	0.692419
100000	113.707700	1.964356	8.060205	1.570906

Tabla 4.3: Tiempos de ejecución (segundos) - Algoritmos lineales.

La Tabla 4.3 recoge los tiempos de ejecución de los algoritmos lineales. Podemos ver que en general son muy rápidos, como es de esperar el tiempo es directamente proporcional al tamaño del conjunto de datos. Los tiempos de ejecución del SVM resultan elevados en comparación con el resto de modelos: tarda hasta 114 segundos en ejecutarse con 100000 datos.

Algoritmos no lineales:

Tamaño	k-Vecinos	XGBoost	P.Multicapa
1000	88.200	88.200	87.400
5000	86.780	86.780	84.260
10000	87.110	87.110	87.110
50000	86.896	86.896	86.896
100000	86.552	86.552	86.552

Tabla 4.4: Medias de precisión (%) - Algoritmos no lineales (10-fold CV).

Tamaño	k-Vecinos	XGBoost	P.Multicapa
1000	10.400000	10.400000	13.600000
5000	2.519556	2.519556	68.151556
10000	1.378778	1.378778	1.378778
50000	0.281760	0.281760	0.281760
100000	0.098840	0.098840	0.098840

Tabla 4.5: Varianza - Algoritmos no lineales (10-fold CV).

En las Tablas 4.4 y 4.5 recogemos las medias y varianzas de precisión relativos a los algoritmos no lineales. Tal como preveíamos en la introducción del apartado los resultados obtenidos con los algoritmos no lineales han sido los mejores en términos de precisión. Los tres alcanzan precisiones en torno al 86 % lo cual es notable teniendo en cuenta que un 20 % de las etiquetas son alteradas aleatoriamente.

La varianza disminuye notablemente al aumentar el tamaño del dataset, por lo que podemos decir que son muy fiables con datasets grandes.

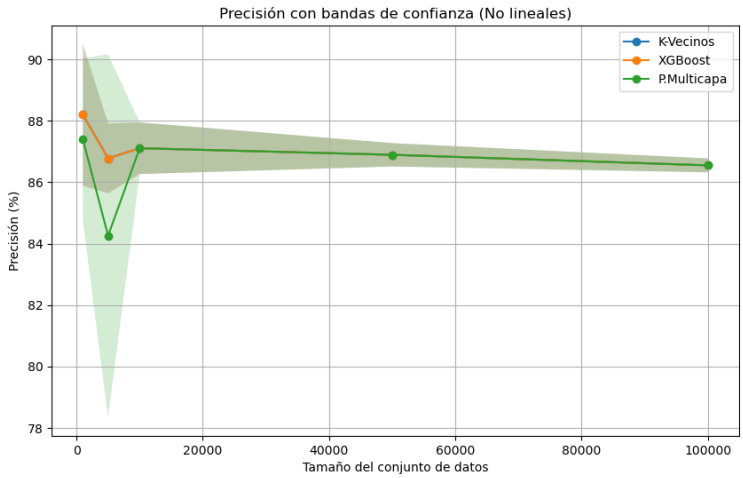

Figura 4.2: Bandas de confianza 95 % para la precisión de los algoritmos no lineales.

En la Figura 4.2 se observa claramente cómo los algoritmos *k*-Vecinos y XGBoost obtienen los mismos resultados superponiéndose ambas curvas. Es notablemente más ancha la banda de confianza del Perceptrón Multicapa en los resultados con un tamaño más pequeño debido a la elevada varianza de estos pero según aumenta el tamaño obtenemos los mismos resultados que con los otros dos algoritmos, estabilizándose en torno al 87 % de precisión.

Tamaño	***k*-Vecinos**	**XGBoost**	**P. Multicapa**
1000	0.105126	0.651190	4.353663
5000	0.293928	0.926704	10.095787
10000	0.783220	1.446979	14.858553
50000	5.118604	3.716069	23.534265
100000	16.917293	7.963166	45.078522

Tabla 4.6: Tiempos de ejecución (segundos) - Algoritmos no lineales.

En la Tabla 4.6 se pueden consultar los tiempos de ejecución de los algoritmos no lineales. Se observa que los tiempos de ejecución son más altos que los de los algoritmos lineales si exceptuamos SVM, y nuevamente son directamente proporcionales al tamaño del conjunto de datos. Cabe destacar la buena relación rendimiento-tiempo de ejecución del algoritmo XGBoost, que ha tenido una precisión media similar al resto de algoritmos con un tiempo mucho más bajo.

Probabilísticos y Clustering :

Tamaño	Naive Bayes	K-Means
1000	43.300	27.900
5000	43.000	27.540
10000	42.150	35.470
50000	42.444	31.530
100000	42.008	26.372

Tabla 4.7: Medias de precisión (%) - Probabilísticos y Clustering (10-fold CV).

Tamaño	Naive Bayes	K-Means
1000	0.233333	86.100000
5000	0.000000	63.013778
10000	0.002778	203.737889
50000	0.000071	65.682333
100000	0.000018	66.869440

Tabla 4.8: Varianza - Probabilísticos y Clustering (10-fold CV).

En las Tablas 4.7 y 4.8, relativas a las medias y varianzas de precisión obtenidas con los algoritmos probabilísticos y de clustering, podemos observar cómo Naive Bayes, al asumir independencia fuerte entre las variables no logra una alta precisión. Sí que observamos que la varianza es muy baja.

En cuanto a *K*-Means el motivo de su mal rendimiento se debe a que es un método no supervisado aplicado a un problema de clasificación multiclase, por tanto no se captura correctamente la estructura del objetivo.

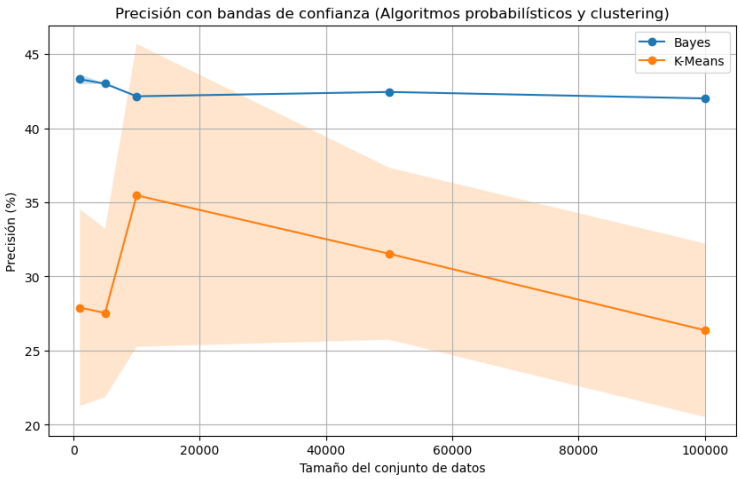

Figura 4.3: Bandas de confianza 95 % para la precisión de los algoritmos probabilísticos y de clustering.

En la Figura 4.3 se observa cómo la anchura de las bandas de confianza es notablemente diferente haciendo visible la diferencia de estabilidad entre los resultados obtenidos con cada algoritmo. Como ya hemos mencionado anteriormente no alcanzamos ni el 45 % de la precisión con estos algoritmos.

Tamaño	**Bayes**	**K-Means**
1000	0.018013	0.488586
5000	0.021074	0.058904
10000	0.028958	0.065962
50000	0.089026	0.193856
100000	0.176982	0.340006

Tabla 4.9: Tiempos de ejecución (segundos) - Algoritmos probabilísticos y clustering.

En cuanto a los tiempos, en la Tabla 4.9, podemos ver que son tiempos buenos y muy competitivos pero su baja precisión los hace una mala solución predictiva para este modelo.

Conclusión

Como esperábamos, los modelos que mejor se han adaptado a nuestro conjunto de datos han sido aquellos que son capaces de encontrar relaciones no lineales complejas. Por ello, a partir de ahora nos centraremos en estos para estudiar sus múltiples variantes, cómo se adaptan a nuestro modelo y cómo estas influyen en los resultados obtenidos.

En cuanto a los tiempos de ejecución vemos cómo el Perceptrón Multicapa es el que más tarda con tamaños más bajos pero el que peor escala es el SVM agrandado llamativamente la diferencia de tiempos según aumentamos el tamaño del conjunto de datos. El resto de algoritmos tiene tiempos similares y más razonables.

Como hemos comentado anteriormente el mejor algoritmo en relación precisión y tiempo de ejecución es el XGBoost.

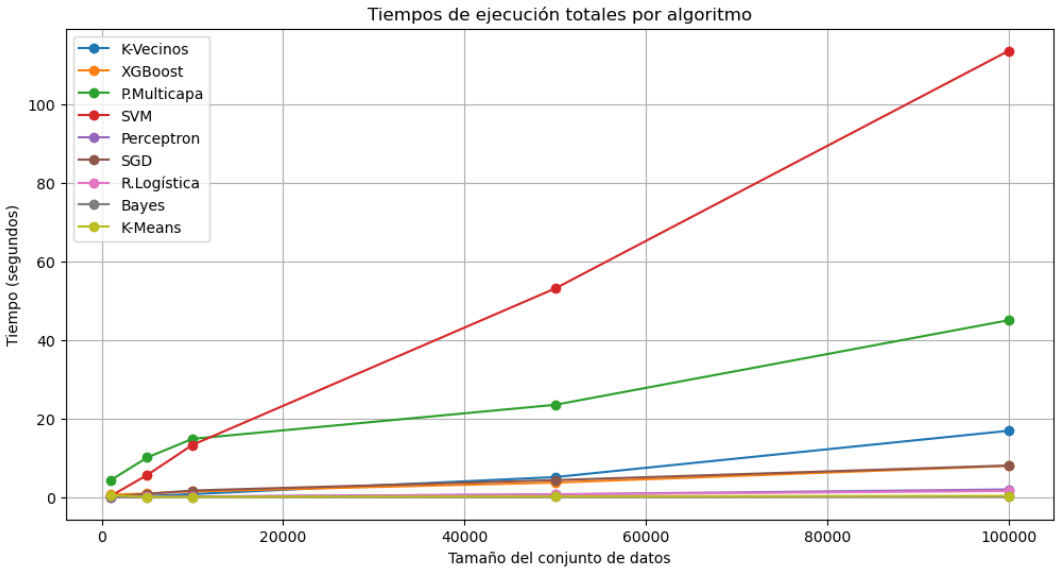

Figura 4.4: Tiempos de ejecución de todos los algoritmos.

Capítulo 5

Algoritmo SVM

El propósito de este capítulo es optimizar el rendimiento del algoritmo SVM. Para ello, se comparan sus principales kernels y se lleva a cabo una experimentación con el kernel RBF, ajustando sus parámetros clave para maximizar la precisión del modelo.

Kernel lineal

El kernel lineal es el único que no transforma el dataset a un espacio de mayor dimensionalidad. Calcula el producto escalar de los dos vectores y añade una constante.

$$K(x,y) = x^T y + c \tag{5.1}$$

No podemos esperar buenos resultados puesto que como hemos visto anteriormente nuestros datos no son linealmente separables. En la Figura 5.1 vemos las fronteras de decisión generadas al entrenar el algortimo SVM con este Kernel.

Al tener unos datos que no son linealmente separables, clasifica todos los puntos a la clase mayoritaria, en este caso la Clase 1.

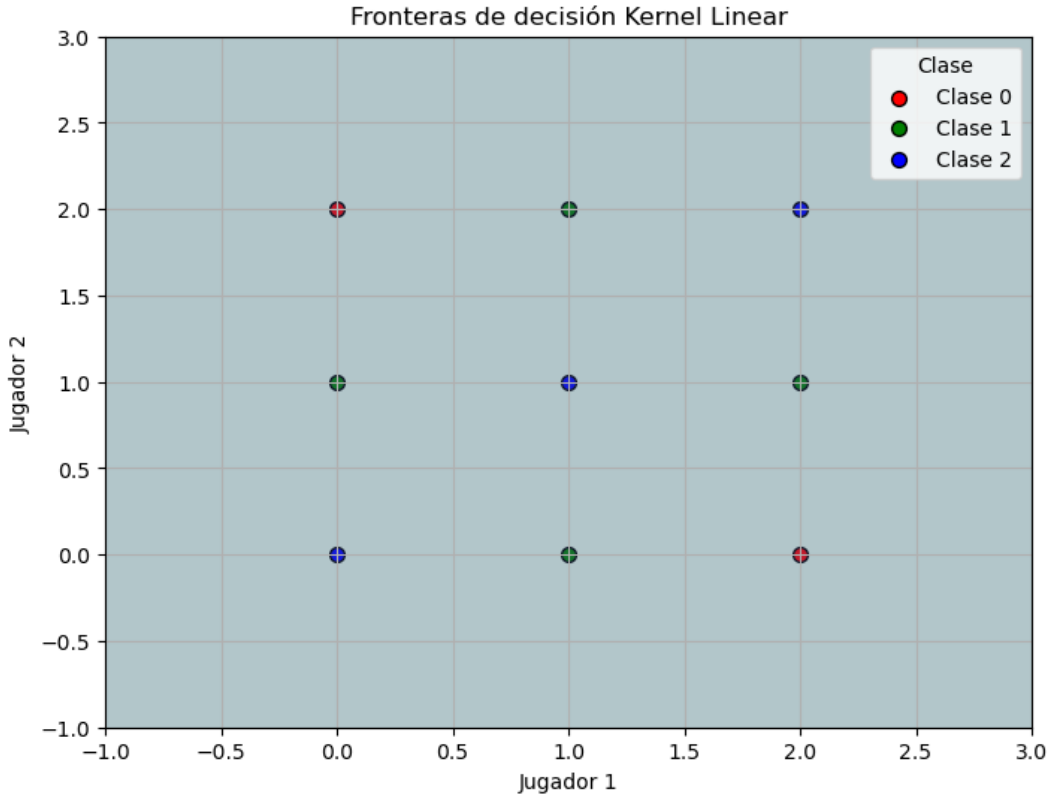

Figura 5.1: Frontera de decisión del algoritmo SVM con kernel lineal.

Función de base radial (RBF)

En este kernel, como podemos ver, tendremos la opción de ajustar el parámetro σ. Para valores muy bajos de σ tenderá a comportarse linealmente mientras que para valores altos tenderá al sobreajuste.

$$K(x,y) = e^{\left(-\frac{\|x-y\|^2}{2\sigma^2}\right)} = e^{(-\gamma\|x-y\|^2)} \quad \text{donde } \gamma = \frac{1}{2\sigma^2} \tag{5.2}$$

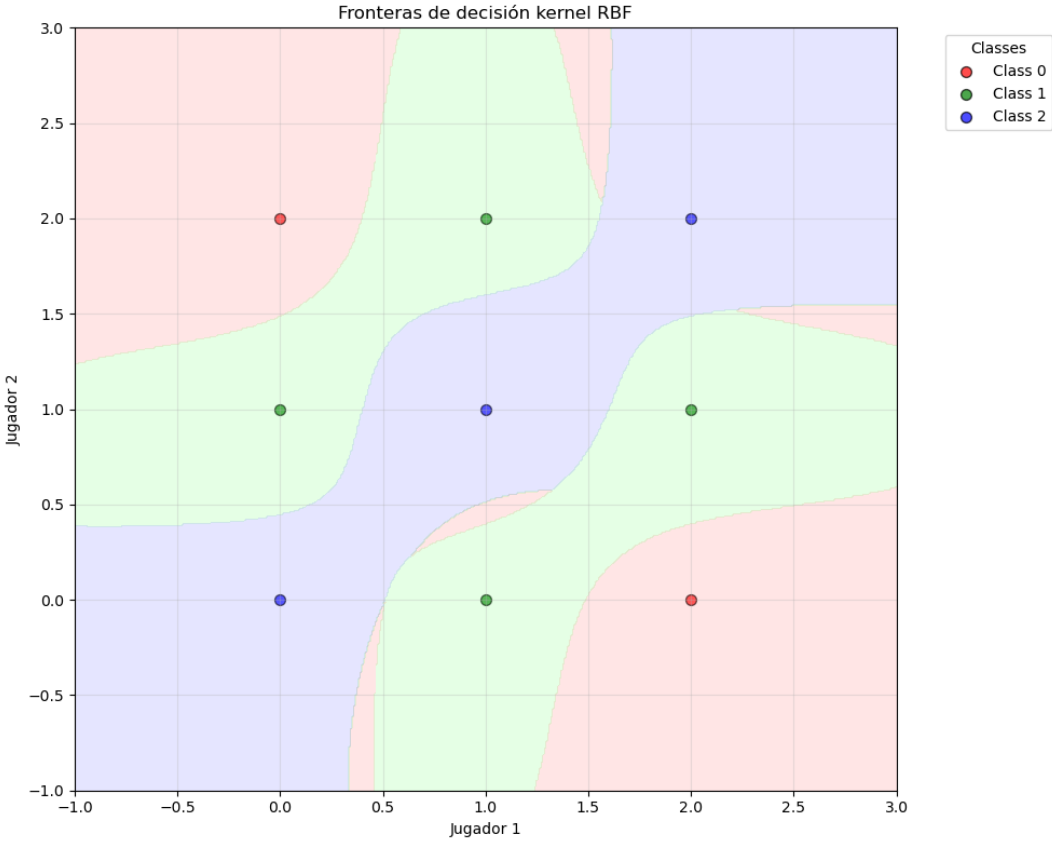

Figura 5.2: Frontera de decisión del SVM (kernel RBF), con parámetros por defecto y n = 10000.

En este caso podemos ver fronteras de decisión más afines a nuestro conjunto de datos. Observamos en la Figura 5.2 cómo las áreas de clasificación son las correctas por lo que podemos esperar que este kernel solo clasifique mal el ruido.

Kernel polinómico

Utilizando este kernel podemos aumentar la dimensionalidad aumentando el grado del polinomio, siendo el kernel lineal un caso específico de este.

$$K(x,y) = (\gamma x^T y + c)^d \tag{5.3}$$

Si fijamos el grado a 2 estaremos convirtiendo nuestro dataset a un espacio de 4 dimensiones, si lo fijamos a grado 3 será de 6 dimensiones y así progresivamente.

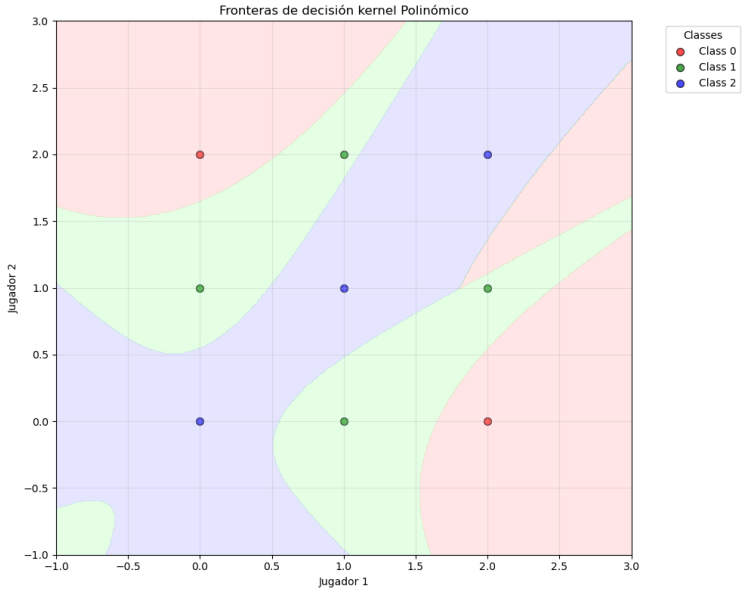

Figura 5.3: Frontera de decisión del algoritmo SVM con kernel polinómico, parámetros iniciales por defecto y n=10000.

En la Figura 5.3, que muestra las fronteras de decisión obtenidas con el algoritmo SVM y kernel polinómico, volvemos a obtener áreas de clasificación correctas para nuestro conjunto de datos por lo que *a priori* podemos esperar buenos resultados. El grado que toma por defecto esta librería es 3, por ello vemos formas carácterísticas de estos polinomios en nuestras fronteras de decisión.

Es importante destacar la diferencia entre estas fronteras de decisión y las del kernel RBF. En la Figura 5.2 puede observarse cómo el kernel RBF tiende a generar fronteras que maximizan la separación entre clases, alejándolas de los puntos de entrenamiento. En contraste, el kernel polinómico produce fronteras más ajustadas a los datos, situándose muy próximas a los puntos clasificados.

En nuestro caso particular, dado que trabajamos con un conjunto de datos discretos, esta diferencia no resulta problemática. Sin embargo, en contextos con datos continuos, el comportamiento del kernel polinómico puede dar lugar a una mayor sensibilidad al sobreajuste, lo que afectaría negativamente al rendimiento general del modelo.

Kernel sigmoide

El kernel sigmoide, a través de la tangente hiperbólica, comprime la salida al rango (-1,1), indicando el resultado 1 que los vectores son muy similares y -1 que son opuestos. Es similar a una red neuronal con dos capas.

$$K(x,y) = \tanh(\gamma x^T y + c) \tag{5.4}$$

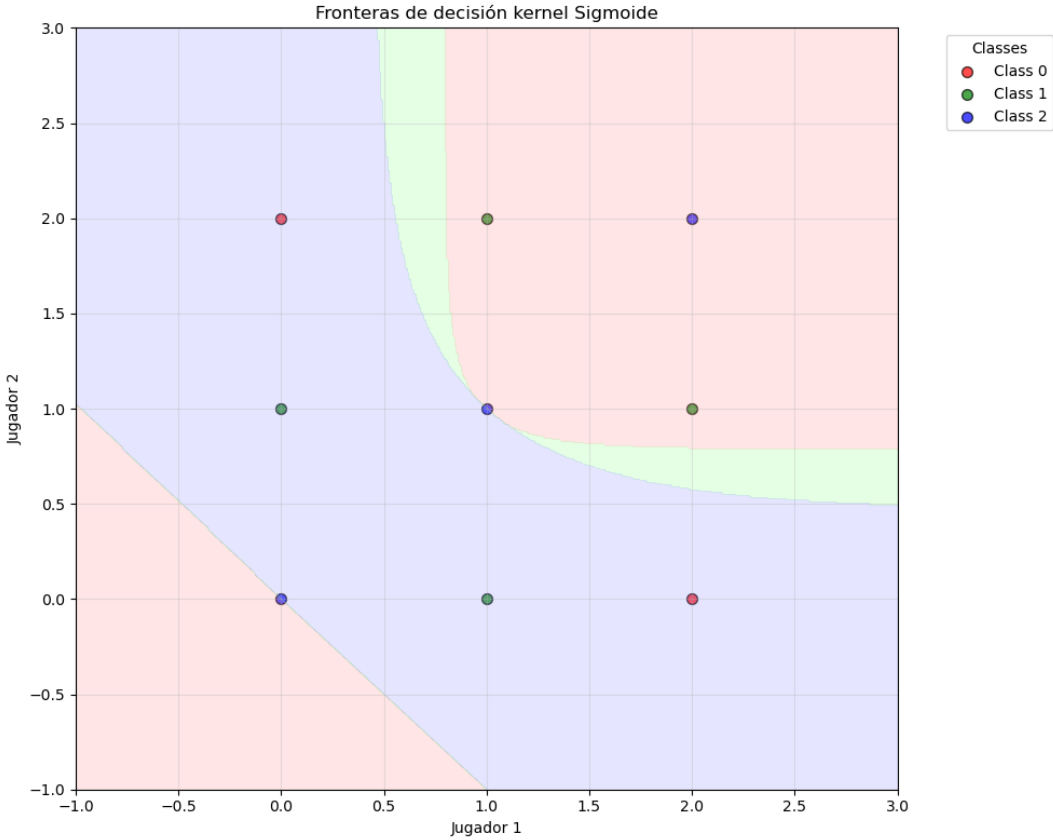

Figura 5.4: Frontera de decisión del algoritmo SVM con kernel sigmoide, parámetros iniciales por defecto y n=10000.

En la Figura 5.4, podemos ver las formas que caracterizan al kernel sigmoide, fronteras prácticamente lineales, y que suelen ser paralelas. No esperamos un buen resultado ya que los áreas de clasificación no concuerdan los nuestros datos.

5.0.1. Resultados

Medias de precisión

En la Tabla 5.1, observamos que el kernel RBF obtiene los mejores resultados en tamaños pequeños y medianos del conjunto de datos, donde alcanza precisiones superiores al 86 %. Sin embargo a partir de los 50.000 ejemplos, su rendimiento cae bruscamente lo que sugiere que los parámetros por defecto no escalan adecuadamente con el volumen de datos y provocan sobreajuste o mala generalización.

El kernel polinómico empieza también con un rendimiento muy alto pero en seguida disminuye, lo que evidencia que, aunque inicialmente puede aproximar bien la frontera de decisión, con más datos se vuelve ineficaz probablemente por exceso de complejidad al utilizar un grado fijo.

El kernel lineal como ya esperábamos mantiene una precisión baja al no poder trazar fronteras lineales entre clases en un problema no lineal.

Por último el sigmoide ofrece el peor rendimiento de todos debido a que genera fronteras de decisión inadecuadas para nuestro problema.

Tamaño	Linear	RBF	Poly	Sigmoid
1000	41.400	86.400	85.900	21.700
5000	32.260	86.020	52.420	6.980
10000	29.860	86.330	47.410	7.100
50000	33.130	43.044	57.180	6.672
100000	33.437	26.202	37.726	27.194

Tabla 5.1: Medias de precisión (%) por kernel (10-fold CV).

Varianzas

En cuanto a la estabilidad de los modelos, observamos en la Tabla 5.2 que la varianza del kernel RBF es baja con tamaños pequeños y medianos, pero a partir de 50.000 ejemplos crece de forma notable lo que confirma su creciente inestabilidad a medida que los parámetros dejan de ser adecuados para el tamaño del dataset.

El kernel polinómico también presenta varianzas muy altas lo que sugiere una alta dependencia de los datos de entrenamiento. Por lo contrario, el kernel lineal presenta varianzas más bajas y decrecientes con el tamaño del dataset.

Por último, el kernel sigmoide también tiene una varianza relativamente baja en la mayoría de casos lo que implica que su comportamiento es consistente y seguramente su mal rendimiento se deba a un mal ajuste estructural a la naturaleza del problema.

Tamaño	Linear	RBF	Poly	Sigmoid
1000	0.266667	10.711111	18.100000	32.011111
5000	40.373778	1.070667	68.101778	0.910667
10000	54.824889	1.149000	104.823222	0.771111
50000	0.228733	200.998916	105.656356	0.088640
100000	0.000023	359.660262	161.270382	71.756804

Tabla 5.2: Varianza de la precisión por kernel.

Tiempos de Ejecución

En cuanto a los tiempos podemos ver en la Tabla 5.3 que el kernel polinómico y el lineal escalan de forma similar así también como el RBF y el sigmoide, pese a que el sigmoide en tamaños pequeños es el más lento.

Tamaño	Linear	RBF	Poly	Sigmoid
1000	0.30	0.15	0.20	0.44
5000	4.63	2.51	2.52	8.03
10000	9.63	9.75	7.26	17.00
50000	48.75	86.57	54.29	83.32
100000	101.84	177.53	110.32	175.23

Tabla 5.3: Tiempos de ejecución (segundos) por kernel.

5.0.2. Mejoras Kernel RBF

Como hemos visto en el apartado anterior, el Kernel RBF es el que *a priori* mejor se ajusta a nuestros datos, pero según aumentamos el número de instancias los parámetros por defecto dejan de ajustar de forma estable, por lo que en este apartado vamos a buscar unos mejores valores para esos parámetros de cara a obtener una buen ajuste a nuestros datos.

El parámetro propio de este kernel, como hemos visto al inicio de este capítulo, es $\gamma = \frac{1}{2\sigma^2}$. A valores más altos de γ las muestras tendrán más influencia por lo que puede tender más al sobreajuste, mientras que para valores bajos hacemos un ajuste más suave tendiendo a la linealidad.

También podemos ajustar el parámetro propio del algoritmo SVM, **C**, el cual marca el equilibrio entre maximizar el margen y minimizar el error de clasificación. Para valores altos de **C** se penalizarán más los errores por tanto tenderemos más al sobreajuste, mientras que para valores bajos de **C** buscaremos un margen amplio permitiendo más errores.

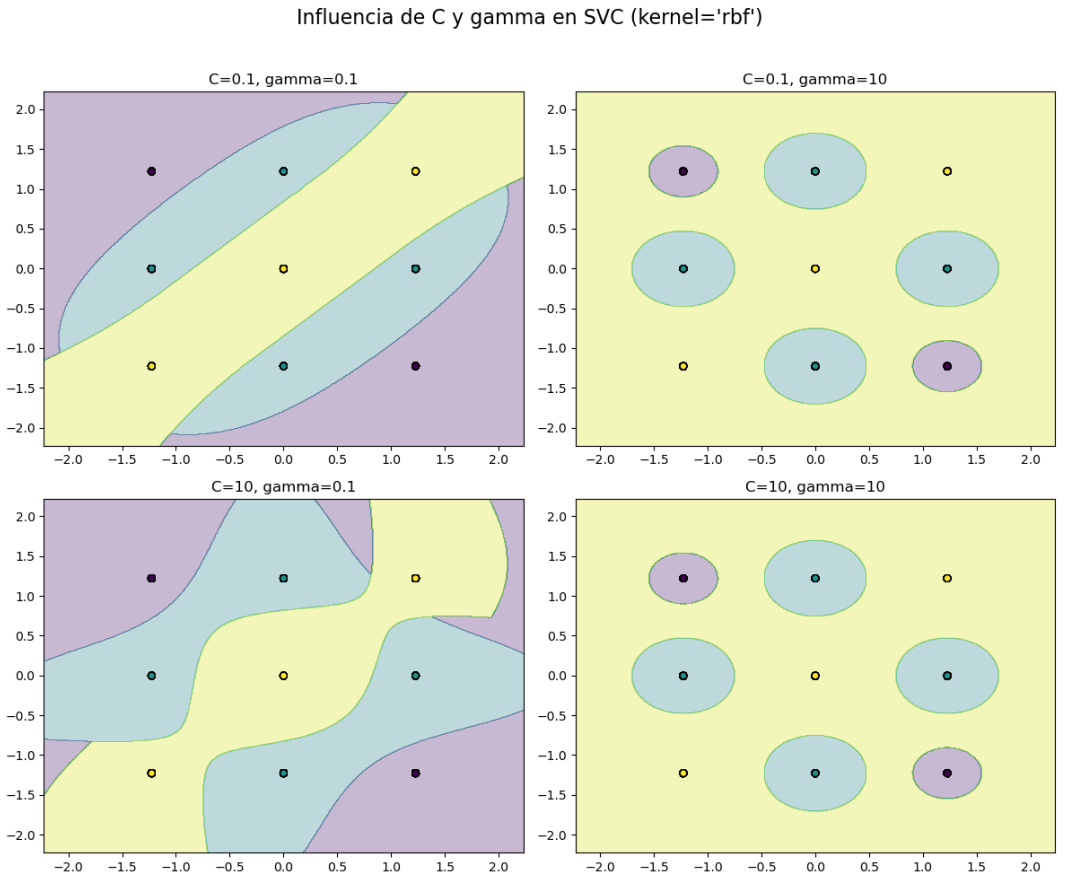

Figura 5.5: Fronteras de decisión del algoritmo RBF en función de sus parámetros.

En la Figura 5.5 se observa que al aumentar γ, las fronteras se ajustan más a los datos, generando prácticamente circunferencias en torno a los puntos clasificados.

En cuanto al parámetro **C** vemos cómo hay menos espacio entre las fronteras de decisión y los puntos mientras que para valores pequeños tenemos áreas más grandes.

En los cuatro casos los resultados obtenidos en términos de precisión media y varianza son óptimos, con una precisión media del 86,7 % y una varianza prácticamente nula. Los parámetros que hemos utilizado anteriormente son los que utiliza la librería Scikit-learn por defecto, **C**=1 y $\gamma = 1/(n\_features * X.var())$, lo cual nos lleva a pensar que dependía de las muestras con las que entrenamos el modelo y al dejar ahora parámetros fijos, como nuestro problema es discreto, siempre converge a la solución óptima con suficientes datos.

Capítulo 6

Redes no lineales

Como se ha mostrado en el Capítulo 4, los modelos no lineales ofrecen un mejor ajuste a nuestros datos, debido a la naturaleza no lineal de los mismos. En este capítulo, profundizaremos en el análisis de estos modelos, explorando cómo se comportan al modificar sus principales parámetros, con el objetivo de comprender mejor su funcionamiento y optimizar su rendimiento.

6.0.1. MLP

El MLP, tal y como hemos visto en el Marco Teórico 2.3, se compone de una o varías capas ocultas entre la capa de entrada y la de salida. En este apartado, analizaremos el impacto que tiene la arquitectura de la red sobre su rendimiento, variando el número de neuronas en la capa oculta y evaluando también el efecto de incrementar el número de capas ocultas.

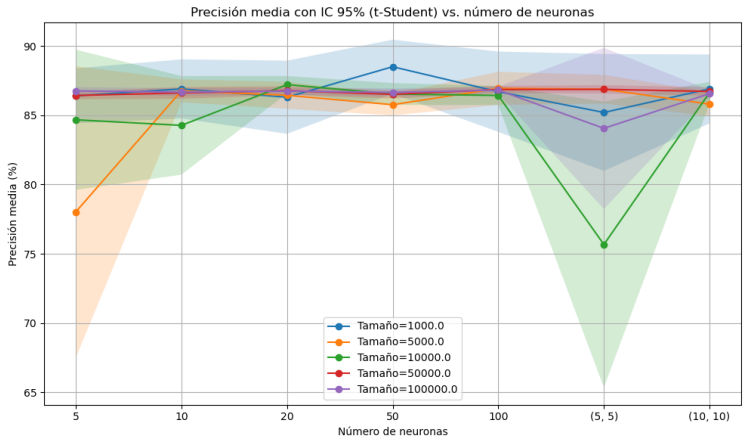

Figura 6.1: Gráfico de bandas de confianza 95 % para la precisión en función del número de neuronas.

Lo más llamativo de los resultados obtenidos en la Figura 6.1 es la variación de la amplitud de las bandas de confianza en función del número de neuronas. Mientras la precisión media podemos decir que es estable, la amplitud de dichas bandas es notablemente mayor en los resultados con 5 neuronas y con dos capas de 5 neuronas.

6.0.2. *k*-Vecinos

En función del número de vecinos que fijemos los resultados pueden ser muy dispares. *A priori* podemos pensar que al coger solo los 2 o 3 vecinos más próximos es más probable que la mayoría sean ruido mientras que si cogemos 15 vecinos es prácticamente imposible que se dé esta situación.

Tamaño	k=**2**	k=**3**	k=**5**	k=**8**	k=**10**	k=**15**	k=**20**
1000	80.300	82.200	85.900	88.900	86.600	84.300	88.000
5000	79.620	79.740	80.380	86.480	86.260	86.780	86.620
10000	71.500	74.090	86.750	87.130	87.100	86.670	87.190
50000	74.830	86.578	85.772	86.612	86.444	86.822	86.710
100000	74.406	84.840	86.748	86.668	86.923	86.568	86.731

Tabla 6.1: Medias de precisión (%) por número de vecinos (k).

Efectivamente, podemos observar en la Tabla 6.1 cómo con 2 y 3 vecinos la precisión media ya es alta pero a partir de 5 vecinos ya nos estabilizamos en torno al 86 % de precisión. Lo vemos gráficamente en la Figura 6.2 en la que, a partir de 10000 datos y de 5 vecinos en adelante, la precisión media se estabiliza en torno al 86 %.

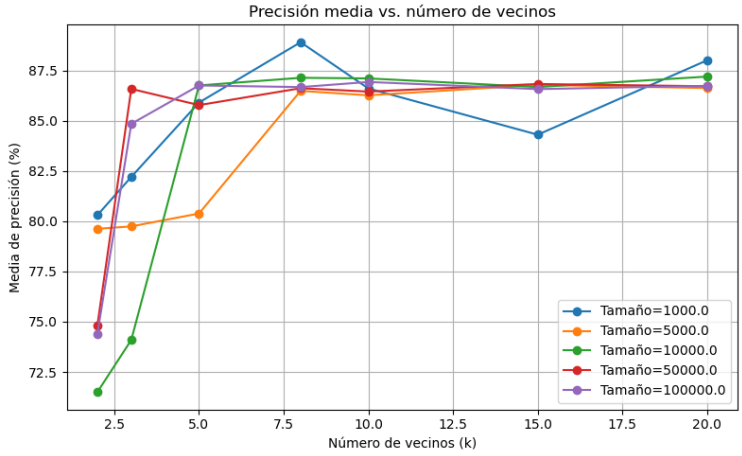

Figura 6.2: Gráfico de la media de precisión en función del número de vecinos.

En cuanto a la varianza, obtenemos resultados parecidos. Vemos en la Tabla 6.2 cómo para tamaños a partir de 10000 datos se empieza a estabilizar a partir de los 8 vecinos, sí que vemos que con 5 vecinos aunque hayamos obtenido antes una buena precisión media no es tan fiable pero a partir de los 8 con suficientes datos se estabiliza prácticamente en el 0.

Tamaño	k**=2**	k**=3**	k**=5**	k**=8**	k**=10**	k**=15**	k**=20**
1000	49.410000	40.160000	7.090000	3.290000	7.040000	19.010000	5.000000
5000	46.403600	20.584400	16.339600	1.609600	2.384400	1.675600	2.547600
10000	26.014000	20.036900	1.814500	0.994100	0.904000	1.954100	0.910900
50000	33.555540	0.179636	6.317296	0.219376	0.066304	0.159396	0.096820
100000	49.101344	29.071760	0.078816	0.074916	0.081921	0.198816	0.040329

Tabla 6.2: Varianza de la precisión por número de vecinos (k).

Gráficamente vemos en la Figura 6.3 los resultados que hemos descrito anteriormente. Nos llama la atención cómo con 1000 muestras la varianza aumenta con el número de vecinos y esto se debe a que con tan pocos datos dependemos mucho de la muestra.

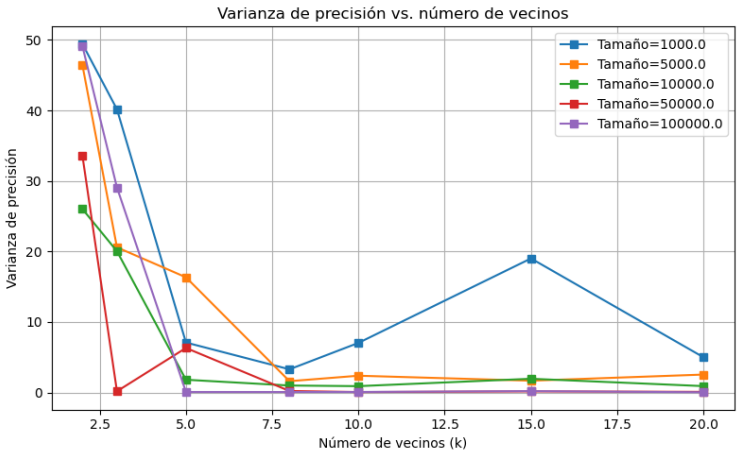

Figura 6.3: Gráfico de la varianza de la precisión en función del número de vecinos.

A continuación vemos las fronteras de decisión obtenidas para la última instancia de la ejecución variando el número de vecinos:

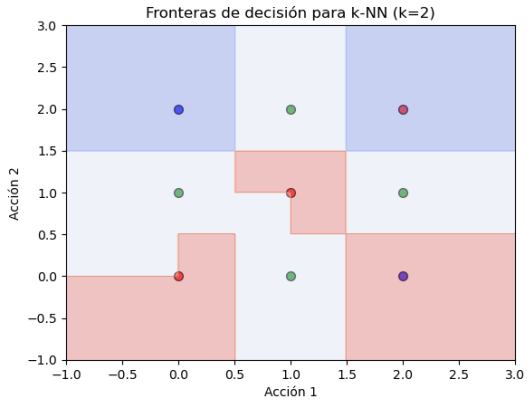

Figura 6.4: Gráfico de fronteras de decisión con k=2.

Las fronteras obtenidas en la Figura 6.4 con solo 2 vecinos no son las correctas. Esto se debe a que con solo dos vecinos si uno de ellos es ruido ya clasificamos mal la instancia. La librería que estamos utilizando por defecto en caso de empate elige la clase más baja.

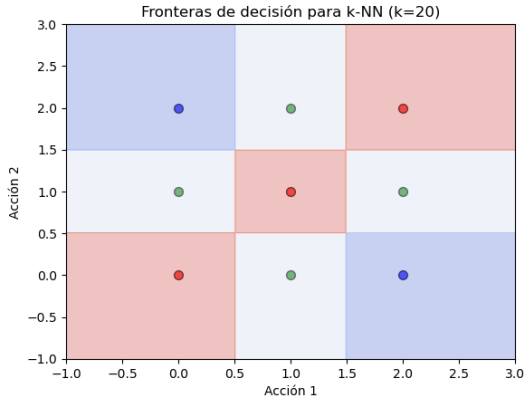

Figura 6.5: Gráfico de fronteras de decisión con k=20.

Como esperábamos según los resultados obtenidos, en la Figura 6.5 se ve cómo las fronteras de decisión con 20 vecinos son las idóneas para nuestro conjunto de datos.

6.0.3. XGBoost

Por último, el algoritmo no lineal que nos queda de los que hemos utilizado en este estudio es el XGBoost. Los parámetros más importantes a tener en cuenta en este algoritmo son *max_depth* (la profundidad de los árboles a entrenar) y *learning_rate* (tasa de aprendizaje).

A continuación vemos dos gráficos, los dos representan las bandas de confianza de la precisión media en función de los parámetros descritos anteriormente pero el primero lo hace con un tamaño de 100 datos y el segundo de 100000.

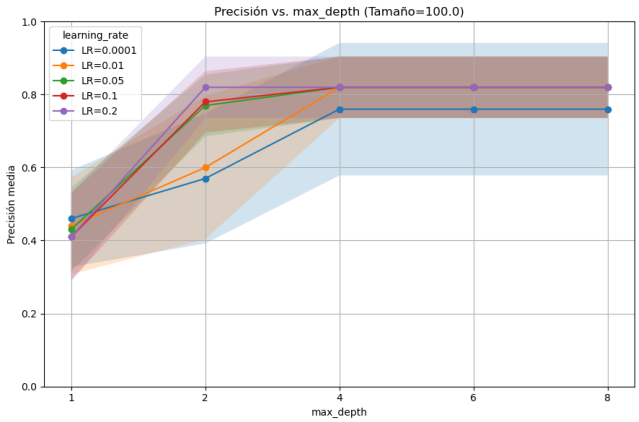

Figura 6.6: Gráfico de bandas de confianza 95 %, dependiendo de los parámetros con n=100.

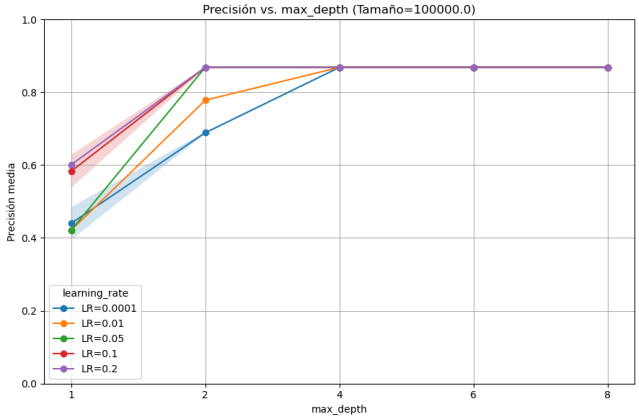

Figura 6.7: Gráfico de bandas de confianza 95 % dependiendo de los parámetros con n=100000.

Podemos observar varias diferencias en función de los distintos parámetros en las figuras 6.6 y 6.7.

La diferencia más clara a simple vista es cómo las anchuras de las bandas de confianza decrecen drásticamente de un tamaño a otro. Y decrece también, aunque no de forma tan notable, según vamos aumentando la máxima profundidad de los árboles.

También vemos cómo la precisión media aumenta con la máxima profundidad. Fijándola en 1 obtenemos prácticamente la mitad de la precisión que se obtiene a partir del valor 3, donde se estabiliza por encima del 80 %.

Por último, se observa que para valores más altos de la tasa de aprendizaje, la convergencia se alcanza antes que para los valores más pequeños, siendo similar la precisión para los valores a partir de 0.05.

Capítulo 7

Conclusiones

Evaluados los distintos algoritmos, podemos confirmar que se cumplen las hipótesis iniciales formuladas tras analizar la Figura 3.2. Una de las conclusiones principales es la importancia de realizar un buen análisis exploratorio inicial, ya que este puede evitar un esfuerzo innecesario en etapas posteriores.

En todos los gráficos de resultados se observa una variación significativa en el rendimiento de los algoritmos en función del tamaño del conjunto de datos, lo que muestra la importancia de disponer de una muestra suficientemente grande para que los modelos aprendan de manera estable la estructura subyacente del problema.

Respecto a la evaluación de los distintos algoritmos, se ha podido comprobar cómo los métodos no lineales se adaptan mejor a nuestro problema, destacando el algoritmo XGBoost que ha obtenido resultados similares a k-Vecinos y al MLP, pero con aproximadamente una cuarta parte del tiempo de ejecución.

Ha sido interesante también observar cómo varían las fronteras de decisión del algoritmo SVM, no solo en función del kernel elegido, sino también de los parámetros configurados, demostrando la flexibilidad de este método según las características del problema.

Por último se ha visto la importancia de realizar una búsqueda adecuada de los parámetros para la inicialización de los algoritmos, observándose mejoras de hasta un 30 % en precisión dependiendo de los valores empleados.

7.1. Líneas de trabajo futuras

Se abren numerosas líneas de investigación desde la perspectiva la Teoría de Juegos y los variados modelos que podemos tratar de evaluar, así como desde el campo del Aprendizaje Automático.

Cada algoritmo presenta múltiples parámetros, cuya influencia y adecuación resulta fundamental estudiar en el contexto de cada modelo de la Teoría de Juegos.

Otra posible extensión sería estudiar la escalabilidad de los algoritmos utilizados al aumentar el número de jugadores, estrategias y clases, manteniendo la estructura y complejidad del problema.

En este trabajo se comparan tamaños totales del conjunto de datos pero otra línea de trabajo a seguir podría ser comparar los resultados obtenidos variando el porcentaje de muestras utilizadas para el entrenamiento

Bibliografía

[1] John von Neumann y Oskar Morgenstern. *Theory of Games and Economic Behavior*. Princeton University Press, 1944.

[2] John F. Nash. "Equilibrium points in *n*-person games". En: *Proceedings of the National Academy of Sciences* 36 (1950), págs. 48-49.

[3] Emilio Cerda Tena, Joaquín Pérez Navarro y José Luis Jimeno Pastor. *Teoría de Juegos*. Pearson Educación, S.A., 2004.

[4] Warren S. McCulloch y Walter Pitts. "A logical calculus of the ideas immanent in nervous activity". En: *The bulletin of mathematical biophysics* 5 (1943), págs. 115-133.

[5] Datademia. *¿Qué es Deep Learning y qué es una red neuronal?* Consultado el 30 de abril de 2025. Jun. de 2021. URL: https://datademia.es/blog/que-es-deep-learning-y-que-es-una-red-neuronal.

[6] Frank Rosenblatt. "The Perceptron: A Probabilistic Model for Information Storage and Organization in the Brain". En: *Psychological Review* 65.6 (1958), págs. 386-408. DOI: 10.1037/h0042519.

[7] DataScientest. *Perceptrón: ¿qué es y para qué sirve?* Consultado el 2 de mayo de 2025. Mar. de 2022. URL: https://datascientest.com/es/perceptron-que-es-y-para-que-sirve.

[8] Juan Sensio. *Perceptrón multicapa y retropropagación*. Consultado el 3 de mayo de 2025. agosto de 2020. URL: https://www.juansensio.com/blog/023_mlp_backprop.

[9] MathWorks. *Support Vector Machines (SVMs)*. Consultado el 6 de mayo de 2025. s.f. URL: https://es.mathworks.com/discovery/support-vector-machine.html.

[10] Cyobero. *K-means Clustering*. Accedido el 6 de mayo de 2025. Feb. de 2018. URL: https://rpubs.com/cyobero/k-means.

[11] IONOS. *Convolutional Neural Network (CNN)*. Accedido el 12 de junio de 2025. Sep. de 2024. URL: https://www.ionos.es/digitalguide/paginas-web/desarrollo-web/convolutional-neural-network/.

Apéndice A

Conceptos

A.0.1. Retropropagación

La retropropagación (backpropagation) es el algoritmo fundamental que permite a una red neuronal aprender a partir de los datos. Su función principal es ajustar los pesos de la red en función del error cometido en cada predicción, de manera que el modelo mejore progresivamente.

Este proceso se basa en la regla de la cadena del cálculo diferencial. En términos sencillos, cuando una red hace una predicción, se calcula la diferencia entre el valor predicho y el valor real (es decir, el error o la función de pérdida). A continuación, ese error se propaga hacia atrás desde la capa de salida hasta las capas anteriores, calculando en cada paso cuánto ha contribuido cada peso al error total.

Con esa información, se actualizan los pesos mediante un algoritmo de optimización con el objetivo de minimizar el error. El proceso se repite muchas veces (épocas), permitiendo a la red ajustarse progresivamente a los datos de entrenamiento.

A.0.2. Capas de convolución

Las capas de convolución son el componente principal de las redes neuronales convolucionales (CNN), un tipo de red especialmente eficaz en el tratamiento de datos con estructura espacial o local, como imágenes, series temporales o señales.

Estas capas aplican pequeños filtros entrenables que se deslizan sobre la entrada, generando mapas de características que permiten extraer patrones locales como bordes, formas o texturas. Como se observa en la Figura A.1, esta información se transmite a través de distintas capas para finalmente clasificar la entrada en función de los patrones detectados.

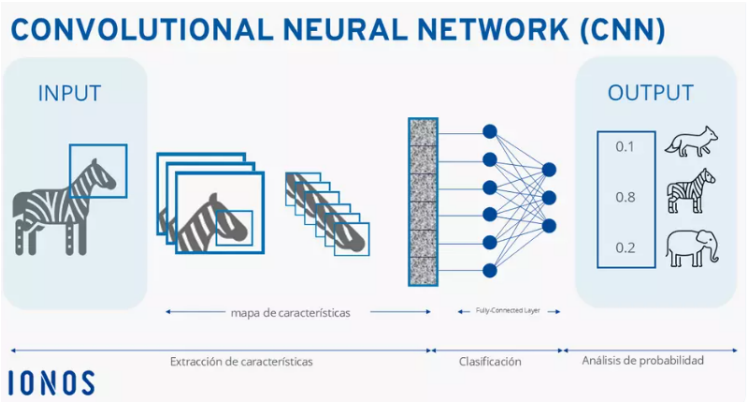

Figura A.1: Funcionamiento de una CNN. *Fuente: IONOS (2024)* [11].

A.0.3. Capas de pooling

Las capas de pooling (o submuestreo) son un componente habitual en las CNN que permite reducir la dimensionalidad de los mapas de características generados por las capas de convolución. Su objetivo principal es simplificar la representación, disminuir el número de parámetros y controlar el sobreajuste, manteniendo al mismo tiempo la información más relevante.

El tipo más común es el max pooling, que selecciona el valor máximo en cada región (normalmente no solapada) del mapa de activación. Esto ayuda a conservar las características más destacadas, reduciendo la resolución espacial de los datos sin perder patrones importantes.

Estas capas también aportan invariancia a pequeñas traslaciones de la entrada, mejorando la capacidad de generalización del modelo.