eman ta zabal zazu
Universidad del País Vasco
Euskal Herriko Unibertsitatea
INFORMATIKA FAKULTATEA
FACULTAD DE INFORMÁTICA

Grado en Ingeniería Informática

Computación

Trabajo de Fin de Grado

# Desarrollo de un modelo interpretable utilizando redes neuronales y SVM

Autor

*Emanuel Dementei*

2023

eman ta zabal zazu
Universidad del País Vasco
Euskal Herriko Unibertsitatea
INFORMATIKA FAKULTATEA
FACULTAD DE INFORMÁTICA

Grado en Ingeniería Informática

Computación

Trabajo de Fin de Grado

# Desarrollo de un modelo interpretable utilizando redes neuronales y SVM

Autor

*Emanuel Dementei*

Director

Jose A. Pascual Saiz, Jesús M. Pérez de la Fuente

# Resumen

Las redes neuronales profundas han demostrado ser herramientas muy eficaces a la hora de aprender patrones en datos como imágenes o audio. Sin embargo, para datos tabulares, es más común utilizar modelos basados en arboles como C4.5 o CART. Estos modelos tienen una propiedad muy importante y es su interpretabilidad. Al ser modelos con estructura de árbol, cualquier persona es capaz de entender como funciona y el razonamiento de este modelo.

En este trabajo de fin de grado se presenta TreeNet y TreeSVM. Son unos algoritmos que generan modelos basados en árboles de decisión donde la construcción de los mismos se hace utilizando redes neuronales o Maquinas de Soporte Vectorial (SVM). De esta forma, aún usando redes neuronales o SVM, se mantiene la propiedad de interpretabilidad.

Se evaluarán los modelos con diferentes conjuntos de datos tabulares para verificar su eficiencia y además compararemos estos modelos con modelos de árboles convencionales y otros. Se compararán también las diferentes métricas relacionadas con interpretabilidad de árboles y se estudiaran las estructuras de estos mismos, estudiando y comparándolas con árboles generados con el modelo CART.

# Índice general

# Índice de figuras

# Índice de tablas

# 1. CAPÍTULO

## Introducción

En los recientes años, se ha avanzado mucho en el campo del aprendizaje automático. Tanto que los nuevos modelos se vuelven tan complejos que pierden su capacidad de interpretabilidad. Esta cualidad, a veces, es imprescindible para abordar algunos problemas de predicción, por ejemplo, en campos como la medicina, el derecho, la banca o cualquier problema que requiera el saber decir por qué se ha tomado una decisión.

Ya existen modelos que son interpretables. Los árboles de decisión se llevan usando desde ya muchas décadas, ya que, a diferencia de modelos como las redes neuronales, estos al ser más simples se pueden interpretar. La idea es que cualquier persona sea capaz de entender cómo el modelo ha llegado a una conclusión sin la necesidad de tener profundos conocimientos sobre aprendizaje automático. En el caso de los árboles, bastaría con seguir la estructura de árbol.

El inconveniente de estos modelos es que no son tan eficaces como los modelos recientes. Las redes neuronales profundas son capaces de aprender patrones muy complejos que los árboles no podrían. Por eso, en este trabajo se presenta un nuevo algoritmo que combina la interpretabilidad de los árboles de decisión con la eficiencia de las redes neuronales. También, como añadido, se propone combinar el algoritmo con máquinas de vectores de soporte (SVM).

En este trabajo de fin de grado se parte del trabajo de Deep Neural Decision Trees (DNDT) [Yang et al., 2018] En él se presenta una estructura de red neuronal la cual, tras entrenarse, consigue el comportamiento de un árbol de decisión. Una vez entrenado el DNDT, las reglas que conforman el árbol se pueden deducir. Aunque el DNDT consigue este objeti-

vo, la escalabilidad del modelo se ve limitada ante el uso de muchas variables. Por eso, el algoritmo que genera el modelo que se propone en este trabajo tiene un enfoque diferente.

La idea es sencilla, la construcción del árbol se hace de forma recursiva al igual que los modelos C4.5 o CART. La diferencia viene a la hora de elegir un punto de corte. Estos puntos de corte se calculan entrenando una red neuronal, un perceptrón en concreto, ya que, este es capaz de dividir un conjunto de datos mediante un hiperplano (Al igual que los SVM). De esta forma entrenando una red neuronal o SVM por cada nodo del árbol, somos capaces de generar un árbol de decisión.

De esta forma, una vez entrenado el modelo se obtiene un árbol de decisión sin ningún tipo de postprocesado como en el DNDT. Además, este algoritmo para generar el modelo no se ve limitado ante el número de variables. A diferencia del DNDT, que tiene un crecimiento exponencial de entrenamiento por el número de variables, el algoritmo que se presenta tiene un crecimiento polinomial. Esto es muy importante en el aprendizaje automático ya que permite buscar patrones en todo el conjunto de datos sin excluir ninguna variable. Esto permite, si es necesario, crear árboles más profundos y complejos, por tanto, más precisos.

Para medir la eficacia del algoritmo, se ha probado con 10 conjuntos de datos que representan diferentes problemas de clasificación. Los resultados se han comparado con modelos clásicos como CART, redes neuronales, SVM y el mencionado anteriormente DNDT y se han comparado las distintas métricas de rendimiento que se han obtenido. Métricas como la tasa de aciertos, la precisión, la sensibilidad y el valor F1. Además de métricas de rendimiento, también se han estudiado otros criterios que miden su capacidad explicativa como el tiempo de construcción o el tamaño de los árboles.

# 2. CAPÍTULO

## Documento de objetivos del proyecto

El objetivo principal de este proyecto es el desarrollo de un modelo de aprendizaje automático que sea interpretable haciendo uso de redes neuronales. Se busca crear un modelo que sea capaz de aprovechar la eficiencia del aprendizaje profundo de las redes neuronales con la interpretabilidad de los árboles de decisión.

Para eso, se realiza primero un estudio de modelos anteriores que tienen el mismo objetivo, como el DNDT, con el objetivo de analizar sus fortalezas y debilidades. Esto nos permitirá identificar oportunidades de mejora o si es necesario tomar un enfoque distinto.

Una vez estudiado el modelo DNDT, se procederá a la implementación de un nuevo algoritmo que genere un modelo diferente. Para eso, se divide el problema en varias fases. En la primera se creará una arquitectura de redes neuronales capaz de dividir un conjunto de datos. La capacidad de dividir un conjunto de datos es indispensable luego para construir un árbol de decisión.

La segunda fase es el desarrollo de un algoritmo recursivo para la construcción de un árbol de decisión. En este caso se hará uso del modelo construido en el paso anterior para dividir el conjunto de datos y de esta forma construir un árbol de decisión. Junto al modelo también es necesaria la creación de un algoritmo que pueda hacer predicciones. Dado que el modelo obtenido es un árbol, este paso es relativamente sencillo ya que solo hace falta recorrer dicho árbol.

Una vez desarrollado el modelo se va a evaluar la efectividad y el rendimiento del mismo. Se llevarán a cabo comparaciones con modelos clásicos basados en árboles como CART, así como otros modelos relevantes como las redes neuronales, SVM y DNDT.

Se utilizarán conjuntos de datos representativos y se aplicarán métricas de evaluación de rendimiento como la tasa de aciertos, la precisión, la sensibilidad y el valor F1.

Los objetivos se podrían resumir de la siguiente forma:

- Estudiar modelos anteriores como el DNDT para saber si se puede realizar alguna mejora o tomar un enfoque distinto.
- Diseñar e implementar una arquitectura de red neuronal capaz de dividir un conjunto de datos.
- Desarrollar un algoritmo de construcción de árboles de decisión recursivo que calcule los puntos de corte mediante la red neuronal junto con su respectivo algoritmo de predicciones.
- Evaluar y comparar el rendimiento del modelo propuesto con modelos clásicos basados en árboles como CART y otros modelos como redes neuronales y DNDT.
- Analizar la interpretabilidad del nuevo modelo y compararla con la de los modelos existentes, mediante métricas específicas de interpretabilidad.

# 3. CAPÍTULO

## Estado del arte

El aprendizaje automático es una rama de la inteligencia artificial, donde se desarrollan técnicas que permiten a las maquinas aprender. En lugar de seguir instrucciones, lo que se hace es dotar a las máquinas de algoritmos para que estas identifiquen patrones en los datos y así tomar decisiones o realizar predicciones basadas en esos patrones. Vea la Figura 3.1.

En el aprendizaje automático existen dos bloques, el aprendizaje supervisado y el no supervisado. Este proyecto está centrado en el aprendizaje supervisado, el cual, se entrena utilizando un conjunto de datos ya etiquetados. Por decirlo de otra forma, se entrena con datos en los que se conocen las respuestas y el objetivo es predecir la respuesta correcta para nuevas entradas.

Entre los algoritmos de aprendizaje automático supervisado tenemos los árboles de decisión, que son modelos de predicción que se utilizan tanto para tareas de clasificación como de regresión. Tienen una estructura de árbol jerárquica, que consta de un nodo raíz, ramas, nodos internos y nodos hoja. Los árboles de decisión se pueden también considerar como una serie de reglas las cuales cada una te lleva a una respuesta distinta. Vea la Figura 3.2.

Estos modelos son muy utilizados por su capacidad de explicativa, ya que, se genera un diagrama fácil de seguir como el siguiente.

La construcción del árbol se hace de forma recursiva en la que se comienza con un nodo raíz, donde tendremos toda la población del conjunto de datos, se selecciona un atributo y se elige la mejor condición para separar este conjunto. Una vez separado el conjunto, se

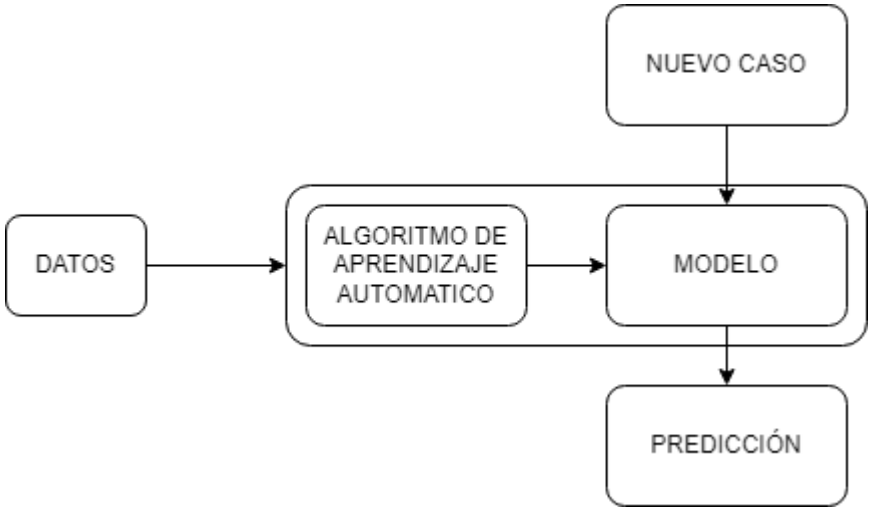


**Figura 3.1:** Diagrama aprendizaje automático

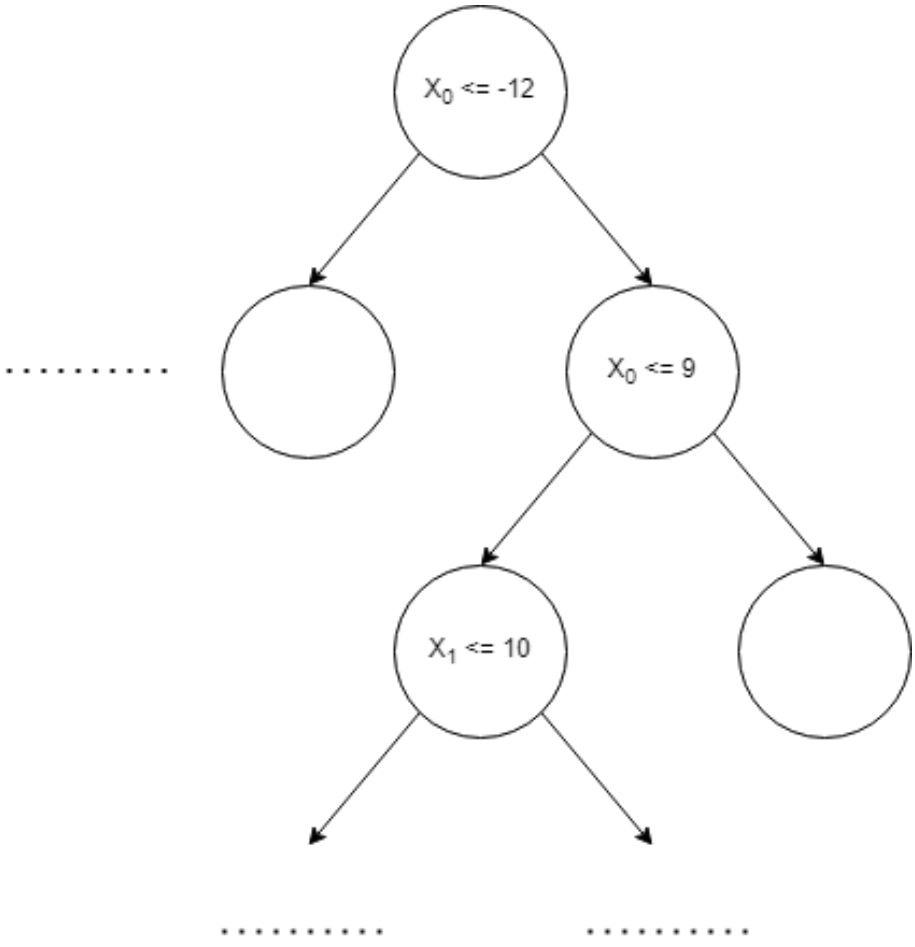


**Figura 3.2:** Árbol de decisión

vuelve a hacer el mismo proceso para los dos subconjuntos que se han creado y así hasta llegar a los nodos hoja, que se crean cuando no se dan más condiciones para dividir la población.

Entre los modelos basados en árboles más conocidos tenemos los siguientes:

- ID3: Este algoritmo aprovecha la entropía y ganancia de información para decidir la mejor condición que divida la población en cada nodo. [Quinlan, 1993]
- C4.5: Este modelo es una versión mejorada de ID3 también desarrollado por Ross Quinlan. Agrega características como el manejo de atributos irrelevantes y el podado del árbol para evitar el sobreajuste de datos. [Quinlan, 1993]
- CART: El término CART es una abreviatura de "Classification And Regresssion Tree" y fue introducido por Leo Breiman. A diferencia de los otros modelos, este utiliza la impureza de Gini para seleccionar el atributo ideal para la división. Esta métrica mide la frecuencia con la que se clasifica incorrectamente un atributo elegido al azar. Lo ideal es elegir la variable con un índice Gini más bajo. [Breiman et al., 1984]

La ganancia de información es una propiedad estadística que mide que tan bien una variable es capaz de separar el conjunto de datos de entrenamiento de acuerdo con su clasificación objetivo. Para ello se utiliza el concepto de entropía, inventado por Shannon, que mide la impureza del conjunto de entrada. La ganancia de información es una disminución de la entropía. Para calcular la ganancia de información, se calcula la diferencia entre la entropía antes de la división y la entropía promedio después de la división del conjunto de datos.

Ganancia de información: Entropia(nodo padre) – [Promedio Entropia(nodos hijo)]

El índice de Gini mide el grado de pureza de un nodo. Nos mide la probabilidad de no sacar dos individuos de la misma clase del nodo. En otras palabras, si seleccionamos dos elementos de una población al azar, entonces deben ser de la misma clase y la probabilidad de eso es 1 si la población es pura. Por tanto, cuanto mayor sea el índice de Gini menor pureza, por lo que se seleccionaría la variable con menor Gini.

Estos modelos son fácilmente interpretables por su estructura de árbol. Pero también se utilizan para crear otro tipo de modelos, por ejemplo, modelos como Random Forest [Breiman, 2001]. En estos modelos se generan múltiples árboles de decisión donde cada uno se entrena con una muestra aleatoria de los datos, por tanto, se generan arboles

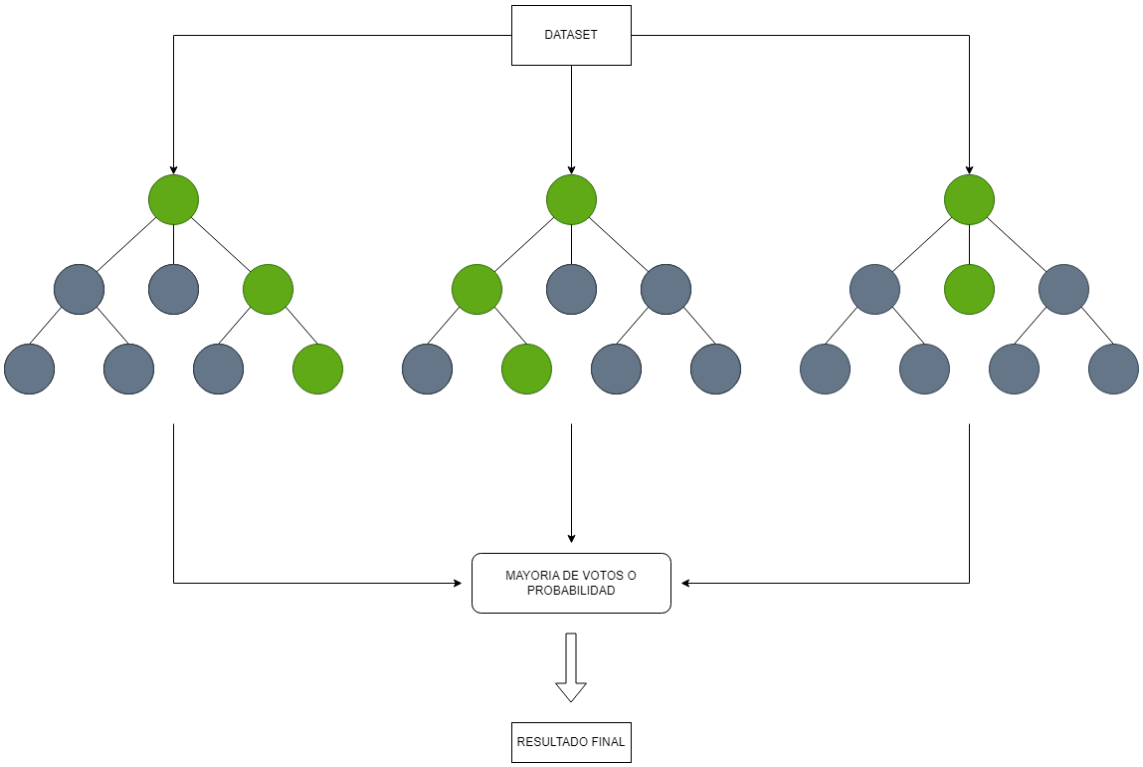


**Figura 3.3:** Random Forest

diferentes. Luego, la predicción final del modelo se obtiene combinando las predicciones individuales de cada árbol, ya sea por votación o promedio de resultados. Vea la Figura 3.3. Estos modelos se utilizan mucho dado que en algunos problemas en concreto dan incluso mejores resultados que las redes neuronales, pero al igual que estas, el modelo pierde su capacidad explicativa.

Las redes neuronales son un modelo matemático que está inspirado por la estructura y/o aspectos funcionales de las redes neuronales biológicas. Vea la Figura 3.4. Una red neuronal consta de un grupo interconectado de neuronas artificiales y procesa información utilizando un enfoque conexionista para la computación. En la mayoría de los casos, una RN es un sistema adaptativo que cambia su estructura basándose en información externa o interna que fluye a través de la red durante la fase de aprendizaje. Las redes neuronales modernas son herramientas de modelado estadístico no lineal de datos. Por lo general, se utilizan para modelar relaciones complejas entre las entradas y salidas o para encontrar patrones en los datos. [D'Addona, 2014]

Las redes neuronales han demostrado ser muy eficientes a la hora de encontrar patrones en los datos y dar unas predicciones muy certeras. Sin embargo, vienen acompañadas de una falta de interpretabilidad por su naturaleza de caja negra donde no puedes

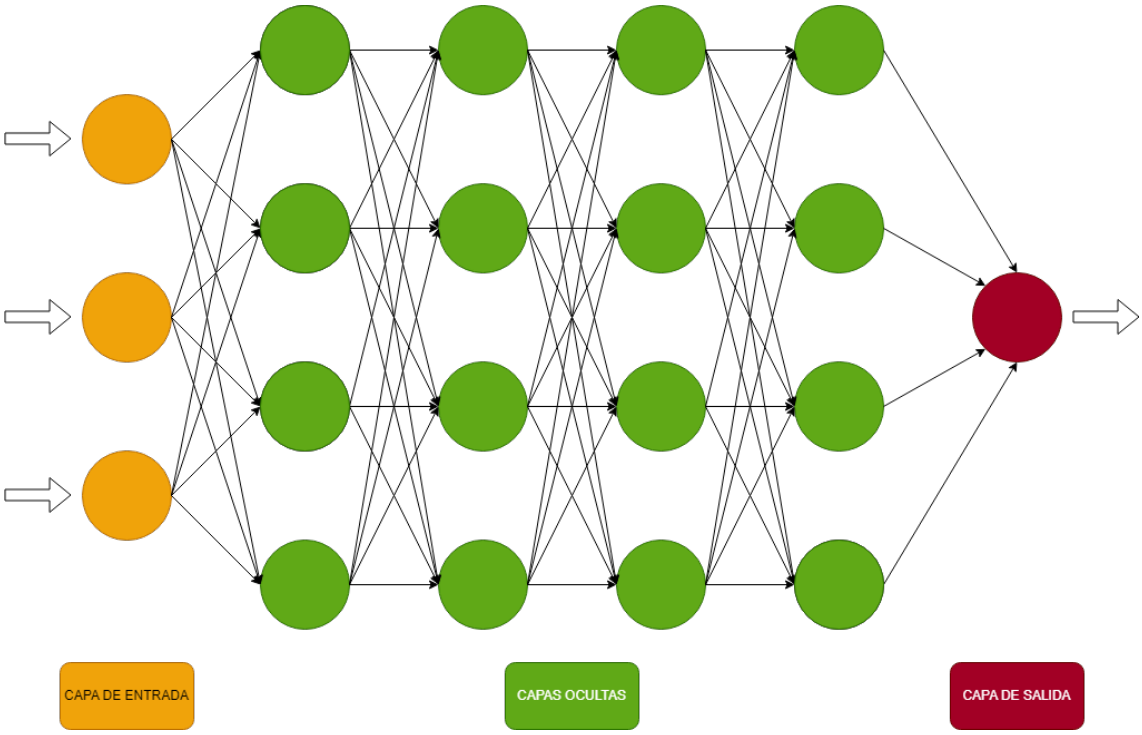


**Figura 3.4:** Red neuronal

encontrar un razonamiento detrás de sus predicciones. Definimos interpretabilidad como la capacidad de explicar o presentar en términos comprensibles para un ser humano [Doshi-Velez and Kim, 2017]. Esto es particularmente importante en aplicaciones donde la ética [Bostrom and Yudkowsky, 2014] y seguridad se tiene en cuenta, por eso es importante buscar modelos que sean transparentes y explicables para verificar la corrección de su proceso de razonamiento o justificar sus decisiones.

Ya existen algunos estudios que proponen unificar las redes neuronales con árboles de decisión. En el trabajo de [Rota Bulo and Kontschieder, 2014] se propuso un modelo llamado Neural Decision Forests (NDF), donde las funciones de división las realizan perceptrones multicapa aleatorizados. Otro modelo del que parte este proyecto es el Deep Neural Decision Trees (DNDT) [Yang et al., 2018] el cual propone una estructura de red neuronal, la cual se limitan algunos parámetros para así obtener como resultado una red neuronal que se comporta como un árbol de decisión. Este modelo es diferente al nuestro ya que en este trabajo se construye un árbol de decisión de la misma forma que los modelos basados en árboles con la diferencia que para la función de división se utiliza una red neuronal.

En este trabajo también se propone combinar las máquinas de vectores de soporte (SVM) de la misma forma que se usan las redes neuronales. El algoritmo SVM busca encontrar un

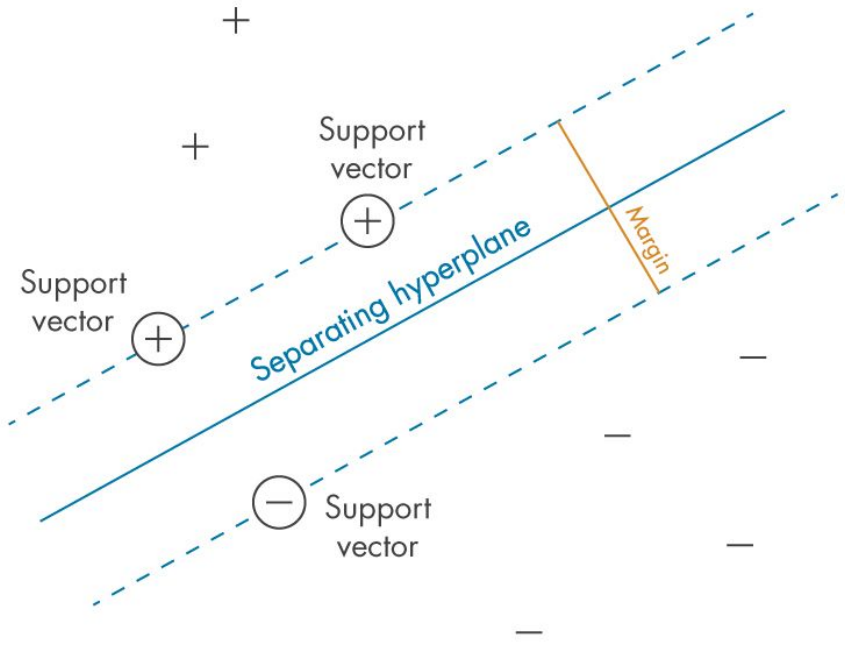


**Figura 3.5:** Support Vector Machine (SVM)
Fuente: https://es.mathworks.com/discovery/support-vector-machine.html

hiperplano que separe de la mejor forma posible dos clases diferentes de puntos de datos. Esto implica un hiperplano con el margen más amplio entre las dos clases, representado por los signos más y menos en la Figura 3.5. El margen se define como la anchura máxima de la región paralela al hiperplano que no tiene puntos de datos interiores. El algoritmo solo puede encontrar este hiperplano en problemas que permiten separación lineal; en la mayoría de los problemas prácticos, el algoritmo maximiza el margen flexible permitiendo un pequeño número de clasificaciones erróneas. De esta forma a la hora de construir el árbol, se puede calcular un hiperplano en cada nodo para dividir la población. En este caso el hiperplano actuaría como punto de corte ya que solo se le alimenta con una variable. Un hiperplano de una sola variable es un punto.

# 4. CAPÍTULO

# Herramientas y librerías utilizadas

Este capítulo está dedicado a describir las bibliotecas y herramientas de *Python* que se han utilizado en el desarrollo de este proyecto. Cada una de estas bibliotecas y herramientas juega un papel crucial en diferentes aspectos del proyecto, desde la manipulación de datos hasta la implementación y evaluación de los modelos de aprendizaje automático. Se discutirá el propósito de cada herramienta y cómo se utilizó en el contexto del proyecto

## 4.1. Python

*Python* es un lenguaje de programación de alto nivel interpretado, es decir, no necesita compilar el código para ejecutarlo. Se trata de un lenguaje de programación multiparadigma, ya que soporta parcialmente la orientación a objetos, programación imperativa y, en menor medida, programación funcional. *Python* también destaca en su sencillez y la legibilidad de su sintaxis.

*Python* ofrece una amplia gama de módulos especializados en el aprendizaje automático y la manipulación de datos lo que hace que *Python* sea la opción preferida hoy en día a la hora de trabajar en este campo. A continuación se presentarán los diferentes módulos que se han utilizado en este trabajo.

## 4.2. Numpy

En primer lugar hablaremos de *NumPy*. *NumPy* es una biblioteca de *Python* que da soporte para crear vectores y matrices multidimensionales grandes, junto con una gran colección de funciones matemáticas de alto nivel para operar con ellas.

La funcionalidad principal de *NumPy* es su estructura de datos "ndarray", para una matriz de n dimensiones. Estas matrices son vistas escalonadas de la memoria, es decir, en lugar de almacenar los datos de una matriz en ubicaciones separadas, *NumPy* coloca todos los elementos de la matriz de forma contigua en la memoria, lo que permite un acceso más rápido y eficiente a los datos.

Además, cuando trabajas con subconjuntos de una matriz en *NumPy* (como cuando seleccionas una columna o una fila específica), *NumPy* no crea una nueva matriz con esos elementos, sino que te da una "vista"de los mismos datos en su ubicación original en la memoria. Esto es más eficiente que copiar los datos a una nueva ubicación cada vez que se selecciona un subconjunto de la matriz.

A diferencia de las listas de *Python*, que son estructuras de datos heterogéneas capaces de almacenar elementos de diferentes tipos de datos, las matrices de *NumPy* son homogéneas: todos los elementos de una matriz deben ser del mismo tipo. Esta homogeneidad permite optimizar el almacenamiento y el acceso a los datos, mejorando el rendimiento para operaciones de gran tamaño.

## 4.3. Pandas

*Pandas* es una libreria especializada en la manipulación y análisis de datos. Ofrece estructuras de datos y operaciones para manipular tablas numéricas y series temporales. *Pandas* utiliza un tipo de dato llamado DataFrame para manipular los datos con una indexación integrada. Ofrece también herramientas para leer y escribir datos en memoria o en el disco. En este trabajo se utiliza esta librería para cargar las bases de datos.

La biblioteca ofrece muchas más funcionalidades donde las mas importantes son las siguientes:

- Alineación de dato y manejo integrado de valores perdidos.
- Reestructuración y segmentación de conjuntos de datos.

- Segmentación vertical basada en etiquetas, indexación elegante, y segmentación horizontal de grandes conjuntos de datos.
- Inserción y eliminación de columnas en estructuras de datos.
- Agrupación predefinida en la biblioteca lo que permite realizar cadenas de operaciones dividir-aplicar-combinar sobre conjuntos de datos.
- Mezcla y unión de datos.
- Indexación jerárquica de ejes para trabajar con datos de altas dimensiones en estructuras de datos de menor dimensión.
- Funcionalidad de series de tiempo: generación de rangos de fechas y conversión de frecuencias, desplazamiento de ventanas estadísticas y de regresiones lineales, desplazamiento de fechas y retrasos.

## 4.4. PyTorch

*PyTorch* es una biblioteca de aprendizaje automático diseñada para realizar cálculos numéricos haciendo uso de la programación de tensores. Un tensor es una generalización de vectores y matrices a un número mayor de dimensiones, y es una estructura de datos clave en muchos campos de la ciencia de datos y la física. Los tensores son utilizados para almacenar los datos de entrada, los pesos y los sesgos de las redes neuronales, entre otros datos. Además da la posibilidad de ejecución en GPU para acelerar los cálculos. *PyTorch* también dispone de una interfaz sencilla para la creación de redes neuronales.

*PyTorch* define una clase Tensor llamada (torch.Tensor) para almacenar y operar con Arrays de números rectangulares, homogéneos y multidimensionales. Los Tensores de PyTorch son similares a los Vectores de NumPy, pero también se pueden usar en una GPU de Nvidia compatible con CUDA.

## 4.5. Scikit-Learn

*Scikit-Learn*, también conocido como *sklearn*, es una biblioteca de aprendizaje automático de código abierto para *Python*. Proporciona una amplia gama de algoritmos de aprendizaje automático. Entre ellos se encuentra los SMV (Support Vector Machines), k-NN

(k-Neares Neighbors), árboles de decisión, regresión lineal y logística. También ofrece herramientas para preprocesar datos, evaluar y ajustar modelos y realizar validación cruzada.

# 5. CAPÍTULO

# Descripción del algoritmo propuesto

En este capítulo, se va a explicar los algoritmos propuestos. Estos son dos algoritmos, TreeNet y TreeSVM, los cuales son muy similares. Por eso, se va a explicar primero TreeNet y después TreeSVM y comentar sus diferencias.

A diferencia de los algoritmos convencionales de construcción de árboles de decisión como por ejemplo CART, que hacen uso de la ganancia de información o gini index para calcular un punto de corte que divida el conjunto de datos, en estos algoritmos la labor de calcular el punto se le delega a una red neuronal o a una SVM.

## 5.1. Árboles de clasificación basados en redes neuronales: TreeNet

El algoritmo TreeNet es un método para la construcción de árboles de decisión. Este algoritmo utiliza redes neuronales para determinar los puntos de corte que dividen el conjunto de datos.

La red neuronal que se utiliza en el algoritmo TreeNet es un perceptrón simple, una red neuronal con una única neurona. Esta red neuronal se entrena con una sola variable a la vez, lo que le permite determinar el mejor punto de corte para esa variable específica. En las figuras 5.1 y 5.2 se pueden ver las fronteras de decisión que es capaz de calcular un perceptrón.

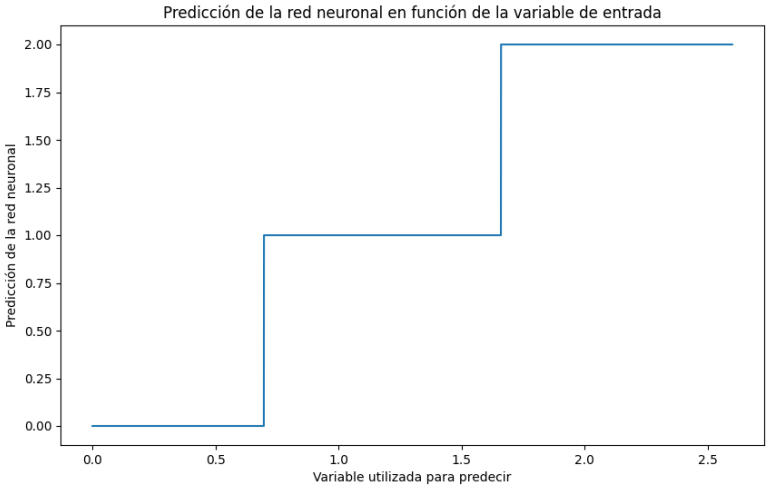


**Figura 5.1:** Puntos de corte aprendidos del conjunto de datos Iris.

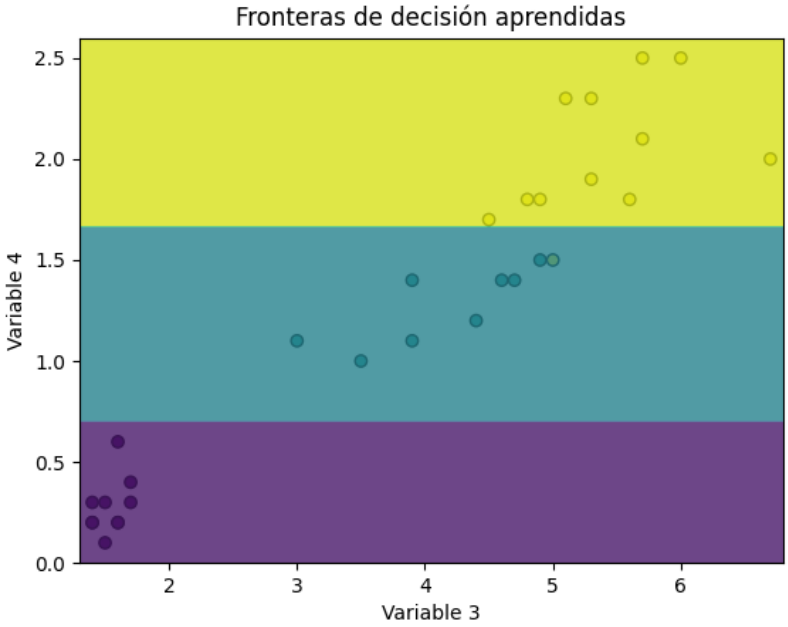


**Figura 5.2:** Fronteras de decisión del conjunto de datos Iris.

La figura 5.1 muestra una gráfica donde en el eje x tenemos la variable predictora y en el eje y su respectiva predicción. Vemos como a medida que recorremos la variable predictora, llega un punto en la que esta cambia y pasa a predecir otro valor. Ese punto es el que consideramos como punto de corte por lo tanto lo guardamos.

Para conseguir esto utilizamos el método de Montecarlo donde creamos un vector con valores desde el mínimo valor que puede coger la variable predictora hasta el máximo valor. Estos valores están uniformemente separados y son utilizados después para encontrar estos puntos de corte. El número de puntos que se utiliza en este método determina luego la precisión del punto de corte. En principio con usar 2000 puntos ha sido suficiente para las pruebas realizadas.

En la figura 5.2 vemos lo mismo que en la primera, pero usamos una variable más para tener una perspectiva en dos dimensiones. Vemos que los puntos de corte calculados en las dos figuras son los mismos, 0.685 y 1.652.

Este enfoque permite a la red neuronal separar las clases del conjunto de datos. Además, si el número de clases es superior a dos, la red neuronal es capaz de aprender múltiples puntos de corte. Tantos como el número de clases a predecir menos una. De momento, el algoritmo solo funciona con variables numéricas, es decir, no funciona con variables categóricas.

### 5.1.1. Construcción del modelo TreeNet

La construcción del modelo TreeNet comienza con la inicialización de la red neuronal, que en este caso es un perceptrón simple con una única neurona. Esta red neuronal se entrena con una sola variable a la vez.

El proceso de entrenamiento de la red neuronal se realiza en varias iteraciones y se lleva a cabo para cada variable en el conjunto de datos. En cada iteración, se selecciona una variable y se entrena la red neuronal con esa variable. Se calcula la pérdida de la red neuronal y se actualizan los pesos de la red neuronal utilizando el optimizador Adam, un algoritmo de descenso de gradiente. Durante el entrenamiento de cada variable, se mantiene un registro de la mejor pérdida, la mejor red neuronal y la mejor variable. Este proceso se repite hasta que todas las variables han sido entrenadas.

Una vez que se ha entrenado las redes neuronales por cada variable, se utiliza la que menor perdida ha dado para determinar los puntos de corte que dividen el conjunto de datos. Estos puntos de corte se calculan utilizando el método de Montecarlo, que genera

un vector de valores uniformemente distribuidos entre el valor mínimo y el valor máximo que puede tomar la variable predictora.

Después de calcular los puntos de corte, se generan nuevos conjuntos de datos basados en las predicciones de la red neuronal. Cada uno de estos nuevos conjuntos de datos contiene solo los casos que fueron predichos como pertenecientes a una determinada clase. Estos nuevos conjuntos de datos se pasan de nuevo a la función de construcción del árbol, lo que resulta en la creación de un nuevo nodo en el árbol para cada conjunto de datos.

Este proceso se repite recursivamente hasta que se alcanza una condición de parada, que puede ser que todos los casos de un conjunto de datos pertenezcan a la misma clase o que se haya alcanzado una profundidad máxima predefinida en el árbol. Cuando se alcanza una condición de parada, se crea un nodo hoja en el árbol que representa la clase de los casos en el conjunto de datos.

Al final del proceso de entrenamiento, la función devuelve una estructura de datos que representa el árbol de decisión construido. Esta estructura de datos contiene información detallada sobre cada nodo en el árbol, incluyendo la variable utilizada para dividir el conjunto de datos en ese nodo, la pérdida asociada con esa división, los puntos de corte determinados por la red neuronal, y los subconjuntos de datos resultantes de la división. Además, para cada nodo que no es una hoja, la estructura de datos también incluye referencias a sus nodos hijos, permitiendo así navegar por todo el árbol.

### 5.1.2. Predicción o clasificación de nuevos casos

Una vez que se ha construido el modelo TreeNet, se puede utilizar para hacer predicciones en nuevos datos. Para predecir se tiene que recorrer el árbol de decisión desde la raíz hasta un nodo hoja, utilizando los puntos de corte en cada nodo para determinar el camino que se debe seguir.

Para cada nodo en el árbol, se toma la variable asociada a ese nodo y se compara su valor con el punto de corte más pequeño. Si el valor que toma la variable es menor que el punto de corte, se sigue el camino hacia el hijo izquierdo del nodo. En caso contrario se prueba lo mismo con el siguiente punto de corte, si el valor asociado a la variable es menor al siguiente punto de corte se toma el camino a la izquierda de ese punto de corte. Así sucesivamente hasta que nos quedemos sin puntos de corte, que en ese caso, tomaríamos el último camino.

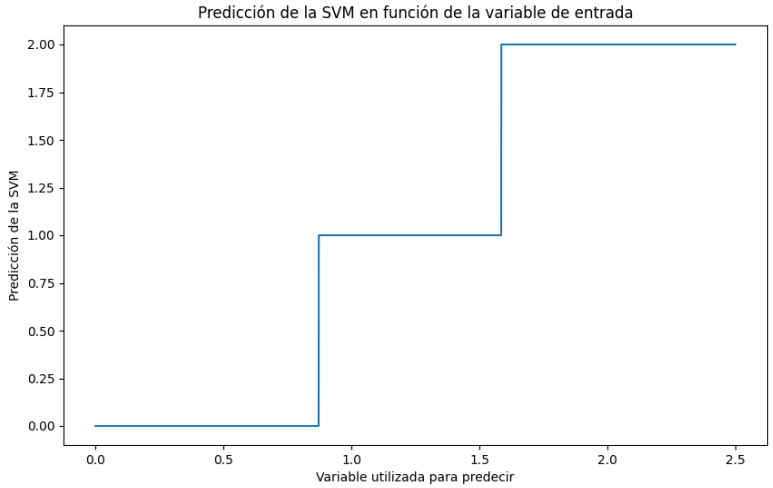


**Figura 5.3:** Puntos de corte aprendidos del conjunto de datos Iris.

Este proceso se repite hasta alcanzar un nodo hoja, donde este contiene la clase que el modelo predice para el caso a predecir.

## 5.2. Árboles de clasificación basados en SVM: TreeSVM

El modelo TreeSVM es un método para la construcción de árboles de decisión. Este modelo utiliza Máquinas de Vectores de Soporte (SVM) para determinar los puntos de corte que dividen el conjunto de datos.

Al igual que en el modelo anterior, la SVM en el modelo TreeSVM se entrena con una sola variable a la vez, lo que le permite determinar los mejores puntos de corte para esa variable. Al igual que el perceptrón de TreeNet, SVM puede calcular más de un punto de corte por cada variable. Si el número de clases es mayor que 2, por supuesto.

En las figuras 5.3 y 5.4 se pueden ver las fronteras de decisión que es capaz de calcular una SVM para el conjunto de datos Iris. Estos gráficos demuestran la capacidad de una SVM para separar el conjunto de datos en diferentes clases. Esta capacidad de separación es justamente lo que se necesita en el algoritmo de construcción de árboles, ya que permite dividir el conjunto de datos en subconjuntos más pequeños. Estos subconjuntos luego se utilizan para construir los diferentes nodos del árbol.

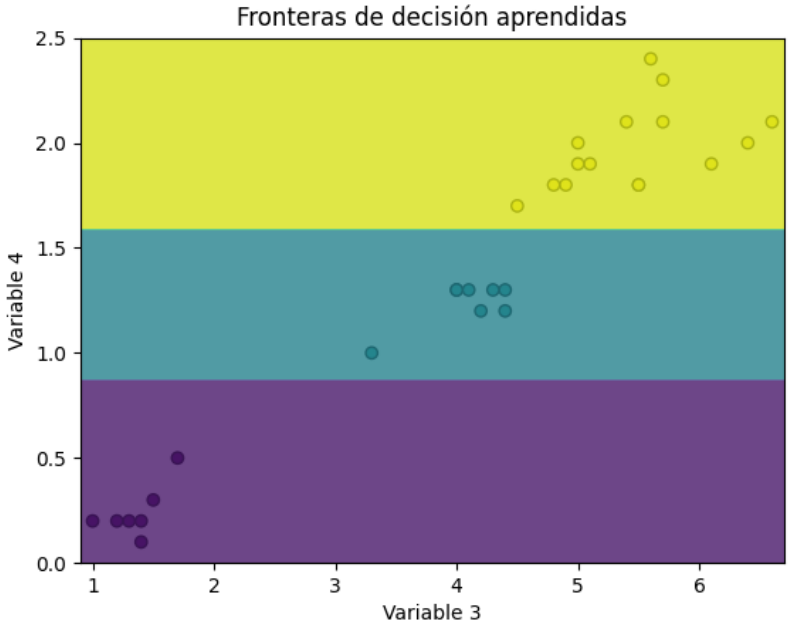


**Figura 5.4:** Fronteras de decisión del conjunto de datos Iris.

### 5.2.1. Construcción del modelo

La construcción del modelo TreeSVM es similar a la del modelo TreeNet, pero en lugar de utilizar una red neuronal para determinar los puntos de corte, utiliza una Máquina de Vectores de Soporte (SVM).

El proceso de construcción comienza iterando sobre todas las variables del conjunto de datos. Para cada variable, se crea un nuevo modelo SVM lineal y se entrena con esa variable. La precisión de cada modelo se calcula y se mantiene un registro de la variable que proporciona la mayor precisión. Este proceso se repite hasta que todas las variables han sido entrenadas.

Una vez que se ha entrenado la SVM, se utiliza para determinar los puntos de corte que dividen el conjunto de datos. Al igual que en el modelo TreeNet, estos puntos de corte se calculan utilizando el método de Montecarlo.

Después de calcular los puntos de corte, se generan nuevos conjuntos de datos basados en las predicciones de la SVM. Cada uno de estos nuevos conjuntos de datos contiene solo los casos que fueron predichos como pertenecientes a una determinada clase. Estos nuevos conjuntos de datos se pasan de nuevo a la función de construcción del árbol, lo que resulta en la creación de un nuevo nodo en el árbol para cada conjunto de datos.

Este proceso se repite recursivamente hasta que se alcanza una condición de parada, que puede ser que todos los casos de un conjunto de datos pertenezcan a la misma clase o que se haya alcanzado una profundidad máxima predefinida en el árbol. Cuando se alcanza una condición de parada, se crea un nodo hoja en el árbol que representa la clase de los casos del conjunto de datos y esta clase será la clase mayoritaria del conjunto de datos.

Al final del proceso de entrenamiento, igual que en TreeNet, la función devuelve una estructura de datos que representa el árbol de decisión construido. Esta estructura de datos contiene información detallada sobre cada nodo en el árbol, incluyendo la variable utilizada para dividir el conjunto de datos en ese nodo, la precisión asociada con esa división, los puntos de corte determinados por la SVM, y los subconjuntos de datos resultantes de la división. Además, para cada nodo que no es una hoja, la estructura de datos también incluye referencias a sus nodos hijos, permitiendo así navegar por todo el árbol.

### 5.2.2. Predicción o clasificación de nuevos casos

Una vez que se ha construido el modelo TreeSVM, se puede utilizar para hacer predicciones en nuevos datos. Al igual que en el modelo TreeNet, la predicción implica recorrer el árbol de decisión desde la raíz hasta un nodo hoja, utilizando los puntos de corte en cada nodo para determinar el camino que se debe seguir.

Para cada nodo en el árbol, se toma la variable asociada a ese nodo y se compara su valor con el punto de corte más pequeño. Si el valor que toma la variable es menor que el punto de corte, se sigue el camino hacia el hijo izquierdo del nodo. En caso contrario, se prueba lo mismo con el siguiente punto de corte. Si el valor asociado a la variable es menor al siguiente punto de corte, se toma el camino a la izquierda de ese punto de corte. Este proceso se repite hasta que nos quedamos sin puntos de corte, en cuyo caso, se tomaría el último camino.

Este proceso se repite hasta alcanzar un nodo hoja, que contiene la clase que el modelo predice para el caso a predecir. De esta manera, el modelo TreeSVM puede clasificar nuevos casos basándose en la estructura del árbol de decisión que ha aprendido durante el entrenamiento.

# 6. CAPÍTULO

# Implementación de los algoritmos

Este capítulo se centra en la implementación práctica de los algoritmos discutidos en este trabajo. Detallamos la construcción de dos modelos clave: TreeNet y TreeSVM, abordando las librerías utilizadas y ofreciendo ejemplos de su aplicación.

## 6.1. Implementación de TreeNet

En esta subsección, nos centramos en la implementación del algoritmo TreeNet. Describiremos las librerías fundamentales que han sido empleadas, la estructura principal del clasificador, y finalmente, ilustraremos su aplicación con un ejemplo práctico.

### 6.1.1. Librerías

Para implementar el modelo TreeNet primero se importan las librerías necesarias.

```
1 import numpy as np
2 import torch
3 import torch.nn as nn
4 import torch.optim as optim
```

Estas son *Numpy* y *Torch* donde también se importan dos submodulos de *Torch* los cuales son *torch.nn*, que proporciona clases para construir redes neuronales, y *torch.optim*, que

implementa varios algoritmos de optimización que se utilizan para entrenar redes neuronales. Estos algoritmos incluyen SGD, Adam, RTMSProp, etc.

### 6.1.2. Perceptrón

A continuación se define la red neuronal.

```
class Net(nn.Module):
    def __init__(self, num_classes):
        super(Net, self).__init__()
        self.fc = nn.Linear(1, num_classes)

    def forward(self, x):
        x = self.fc(x)
        return x
```

La clase `Net` es una subclase de `nn.Module`, que es la clase base para todos los módulos de red neuronal en PyTorch. Esta clase `Net` define una red neuronal muy simple con una sola capa lineal. El perceptrón del que hemos hablado antes.

`__init__(self, num_classes)`: Este es el constructor de la clase `Net`. Aquí, se inicializa la superclase `nn.Module` utilizando el método `super()`. Luego, se define una capa lineal `self.fc` con una entrada y `num_classes` salidas. `num_classes` es el número de clases en el conjunto de datos, que se pasa como argumento al constructor.

`forward(self, x)`: Este método define la pasada hacia adelante de la red. Toma un tensor de entrada `x`, lo pasa a través de la capa lineal `self.fc`, y devuelve el resultado.

### 6.1.3. TreeNetClassifier

La clase TreeNetClassifier es la implementación de nuestro modelo TreeNet. Esta clase tiene varios atributos y métodos que se utilizan para entrenar el modelo y hacer predicciones. Empezamos explicando la función constructora:

```
class TreeNetClassifier():

    def __init__(self, num_features, num_classes, epoch, depth, stop_condition):
        self.num_features = num_features
        self.num_classes = num_classes
        self.epoch = epoch
        self.depth = depth
        self.stop_condition = stop_condition
        self.root = None

        self.n_nodos = None
        self.n_hojas = None
        self.profundidad = None
        self.longitudMediaRamas = None
```

La función `__init__` es el constructor de la clase TreeNetClassifier. Esta función se llama automáticamente cuando se crea una nueva instancia de la clase. Los parámetros de esta función son:

- `num_features`: El número de variables en los datos de entrada. Este valor se utiliza para determinar el tamaño de la entrada de la red neuronal.
- `num_classes`: El número de clases en los datos de salida. Este valor se utiliza para determinar el tamaño de la salida de la red neuronal.
- `epoch`: El número de épocas para entrenar la red neuronal.
- `depth`: La profundidad máxima del árbol de decisión.
- `stop_condition`: La condición de parada para el entrenamiento de la red neuronal. Si la diferencia entre la pérdida actual y la pérdida anterior es menor que esta condición de parada, el entrenamiento se detiene.

Además, se inicializan varios otros atributos a None. Estos atributos se utilizan para almacenar información sobre el árbol de decisión después de que se ha entrenado. Estos incluyen `root` (la raíz del árbol de decisión), `n_nodos` (el número de nodos en el árbol),

`n_hojas` (el número de hojas en el árbol), `profundidad` (la profundidad del árbol), y `longitudMediaRamas` (la longitud media de las ramas del árbol).

La función `fit` es el método que se utiliza para entrenar el modelo.

```
def fit(self, X, y):
    self.root = self.tree_net(X, y, self.epoch, self.num_classes,
                               self.num_features, len(y),
                               self.stop_condition, self.depth)
    self.n_nodos = self.num_nodos(self.root)
    self.n_hojas = self.num_hojas(self.root)
    self.profundidad = self.branch_length(self.root)
    self.longitudMediaRamas = self.averageBranchesLength(self.root)
    return self
```

La función `fit` toma dos argumentos: `X` y `y`. Donde `X` es una matriz de los datos de entrada y `y` es un vector con sus correspondientes etiquetas.

En primer lugar, se llama a la función `tree_net` para entrenar el modelo y construir el árbol de decisión. Los parámetros de esta función son los datos de entrada y salida, el número de épocas, el número de clases, el número de características, la longitud de los datos de salida, la condición de parada y la profundidad máxima del árbol. El resultado de esta función se almacena en el atributo `root`.

Después de entrenar el modelo, se calculan varias métricas sobre el árbol de decisión. Estas incluyen el número de nodos (n_nodos), el número de hojas (n_hojas), la profundidad del árbol (profundidad) y la longitud media de las ramas del árbol (longitudMediaRamas). Estas métricas se calculan utilizando las funciones `num_nodos`, `num_hojas`, `branch_length` y `averageBranchesLength`, respectivamente.

La función `tree_net` es una parte crucial de la clase `TreeNetClassifier`. Esta función es responsable de construir el árbol de decisión que se utiliza para clasificar las muestras.

A continuación se explicará paso a paso la función `tree_net`. El código completo está disponible en el Apéndice A.

La función `tree_net` toma varios argumentos:

- `X`: Los datos de las muestras de entrenamiento.
- `y`: Las etiquetas de las muestras de entrenamiento.
- `epoch`: El número de épocas para el entrenamiento de la red neuronal.
- `num_classes`: El número de clases en el conjunto de datos.
- `num_variables`: El número de características en el conjunto de datos.
- `longitud_inicial`: La longitud inicial del conjunto de datos.
- `stop_condition`: La condición de parada para el entrenamiento de la red neuronal.
- `depth`: La profundidad máxima del árbol de decisión.

La función comienza verificando si la profundidad actual del árbol ha alcanzado el límite máximo (depth == 0). Si es así, devuelve la clase más frecuente en las etiquetas y como un nodo hoja.

```
1 if depth == 0:
2     return np.bincount(y).argmax()
```

Luego, se inicializa algunas variables para almacenar la mejor pérdida (mejorLoss), la mejor red neuronal (mejorNet) y la mejor variable (mejorVar) encontrada hasta ahora.

```
1     mejorLoss = float("Inf")
2     mejorNet = None
3     mejorVar = 0
```

Se itera sobre todas las variables en X. Para cada variable, entrena una red neuronal (Net) para predecir las etiquetas y basándose solo en esa variable. La red se entrena durante un número específico de épocas, y el entrenamiento se detiene si la pérdida no mejora significativamente (según la stop_condition).

```
        for i in range(num_variables):
        Xaux = X[:, i:i + 1]

        net = Net(num_classes)
        criterion = nn.CrossEntropyLoss()
        optimizer = optim.Adam(net.parameters())

        lossAux = float("Inf")
        for _ in range(epoch * int(longitud_inicial / len(y))):
            optimizer.zero_grad()
            outputs = net(torch.tensor(Xaux, dtype=torch.float32))
            loss = criterion(outputs, torch.tensor(y, dtype=torch.long))
            loss.backward()
            optimizer.step()

            if abs(loss.item() - lossAux) < stop_condition:
                break
            lossAux = loss.item()

            if _ % 100 == 0:
                print(f"Epoch {_}: Loss={loss.item():.4f}")

        if loss.item() < mejorLoss:
            tree["Variable"], tree["Loss"] = i, loss.item()
            mejorVar = i
            mejorNet = net
            mejorLoss = loss.item()
```

Si la pérdida de la red entrenada es menor que la mejorLoss actual, la función actualiza mejorLoss, mejorNet y mejorVar con la pérdida, la red y la variable actuales, respectivamente. De esta forma, cuando se termine de entrenar con cada una de las variables, nos quedaremos con la variable y la red neuronal que mejor resultados nos ha dado.

```
    if loss.item() < mejorLoss:
        tree["Variable"], tree["Loss"] = i, loss.item()
        mejorVar = i
        mejorNet = net
        mejorLoss = loss.item()
```

Una vez que la función ha iterado sobre todas las variables, utilizamos el método de Montecarlo para encontrar los puntos de corte que ha calculado el perceptrón. Para eso, se siguen los siquientes pasos:

- `x_values = np.linspace(X.min() - 1, X.max() + 1, 2000).reshape(-1, 1)`: Aquí, estamos generando 2000 puntos de datos uniformemente espaciados en

el rango de los valores de X (extendido por 1 en ambos extremos). Estos puntos de datos se utilizarán para hacer predicciones utilizando la red neuronal entrenada.

- `y_pred = mejorNet(torch.tensor(x_values, dtype=torch.float32)) .argmax(axis=1).numpy()`: Aquí, estamos utilizando la red neuronal entrenada (`mejorNet`) para hacer predicciones en los puntos de datos generados en el paso anterior. Las predicciones son las clases que la red neuronal predice para cada punto de datos.

- `c = y_pred[0]`: Aquí, estamos inicializando una variable `c` con la primera predicción.

- `Cutpoints = []` y `Valores = [c]`: Aquí, estamos inicializando una lista vacía `Cutpoints` para almacenar los puntos de corte y una lista `Valores` para almacenar los valores de las predicciones (Las etiquetas que predice la red neuronal).

- En el bucle for, recorremos las predicciones. Si la predicción actual es diferente de `c`, actualizamos `c` con la predicción actual, y agregamos el valor de `x` correspondiente a la predicción actual a `Cutpoints` y la predicción actual a `Valores`.

- Finalmente, almacenamos `Cutpoints` y `Valores` en el diccionario `tree`. Estos representan los puntos de corte y los valores de las predicciones en esos puntos de corte, respectivamente.

```
x_values = np.linspace(X.min() - 1, X.max() + 1, 2000).reshape(-1, 1)

y_pred = mejorNet(torch.tensor(x_values,
                               dtype=torch.float32)).argmax(axis=1).numpy()

c = y_pred[0]
Cutpoints = []
Valores = [c]
for i, prediccion in enumerate(y_pred):
    if prediccion != c:
        c = prediccion
        Cutpoints.append(x_values[i].item())
        Valores.append(c)

tree["Cutpoints"] = Cutpoints
tree["Valores"] = Valores
```

A continuación se crean dos diccionarios, `pobx` y `poby`, que representan nuevos conjuntos de datos. `pobx` guarda los datos y `poby` las etiquetas. Se recorre todo el conjunto de

datos de X y se van añadiendo a su correspondiente conjunto según lo que predice la red neuronal. De esta forma, se generan nuevos conjuntos de datos que luego se pasarán por la misma función.

```
pobx = {}
poby = {}
for i in enumerate(X[:, mejorVar:mejorVar + 1]):
    pre = mejorNet(torch.tensor(i[1], dtype=torch.float32)).argmax().item()
    if pre not in poby:
        pobx[pre], poby[pre] = [], []
    pobx[pre].append(X[i[0]])
    poby[pre].append(y[i[0]])
```

En este fragmento de código, se está creando la estructura del árbol de decisión. Primero, se calcula el número de hijos o subconjuntos de datos que se han creado en el paso anterior. Luego, se recorren estos subconjuntos.

```
num_hijos = len(poby)
tree["Hijos"] = {}
for key, value in poby.items():
    if len(np.unique(np.array(value))) == 1 or num_hijos == 1:
        tree["Hijos"][key] = np.bincount(value).argmax()
    else:
        tree["Hijos"][key] = self.tree_net(
            np.array(pobx[key]).reshape((len(pobx[key]), num_variables)),
            np.array(value),
            epoch,
            num_classes,
            num_variables,
            longitud_inicial,
            stop_condition,
            depth - 1)
```

Para cada subconjunto, se comprueba si todas las etiquetas son iguales (es decir, si todos los elementos del subconjunto pertenecen a la misma clase) o si sólo hay un subconjunto. Si alguna de estas condiciones se cumple, se considera que se ha alcanzado un nodo hoja del árbol, y se asigna la clase más común de las etiquetas del subconjunto a este nodo hoja.

Si no se cumple ninguna de estas condiciones, significa que aún hay más de una clase presente en el subconjunto y que hay más de un subconjunto. En este caso, se realiza una llamada recursiva a la función tree_net con el subconjunto actual. Esto creará un nuevo nivel en el árbol de decisión, donde cada nodo representa una decisión basada en la red neuronal entrenada.

La recursividad continúa hasta que se alcanzan las condiciones de los nodos hoja, creando así un árbol de decisión completo. La profundidad del árbol está limitada por el parámetro depth que se pasa a la función tree_net.

Finalmente, la función tree_net devuelve el diccionario tree. Este diccionario representa la estructura completa del árbol de decisión que se ha generado. Cada nodo del árbol se representa como una entrada en el diccionario, que contiene información sobre la variable que se utiliza para la decisión en ese nodo, los puntos de corte que se utilizan para dividir los datos, los valores que se predicen en cada región definida por los puntos de corte, y los subárboles (o clases, en el caso de los nodos hoja) que se derivan de cada decisión.

```
    return tree
```

Una vez construido el árbol, necesitamos una función que haga uso de él para hacer predicciones.

```
    def predict(self, X):

    predictions = []
    for sample in X:
        label = self.traverse_tree(sample.reshape(1, self.num_features), self.root)
        predictions.append(label)

    return predictions
```

`predict(self, X)`: Esta función se utiliza para hacer predicciones en un conjunto de datos `X`. Para cada caso en `X`, se llama a la función `traverse_tree` para obtener la etiqueta predicha, que se agrega a la lista de predicciones. Finalmente, se devuelve la lista de predicciones.

```
    def traverse_tree(self, X, node):
    if isinstance(node, np.int64):
        return node
    else:
        var = node["Variable"]
        xaux = X[:, var:var + 1]
        xaux = xaux[(0, 0)]

        ind = self.find_index_to_insert(node['Cutpoints'], xaux)

        pre = node["Valores"][ind]

        return self.traverse_tree(X, node["Hijos"][pre])
```

`traverse_tree(self, X, node)`: Esta función se utiliza para recorrer el árbol de decisión y obtener la etiqueta predicha para una muestra X. Comienza en el nodo proporcionado (inicialmente, este será el nodo raíz del árbol) y verifica si el nodo es una hoja del árbol (es decir, si es un número entero, que representa una etiqueta de clase). Si es así, devuelve el nodo (la etiqueta de clase). Si no, selecciona la variable correspondiente de la muestra X, encuentra el índice para insertar el valor de la variable en la lista de puntos de corte del nodo (usando find_index_to_insert), y luego recorre el árbol a partir del hijo correspondiente del nodo.

```
    def find_index_to_insert(self, arr, new_int):
    left = 0
    right = len(arr) - 1

    while left <= right:
        mid = (left + right) // 2

        if new_int < arr[mid]:
            right = mid - 1
        elif new_int > arr[mid]:
            left = mid + 1
        else:
            return mid
    return left
```

`find_index_to_insert(self, arr, new_int)`: Esta función es una función auxiliar que normalmente se utiliza para encontrar el índice en el que se insertaría un nuevo entero `new_int` en un array `arr` ordenado de forma ascendente, manteniendo el orden del array. En nuestro caso lo utilizamos para saber a qué etiqueta pertenece un caso en particular. Dada la lista de puntos de corte, y el valor de una variable del caso, nos dice donde debería colocarse este valor entre los puntos de corte.

## 6.2. Ejemplo de uso de TreeNet

Este código es un ejemplo de cómo usar el modelo `TreeNetClassifier` para clasificar el conjunto de datos Iris, que es un conjunto de datos multiclase estándar en aprendizaje automático.

```
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
import numpy as np
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score
from DecisionTreeRN import TreeNetClassifier

# Cargamos el dataset
dataset = load_iris()

# Separamos el dataset en datos (X) y etiquetas (y)
X = dataset.data
y = dataset.target

num_features = X.shape[1]
num_classes = np.unique(y).shape[0]

#Entrenamiento por nivel
epoch = 30000

#Cuantos niveles maximos
depth = 12

#Condición de parada. Si la diferencia entre 2 iteraciones es menor a este valor
#deja de entrenar
stop_condition = 10**(-7)

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2)

model = TreeNetClassifier.TreeNetClassifier(num_features=num_features,
                                            num_classes=num_classes,
                                            stop_condition=stop_condition,
                                            depth=depth,
                                            epoch=epoch)

model.fit(X_train, y_train)

# HAcemos predicciones sobre el test.
y_pred = model.predict(X_test)

# Calculamos las metricas
accuracy = accuracy_score(y_test, y_pred)
precision = precision_score(y_test, y_pred, average='weighted')
recall = recall_score(y_test, y_pred, average='weighted')
f1 = f1_score(y_test, y_pred, average='weighted')

print("Accuracy:", accuracy)
print("Precision:", precision)
print("Recall:", recall)
print("F1-score:", f1)
```

1. Primero, se importan las bibliotecas y módulos necesarios. Esto incluye `load_iris` para cargar el conjunto de datos, `train_test_split` para dividir el conjunto de datos en conjuntos de entrenamiento y prueba, y varias métricas de `sklearn.metrics` para evaluar el rendimiento del modelo. También se importa la clase `TreeNetClassifier` del archivo `DecisionTreeRN.py`.

2. Luego, se carga el conjunto de datos Iris y se divide en datos (X) y etiquetas (y). También se calcula el número de variables y el número de clases.

3. Se establecen los parámetros para el entrenamiento del modelo. Esto incluye el número de épocas para el entrenamiento (`epoch`), la profundidad máxima del árbol (`depth`) y la condición de parada (`stop_condition`), que es el umbral de cambio en la pérdida entre dos iteraciones consecutivas para detener el entrenamiento.

4. Se divide el conjunto de datos en conjuntos de entrenamiento y prueba utilizando `train_test_split`.

5. Se crea una instancia del modelo `TreeNetClassifier` con los parámetros especificados y se ajusta al conjunto de entrenamiento utilizando el método `fit`.

6. Se hacen predicciones en el conjunto de prueba utilizando el método `predict` del modelo.

7. Finalmente, se calculan varias métricas de evaluación, incluyendo precisión, recall y F1-score, y se imprimen. Estas métricas proporcionan una cuantificación del rendimiento del modelo en el conjunto de prueba.

## 6.3. Implementación de TreeSVM

En esta sección se muestra la implementación del modelo TreeSVM. Al ser un modelo muy semejante al anterior, se mostrarán solo las diferencias que tienen uno respecto del otro, para así no repetirnos.

### 6.3.1. Librerías

Las librerías necesarias para el modelo TreeSVM son *Numpy* y a diferencia de TreeNet que utiliza *Torch* para tratar con redes neuronales, aquí se importa *SVC* (Support Vector Classification) de *Sklearn*.

```
from sklearn.svm import SVC
from sklearn.metrics import accuracy_score

import numpy as np
```

También se importa la métrica de tasa de aciertos ya que es la que vamos a utilizar para medir que variable o que modelo SVM ha dividido mejor el conjunto de datos.

### 6.3.2. TreeSVMClassifier

A diferencia del modelo TreeNet, que se implementa la clase de la red neuronal *Net*, el modelo SVM no hace falta implementarlo porque ya viene implementado en la propia librería. Por tanto pasamos directamente a la construcción del árbol.

La clase TreeSVMClassifier es la implementación de nuestro modelo TreeSVM. Esta clase tiene varios atributos y métodos que se utilizan para entrenar el modelo y hacer predicciones. Empezamos explicando la función constructora:

```
class TreeSVMClassifier():

def __init__(self, num_features, num_classes, depth):
    self.num_features = num_features
    self.num_classes = num_classes
    self.depth = depth
    self.root = None

    self.n_nodos = None
    self.n_hojas = None
    self.profundidad = None
    self.longitudMediaRamas = None
```

La constructora se inicializa con los mismos atributos que la clase TreeNetClassifier con la diferencia de que no tienen los atributos `epoch` y `stop_condition` ya que no los necesita.

A continuación, la función `fit` se encarga de entrenar el modelo, o más bien, de llamar a la función que construye el árbol.

```
    def fit(self, X, y):

    self.root = self.tree_SVM(X, y, self.num_classes, self.num_features, self.depth)


    self.n_nodos = self.num_nodos(self.root)
    self.n_hojas = self.num_hojas(self.root)
    self.profundidad = self.branch_length(self.root)
    self.longitudMediaRamas = self.averageBranchesLength(self.root)

    return self
```

La principal diferencia con la función fit en la clase `TreeNetClassifier` es la función que se llama para construir el árbol de decisión. En `TreeSVMClassifier`, se llama a la función `tree_SVM`, que construye el árbol de decisión utilizando un clasificador SVM. En contraste, en `TreeNetClassifier`, se llama a la función `tree_net`, que construye el árbol de decisión utilizando una red neuronal.

Después de construir el árbol de decisión, al igual que la clase `TreeNetClassifier`, se actualizan los atributos `n_nodos` (número de nodos), `n_hojas` (número de hojas), `profundidad` (profundidad del árbol) y `longitudMediaRamas` (longitud media de las ramas).

A continuación la función `tree_SVM` que es similar a `tree_Net` en su estructura general, pero hay algunas diferencias clave. Se van a explicar estas diferencias. Para el código completo consultar el Apéndice B.

En `tree_SVM`, se utiliza un modelo de Máquina de Soporte Vectorial (SVM) con un kernel lineal para entrenar en los datos, en lugar de una red neuronal como en `tree_net`. Además, en la función `tree_SVM`, en lugar de usar una condición de parada basada en la pérdida de la red neuronal, este modelo entrena el SVM durante un número fijo de iteraciones (hasta que el modelo converge).

```
    model = SVC(kernel='linear')
```

En `tree_SVM`, se calcula la tasa de aciertos SVM en los datos de entrenamiento y se utiliza para seleccionar la mejor variable y el mejor modelo. En `tree_net`, se utiliza la pérdida de la red neuronal para seleccionar la mejor variable y la mejor red.

```
    y_pred = model.predict(X[:, variable: variable + 1])

    accuracy = accuracy_score(y, y_pred)
    print("Precisión del modelo SVM: {:.2f}%".format(accuracy * 100))

    if accuracy > mejorAcc:
        mejorV = variable
        mejorAcc = accuracy
        mejorModelo = model
```

El resto es igual que en `tree_net`, salvo que para hacer las predicciones en el método de Montecarlo, o para hacer las predicciones de dividir el conjunto de datos, se utiliza el modelo SVM entrenado.

```
    y_pred = mejorModelo.predict(x_values)

    ...

    y_pred = mejorModelo.predict(X[:, mejorV: mejorV + 1])
```

El resto de código no merece la pena explicar dado que es completamente igual que en el modelo `tree_net`. La estructura del árbol generado es idéntica a la del modelo `tree_net` por tanto para las predicciones y el recorrido del árbol se utilizan los mismo algoritmos.

## 6.4. Ejemplo de uso de TreeSVM

A continuación se muestra un código de ejemplo para el uso del modelo TreeSVM.

```
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
import numpy as np

from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score

from DecisionTreeSVM import TreeSVMClassifier  # Import the TreeSVMClassifier

# Se carga el conjunto de datos
dataset = load_iris()

# Se separa el conjunto en datos (X) y etiquetas (y)
X = dataset.data
y = dataset.target

```

```
num_features = X.shape[1]
num_classes = np.unique(y).shape[0]

# Maxima profundidad que puede alcanzar el árbol
depth = 12

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2)

model = TreeSVMClassifier.TreeSVMClassifier(num_features=num_features,
                                            num_classes=num_classes,
                                            depth=depth)

model.fit(X_train, y_train)

# Se hacen predicciones sobre el conjunto test.
y_pred = model.predict(X_test)

# Se calculan las métricas de rendimiento.
accuracy = accuracy_score(y_test, y_pred)
precision = precision_score(y_test, y_pred, average='weighted')
recall = recall_score(y_test, y_pred, average='weighted')
f1 = f1_score(y_test, y_pred, average='weighted')

print("Accuracy:", accuracy)
print("Precision:", precision)
print("Recall:", recall)
print("F1-score:", f1)
```

La implementación es igual que en el modelo TreeNet salvo algunas pequeñas diferencias. No se utilizan los atributos de `epoch` ni de `stop_condition` porque no hacen falta en este modelo.

# 7. CAPÍTULO

# Experimentación

Una vez implementados los algoritmos de construcción de árboles, tanto con redes neuronales como con SVM, se ha experimentado con ellos para ver los resultados, así como su rendimiento y compararlos con otros modelos como CART y redes neuronales.

Se ha experimentado con 10 conjuntos de datos distintos, con los cuales se ha hecho 10-fold Cross-Validation para calcular las estimaciones de criterios de bondad como la tasa de aciertos, la precisión, sensibilidad y f1-score.

Dado que el trabajo tiene como objetivo la interpretabilidad de los modelos. Se estudia también las características estructurales de estos árboles construidos y se comparan con los árboles construidos con CART.

## 7.1. Conjuntos de datos

Para la experimentación se han utilizado 10 conjuntos de datos multivariables donde el problema a resolver es clasificar. Los conjuntos de datos son los siguientes:

- Iris: El conjunto de datos contiene 3 clases de 50 instancias cada una, donde cada clase representa un tipo de la planta iris. Una clase es linealmente separable de las otras 2. Estas últimas no son linealmente separables entre sí.
  https://archive.ics.uci.edu/dataset/53/iris

- Wine: El conjunto de datos contiente 3 clases, 178 instancias y 13 variables predictoras. Estos datos son los resultados de un análisis químico de vinos cultivados en la misma región de Italia pero procedentes de tres cultivares diferentes. El análisis determinó las cantidades de 13 componentes presentes en cada uno de los tres tipos de vinos.
  https://archive.ics.uci.edu/dataset/109/wine

- Breast Cancer Wisconsin: El conjunto de datos sobre el cáncer de mama es un conjunto de datos de clasificación binaria clásico y muy sencillo. Contiene 2 clases (Maligno o Benigno) donde 212 instancias son benignas y 357 instancias malignas. El total de instancias son 569 y tiene 30 variables predictoras.
  https://archive.ics.uci.edu/dataset/17/breast+cancer+wisconsin+diagnostic

- Haberman Survival: El conjunto de datos contiene casos de un estudio realizado entre 1958 y 1970 en el Hospital Billings de la Universidad de Chicago sobre la supervivencia de pacientes operadas de cáncer de mama. El conjuntos de datos contiene 306 instancias, 2 clases y 3 variables predictoras.
  https://archive.ics.uci.edu/dataset/43/haberman+s+survival

- Diabetes (Kaggle): Este conjunto de datos procede del Instituto nacional de Diabetes y enfermedades renales. El objetivo del conjunto de datos es predecir diagnósticamente si un paciente tiene diabetes basándose en ciertas mediciones diagnósticas incluidas en el conjunto de datos. El conjunto tiene 2 clases, 8 variables predictoras y 768 instancias.
  https://www.kaggle.com/datasets/uciml/pima-indians-diabetes-database

- Car Evaluation: El conjunto de datos Car Evaluation contiene información sobre diferentes atributos de automóviles y una clasificación de evaluación correspondiente. Estos atributos incluyen el precio del automóvil, el costo de mantenimiento, el número de puertas, la capacidad del maletero y la seguridad, entre otros. La clasificación de evaluación indica si el automóvil es inaceptable, aceptable, bueno o muy bueno. Por tanto tiene 6 variables predictoras, 4 clases y 1728 instancias.
  https://archive.ics.uci.edu/dataset/19/car+evaluation

- Vehicle: El objetivo es clasificar una silueta dada como uno de los cuatro tipos de vehículo, utilizando un conjunto de características extraídas de la silueta. El vehículo puede verse desde uno de muchos ángulos diferentes. Contiene 18 variables predictoras, 4 clases (OPEL. SAAB, BUS y VAN) y 946 instancias.
  https://archive.ics.uci.edu/dataset/149/statlog+vehicle+silhouettes

- Glass Identification: Del Servicio de Ciencias Forenses de EE.UU. se quieren clasificar 6 tipos de vidrio definidos en función de su contenido en óxidos. Por tanto el numero de clases es 6 con 8 variables predictoras en 214 instancias.
  https://archive.ics.uci.edu/dataset/42/glass+identification

- Image Segmentation: Las instancias se extrajeron aleatoriamente de una base de datos de 7 imágenes de exteriores. Las imágenes se segmentaron manualmente para crear una clasificación para cada píxel. Cada instancia es una región de 3x3. Contiene 7 clases, 19 variables predictoras y 2310 instancias.
  https://archive.ics.uci.edu/dataset/50/image+segmentation

- Cancer Data (Kaggle): 570 células de cáncer y 30 variables predictoras para determinar si las células cancerígenas son malignas o benignas.
  https://www.kaggle.com/datasets/erdemtaha/cancer-data

## 7.2. Resultados de rendimiento

En esta sección se van a mostrar los resultados obtenidos con las métricas de tasa de aciertos, precisión, sensibilidad, y f1-score para cada conjunto de datos y los diferentes modelos.

Se prueban los modelos presentados en este trabajo (TreeNet y TreeSVM) y se comparan con el rendimiento obtenido de las redes neuronales (RN), el modelo de arbol de decisión CART, con las máquinas de soporte vectorial (SVM) y con el Deep Neural Decision Tree (DNDT) ya mencionado anteriormente.

| Conjunto de Datos | TreeNet | TreeSVM | RN | CART | SVM | DNDT |
|---|---|---|---|---|---|---|
| Iris | 0.93 | 0.94 | **0.97** | 0.93 | 0.96 | 0.95 |
| Wine | 0.92 | 0.88 | 0.65 | 0.92 | 0.66 | **0.95** |
| Breast Cancer | 0.91 | 0.91 | 0.90 | **0.93** | 0.92 | 0.91 |
| Haberman Survival | **0.73** | **0.73** | 0.70 | **0.73** | **0.73** | **0.73** |
| Diabetes (K) | 0.74 | 0.74 | 0.64 | 0.75 | **0.76** | 0.65 |
| Car Evaluation | 0.80 | 0.68 | **0.95** | 0.85 | 0.93 | 0.94 |
| Vehicle | 0.82 | 0.81 | **0.97** | 0.90 | 0.80 | 0.76 |
| Glass Identification | 0.84 | 0.98 | **0.99** | 0.88 | 0.81 | 0.62 |
| Image Segmentation | 0.90 | 0.93 | **0.97** | 0.95 | 0.86 | 0.69 |
| Cancer Data (K) | 0.90 | 0.91 | **0.93** | 0.93 | 0.92 | 0.92 |
| **Media** | 0.85 | 0.85 | 0.87 | **0.88** | 0.83 | 0.81 |

**Tabla 7.1:** Tasa de Aciertos

Echando un vistazo a la tabla, se ve claramente que no hay un modelo que gane en todas las situaciones. En realidad, depende mucho del conjunto de datos que estemos mirando. Por ejemplo, si nos fijamos en el conjunto de datos Iris, las redes neuronales (RN) son las que mejor resultado dan con una tasa de acierto del 97 %. TreeSVM, SVM y DNDT no están muy lejos, pero no consiguen un resultado tan bueno. Al final, una red neuronal tiene más flexibilidad para adaptarse a los patrones de los datos, que puede ser especialmente útil en situaciones donde las clases no están claramente separadas por líneas rectas.

Por otro lado, en el conjunto de datos Wine, DNDT obtiene la mejor tasa de acierto que es del 95 %. TreeNet y CART no se quedan atrás y muestran un rendimiento bastante sólido.

Lo que es interesante es que en el conjunto de datos Haberman Survival, todos los modelos están en el mismo nivel. Todos consiguen una tasa de acierto del 73 %, lo que me hace pensar que este conjunto de datos es un desafío para todos los modelos.

Si miramos la media de las tasas de acierto, que se encuentra en la última fila de la tabla, vemos que CART coloca en primer lugar con una media del 88 %. Las redes neuronales (RN) están muy cerca con el 87 %. Y nuestros modelos, TreeNet y TreeSVM, con un rendimiento decente de 85 %.

| Conjunto de Datos | TreeNet | TreeSVM | RN | CART | SVM | DNDT |
|---|---|---|---|---|---|---|
| Iris | 0.94 | 0.95 | **0.97** | 0.93 | **0.97** | 0.93 |
| Wine | 0.94 | 0.91 | 0.57 | 0.94 | 0.64 | **0.96** |
| Breast Cancer | 0.91 | 0.92 | 0.91 | **0.93** | 0.92 | 0.92 |
| Haberman Survival | **0.71** | 0.55 | 0.68 | **0.71** | 0.54 | 0.54 |
| Diabetes (K) | 0.73 | 0.75 | 0.65 | 0.74 | **0.76** | 0.62 |
| Car Evaluation | 0.80 | 0.48 | **0.95** | 0.83 | 0.94 | **0.95** |
| Vehicle | 0.81 | 0.81 | **0.97** | 0.92 | 0.79 | 0.59 |
| Glass Identification | 0.83 | **0.99** | **0.99** | 0.85 | 0.73 | 0.60 |
| Image Segmentation | 0.91 | 0.93 | **0.97** | 0.95 | 0.87 | 0.71 |
| Cancer Data (K) | 0.91 | 0.91 | **0.93** | 0.93 | 0.92 | 0.92 |
| **Media** | 0.85 | 0.82 | 0.86 | **0.87** | 0.81 | 0.77 |

**Tabla 7.2:** Precisión

La Tabla 7.2 representa la precisión de los mismos modelos y conjuntos de datos que la tabla anterior. La precisión es una métrica que nos indica cuántas de las predicciones positivas del modelo fueron realmente correctas. Es especialmente importante en contextos donde los falsos positivos son problemáticos.

En general, los resultados son similares a los de la tabla de tasas de acierto. Sin embargo, se observan algunas diferencias notables. Por ejemplo, en el conjunto de datos de Car Evaluation, los modelos TreeNet y TreeSVM tienen una precisión considerablemente menor en comparación con las RN y DNDT. Similarmente, en el conjunto de datos de Haberman Survival, TreeSVM tiene una precisión significativamente más baja que los otros modelos.

En cuanto a las medias, CART sigue siendo el modelo con la precisión promedio más alta, seguido de cerca por las RN. TreeNet tiene una precisión promedio similar a la tasa de acierto, mientras que TreeSVM, SVM y DNDT tienen precisiones promedio más bajas.

| Conjunto de Datos | TreeNet | TreeSVM | RN | CART | SVM | DNDT |
|---|---|---|---|---|---|---|
| Iris | 0.93 | 0.94 | **0.97** | 0.93 | 0.96 | 0.95 |
| Wine | 0.92 | 0.88 | 0.65 | 0.92 | 0.66 | **0.95** |
| Breast Cancer | 0.91 | 0.91 | 0.90 | **0.93** | 0.92 | 0.91 |
| Haberman Survival | **0.73** | **0.73** | 0.70 | **0.73** | **0.73** | **0.73** |
| Diabetes (K) | 0.74 | 0.74 | 0.64 | 0.75 | **0.76** | 0.65 |
| Car Evaluation | 0.80 | 0.68 | **0.95** | 0.85 | 0.93 | 0.94 |
| Vehicle | 0.82 | 0.81 | **0.97** | 0.90 | 0.80 | 0.76 |
| Glass Identification | 0.84 | 0.98 | **0.99** | 0.88 | 0.81 | 0.62 |
| Image Segmentation | 0.90 | 0.93 | **0.97** | 0.95 | 0.86 | 0.69 |
| Cancer Data (K) | 0.90 | 0.91 | **0.93** | **0.93** | 0.92 | 0.92 |
| **Media** | 0.85 | 0.85 | 0.87 | **0.88** | 0.83 | 0.81 |

**Tabla 7.3:** Sensibilidad

La Tabla 7.3 presenta la sensibilidad de los modelos, que es una métrica que indica qué tan bien el modelo puede identificar los casos positivos. Es especialmente útil cuando te importa no pasar por alto los positivos.

Al igual que antes, los resultados varían dependiendo del conjunto de datos. Por ejemplo, las redes neuronales (RN) obtienen el mejor rendimiento en los conjuntos de datos Iris, Car Evaluation, Vehicle, Glass Identification, Image Segmentation, y Cancer Data. DNDT tiene el mejor rendimiento en el conjunto de datos Wine, y varios modelos comparten el mejor rendimiento en los conjuntos de datos Haberman Survival y Diabetes.

Mirando las medias, CART tiene la sensibilidad promedio más alta con un 88 %, seguido por las RN con un 87 %. Nuestros modelos, TreeNet y TreeSVM, muestran una sensibilidad promedio de 85 %, lo que sugiere que pueden detectar una gran proporción de los casos positivos, aunque no sean los mejores en todos los conjuntos de datos.

| Conjunto de Datos | TreeNet | TreeSVM | RN | CART | SVM | DNDT |
|---|---|---|---|---|---|---|
| Iris | 0.93 | 0.94 | **0.97** | 0.93 | 0.96 | 0.94 |
| Wine | 0.92 | 0.88 | 0.57 | 0.92 | 0.63 | **0.95** |
| Breast Cancer | 0.90 | 0.91 | 0.90 | **0.93** | 0.91 | 0.91 |
| Haberman Survival | 0.69 | 0.63 | 0.67 | **0.70** | 0.62 | 0.62 |
| Diabetes (K) | 0.72 | 0.73 | 0.64 | **0.74** | **0.74** | 0.61 |
| Car Evaluation | 0.77 | 0.56 | **0.95** | 0.83 | 0.93 | 0.94 |
| Vehicle | 0.80 | 0.77 | **0.97** | 0.90 | 0.77 | 0.66 |
| Glass Identification | 0.82 | 0.98 | **0.99** | 0.86 | 0.76 | 0.59 |
| Image Segmentation | 0.90 | 0.93 | **0.97** | 0.95 | 0.86 | 0.68 |
| Cancer Data (K) | 0.89 | 0.91 | **0.93** | **0.93** | 0.91 | 0.92 |
| **Media** | 0.84 | 0.82 | 0.86 | **0.87** | 0.81 | 0.78 |

**Tabla 7.4:** F1-Score

La Tabla 7.4 muestra el F1-Score de los modelos en cada conjunto de datos. El F1-Score es una medida que combina precisión y sensibilidad, y es especialmente útil cuando las clases están desbalanceadas.

Los resultados son consistentes con las tablas anteriores. En la mayoría de los conjuntos de datos, las redes neuronales (RN) obtienen el mejor rendimiento. Sin embargo, en los conjuntos de datos Wine y Haberman Survival, DNDT y CART tienen el mejor rendimiento respectivamente.

En cuanto a las medias, CART tiene el F1-Score promedio más alto con un 87 %, seguido de cerca por las RN con un 86 %. TreeNet y TreeSVM tienen F1-Scores promedio de 84 % y 82 % respectivamente, lo que indica que, aunque no son los mejores en todas las situaciones, ofrecen un rendimiento equilibrado en términos de precisión y sensibilidad.

Los modelos por los general tanto TreeNet como TreeSVM, obtienen resultados muy similares entre ellos. Menos algunos casos en concreto. Luego si los comparamos con el modelo CART, estos no alcanzan la precisión de CART. Además se podría pensar que si dedicamos más tiempo a entrenar la red neuronal en cada nodo del árbol, obtendríamos mejores resultados. Sin embargo, en nuestra experiencia, eso no es necesariamente el caso. El rendimiento no siempre mejora con más iteraciones de entrenamiento, lo que sugiere que los resultados que estamos viendo son el mejor rendimiento que podemos esperar de nuestros modelos, dado el enfoque actual.

## 7.3. Análisis de complejidad de los árboles

Para evaluar la complejidad de los árboles generados, utilizamos tres métricas distintas: el número de nodos, el número de hojas y la profundidad del árbol. Estas métricas nos dan una idea de la estructura general del árbol, lo que puede ayudarnos a entender su interpretabilidad.

En general, un árbol con menos nodos y hojas y una menor profundidad puede ser más fácil de interpretar porque hay menos decisiones que seguir y entender. Sin embargo, un árbol más complejo con más nodos y hojas y una mayor profundidad puede ser más difícil de interpretar, ya que hay más decisiones y caminos posibles a considerar. En los modelos TreeNet y TreeSVM, observamos que los árboles no tienden a expandirse tanto como en el caso de CART. Esta característica limitada de expansión puede proporcionar una ventaja en términos de control de la varianza y prevención del sobreajuste.

Es importante destacar que estas métricas de complejidad del árbol son específicas para los modelos basados en árboles, como TreeNet y TreeSVM, y no se pueden aplicar a las redes neuronales (RN) y a las máquinas de vectores de soporte (SVM) dado que estas métricas solo sirven para modelos basados en árboles.

| **Conjunto de Datos** | **TreeNet** | **TreeSVM** | **CART** |
|---|---|---|---|
| Iris | 2 | 2 | 17 |
| Wine | 9 | 9 | 21 |
| Breast Cancer | 5 | 1 | 39 |
| Haberman Survival | 16 | 1 | 195 |
| Diabetes (K) | 6 | 0 | 167 |
| Car Evaluation | 7 | 0 | 93 |
| Vehicle | 12 | 4 | 65 |
| Glass Identification | 8 | 1 | 11 |
| Image Segmentation | 71 | 44 | 135 |
| Cancer Data (K) | 11 | 4 | 43 |
| **Media** | 14.7 | 6.6 | 90.6 |

**Tabla 7.5:** Número de nodos

La tabla 7.5 muestra el número de nodos para diferentes tipos de árboles de decisión (TreeNet, TreeSVM y CART) aplicados a varios conjuntos de datos. En general, se observa que los árboles TreeNet y TreeSVM tienden a tener menos nodos en comparación con los árboles CART. Esto indica que los árboles TreeNet y TreeSVM suelen tener estructu-

ras más simples, lo que puede facilitar su interpretación ya que hay menos decisiones que seguir y entender.

Por otro lado, los árboles CART tienden a tener un número mayor de nodos, lo que sugiere estructuras más complejas.

También se puede ver que el conjunto de Image Segmentation se destaca por tener el mayor número de nodos en todos los tipos de árboles. Esto puede indicar que este conjunto de datos tiene una estructura más compleja o una mayor variabilidad que requiere árboles de decisión más complejos para su clasificación.

| **Conjunto de Datos** | **TreeNet** | **TreeSVM** | **CART** |
|---|---|---|---|
| Iris | 3 | 4 | 9 |
| Wine | 12 | 11 | 11 |
| Breast Cancer | 6 | 2 | 20 |
| Haberman Survival | 17 | 2 | 98 |
| Diabetes (K) | 7 | 1 | 84 |
| Car Evaluation | 11 | 1 | 47 |
| Vehicle | 13 | 5 | 33 |
| Glass Identification | 10 | 6 | 6 |
| Image Segmentation | 96 | 70 | 68 |
| Cancer Data (K) | 11 | 5 | 22 |
| **Media** | 18.6 | 10.7 | 39.8 |

**Tabla 7.6:** Número de hojas

La tabla 7.6 refleja el número de hojas en los mismos tres tipos de árboles de decisión (TreeNet, TreeSVM y CART) para los conjuntos de datos dados.

Al igual que antes, los árboles TreeNet y TreeSVM tienden a tener menos hojas, lo que sugiere una estructura de árbol más simple. Por otro lado, los árboles CART suelen tener más hojas, lo que indica una estructura de árbol más compleja.

Además, en algunos casos, el árbol TreeSVM tiene un número muy bajo de hojas, lo que indica una estructura de árbol extremadamente simple para esos conjuntos de datos específicos.

Por lo tanto, tanto el número de nodos como el número de hojas refuerzan la misma tendencia: los árboles TreeNet y TreeSVM tienden a ser más simples y posiblemente más interpretables, mientras que los árboles CART son generalmente más complejos.

| Conjunto de Datos | TreeNet | TreeSVM | CART |
|---|---|---|---|
| Iris | 3 | 3 | 5 |
| Wine | 5 | 5 | 6 |
| Breast Cancer | 4 | 2 | 7 |
| Haberman Survival | 6 | 2 | 12 |
| Diabetes (K) | 5 | 1 | 12 |
| Car Evaluation | 5 | 1 | 12 |
| Vehicle | 6 | 5 | 10 |
| Glass Identification | 5 | 2 | 5 |
| Image Segmentation | 8 | 8 | 12 |
| Cancer Data (K) | 8 | 4 | 8 |
| **Media** | 5.5 | 3.3 | 8.9 |

**Tabla 7.7:** Profundidad

La profundidad de un árbol de decisión es una medida importante ya que puede impactar tanto en la precisión del modelo como en su interpretabilidad. Árboles más profundos pueden capturar relaciones más complejas, pero también pueden ser más propensos al sobreajuste y ser más difíciles de interpretar.

En la tabla 7.7, vemos que el modelo CART tiende a generar árboles más profundos, con una profundidad media de 8.9. Esto puede indicar que CART encuentra relaciones más complejas en los datos, pero también puede sugerir un mayor riesgo de sobreajuste.

Por otro lado, los modelos TreeNet y TreeSVM tienden a generar árboles más cortos, con profundidades medias de 5.5 y 3.3 respectivamente. Esto puede hacer que estos modelos sean más fáciles de interpretar, pero también puede limitar su capacidad para encontrar relaciones más complejas.

## 7.4. Coste computacional

En esta sección se van a mostrar los tiempos de ejecución de cada modelo con su correspondiente conjunto de datos.

Cabe destacar que estos resultados pueden variar dependiendo del número de épocas que se elijan en los modelos basados en redes neuronales. Para TreeNet se han utilizado 30000 épocas, para las RN se han utilizado 10000 épocas y para DNDT 10000 épocas.

| Conjunto de Datos | TreeNet | TreeSVM | RN | CART | SVM | DNDT |
|---|---|---|---|---|---|---|
| Iris | 103.295 | 0.214 | 39.681 | 0.0006 | 0.004 | 15.618 |
| Wine | 385.121 | 3.09 | 49.677 | 0.0007 | 0.007 | 38.530 |
| Breast Cancer | 2084.98 | 83.266 | 75.377 | 0.008 | 0.015 | 47.803 |
| Haberman Survival | 307.14 | 0.226 | 43.029 | 0.0005 | 0.005 | 12.901 |
| Diabetes (K) | 1042.317 | 51.117 | 59.842 | 0.002 | 0.015 | 48.206 |
| Car Evaluation | 148.00 | 0.793 | 59.868 | 0.0005 | 0.01 | 41.670 |
| Vehicle | 166.783 | 18.917 | 106.946 | 0.002 | 0.015 | 54.913 |
| Glass Identification | 656.63 | 0.304 | 43.530 | 0.0006 | 0.005 | 42.944 |
| Image Segmentation | 2082.25 | 188.281 | 90.487 | 0.011 | 0.055 | 84.561 |
| Cancer Data (K) | 403.90 | 93.335 | 79.322 | 0.006 | 0.007 | 48.636 |
| **Media** | 738.0416 | 43.9543 | 57.2390 | 0.0954 | 0.0138 | 43.5782 |

**Tabla 7.8:** Tiempos de ejecución

La tabla 7.8 refleja los tiempos de ejecución en segundos que ha necesitado cada modelo en entrenarse para cada conjunto de datos.

Podemos observar que los modelos basados en redes neuronales tardan mucho más que los otros, sin embargo entre estos también hay una diferencia notable. Los resultados de TreeNet dejan mucho que desear ya que requieren un tiempo excesivo.

En cuanto a los modelos TreeSVM, CART y SVM los tiempos son rápido, casi instantáneos. Sin embargo TreeSVM tiene algunos tiempos que superan el minuto. Esto se debe a que el algoritmo de construcción es igual que el de TreeNet. Esto quiere decir que su forma recursiva de construcción no es lo más óptima que podría ser.

Cabe también mencionar que para DNDT se limita el numero de variables a 8. De lo contrario los tiempos crecerían exponencialmente y sería imposible de probar el modelo. En ese sentido es posible que los tiempos de DNDT en la tabla no sean del todo representativos de lo que realmente deberían ser.

## 7.5. Discusión de Resultados

Después de ver estos resultados nos surgen algunas preguntas. Por ejemplo, ¿cómo es que, usando el modelo TreeSVM, los árboles generados con los conjuntos de datos Breast Cancer o Haberman solo tengan un nodo? O usando el conjunto de datos de Diabetes ¿Cómo puede ser que no tengan ningún nodo?

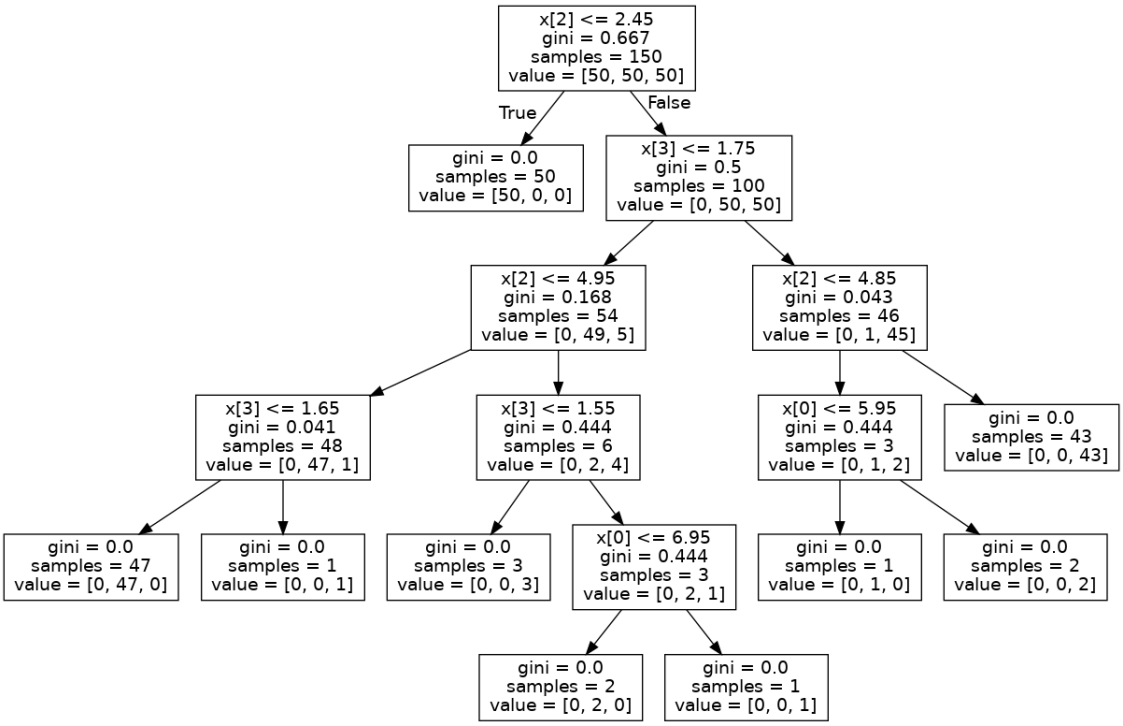


**Figura 7.1:** Árbol generado por CART

La respuesta a esta pregunta es sencilla. El modelo SVM no es capaz de calcular un hiperplano que separe el conjunto inicial. Esto puede pasar por dos razones, una, que las clases del conjunto están muy mezcladas y por tanto da igual dónde coloques un hiperplano que no se va a mejorar tu resultado. La segunda razón posible es que el conjunto de datos esté desbalanceado.

De ser así, una red neuronal, por ejemplo, tampoco es capaz de calcular un punto de corte dado que si te dice que la respuesta es la clase mayoritaria va a acertar casi siempre. Por tanto opta por no aprender ningún punto de corte y simplemente devolver la clase mayoritaria. De esta forma la tasa de aciertos es alta pero realmente no ha aprendido nada. Para compensar esto nos fijamos en otras métricas como podría ser F1-score.

En las figuras 7.1, 7.2 y 7.3 se muestran tres árboles distintos generados con CART, TreeNet y TreeSVM con el conjunto de datos Iris. Podemos observar que los tres árboles eligen por lo general las mismas variables para separar el conjunto. La diferencia, como se ha explicado anteriormente, es la forma de elegir esta variable. CART lo hace con el índice gini, TreeNet con la función de pérdida y TreeSVM con la tasa de aciertos. También se ve que los árboles generados por TreeSVM y TreeNet son casi idénticos ya que su forma de construirse es muy similar.

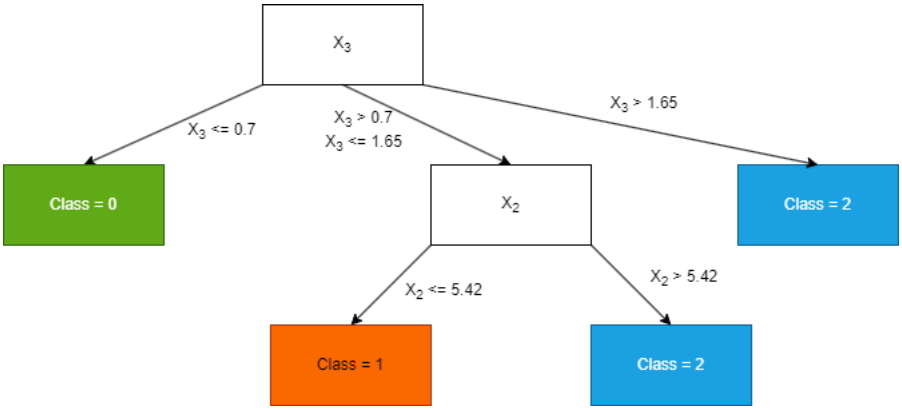


**Figura 7.2:** Árbol generado por TreeSVM

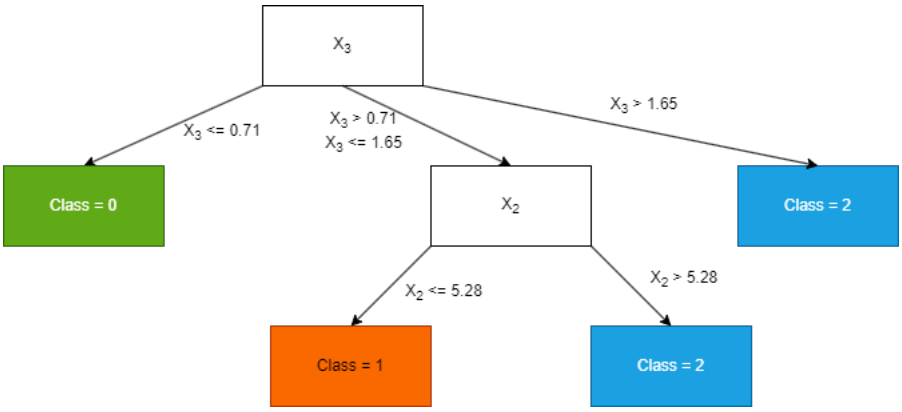


**Figura 7.3:** Árbol generado por TreeNet

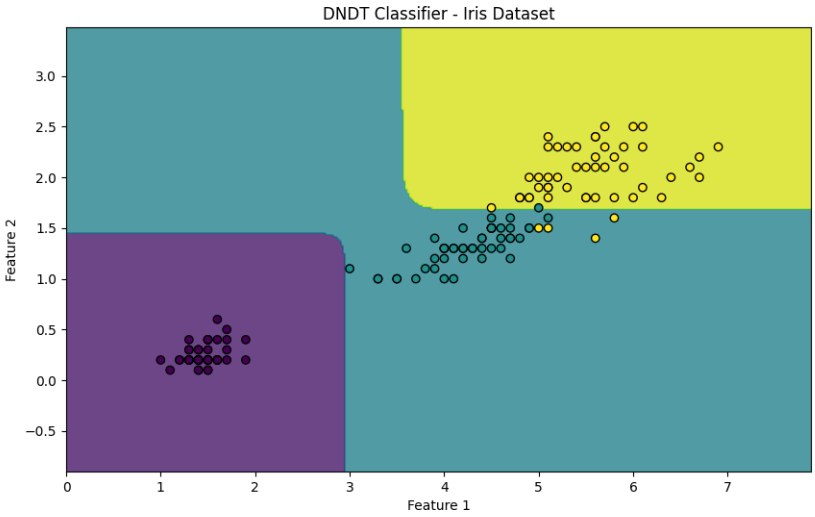


**Figura 7.4:** Fronteras de decisión DNDT

## 7.6. Observaciones de DNDT

Aunque DNDT se presenta como un árbol de decisión, en realidad es una red neuronal que intenta aproximar el comportamiento de un árbol. Esto significa que los puntos de corte que genera no son fijos, sino más bien curvos, lo que dificulta la abstracción del árbol generado (Véase la figura 7.4). Luego los puntos que ha calculado son 3.225 para la variable 1 y 1.77 para la variable 2 y ya vemos en la figura que no son muy precisos. Nuestros modelos, por otro lado, respetan los puntos de corte que obtienen.

También se ha observado que los árboles generados por DNDT tienen una peculiaridad que limita su utilidad. En cada nivel del árbol, se utiliza el mismo punto de corte en todas las ramas. Esto no es ideal, ya que limita la flexibilidad del árbol para adaptarse a diferentes estructuras de datos (Véase 7.5). En contraste, nuestros modelos, TreeNet y TreeSVM, no presentan este problema y pueden utilizar diferentes puntos de corte en diferentes ramas, lo que les permite adaptarse mejor a los datos.

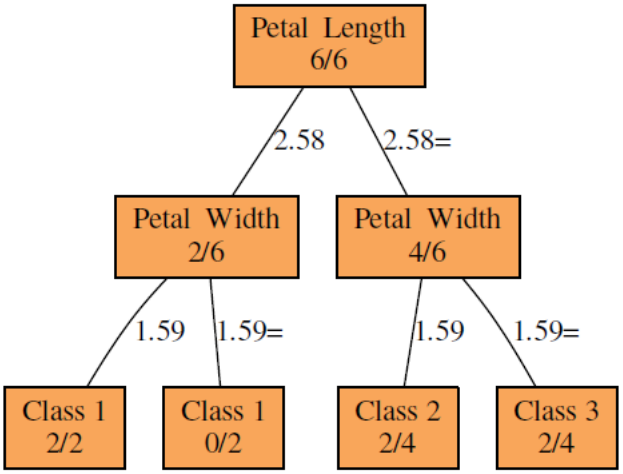


**Figura 7.5:** Árbol abstraído de DNDT

# 8. CAPÍTULO

# Conclusiones y Líneas abiertas

## 8.1. Conclusiones

El objetivo principal de este proyecto ha sido profundizar en el algoritmo DNDT, analizar sus limitaciones y, si es posible, superarlo con nuestras propuestas, TreeNet y TreeSVM. Tras un análisis exhaustivo, podemos afirmar que hemos logrado nuestro objetivo.

Los modelos basados en árboles, por lo general, funcionan bien con conjuntos de datos de tipo tabular. Dicho esto, los algoritmos propuestos TreeNet y TreeSVM demuestran una eficacia similar al otro algoritmo basado en árboles CART, aunque no logran alcanzarlo.

Ahora bien, una consideración importante al trabajar con este modelo es el tiempo que se necesita para construir el árbol de decisión. El algoritmo TreeNet entrena una red neuronal por cada nodo del árbol, esto supone unos tiempos de entrenamientos costosos. Por ejemplo, el conjunto de datos de Image Segmentation tarda casi una hora en entrenarse.

Por eso, se ha integrado también un algoritmo basado en SVM. Sustituyendo a las redes neuronales para así tener una mejor eficiencia. Por ejemplo, el conjunto de datos Image Segmentation se entrena aproximadamente en 3 minutos frente a las la hora que tarda usando TreeNet. Estos tiempos pueden variar dependiendo del hardware con el que se trabaja.

También se ha visto que los modelos DNDT no se comportan totalmente como árboles de decisión, lo que no los convierte del todo en modelos interpretables si no más bien en una aproximación de un árbol de decisión.

Al final, se han conseguido dos algoritmos capaces de generar modelos basados en árboles de decisión utilizando redes neuronales y SVM. Estos modelos son totalmente interpretables y capaces de dar resultados decentes en diferentes conjuntos de datos.

## 8.2. Líneas Abiertas

En cuanto a las líneas de trabajo futuras, hay varias áreas en las que estos modelos podrían mejorarse. Una de las limitaciones actuales es que los modelos, TreeNet y TreeSVM, solo pueden trabajar con valores numéricos. Esto significa que no pueden manejar directamente variables categóricas. Se podría extender estos modelos para que puedan manejar este tipo de datos de manera más efectiva.

Además, también estamos interesados en explorar formas de mejorar la eficiencia de nuestros modelos. Aunque ya hemos logrado reducir significativamente los tiempos de entrenamiento en comparación con DNDT, creemos que todavía hay margen para mejorar. Esto podría implicar la exploración de técnicas de optimización más avanzadas o la adaptación de nuestros modelos para que puedan aprovechar mejor el hardware moderno. Por ejemplo, la utilización de plataformas como CUDA.

# A. ANEXO

## Código tree_net

```
def tree_net(self,
             X,
             y,
             epoch,
             num_classes,
             num_variables,
             longitud_inicial,
             stop_condition,
             depth):

    if depth == 0:
        return np.bincount(y).argmax()

    mejorLoss = float("Inf")
    mejorNet = None
    mejorVar = 0

    tree = {"Variable": 0, "Loss": float("Inf"), "data": X, "target": y}

    for i in range(num_variables):
        Xaux = X[:, i:i + 1]

        net = Net(num_classes)
        criterion = nn.CrossEntropyLoss()
        optimizer = optim.Adam(net.parameters())

        lossAux = float("Inf")
        for _ in range(epoch * int(longitud_inicial / len(y))):
            optimizer.zero_grad()
            outputs = net(torch.tensor(Xaux, dtype=torch.float32))
            loss = criterion(outputs, torch.tensor(y, dtype=torch.long))
```

```
            loss.backward()
            optimizer.step()

            if abs(loss.item() - lossAux) < stop_condition:
                break
            lossAux = loss.item()

            if _ % 100 == 0:
                print(f"Epoch {_}: Loss={loss.item():.4f}")

        if loss.item() < mejorLoss:
            tree["Variable"], tree["Loss"] = i, loss.item()
            mejorVar = i
            mejorNet = net
            mejorLoss = loss.item()

    x_values = np.linspace(X.min() - 1, X.max() + 1, 2000).reshape(-1, 1)

    y_pred = mejorNet(torch.tensor(x_values,
                                   dtype=torch.float32)).argmax(axis=1).numpy()

    c = y_pred[0]
    Cutpoints = []
    Valores = [c]
    for i, prediccion in enumerate(y_pred):
        if prediccion != c:
            c = prediccion
            Cutpoints.append(x_values[i].item())
            Valores.append(c)

    tree["Cutpoints"] = Cutpoints
    tree["Valores"] = Valores

    pobx = {}
    poby = {}
    for i in enumerate(X[:, mejorVar:mejorVar + 1]):
        pre = mejorNet(torch.tensor(i[1], dtype=torch.float32)).argmax().item()
        if pre not in poby:
            pobx[pre], poby[pre] = [], []
        pobx[pre].append(X[i[0]])
        poby[pre].append(y[i[0]])

    num_hijos = len(poby)
    tree["Hijos"] = {}
    for key, value in poby.items():
        if len(np.unique(np.array(value))) == 1 or num_hijos == 1:
            tree["Hijos"][key] = np.bincount(value).argmax()
        else:
            tree["Hijos"][key] = self.tree_net(
                np.array(pobx[key]).reshape((len(pobx[key]), num_variables)),
                np.array(value),
                epoch,
```

```
                num_classes,
                num_variables,
                longitud_inicial,
                stop_condition,
                depth - 1)

    return tree
```

# B. ANEXO

## Código tree_SVM

```
def tree_SVM(self, X, y, num_classes, num_variables, depth=3):

    if depth == 0:
        return np.bincount(y).argmax()

    mejorV = 0
    mejorAcc = 0
    mejorModelo = None

    for variable in range(num_variables):

        model = SVC(kernel='linear')
        model.fit(X[:, variable: variable + 1], y)
        y_pred = model.predict(X[:, variable: variable + 1])

        accuracy = accuracy_score(y, y_pred)
        print("Precisión del modelo SVM: {:.2f}%".format(accuracy * 100))

        if accuracy > mejorAcc:
            mejorV = variable
            mejorAcc = accuracy
            mejorModelo = model

    tree = {
        "Variable": mejorV,
        "Accuracy": mejorAcc,
        "data": X,
        "target": y
    }

    x_values = np.linspace(X.min() - 1, X.max() + 1, 2000).reshape(-1, 1)
```

```
32
33      y_pred = mejorModelo.predict(x_values)
34
35      c = y_pred[0]
36      Cutpoints = []
37      Valores = [c]
38      for i, prediccion in enumerate(y_pred):
39          if prediccion != c:
40              c = prediccion
41              Cutpoints.append(x_values[i].item())
42              Valores.append(c)
43
44      tree["Cutpoints"] = Cutpoints
45      tree["Valores"] = Valores
46
47      y_pred = mejorModelo.predict(X[:, mejorV: mejorV + 1])
48
49      pobx = {}
50      poby = {}
51
52      for i in enumerate(y_pred):
53          pre = i[1]
54          if pre not in poby:
55              pobx[pre], poby[pre] = [], []
56          pobx[pre].append(X[i[0]])
57          poby[pre].append(y[i[0]])
58
59      tree["pobx"] = pobx
60      tree["poby"] = poby
61
62      num_hijos = len(poby)
63      tree["Hijos"] = {}
64      for key, value in poby.items():
65          if len(np.unique(np.array(value))) == 1 or num_hijos == 1:
66              tree["Hijos"][key] = np.bincount(value).argmax()
67          else:
68              tree["Hijos"][key] = self.tree_SVM(
69                  np.array(pobx[key]).reshape((len(pobx[key]), num_variables)),
70                  np.array(value),
71                  num_classes,
72                  num_variables,
73                  depth - 1)
74
75      return tree
```

# Bibliografía

[Bostrom and Yudkowsky, 2014] Bostrom, N. and Yudkowsky, E. (2014). The ethics of artificial intelligence. In *The Cambridge Handbook of Artificial Intelligence*, pages 316–334. Cambridge University Press.

[Breiman, 2001] Breiman, L. (2001). Random forests. *Mach. Learn.*, 45(1):5–32.

[Breiman et al., 1984] Breiman, L., Friedman, J. H., Olshen, R. A., and Stone, C. J. (1984). Classification and regression trees.

[D'Addona, 2014] D'Addona, D. M. (2014). *Neural Network*, pages 911–918. Springer Berlin Heidelberg, Berlin, Heidelberg.

[Doshi-Velez and Kim, 2017] Doshi-Velez, F. and Kim, B. (2017). Towards a rigorous science of interpretable machine learning.

[Quinlan, 1993] Quinlan, J. R. (1993). *C4.5: programs for machine learning*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[Rota Bulo and Kontschieder, 2014] Rota Bulo, S. and Kontschieder, P. (2014). Neural decision forests for semantic image labelling. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.

[Yang et al., 2018] Yang, Y., Morillo, I. G., and Hospedales, T. M. (2018). Deep neural decision trees.