INSTITUTO POLITÉCNICO NACIONAL

CENTRO DE INVESTIGACIÓN EN COMPUTACIÓN

Tarea final

Edgar Fernando Espinosa Torres

Clasificación inteligente de patrones

Dr. Cornelio Yáñez Márquez

Ciudad de México, 14 de enero de 2024.

Índice general

Índice de figuras

Índice de tablas

Introducción

En esta tarea final, hemos explorado y descrito el funcionamiento de nueve clasificadores distintos, cada uno con sus características únicas. Estos clasificadores incluyen:

1. La familia k-Nearest Neighbors (k-NN) para k=1, 3, 5: este es un método intuitivo que clasifica los objetos basándose en los k-vecinos más cercanos en el espacio de atributos.

2. Clasificador bayesiano: un enfoque probabilístico que utiliza el teorema de Bayes para predecir la clase de una instancia.

3. Árboles de decisión: Un algoritmo que utiliza una estructura de árbol para tomar decisiones basándose en los atributos.

4. Random Forest: un ensemble de árboles de decisión que puede mejorar el rendimiento.

5. AdaBoost: un algoritmo que combina múltiples *weak learnes* para formar un *strong learner*.

6. Support Vector Machine (SVM): un enfoque poderoso y versátil que encuentra el hiperplano óptimo para separar diferentes clases.

7. Multilayer Perceptron (MLP): Una red neuronal artificial con múltiples capas que puede capturar relaciones complejas entre las instancias.

Para una evaluación exhaustiva, se trabajó con el *dataset* de hepatitis biclase. Este *dataset* presenta la particularidad de que todos sus atributos son numéricos, pero tiene algunos valores faltantes, así que se procedió a realizar una imputación de ellos utilizando la media por clase. Posteriormente, se aplicó el método de validación *leave-one-out*. Todos los clasificadores fueron implementados en Python, pero también se realizó un ejercicio paralelo en el *software* WEKA para propósitos comparativos.

El análisis se profundizó mediante la evaluación de varias métricas de desempeño, como *Balanced accuracy*, *TP Rate*, *TN Rate*, *Precision*, *F1-Score* y *MCC*. Estas métricas proporcionan una visión integral del rendimiento de los clasificadores.

Posteriormente, se aplicó el test de Friedman para evaluar diferencias generales en el rendimiento entre los clasificadores. La hipótesis nula planteada fue que no existen diferencias significativas en el rendimiento entre los clasificadores. Tras rechazar la hipótesis nula, se procedió con un análisis post-hoc utilizando el test de Nemenyi. Este análisis permitió identificar diferencias significativas en el rendimiento entre pares específicos de clasificadores, con un umbral de significancia estadística establecido en $p=0.05$.

Los resultados revelaron diferencias estadísticamente significativas entre varios pares de clasificadores, como entre 1-NN y Random Forest, 1-NN y AdaBoost, entre otros. Estos hallazgos son cruciales para entender las fortalezas y limitaciones de cada clasificador en contextos específicos y proporcionan una base sólida para futuras investigaciones y aplicaciones prácticas en el campo de la clasificación de patrones.

Desarrollo y discusión

Propósitos de la tarea

Ejemplificar el uso y aplicación de los algoritmos de clasificación inteligente de patrones más relevantes del estado del arte.

Clasificadores

k-Nearest Neighbors (kNN)

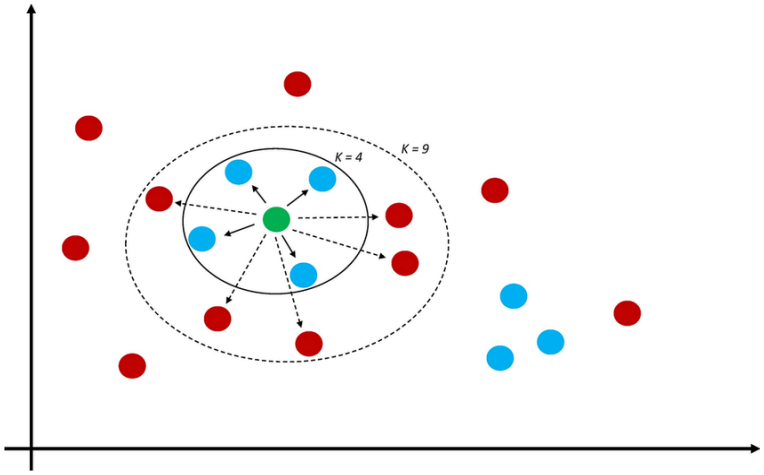

Figura 1: k-NN para k=4 y k=5.

La familia de los k-NN (k-vecinos más cercanos) son algoritmos de aprendizaje automático utilizado para clasificación y regresión. Funciona encontrando los k puntos más cercanos (vecinos) a una instancia dada y realiza predicciones basadas en estos vecinos: para clasificación, se toma la clase más común entre los vecinos, y para regresión, se calcula el promedio de sus valores. Es simple y efectivo, con la clave siendo la elección del número de vecinos, k, y la medida de distancia utilizada [1].

El primer paso en k-NN es calcular la distancia entre la instancia a clasificar del conjunto de prueba y todas las instancias en el conjunto de entrenamiento.

Para la distancia entre dos puntos $P = (p_1, p_2, \dots, p_n)$ y $Q = (q_1, q_2, \dots, q_n)$ en un espacio $n - dimensional$ las medidas de distancia más comunes se calculan de la siguiente manera:

Distancia euclidiana: $d(P, Q) = \sqrt{\sum_{i=1}^{n}(q_i - p_i)^2}$

Distancia de Manhattan: $d(P, Q) = \sum_{i=1}^{n}|q_i - p_i|$

Distancia de Minkowski: $d(P,Q) = (\sum_{i=1}^{n} |q_i - p_i|^p)^{\frac{1}{p}}$ es una generalización de las distancias anteriores.

Una vez calculadas las distancias, se seleccionan los k vecinos más cercanos. Esto significa elegir las k instancias (gráficamente representadas como puntos) en el conjunto de entrenamiento que están más cerca del punto que estamos clasificando, según la métrica elegida.

Para ello, se cuentan las frecuencias de las diferentes clases (etiquetas) entre los k vecinos más cercanos. La clase que más se repite entre los vecinos es asignada a la instancia que estamos clasificando. En caso de empate, se puede elegir la primera instancia que aparece en el empate, o al azar [1].

Según [2], las ventajas y desventajas del clasificador son las siguientes:

Ventajas de k-NN

1. Simplicidad y facilidad de implementación.

2. No paramétrico: al ser un algoritmo no paramétrico, no hace suposiciones sobre la forma de los datos, lo que lo hace útil en situaciones donde la relación entre datos no es bien conocida.

3. Alto rendimiento en *datasets* apropiados: en conjuntos de datos donde las instancias similares tienden a estar cerca unas de otras, k-NN puede ser muy eficaz.

Desventajas de k-NN

1. Alto costo computacional: Requiere calcular la distancia de una nueva instancia a todas las instancias en el conjunto de entrenamiento, lo que puede ser muy costoso computacionalmente, especialmente para *datasets* grandes.

2. Sensible a la escala de los atributos: los atributos con mayor rango numérico pueden dominar la medida de la distancia, por lo que generalmente se requiere normalización o estandarización de los datos.

3. Rendimiento en alta dimensionalidad: k-NN puede funcionar mal en *datasets* con muchos atributos, ya que la distancia en espacios de alta dimensión no es tan intuitiva.

Fundamentos de los clasificadores bayesianos

Los clasificadores bayesianos son una categoría de algoritmos de aprendizaje automático que se basan en aplicar el Teorema de Bayes para la tarea de clasificación. Utilizan la probabilidad estadística para predecir la clase o categoría de una muestra dada, basándose en la evidencia previa y las características observadas de esa muestra. Uno de sus ejemplos más comunes es el clasificador Naïve Bayes, que asume que las características de los datos son independientes entre sí dada una clase [3].

El Teorema de Bayes describe cómo actualizar nuestras creencias en la probabilidad de una hipótesis, basada en nuevas evidencias o información. La ecuación se expresa como:

$$P(A|B) = \frac{P(B|A) \times P(A)}{P(B)}$$

Donde:

- $P(A|B)$ es la probabilidad posterior de la hipótesis A dada la evidencia B.
- $P(B|A)$ es la probabilidad posterior de la evidencia B dado que la hipótesis A es cierta.
- $P(A)$ es la probabilidad a priori de la hipótesis A.
- $P(B)$ es la probabilidad total de la evidencia B.

Un clasificador bayesiano utiliza el teorema de Bayes para predecir la probabilidad de que un dato pertenezca a una clase particular.

Clasificador bayesiano Naïve Bayes

El tipo de clasificador bayesiano más común es el Naïve Bayes, que supone independencia entre los atributos. De acuerdo con [4], la probabilidad de que una muestra pertenezca a una clase se calcula como:

$$P(C_k|x) = \frac{P(x|C_k) \times P(C_k)}{P(x)}$$

Donde:

- $P(C_k|x)$ es la probabilidad de que la instancia x pertenezca a la clase C_k.
- $P(x|C_k)$ se calcula como el producto de las probabilidades de cada atributo individual x_i dado C_k, suponiendo su independencia.
- $P(C_k)$ es la probabilidad a priori de la clase C_k.
- $P(x)$ es un factor de normalización que se calcula sumando las probabilidades de x sobre todas las clases posibles.

Para un conjunto de atributos independientes $x_1, x_2, \dots, x_n$, la probabilidad de la instancia x en la clase C_k se calcula como:

$$P(C_k|x) = \frac{P(C_k) \times \prod_{i=1}^{n} P(x_i|C_k)}{P(x)}$$

Algunas observaciones:

- Las probabilidades condicionales $P(x_i|C_k)$ y las probabilidades a priori $P(C_k)$ se estiman a partir de las instancias del conjunto de entrenamiento.
- Para evitar la probabilidad cero en el caso de que un atributo no aparezca en los datos de entrenamiento, se utiliza un enfoque de suavizado como la corrección de Laplace.

Según [3], se tienen algunas de las siguientes variantes de Naïve Bayes:

- Naïve Bayes gaussiano:
    - Utilizado cuando los atributos son continuos.
    - Asume que los atributos para cada clase siguen una distribución normal (gaussiana).
    - La probabilidad $P(x_i|C_k)$ se calcula usando la función de densidad de probabilidad gaussiana:

$$P(x_i|C_k) = \frac{1}{\sqrt{2\pi\sigma_{C_k}^2}} \exp\left(-\frac{(x_i - \mu C_k)^2}{2\sigma_{C_k}^2}\right)$$

Donde μC_k y $\sigma_{C_k}^2$ son la media y la varianza de la característica x_i para la clase C_k.

- Naïve Bayes multinomial:
    - Adecuado para atributos discretos como la frecuencia de palabras en la clasificación de textos.
    - La probabilidad $P(x_i|C_k)$ se calcula como la frecuencia relativa del atributo x_i en la clase C_k.
- **Naïve Bayes de Bernoulli:**
    - Utilizado para atributos binarios (presencia/ausencia de una característica).
    - Similar al Naïve Bayes Multinomial pero diseñado para entradas binarias.
    - La probabilidad $P(x_i|C_k)$ se calcula como la frecuencia relativa del atributo x_i en la clase C_k.

Funcionamiento de Naïve Bayes gaussiano.

Según [4] Naïve Bayes se describe matemáticamente de la siguiente manera:

El Naïve (ingenuo) en Naïve Bayes se refiere a la suposición de que todos los atributos son independientes entre sí, dado el valor de la clase. Esto simplifica el cálculo de $P(x|C_k)$ que se convierte en el producto de las probabilidades individuales de cada atributo:

$$P(x|C_k) = \prod_{i=1}^{n} P(x_i|C_k)$$

Para cada atributo x_i y cada clase C_k , se supone que x_i sigue una distribución gaussiana. Esto implica calcular la media μC_k y la desviación estándar σC_k de x_i para cada clase C_k durante la fase de entrenamiento. La probabilidad $P(x_i|C_k)$ se calcula entonces mediante la función de densidad de la distribución normal:

$$P(x_i|C_k) = \frac{1}{\sqrt{2\pi\sigma_{C_k}^2}} \exp\left(-\frac{(x_i - \mu C_k)^2}{2\sigma_{C_k}^2}\right)$$

Durante la fase de entrenamiento:

Se calcula $P(C_k)$ para para cada clase como la frecuencia relativa de la clase en el conjunto de entrenamiento.

Para cada atributo y clase, se calculan μC_k y σC_k a partir de las instancias de entrenamiento.

Para una nueva instancia con un vector de atributos (instancia) x, se calcula la probabilidad posterior para cada clase C_k y se selecciona la clase con la mayor probabilidad posterior:

$$\hat{C} = \arg max_{C_k} P(C_k|x)$$

Donde $\hat{C}$ es la clase predicha para la instancia.

Árboles de decisión

Un árbol de decisión es un algoritmo de aprendizaje supervisado no paramétrico, empleado en tareas de clasificación y regresión. Presenta una estructura jerárquica similar a un árbol, compuesta por un nodo raíz, ramificaciones, nodos internos y nodos hoja [5].

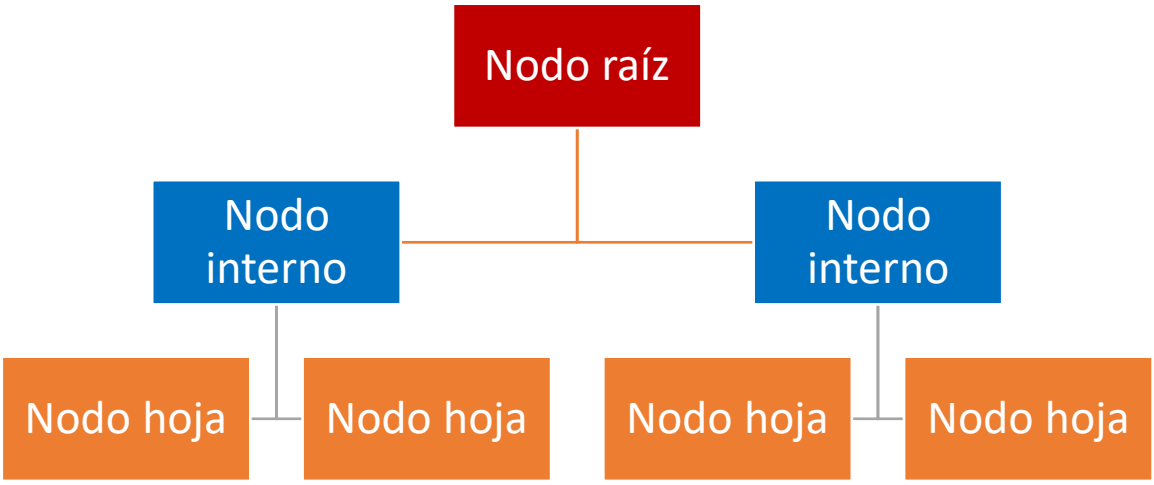

Figura 2: estructura del árbol de decisión.

Observando el diagrama anterior, se aprecia que un árbol de decisión comienza con un nodo raíz, que no tiene ramas entrantes. Las ramas salientes de este nodo raíz se dirigen a los nodos internos, conocidos como nodos de decisión. Estos nodos evalúan las características disponibles para crear subgrupos homogéneos, representados por los nodos hoja o terminales. Los nodos hoja simbolizan todas las salidas posibles en el *dataset*.

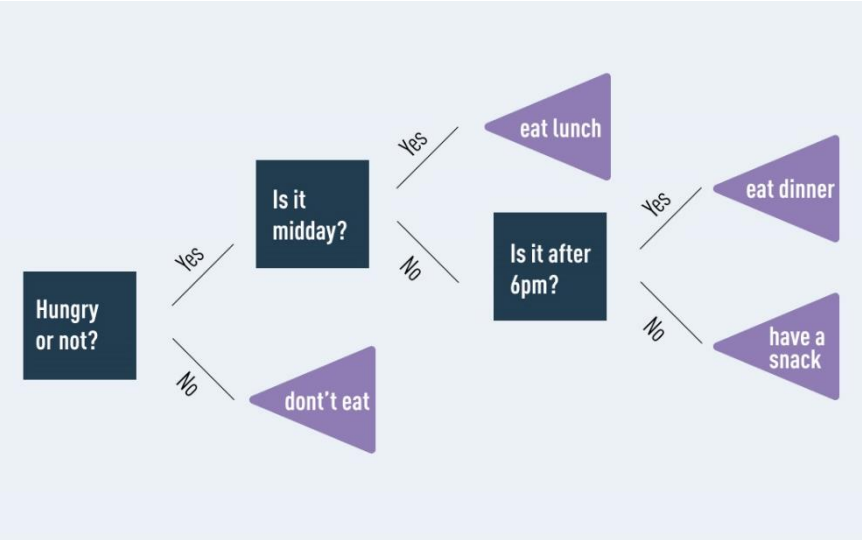

Figura 3: los árboles de decisión se utilizan para ayudar a las personas o las organizaciones a tomar decisiones lógicas y basadas en datos al seguir un conjunto de reglas predefinidas.

En el aprendizaje basado en árboles de decisión, se aplica una estrategia efectiva de divide y vencerás. Esto implica realizar una búsqueda minuciosa para encontrar los puntos de división óptimos en el árbol. Este proceso, que se ejecuta de forma descendente y recursiva, continúa hasta clasificar todos o la mayoría de las instancias en clases. La homogeneidad en la clasificación de las instancias, y si estos se agrupan en conjuntos uniformes, depende en gran medida de la complejidad del árbol. Los árboles más pequeños tienden a lograr nodos hoja puros, es decir, instancias agrupadas en una sola clase. Sin embargo, a medida que un árbol crece en tamaño, mantener esta homogeneidad se vuelve más desafiante, lo que puede resultar en una distribución insuficiente de datos en subárboles específicos, un fenómeno conocido como fragmentación de instancias, que a menudo conduce al sobreajuste [5].

Por esta razón, se prefiere la utilización de árboles de decisión más pequeños, alineándose con el principio de simplicidad de la navaja de Ockham, que aboga por evitar complicaciones innecesarias. Los árboles de decisión, por lo tanto, deberían aumentar su complejidad solo cuando sea imprescindible, siguiendo la premisa de que la explicación más simple suele ser la más adecuada. Para controlar la complejidad y prevenir el sobreajuste, se emplea la técnica de poda, que consiste en eliminar las ramas que se basan en atributos de menor importancia.

Tipos de árboles de decisión

Existen varios tipos de árboles de decisión, cada uno con sus propias características. A continuación, se detallan tres algoritmos de árboles de decisión particularmente influyentes: ID3, C4.5 y CART, que se basan en el algoritmo de Hunt desarrollado en la década de 1960 [5].

3. **ID3** (Iterative Dichotomiser 3)

    - Desarrollado por Ross Quinlan en los años 80, el ID3 representa un avance significativo en la modelización de árboles de decisión.
    - Utiliza conceptos de entropía y ganancia de información, derivados de la teoría de la información, para seleccionar el mejor atributo en cada nodo del árbol.
    - En cada nodo del árbol, el algoritmo elige el atributo que ofrece la máxima ganancia de información, es decir, aquel que mejor divide el conjunto de datos en subconjuntos más homogéneos.
    - Aunque eficaz para conjuntos de datos categóricos, el ID3 tiene limitaciones en el manejo de atributos continuos y valores faltantes.

2. **C4.5**

    - Desarrollado también por Ross Quinlan, el C4.5 es una iteración mejorada del ID3.
    - Introduce la capacidad de trabajar con atributos tanto categóricos como numéricos. Implementa métodos para manejar valores faltantes y un sistema de poda para reducir el tamaño del árbol y mejorar su generalización.
    - El C4.5 puede utilizar la ganancia de información o la razón de ganancia (ganancia de información normalizada) para seleccionar atributos, lo que permite un balance más justo, especialmente en casos donde los atributos tienen un número desigual de valores.

3. **CART** (Classification and Regression Trees)

    - Introducido por Leo Breiman, el CART es único por su capacidad de construir árboles tanto para clasificación como para regresión.
    - Utiliza la impureza de Gini o índice de Gini como criterio para seleccionar atributos en tareas de clasificación. La impureza de Gini mide la frecuencia con la que un atributo clasifica incorrectamente si se elige al azar, prefiriendo los atributos con menor impureza.

- Los árboles generados por CART son binarios, lo que significa que cada nodo se divide en exactamente dos nodos hijos.
- Implementa un enfoque de poda para mejorar la generalización del modelo, construyendo primero un árbol grande y luego eliminando secciones que no aportan a la precisión.

Elección del mejor atributo en cada nodo

Existen varias técnicas para identificar el atributo más adecuado en cada nodo de un árbol de decisión. Entre estas, la ganancia de información y la impureza de Gini son criterios destacados para la división en los modelos de árboles de decisión. Estos métodos son cruciales para evaluar la eficacia de cada condición de prueba y su capacidad para clasificar correctamente las muestras en categorías específicas.

Entropía y ganancia de información

Para comprender la ganancia de información, es esencial primero entender la entropía. La entropía, un término originado en la teoría de la información, mide la variabilidad o impureza de los valores en un conjunto de muestras [6]. La fórmula para calcular la entropía es la siguiente:

$$Entropía(S) = -\sum_{c \in C} p(c) \log_2 p(c)$$

Donde:

- S representa el conjunto de datos
- c las clases dentro de S
- $p(c)$ la proporción de puntos de datos que pertenecen a la clase c en relación con el total de puntos en S.

Los valores de entropía pueden variar de 0 a 1. Una entropía de 0 significa que todas las instancias en S pertenecen a una sola clase, mientras que un valor de 1 indica una distribución equitativa entre dos clases.

Para seleccionar el mejor atributo de división y desarrollar el árbol de decisión óptimo, se debe elegir el atributo que produce la mayor reducción en la entropía, conocida como ganancia de información. La ganancia de información mide cuánto disminuye la entropía antes y después de dividir el conjunto de datos basándose en un atributo específico. El atributo que logra la mayor

ganancia de información es el más eficaz para dividir los datos de acuerdo con la clasificación deseada. La ganancia de información se calcula generalmente con la fórmula:

$$Ganancia\ de\ información(S, \alpha) = Entropía(S) - \sum_{v \in valores(\alpha)} \frac{|S_V|}{|S|} Entropía(S_V)$$

Donde:

- α indica un atributo específico o la clase
- $Entropía(S)$ la entropía del *dataset* S,
- $\frac{|S_V|}{|S|}$ la proporción de valores en el subconjunto S_V con respecto al total en S.
- $Entropía(S)$ es la entropía del *dataset*, S_V.

Para ejemplificar el concepto de entropía, vamos a abordar el siguiente ejemplo sencillo:

Día	**Ruta**	**Hora pico**	**Tráfico**	**Mantenimiento**	**A tiempo**
1	Norte	Sí	Alto	Regular	No
2	Norte	Sí	Alto	Urgente	No
3	Sur	Sí	Alto	Regular	Sí
4	Este	No	Medio	Ninguno	Sí
5	Oeste	No	Bajo	Regular	Sí
6	Oeste	No	Bajo	Urgente	No
7	Sur	No	Bajo	Regular	Sí

Tabla 1: ejemplo de dataset para aplicar el concepto de entropía.

Para este conjunto de datos, calculamos la entropía para "A tiempo":

Supongamos que la proporción de "Sí" es 4/7 y la de "No" es 3/7.

La entropía se calcula como:

$$Entropía(Atiempo) = -\left(\frac{4}{7}\right) log_2\left(\frac{4}{7}\right) - \left(\frac{3}{7}\right) log_2\left(\frac{3}{7}\right) = 0.985$$

Luego, calculamos la ganancia de información para cada atributo. Por ejemplo, para "Tráfico":

Supongamos que 4 de los 7 días el tráfico es "Alto", y 3 de los 7 días es "Medio" o "Bajo".

Calculamos la entropía para "Tráfico = Alto" y "Tráfico ≠ Alto".

La ganancia de información para "Tráfico" sería:

$Ganancia(ATiempo, Tráfico)$

$$= 0.985 - \left(\frac{4}{7}\right) \times Entropía(Tráfico = Alto) - \left(\frac{3}{7}\right) \times Entropía(Tráfico \neq Alto)$$

Resumiendo:

- 4/7 representa la proporción de días con tráfico "Alto".
- Calculamos la entropía para cada categoría de tráfico y su contribución a la ganancia de información.
- Repetimos el cálculo para "HoraPico", "Ruta", "Mantenimiento" y elegimos el atributo con la mayor ganancia de información como el primer punto de división. Supongamos que resulta ser "Ruta".
- Continuamos este proceso para cada subdivisión del árbol.
-

Impureza de Gini

La impureza de Gini o índice de Gini es un concepto fundamental en la construcción de árboles de decisión, especialmente en el contexto del algoritmo CART. Se utiliza para medir qué tan a menudo se clasificaría incorrectamente una instancia aleatoria si se etiquetara de manera aleatoria según la distribución de clases en el conjunto de datos [6].

Este concepto representa la probabilidad de que una instancia sea clasificada erróneamente si se asignara una etiqueta basándose en las proporciones de las clases presentes en el conjunto de datos.

$$Gini(S) = 1 - \sum_i (p_i)^2$$

donde p_i es la proporción de puntos de datos en la clase i dentro del conjunto S.

Aplicación en árboles de decisión

- Selección de atributos: Al construir un árbol de decisión, la impureza de Gini se utiliza para evaluar los atributos y determinar cuál proporcionará la división más significativa del *dataset*.
- Minimización de la impureza: El objetivo es elegir el atributo que resulta en la menor impureza de Gini después de la división, lo que indica una clasificación más clara y distintiva entre las clases.

Comparación con la entropía

- Al igual que la entropía, la Impureza de Gini mide la heterogeneidad de un conjunto de datos. Ambas métricas son utilizadas para evaluar la calidad de las divisiones en un árbol de decisión.
- Aunque similar en propósito a la entropía, la impureza de Gini a menudo es computacionalmente más simple de calcular, ya que no implica operaciones logarítmicas.

Random Forest

El algoritmo Random Forest, ampliamente reconocido en el campo del aprendizaje automático y creado por Leo Breiman y Adele Cutler, se destaca por su método de combinar los resultados de múltiples árboles de decisión para obtener un resultado unificado. Este algoritmo es especialmente apreciado por su facilidad de uso y versatilidad, lo que le permite abordar con eficacia tanto problemas de regresión como de clasificación [7].

Antes de comprender cómo funciona el algoritmo de Random Forest, es importante entender lo que es un *ensemble*. Esta palabra en el contexto del aprendizaje automático hace referencia a la combinación de múltiples modelos en el aprendizaje automático. De esta manera, una colección de modelos podría realizar mejores predicciones que un modelo individual.

De acuerdo con [8], el *ensemble* utiliza dos tipos de métodos:

- *Bagging*, también conocido como agregación de *boostrap* es utilizado en Random Forest.
- *Boosting* es utilizado en AdaBoost y XGBoost.

El método *Boosting* se detallada en la siguiente sección referente a *AdaBoost*.

En esta sección, se explica en que consiste el método *Bagging*:

1. **Selección de subconjuntos:** El proceso comienza seleccionando de forma aleatoria una muestra o subconjunto del *dataset* completo.
2. **Muestreo Bootstrap:** Se crean los modelos a partir de estas muestras, llamadas muestras Bootstrap, obtenidas del conjunto de datos original con reemplazo, un proceso conocido como muestreo de filas.
3. **Bootstraping:** Este muestreo de filas con reemplazo constituye el bootstrapping.
4. **Entrenamiento de modelos de forma independiente**: Cada modelo, que es un árbol de decisión, se entrena de manera independiente con su respectiva muestra Bootstrap. Al emplear diferentes muestras para cada árbol, se garantiza la diversidad en el modelo de Random Forest, lo que contribuye a un mejor rendimiento y robustez del modelo general.

5. **Votación mayoritaria:** La salida final se determina combinando los resultados de todos los modelos (árboles de decisión) a través de una votación mayoritaria. El resultado final más frecuentemente predicho por los modelos es el elegido.
6. **Agregación:** Esta fase implica combinar todos los resultados y generar la salida final basada en la votación mayoritaria.

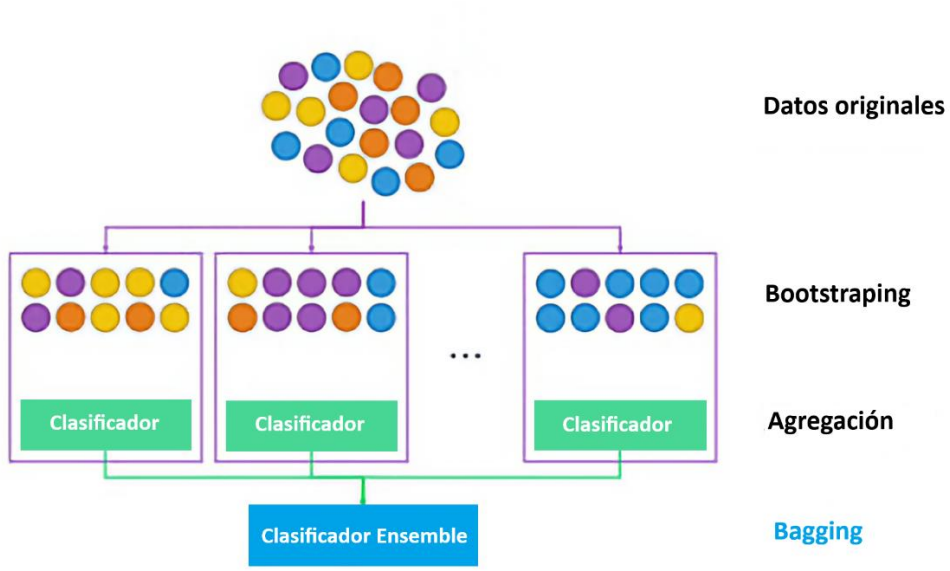

Figura 4: ejemplo de Bagging.

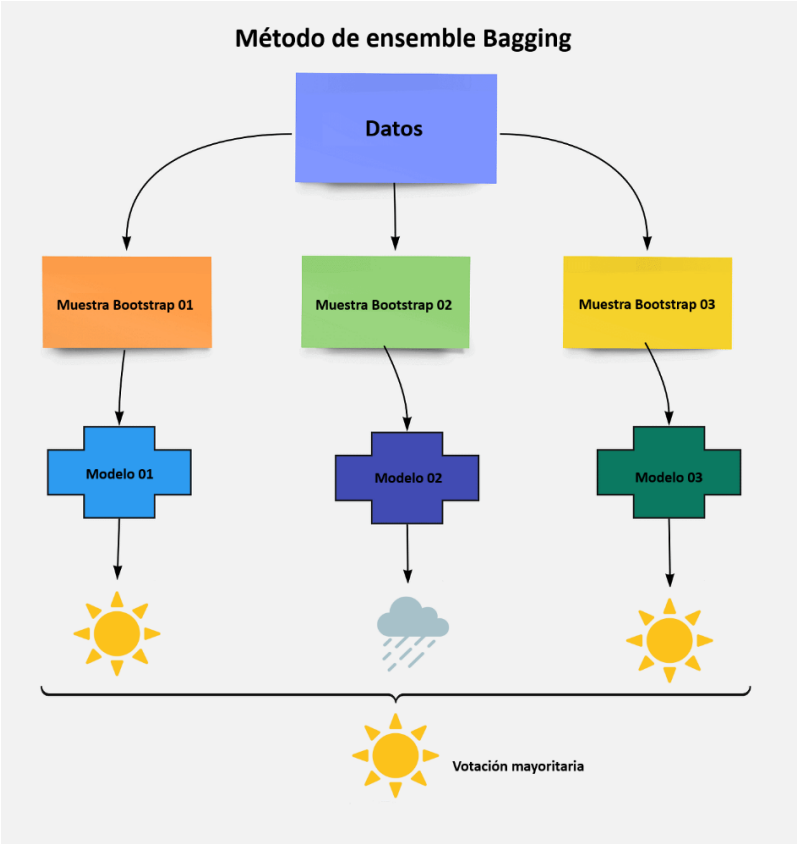

Figura 5: votación mayoritaria en Bagging.

La figura X muestra un ejemplo de *bootstrap* que considera un *dataset* cuya finalidad sirve para clasificar el estado del tiempo (muestra *bootstrap* 01, muestra *bootstrap* 02 y muestra *bootstrap* 03) con un reemplazo, lo que significa que hay una alta posibilidad de que cada muestra no contenga datos únicos. El modelo (Modelo 01, Modelo 02 y Modelo 03) obtenido a partir de esta muestra *bootstrap* se entrena de forma independiente. Cada modelo genera los resultados que se muestran: día soleado, día lluvioso, día soleado. Basándose en la votación por mayoría, el resultado final es día soleado.

Los pasos del algoritmo de acuerdo con [7] y [8] son:

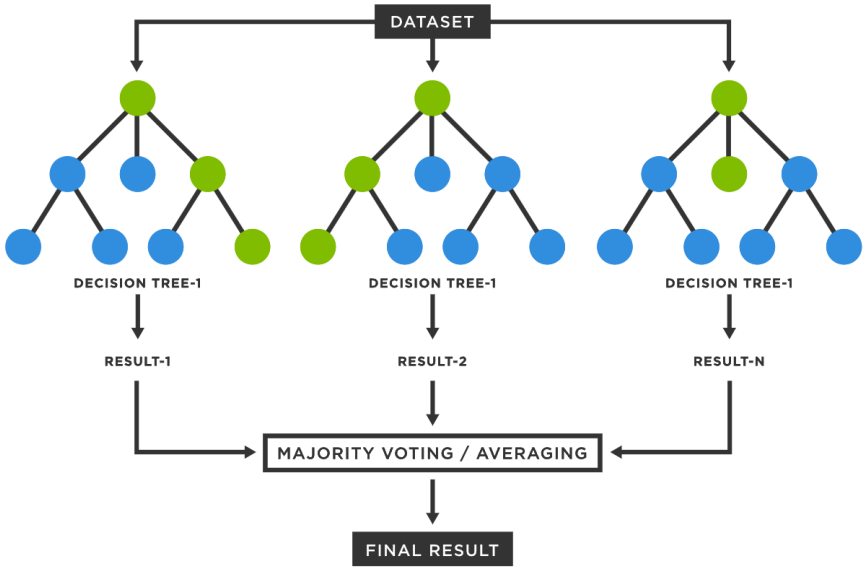

Figura 6: proceso de Random Forest

- **Paso 1: selección de subconjuntos de instancias y atributos**

  El diagrama muestra múltiples árboles de decisión (Tree-1, Tree-2, ..., Tree-n), cada uno representando un modelo individual dentro del ensamble de Random Forest. Aunque no se muestra explícitamente en el diagrama, cada árbol se construye a partir de un subconjunto de instancias y atributos seleccionados aleatoriamente del *dataset* completo, que es el primer paso del proceso. Esta selección es lo que garantiza que cada árbol tenga una "perspectiva única" del problema, reduciendo la correlación entre los árboles y aumentando la robustez del modelo.

- **Paso 2: construcción de árboles de decisión individuales**

  Cada árbol en el diagrama representa un modelo construido a partir de su propio subconjunto de instancias, como se describió en el segundo paso. Aunque el diagrama simplifica la estructura de los árboles, en la práctica, cada árbol se desarrolla de manera compleja, ramificándose en diferentes puntos según los atributos y las instancias específicas de su subconjunto.

- **Paso 3: generación de una salida por cada árbol de decisión**

  En el diagrama, cada árbol produce un resultado individual (Class-A, Class-B), lo que corresponde al tercer paso. Este resultado es la predicción del árbol basada en la

información que ha aprendido de su subconjunto de instancias. En un problema de clasificación real, esto podría ser una etiqueta de clase como "spam" o "no spam", "enfermo" o "saludable", y así sucesivamente.

- **Paso 4: Decisión final basada en la votación mayoritaria**
  Finalmente, como se ilustra en el diagrama y se menciona en el cuarto paso, todas las salidas individuales de los árboles se combinan para llegar a una decisión final. En el diagrama, esto se muestra como "Majority-Voting" que lleva a "Final-Class". La clase final se determina por la mayoría de los votos entre todas las predicciones de los árboles. Si la mayoría de los árboles predicen Class-B, entonces esa será la clasificación final asignada a la instancia de entrada.
  El diagrama visualiza el proceso descrito, destacando la naturaleza colaborativa de los árboles de decisión dentro del modelo de Random Forest y cómo sus predicciones colectivas determinan la clase final de una instancia dada a través de la votación mayoritaria.

Las ventajas y desventajas del clasificador según [9] son las siguientes:

Ventajas de Random Forest

1. Robustez: al ser un ensamble de árboles de decisión, Random Forest es generalmente más robusto y con mejor rendimiento que un árbol de decisión único, ya que reduce la varianza sin aumentar sustancialmente el sesgo.
2. Manejo de sobreajuste: a través del mecanismo de promedio o votación mayoritaria, Random Forest puede mitigar el sobreajuste, un problema común en los árboles de decisión individuales, especialmente cuando son muy profundos.
3. Flexibilidad: puede ser utilizado para tareas de clasificación y regresión, mostrando un buen rendimiento en ambas.
4. Manejo de datos no balanceados: tiene la capacidad de manejar instancias del conjunto de entrenamiento con un balance de clases desigual.
5. Importancia de las características: proporciona medidas intuitivas de la importancia de las características, lo que puede ser útil para la selección de atributos y la interpretación del modelo.

6. Menos preprocesamiento de datos: no requiere escalado de atributos, y puede manejar tanto variables categóricas como numéricas.
7. Manejo de valores faltantes: puede manejar datos con valores faltantes, aunque la implementación específica puede variar.
8. Paralelización: se presta bien para la paralelización, permitiendo un uso eficiente de los recursos computacionales y reduciendo el tiempo de entrenamiento.

Desventajas de Random Forest

1. Interpretabilidad: a diferencia de un árbol de decisión simple, los modelos de Random Forest son más difíciles de interpretar debido a su naturaleza de ensamble.
2. Tiempo de entrenamiento: a pesar de que se puede paralelizar, el entrenamiento de múltiples árboles puede ser computacionalmente costoso, especialmente con grandes *datasets*.
3. Complejidad del modelo: puede ser menos eficiente en términos de memoria y predicción en tiempo real, debido a la cantidad de árboles y la profundidad de estos.
4. Tendencia al sobreajuste con datos muy ruidosos: aunque maneja bien el sobreajuste en general, en situaciones con ruido extremo y variables irrelevantes, puede sobreajustarse.
5. Actualizaciones del modelo: no es tan fácil de actualizar como otros modelos; añadir nuevos datos puede requerir una reconstrucción completa del modelo, o al menos de una parte significativa de los árboles.
6. Pérdida de detalle en las clasificaciones: para datos con límites de decisión muy complejos o ruido, Random Forest puede perder algunos detalles.
7. Resultados aleatorios: la aleatoriedad en la construcción de árboles puede conducir a variabilidad en los resultados, lo que significa que pequeñas alteraciones en los datos pueden cambiar el modelo.
8. Rendimiento en datos de alta dimensionalidad: aunque maneja bien la "maldición de la dimensionalidad" hasta cierto punto, el rendimiento puede degradarse con un número extremadamente alto de atributos debido a la dilución del efecto de las características importantes.

AdaBoost

AdaBoost, cuyo nombre completo es "Adaptive Boosting", es un algoritmo de aprendizaje automático fundamentalmente ligado al concepto de métodos de ensemble, específicamente al "boosting".

Boosting es otra de las técnicas de ensemble además del bagging que busca crear un *strong learner* o a veces referido en español como clasificador fuerte a partir de la combinación de varios *weak learners* o clasificadores débiles, donde los *weak learners* son mejores que la clasificación aleatoria y el *strong learner* incorpora la salida de todos los clasificadores débiles. A cada *weak learner* se le asigna un peso en función de la calidad de sus resultados. Cuanto mejor sea un *weak learner*, mayor será el peso que se le asigne [10].

Se dice que AdaBoost es adaptativo en el sentido de que los clasificadores se ajustan en función de los errores de cálculo de clasificadores anteriores.

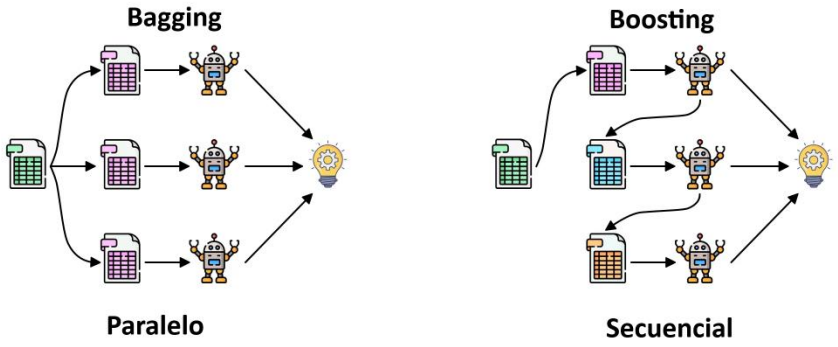

Figura 7: paralelización en bagging y secuenciación en boosting.

Según [12] las diferencias entre los métodos de ensemble Bagging y Boosting:

1. Bagging:

- Proceso en paralelo: se muestra que varios clasificadores (representados por robots) son entrenados en paralelo, cada uno con una versión diferente del conjunto de entrenamiento.
- Conjuntos de datos independientes: estos conjuntos de datos se crean a través de muestreo aleatorio con reemplazo (*bootstrap sampling*) del *dataset* original, lo que puede resultar en subconjuntos con algunas muestras repetidas.

- Agregación de resultados: los resultados de los clasificadores individuales se combinan, a menudo mediante votación mayoritaria, para formar un resultado final.

2. Boosting:

- Proceso secuencial: a diferencia de Bagging, el Boosting entrena clasificadores de forma secuencial.
- Ponderación de errores: cada clasificador subsiguiente se enfoca en los errores del clasificador anterior, asignando más peso a las muestras que fueron clasificadas incorrectamente.
- Combinación ponderada: el resultado final es una combinación ponderada de las clasificaciones de todos los clasificadores, donde cada clasificador aporta en función de su rendimiento.

La imagen resalta la naturaleza paralela del Bagging, donde cada clasificador se entrena independientemente de los demás, y la naturaleza secuencial del Boosting, donde el rendimiento de cada clasificador influye en el entrenamiento del siguiente. Esto visualiza la estrategia principal detrás de cada método: Bagging reduce la varianza y el overfitting al promediar múltiples estimaciones, mientras que Boosting intenta reducir el sesgo al corregir los errores de clasificación de forma iterativa.

En AdaBoost, los *weak learners* suelen ser *stumps*, pero no se limitan exclusivamente a ellos. Un *stump* es una forma de árbol de decisión muy simple, que básicamente consta de una sola decisión o división, es decir, tiene una profundidad de 1. Estos *stumps* son populares en AdaBoost debido a su simplicidad y efectividad para el propósito del algoritmo [12].

Sin embargo, es importante señalar que AdaBoost es un algoritmo flexible en términos de los tipos de clasificadores débiles que puede utilizar. Aunque los *stumps* son comunes, también se pueden usar otros tipos de clasificadores débiles, siempre y cuando cumplan con el criterio fundamental de ser solo ligeramente mejores que el azar. Esto podría incluir, por ejemplo, pequeños árboles de decisión con unas pocas divisiones, clasificadores lineales simples u otros modelos básicos.

Funcionamiento de AdaBoost

De acuerdo con [10], [11], [12] AdaBoost funciona de la siguiente manera:

- AdaBoost inicia su procedimiento asignando un peso inicial igual a todas las muestras en el conjunto de entrenamiento. Este peso, establecido en $w_i = \frac{1}{N}$ para cada muestra i, donde N es el número total de muestras, asegura que inicialmente todas las muestras tengan igual relevancia en el entrenamiento del primer *weak learner*.
- A lo largo de sus iteraciones, AdaBoost se enfoca en entrenar una serie de clasificadores débiles. Estos $learners$ son simples por diseño, con la idea de que individualmente solo sean ligeramente mejores que un clasificador aleatorio. En cada iteración, el algoritmo entrena un *weak learner* h_t y calcula su error ponderado ϵ_t mediante $\epsilon_t = \frac{\sum_{i=1}^{N} w_i \cdot error(i)}{\sum_{i=1}^{N} w_i}$. Aquí, $error(i)$ es 1 si la muestra i es incorrectamente clasificada y 0 si es correcta. La importancia del clasificador, o Alpha (α_t), se determina a partir de este error con la fórmula $\alpha_t = \frac{1}{2} \ln\left(\frac{1-\epsilon_t}{\epsilon_t}\right)$.
- El siguiente paso crucial es la actualización de los pesos de las muestras. AdaBoost incrementa los pesos de las muestras mal clasificadas usando $w_i \leftarrow w_i \cdot \exp\left(\alpha_t \cdot error(i)\right)$, haciendo que estos casos sean más influyentes en la siguiente iteración. Esto refleja la naturaleza adaptativa del algoritmo, donde se pone mayor énfasis en los errores cometidos anteriormente.
- Después de cada iteración, se normalizan los pesos de las muestras para que sumen 1, manteniendo la coherencia en la distribución de los pesos para las iteraciones subsiguientes.
- Finalmente, el *strong learner* de AdaBoost se forma al combinar todos los *weak learners* entrenados. En este modelo final, cada *weak learner* contribuye con una votación ponderada a la clasificación de nuevas muestras, siendo esta ponderación proporcional a su valor de Alpha. Matemáticamente, el $strong\ learner$ se define como $H(x) = sign(\sum_{t=1}^{T} \alpha_t h_t(x))$.

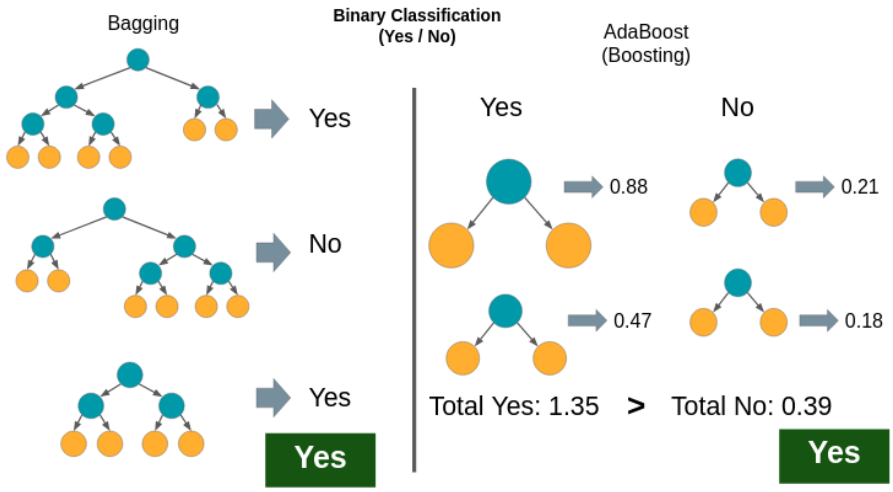

Figura 8: diferencias entre bagging y boosting.

- Proceso de *Bagging*: se muestran tres árboles de decisión que representan clasificadores individuales. Cada uno ha sido entrenado en un subconjunto diferente del conjunto de datos, lo que es característico del muestreo con reemplazo utilizado en *bagging*.

  La decisión final se toma por votación mayoritaria. Dos árboles votan "Sí" y uno vota "No", resultando en una clasificación final de "Sí".

- Proceso de AdaBoost (Boosting): se ilustran tres clasificadores con pesos asignados a sus decisiones. Estos pesos se derivan del rendimiento de cada clasificador en las iteraciones de entrenamiento de AdaBoost.

  Las decisiones de los clasificadores se suman de manera ponderada, dando como resultado un total de "Sí" de 1.35 y un total de "No" de 0.39. Debido a que el total de "Sí" es mayor, la clasificación final es "Sí".

Las ventajas y desventajas de AdaBoost de acuerdo con [12] son las siguientes:

Ventajas de AdaBoost:

1. Combina clasificadores débiles para formar un modelo más fuerte y preciso.

2. Identifica y utiliza las características más relevantes.

4. Menor tendencia al sobreajuste en comparación con otros algoritmos de aprendizaje profundo o complejo.

Desventajas de AdaBoost:

1. Los datos erróneos o atípicos pueden llevar al algoritmo a ajustarse demasiado a estos.

2. Puede sobreajustar en *datasets* con clases desproporcionadas.

3. No es óptimo para *datasets* con muchas características en relación con el número de muestras.

Support Vector Machine (SVM)

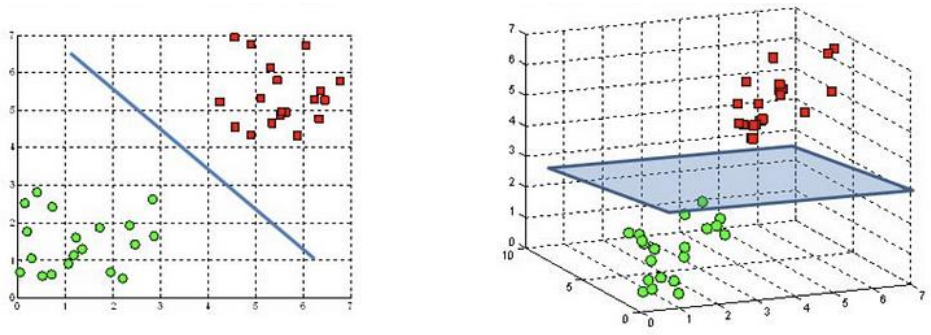

Figura 9: hiperplanos que dividen a los espacios 2D y 3D.

Una máquina de soporte vectorial o Support Vector Machine (SVM, por sus siglas en inglés) es un método de aprendizaje supervisado utilizado para la clasificación y regresión de patrones. La idea principal detrás de una SVM tradicional para clasificación es encontrar un hiperplano (una línea en 2D, un plano en 3D, etc.) que mejor separe dos clases de datos [13]. [15] remarca algunos puntos clave sobre cómo funciona una SVM para la clasificación de patrones:

- En un *dataset* con dos clases, la SVM busca un hiperplano que separe lo mejor posible estas dos clases. El hiperplano se elige de manera que la distancia entre el hiperplano y el punto (que representa a una instancia) más cercano de cada clase sea máxima.
- Las instancias que están más cerca del hiperplano y que afectan la posición y orientación del hiperplano se conocen como vectores de soporte. Estos son esencialmente las instancias más críticas.

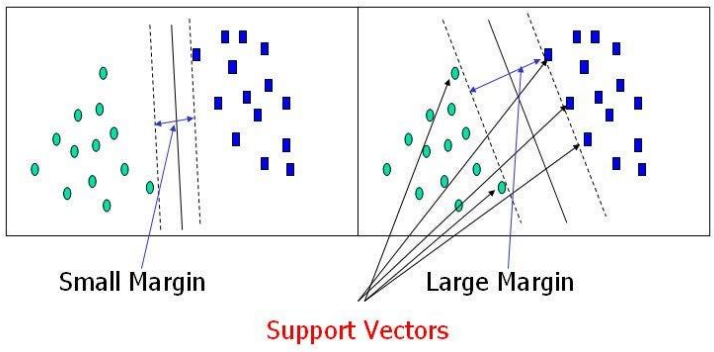

Figura 10: margen en las SVM.

- El margen se define como el espacio interpuesto entre los dos hiperplanos que se encuentran más próximos a cada una de las clases diferenciadas. Durante el proceso de entrenamiento, el objetivo primordial de la SVM es maximizar este margen. Este procedimiento es fundamental, ya que un margen ampliado incrementa significativamente la capacidad de generalización del modelo. En términos prácticos, esto implica que el modelo es más competente en clasificar instancias pertenecientes al conjunto de prueba, que no fueron expuestas al modelo durante la fase de entrenamiento. De este modo, un margen mayor es indicativo de una mejor habilidad de la SVM para adaptarse y clasificar acertadamente instancias bajo condiciones o escenarios no previamente observados durante el entrenamiento
- En casos donde las instancias no son linealmente separables en su espacio original, las SVM utilizan una función llamada "kernel" para transformarlas y llevarlas a un espacio de mayor dimensión donde es posible encontrar un hiperplano separador. Los kernels comunes incluyen lineal, polinómico, y gaussiano (RBF).
- El proceso de encontrar el hiperplano óptimo es un problema de optimización convexa. Además, las SVM tienen un parámetro de regularización que controla el *trade-off* entre lograr un margen alto y asegurarse de que las instancias de entrenamiento se clasifiquen correctamente.
- Las técnicas *one-vs-rest* y *one-vs-one* permiten extender las SVM a *datasets* que tienen más de dos clases.

Una vez que se tiene una idea general de la SVM, se procede a describirla de manera más formal. Para simplificar, a continuación, se considera el caso de una SVM lineal para clasificación binaria [13].

1. **Definición del hiperplano**

   Un hiperplano en un espacio $n - dimensional$ se define como el conjunto de puntos x que satisfacen la ecuación:

   $$w \cdot x + b = 0$$

   Donde:

   - w es el vector de pesos (normal al hiperplano).
   - x es el vector de características.

- b es el sesgo.

2. **Clasificación de datos**

   Para una instancia x_i, la clasificación se realiza utilizando la función de decisión:

   $$f(x_i) = sign(w \cdot x_i + b)$$

   Donde:

   $sign$ es una función que devuelve $+1$ o -1, representando las dos clases.

3. **Margen y vectores de soporte**

   El margen se define como la distancia entre los vectores de soporte más cercanos al hiperplano. Para un hiperplano óptimo, los vectores de soporte x_i satisfacen: $y_i(w \cdot x_i + b) = 1$.

   Donde:

   y_i es la etiqueta de clase del vector de soporte $(+1\ o\ -1)$.

4. **Problema de optimización**

   El objetivo es maximizar el margen, que es $\frac{2}{\|x\|}$, equivalente a minimizar $\frac{1}{2} \| w \|^2$. El problema de optimización se formula como: $min_{w,b} \frac{1}{2} \| w \|^2$ sujeto a las restricciones: $y_i(w \cdot x_i + b) \geq 1, \forall i$.

5. **Multiplicadores de Lagrange**

   Para resolver este problema, se introducen los multiplicadores de Lagrange α_i para cada restricción, llevando al Lagrangiano:

   $$L(w, b, \alpha) = \frac{1}{2} \| w \|^2 - \sum_{i=1}^{n} \alpha_i [y_i(w \cdot x_i + b) - 1]$$

   Donde n es el número de instancias de entrenamiento.

6. **Condiciones de Karush-Kuhn-Tucker (KKT)**

   Al minimizar con respecto a w y b, y maximizar con respecto a α, se obtienen las condiciones de KKT, que son necesarias para la solución óptima. Estas condiciones incluyen:

   $$w = \sum_{i=1}^{n} \alpha_i y_i x_i$$

   $$\sum_{i=1}^{n} \alpha_i y_i = 0$$

   $$\alpha_i \geq 0, \forall i$$

7. **Solución y función de decisión final**

   Después de resolver estas ecuaciones, obtenemos los valores de α, w y b. Los vectores de soporte corresponden a los datos para los cuales $\alpha_i > 0$.

   La función de decisión final, utilizada para clasificar nuevos patrones (del conjunto de prueba), se convierte en:

$$f(x) = sign\left(\sum_{i=1}^{n} \alpha_i y_i (x_i \cdot x) + b\right)$$

   Este proceso describe una SVM lineal. En casos donde los datos no son linealmente separables, se utiliza una función kernel para mapear los datos a un espacio de mayor dimensión, pero la esencia del proceso de optimización y clasificación permanece similar.

Kernel trick

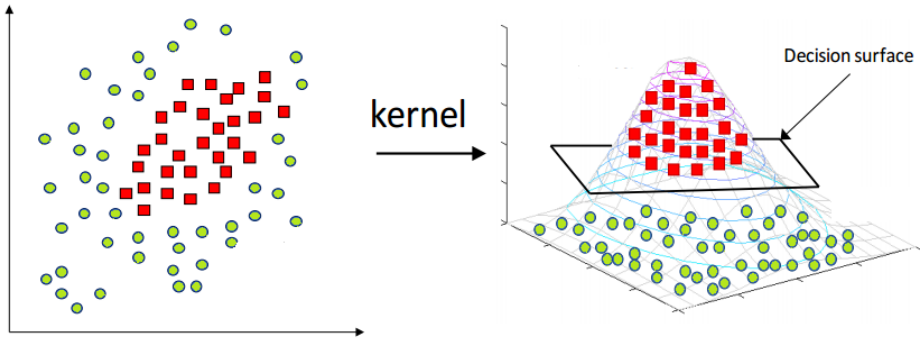

Figura 11: el truco del Kernel o Kernel Trick.

El "kernel trick" se utiliza en situaciones donde las instancias de entrada no pueden ser separadas linealmente. En otras palabras, no existe un hiperplano en el espacio de atributos original que pueda dividir las instancias en dos clases sin errores. Esto es común en problemas reales donde las relaciones entre los atributos y las clases no son simples [14].

El kernel es una función que permite calcular la similitud (o el producto interno) entre dos puntos en un espacio de atributos transformado, sin tener que realizar explícitamente esta transformación. Esto es particularmente útil porque algunas transformaciones implican aumentar la dimensionalidad de los datos a dimensiones muy altas, lo que puede ser computacionalmente costoso.

Transformación a un espacio de mayor dimensión

Supongamos que se tiene un *dataset* que no es linealmente separable en su espacio de atributos actual. Lo que hace un kernel es mapear estos datos a un nuevo espacio de atributos (generalmente de mayor dimensión), donde los datos se vuelven linealmente separables.

Un ejemplo clásico es el kernel polinómico. Supongamos que se tiene dos atributos, x_1 y x_2. Un kernel polinómico de grado 2 transformaría estos datos a un espacio de atributosque incluye términos como x_1^2, x_2^2 y $x_1 x_2$. Esta transformación puede hacer que las instancias que no son linealmente separables en el espacio original (con solo x_1 y x_2) sean linealmente separables en este nuevo espacio.

Lo admirable del *kernel trick* radica en su eficiencia. En lugar de hacer la transformación y luego calcular el producto interno (que sería costoso en términos de cálculo), el kernel permite calcular directamente el resultado del producto interno en el espacio transformado.

De acuerdo con [14] los tipos comunes de Kernels son:

1. Lineal: no hay transformación; es equivalente a una SVM lineal.
2. Polinómico.
3. Radial Basis Function (RBF) o Gaussiano.
4. Sigmoide.

De acuerdo con [13] y [15] las ventajas y desventajas de las SVM son:

Ventajas de las SVM:

1. Eficaz en espacios de alta dimensión: las SVM funcionan bien en *datasets* con un gran número de atributos, como la clasificación de texto y de imágenes.

2. Versatilidad: a través de la función del kernel, las SVM pueden adaptarse a diferentes tipos de datos y problemas de clasificación.

3. Generalización: Debido a la maximización del margen, las SVM tienden a generalizar bien, evitando el sobreajuste en muchos casos.

Desventajas de las SVM:

1. No adecuada para grandes conjuntos de datos: al aumentar el tamaño de instancias del *dataset*, el proceso de entrenamiento de las SVM puede volverse ineficientemente lento.

2. Selección de Kernel y parámetros: elegir el kernel adecuado y ajustar los parámetros puede ser un tanto complicado y requiere experiencia.

3. Sensibilidad a los datos no balanceados: las SVM pueden ser sensibles a *datasets* donde las clases están desbalanceadas.

Multilayer Perceptron (MLP)

Los antecedentes del *Multilayer Perceptron* (MLP) o Perceptrón multicapa datan de la década de 1940 con el modelo de neurona de McCulloch y Pitts, un sistema simple que imitaba el funcionamiento de las neuronas cerebrales. Sin embargo, este modelo inicial no podía aprender por sí mismo. Este modelo, simple y lineal, imitaba la forma en que las neuronas cerebrales procesan señales eléctricas, transmitiendo información solo cuando las señales eran lo suficientemente fuertes. Este modelo inicial replicaba puertas lógicas, operando con entradas binarias y funciones booleanas activadas bajo condiciones específicas [16].

No obstante, la limitación de este modelo era su incapacidad para aprender, ya que los pesos, actuando como catalizadores en el modelo, debían ser preestablecidos. La verdadera evolución llegó con Frank Rosenblatt, quien, una década después, amplió este modelo creando el Perceptrón simple. Conceptualmente puede considerarse como la unidad fundamental de una red neuronal más compleja.

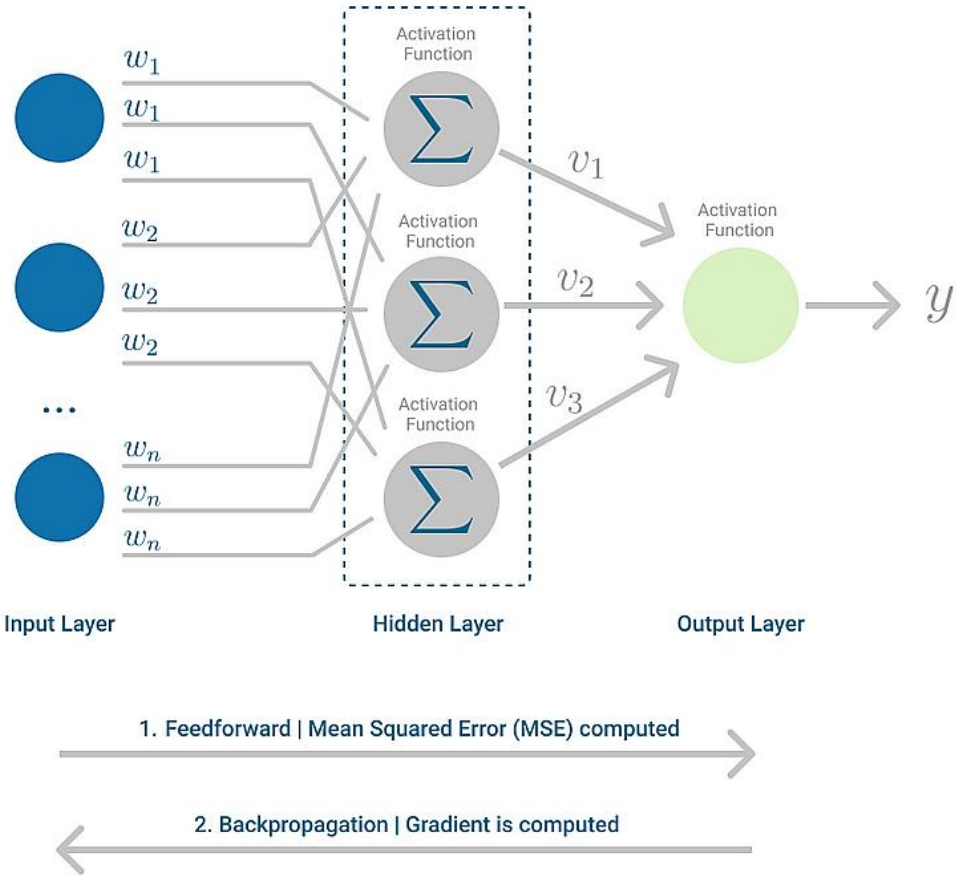

Figura 12: perceptrón multicalpa (MLP).

De acuerdo con [17] y [18] la arquitectura del perceptrón multicapa es la siguiente:

1. Arquitectura del perceptrón multicapa

La arquitectura de un perceptrón multicapa (MLP) es una disposición estructurada de neuronas dispuestas en capas, que permite procesar información de manera compleja y sofisticada. A continuación, se describen sus componentes principales:

Capa de entrada:

- Es el punto inicial donde la red recibe la entrada externa.
- Actúa como distribuidor de los datos de entrada a las capas subsiguientes.

Capas ocultas:

- Son el núcleo de un MLP donde ocurre la mayor parte del procesamiento.

- Cada capa oculta contiene un número de neuronas que transforman las entradas recibidas de la capa anterior.
- Las capas ocultas permiten al MLP capturar y aprender patrones complejos en los datos.

Capa de salida:

- Proporciona la salida final de la red, que puede ser una clasificación, una predicción, una puntuación, etc.
- El número y tipo de neuronas en esta capa dependen del problema específico que se está abordando (por ejemplo, una neurona para una tarea de clasificación binaria, múltiples neuronas para clasificación multiclase).

Neuronas y sus funciones:

- Cada neurona en las capas ocultas y de salida realiza cálculos basados en sus entradas y produce una salida.
- Las funciones de activación dentro de estas neuronas introducen no linealidades, lo que es crucial para aprender y modelar relaciones complejas en los datos.

2. Funciones de activación

Las funciones de activación en un Perceptrón Multicapa son cruciales para introducir no linealidades en el modelo, permitiendo que la red aprenda y modele relaciones complejas en los datos [17].

Tipos de funciones de activación:

1. Sigmoide: produce una salida entre 0 y 1, útil para probabilidades.
2. Tangente Hiperbólica (tanh): varía de -1 a 1, centrando las instancias.
3. ReLU (Rectified Linear Unit): proporciona salida lineal para valores positivos y cero para negativos, eficiente y efectiva para muchos casos.

Rol en las MLP:

- Las funciones de activación no lineales son esenciales para que los MLP superen la limitación de modelar solo relaciones lineales, como en el caso del perceptrón simple.
- Permiten a la red capturar y aprender patrones complejos en las instancias, lo cual es crucial para tareas como clasificación, regresión y más.

3. Proceso de feedforward

El proceso de feedforward en un perceptrón multicapa (MLP) es el flujo de información desde la entrada hasta la salida, pasando a través de las capas intermedias [18]. Aquí está cómo funciona matemáticamente:

1. Capa de entrada:
    - La red recibe el vector de entrada $\mathbb{X} = [x_1, x_2, \dots, x_n]$.
2. Capas ocultas:
    - En cada capa oculta, las entradas se procesan mediante una suma ponderada seguida de una función de activación.
    - Para una neurona j en la capa l, la operación es:

$$z_j^{(l)} = \sum_i w_{ji}^{(l)} x_i^{(l-1)} + b_j^{(l)}$$

$$a_j^{(l)} = f\left(z_j^{(l)}\right)$$

      Donde $w_{ji}^{(l)}$ y $b_j^{(l)}$ son los pesos y sesgos, $x_i^{(l-1)}$ son las salidas de la capa anterior, y f es la función de activación.

3. Capa de salida:
    - La capa final produce la salida de la red. Para una neurona de salida k: $y_k = g\left(\sum_j v_{kj}\, a_j^{(L)} + c_k\right)$.

- Donde g puede ser una función de activación distinta, v_{kj} son los pesos de la capa de salida, y c_k es el sesgo.

El proceso de feedforward es esencial para que la red haga predicciones o clasificaciones basadas en los datos de entrada. Cada capa captura diferentes niveles de abstracción y complejidad de los datos, permitiendo a la red aprender desde características simples hasta patrones más complejos.

4. Algoritmo de *backpropagation*

El algoritmo de backpropagation es un método fundamental para entrenar perceptrones multicapa (MLP). Permite ajustar los pesos y sesgos de la red en función del error en las clasificaciones. Aquí está el proceso matemáticamente de acuerdo con [18]:

1. **Cálculo del error:**
    - Después de un paso de *feedforward*, se calcula el error en la salida. Por ejemplo, para una tarea de clasificación, esto podría ser la diferencia entre la salida predicha y la etiqueta real.
    - El error se mide generalmente usando una función de coste, como la entropía cruzada o el error cuadrático medio.
2. **Propagación del error hacia atrás:**
    - Se calcula cómo cada peso y sesgo de la red contribuye al error total. Esto se hace utilizando la regla de la cadena para derivar el gradiente del error con respecto a cada parámetro.
    - Para una capa l y una neurona j, el gradiente del error con respecto al peso $w_{ji}^{(l)}$ es:

$$\frac{\partial Error}{\partial w_{ji}^{(l)}} = \frac{\partial Error}{\partial a_j^{(l)}} \cdot f'\left(z_j^{(l)}\right) \cdot x_i^{(l-1)}$$

    - Donde $f'\left(z_j^{(l)}\right)$ es la derivada de la función de activación.

3. **Actualización de pesos y sesgos:**

    - Los pesos y sesgos se actualizan en dirección opuesta al gradiente para minimizar el error. La regla de actualización es:

$$w_{ji}^{(l)} \leftarrow w_{ji}^{(l)} - \alpha \frac{\partial Error}{\partial w_{ji}^{(l)}}$$

$$b_{j}^{(l)} \leftarrow b_{j}^{(l)} - \alpha \frac{\partial Error}{\partial b_{j}^{(l)}}$$

      Donde α es la tasa de aprendizaje.

El *backpropagation* es crucial para el aprendizaje en redes neuronales profundas. Permite que la red ajuste sus pesos y sesgos de manera eficiente en respuesta al error calculado, mejorando así su capacidad para hacer predicciones o clasificaciones precisas.

4. **Optimización y aprendizaje**

En el contexto de los Perceptrones Multicapa (MLP), la optimización y el aprendizaje se refieren al proceso de ajustar los parámetros de la red (pesos y sesgos) para mejorar su rendimiento en una tarea específica. Este proceso se realiza mediante algoritmos de optimización que utilizan el gradiente del error calculado durante el *backpropagation*. Según [18] se presentan los conceptos clave:

Descenso del gradiente:

- Es el algoritmo de optimización más utilizado para entrenar MLP.
- Actualiza los pesos y sesgos en la dirección opuesta al gradiente del error para minimizar la función de coste.
- La actualización se realiza según: $w \leftarrow w - \alpha \frac{\partial Error}{\partial w}$ y $b \leftarrow b - \alpha \frac{\partial Error}{\partial b}$.

  Donde α es la tasa de aprendizaje, un hiperparámetro crítico que controla el tamaño de los pasos en la actualización.

Tasa de aprendizaje y otros hiperparámetros:

- La tasa de aprendizaje α determina cuánto se ajustan los parámetros en cada paso del entrenamiento.
- Si es demasiado alta puede causar que el algoritmo oscile o diverja, mientras que demasiado baja puede llevar a una convergencia muy lenta.
- Otros hiperparámetros incluyen el número de épocas o epochs (iteraciones sobre el *dataset* completo), el tamaño del lote (número de muestras procesadas antes de actualizar los parámetros), entre otros.

5. Funciones de coste y pérdida

Las funciones de coste y pérdida en un perceptrón multicapa (MLP) juegan un papel crucial al cuantificar el error o la diferencia entre las predicciones de la red y los valores reales o esperados. Son esenciales para guiar el proceso de entrenamiento durante el *backpropagation* y la optimización [18]. Aquí están los conceptos clave:

Tipos de funciones de coste/pérdida:

1. Error cuadrático medio (MSE): comúnmente usado para problemas de regresión.
2. Entropía cruzada: frecuentemente utilizada en tareas de clasificación.
    - Para clasificación binaria: *Binary Cross-Entropy*.
    - Para clasificación multiclase: *Cross-Entropy*.

Importancia en el entrenamiento:

- La función de coste/pérdida proporciona una medida cuantitativa del rendimiento de la red.
- Durante el entrenamiento, el objetivo es minimizar esta función, lo que indica que la red está aprendiendo a realizar predicciones más acertadas.
- El gradiente de la función de coste se utiliza en el *backpropagation* para actualizar los pesos y sesgos de la red.

Elegir la función de coste adecuada es crucial para el éxito del entrenamiento de la red, ya que diferentes problemas y tipos de datos pueden requerir diferentes enfoques para medir el error o la discrepancia entre las predicciones y los valores reales.

9. Entrenamiento del perceptrón multicapa

El entrenamiento de un perceptrón multicapa (MLP) implica ajustar sus pesos y sesgos para mejorar su capacidad de hacer predicciones precisas o clasificaciones [16] [17]. Aquí están los aspectos más importantes del proceso de entrenamiento:

Conjuntos de datos de entrenamiento, validación y prueba:

- Conjunto de entrenamiento: utilizado para entrenar el modelo. El modelo aprende ajustando sus parámetros para minimizar la función de pérdida.
- Conjunto de validación: utilizado para ajustar los hiperparámetros y evitar el sobreajuste. Permite evaluar el rendimiento del modelo durante el entrenamiento.
- Conjunto de prueba: utilizado para evaluar el rendimiento final del modelo. Es un conjunto de datos independiente que no se utiliza durante el entrenamiento o la validación.

Overfitting y métodos para evitarlo:

- Overfitting: ocurre cuando el modelo se ajusta demasiado bien a las instancias de entrenamiento, perdiendo capacidad de generalización.
- Prevención del overfitting:
    - Regularización: técnicas como L1 o L2 añaden un término de penalización a la función de coste para limitar la magnitud de los pesos.
    - Dropout: consiste en desactivar aleatoriamente neuronas durante el entrenamiento para evitar la dependencia excesiva en ciertas rutas.
    - Aumento de datos: en tareas como la visión por computadora, modificar los datos de entrenamiento puede ayudar a generalizar mejor.

Proceso de entrenamiento:

- El entrenamiento se realiza en iteraciones o épocas. En cada época, el conjunto de entrenamiento se pasa a través de la red (feedforward), se calcula el error (función de pérdida), y se ajustan los parámetros mediante backpropagation y descenso del gradiente.
- El rendimiento del modelo se monitoriza usando el conjunto de validación para hacer ajustes necesarios en los hiperparámetros.

Clasificación del dataset Hepatitis

El *dataset* Hepatitis se utiliza para clasificar si los pacientes tienen o no la enfermedad [19].

A continuación, se presenta la información general del *dataset*:

Dataset Hepatitis			
Tipo	Clasificación	**Origen**	Mundo real
Atributos	19	**(Real/Entero/Nominal)**	(2/17/0)
Clases	2	**¿Valores faltantes?**	Sí
Total de instancias	155	**Instancias sin valores faltantes**	80

Tabla 2: datos generales del dataset Hepatitis.

La siguiente tabla muestra los atributos del *dataset*, todos del tipo numérico.

Atributo	**Dominio**	**Atributo**	**Dominio**
Age	[7, 78]	**Spiders**	[1, 2]
Sex	[1, 2]	**Ascites**	[1, 2]
Steroid	[1, 2]	**Varices**	[1, 2]
Antivirals	[1, 2]	**Bilirubin**	[0.3, 8.0]
Fatigue	[1, 2]	**AlkPhosphate**	[26, 295]
Malaise	[1, 2]	**Sgot**	[14, 648]
Anorexia	[1, 2]	**AlbuMin**	[2.1, 6.4]
LiverBig	[1, 2]	**ProTime**	[0, 100]
LiverFirm	[1, 2]	**Histology**	[1, 2]
SpleenPalpable	[1, 2]	**Class**	{1, 2}

Tabla 3: atributos del dataset Hepatitis.

Para cubrir los datos faltantes, se realizó imputación media-moda.

Para clasificar los resultados se utilizó el método de validación leave-one-out y 9 algoritmos de clasificación. La clase positiva es la 1, representa a los pacientes que tienen la enfermedad.

Las implementaciones de los programas se realizaron en Python utilizando bibliotecas como Pandas y Scikit-Learn. También se utilizó el software WEKA para comparar los resultados. Es importante mencionar que WEKA utiliza implementaciones diferentes a los de Scikit-Learn para cada algoritmo; por lo que los resultados pueden variar.

Los algoritmos de clasificación Python/WEKA utilizados fueron los siguientes:

1. **1-NN/IBk para k=1**
2. **3-NN/ IBk para k=3**
3. **5-NN/ IBk para k=5**

4. **Naïve Bayes gaussiano/NaïveBayes**
5. **Árboles de decisión/J48**
6. **Random Forest/RandomForest**
7. **AdaBoost/AdaBoostM1**
8. **SVM con kernel lineal/SMO**
9. **Multi Layer Perceptron/MultiLayerPerceptron**

Matrices de confusión

1. **1-NN/IBk para k=1**

Python	
17	15
18	105

WEKA	
15	17
12	111

2. **3-NN/ IBk para k=3**

Python	
18	14
13	110

WEKA	
19	13
12	111

3. **3-NN/ IBk para k=5**

Python	
15	17
7	116

WEKA	
16	16
11	112

4. **Naïve Bayes gaussiano/NaïveBayes**

Python	
26	6
20	103

WEKA	
22	10
15	108

5. **Árboles de decisión/J48**

Python	
26	6
13	110

WEKA	
23	9
8	115

6. **Random Forest/RandomForest**

Python	
24	8
4	119

WEKA	
26	6
4	119

7. **AdaBoost/AdaBoostM1**

Python	
27	5
6	117

WEKA	
24	8
11	112

8. **SVM con kernel lineal/SMO**

Python	
25	7
10	113

WEKA	
23	9
8	115

9. **MLP/MultiLayerPerceptron**

Python	
19	**13**
5	**118**

WEKA	
21	**11**
10	**113**

Tabla de métricas de desempeño

	Balanced accuracy	**TP Rate**	**TN Rate**	**Precision**	**F1-Score**	**MCC**
1-NN	0.69250	0.53125	0.85366	0.48571	0.50746	0.37541
3-NN	0.7280	0.5625	0.8943	0.5806	0.5714	0.3694
5-NN	0.7056	0.46875	0.9431	0.6818	0.5556	0.4537
Naïve Bayes gaussiano	0.8250	0.8125	0.8374	0.5652	0.6667	0.5774
Árboles de decisión	0.8530	0.8125	0.8943	0.6667	0.7324	0.7156
Random Forest	0.8590	0.7500	0.9675	0.8571	0.8000	0.7540
AdaBoost	0.8970	0.8440	0.9510	0.8180	0.8310	0.7890
SVM con kernel lineal	0.8500	0.7810	0.9180	0.7140	0.7460	0.6460
Multilayer Perceptron	0.776	0.594	0.959	0.7920	0.6790	0.5940

Se realizó el test de Friedman y posteriormente un análisis post-hoc (test de Nemenyi). Aquí las observaciones al respecto:

1. Evaluación global con el test de Friedman:

- Hipótesis Nula (H_0): no hay diferencias en el rendimiento entre los clasificadores.

Se aplicó el test de Friedman a las métricas de rendimiento de todos los clasificadores.

Si la hipótesis nula es rechazada, lo que indica la presencia de diferencias, se procede al análisis post-hoc. Se utiliza un valor de p estándar, típicamente 0.05, para determinar la

significancia estadística [22]. Este valor de p es un umbral convencional que indica que hay solo un 5% de probabilidad de que los resultados observados se deban al azar si la hipótesis nula es cierta.

2. Análisis con el test de Nemenyi:

- Hipótesis nula para cada par de clasificadores (H_0): no hay diferencia significativa en el rendimiento entre cada par de clasificadores.

Después de que el test de Friedman indicó diferencias, se realizó un análisis post-hoc de Nemenyi para cada par de clasificadores. Este análisis determinó que la diferencia de rendimiento entre cada par es estadísticamente significativa, rechazando o no la hipótesis nula para cada comparación. Se emplea el mismo valor de p de 0.05 para juzgar la significancia estadística, lo que significa que una diferencia se considera significativa si hay menos del 5% de probabilidad de que se haya observado por casualidad bajo la hipótesis nula.

3. Identificación de diferencias significativas:

se identificaron los pares de clasificadores cuya comparación resulta en el rechazo de la hipótesis nula con un valor de p menor a 0.05.

Los pares de clasificadores con diferencias significativas son:

- 1-NN vs Random Forest: $p = 0.0020$
- 1-NN vs AdaBoost: $p = 0.0010$
- 3-NN vs Random Forest: $p = 0.0354$
- 3-NN vs AdaBoost: $p = 0.0085$
- 5-NN vs AdaBoost: $p = 0.0212$

En resumen, se utilizó el test de Friedman para evaluar si hay diferencias generales en el rendimiento entre varios clasificadores. Si se detectan diferencias, se aplica el análisis post-hoc de Nemenyi a cada par de clasificadores, utilizando un valor de p de 0.05 como umbral para determinar la significancia estadística. Esto ayuda a identificar qué pares de clasificadores difieren significativamente en su rendimiento.

Conclusiones

En esta tarea, se llevó a cabo un estudio exhaustivo sobre varios clasificadores de patrones, abarcando desde algoritmos sencillos como la familia k-Nearest Neighbors hasta algoritmos complejos como AdaBoost, Support Vector Machines y redes neuronales multicapa (MLP). Cada uno de estos clasificadores ofreció una buena oportunidad para comparar su rendimiento.

El uso del método de validación *leave-one-out*, la implementación de los clasificadores en Python y la posterior comparación con los resultados obtenidos en WEKA revelaron resultados similares a pesar de que las implementaciones de la biblioteca Scikit-learn y WEKA difieren de algún modo.

El uso de distintas métricas como *Balanced accuracy*, *TP Rate*, *TN Rate*, *Precision*, *F1-Score* y *MCC* permitieron una comparación entre los rendimientos de los clasificadores.

El uso del test de Friedman y el análisis post-hoc de Nemenyi pudieron identificar diferencias significativas en el rendimiento entre los clasificadores. Los resultados han mostrado que, aunque casi todos los clasificadores tienen sus méritos, ciertos pares, como 1-NN frente a Random Forest y AdaBoost, exhiben diferencias significativas en su rendimiento.

En conclusión, esta tarea no solo ha ampliado nuestro portafolio de clasificadores, sino que también ha permitido explorar el tan importante tema del tratamiento de datos faltantes y algunos temas estadísticos.

Referencias bibliográficas

[1] Yáñez-Márquez C. RD 10: Panorama histórico de la familia k-NN, empates, algoritmo k-NN y método de validación Leave-one-out [Diapositivas de clase]. Centro de Investigación en Computación; 2023.

[2] IBM. ¿Qué es KNN? [Internet]. [citado 2023]. Disponible en: https://www.ibm.com/mx-es/topics/knn

[3] IBM. Naïve Bayes. [Internet]. [citado 2023]. Disponible en: https://www.ibm.com/docs/fi/ias?topic=procedures-Naïve-bayes

[4] Devroye L, Gyorfi L, Lugosi G. A probabilistic theory of pattern recognition. Springer; 1996.

[5] Kamiński B, Jakubczyk M, Szufel P. A framework for sensitivity analysis of decision trees. Cent Eur J Oper Res. 2017;26(1):135-159. doi: 10.1007/s10100-017-0479-6. PMC 5767274. PMID 29375266.

[6] IBM. Decision Trees. [Internet]. [citado 2023]. Disponible en: https://www.ibm.com/topics/decision-trees

[7] IBM. Random Forest. [Internet]. [citado 2023]. Disponible en:

https://www.ibm.com/topics/random-forest

[8] James G, Witten D, Hastie T, Tibshirani R. An Introduction to Statistical Learning. Springer; 2013. p. 316-319.

[9] Analytics Vidhya. [Internet]. [citado 2023]. Disponible en: https://www.analyticsvidhya.com/blog/2021/06/understanding-random-forest/

[10] IBM. AdaBoost. [Internet]. [citado 2023]. Disponible en:

https://developer.ibm.com/articles/cc-supervised-learning-models/

[11] IBM. AdaBoost. [Internet]. [citado 2023]. Disponible en:

https://www.ibm.com/topics/bagging

[12] Wyner AJ, Olson M, Bleich J, Mease D. Explaining the Success of AdaBoost and Random Forests as Interpolating Classifiers. J Mach Learn Res. 2017;18(48):1-33.

[13] Cortes C, Vapnik V. Support-vector networks. Mach Learn. 1995;20(3):273-297. doi:10.1007/BF00994018.

[14] Scikit-learn. Support Vector Machines. [Internet]. [citado 2023]. Disponible en:

https://scikit-learn.org/stable/modules/svm.html

[15] IBM. How SVM Works. [Internet]. [citado 2023]. Disponible en: https://www.ibm.com/docs/en/spss-modeler/saas?topic=models-how-svm-works

[16] IBM. Multilayer Perceptron. [Internet]. [citado 2023]. Disponible en: https://www.ibm.com/docs/en/spss-statistics/25.0.0?topic=networks-multilayer-perceptron

[17] James G, Witten D, Hastie T, Tibshirani R. An introduction to statistical learning: with applications in R. New York: Springer; 2013.

[18] Rumelhart D, Hinton G, Williams R. Learning representations by back-propagating errors. Nature. 1986;323(6088):533-536.

[19] KEEL: Hepatitis dataset. [Internet]. [citado 2023]. https://sci2s.ugr.es/keel/dataset.php?cod=100

[20] Yáñez-Márquez C, Solorio-Ramírez J. RD 22: Tests estadísticos [Diapositivas de clase]. Centro de Investigación en Computación; 2023.

Anexos

Los códigos de las implementaciones de los clasificadores pueden consultarse en:

https://colab.research.google.com/drive/15uBaZlEMvZTDejhgvn4fAbyXQVRLPEQY?usp=sharing