# Ensamblaje en Machine Learning

# 1. Introducción al Ensamblaje

El ensamblaje en machine learning es una técnica que combina las predicciones de varios modelos (conocidos como modelos base o débiles) para generar una predicción más precisa y robusta. El objetivo principal es mejorar el rendimiento general, aprovechando la sabiduría colectiva. Un ejemplo intuitivo es cómo se toman decisiones grupales: la combinación de múltiples perspectivas a menudo produce mejores resultados que depender de un solo individuo.

En machine learning, esta técnica se utiliza principalmente para resolver problemas de clasificación y regresión. Los ensamblajes suelen ser más robustos frente a sobreajuste y ruido en los datos.

# 2. Características Principales del Ensamblaje

Las principales características del ensamblaje incluyen:

* **Mejora de la precisión**: al combinar múltiples modelos, se reducen errores de predicción.
* **Robustez**: reduce el impacto de modelos débiles o incorrectos.
* **Flexibilidad**: se pueden usar diversas técnicas como bagging, boosting y stacking.

Las desventajas que Podemos encontrar en estos procesos:

* Mayor complejidad computacional
* Riesgo de sobreajuste si no se configura correctamente.

# 3. Cuándo Utilizar Ensamblaje

El ensamblaje es útil en situaciones donde:

* El rendimiento de un modelo individual no es suficiente.
* Se dispone de datos con ruido o características complejas.
* Se necesita mejorar la estabilidad y la precisión del modelo.

Por ejemplo, en tareas de clasificación de imágenes o predicción financiera.

# 4. Cómo Implementarlo

El ensamblaje se implementa utilizando bibliotecas como Scikit-learn. Aquí hay un ejemplo práctico de Random Forest. En este ejemplo, el Random Forest combina múltiples árboles de decisión para mejorar la precisión.

from sklearn.ensemble import RandomForestClassifier  
from sklearn.datasets import load\_iris  
from sklearn.model\_selection import train\_test\_split  
  
# Cargar datos  
X, y = load\_iris(return\_X\_y=True)  
X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42)  
  
# Crear y entrenar el modelo  
model = RandomForestClassifier(n\_estimators=100, random\_state=42)  
model.fit(X\_train, y\_train)  
  
# Evaluar el modelo  
accuracy = model.score(X\_test, y\_test)  
print(f'Precisión: {accuracy:.2f}')

# Consideraciones Prácticas

Aspectos a tener en cuenta al usar ensamblaje:

* Requiere mayor capacidad computacional.
* Puede ser complicado ajustar los hiperparámetros de todos los modelos base.
* Es importante verificar la independencia de los modelos base para maximizar su efectividad.

# Técnicas Específicas

## Clasificador por Votos - Voting Classifier

Un clasificador de votos combina las predicciones de múltiples modelos base y elige la predicción final mediante:

* **Hard Voting**: Predicción basada en la mayoría de votos (clase más votada).
* **Soft Voting**: Promedia las probabilidades predichas y selecciona la clase con la probabilidad más alta.

### Cuándo usar:

* Cuando se tienen varios modelos bien entrenados que son diversos (predicen de manera diferente).
* Ideal para mejorar la estabilidad y la precisión general.

### Visualización:

Diagrama

Descripción generada automáticamenteDiagrama

Descripción generada automáticamente

### Implementación en Python:

from sklearn.ensemble import VotingClassifier

from sklearn.linear\_model import LogisticRegression

from sklearn.svm import SVC

from sklearn.ensemble import RandomForestClassifier

from sklearn.datasets import load\_iris

from sklearn.model\_selection import train\_test\_split

from sklearn.metrics import accuracy\_score

# Cargar el dataset Iris

X, y = load\_iris(return\_X\_y=True)

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42)

# Crear los modelos base

log\_clf = LogisticRegression(random\_state=42, max\_iter=1000)

svc\_clf = SVC(probability=True, random\_state=42)

rf\_clf = RandomForestClassifier(n\_estimators=100, random\_state=42)

# Crear el Voting Classifier

voting\_clf = VotingClassifier(

estimators=[

('log\_reg', log\_clf),

('svc', svc\_clf),

('random\_forest', rf\_clf)

],

voting='hard' # Cambiar a 'soft' para promedio de probabilidades

)

# Entrenar el modelo de ensamble

voting\_clf.fit(X\_train, y\_train)

# Hacer predicciones

y\_pred = voting\_clf.predict(X\_test)

# Evaluar precisión

accuracy = accuracy\_score(y\_test, y\_pred)

print(f"Precisión del clasificador por votos: {accuracy:.2f}")

## Bagging y Pasting

Ambos metodos entrenan múltiples modelos base en subconjuntos aleatorios del conjunto de datos:

* **Bagging**: Muestreo con reemplazo (puede incluir duplicados en los subconjuntos).
* **Pasting**: Muestreo sin reemplazo (subconjuntos únicos).

### Cuándo usar:

* **Bagging**: Cuando el conjunto de datos es pequeño o se desea reducir la varianza.
* **Pasting**: Cuando el conjunto de datos es grande y se busca evitar duplicados.

### Visualización:

Diagrama

Descripción generada automáticamente

### Implementación en Python (Bagging y Pasting):

from sklearn.ensemble import BaggingClassifier

from sklearn.tree import DecisionTreeClassifier

from sklearn.datasets import make\_classification

from sklearn.model\_selection import train\_test\_split

from sklearn.metrics import accuracy\_score

# Generar datos sintéticos

X, y = make\_classification(n\_samples=100, n\_features=5, n\_classes=2, random\_state=42)

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42)

# Pasting: muestreo sin reemplazo

pasting\_clf = BaggingClassifier(

base\_estimator=DecisionTreeClassifier(),

n\_estimators=10, # Número de predictors

# bootstrap=True, # Muestreo con reemplazo

bootstrap=False, # Muestreo sin reemplazo

random\_state=42

)

# Entrenamiento y predicción

pasting\_clf.fit(X\_train, y\_train)

y\_pred = pasting\_clf.predict(X\_test)

# Evaluar el rendimiento

accuracy = accuracy\_score(y\_test, y\_pred)

print(f"Precisión con Pasting: {accuracy:.2f}")

### Diferencias Clave

* **Bagging**:
  + bootstrap=True: Muestreo con reemplazo, permitiendo duplicados en los subconjuntos.
  + Es útil para reducir la varianza en conjuntos de datos pequeños o ruidosos.
* **Pasting**:
  + bootstrap=False: Muestreo sin reemplazo, asegurando subconjuntos únicos.
  + Ideal para conjuntos de datos más grandes y sin ruido significativo.

## Random Forest

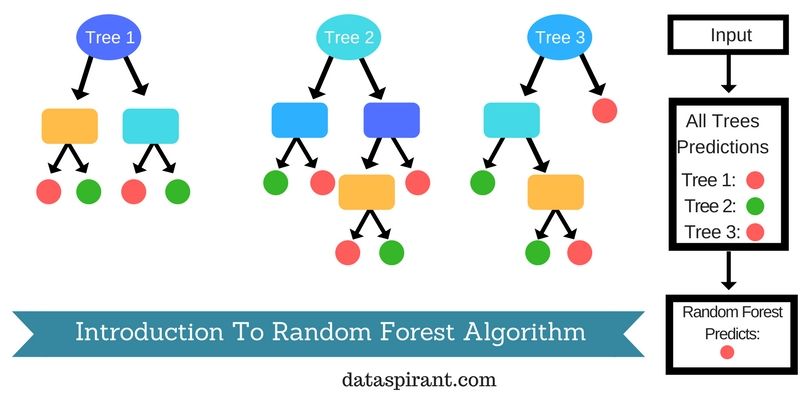
Random Forest es un ensamblaje de árboles de decisión entrenados mediante bagging. Es eficiente y altamente preciso, especialmente en clasificación y regresión.

* Características únicas:
  + Se seleccionan subconjuntos aleatorios de características en cada división.
  + Reduce el sobreajuste y mejora la precisión.

### Cuándo usar:

* En problemas con alta dimensionalidad o conjuntos de datos ruidosos.
* Tanto para clasificación como para regresión.

### Visualización:



### Implementación en Python:

from sklearn.ensemble import RandomForestClassifier

from sklearn.datasets import make\_classification

import numpy as np

# Generar un conjunto de datos de ejemplo

X, y = make\_classification(n\_samples=100, n\_features=5, n\_classes=2, random\_state=42)

# Entrenar el modelo Random Forest

model = RandomForestClassifier(n\_estimators=3, random\_state=42) # 3 árboles

model.fit(X, y)

# Crear un ejemplo de entrada

input\_data = np.array([[0.5, -1.2, 0.3, 0.8, -0.5]]) # Cambia estos valores como prefieras

# Predicciones individuales de cada árbol

tree\_predictions = [tree.predict(input\_data) for tree in model.estimators\_]

print(f"Predicciones de los árboles individuales: {tree\_predictions}")

# Predicción final del Random Forest

final\_prediction = model.predict(input\_data)

print(f"Predicción final del Random Forest: {final\_prediction[0]}")

## Boosting

Boosting combina modelos secuenciales, donde cada modelo intenta corregir los errores del anterior. Ejemplos:

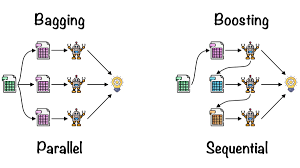
* **AdaBoost**: Asigna pesos a las instancias mal clasificadas.
* **Gradient Boosting**: Optimiza los errores residuales.

### Cuándo usar:

* Cuando se necesita mejorar el rendimiento con modelos base débiles.
* En problemas donde los errores tienen un patrón que puede corregirse.

### Visualización:

***Comparación Baggin / Boosting***



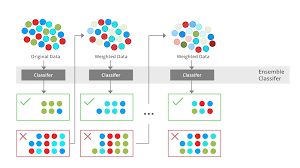
### Comparación

| **Aspecto** | **Bagging (Paralelo)** | **Boosting (Secuencial)** |
| --- | --- | --- |
| **Ejecución** | Paralelo: los modelos se entrenan simultáneamente. | Secuencial: cada modelo depende del anterior. |
| **Objetivo** | Reducir la varianza. | Reducir el sesgo. |
| **Muestreo de Datos** | Muestreo con reemplazo (bootstrap). | Uso completo del conjunto, ajustando pesos. |
| **Combinación** | Votación (clasificación) o promedio (regresión). | Ponderación basada en el rendimiento. |
| **Ejemplo** | Random Forest. | AdaBoost, Gradient Boosting. |

### ¿Cuál elegir?

* **Bagging**: Útil si tu modelo base tiende a sobreajustarse (reduce varianza).
* **Boosting**: Mejor cuando tu modelo base tiene un sesgo alto y quieres aumentar su precisión.

#### Boosting



Esta imagen ilustra el proceso de **Boosting**, un método de ensamblaje secuencial que busca mejorar el rendimiento de un modelo combinando múltiples clasificadores débiles en un clasificador fuerte. A continuación, te explico cada parte:

### ****1. Conjunto de Datos Original****

* Se parte de un conjunto de datos original, donde todas las instancias tienen el mismo peso inicial.
* El primer clasificador se entrena con este conjunto de datos.

### ****2. Pesado de los Datos****

* Tras entrenar el primer clasificador:
  + Las instancias que se clasificaron correctamente reciben un peso menor.
  + Las instancias mal clasificadas reciben un peso mayor.
* Este ajuste de pesos asegura que el siguiente clasificador se enfoque en los datos difíciles.

### ****3. Clasificadores Secuenciales****

* Cada clasificador en la secuencia se entrena con los datos reponderados, es decir, aquellos datos que son más difíciles de clasificar tienen más relevancia en las iteraciones siguientes.
* Este proceso se repite para cada clasificador.

### ****4. Combinación Final (Ensemble Classifier)****

* Una vez que todos los clasificadores han sido entrenados, sus predicciones se combinan.
* La combinación puede realizarse mediante un promedio ponderado o una votación ponderada.
* El resultado final es una predicción más precisa, ya que se aprovechan los puntos fuertes de cada clasificador.

### ****Boosting en Acción****

Un ejemplo práctico es **AdaBoost**:

1. Entrena un clasificador débil (como un árbol de decisión con profundidad 1).
2. Ajusta los pesos de las instancias en función de los errores.
3. Repite el proceso para un número fijo de clasificadores.
4. Combina los resultados de los clasificadores utilizando sus pesos de precisión.

### Implementación en Python (AdaBoost y Gradient Boosting):

*1. Bagging (Parallel)*

from sklearn.ensemble import BaggingClassifier

from sklearn.tree import DecisionTreeClassifier

from sklearn.datasets import make\_classification

from sklearn.model\_selection import train\_test\_split

from sklearn.metrics import accuracy\_score

# Generar datos sintéticos

X, y = make\_classification(n\_samples=100, n\_features=5, n\_classes=2, random\_state=42)

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42)

# Bagging (Ejecución Paralela)

bagging\_clf = BaggingClassifier(

base\_estimator=DecisionTreeClassifier(),

n\_estimators=10, # Número de predictores

bootstrap=True, # Muestreo con reemplazo

n\_jobs=-1, # Paralelización

random\_state=42

)

# Entrenamiento y predicción

bagging\_clf.fit(X\_train, y\_train)

y\_pred = bagging\_clf.predict(X\_test)

# Evaluar el rendimiento

accuracy = accuracy\_score(y\_test, y\_pred)

print(f"Precisión con Bagging (Parallel): {accuracy:.2f}")

*Explicación:*

* BaggingClassifier crea múltiples árboles de decisión.
* n\_jobs=-1 permite paralelización, utilizando todos los núcleos disponibles para entrenar los predictores simultáneamente.

*2. Boosting (Sequential)*

from sklearn.ensemble import AdaBoostClassifier

from sklearn.tree import DecisionTreeClassifier

from sklearn.datasets import make\_classification

from sklearn.model\_selection import train\_test\_split

from sklearn.metrics import accuracy\_score

# Generar datos sintéticos

X, y = make\_classification(n\_samples=100, n\_features=5, n\_classes=2, random\_state=42)

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42)

# Boosting (Ejecución Secuencial)

boosting\_clf = AdaBoostClassifier(

base\_estimator=DecisionTreeClassifier(max\_depth=1), # Árboles débiles

n\_estimators=10, # Número de predictores secuenciales

learning\_rate=1.0, # Peso de cada modelo

random\_state=42

)

# Entrenamiento y predicción

boosting\_clf.fit(X\_train, y\_train)

y\_pred = boosting\_clf.predict(X\_test)

# Evaluar el rendimiento

accuracy = accuracy\_score(y\_test, y\_pred)

print(f"Precisión con Boosting (Sequential): {accuracy:.2f}")

*Explicación:*

* AdaBoostClassifier entrena secuencialmente árboles débiles (profundidad 1).
* Cada modelo posterior se ajusta para corregir los errores de los modelos anteriores.

### Diferencias Clave

| **Método** | **Característica** | **Ejecución** | **Clase Usada** |
| --- | --- | --- | --- |
| **Bagging** | Modelos entrenados en paralelo | Paralelo | BaggingClassifier |
| **Boosting** | Modelos ajustados secuencialmente | Secuencial | AdaBoostClassifier |

## Stacking

Stacking utiliza un modelo meta-aprendiz (blender) para combinar predicciones de múltiples modelos base.

* Es una técnica poderosa para mejorar el rendimiento en tareas complejas.
* Mejora el rendimiento al aprovechar las fortalezas de cada modelo base.

### Cuándo usar:

* Cuando los modelos base tienen patrones complementarios.
* Ideal para problemas complejos donde una combinación simple (e.g., voting) no es suficiente.

### Visualización:

Diagrama

Descripción generada automáticamente

### Implementación en Python:

from sklearn.ensemble import StackingClassifier

# Modelos base

base\_estimators = [

('lr', LogisticRegression()),

('svc', SVC(probability=True)),

('dt', DecisionTreeClassifier())

]

# Meta-modelo

stacking\_clf = StackingClassifier(

estimators=base\_estimators,

final\_estimator=LogisticRegression(),

passthrough=True # Incluye características originales en el meta-modelo

)

stacking\_clf.fit(X\_train, y\_train)

accuracy\_stacking = stacking\_clf.score(X\_test, y\_test)

print(f"Precisión con Stacking: {accuracy\_stacking:.2f}")

## Resumen Comparativo

| **Técnica** | **Características Clave** | **Uso Principal** | **Clase de Scikit-learn** | **Temas Importantes a Considerar** |
| --- | --- | --- | --- | --- |
| **Voting** | Combinación de predicciones finales mediante hard/soft voting | Mejorar estabilidad y precisión | VotingClassifier | - Los modelos base deben ser diversos para evitar redundancia.  - "Soft voting" requiere modelos con predict\_proba. |
| **Bagging/Pasting** | Subconjuntos aleatorios de datos | Reducir varianza (Bagging) | BaggingClassifier, BaggingRegressor | - Bagging es mejor para datos pequeños y ruidosos.  - Pasting funciona bien con conjuntos grandes.  - La independencia de los modelos base mejora los resultados. |
| **Random Forest** | Subconjuntos de datos y características aleatorios | Clasificación y regresión complejas | RandomForestClassifier, RandomForestRegressor | - Escalable con grandes cantidades de datos.  - Sensible a hiperparámetros como max\_depth y n\_estimators.  - Reduce sobreajuste comparado con árboles de decisión simples. |
| **Boosting** | Corrección secuencial de errores | Incrementar precisión | AdaBoostClassifier, AdaBoostRegressor, GradientBoostingClassifier, GradientBoostingRegressor | - AdaBoost es más sensible al ruido en los datos.  - Gradient Boosting puede ajustarse demasiado; regularización como learning\_rate y n\_estimators es clave.  - Costoso computacionalmente para conjuntos de datos grandes. |
| **Stacking** | Meta-aprendiz que combina modelos base | Mejorar rendimiento en problemas complejos | StackingClassifier, StackingRegressor | - Seleccionar un buen meta-modelo es crítico.  - La diversidad entre los modelos base mejora la precisión.  - passthrough=True puede incluir características originales para más contexto. |

# Hiperparámetros Clave

Los hiperparámetros en técnicas de ensamblaje son esenciales para controlar el rendimiento de los modelos, ya que afectan su precisión, velocidad y robustez. A continuación, se detallan los más relevantes y cómo configurarlos:

### Hiperparámetros

* **Número de Estimadores (n\_estimators)**
* **Descripción**: Determina la cantidad de modelos base que se entrenarán en el ensamble (por ejemplo, árboles en Random Forest o iteraciones en AdaBoost).
* **Impacto**:
  + **Mayor número**:
    - Mejora la precisión al reducir la varianza o sesgo, pero aumenta el tiempo de entrenamiento.
    - Puede provocar sobreajuste en algunos casos, especialmente con boosting.
  + **Menor número**:
    - Reduce el tiempo de entrenamiento, pero puede generar modelos subóptimos.
* **Valores recomendados**:
  + Random Forest: Entre 100 y 500.
  + Boosting (AdaBoost/Gradient Boosting): Comienza con 50-100.
* **Muestreo (bootstrap y bootstrap\_features)**
* **Bootstrap (bootstrap)**:
  + Indica si se realiza muestreo con reemplazo para los subconjuntos de entrenamiento.
    - **True**: Usado en Bagging y Random Forest; permite duplicados, mejor para datos pequeños.
    - **False**: Usado en Pasting, mejor para datos grandes y ricos.
* **Muestreo de Características (bootstrap\_features)**:
  + Define si las características (columnas) deben seleccionarse aleatoriamente con reemplazo.
  + Muy útil en Random Forest para reducir la correlación entre árboles.
* **Profundidad Máxima de los Modelos Base (max\_depth)**
* **Descripción**: Controla la complejidad de cada modelo base (por ejemplo, árboles de decisión).
* **Impacto**:
  + **Mayor profundidad**:
    - Captura relaciones más complejas, pero aumenta el riesgo de sobreajuste.
  + **Menor profundidad**:
    - Generaliza mejor, pero puede sufrir de infraajuste.
* **Valores recomendados**:
  + Random Forest: Prueba con valores entre 5 y 20.
  + Boosting: Mantén los modelos débiles, como max\_depth=3.
* **Fracción de Datos Usados en el Muestreo (max\_samples)**
* **Descripción**: Porcentaje de datos seleccionados para entrenar cada modelo base.
* **Impacto**:
  + **Mayor fracción**:
    - Mejora la precisión, pero aumenta la correlación entre modelos base.
  + **Menor fracción**:
    - Reduce la correlación, ideal para datos muy grandes.
* **Valores recomendados**:
  + Comienza con 0.5-0.8 y ajusta según los resultados.
* **Tasa de Aprendizaje (learning\_rate) (para Boosting)**
* **Descripción**: Determina cuánto peso se da a cada modelo base en el boosting.
* **Impacto**:
  + **Alta tasa de aprendizaje**:
    - Converge rápidamente, pero puede perder detalles.
  + **Baja tasa de aprendizaje**:
    - Convergencia más lenta, pero mejora la precisión al final.
* **Valores recomendados**:
  + AdaBoost/Gradient Boosting: 0.01 a 0.1.
* **Selección de Características (max\_features)**
* **Descripción**: Número máximo de características consideradas al entrenar cada modelo base.
* **Impacto**:
  + **Menor número**:
    - Reduce la correlación entre modelos base (mejora en Random Forest).
  + **Mayor número**:
    - Captura más información, pero puede aumentar la correlación.
* **Valores recomendados**:
  + Random Forest:
    - Clasificación: √(número de características).
    - Regresión: Entre 1/3 y 2/3 del total de características.
* **Penalización para Boosting (subsample)**
* **Descripción**: Proporción de datos usada para entrenar cada iteración de Boosting.
* **Impacto**:
  + **Menor fracción**:
    - Reduce el sobreajuste al aumentar la diversidad entre iteraciones.
  + **Mayor fracción**:
    - Mejora la estabilidad, pero puede aumentar el riesgo de sobreajuste.
* **Valores recomendados**:
  + Comienza con 0.5-0.8.
* **Peso de las Clases (class\_weight)**
* **Descripción**: Controla cómo se penalizan las clases desbalanceadas.
* **Impacto**:
  + Mejora la precisión para clases minoritarias en conjuntos desbalanceados.
* **Opciones**:
  + balanced: Calcula automáticamente pesos basados en la proporción de clases.
  + Diccionario: Especifica pesos manualmente.
* **Número de Hilos Paralelos (n\_jobs)**
* **Descripción**: Número de núcleos CPU utilizados para entrenar los modelos base.
* **Impacto**:
  + **-1**: Utiliza todos los núcleos disponibles (recomendado para grandes datos).
  + Acelera el entrenamiento de Bagging y Random Forest.

### Cómo Ajustar Hiperparámetros

* **Validación Cruzada**:
  + Usa GridSearchCV o RandomizedSearchCV para encontrar los mejores valores.
* **Empieza con Valores por Defecto**:
  + Ajusta manualmente solo los hiperparámetros más sensibles (como n\_estimators y max\_depth).
* **Evita el Sobreajuste**:
  + Usa validación cruzada y pruebas con datos no vistos.