



## SESIÓN OPTIMIZACIÓN DE UN MODELO

### CONTENIDOS:

- Mecanismos para mejorar el desempeño de un algoritmo.
- Qué es feature engineering.
- Qué es un hiperparámetro.
- Qué es la regularización.
- Optimización de parámetros.
- Grilla de parámetros.
- Balanceo de datos.

### MECANISMOS PARA MEJORAR EL DESEMPEÑO DE UN ALGORITMO

El desempeño de un algoritmo de Machine Learning puede mejorarse mediante diversas técnicas y estrategias. Veamos los mecanismos más comunes:

#### QUÉ ES FEATURE ENGINEERING

**Feature Engineering** es el proceso de **transformar, crear y seleccionar** las variables de entrada (también conocidas como **características o features**) para mejorar el rendimiento de un modelo de Machine Learning. En otras palabras, es el arte de preparar los datos de manera que el modelo pueda aprender mejor de ellos y hacer predicciones más precisas.

En Machine Learning, los modelos aprenden patrones a partir de los datos que les proporcionamos. Si los datos no están bien preparados o no contienen la información adecuada, el modelo no podrá aprender correctamente. Aquí es donde entra en juego el **Feature Engineering**:

- **Mejora la capacidad del modelo:** Al crear o transformar características, ayudamos al modelo a capturar mejor la relación entre las variables de entrada y la variable objetivo (la que queremos predecir).
- **Reduce el ruido:** Eliminar características irrelevantes o redundantes puede ayudar a que el modelo se enfoque en la información más importante.

- **Facilita el aprendizaje:** Algunos algoritmos funcionan mejor con datos normalizados, escalados o codificados de cierta manera.

## Pasos en Feature Engineering

El proceso de Feature Engineering generalmente incluye los siguientes pasos:

### 1. Creación de Nuevas Características

- A partir de las variables existentes, se pueden generar nuevas características que capturen mejor la relación con la variable objetivo.

### 2. Selección de Características

- Consiste en eliminar características irrelevantes o redundantes para reducir la dimensionalidad del problema.

### 3. Transformación de Datos

- Aplicar técnicas como normalización, escalado, codificación de variables categóricas, etc.

## Ejemplos de Feature Engineering

### Ejemplo 1: Predicción de Precios de Casas

- **Datos originales:** Tamaño de la casa (en metros cuadrados), número de habitaciones, ubicación.
- **Nuevas características:**
  - "Tamaño por habitación": Tamaño de la casa dividido por el número de habitaciones.
  - "Antigüedad de la casa": Año actual menos el año de construcción.
- **Transformaciones:**
  - Normalizar el tamaño de la casa y el número de habitaciones para que estén en la misma escala.
  - Codificar la ubicación (por ejemplo, "centro", "suburbio") usando One-Hot Encoding.

## Ejemplo 2: Clasificación de Texto

- **Datos originales:** Texto de reseñas de productos.
- **Nuevas características:**
  - Longitud del texto: Número de palabras en la reseña.
  - Frecuencia de palabras clave: Número de veces que aparece una palabra específica (por ejemplo, "excelente" o "pésimo").
- **Transformaciones:**
  - Usar **TF-IDF** (Term Frequency-Inverse Document Frequency) para representar el texto numéricamente.
  - Codificar la puntuación de la reseña (por ejemplo, 1 a 5 estrellas) como una variable numérica.

## QUÉ ES UN HIPERPARÁMETRO

Un **hiperparámetro** es un parámetro que se configura antes de entrenar un modelo de Machine Learning. A diferencia de los parámetros del modelo (que se aprenden automáticamente durante el entrenamiento), los hiperparámetros no se ajustan durante el entrenamiento y deben ser definidos por el usuario.

En otras palabras, los hiperparámetros son **configuraciones** que controlan cómo el modelo aprende a partir de los datos. Estos afectan directamente el comportamiento del algoritmo y, por lo tanto, su rendimiento.

## Diferencia entre parámetros e hiperparámetros

- **Parámetros del modelo:** Son los valores que el modelo aprende durante el entrenamiento. Por ejemplo:
  - En una **regresión lineal**, los parámetros son los coeficientes (pesos) que multiplican cada característica.
  - En una **red neuronal**, los parámetros son los pesos y sesgos de las conexiones entre neuronas.

- **Hiperparámetros:** Son configuraciones que se eligen antes de entrenar el modelo. Por ejemplo:
  - En un **árbol de decisión**, el hiperparámetro `max_depth` controla la profundidad máxima del árbol.
  - En una **regresión lineal regularizada**, el hiperparámetro `alpha` controla el grado de regularización.

### Importancia de hiperparámetros

Los hiperparámetros son cruciales porque controlan el comportamiento del algoritmo y afectan directamente su rendimiento. Aquí te explico por qué son importantes:

#### 1. Afectan el rendimiento del modelo:

- Unos hiperparámetros mal elegidos pueden llevar a un modelo que no generaliza bien (sobreajuste o subajuste).
- Por ejemplo, si en un árbol de decisión eliges una `max_depth` muy grande, el modelo puede memorizar los datos de entrenamiento (sobreajuste). Si es muy pequeña, el modelo puede ser demasiado simple y no capturar los patrones de los datos (subajuste).

#### 2. Controlan la complejidad del modelo:

- Algunos hiperparámetros controlan cuán complejo es el modelo. Por ejemplo, en una red neuronal, el número de capas y neuronas determina cuán complejas son las relaciones que el modelo puede aprender.

#### 3. Ayudan a evitar el sobreajuste:

- Hiperparámetros como `alpha` en la regresión regularizada o `max_depth` en los árboles de decisión ayudan a controlar el sobreajuste, penalizando la complejidad del modelo.

## QUÉ ES LA REGULARIZACIÓN

La **regularización** es una técnica utilizada en Machine Learning para prevenir el sobreajuste (overfitting) en los modelos. El sobreajuste ocurre cuando un modelo se ajusta demasiado a los datos de entrenamiento, capturando no solo los patrones generales, sino también el ruido o las fluctuaciones aleatorias presentes en los datos. Como resultado, el modelo puede tener un rendimiento excelente en los datos de entrenamiento, pero un rendimiento pobre en datos nuevos (es decir, no generaliza bien).

La regularización **penaliza la complejidad del modelo**, evitando que los coeficientes (o pesos) del modelo tomen valores extremos. Esto ayuda a que el modelo sea más simple y generalice mejor a datos no vistos.

La regularización nos ayuda a:


1. **Evita el sobreajuste:** La regularización ayuda a que el modelo no se ajuste demasiado a los datos de entrenamiento, lo que mejora su capacidad para generalizar a nuevos datos.
2. **Controla la complejidad del modelo:** Al penalizar los coeficientes grandes, la regularización reduce la complejidad del modelo, lo que puede ser especialmente útil cuando hay muchas características (variables de entrada).
3. **Mejora el rendimiento en datos no vistos:** Un modelo regularizado tiende a tener un mejor rendimiento en datos de prueba o validación, ya que no está sobreajustado a los datos de entrenamiento.

## Tipos de regularización

Existen varios tipos de regularización, dependiendo del tipo de penalización que se aplica a los coeficientes del modelo. Los más comunes son:

### 1. Regularización L2 (Ridge Regression)

- **Qué hace:** Añade una penalización proporcional al **cuadrado de los coeficientes** del modelo. Esto reduce la magnitud de los coeficientes, pero no los elimina por completo.
- **Fórmula:** La función de costo se modifica añadiendo un término de penalización:


$$\text{Costo} = \text{Error} + \lambda \sum_{i=1}^n \beta_i^2$$

*Ilustración 1 Fórmula regularización L2.*

Donde:

- $\lambda$  (lambda) es el parámetro de regularización (controla la fuerza de la penalización).
- $\beta_i$  son los coeficientes del modelo.
- **Cuándo usarla:** Es útil cuando todas las características son relevantes y no se desea eliminar ninguna.

## 2. Regularización L1 (Lasso Regression)

- **Qué hace:** Añade una penalización proporcional al **valor absoluto de los coeficientes**. Esto puede llevar a que algunos coeficientes sean exactamente cero, lo que efectivamente elimina esas características del modelo (selección de características).
- **Fórmula:** La función de costo se modifica añadiendo un término de penalización:

$$\text{Costo} = \text{Error} + \lambda \sum_{i=1}^n |\beta_i|$$

*Ilustración 2 Fórmula regularización L1.*

- **Cuándo usarla:** Es útil cuando se sospecha que muchas características son irrelevantes y se desea realizar una selección automática de características.

## 3. Elastic Net

- **Qué hace:** Combina las penalizaciones de L1 y L2. Es una mezcla de Ridge y Lasso, lo que permite un equilibrio entre ambas.

- **Fórmula:** La función de costo incluye ambos términos de penalización:

$$\text{Costo} = \text{Error} + \lambda_1 \sum_{i=1}^n |\beta_i| + \lambda_2 \sum_{i=1}^n \beta_i^2$$

*Ilustración 3 Fórmula Elastic Net.*

- **Cuándo usarla:** Es útil cuando hay muchas características y se desea un equilibrio entre la selección de características (L1) y la reducción de la magnitud de los coeficientes (L2).

### Ejemplo de regularización en un modelo de regresión lineal

Imaginemos que tenemos un modelo de regresión lineal que predice el precio de una casa en función de características como el tamaño, el número de habitaciones y la antigüedad. Sin regularización, el modelo podría ajustarse demasiado a los datos de entrenamiento, especialmente si hay muchas características o si algunas de ellas son irrelevantes.

Al aplicar regularización:

- **Ridge Regression (L2):** Reduciría la magnitud de los coeficientes, pero mantendría todas las características en el modelo.
- **Lasso Regression (L1):** Podría eliminar algunas características (por ejemplo, si el número de habitaciones no es relevante, su coeficiente se reduciría a cero).
- **Elastic Net:** Encontraría un equilibrio entre reducir la magnitud de los coeficientes y eliminar características irrelevantes.

### Cómo elegir el parámetro de regularización ( $\lambda$ )

El parámetro  $\lambda$  (lambda) controla la fuerza de la regularización:

- Si  $\lambda$  es **muy grande**, la penalización será muy fuerte, lo que puede llevar a un modelo demasiado simple (subajuste).
- Si  $\lambda$  es **muy pequeño**, la penalización será débil, lo que puede llevar a un modelo complejo y sobreajustado.

Para elegir el valor óptimo de  $\lambda$ , se suele usar **validación cruzada** (cross-validation) o técnicas como **Grid Search**.

### Implementación con Scikit-Learn

Aquí vemos un ejemplo de cómo aplicar regularización en un modelo de regresión lineal usando Scikit-learn:

```
1  from sklearn.linear_model import Ridge, Lasso, ElasticNet
2  from sklearn.model_selection import train_test_split
3  from sklearn.metrics import mean_squared_error
4  from sklearn.datasets import make_regression
5
6  # 1. Generar datos sintéticos
7  X, y = make_regression(n_samples=1000, n_features=20, noise=0.5, random_state=42)
8
9  # 2. Dividir los datos en conjuntos de entrenamiento y prueba
10 X_train, X_test, y_train, y_test = train_test_split(
11     X, y, test_size=0.2, random_state=42
12 )
13
14 # 3. Aplicar Ridge Regression (L2)
15 ridge = Ridge(alpha=1.0) # alpha es el parámetro de regularización (lambda)
16 ridge.fit(X_train, y_train)
17 y_pred_ridge = ridge.predict(X_test)
18 print("Error de Ridge Regression:", mean_squared_error(y_test, y_pred_ridge))
19
20 # 4. Aplicar Lasso Regression (L1)
21 lasso = Lasso(alpha=1.0)
22 lasso.fit(X_train, y_train)
23 y_pred_lasso = lasso.predict(X_test)
24 print("Error de Lasso Regression:", mean_squared_error(y_test, y_pred_lasso))
25
26 # 5. Aplicar Elastic Net
27 elastic_net = ElasticNet(
28     alpha=1.0, l1_ratio=0.5
29 ) # l1_ratio controla la mezcla entre L1 y L2
30 elastic_net.fit(X_train, y_train)
31 y_pred_elastic = elastic_net.predict(X_test)
32 print("Error de Elastic Net:", mean_squared_error(y_test, y_pred_elastic))
```

*Ilustración 4 Técnicas regularización.*



## OPTIMIZACIÓN DE PARÁMETROS

La optimización de parámetros es el proceso de **ajustar los hiperparámetros** de un modelo de Machine Learning para encontrar la combinación que maximice su rendimiento. Los **hiperparámetros** son configuraciones que se definen antes de entrenar el modelo y que no se aprenden automáticamente durante el entrenamiento. A diferencia de los **parámetros del modelo** (que se ajustan durante el entrenamiento, como los coeficientes en una regresión lineal), los hiperparámetros controlan el comportamiento del algoritmo y afectan directamente su capacidad para aprender de los datos.

La optimización de parámetros:

1. **Mejora el rendimiento del modelo:** Unos hiperparámetros bien ajustados pueden mejorar significativamente la precisión del modelo.
2. **Evita el sobreajuste (overfitting):** Algunos hiperparámetros controlan la complejidad del modelo. Si no se ajustan correctamente, el modelo puede sobreajustarse a los datos de entrenamiento y no generalizar bien a nuevos datos.
3. **Ahorra tiempo y recursos:** En lugar de probar manualmente diferentes combinaciones de hiperparámetros, la optimización automatizada permite encontrar la mejor configuración de manera eficiente.

### Tipos de hiperparámetros en diferentes algoritmos

Algunos ejemplos de hiperparámetros comunes en diferentes algoritmos que ya hemos aprendido:

#### 1. Árboles de Decisión y Random Forest

- **max\_depth:** Profundidad máxima del árbol. Si es demasiado grande, el árbol puede sobreajustarse; si es demasiado pequeño, puede subajustarse.
- **min\_samples\_split:** Número mínimo de muestras requeridas para dividir un nodo.
- **min\_samples\_leaf:** Número mínimo de muestras requeridas en una hoja (nodo final).

#### 2. Regresión Lineal Regularizada (Ridge, Lasso)

- **alpha:** Controla el grado de regularización. Un valor alto de alpha aumenta la penalización, lo que reduce la magnitud de los coeficientes.

#### 4. K-Vecinos Más Cercanos (K-Nearest Neighbors, KNN)

- **n\_neighbors:** Número de vecinos que se consideran para hacer una predicción.

#### 5. Máquinas de Soporte Vectorial (SVM)

- **C:** Controla el trade-off entre maximizar el margen y minimizar el error de clasificación.
- **kernel:** Tipo de función de kernel (lineal, polinómico, radial, etc.).

### Elección de hiperpárametros

Elegir los hiperpárametros correctos es un proceso clave en Machine Learning. Algunas técnicas comunes son:

#### 1. Búsqueda en Grilla (Grid Search)

- Consiste en probar todas las combinaciones posibles de hiperpárametros en una grilla predefinida.
- **Ejemplo:** Si tienes dos hiperpárametros, `max_depth` y `min_samples_split`, podrías probar todas las combinaciones de valores para estos hiperpárametros.
- **Ventaja:** Es exhaustivo, ya que prueba todas las combinaciones.
- **Desventaja:** Puede ser computacionalmente costoso si hay muchos hiperpárametros o valores posibles.

#### 2. Búsqueda Aleatoria (Random Search)

- En lugar de probar todas las combinaciones, se prueban combinaciones aleatorias de hiperpárametros dentro de un rango.
- **Ventaja:** Es más eficiente que Grid Search, especialmente cuando hay muchos hiperpárametros.
- **Desventaja:** No garantiza encontrar la mejor combinación, pero suele ser suficiente en la práctica.

### 3. Optimización Bayesiana

- Es una técnica más avanzada que utiliza métodos probabilísticos para encontrar la mejor combinación de hiperparámetros.
- **Ventaja:** Es más eficiente que Grid Search y Random Search, ya que se enfoca en las áreas más prometedoras del espacio de hiperparámetros.
- **Desventaja:** Es más compleja de implementar.

#### Implementación con la librería scikit-learn

Ahora veremos implementaciones sencillas de **Grid Search**, **Random Search** y **Optimización Bayesiana** en Python, usando scikit-learn, para la elección de hiperparámetros. Para la **Optimización Bayesiana** es necesario instalar la biblioteca **scikit-optimize** (skopt), que proporciona BayesSearchCV.

```
pip install scikit-optimize
```

```
1 import numpy as np
2 from sklearn.model_selection import GridSearchCV, RandomizedSearchCV
3 from sklearn.ensemble import RandomForestClassifier
4 from skopt import BayesSearchCV
5 from sklearn.datasets import load_iris
6 from sklearn.model_selection import train_test_split
7
8 # Cargar datos de ejemplo (Iris dataset)
9 data = load_iris()
10 X_train, X_test, y_train, y_test = train_test_split(
11     data.data, data.target, test_size=0.2, random_state=42
12 )
13
14 # Modelo base
15 model = RandomForestClassifier(random_state=42)
16
17 # 1. Grid Search
18 param_grid = {"n_estimators": [10, 50, 100], "max_depth": [None, 5, 10]}
19 grid_search = GridSearchCV(model, param_grid, cv=3, scoring="accuracy", n_jobs=-1)
20 grid_search.fit(X_train, y_train)
21 print("Mejor configuración con Grid Search:", grid_search.best_params_)
22
```

Ilustración 5 elección de hiperparámetros parte 1

```

23 # 2. Random Search
24 param_dist = {
25     "n_estimators": np.arange(10, 200, 10),
26     "max_depth": [None] + list(np.arange(3, 20, 3)),
27 }
28 random_search = RandomizedSearchCV(
29     model, param_dist, n_iter=10, cv=3, scoring="accuracy", n_jobs=-1, random_state=42
30 )
31 random_search.fit(X_train, y_train)
32 print("Mejor configuración con Random Search:", random_search.best_params_)
33
34 # 3. Optimización Bayesiana
35 bayes_search = BayesSearchCV(
36     model,
37     {"n_estimators": (10, 200), "max_depth": (3, 20)},
38     n_iter=10,
39     cv=3,
40     scoring="accuracy",
41     n_jobs=-1,
42     random_state=42,
43 )
44 bayes_search.fit(X_train, y_train)
45 print("Mejor configuración con Optimización Bayesiana:", bayes_search.best_params_)

```

*Ilustración 5 elección de hiperparámetros parte 2*

1. **Grid Search:** Busca en una cuadrícula fija de hiperparámetros.
2. **Random Search:** Explora combinaciones aleatorias dentro de un rango de valores.
3. **Optimización Bayesiana:** Usa modelos probabilísticos para elegir la mejor combinación de hiperparámetros de manera más eficiente.

## GRILLA DE PARÁMETROS

Una Grilla de Parámetros (también conocida como **Grid de Parámetros**) es una **estructura** que define un **conjunto de valores posibles** para los **hiperparámetros** de un modelo de Machine Learning. En otras palabras, es una **lista organizada de combinaciones de hiperparámetros** que se van a probar durante el proceso de optimización.

La grilla de parámetros se utiliza en técnicas como **Grid Search** (Búsqueda en Grilla), donde el objetivo es **probar todas las combinaciones posibles** de los valores definidos en la grilla para encontrar la **mejor combinación de hiperparámetros** que maximice el rendimiento del modelo.

## Funcionamiento Grilla de Parámetros

**Definición de los hiperparámetros:** Primero, se identifican los hiperparámetros que se desean optimizar. Por ejemplo, en un modelo de **Random Forest**, los hiperparámetros podrían ser:

- **n\_estimators:** Número de árboles en el bosque.
  - **max\_depth:** Profundidad máxima de cada árbol.
  - **min\_samples\_split:** Número mínimo de muestras requeridas para dividir un nodo.
2. **Especificación de los valores:** Para cada hiperparámetro, se define un **rango de valores posibles** que se van a probar. Por ejemplo:
- **n\_estimators:** [100, 200, 300]
  - **max\_depth:** [None, 10, 20]
  - **min\_samples\_split:** [2, 5, 10]
3. **Creación de la grilla:** La grilla de parámetros es una **combinación de todos los valores posibles** de los hiperparámetros. En este caso, la grilla tendría 3 (valores de **n\_estimators**) × 3 (valores de **max\_depth**) × 3 (valores de **min\_samples\_split**) = **27 combinaciones** diferentes.
4. **Búsqueda en la grilla:** Durante el proceso de **Grid Search**, el modelo se entrena y evalúa para **cada combinación** de la grilla. La combinación que da el mejor rendimiento (por ejemplo, la mayor precisión o el menor error) se selecciona como la **mejor configuración de hiperparámetros**.

## Ejemplo conceptual de una grilla de parámetros


Imaginemos que estamos trabajando con un modelo de **Random Forest** y queremos optimizar los siguientes hiperparámetros:

1. **n\_estimators:** Número de árboles en el bosque.
  - Valores a probar: [100, 200, 300]
2. **max\_depth:** Profundidad máxima de cada árbol.
  - Valores a probar: [None, 10, 20]
3. **min\_samples\_split:** Número mínimo de muestras requeridas para dividir un nodo.

- Valores a probar: [2, 5, 10]

La **grilla de parámetros** sería una lista de todas las combinaciones posibles de estos valores:

| n_estimators | max_depth | min_samples_split |
|--------------|-----------|-------------------|
| 100          | None      | 2                 |
| 100          | None      | 5                 |
| 100          | None      | 10                |
| 100          | 10        | 2                 |
| 100          | 10        | 5                 |
| 100          | 10        | 10                |
| 100          | 20        | 2                 |
| 100          | 20        | 5                 |
| 100          | 20        | 10                |
| 200          | None      | 2                 |
| 200          | None      | 5                 |
| 200          | None      | 10                |
| 200          | 10        | 2                 |
| 200          | 10        | 5                 |
| 200          | 10        | 10                |
| 200          | 20        | 2                 |



|     |      |    |
|-----|------|----|
| 200 | 20   | 5  |
| 200 | 20   | 10 |
| 300 | None | 2  |
| 300 | None | 5  |
| 300 | None | 10 |
| 300 | 10   | 2  |
| 300 | 10   | 5  |
| 300 | 10   | 10 |
| 300 | 20   | 2  |
| 300 | 20   | 5  |
| 300 | 20   | 10 |

En total, hay **27 combinaciones** en esta grilla. Durante el **Grid Search**, el modelo se entrenará y evaluará para cada una de estas combinaciones, y se seleccionará la que dé el mejor rendimiento.

#### Ventajas y desventajas de usar una grilla de parámetros

- **Ventajas:**
  - **Exhaustividad:** Prueba todas las combinaciones posibles de hiperparámetros, lo que aumenta las posibilidades de encontrar la mejor configuración.
  - **Facilidad de implementación:** Es fácil de entender y aplicar, especialmente con herramientas como **Scikit-learn**.
  - **Transparencia:** Al probar todas las combinaciones, puedes ver claramente cómo cada hiperparámetro afecta el rendimiento del modelo.

- **Desventajas:**

- **Coste computacional:** Probar todas las combinaciones puede ser muy costoso en términos de tiempo y recursos, especialmente si hay muchos hiperparámetros o valores posibles.
- **Ineficiencia:** Si algunos valores de hiperparámetros no son relevantes, se perderá tiempo probándolos.

Algunas alternativas a la Grilla de parámetros, que ya vimos son:

- **Búsqueda Aleatoria (Random Search):** En lugar de probar todas las combinaciones, se prueban combinaciones aleatorias de hiperparámetros. Es más eficiente que Grid Search, especialmente cuando hay muchos hiperparámetros.
- **Optimización Bayesiana:** Usa métodos probabilísticos para encontrar la mejor combinación de hiperparámetros de manera más eficiente.

## Implementación con la librería scikit-learn

Veamos un ejemplo de cómo ajustar hiperparámetros usando **Grid Search** en Scikit-learn con datos sintéticos:

```
1 from sklearn.model_selection import GridSearchCV, train_test_split
2 from sklearn.ensemble import RandomForestClassifier
3 from sklearn.datasets import make_classification
4
5 # 1. Generar datos sintéticos
6 X, y = make_classification(
7     n_samples=1000, # Número de muestras
8     n_features=20, # Número de características
9     n_informative=2, # Número de características informativas
10    random_state=42, # Semilla para reproducibilidad
11)
12
13 # 2. Dividir los datos en conjuntos de entrenamiento y prueba
14 X_train, X_test, y_train, y_test = train_test_split(
15     X, y, test_size=0.2, random_state=42
16 )
17
18 # 3. Definir la grilla de hiperparámetros
19 param_grid = {
20     "n_estimators": [100, 200, 300], # Número de árboles en el bosque
21     "max_depth": [None, 10, 20], # Profundidad máxima de cada árbol
22     "min_samples_split": [2, 5, 10], # Mínimo de muestras para dividir un nodo
23 }
24
25 # 4. Crear el modelo
26 model = RandomForestClassifier(random_state=42)
27
28 # 5. Aplicar Grid Search
29 grid_search = GridSearchCV(estimator=model, param_grid=param_grid, cv=5)
30 grid_search.fit(X_train, y_train)
31
32 # 6. Mejores hiperparámetros encontrados
33 print("Mejores hiperparámetros:", grid_search.best_params_)
```

*Ilustración 6 Implementación Grilla de parámetros.*



## BALANCEO DE DATOS

El **Balanceo de Datos** es una técnica utilizada para manejar problemas de **clasificación** donde las **clases** (categorías o etiquetas) están **desequilibradas**. Esto significa que una clase tiene muchas más muestras que otra (o que otras). Por ejemplo, en un problema de detección de fraude, la mayoría de las transacciones son legítimas, y solo una pequeña fracción son fraudulentas. Este desequilibrio puede causar problemas en el entrenamiento del modelo, ya que este podría sesgarse hacia la clase mayoritaria, ignorando la clase minoritaria.

El objetivo del balanceo de datos es **equilibrar la distribución de las clases** para que el modelo pueda aprender de manera más efectiva a predecir ambas clases.

### Importancia del balanceo de datos

1. **Evita el sesgo hacia la clase mayoritaria:** Si una clase tiene muchas más muestras que otra, el modelo puede aprender a predecir siempre la clase mayoritaria, lo que resulta en un rendimiento pobre para la clase minoritaria.
2. **Mejora la capacidad de generalización:** Un modelo entrenado con datos balanceados es más capaz de generalizar y predecir correctamente ambas clases.
3. **Métricas de evaluación más confiables:** En conjuntos de datos desequilibrados, métricas como la **precisión** pueden ser engañosas. Por ejemplo, si el 95% de las muestras son de una clase, un modelo que siempre predice esa clase tendrá una precisión del 95%, pero no será útil. El balanceo de datos permite usar métricas más confiables, como la **precisión equilibrada** o el **F1-score**.

### Técnicas de balanceo de datos

Existen varias técnicas para balancear los datos, y se pueden dividir en dos categorías principales:

#### 1. Submuestreo (Undersampling)

- **Qué hace:** Reduce el número de muestras de la clase mayoritaria para equilibrar las clases.
- **Cómo funciona:** Se eliminan aleatoriamente muestras de la clase mayoritaria hasta que el número de muestras sea similar al de la clase minoritaria.

- **Ventaja:** Es simple y rápido.
- **Desventaja:** Puede llevar a la pérdida de información valiosa, ya que se descartan muestras de la clase mayoritaria.
- **Ejemplo:** Si tienes 1000 muestras de la clase mayoritaria y 100 de la minoritaria, el submuestreo reduciría las muestras de la clase mayoritaria a 100.

## 2. Sobremuestreo (Oversampling)

- **Qué hace:** Aumenta el número de muestras de la clase minoritaria para equilibrar las clases.
- **Cómo funciona:** Se duplican muestras de la clase minoritaria o se generan nuevas muestras sintéticas.
- **Ventaja:** No se pierde información, ya que no se eliminan muestras.
- **Desventaja:** Puede llevar a sobreajuste, especialmente si se duplican muestras exactas.
- **Ejemplo:** Si tienes 1000 muestras de la clase mayoritaria y 100 de la minoritaria, el sobremuestreo aumentaría las muestras de la clase minoritaria a 1000.

## 3. Técnicas Híbridas (Combinación de Submuestreo y Sobremuestreo)

- **Qué hace:** Combina el submuestreo de la clase mayoritaria y el sobremuestreo de la clase minoritaria.
- **Ventaja:** Equilibra las clases sin perder demasiada información ni generar demasiado sobreajuste.
- **Ejemplo:** Reducir las muestras de la clase mayoritaria a 500 y aumentar las muestras de la clase minoritaria a 500.

## 4. Generación de Muestras Sintéticas (SMOTE)

- **Qué hace:** Crea nuevas muestras sintéticas de la clase minoritaria en lugar de simplemente duplicar las existentes.
- **Cómo funciona:** SMOTE (Synthetic Minority Over-sampling Technique) genera nuevas muestras interpolando entre muestras existentes de la clase minoritaria.
- **Ventaja:** Reduce el riesgo de sobreajuste en comparación con el sobremuestreo simple.
- **Desventaja:** Puede generar muestras poco realistas si no se aplica correctamente.

- **Ejemplo:** Si tienes 100 muestras de la clase minoritaria, SMOTE podría generar 900 muestras sintéticas para equilibrar las clases.

### Ejemplo conceptual de balanceo de datos

Imagina que tienes un conjunto de datos para predecir si un correo electrónico es **spam** o **no spam**. Supongamos que tienes 1000 correos electrónicos, de los cuales 950 son no spam y 50 son spam. Esto es un problema de clases desequilibradas.

#### Sin Balanceo de Datos:

- El modelo podría aprender a predecir siempre "no spam" y tendría una precisión del 95%, pero no sería útil para detectar spam.

#### Con Balanceo de Datos:

1. **Submuestreo:** Reducirías el número de correos no spam a 50, equilibrando las clases.
2. **Sobremuestreo:** Aumentarías el número de correos spam a 950, duplicando o generando muestras sintéticas.
3. **SMOTE:** Generarías 900 correos spam sintéticos para equilibrar las clases.

Después del balanceo, el modelo tendría una distribución equilibrada de clases (por ejemplo, 950 no spam y 950 spam), lo que le permitiría aprender a predecir ambas clases de manera más efectiva.

#### Ventajas:

1. **Mejora el rendimiento del modelo:** Un modelo entrenado con datos balanceados es más capaz de predecir correctamente ambas clases.
2. **Reduce el sesgo:** Evita que el modelo se incline hacia la clase mayoritaria.
3. **Métricas más confiables:** Permite usar métricas como el **F1-score** o la **precisión equilibrada**, que son más adecuadas para problemas con clases desequilibradas.

#### Desventajas:

1. **Pérdida de información (Submuestreo):** Al eliminar muestras de la clase mayoritaria, se puede perder información valiosa.

2. **Sobreajuste (Sobremuestreo)**: Duplicar muestras de la clase minoritaria puede llevar a que el modelo se sobreajuste a esas muestras.
3. **Complejidad (SMOTE)**: Generar muestras sintéticas puede ser computacionalmente costoso y requiere un buen entendimiento de los datos.

## ACTIVIDAD PRÁCTICA GUIADA

Para ver la implementación en Python con Scikit-Learn, haremos un ejercicio guiado:

### 1. Importar Librerías

- Instalamos librerías **Scikit-learn** y **imbalanced-learn** (una extensión de Scikit-learn para manejar datos desequilibrados) y las importamos.

```
1 # 1. Importar librerías necesarias
2 from sklearn.datasets import make_classification
3 from sklearn.model_selection import train_test_split
4 from sklearn.ensemble import RandomForestClassifier
5 from sklearn.metrics import classification_report, confusion_matrix
6 from imblearn.under_sampling import RandomUnderSampler
7 from imblearn.over_sampling import RandomOverSampler, SMOTE
8 from imblearn.combine import SMOTEENN
9 import pandas as pd
```

### 2. Generación de datos desbalanceados:

- Usamos `make_classification` para crear un conjunto de datos sintético con 1000 muestras, 20 características y una distribución desbalanceada (95% clase 0 y 5% clase 1).

```
11 # 2. Generar un conjunto de datos desbalanceado sintético
12 X, y = make_classification(
13     n_samples=1000, # 1000 muestras
14     n_features=20, # 20 características
15     n_classes=2, # 2 clases (binario)
16     weights=[0.95, 0.05], # 95% clase 0, 5% clase 1 (desbalanceado)
17     random_state=42, # Semilla para reproducibilidad
18 )
19
20 # Convertir a un DataFrame para visualización (opcional)
21 df = pd.DataFrame(X, columns=[f"Feature_{i}" for i in range(20)])
22 df["Target"] = y
23
24 # Ver la distribución de las clases antes del balanceo
25 print("Distribución de clases antes del balanceo:")
26 print(df["Target"].value_counts())
```

### 3. División de datos:

- Dividimos los datos en conjuntos de entrenamiento (X\_train, y\_train) y prueba (X\_test, y\_test) usando train\_test\_split.

```
28 # 3. Dividir los datos en conjuntos de entrenamiento y prueba
29 X_train, X_test, y_train, y_test = train_test_split(
30     X, y, test_size=0.2, random_state=42
31 )
```

### 4. Aplicación de las técnicas de balanceo:

- **Submuestreo (Undersampling):** Usamos RandomUnderSampler para reducir las muestras de la clase mayoritaria.

```
33 # 4. Aplicar las 4 técnicas de balanceo de datos
34 # Técnica 1: Submuestreo (Undersampling)
35 print("\n--- Submuestreo (Undersampling) ---")
36 undersampler = RandomUnderSampler(random_state=42)
37 X_train_under, y_train_under = undersampler.fit_resample(X_train, y_train)
38
39 # Ver la distribución de clases después del submuestreo
40 print("Distribución de clases después del submuestreo:")
41 print(pd.Series(y_train_under).value_counts())
```

- **Sobremuestreo (Oversampling):** Usamos RandomOverSampler para aumentar las muestras de la clase minoritaria.

```
43 # 4. Aplicar las 4 técnicas de balanceo de datos
44 # Técnica 2: Sobremuestreo (Oversampling)
45 print("\n--- Sobremuestreo (Oversampling) ---")
46 oversampler = RandomOverSampler(random_state=42)
47 X_train_over, y_train_over = oversampler.fit_resample(X_train, y_train)
48
49 # Ver la distribución de clases después del sobremuestreo
50 print("Distribución de clases después del sobremuestreo:")
51 print(pd.Series(y_train_over).value_counts())
```

- **Combinación de Submuestreo y Sobremuestreo (SMOTEENN):** Usamos SMOTEENN, que combina SMOTE (sobremuestreo) y ENN (Edited Nearest Neighbours, una técnica de submuestreo).

```

53 # 4. Aplicar las 4 técnicas de balanceo de datos
54 # Técnica 3: Combinación de Submuestreo y Sobremuestreo (SMOTEENN)
55 print("\n--- Combinación de Submuestreo y Sobremuestreo (SMOTEENN) ---")
56 smote_enn = SMOTEENN(random_state=42)
57 X_train_smoteenn, y_train_smoteenn = smote_enn.fit_resample(X_train, y_train)
58
59 # Ver la distribución de clases después de SMOTEENN
60 print("Distribución de clases después de SMOTEENN:")
61 print(pd.Series(y_train_smoteenn).value_counts())

```

- **Generación de Muestras Sintéticas (SMOTE):** Usamos SMOTE para generar muestras sintéticas de la clase minoritaria.

```

63 # 4. Aplicar las 4 técnicas de balanceo de datos
64 # Técnica 4: Generación de Muestras Sintéticas (SMOTE)
65 print("\n--- Generación de Muestras Sintéticas (SMOTE) ---")
66 smote = SMOTE(random_state=42)
67 X_train_smote, y_train_smote = smote.fit_resample(X_train, y_train)
68
69 # Ver la distribución de clases después de SMOTE
70 print("Distribución de clases después de SMOTE:")
71 print(pd.Series(y_train_smote).value_counts())

```

## 5. Entrenamiento y evaluación del modelo:

- Para cada técnica de balanceo, entrenamos un modelo de **Random Forest** y evaluamos su rendimiento en el conjunto de prueba usando métricas como la **matriz de confusión** y el **reporte de clasificación**.

```

73 # 5. Entrenar y evaluar un modelo con cada técnica
74
75 # Función para entrenar y evaluar un modelo
76 def train_and_evaluate(X_train, y_train, X_test, y_test, technique_name):
77     print(f"\n--- Evaluación del modelo con {technique_name} ---")
78     model = RandomForestClassifier(random_state=42)
79     model.fit(X_train, y_train)
80     y_pred = model.predict(X_test)
81
82     # Métricas de evaluación
83     print("Matriz de confusión:")
84     print(confusion_matrix(y_test, y_pred))
85
86     print("\nReporte de clasificación:")
87     print(classification_report(y_test, y_pred))
88
89
90 # Evaluar cada técnica
91 train_and_evaluate(X_train_under, y_train_under, X_test, y_test, "Submuestreo")
92 train_and_evaluate(X_train_over, y_train_over, X_test, y_test, "Sobremuestreo")
93 train_and_evaluate(X_train_smoteenn, y_train_smoteenn, X_test, y_test, "SMOTEENN")
94 train_and_evaluate(X_train_smote, y_train_smote, X_test, y_test, "SMOTE")

```

## Salida Esperada:

```
Distribución de clases antes del balanceo:
Target
0    947
1     53
Name: count, dtype: int64

--- Submuestreo (Undersampling) ---
Distribución de clases después del submuestreo:
0     42
1     42
Name: count, dtype: int64

--- Sobremuestreo (Oversampling) ---
Distribución de clases después del sobremuestreo:
0     758
1     758
Name: count, dtype: int64

--- Combinación de Submuestreo y Sobremuestreo (SMOTEENN) ---
Distribución de clases después de SMOTEENN:
1     758
0     549
Name: count, dtype: int64

--- Generación de Muestras Sintéticas (SMOTE) ---
Distribución de clases después de SMOTE:
0     758
1     758
Name: count, dtype: int64
```

```
--- Evaluación del modelo con Submuestreo ---
Matriz de confusión:
[[162  27]
 [  3   8]]

Reporte de clasificación:

```

|              | precision | recall | f1-score | support |
|--------------|-----------|--------|----------|---------|
| 0            | 0.98      | 0.86   | 0.92     | 189     |
| 1            | 0.23      | 0.73   | 0.35     | 11      |
| accuracy     |           |        | 0.85     | 200     |
| macro avg    | 0.61      | 0.79   | 0.63     | 200     |
| weighted avg | 0.94      | 0.85   | 0.88     | 200     |

```

--- Evaluación del modelo con Sobremuestreo ---
Matriz de confusión:
[[187   2]
 [  8   3]]

Reporte de clasificación:

```

|              | precision | recall | f1-score | support |
|--------------|-----------|--------|----------|---------|
| 0            | 0.96      | 0.99   | 0.97     | 189     |
| 1            | 0.60      | 0.27   | 0.38     | 11      |
| accuracy     |           |        | 0.95     | 200     |
| macro avg    | 0.78      | 0.63   | 0.67     | 200     |
| weighted avg | 0.94      | 0.95   | 0.94     | 200     |

```
--- Evaluación del modelo con SMOTEENN ---
Matriz de confusión:
[[176 13]
 [ 5  6]]

Reporte de clasificación:
      precision    recall  f1-score   support

     0       0.97      0.93      0.95        189
     1       0.32      0.55      0.40         11

   accuracy          0.91        200
  macro avg          0.64        200
weighted avg          0.94        200


--- Evaluación del modelo con SMOTE ---
Matriz de confusión:
[[186  3]
 [ 7  4]]

     0       0.97      0.93      0.95        189
     1       0.32      0.55      0.40         11

   accuracy          0.91        200
  macro avg          0.64        200
weighted avg          0.94        200
```