Portada apuntes IA

Tabla de contenido

[Regresión 3](#_Toc183981737)

[Tabla: Modelos de Regresión y Hiperparámetros con Explicación 3](#_Toc183981738)

[Métricas para Modelos de Regresión 6](#_Toc183981739)

[Clasificación 7](#_Toc183981740)

[Tabla: Modelos de Clasificación, Hiperparámetros y Métricas 7](#_Toc183981741)

[Metricas 8](#_Toc183981742)

[Optimización de Hiperparámetros 9](#_Toc183981743)

[1. Búsqueda Exhaustiva (Grid Search) 9](#_Toc183981744)

[2. Búsqueda Aleatoria (Randomized Search) 9](#_Toc183981745)

# Regresión

## **Tabla: Modelos de Regresión y Hiperparámetros con Explicación**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Modelo** | **Hiperparámetro** | **Rango de Valores** | **Valor por Defecto** | **Explicación del Hiperparámetro** |
| **Regresión Lineal** | fit\_intercept | True, False | True | Indica si se debe calcular el intercepto del modelo. Si es False, no se añade un término constante. |
|  | normalize | True, False | False | Determina si las características deben ser normalizadas antes de ajustar el modelo. Si True, se normalizan. |
| modelo = LinearRegression(fit\_intercept=True, normalize=False) | | | | |
| **Regresión Polinómica** | degree | 1 a n (entero) | 2 | Grado del polinomio. A mayor grado, el modelo se ajusta más a los datos, pero puede haber sobreajuste. |
|  | interaction\_only | True, False | False | Si es True, solo se consideran interacciones entre las características, sin términos cuadráticos. |
|  | include\_bias | True, False | True | Si es True, se incluye un término de sesgo en el modelo. Si es False, no se incluye. |
| modelo\_polinomico = make\_pipeline( PolynomialFeatures(degree=2, interaction\_only=False, include\_bias=True), LinearRegression() | | | | |
| **Regresión Ridge** | alpha | 0.0001 a 1000 | 1.0 | Término de regularización. A mayor valor de alpha, más fuerte es la regularización, lo que reduce la magnitud de los coeficientes. |
|  | fit\_intercept | True, False | True | Si es True, el modelo ajusta un intercepto (sesgo). Si es False, no se ajusta ningún término constante. |
|  | normalize | True, False | False | Si es True, las características se normalizan antes de ajustar el modelo. |
|  | solver | 'auto', 'svd', 'cholesky', 'lsqr', 'saga' | 'auto' | Método utilizado para resolver la ecuación. auto selecciona el mejor según las características del modelo. |
|  | max\_iter | Entero positivo | None | Número máximo de iteraciones para los métodos iterativos, como 'saga'. Define el número máximo de intentos de ajuste. |
| modelo\_ridge = Ridge(alpha=1.0, fit\_intercept=True, normalize=False, solver='auto', max\_iter=None) | | | | |
| **Regresión Lasso** | alpha | 0.0001 a 1000 | 1.0 | Término de regularización. Similar a Ridge, controla la magnitud de los coeficientes, pero en este caso puede reducir algunos coeficientes a cero. |
|  | fit\_intercept | True, False | True | Si es True, se ajusta un término constante. Si es False, no se incluye el intercepto. |
|  | normalize | True, False | False | Si es True, las características se normalizan antes de ajustar el modelo. |
|  | max\_iter | Entero positivo | 1000 | Número máximo de iteraciones para el ajuste del modelo. Si el valor de max\_iter es alcanzado antes de converger, se detiene. |
|  | selection | 'cyclic', 'random' | 'cyclic' | Método de selección de características en el modelo. 'cyclic' usa todas las características de forma cíclica, mientras que 'random' selecciona aleatoriamente. |
| modelo\_lasso = Lasso(alpha=1.0, fit\_intercept=True, normalize=False, max\_iter=1000, selection='cyclic') | | | | |
| **ElasticNet** | alpha | 0.0001 a 1000 | 1.0 | Término de regularización, similar al de Ridge y Lasso. Controla el grado de penalización de los coeficientes. |
|  | l1\_ratio | 0 a 1 (proporción entre Lasso y Ridge) | 0.5 | Controla la mezcla entre las regularizaciones L1 (Lasso) y L2 (Ridge). Un valor de 1 se comporta como Lasso, mientras que 0 es igual a Ridge. |
|  | fit\_intercept | True, False | True | Si es True, el modelo ajusta un intercepto (sesgo). Si es False, no se ajusta un término constante. |
|  | normalize | True, False | False | Si es True, las características se normalizan antes de ajustar el modelo. |
|  | max\_iter | Entero positivo | 1000 | Número máximo de iteraciones para el ajuste del modelo. |
| modelo\_elasticnet = ElasticNet(alpha=1.0, l1\_ratio=0.5, fit\_intercept=True, normalize=False, max\_iter=1000) | | | | |
| **SGDRegressor** | loss | 'squared\_error', 'huber', 'epsilon\_insensitive', 'squared\_epsilon\_insensitive' | 'squared\_error' | Define la función de pérdida que se utiliza para el ajuste del modelo. 'squared\_error' es para regresión estándar, mientras que 'huber' y 'epsilon\_insensitive' son útiles para datos con valores atípicos. |
|  | alpha | 0.0001 a 1000 | 0.0001 | Término de regularización. Controla la magnitud de los coeficientes del modelo. |
|  | fit\_intercept | True, False | True | Si es True, el modelo ajusta un término constante. Si es False, no se ajusta ningún intercepto. |
|  | max\_iter | Entero positivo | 1000 | Número máximo de iteraciones para el ajuste del modelo. |
| modelo\_sgd = SGDRegressor(loss='squared\_error', alpha=0.0001, fit\_intercept=True, max\_iter=1000) | | | | |
| **LinearSVC** | C | 0.0001 a 1000 | 1.0 | Controla la regularización. Un valor más alto de C reduce la regularización y ajusta el modelo más estrechamente a los datos. |
|  | loss | 'hinge', 'squared\_hinge' | 'hinge' | Función de pérdida que se utiliza. 'hinge' es para clasificación estándar, y 'squared\_hinge' es para pérdida cuadrática. |
|  | max\_iter | Entero positivo | 1000 | Número máximo de iteraciones para el ajuste del modelo. |
| modelo\_svc = LinearSVC(C=1.0, loss='hinge', max\_iter=1000) | | | | |
| **Árboles de Decisión** | max\_depth | None, 1 a n (entero) | None | Profundidad máxima del árbol. Si es None, se expande hasta que cada hoja contenga menos de min\_samples\_split muestras. |
|  | min\_samples\_split | 2 a n (entero) | 2 | Número mínimo de muestras necesarias para dividir un nodo. |
|  | min\_samples\_leaf | 1 a n (entero) | 1 | Número mínimo de muestras necesarias para estar en una hoja del árbol. |
|  | criterion | 'gini', 'entropy' | 'gini' | Función utilizada para medir la calidad de la división en los nodos del árbol. 'gini' es para el índice de Gini, y 'entropy' para la entropía de la información. |
| modelo\_arbol = DecisionTreeClassifier(max\_depth=None, min\_samples\_split=2, min\_samples\_leaf=1, criterion='gini') | | | | |

## **Métricas para Modelos de Regresión**

|  |  |  |
| --- | --- | --- |
| **Métrica** | **Explicación** | **Código Python** |
| **MSE** | El **Error Cuadrático Medio (MSE)** mide el promedio de los errores al cuadrar las diferencias entre las predicciones y los valores reales. Penaliza más los grandes errores. | from sklearn.metrics import mean\_squared\_error mse = mean\_squared\_error(y\_train, y\_pred) |
| **R²** | El **Coeficiente de Determinación (R²)** mide qué tan bien el modelo explica la variabilidad de la variable dependiente. Un valor cercano a 1 indica que el modelo es bueno. | from sklearn.metrics import r2\_score r2 = r2\_score(y\_train, y\_pred) |
| **RMSE** | La **Raíz del Error Cuadrático Medio (RMSE)** es la raíz cuadrada del MSE. Proporciona una medida de error en las mismas unidades que los datos. | import numpy as np rmse = np.sqrt(mean\_squared\_error(y\_train, y\_pred)) |
| **MAE** | El **Error Absoluto Medio (MAE)** calcula el promedio de las diferencias absolutas entre las predicciones y los valores reales. Es más robusto frente a los valores atípicos que el MSE. | from sklearn.metrics import mean\_absolute\_error mae = mean\_absolute\_error(y\_train, y\_pred) |

# Clasificación

## **Tabla: Modelos de Clasificación, Hiperparámetros y Métricas**

|  |  |  |  |
| --- | --- | --- | --- |
| **Modelo** | **Cuándo Usarlo** | **Hiperparámetros** | **Métricas y Cuándo Usarlas** |
| **Regresión Logística** (LogisticRegression) | Problemas binarios/multiclase con datos linealmente separables o casi lineales. | - **penalty**: 'l1', 'l2', 'elasticnet', 'none' (Default: 'l2'). - **C**: Regularización (float > 0, Default: 1.0). - **solver**: 'lbfgs', 'liblinear', 'sag', 'saga' (Default depende del penalty). - **max\_iter**: Máx. iteraciones (int, Default: 100). - **multi\_class**: 'auto', 'ovr', 'multinomial' (Default: 'auto'). | - **Accuracy**: Usar en problemas balanceados para medir qué tan frecuentemente el modelo predice correctamente. - **ROC-AUC**: Ideal para problemas desbalanceados y medir la capacidad de distinguir entre clases. - **Precision/Recall**: Útil si los falsos positivos o negativos tienen un costo mayor. |
| **Árbol de Decisión** (DecisionTreeClassifier) | Relaciones no lineales en datos, interpretabilidad visual. | - **criterion**: 'gini', 'entropy' (Default: 'gini'). - **max\_depth**: Profundidad máxima (int > 0 o None, Default: None). - **min\_samples\_split**: Muestras mínimas para dividir (int > 1 o float, Default: 2). - **min\_samples\_leaf**: Muestras mínimas por hoja (int o float, Default: 1). - **max\_features**: 'auto', 'sqrt', 'log2', None (Default: None). | - **Accuracy**: Medir desempeño en problemas balanceados. - **F1-Score**: Útil para problemas desbalanceados ya que balancea precisión y exhaustividad. - **Confusión**: Para observar distribuciones de errores entre clases. - **Precision**: Si los falsos positivos son críticos (ej., diagnóstico médico). |
| **Bosques Aleatorios** (RandomForestClassifier) | Datos no lineales o de alta dimensionalidad. Modelo robusto frente al sobreajuste. | - **n\_estimators**: Nº de árboles (int > 0, Default: 100). - **criterion**: 'gini', 'entropy', 'log\_loss' (Default: 'gini'). - **max\_depth**: Profundidad máxima (int > 0 o None, Default: None). - **min\_samples\_split**: Divisiones mínimas (int > 1 o float, Default: 2). - **max\_features**: 'sqrt', 'log2', None (Default: 'sqrt'). | - **ROC-AUC**: Recomendado para problemas desbalanceados. - **F1-Score**: Evaluar el equilibrio entre precisión y recall en datos desbalanceados. - **Precision/Recall**: Dependiendo del costo de errores. - **Confusión**: Para entender errores entre clases específicas. |
| **Máquinas de Soporte Vectorial** (SVC) | Datos no linealmente separables; ideal para baja/moderada dimensionalidad. | - **C**: Regularización (float > 0, Default: 1.0). - **kernel**: 'linear', 'poly', 'rbf', 'sigmoid', 'precomputed' (Default: 'rbf'). - **degree**: Grado (si kernel='poly') (int > 0, Default: 3). - **gamma**: 'scale', 'auto', float > 0 (Default: 'scale'). - **probability**: Calcular probabilidades (True o False, Default: False). | - **ROC-AUC**: Medir capacidad del modelo para distinguir entre clases en problemas desbalanceados. - **F1-Score**: Si el dataset está desbalanceado. - **Precision**: Cuando falsos positivos tienen más impacto. - **Confusión**: Análisis detallado de predicciones erróneas. |
| **K-Vecinos Más Cercanos** (KNeighborsClassifier) | Problemas pequeños con datos bien distribuidos; no asume distribución específica de los datos. | - **n\_neighbors**: Nº de vecinos (int > 0, Default: 5). - **weights**: 'uniform', 'distance' o función personalizada (Default: 'uniform'). - **metric**: 'minkowski', 'euclidean', 'manhattan' (Default: 'minkowski'). - **p**: Parámetro para Minkowski (int > 0, Default: 2). | - **Accuracy**: Evaluar modelos en problemas balanceados. - **Confusión**: Explorar qué errores son más comunes. - **F1-Score**: Si el dataset tiene clases desbalanceadas. - **Precision/Recall**: Dependiendo de los costos de errores en el problema. |

## Metricas

|  |  |  |
| --- | --- | --- |
| **Métrica** | **Código en Python** | **Descripción** |
| **Accuracy** | clf.fit(X\_train, y\_train)  y\_pred = clf.predict(X\_test  accuracy = accuracy\_score(y\_test, y\_pred) | Proporción de predicciones correctas respecto al total. Ideal para problemas balanceados. |
| **Precision** | precision = precision\_score(y\_test, y\_pred) | Proporción de verdaderos positivos entre todas las predicciones positivas. Útil si los falsos positivos son costosos. |
| **Recall (Sensibilidad)** | recall = recall\_score(y\_test, y\_pred) | Proporción de verdaderos positivos correctamente identificados. Importante si los falsos negativos son más costosos. |
| **F1-Score** | f1 = f1\_score(y\_test, y\_pred) | Promedio armónico de precisión y sensibilidad. Recomendado para problemas desbalanceados. |
| **ROC-AUC** | roc\_auc = roc\_auc\_score(y\_test, clf.predict\_proba(X\_test)[:,1]) | Medida evaluando una capacidad efectiva para distinguir con área bajo la curva perfecta |

## **Optimización de Hiperparámetros**

### **1. Búsqueda Exhaustiva (Grid Search)**

La búsqueda exhaustiva explora todas las combinaciones posibles de los hiperparámetros dentro de los rangos especificados.

* **Código:**

python

Copiar código

from sklearn.model\_selection import GridSearchCV

# Ejemplo con Random Forest

param\_grid = {

'n\_estimators': [50, 100, 200],

'max\_depth': [None, 10, 20],

'min\_samples\_split': [2, 5, 10]

}

grid\_search = GridSearchCV(estimator=RandomForestClassifier(), param\_grid=param\_grid, cv=5, scoring='accuracy')

grid\_search.fit(X\_train, y\_train)

best\_params = grid\_search.best\_params\_

print("Mejores Hiperparámetros:", best\_params)

* **Ventajas**: Encuentra la combinación óptima de hiperparámetros dentro de los rangos definidos.
* **Desventajas**: Costosa en tiempo si los rangos de hiperparámetros son amplios.

### **2. Búsqueda Aleatoria (Randomized Search)**

La búsqueda aleatoria selecciona combinaciones al azar dentro de los rangos definidos, explorando menos combinaciones pero potencialmente encontrando una configuración adecuada más rápido.

* **Código:**

python

Copiar código

from sklearn.model\_selection import RandomizedSearchCV

from scipy.stats import randint

# Ejemplo con Random Forest

param\_dist = {

'n\_estimators': randint(50, 200),

'max\_depth': [None, 10, 20, 30],

'min\_samples\_split': randint(2, 10)

}

random\_search = RandomizedSearchCV(estimator=RandomForestClassifier(), param\_distributions=param\_dist, n\_iter=50, cv=5, scoring='accuracy')

random\_search.fit(X\_train, y\_train)

best\_params = random\_search.best\_params\_

print("Mejores Hiperparámetros:", best\_params)

* **Ventajas**: Más rápido que la búsqueda exhaustiva, especialmente en hiperparámetros con muchos valores posibles.
* **Desventajas**: Puede no explorar todas las configuraciones óptimas.

# Ensamblaje

## **Modelos de Ensamblaje y Hiperparámetros con Explicación**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Modelo** | **Hiperparámetro** | **Rango de Valores** | **Valor por Defecto** | **Explicación del Hiperparámetro** |
| **Bagging** | n\_estimators | 1 a n (entero positivo) | 10 | Número de clasificadores o estimadores base a entrenar en paralelo. Un valor mayor mejora el rendimiento hasta cierto punto. |
|  | max\_samples | 1 a n (fracción o número entero de muestras) | 1.0 | Fracción de muestras utilizadas para entrenar cada clasificador. Un valor menor reduce el sobreajuste pero también puede reducir la precisión. |
|  | max\_features | 1 a n (fracción o número entero de características) | 1.0 | Número de características utilizadas por cada estimador base. Puede ser un número o una fracción de las características disponibles. |
|  | bootstrap | True, False | True | Si se utiliza muestreo con reemplazo (True) o sin reemplazo (False) para crear las muestras de entrenamiento. |
| from sklearn.ensemble import BaggingClassifier  from sklearn.tree import DecisionTreeClassifier  import numpy as np  # Datos de ejemplo  X = np.array([[1, 2], [2, 3], [3, 4], [4, 5]])  y = np.array([0, 1, 0, 1])  # Crear el modelo BaggingClassifier con los hiperparámetros especificados  modelo\_bagging = BaggingClassifier(  base\_estimator=DecisionTreeClassifier(),  n\_estimators=10, # Número de clasificadores base  max\_samples=1.0, # Fracción de muestras para entrenar cada clasificador  max\_features=1.0, # Número de características utilizadas por cada estimador base  bootstrap=True # Muestreo con reemplazo  )  # Ajustar el modelo  modelo\_bagging.fit(X, y)  # Hacer predicciones  predicciones = modelo\_bagging.predict(X)  print("Predicciones:", predicciones) | | | | |
| **Random Forest** | n\_estimators | 1 a n (entero positivo) | 100 | Número de árboles de decisión en el bosque. Un valor más alto puede mejorar el rendimiento pero aumenta el tiempo de computación. |
|  | max\_depth | None, 1 a n (entero) | None | Profundidad máxima de los árboles. Si es None, se expanden hasta que cada hoja contenga menos de min\_samples\_split muestras. |
|  | min\_samples\_split | 2 a n (entero) | 2 | Número mínimo de muestras necesarias para dividir un nodo. Controla el tamaño de los árboles. |
|  | min\_samples\_leaf | 1 a n (entero) | 1 | Número mínimo de muestras necesarias para estar en una hoja del árbol. |
|  | max\_features | 1 a n (fracción o número entero de características) | sqrt | Número máximo de características para dividir cada nodo. 'sqrt' es el valor predeterminado, que selecciona la raíz cuadrada de las características. |
| from sklearn.ensemble import RandomForestClassifier  import numpy as np  # Datos de ejemplo  X = np.array([[1, 2], [2, 3], [3, 4], [4, 5]])  y = np.array([0, 1, 0, 1])  # Crear el modelo RandomForestClassifier con los hiperparámetros especificados  modelo\_random\_forest = RandomForestClassifier(  n\_estimators=100, # Número de árboles en el bosque  max\_depth=None, # Profundidad máxima de los árboles  min\_samples\_split=2, # Número mínimo de muestras para dividir un nodo  min\_samples\_leaf=1, # Número mínimo de muestras en una hoja del árbol  max\_features='sqrt' # Número máximo de características para dividir cada nodo  )  # Ajustar el modelo  modelo\_random\_forest.fit(X, y)  # Hacer predicciones  predicciones = modelo\_random\_forest.predict(X)  print("Importancia de características:", modelo\_random\_forest.feature\_importances\_)  print("Predicciones:", predicciones) | | | | |
| **AdaBoost** | n\_estimators | 1 a n (entero positivo) | 50 | Número de estimadores base a entrenar. Un valor mayor puede mejorar el rendimiento, pero también puede aumentar el riesgo de sobreajuste. |
|  | learning\_rate | 0 a 1 (flotante) | 1.0 | Tasa de aprendizaje que ajusta la contribución de cada estimador al modelo final. Un valor pequeño puede mejorar el rendimiento, pero aumentar el tiempo de entrenamiento. |
|  | algorithm | 'SAMME', 'SAMME.R' | 'SAMME.R' | Algoritmo utilizado para el entrenamiento. 'SAMME.R' es más rápido y eficaz, mientras que 'SAMME' es más clásico. |
| from sklearn.ensemble import AdaBoostClassifier  from sklearn.tree import DecisionTreeClassifier  import numpy as np  # Datos de ejemplo  X = np.array([[1, 2], [2, 3], [3, 4], [4, 5]])  y = np.array([0, 1, 0, 1])  # Crear el modelo AdaBoostClassifier con los hiperparámetros especificados  modelo\_adaboost = AdaBoostClassifier(  base\_estimator=DecisionTreeClassifier(),  n\_estimators=50, # Número de estimadores base  learning\_rate=1.0, # Tasa de aprendizaje  algorithm='SAMME.R' # Algoritmo utilizado para el entrenamiento  )  # Ajustar el modelo  modelo\_adaboost.fit(X, y)  # Hacer predicciones  predicciones = modelo\_adaboost.predict(X)  print("Predicciones:", predicciones) | | | | |
| **Gradient Boosting** | n\_estimators | 1 a n (entero positivo) | 100 | Número de árboles a entrenar. Un número mayor generalmente mejora el rendimiento, pero puede aumentar el riesgo de sobreajuste. |
|  | learning\_rate | 0.0 a 1.0 (flotante) | 0.1 | Tasa de aprendizaje que controla el impacto de cada estimador. Un valor pequeño permite que el modelo se entrene durante más iteraciones sin sobreajustar. |
|  | max\_depth | 1 a n (entero positivo) | 3 | Profundidad máxima de cada árbol. El valor predeterminado ayuda a evitar el sobreajuste en la mayoría de los casos. |
|  | min\_samples\_split | 2 a n (entero) | 2 | Número mínimo de muestras necesarias para dividir un nodo. Esto controla el crecimiento de los árboles. |
| from sklearn.ensemble import GradientBoostingClassifier  import numpy as np  # Datos de ejemplo  X = np.array([[1, 2], [2, 3], [3, 4], [4, 5]])  y = np.array([0, 1, 0, 1])  # Crear el modelo GradientBoostingClassifier con los hiperparámetros especificados  modelo\_gb = GradientBoostingClassifier(  n\_estimators=100, # Número de árboles a entrenar  learning\_rate=0.1, # Tasa de aprendizaje  max\_depth=3, # Profundidad máxima de cada árbol  min\_samples\_split=2 # Número mínimo de muestras para dividir un nodo  )  # Ajustar el modelo  modelo\_gb.fit(X, y)  # Hacer predicciones  predicciones = modelo\_gb.predict(X)  print("Importancia de características:", modelo\_gb.feature\_importances\_)  print("Predicciones:", predicciones) | | | | |
| **XGBoost** | n\_estimators | 1 a n (entero positivo) | 100 | Número de árboles de decisión a entrenar. Un número más alto puede mejorar la precisión, pero aumenta el tiempo de entrenamiento. |
|  | learning\_rate | 0.0 a 1.0 (flotante) | 0.3 | Tasa de aprendizaje que controla el impacto de cada árbol. Un valor pequeño reduce el sobreajuste y mejora la generalización. |
|  | max\_depth | 1 a n (entero positivo) | 6 | Profundidad máxima de los árboles. Un valor mayor permite modelos más complejos, pero puede causar sobreajuste. |
|  | subsample | 0.5 a 1.0 (flotante) | 1.0 | Fracción de muestras utilizada para entrenar cada árbol. Si es menor que 1.0, reduce el riesgo de sobreajuste. |
|  | colsample\_bytree | 0.5 a 1.0 (flotante) | 1.0 | Fracción de características utilizadas para entrenar cada árbol. Reducir este valor ayuda a mejorar la generalización. |
| from xgboost import XGBClassifier  import numpy as np  # Datos de ejemplo  X = np.array([[1, 2], [2, 3], [3, 4], [4, 5]])  y = np.array([0, 1, 0, 1])  # Crear el modelo XGBClassifier con los hiperparámetros especificados  modelo\_xgb = XGBClassifier(  n\_estimators=100, # Número de árboles de decisión  learning\_rate=0.3, # Tasa de aprendizaje  max\_depth=6, # Profundidad máxima de los árboles  subsample=1.0, # Fracción de muestras utilizada  colsample\_bytree=1.0 # Fracción de características utilizada  )  # Ajustar el modelo  modelo\_xgb.fit(X, y)  # Hacer predicciones  predicciones = modelo\_xgb.predict(X)  print("Importancia de características:", modelo\_xgb.feature\_importances\_)  print("Predicciones:", predicciones) | | | | |
| **LightGBM** | n\_estimators | 1 a n (entero positivo) | 100 | Número de árboles de decisión. |
|  | learning\_rate | 0.0 a 1.0 (flotante) | 0.1 | Tasa de aprendizaje que ajusta la contribución de cada árbol al modelo final. |
|  | max\_depth | -1, 1 a n (entero positivo) | -1 | Profundidad máxima de los árboles. Si es -1, no hay límite. |
|  | num\_leaves | 31 a 2^max\_depth (entero) | 31 | Número de hojas por árbol. Un valor mayor permite árboles más complejos. |
| from lightgbm import LGBMClassifier  import numpy as np  # Datos de ejemplo  X = np.array([[1, 2], [2, 3], [3, 4], [4, 5]])  y = np.array([0, 1, 0, 1])  # Crear el modelo LGBMClassifier con los hiperparámetros especificados  modelo\_lgbm = LGBMClassifier(  n\_estimators=100, # Número de árboles de decisión  learning\_rate=0.1, # Tasa de aprendizaje  max\_depth=-1, # Profundidad máxima de los árboles  num\_leaves=31 # Número de hojas por árbol  )  # Ajustar el modelo  modelo\_lgbm.fit(X, y)  # Hacer predicciones  predicciones = modelo\_lgbm.predict(X)  print("Importancia de características:", modelo\_lgbm.feature\_importances\_)  print("Predicciones:", predicciones) | | | | |
| **CatBoost** | iterations | 1 a n (entero positivo) | 1000 | Número de árboles a entrenar. |
|  | learning\_rate | 0.0 a 1.0 (flotante) | 0.03 | Tasa de aprendizaje. |
|  | depth | 1 a 16 (entero positivo) | 6 | Profundidad máxima de los árboles. A mayor profundidad, más complejo es el modelo, pero mayor riesgo de sobreajuste. |
|  | l2\_leaf\_reg | 0 a 100 (flotante) | 3.0 | Término de regularización L2 que controla el sobreajuste. |
|  | border\_count | 32 a 255 (entero positivo) | 128 | Número de valores discretos para representar las características continuas. |
| # Datos de ejemplo  X = np.array([[1, 2], [2, 3], [3, 4], [4, 5]])  y = np.array([0, 1, 0, 1])  # Crear el modelo CatBoostClassifier con los hiperparámetros especificados  modelo\_catboost = CatBoostClassifier(  iterations=1000, # Número de árboles a entrenar  learning\_rate=0.03, # Tasa de aprendizaje  depth=6, # Profundidad máxima de los árboles  l2\_leaf\_reg=3.0, # Término de regularización L2  border\_count=128 # Número de valores discretos para características continuas  )  # Ajustar el modelo (nota: verbose=0 para suprimir salida durante el ajuste)  modelo\_catboost.fit(X, y, verbose=0)  # Hacer predicciones  predicciones = modelo\_catboost.predict(X)  print("Predicciones:", predicciones) | | | | |
| **StackingClassifier** | estimators | List of tuples (modelo, nombre) | None | Lista de modelos base (estimadores). Cada elemento de la lista es una tupla con el nombre del modelo y el estimador correspondiente. |
|  | final\_estimator | Clasificador (puede ser cualquier estimador) | LogisticRegression() | Clasificador final que se utiliza para combinar las predicciones de los modelos base. Por defecto es una regresión logística. |
|  | cv | None, int (entero positivo o Cross-Validation Generator) | 5 | Número de divisiones para la validación cruzada utilizada para entrenar los modelos base. |
|  | stack\_method | 'auto', 'predict\_proba', 'predict' | 'auto' | Método para combinar las predicciones de los modelos base. 'auto' elige automáticamente el mejor. |
|  | passthrough | True, False | False | Si es True, las predicciones de los estimadores base se pasan directamente al final\_estimator además de las predicciones combinadas. |
|  | n\_jobs | -1 a n (entero) | None | Número de núcleos a utilizar para el entrenamiento paralelo de los modelos base. Si es -1, se usan todos los núcleos. |
| # Datos de ejemplo  X = np.array([[1, 2], [2, 3], [3, 4], [4, 5]])  y = np.array([0, 1, 0, 1])  # Crear los estimadores base  estimadores = [  ('decision\_tree', DecisionTreeClassifier()),  ('svm', SVC(probability=True))  ]  # Crear el modelo StackingClassifier con los hiperparámetros especificados  modelo\_stacking = StackingClassifier(  estimators=estimadores,  final\_estimator=LogisticRegression(), # Clasificador final  cv=5, # Validación cruzada con 5 divisiones  stack\_method='auto', # Método para combinar las predicciones  passthrough=False, # No pasar las predicciones de los estimadores base al final\_estimator  n\_jobs=-1 # Usar todos los núcleos disponibles para el entrenamiento  )  # Ajustar el modelo  modelo\_stacking.fit(X, y)  # Hacer predicciones  predicciones = modelo\_stacking.predict(X)  print("Predicciones:", predicciones) | | | | |

# Reducción de dimensionalidad

|  |  |  |
| --- | --- | --- |
| **Elemento** | **Descripción** | **Código en Python** |
| **PCA (Principal Component Analysis)** | Técnica de reducción de dimensionalidad que transforma las características originales en un conjunto de componentes principales, que explican la mayor parte de la varianza. Es útil cuando se tienen muchas variables correlacionadas. | from sklearn.decomposition import PCA  pca = PCA(n\_components=2)  X\_reduced = pca.fit\_transform(X\_train) |
| **n\_components** | Número de componentes principales a retener en la reducción de la dimensionalidad. Si se establece un valor entre 0 y 1, se seleccionan suficientes componentes para explicar esa proporción de la varianza. | PCA(n\_components=0.95) (explica el 95% de la varianza) |
| **svd\_solver** | Algoritmo utilizado para calcular los componentes principales. Los valores posibles son 'auto', 'full', 'arpack' y 'randomized'. Dependiendo de los datos, algunos algoritmos pueden ser más rápidos o más precisos. | PCA(svd\_solver='randomized') |
| **whiten** | Si es True, los componentes principales serán reescalados (whitened) para tener varianza unitaria. Esto puede ser útil para algunos modelos de aprendizaje automático que requieren variables con varianza similar. | PCA(whiten=True) |
| **copy** | Si es True, se hace una copia de los datos originales al realizar la transformación. Si es False, la transformación se realiza en los datos originales (puede ahorrar memoria). | PCA(copy=True) |
| **random\_state** | Semilla para asegurar la reproducibilidad de la descomposición si se usa el svd\_solver='randomized'. Especifica la semilla del generador de números aleatorios. | PCA(random\_state=42) |
| **mean\_centering** | Centra los datos (resta la media de cada característica) antes de aplicar PCA. Es la práctica estándar en la mayoría de los casos. Este parámetro se aplica de manera predeterminada. | PCA() (automáticamente centra los datos) |