Contenido

[1. Regresión 3](#_Toc183692994)

[¿Cuándo Utilizar Regresión? 3](#_Toc183692995)

[Conceptos Clave en Regresión 3](#_Toc183692996)

[2. Tipos de Regresión 3](#_Toc183692997)

[2.1. Regresión Lineal 4](#_Toc183692998)

[2.1.1. Regresión Lineal Simple 4](#_Toc183692999)

[2.1.2. Regresión Lineal Múltiple 5](#_Toc183693000)

[2.2. Regresión Polinomial 7](#_Toc183693001)

[2.3. Regresión Logística 10](#_Toc183693002)

[2.4. Tabla comparativa de Regresiones 10](#_Toc183693003)

[Resumen de las Diferencias 10](#_Toc183693004)

[2.5. Flujo de Trabajo con Técnicas de Optimización y Ajuste en Regresión Lineal 11](#_Toc183693005)

[3. Métodos de optimización 11](#_Toc183693006)

[Descenso de Gradiente: 11](#_Toc183693007)

[¿En Qué Parte del Proceso se Usan los Métodos de Optimización? 15](#_Toc183693008)

[Resumen del Proceso 16](#_Toc183693009)

[Comparación entre tres métodos de optimización para resolver problemas de regresión lineal: 16](#_Toc183693010)

[4. Regularización 16](#_Toc183693011)

[Tipos de Regularización en Regresión 17](#_Toc183693012)

[Objetivo de la Regularización 17](#_Toc183693013)

[4.1. Regularización Ridge(L2) 17](#_Toc183693014)

[Resumen 17](#_Toc183693015)

[Características: 17](#_Toc183693016)

[Código ejemplo Python: 17](#_Toc183693017)

[Tabla comparativa de posibles escenarios 18](#_Toc183693018)

[Explicación de la Tabla 18](#_Toc183693019)

[4.2. Regularización Lasso(L1) 19](#_Toc183693020)

[Resumen 19](#_Toc183693021)

[Características: 19](#_Toc183693022)

[Código ejemplo Python: 19](#_Toc183693023)

[Tabla comparativa de posibles escenarios 19](#_Toc183693024)

[Explicación de la Tabla Lasso 20](#_Toc183693025)

[4.3. Regularización Elastic Net 20](#_Toc183693026)

[Resumen 20](#_Toc183693027)

[Gestión del hiperparámetros R 20](#_Toc183693028)

[Código ejemplo Python 21](#_Toc183693029)

[Tabla comparativa de posibles escenarios 21](#_Toc183693030)

[Explicación de la Tabla Elastic Net 22](#_Toc183693031)

[Resumen de Acciones en Ambas Regularizaciones 22](#_Toc183693032)

[4.4. Detención temprana 22](#_Toc183693033)

[5. Algoritmos de trabajo Regresión 24](#_Toc183693034)

[6. Evaluación 25](#_Toc183693035)

[6.1. Métricas Comunes para la Evaluación de Modelos de Regresión 25](#_Toc183693036)

[6.2. Métodos de Validación 26](#_Toc183693037)

[6.3. Curvas de Aprendizaje 26](#_Toc183693038)

[6.4. Comparación entre Modelos 27](#_Toc183693039)

[7. Ejemplo Código Python 28](#_Toc183693040)

# Regresión

La **regresión** es una técnica de análisis estadístico y aprendizaje automático utilizada para modelar la **relación entre una variable dependiente (respuesta)** y **una o más variables independientes (predictoras)**. Su propósito principal es predecir valores continuos de la variable dependiente en función de los valores de las variables independientes. A diferencia de la clasificación, que predice categorías, la regresión predice valores continuos, lo que la hace ideal para problemas como:

* Predicción de precios (casas, acciones, productos).
* Estimaciones de rendimiento (deportivos, financieros, académicos).
* Modelado de relaciones cuantitativas (crecimiento de población, evolución de ventas).

### ¿Cuándo Utilizar Regresión?

La regresión es adecuada cuando:

* El objetivo es predecir un valor continuo: Por ejemplo, estimar la temperatura o las ganancias.
* Existe una relación entre las variables: Especialmente si las variables independientes afectan de forma medible al resultado.
* Se necesita interpretar los resultados: Modelos simples como la regresión lineal permiten entender cómo cambian las predicciones en función de los cambios en las variables.

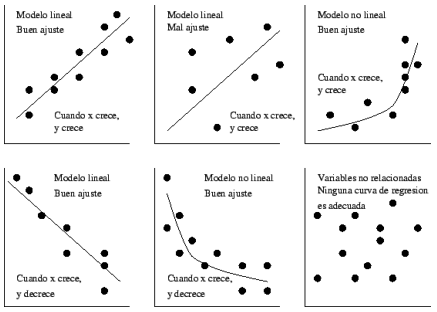
### Conceptos Clave en Regresión

* **Función de Pérdida**: Durante el entrenamiento, el modelo minimiza una función de pérdida, como el **error cuadrático medio (MSE)**, que mide la diferencia entre los valores predichos y los valores reales.
* **Coeficientes o Parámetros**: Los coeficientes de cada variable independiente representan su peso o importancia en la predicción de la variable dependiente.
* **Regularización**: Técnica que agrega una penalización a los coeficientes para evitar el sobreajuste y mejorar la capacidad de generalización del modelo.

## Tipos de Regresión

Existen diversos métodos de regresión, cada uno adecuado para diferentes tipos de problemas:

* **Regresión Lineal Simple**
* **Regresión Lineal Múltiple**
* **Regresión Polinomial**
* **Regresión Logística**



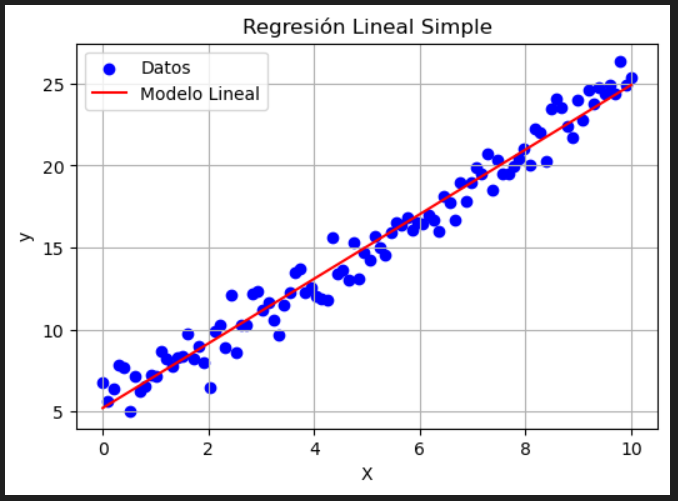
# Regresión Lineal

## Regresión Lineal Simple

* **Propósito**: Modelar una **relación lineal** entre una variable dependiente y y una sola variable independiente x. Se representa con una línea recta
* **Ecuación General**:



* **Características**:
  + Es el modelo más sencillo de regresión, con solo una variable independiente.
  + La relación entre x e y es **lineal** (representada por una línea recta).
  + Solo tiene un coeficiente  que representa la pendiente de la línea.
* **Ejemplo**: Predecir el precio de un producto en función de una sola característica, como la cantidad de unidades vendidas. La relación se espera que sea una línea recta entre las dos variables.
* **Clase LinearRegression**: La clase de LinearRegression de Scikit-Learn tienen un coste computacional de O(n2). Si duplicamos el número de características, multiplicamos el tiempo de computación por 4, aproximadamente.
* **Visualización**:



* **Ejemplo código Python:**

import numpy as np

from sklearn.linear\_model import LinearRegression

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

from sklearn.metrics import mean\_squared\_error, r2\_score

# Datos de ejemplo

X = np.array([[1], [2], [3], [4], [5]]) # Variable independiente (e.g., unidades vendidas)

y = np.array([2.5, 4.5, 6.5, 8.5, 10.5]) # Variable dependiente (e.g., precios)

# Dividir los datos en conjunto de entrenamiento y prueba

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

# Crear el modelo de regresión lineal

model = LinearRegression()

# Entrenar el modelo

model.fit(X\_train, y\_train)

# Realizar predicciones

y\_pred = model.predict(X\_test)

# Evaluar el modelo

mse = mean\_squared\_error(y\_test, y\_pred)

r2 = r2\_score(y\_test, y\_pred)

print("Coeficiente (Pendiente):", model.coef\_[0])

print("Intercepto:", model.intercept\_)

print("Error Cuadrático Medio (MSE):", mse)

print("Coeficiente de Determinación (R²):", r2)

# Predicción para un nuevo dato

nuevo\_valor = np.array([[6]])

prediccion = model.predict(nuevo\_valor)

print(f"Predicción para {nuevo\_valor[0][0]} unidades vendidas:", prediccion[0])

#Resultados

Coeficiente (Pendiente): 1.9999999999999996

Intercepto: 0.5000000000000018

Error Cuadrático Medio (MSE): 7.888609052210118e-31

Coeficiente de Determinación (R²): nan

Predicción para 6 unidades vendidas: 12.499999999999998

## Regresión Lineal Múltiple

* **Propósito**: Modelar una relación **lineal** entre la variable dependiente y y **varias variables independientes** (o características)

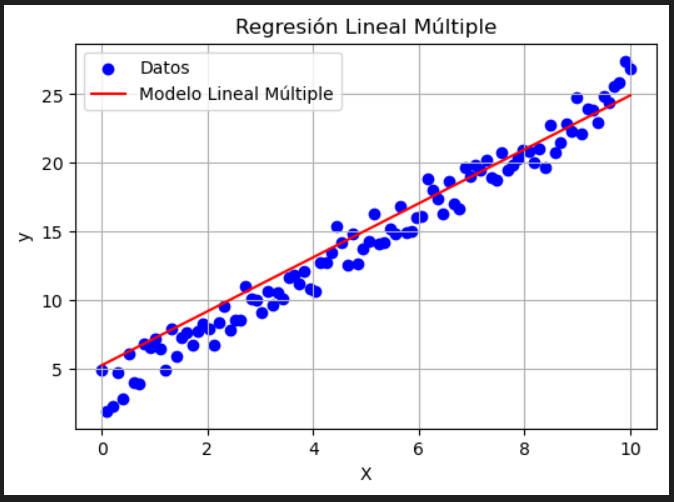
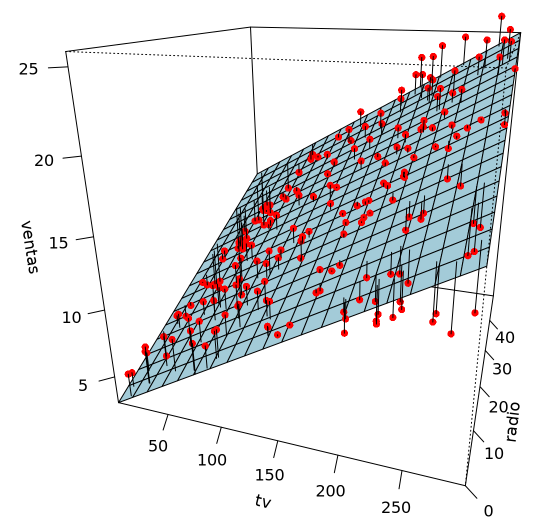
La relación es una extensión de la lineal en un espacio **multidimensional.**

* **Ecuación General:**



* **Características**:
* Cada xi representa una **característica distinta** o variable independiente.
* La relación entre y y cada xi​ es **lineal**. Es decir, el efecto de cada xi ​ en y se representa con un término θixi​, sin potencias ni combinaciones de xi.
* **No incluye términos no lineales** de cada x, como 
* **Clase LinearRegression:**
* **Ejemplo**: Predecir el precio de una casa en función de variables como el tamaño (metros cuadrados), el número de habitaciones y la edad de la casa. Cada variable independiente tiene un efecto lineal en el precio.
* **Visualización**:

*Regresión Linear Multiple 2D (grado 2) / Regresión Linear Multiple 3D (grado 3)*

* **Ejemplo código Python:**

import numpy as np

from sklearn.linear\_model import LinearRegression

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

from sklearn.metrics import mean\_squared\_error, r2\_score

# Datos de ejemplo

# Variables independientes (e.g., tamaño y número de habitaciones)

X = np.array([

[50, 1], # Tamaño: 50 m², Habitaciones: 1

[60, 2],

[75, 2],

[80, 3],

[100, 4],

[120, 4]

])

# Variable dependiente (e.g., precio del inmueble)

y = np.array([150000, 180000, 250000, 300000, 400000, 450000])

# Dividir los datos en conjunto de entrenamiento y prueba

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

# Crear el modelo de regresión lineal

model = LinearRegression()

# Entrenar el modelo

model.fit(X\_train, y\_train)

# Realizar predicciones

y\_pred = model.predict(X\_test)

# Evaluar el modelo

mse = mean\_squared\_error(y\_test, y\_pred)

r2 = r2\_score(y\_test, y\_pred)

print("Coeficientes:", model.coef\_) # Pendientes para cada variable independiente

print("Intercepto:", model.intercept\_)

print("Error Cuadrático Medio (MSE):", mse)

print("Coeficiente de Determinación (R²):", r2)

# Predicción para un nuevo dato

nuevo\_valor = np.array([[85, 3]]) # Tamaño: 85 m², Habitaciones: 3

prediccion = model.predict(nuevo\_valor)

print(f"Predicción para un inmueble de {nuevo\_valor[0][0]} m² y {nuevo\_valor[0][1]} habitaciones:", prediccion[0])

#Resultado

Coeficientes: [ 2681.15942029 40942.02898551]

Intercepto: -34420.28985507239

Error Cuadrático Medio (MSE): 445759819.3656788

Coeficiente de Determinación (R²): -0.9811547527363502

Predicción para un inmueble de 85 m² y 3 habitaciones: 316304.3478260869

# Regresión Polinomial

* **Propósito:** Modelar una relación no lineal entre la variable dependiente y y una o más variables independientes mediante el uso de **potencias** de las variables independientes. Extiende la regresión lineal añadiendo términos polinómicos.
* **Ecuación General:**
  + **(Polinomio de grado 2)**:



* + **(Polinomio de grado nnn)**:

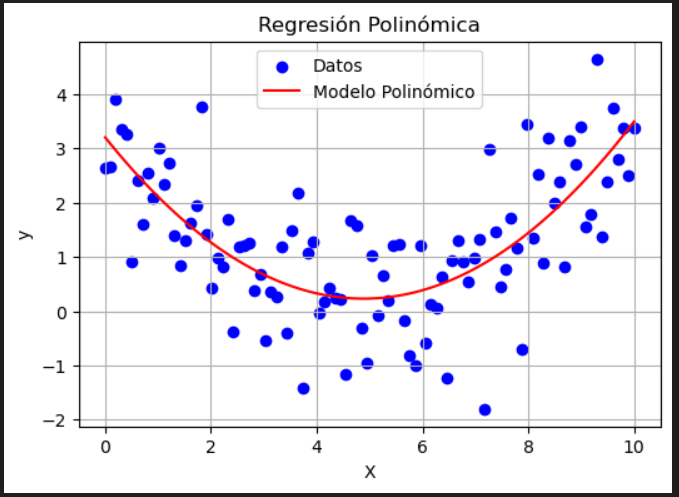


* **Características**:
  + La regresión polinómica **incluye potencias** de la misma variable x (por ejemplo, x, , etc.) para capturar relaciones **no lineales** entre x e y.
  + Si la regresión polinómica se usa con más de una variable independiente, podría incluir términos como  ​, etc., para capturar interacciones entre las variables.
  + Permite que el modelo ajuste **curvas** en lugar de solo líneas rectas.
  + Cuando hay múltiples características, la regresión polinomial es capaz de encontrar relaciones entre características
  + La importancia de los grados. Este modelo de regresión polinomial está sobreajustando mucho los datos de entrenamiento, mientras que el modelo lineal los está subajustando

Gráfico, Gráfico de dispersión

Descripción generada automáticamente

* **Ejemplo**: Predecir la altura de una planta en función del tiempo (días). Si el crecimiento de la planta sigue una relación cuadrática o cúbica, usar un polinomio de grado 2 o 3 puede ayudar a capturar mejor esa relación no lineal.
* **Clase PolynomialFeatures:** de Scikit-Learn para transformar nuestros datos de entrenamiento, añadiendo el cuadrado (polinomio de segundo grado) de cada característica en el conjunto de entrenamiento como una característica nueva (en este caso, solo hay una característica)
* **Visualización**:



* **Ejemplo código Python:**

import numpy as np

import matplotlib.pyplot as plt

from sklearn.preprocessing import PolynomialFeatures

from sklearn.linear\_model import LinearRegression

from sklearn.metrics import mean\_squared\_error, r2\_score

# Datos de ejemplo (relación no lineal)

X = np.array([[1], [2], [3], [4], [5], [6], [7], [8], [9], [10]])

y = np.array([1, 4, 9, 16, 25, 36, 49, 64, 81, 100]) # Relación cuadrática: y = x^2

# Crear características polinómicas de grado 2

poly = PolynomialFeatures(degree=2)

X\_poly = poly.fit\_transform(X)

# Crear y entrenar el modelo de regresión lineal con características polinómicas

model = LinearRegression()

model.fit(X\_poly, y)

# Predicciones

y\_pred = model.predict(X\_poly)

# Evaluar el modelo

mse = mean\_squared\_error(y, y\_pred)

r2 = r2\_score(y, y\_pred)

print("Coeficientes:", model.coef\_)

print("Intercepto:", model.intercept\_)

print("Error Cuadrático Medio (MSE):", mse)

print("Coeficiente de Determinación (R²):", r2)

# Visualizar los resultados

plt.scatter(X, y, color='blue', label='Datos reales')

plt.plot(X, y\_pred, color='red', label='Predicción (Regresión Polinomial)')

plt.title('Regresión Polinomial')

plt.xlabel('Variable Independiente (X)')

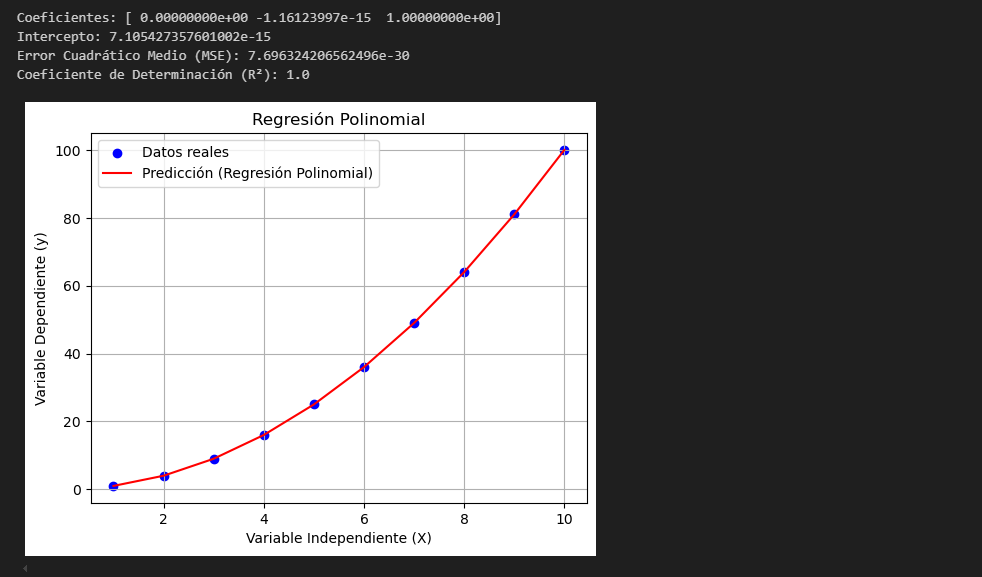
plt.ylabel('Variable Dependiente (y)')

plt.legend()

plt.grid()

plt.show()

#Resultado



* **Validación cruzada:** La validación cruzada es una técnica fundamental en modelos de regresión para evaluar su capacidad de generalización y evitar problemas como el sobreajuste o subajuste. En este contexto, el objetivo es medir qué tan bien el modelo predice valores continuos en datos no vistos
* **Tipos Comunes de Validación Cruzada**:
  + **K-Fold Cross-Validation**:
    - Divide los datos en K subconjuntos (folds).
    - Entrena el modelo en K−1 folds y lo valida en el fold restante.
    - Repite este proceso KKK veces y promedia los resultados.
  + **Leave-One-Out Cross-Validation (LOOCV)**:
    - Usa un solo ejemplo como validación y entrena en el resto de los datos.
* **Propósito**:
  + Evaluar la capacidad del modelo para generalizar a nuevos datos.
  + Detectar sobreajuste o subajuste sin depender de una única partición fija.
* **Herramienta en scikit-learn**:
  + La función cross\_val\_score() facilita la implementación de validación cruzada:
* Utilizando la validación cruzada podemos obtener una estimación del rendimiento de la generalización de un modelo:
* Si un modelo tiene un buen rendimiento en los datos de entrenamiento, pero generaliza mal según las métricas de validación cruzada, entonces está sobreajustando.
* Si tiene un mal rendimiento en ambos casos, entonces está subajustando.
* **Codigo ejemplo Python:**

# Ejemplo K-Fold Cross-Validation

from sklearn.model\_selection import cross\_val\_score

from sklearn.linear\_model import LinearRegression

import numpy as np

# Datos de ejemplo

X = np.array([[1], [2], [3], [4], [5], [6], [7], [8], [9], [10]]) # Variable independiente

y = np.array([1.5, 2.5, 3.0, 4.0, 5.5, 6.5, 7.0, 8.0, 9.5, 10.5]) # Variable dependiente

# Modelo

model = LinearRegression()

# Validación cruzada con 5 folds

scores = cross\_val\_score(model, X, y, cv=5, scoring='neg\_mean\_squared\_error')

print("Errores Cuadráticos Medios por Fold:", -scores)

print("MSE Promedio:", -scores.mean())

# Ejemplo Leave-One-Out Cross-Validation (LOOCV)

from sklearn.model\_selection import LeaveOneOut

loo = LeaveOneOut()

scores = cross\_val\_score(model, X, y, cv=loo, scoring='neg\_mean\_squared\_error')

print("MSE promedio con LOOCV:", -scores.mean())

* **Curvas de aprendizaje:** Las curvas de aprendizaje son gráficos que muestran cómo varía el error de entrenamiento y el error de validación del modelo a medida que se entrena. Estos gráficos permiten identificar si un modelo está **sobreajustado** o **subajustado**.
* **¿Cómo se construyen?**:
  + Evalúa el modelo en intervalos regulares durante el entrenamiento, tanto en el conjunto de entrenamiento como en el de validación.
  + Traza el gráfico de los errores obtenidos (e.g., error cuadrático medio o precisión).
* **Propósito**:
  + Identificar **sobreajuste**: Ocurre cuando el error de entrenamiento es mucho menor que el error de validación.
  + Identificar **subajuste**: Ocurre cuando ambos errores son altos y el modelo no captura adecuadamente la complejidad de los datos.
* **Herramienta en scikit-learn**:
  + La función learning\_curve() entrena y evalúa el modelo en subconjuntos crecientes del conjunto de entrenamiento y genera datos para trazar las curvas.
  + Ejemplo básico python:

from sklearn.model\_selection import learning\_curve

import matplotlib.pyplot as plt

# Datos y modelo

X, y = ... # Tus datos

model = ... # Tu modelo (e.g., LinearRegression())

# Obtener resultados

train\_sizes, train\_scores, val\_scores = learning\_curve(model, X, y, cv=5)

# Graficar las curvas

plt.plot(train\_sizes, train\_scores.mean(axis=1), label="Error de Entrenamiento")

plt.plot(train\_sizes, val\_scores.mean(axis=1), label="Error de Validación")

plt.legend()

plt.title("Curvas de Aprendizaje")

plt.xlabel("Tamaño del conjunto de entrenamiento")

plt.ylabel("Precisión")

plt.grid()

plt.show()

# Regresión Logística

**Regresión Logística (aunque no es exactamente regresión en el sentido clásico):** Se usa para problemas de clasificación binaria, donde la variable dependiente es categórica. Modela la probabilidad de pertenencia a una categoría.

# Tabla comparativa de Regresiones

| **Aspecto** | **Regresión Lineal Simple** | **Regresión Lineal Múltiple** | **Regresión Polinómica** |
| --- | --- | --- | --- |
| *Tipo de Relación* | Lineal | Lineal | No lineal |
| *Variables Independientes* | Una sola variable x | Varias variables | Una o más variables elevadas a potencias |
| *Ecuación* |  |  |  |
| *Complejidad* | Baja | Moderada | Alta |
| *Ejemplo de Uso* | Predecir precio en función de una sola característica | Predecir precio en función de varias características | Modelar crecimiento o relaciones no lineales |
| *Limitaciones* | Solo una variable, no captura relaciones complejas | No captura relaciones no lineales | Mayor riesgo de sobreajuste con grados altos |

### ****Resumen de las Diferencias****

* **Regresión Lineal Simple**: Usa una sola variable independiente con una relación lineal. Es el modelo más básico y fácil de interpretar.
* **Regresión Lineal Múltiple**: Extiende la regresión lineal simple para incluir varias variables independientes, pero mantiene una relación lineal con cada variable.
* **Regresión Polinómica**: Introduce potencias de las variables para capturar relaciones no lineales, permitiendo modelar patrones más complejos, aunque con mayor riesgo de sobreajuste si el grado es alto.

# Flujo de Trabajo con Técnicas de Optimización y Ajuste en Regresión Lineal

1. **Inicia con una Regresión Lineal Básica**:

* En general, comienzas con una regresión lineal simple o múltiple para ver si el modelo ajusta bien los datos y proporciona buenos resultados.
* Realiza una primera evaluación para ver los errores de entrenamiento y validación y analizar si el modelo generaliza bien o si hay indicios de sobreajuste o subajuste.

1. **Aplicación de Métodos de Optimización**:

* Si notas que el modelo no está ajustando bien, puedes **utilizar métodos de optimización** como el **descenso de gradiente por lotes** o el **descenso de gradiente estocástico** para ajustar los coeficientes.
* Estos métodos iterativos pueden mejorar el ajuste, especialmente cuando se trabaja con conjuntos de datos grandes y no se puede usar un método exacto (como una solución cerrada) debido a problemas de rendimiento.
* **SGD** es particularmente útil si tienes muchos datos o necesitas un modelo que se ajuste rápidamente en aplicaciones en tiempo real.

1. **Aplicación de la** **Descomposición de Valores Singulares (SVD)**:

* Si el conjunto de datos tiene características **altamente correlacionadas** (multicolinealidad) o si estás trabajando con una matriz muy grande y compleja, la SVD puede ayudar a estabilizar la solución.
* La SVD descompone los datos en componentes principales, permitiendo obtener una solución más estable y reducir la dimensionalidad si es necesario.

1. **Ajuste de Regularización si es Necesario**:

* Si después de entrenar el modelo aún ves problemas de sobreajuste (el modelo funciona bien en entrenamiento, pero mal en validación), puedes añadir **regularización** (como Ridge o Lasso) para penalizar los coeficientes grandes y reducir la complejidad del modelo.
* La regularización también ayuda a mejorar el rendimiento del modelo en datos nuevos al evitar que el modelo se ajuste demasiado al conjunto de datos de entrenamiento.

# Métodos de optimización

Los **métodos de optimización** son algoritmos matemáticos diseñados para encontrar los valores óptimos de los parámetros de un modelo, de modo que se minimice (o maximice) una **función objetivo** o **función de pérdida**. En el contexto de la **regresión lineal** y de muchos otros modelos de machine learning, los métodos de optimización se utilizan para encontrar los coeficientes o pesos que minimizan el error de predicción de forma iterativa.

### Descenso de Gradiente:

El descenso de gradiente es un método iterativo para minimizar una función de pérdida ajustando los parámetros del modelo (como los coeficientes en regresión lineal). Funciona calculando la dirección del gradiente (derivada parcial) y actualizando los parámetros en la dirección opuesta al gradiente para reducir el error.

La idea general del descenso de gradiente es ajustar parámetros de forma iterativa para minimizar la función de pérdida. En la práctica, se empieza por rellenar θ con valores aleatorios (esto se denomina inicialización aleatoria). Después, se mejora de manera gradual, pasito a pasito, intentando con cada paso reducir la función de pérdida (por ejemplo, el ECM), hasta que el algoritmo converge en un mínimo.

Gráfico

Descripción generada automáticamente

Un parámetro importante en el descenso de gradiente es el tamaño de los pasos (el hiperparámetro tasa de aprendizaje). Si la tasa de aprendizaje es demasiado pequeña, entonces el algoritmo tendrá que pasar por muchas iteraciones para converger, lo cual llevará mucho tiempo.

Un dibujo de una persona

Descripción generada automáticamente con confianza baja

Por otra parte, si la tasa de aprendizaje es demasiado alta, puede que saltes por encima del valle y acabes al otro lado, posiblemente más alto de lo que estabas antes. Esto podría hacer que el algoritmo diverja, con valores cada vez más grandes, y no consiga encontrar una buena solución.

Imagen que contiene Diagrama

Descripción generada automáticamente

Por suerte, está garantizado que aplicando el descenso de gradiente sobre la función de pérdida del ECM se queda arbitrariamente cerca del mínimo global (si esperamos lo suficiente y si la tasa de aprendizaje no es demasiado alta), puesto que la función de perdida tiene forma de bol.

Cuantos más parámetros tenga un modelo, más dimensiones tendrá este espacio y más difícil será la búsqueda.

#### **Variantes de Descenso de Gradiente**:

1. **Descenso de gradiente por lotes - Batch Gradient Descent**:
   * Utiliza todo el conjunto de datos para calcular el gradiente.
   * Es computacionalmente costoso para grandes conjuntos de datos.
   * Necesitamos calcular cuánto cambiará la función de pérdida si cambiamos θj solo un poquito (derivada parcial).
   * El vector de gradiente contiene todas las derivadas parciales de la función de pérdida (una para cada parámetro del modelo). Esto implica cálculos sobre el conjunto de entrenamiento completo X y como resultado, es lentísimo en conjuntos de entrenamiento muy grandes.
   * El descenso de gradiente escala bien con el número de características.
   * **Clase SGDRegressor**, que tiende por defecto a optimizar la función de pérdida del ECM.
   * Ejemplo:

La siguiente figura muestra los 20 primeros pasos del descenso de gradiente utilizando tres tasas de aprendizaje diferentes. La línea de la parte inferior de cada gráfico representa el punto de partida aleatorio y, después, cada repetición se representa con una línea cada vez más oscura.

Gráfico, Gráfico de dispersión

Descripción generada automáticamente

A la izquierda, la tasa de aprendizaje es demasiado baja: el algoritmo acabará llegando a la solución, pero tardará mucho. En el centro, la tasa de aprendizaje tiene un aspecto bastante bueno: en solo unas repeticiones, ya ha convergido en la solución. A la derecha, la tasa de aprendizaje es demasiado alta: el algoritmo diverge, salta por todas partes y, en realidad, se aleja más y más de la solución con cada paso.

* + Ejemplo en Python:

import numpy as np

# Datos de ejemplo (X: cantidad de unidades vendidas, y: precio del producto)

X = np.array([[1], [2], [3], [4], [5]], dtype=float)

y = np.array([2.5, 4.5, 6.5, 8.5, 10.5], dtype=float) # Relación lineal y = 2 \* X + 0.5

# Añadir el término de sesgo (columna de 1s para intercepto)

X\_b = np.c\_[np.ones((X.shape[0], 1)), X] # X\_b = [[1, x1], [1, x2], ...]

# Inicialización de parámetros

theta = np.random.randn(2, 1) # [intercepto, pendiente]

learning\_rate = 0.01

n\_iterations = 1000

m = len(X) # Número de muestras

# Gradiente descendente por lotes

for iteration in range(n\_iterations):

gradients = (2/m) \* X\_b.T.dot(X\_b.dot(theta) - y.reshape(-1, 1))

theta = theta - learning\_rate \* gradients # Actualización de parámetros

# Mostrar parámetros finales

print("Intercepto:", theta[0][0])

print("Pendiente:", theta[1][0])

# Predicciones

y\_pred = X\_b.dot(theta)

print("Predicciones:", y\_pred.flatten())

1. **Descenso del gradiente Estostatico - Stochastic Gradient Descent (SGD)**:
   * Actualiza los parámetros utilizando una sola muestra de datos a la vez.
   * Más rápido pero menos estable.
   * Elige una instancia aleatoria en el conjunto de entrenamiento a cada paso y calcula los gradientes basándose solo en esa única instancia
   * Es más rápido y agíl con conjuntos de entrenamiento enormes
   * Lo malo, es que debido a su naturaleza aleatoria, es mucho menos regular que el descenso por lotes.
   * **Clase SGDRegressor**, que tiende por defecto a optimizar la función de pérdida del ECM.
   * Ejemplo en Python

from sklearn.linear\_model import SGDRegressor

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

import numpy as np

# Datos de ejemplo

X = np.random.rand(100, 1) \* 10

y = 3 \* X.flatten() + np.random.randn(100) \* 2 # Relación lineal con ruido

# Dividir datos

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

# Modelo SGD

sgd = SGDRegressor(max\_iter=1000, learning\_rate='constant', eta0=0.01, random\_state=42)

sgd.fit(X\_train, y\_train)

y\_pred = sgd.predict(X\_test)

# Mostrar coeficientes

print("Coeficientes:", sgd.coef\_)

print("Intercepto:", sgd.intercept\_)

# Resultados

print("Coeficiente (Pendiente):", sgd.coef\_[0])

print("Intercepto:", sgd.intercept\_[0])

print("Error Cuadrático Medio (MSE):", mean\_squared\_error(y, y\_pred))

# Predicción para un nuevo dato

nuevo\_valor = np.array([[6]], dtype=float)

nuevo\_valor\_scaled = scaler.transform(nuevo\_valor)

prediccion = sgd.predict(nuevo\_valor\_scaled)

print(f"Predicción para {nuevo\_valor[0][0]} unidades vendidas:", prediccion[0])

#### **Aplicación en Regresión**:

El descenso de gradiente se utiliza para optimizar funciones de pérdida como el error cuadrático medio (MSE). Es común en modelos como la regresión lineal y regularizada (Ridge, Lasso).

1. **Descomposición de Valores singulares (SVD - Singular Value Decomposition)**

**Es una técnica de algebra lineal que descompone una matriz en tres matrices componentes y es la técnica usada por Scikit-Learn para calcular la regresión lineal con su algoritmo LinearRegression.**

**Sirve para:**

* + **Resolución de Problemas de Colinealidad**: La SVD proporciona soluciones estables en regresión lineal al manejar datos donde las variables independientes están altamente correlacionadas.
  + **Reducción Dimensional y Optimización**: Es clave para simplificar datos complejos manteniendo la mayor cantidad de información posible, utilizada en análisis y compresión de datos.
  + **Soluciones Directas y Eficientes**: Permite encontrar soluciones exactas de regresión lineal en una sola operación, sin necesidad de iteraciones, ahorrando tiempo y recursos computacionales.
  + **Ejemplo de Python básico:**

import numpy as np

from sklearn.linear\_model import LinearRegression

from sklearn.metrics import mean\_squared\_error, r2\_score

# Datos de ejemplo (X: cantidad de unidades vendidas, y: precio del producto)

X = np.array([[1], [2], [3], [4], [5]]) # Variable independiente

y = np.array([2.5, 4.5, 6.5, 8.5, 10.5]) # Variable dependiente

# Crear el modelo de regresión lineal

model = LinearRegression()

# Entrenar el modelo

model.fit(X, y)

# Predicción de valores

y\_pred = model.predict(X)

# Resultados

print("Coeficiente:", model.coef\_[0]) # Pendiente de la recta

print("Intercepto:", model.intercept\_) # Punto donde la recta cruza el eje Y

# Evaluación del modelo

mse = mean\_squared\_error(y, y\_pred)

r2 = r2\_score(y, y\_pred)

print("Error Cuadrático Medio (MSE):", mse)

print("Coeficiente de Determinación (R²):", r2)

# Reconstrucción con SVD (Internamente usado por LinearRegression)

U, S, Vt = np.linalg.svd(X, full\_matrices=False)

Sigma = np.diag(S)

X\_reconstructed = U @ Sigma @ Vt

print("\nReconstrucción de X (con SVD):")

print(X\_reconstructed)

### ¿En Qué Parte del Proceso se Usan los Métodos de Optimización?

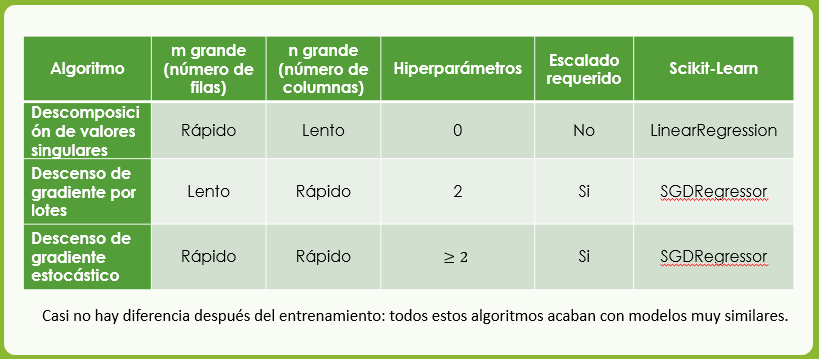
Los métodos de optimización se aplican durante la fase de **entrenamiento del modelo**. Este es el proceso general:

1. **Definición de la Función de Pérdida**:
   * Primero, se define una función que mida el error del modelo. En la regresión lineal, comúnmente se utiliza el **error cuadrático medio (MSE)**, que calcula la diferencia entre los valores predichos y los valores reales al cuadrado y luego promedia estos errores.
2. **Inicialización de Parámetros**:
   * Los parámetros del modelo (los coeficientes en una regresión lineal, por ejemplo) se inicializan. Pueden empezar con valores aleatorios o con ceros.
3. **Optimización**:
   * Aquí es donde entran los métodos de optimización. Su objetivo es **ajustar los parámetros del modelo** para minimizar la función de pérdida.
   * El optimizador ajusta iterativamente los parámetros hasta que el modelo se ajuste de la mejor manera posible a los datos de entrenamiento.
   * Esto se logra calculando el **gradiente** de la función de pérdida con respecto a los parámetros y actualizando los parámetros en la dirección opuesta al gradiente para reducir el error (en métodos como el descenso de gradiente).

### Resumen del Proceso

1. **Inicio del Entrenamiento**: Se define una función de pérdida que mide el error del modelo.
2. **Optimización**: Se aplica un método de optimización que ajusta los parámetros del modelo iterativamente para reducir el error de la función de pérdida.
3. **Convergencia**: El método de optimización finaliza cuando se ha alcanzado un mínimo de error (o un número de iteraciones determinado), y el modelo está ajustado para hacer predicciones.

### Comparación entre tres métodos de optimización para resolver problemas de regresión lineal:



# Regularización

La **regularización** en la **regresión** es una técnica utilizada para **evitar el sobreajuste** (overfitting) al penalizar los valores grandes de los coeficientes del modelo. Esto se logra agregando un término de penalización a la **función de pérdida** que el modelo intenta minimizar, lo cual limita la complejidad del modelo y le permite generalizar mejor en datos nuevos.

En términos simples, la regularización controla el tamaño de los coeficientes:

* Un modelo sin regularización puede ajustarse demasiado a los datos de entrenamiento, capturando ruido y patrones irrelevantes.
* Con la regularización, se evita que los coeficientes crezcan mucho, lo que simplifica el modelo y reduce la probabilidad de que se ajuste a variaciones aleatorias en los datos de entrenamiento.

### Tipos de Regularización en Regresión

* **Ridge (L2)**: Agrega una penalización de tipo L2, que es la suma de los cuadrados de los coeficientes. Esto reduce los valores de los coeficientes, pero sin eliminarlos completamente.
* **Lasso (L1)**: Agrega una penalización de tipo L1, que es la suma de los valores absolutos de los coeficientes. Esta penalización puede reducir algunos coeficientes a cero, eliminando características irrelevantes y ayudando en la selección de características.
* **Elastic Net**: Combina ambas penalizaciones L1 y L2, logrando un balance entre la reducción de los coeficientes y la eliminación de características.

### Objetivo de la Regularización

La regularización busca **mejorar la capacidad de generalización del modelo** al reducir el riesgo de sobreajuste, lo que permite que el modelo sea más robusto y preciso en datos que no ha visto antes.

## Regularización Ridge(L2)

### Resumen

La clave para utilizar la regularización **Ridge** correctamente es encontrar el valor óptimo de α que minimice tanto el error de entrenamiento como el de validación. Esto se puede lograr utilizando **validación cruzada** para probar diferentes valores de α y evitar tanto el **sobreajuste** como el **subajuste**.

### Características:

* El término de regularización solo debería añadirse a la función de pérdida durante el entrenamiento.
* Una vez que el modelo está entrenado, conviene utilizar el ECM (o la RECM) no regularizado para evaluar el rendimiento del modelo
* El hiperparámetro controla cuánto queremos regularizar el modelo
* Es importante escalar los datos(por ejemplo, utilizando StandardScaler) antes de realizar la regresión de Ridge ya que es sensible a la escala de las características de entrada

### Código ejemplo Python:

from sklearn.linear\_model import Ridge

from sklearn.metrics import mean\_squared\_error

# Datos de ejemplo

X = [[1], [2], [3], [4], [5]] # Variable independiente

y = [2.5, 4.5, 6.5, 8.5, 10.5] # Variable dependiente

# Modelo Ridge

ridge = Ridge(alpha=1.0) # El parámetro alpha controla la regularización

ridge.fit(X, y)

# Predicción

y\_pred = ridge.predict(X)

# Resultados

print("Coeficiente (Pendiente):", ridge.coef\_[0])

print("Intercepto:", ridge.intercept\_)

print("MSE:", mean\_squared\_error(y, y\_pred))

# Salida consola

Coeficiente (Pendiente): 1.8181818181818181

Intercepto: 1.0454545454545459

MSE: 0.06611570247933886

### Tabla comparativa de posibles escenarios

Aquí tienes una **tabla comparativa** para la regularización **Ridge (L2)**, donde se muestran los posibles escenarios en función del valor de α, las observaciones sobre el error, y la acción recomendada sobre α para evitar **sobreajuste** o **subajuste**.

| **Escenario** | **Valor de α** | **Observación del Error** | **Estado del Modelo** | **Acción Recomendada** | **Riesgo Potencial** |
| --- | --- | --- | --- | --- | --- |
| **Sin Regularización** | α=0 | Error de entrenamiento bajo, error de validación alto | Sobreajuste | Aumentar α para reducir la complejidad | Sobreajuste |
| **Regularización Moderada** | Valor bajo de α | Error de entrenamiento y validación equilibrados y bajos | Buen ajuste | Mantener o ajustar ligeramente según los datos | Ninguno |
| **Regularización Moderada-Alta** | Valor medio de α | Error de entrenamiento y validación similares pero algo altos | Ligeramente subajustado | Reducir α para mejorar el ajuste | Subajuste |
| **Regularización Alta** | Valor alto de α | Error de entrenamiento y validación similares y altos | Subajuste | Reducir α para permitir mayor complejidad | Subajuste |
| **Regularización Muy Alta** | α→∞ | Error de entrenamiento y validación muy altos, modelo casi plano | Fuerte Subajuste | Reducir α significativamente | Subajuste severo |

### Explicación de la Tabla

* **Sin Regularización (α=0)**:
  + Sin regularización, el modelo no tiene restricciones en los coeficientes, lo que puede llevar a un sobreajuste, especialmente si el modelo es complejo.
  + **Acción**: Aumentar α para controlar los coeficientes y reducir el sobreajuste.
* **Regularización Moderada (Valor Bajo de α)**:
  + Con una regularización ligera, el modelo puede estar bien ajustado si el error de entrenamiento y el de validación son bajos y equilibrados.
  + **Acción**: Mantener el valor de α o ajustarlo ligeramente si se observan cambios en el error.
* **Regularización Moderada-Alta (Valor Medio de α)**:
  + Con un valor medio de α, el modelo puede tener un ligero subajuste si el error es similar entre entrenamiento y validación, pero un poco alto.
  + **Acción**: Reducir α ligeramente para mejorar el ajuste.
* **Regularización Alta (Valor Alto de α)**:
  + Con un valor alto de α, el modelo tiende a subajustarse, ya que los coeficientes están fuertemente penalizados, lo que limita su capacidad de ajustarse a los datos.
  + **Acción**: Reducir α para permitir que el modelo capture más complejidad en los datos.
* **Regularización Muy Alta (α→∞)**
  + Con un α extremadamente alto, el modelo se convierte casi en una línea plana, resultando en un subajuste severo.
  + **Acción**: Reducir α significativamente, ya que el modelo es demasiado simple y no captura patrones relevantes en los datos.

## Regularización Lasso(L1)

### Resumen

Lasso (L1) agrega una penalización basada en la suma de los valores absolutos de los coeficientes. Esto puede llevar algunos coeficientes a ser exactamente cero, eliminando características irrelevantes y ayudando con la selección de características.

### Características:

* Añade un término de regularización a la función de pérdida (en este caso al termino de regularización se le conoce como L1)
* Tiende a eliminar los pesos de las características menos importantes (es decir, los pone a cero).

### Código ejemplo Python:

from sklearn.linear\_model import Lasso

from sklearn.metrics import mean\_squared\_error

# Modelo Lasso

lasso = Lasso(alpha=0.1) # El parámetro alpha controla la regularización

lasso.fit(X, y)

# Predicción

y\_pred = lasso.predict(X)

# Resultados

print("Coeficiente (Pendiente):", lasso.coef\_[0])

print("Intercepto:", lasso.intercept\_)

print("MSE:", mean\_squared\_error(y, y\_pred))

# Salida consola

Coeficiente (Pendiente): 1.95

Intercepto: 0.6500000000000004

MSE: 0.005000000000000036

### Tabla comparativa de posibles escenarios

| **Escenario** | **Valor de α** | **Observación del Error** | **Estado del Modelo** | **Acción Recomendada** | **Riesgo Potencial** |
| --- | --- | --- | --- | --- | --- |
| **Sin Regularización** | α=0 | Error de entrenamiento bajo, error de validación alto | Sobreajuste | Aumentar α para reducir la complejidad y eliminar características irrelevantes | Sobreajuste |
| **Regularización Moderada** | Valor bajo de α | Error de entrenamiento y validación equilibrados y bajos | Buen ajuste | Mantener o ajustar ligeramente según los datos | Ninguno |
| **Regularización Moderada-Alta** | Valor medio de α | Error de entrenamiento y validación similares, pero algo altos | Ligeramente subajustado | Reducir α para mejorar el ajuste | Subajuste |
| **Regularización Alta** | Valor alto de α | Error de entrenamiento y validación similares y altos, varios coeficientes en cero | Subajuste | Reducir α para permitir mayor complejidad y evitar eliminación excesiva de características | Subajuste |
| **Regularización Muy Alta** | α→∞ | Error de entrenamiento y validación muy altos, la mayoría de los coeficientes en cero | Fuerte Subajuste | Reducir α significativamente | Subajuste severo |

### Explicación de la Tabla Lasso

* **Sin Regularización (**α=0**)**:
  + Sin regularización (α=0), el modelo no elimina ninguna característica, lo que puede llevar al sobreajuste si hay variables irrelevantes.
  + Acción: Aumentar α para reducir la complejidad y eliminar características irrelevantes
* **Regularización Moderada** **(**Valor bajo de α**)**:
  + Un valor bajo de α permite eliminar algunas características irrelevantes mientras mantiene un buen ajuste.
  + Acción: Mantener o ajustar ligeramente según los datos
* **Regularización Moderada-Alta** **(**Valor medio de α**)**:
  + Si el error de entrenamiento y validación son similares, pero algo altos, reducir α permite un ajuste más preciso.
  + Acción: Reducir α para permitir mayor complejidad y evitar eliminación excesiva de características
* **Regularización Alta** **(**Valor alto de α**)**:
  + Un valor alto de α puede eliminar demasiadas características, causando subajuste.
  + Acción: Reducir α para permitir mayor complejidad y evitar eliminación excesiva de características
* **Regularización Muy Alta** **(**α→∞**)**:
  + Un α muy alto elimina la mayoría de los coeficientes, causando subajuste severo.
  + Acción: Reducir α significativamente

## Regularización Elastic Net

### Resumen

Elastic Net combina las penalizaciones de Lasso (L1) y Ridge (L2), controladas por dos hiperparámetros: α, que determina la magnitud de la regularización, y r, que define la proporción entre L1 y L2 (por ejemplo, si r=0.5, ambos tipos de regularización se aplican en la misma medida).

### Gestión del hiperparámetros R

En **Elastic Net**, el hiperparámetro **r** (a veces llamado **ratio** o **mezcla**) controla la combinación de regularización **L1 (Lasso)** y **L2 (Ridge)**. Su valor puede variar entre **0 y 1**, donde:

* **r=0**: El modelo se comporta como una **regularización Ridge pura** (L2). Esto significa que la penalización solo considera la suma de los cuadrados de los coeficientes, sin eliminar características.
* **r=1**: El modelo se comporta como una **regularización Lasso pura** (L1). Esto significa que la penalización se basa en la suma de los valores absolutos de los coeficientes, lo que puede llevar algunos coeficientes a cero, eliminando características irrelevantes.
* **Valores intermedios (0 < r < 1)**: El modelo aplica una **combinación de L1 y L2**, con una proporción específica entre ambas regularizaciones. Por ejemplo:
  + Si r=0.5, se aplica una mezcla equilibrada de regularización L1 y L2.
  + Si r=0.25, el modelo se inclina más hacia Ridge (L2) que hacia Lasso (L1).
  + Si r=0.75, el modelo se inclina más hacia Lasso (L1) que hacia Ridge (L2).

#### ¿Cuándo usar cada rango de r?

* **r cercano a 0** (más parecido a Ridge): Útil cuando tienes muchas variables correlacionadas y quieres mantener todas las características en el modelo sin eliminarlas.
* **r cercano a 1** (más parecido a Lasso): Útil cuando quieres realizar una **selección de características**, es decir, eliminar las variables menos importantes estableciendo algunos coeficientes en cero.
* **Valores intermedios**: Útiles cuando tienes datos complejos y necesitas una combinación de ambos métodos: la capacidad de eliminar características irrelevantes (L1) y la estabilidad frente a características correlacionadas (L2).

### Código ejemplo Python

from sklearn.linear\_model import ElasticNet

from sklearn.metrics import mean\_squared\_error

# Modelo ElasticNet

elasticnet = ElasticNet(alpha=0.1, l1\_ratio=0.5) # l1\_ratio controla el balance entre L1 y L2

elasticnet.fit(X, y)

# Predicción

y\_pred = elasticnet.predict(X)

# Resultados

print("Coeficiente (Pendiente):", lasso.coef\_[0])

print("Intercepto:", lasso.intercept\_)

print("MSE:", mean\_squared\_error(y, y\_pred))

# Salida consola

Coeficiente (Pendiente): 1.95

Intercepto: 0.6500000000000004

MSE: 0.010707911957168501

### Tabla comparativa de posibles escenarios

| **Escenario** | **Valor de α y r** | **Observación del Error** | **Estado del Modelo** | **Acción Recomendada** | **Riesgo Potencial** |
| --- | --- | --- | --- | --- | --- |
| **Sin Regularización** | α=0 | Error de entrenamiento bajo, error de validación alto | Sobreajuste | Aumentar α para reducir complejidad y eliminar características irrelevantes | Sobreajuste |
| **Regularización Moderada** | Valor bajo de α, r moderado | Error de entrenamiento y validación equilibrados y bajos | Buen ajuste | Mantener o ajustar ligeramente según los datos | Ninguno |
| **Regularización Moderada-Alta** | Valor medio de α, r moderado | Error de entrenamiento y validación similares pero algo altos | Ligeramente subajustado | Reducir α para mejorar el ajuste | Subajuste |
| **Regularización Alta** | Valor alto de α, r alto (predominante L1) | Error de entrenamiento y validación altos, varios coeficientes en cero | Subajuste | Reducir α y/o ajustar r para reducir la eliminación excesiva de características | Subajuste |
| **Regularización Muy Alta** | α→∞, r bajo (predominante L2) | Error de entrenamiento y validación muy altos, coeficientes muy pequeños, pocos en cero | Fuerte Subajuste | Reducir α significativamente y ajustar r para mejorar el ajuste | Subajuste severo |

### Explicación de la Tabla Elastic Net

* **Sin Regularización (**α=0**)**:
  + Con α=0, no hay ninguna penalización, lo que puede llevar al sobreajuste si el modelo es demasiado complejo.
  + Acción: Aumentar α para reducir complejidad y eliminar características irrelevantes
* **Regularización Moderada (**Valor bajo de α, r moderado**)**:
  + Un valor bajo de α y un valor moderado de r mantienen un buen ajuste y seleccionan algunas características irrelevantes.
  + Acción: Mantener o ajustar ligeramente según los datos
* **Regularización Moderada-Alta (**Valor medio de α, r moderado**)**:
  + Si el modelo muestra subajuste ligero, reducir α permite una mejor generalización.
  + Acción: Reducir α para mejorar el ajuste
* **Regularización Alta (**Valor alto de α, r alto (predominante L1)**)**:
  + Un α alto y un r alto (predominancia de L1) eliminan varias características, lo que puede causar subajuste.
  + Acción: Reducir α y/o ajustar r para reducir la eliminación excesiva de características
* **Regularización Muy Alta (**α→∞, r bajo (predominante L2)**)**:
  + Con α muy alto y r bajo (predominancia de L2), los coeficientes son muy pequeños pero no eliminados, causando un subajuste severo.
  + Acción: Reducir α significativamente y ajustar r para mejorar el ajuste

### Resumen de Acciones en Ambas Regularizaciones

* **Sobreajuste**: Si ves que el modelo sobreajusta, **aumenta α** (en Ridge, Lasso, o Elastic Net) para penalizar los coeficientes y reducir la complejidad.
* **Subajuste: Si el modelo subajusta, reduce α para que el modelo capture mejor las relaciones. En Elastic Net, puedes también ajustar el valor de r para cambiar la proporción entre L1 y L2 según el problema.**

## Detención temprana

Una forma muy diferente de regularizar algoritmos de aprendizaje iterativo, como el descenso de gradiente, es detener el entrenamiento en cuanto el error de validación alcanza un mínimo. Esto se denomina detención temprana.

La siguiente figura muestra un modelo complejo (en este caso, un modelo de regresión polinomial de alto grado) que se está entrenando con descenso de gradiente por lotes en el conjunto de datos cuadrático que hemos utilizado antes.

Gráfico, Gráfico de líneas

Descripción generada automáticamente

A medida que avanzan las repeticiones, el algoritmo aprende y su error de predicción (RECM) en el conjunto de entrenamiento baja, junto con su error de predicción en el conjunto de validación. Sin embargo, después de un tiempo, el error de validación deja de reducirse y empieza a subir otra vez. Eso indica que el modelo ha empezado a sobreajustar los datos de entrenamiento.

**Puede parar el entrenamiento en cuanto el error de validación alcanza el mínimo o parar solo después de que el error de validación lleve un tiempo por encima del mínimo.**

from copy import deepcopy

from sklearn.metrics import mean\_squared\_error

from sklearn.preprocessing import StandardScaler

np.random.seed(42)

m = 100

X = 6 \* np.random.rand(m, 1) - 3

y = 0.5 \* X \*\* 2 + X + 2 + np.random.randn(m, 1)

X\_train, y\_train = X[: m // 2], y[: m // 2, 0]

X\_valid, y\_valid = X[m // 2 :], y[m // 2 :, 0]

preprocessing = make\_pipeline(PolynomialFeatures(degree=90, include\_bias=False), StandardScaler())

X\_train\_prep = preprocessing.fit\_transform(X\_train)

X\_valid\_prep = preprocessing.transform(X\_valid)

sgd\_reg = SGDRegressor(penalty=None, eta0=0.002, random\_state=42)

n\_epochs = 500

best\_valid\_rmse = float('inf')

train\_errors, val\_errors = [], []

for epoch in range(n\_epochs):

sgd\_reg.partial\_fit(X\_train\_prep, y\_train)

y\_valid\_predict = sgd\_reg.predict(X\_valid\_prep)

val\_error = mean\_squared\_error(y\_valid, y\_valid\_predict,

squared=False)

if val\_error < best\_valid\_rmse:

best\_valid\_rmse = val\_error

best\_model = deepcopy(sgd\_reg)

y\_train\_predict = sgd\_reg.predict(X\_train\_prep)

train\_error = mean\_squared\_error(y\_train, y\_train\_predict,

squared=False)

val\_errors.append(val\_error)

train\_errors.append(train\_error)

# para dibujar la imagen

best\_epoch = np.argmin(val\_errors)

plt.figure(figsize=(6, 4))

plt.annotate('Best model',

xy=(best\_epoch, best\_valid\_rmse),

xytext=(best\_epoch, best\_valid\_rmse + 0.5),

ha="center",

arrowprops=dict(facecolor='black', shrink=0.05))

plt.plot([0, n\_epochs], [best\_valid\_rmse, best\_valid\_rmse], "k:", linewidth=2)

plt.plot(val\_errors, "b-", linewidth=3, label="Validation set")

plt.plot(best\_epoch, best\_valid\_rmse, "bo")

plt.plot(train\_errors, "r--", linewidth=2, label="Training set")

plt.legend(loc="upper right")

plt.xlabel("Epoch")

plt.ylabel("RMSE")

plt.axis([0, n\_epochs, 0, 3.5])

plt.grid()

# Algoritmos de trabajo Regresión

| **Algoritmo** | **Descripción** | **Clase en Scikit-Learn** |
| --- | --- | --- |
| **Regresión Lineal** | Modelo básico de regresión que asume una relación lineal entre la variable dependiente y una o más variables independientes. | LinearRegression |
| **Regresión Lineal Múltiple** | Extiende la regresión lineal simple para incluir múltiples variables independientes. | LinearRegression |
| **Regresión Polinómica** | Expande las características con potencias de cada variable independiente para modelar relaciones no lineales. | PolynomialFeatures + LinearRegression |
| **Descomposición en Valores Singulares (SVD)** | Técnica de álgebra lineal para resolver problemas de regresión lineal con matrices mal condicionadas o colineales. | LinearRegression (con solver SVD) |
| **Descenso de Gradiente Estocástico** | Ajusta los coeficientes iterativamente usando un solo dato de entrenamiento a la vez, lo cual es eficiente para grandes conjuntos de datos. | SGDRegressor |
| **Ridge Regression (L2)** | Regresión lineal con regularización L2, que penaliza la suma de los cuadrados de los coeficientes para evitar sobreajuste. | Ridge |
| **Lasso Regression (L1)** | Regresión lineal con regularización L1, que penaliza la suma absoluta de los coeficientes, haciendo que algunos coeficientes se reduzcan a cero (selección de características). | Lasso |
| **Elastic Net** | Combina L1 y L2 para regularización, balanceando entre selección de características y estabilidad en la predicción. | ElasticNet |
| **Regresión con Support Vector Machines (SVR)** | Extensión de las máquinas de soporte vectorial (SVM) para problemas de regresión, permite ajustar márgenes para mejorar la precisión. | SVR |
| **Árboles de Decisión para Regresión** | Utiliza una estructura de árbol para realizar predicciones basadas en la división de los datos en nodos y ramas. | DecisionTreeRegressor |
| **Random Forest** | Conjunto de árboles de decisión para mejorar la precisión y reducir el sobreajuste mediante la combinación de múltiples predicciones de árboles individuales. | RandomForestRegressor |
| **Gradient Boosting Machines (GBM)** | Método de boosting que combina múltiples modelos de árboles débiles para mejorar la precisión de predicción en problemas de regresión. | GradientBoostingRegressor |
| **K-Nearest Neighbors (KNN) Regression** | Algoritmo basado en la distancia, predice el valor en función de los valores de sus vecinos más cercanos. | KNeighborsRegressor |

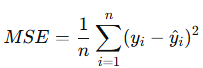
# Evaluación

La evaluación es un paso crucial en el proceso de regresión. Nos permite medir la calidad del modelo, identificar posibles problemas (como sobreajuste o subajuste) y comparar diferentes enfoques. Las métricas utilizadas dependen del tipo de problema, pero generalmente se enfocan en medir el error entre los valores predichos y los valores reales. **En el caso de la regresión** vamos a predecir **valores no numéricos continuos, en vez de categorías.**

## Métricas Comunes para la Evaluación de Modelos de Regresión

### Error Cuadrático Medio (MSE):

* + **Detalle**:
  + Mide la media de los cuadrados de los errores (diferencias entre los valores reales y predichos).
  + Los errores graves tienen una **penalización mayor** que con MAE
  + **Fórmula**:



* + **Interpretación**:
    - Cuanto más pequeño, mejor. Sin embargo, puede ser sensible a valores atípicos debido al cuadrado de las diferencias.

### Raíz del Error Cuadrático Medio (RMSE):

* + **Detalle**:
  + La raíz cuadrada del MSE. Se encuentra en las mismas unidades que la variable dependiente.
  + Los errores graves tienen una penalización mayor que con MAE
  + **Tiene un orden de magnitud cercano a** **los valores de y (el más popular)**
  + **Fórmula**:



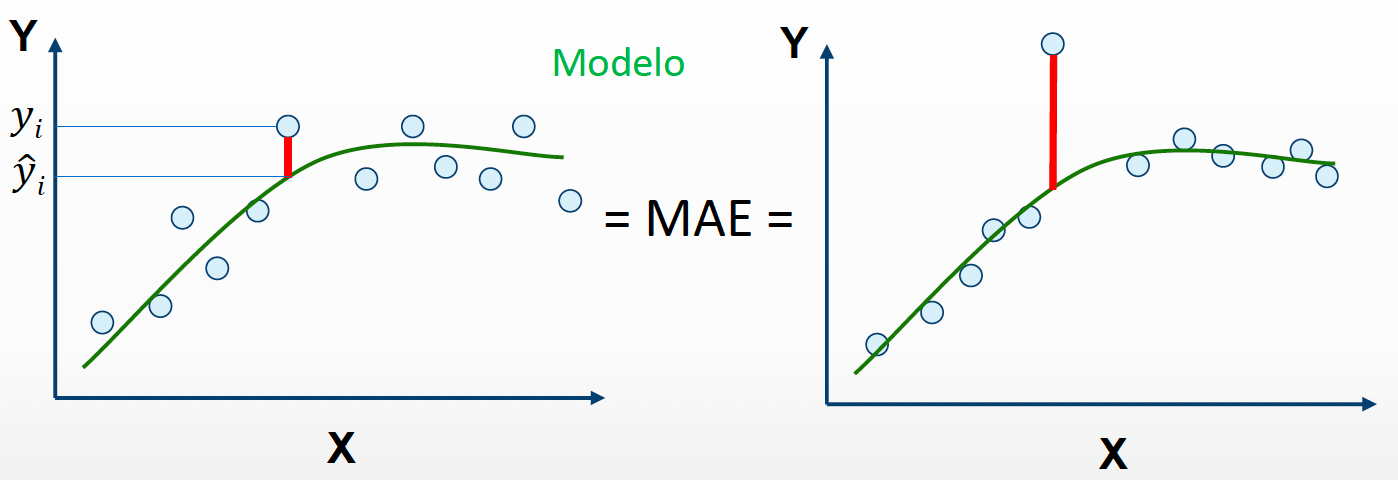
* + **Interpretación**:
    - Proporciona una medida intuitiva del error promedio en las mismas unidades que la salida.

### Error Absoluto Medio (MAE):

* **Detalle**
  + **No penaliza los errores graves**
  + Mide la media de las diferencias absolutas entre los valores reales y predichos.
  + Sencillo de entender
  + Tiene un orden de magnitud cercano a los valores de y.
* **Fórmula**:



* **Interpretación**:
  + Es menos sensible a valores atípicos en comparación con el MSE.



## Métodos de Validación

### División de Datos en Conjuntos de Entrenamiento y Prueba:

* Se entrena el modelo en una parte de los datos (entrenamiento) y se evalúa en una parte separada (prueba).
* Proporciones comunes: 80% para entrenamiento y 20% para prueba.

**Validación Cruzada**:

* Divide los datos en múltiples subconjuntos (folds) y evalúa el modelo en cada uno de ellos.
* Tipos:
  + **K-Fold Cross-Validation**: Divide los datos en K subconjuntos y entrena el modelo en K-1 folds mientras valida en el restante.
  + **Leave-One-Out Cross-Validation (LOOCV)**: Usa un solo dato para validar y el resto para entrenar.

**Validación por Bootstrapping**:

* Genera subconjuntos aleatorios con reemplazo para evaluar la variabilidad del modelo.

## Curvas de Aprendizaje

* Las curvas de aprendizaje muestran cómo varía el error en función del tamaño del conjunto de entrenamiento.
* Permiten identificar:
  + **Sobreajuste**: Error de entrenamiento bajo y error de validación alto.
  + **Subajuste**: Ambos errores altos.

**Construcción:**

* Dividir el conjunto de entrenamiento en subconjuntos crecientes.
* Entrenar y evaluar el modelo en cada subconjunto.

**Visualización:**

* Ejes:
  + X: Tamaño del conjunto de entrenamiento.
  + Y: Error (MSE, MAE, etc.).
* Curvas separadas para entrenamiento y validación.

## Comparación entre Modelos

* Para seleccionar el mejor modelo de regresión, se evalúan múltiples métricas y se comparan diferentes enfoques:
  + Regresión Lineal, Ridge, Lasso, Elastic Net, Árboles de Decisión, etc.
* Se prioriza el modelo que tenga:
  + **Buen rendimiento en métricas clave (MSE, R², MAE).**

1. **Seleccionar Métricas Relevantes**:
   * + Dependiendo del problema, selecciona las métricas clave que mejor representen el rendimiento del modelo.
       - **Regresión**:
         1. **MSE**: Útil si los errores grandes son penalizados.
         2. **MAE**: Adecuado si los errores pequeños y grandes deben ser tratados por igual.
2. **Calcular las Métricas**:
   * + Divide los datos en conjuntos de entrenamiento y prueba.
     + Entrena el modelo en el conjunto de entrenamiento.
     + Predice sobre el conjunto de prueba.
     + Calcula cada métrica usando funciones como mean\_squared\_error y mean\_absolute\_error.
3. **Comparar Modelos**:
   * + Entrena múltiples modelos y compara los resultados en las métricas seleccionadas.
     + Prioriza los modelos con:
       - **Menor MSE/MAE**: Indican menor error.
   * **Menor riesgo de sobreajuste o subajuste.**
4. **Identificar Indicadores de Sobreajuste/Subajuste**:

* **Sobreajuste**:
  + - * Bajo error en entrenamiento, alto error en prueba.
      * Indicador de que el modelo es demasiado complejo.
* Subajuste**:**
  + - Alto error en ambos conjuntos (entrenamiento y prueba).
    - Indicador de que el modelo no captura patrones en los datos.

1. **Evaluar Curvas de Aprendizaje**:

* Entrena el modelo con diferentes tamaños de conjuntos de entrenamiento.
* Calcula y grafica los errores en entrenamiento y validación.
* Identifica si hay una gran discrepancia (sobreajuste) o errores consistentemente altos (subajuste).

1. **Ajustar la Complejidad del Modelo**:

* **Para reducir el sobreajuste:**
  + - * Regulariza el modelo (e.g., alpha en Ridge/Lasso).
      * Reduce la profundidad de los árboles (max\_depth) o ajusta min\_samples\_split.
* **Para reducir el subajuste:**
  + - Aumenta la complejidad del modelo (e.g., degree en polinomios, mayor max\_depth).
  + **Mejor capacidad de generalización (según validación cruzada).**

1. **Aplicar Validación Cruzada**:

* Divide los datos en múltiples subconjuntos (folds).
* Entrena y evalúa el modelo en diferentes particiones, asegurando que cada dato se use tanto para entrenamiento como para prueba.

1. **Obtener Métricas Promedio:**

* Calcula la métrica seleccionada (e.g., MSE) en cada fold.
* Promedia los resultados para obtener una métrica representativa del modelo.

1. **Comparar Resultados Entre Modelos:**

* Prioriza los modelos con menor varianza y mejor desempeño promedio en validación cruzada.
* Los modelos con alta varianza pueden ser inestables y propensos a sobreajuste.

1. **Usar Técnicas de Validación Avanzada:**

* **K-Fold Cross Validation**: Usa particiones de datos estándar.
* **Stratified K-Fold**: Asegura que cada fold tenga una proporción similar de clases (para clasificación).
* **Leave-One-Out Cross Validation (LOOCV)**: Evalúa el modelo dejando una sola instancia fuera en cada iteración.

## Resumen del Proceso

| **Criterio** | **Proceso Clave** | **Resultado Esperado** |
| --- | --- | --- |
| **Métricas Clave** | Calcular MSE, MAE, R². | Selección de modelos con menor error y mejor ajuste. |
| **Riesgo de Sobreajuste/Subajuste** | Analizar curvas de aprendizaje, ajustar complejidad del modelo. | Modelo balanceado que generalice bien a datos nuevos. |
| **Capacidad de Generalización** | Aplicar validación cruzada, obtener métricas promedio y varianza. | Selección del modelo con menor error promedio y baja varianza. |

# Ejemplo Código Python

import numpy as np

import matplotlib.pyplot as plt

from sklearn.datasets import make\_regression

from sklearn.model\_selection import train\_test\_split, cross\_val\_score, learning\_curve, GridSearchCV

from sklearn.linear\_model import Ridge, Lasso

from sklearn.tree import DecisionTreeRegressor

from sklearn.preprocessing import StandardScaler

from sklearn.pipeline import Pipeline

from sklearn.metrics import mean\_squared\_error, mean\_absolute\_error

# 1. Generar un conjunto de datos sintético

X, y = make\_regression(n\_samples=500, n\_features=5, noise=15, random\_state=42)

# Dividir el conjunto de datos en entrenamiento y prueba

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

# 2. Definir modelos y espacios de búsqueda de hiperparámetros

param\_grids = {

"Ridge Regression": {

'model\_\_alpha': [0.1, 1.0, 10.0]

},

"Lasso Regression": {

'model\_\_alpha': [0.01, 0.1, 1.0]

},

"Decision Tree": {

'model\_\_max\_depth': [3, 5, 10],

'model\_\_min\_samples\_split': [2, 5, 10]

}

}

pipelines = {

"Ridge Regression": Pipeline([

('scaler', StandardScaler()),

('model', Ridge())

]),

"Lasso Regression": Pipeline([

('scaler', StandardScaler()),

('model', Lasso())

]),

"Decision Tree": Pipeline([

('model', DecisionTreeRegressor(random\_state=42))

])

}

# 3. Realizar la optimización de hiperparámetros con GridSearchCV

optimized\_models = {}

for model\_name, pipeline in pipelines.items():

print(f"Optimizando {model\_name}...")

grid\_search = GridSearchCV(pipeline, param\_grid=param\_grids[model\_name], cv=5, scoring='neg\_mean\_squared\_error')

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

optimized\_models[model\_name] = grid\_search.best\_estimator\_

print(f"Mejores parámetros para {model\_name}: {grid\_search.best\_params\_}")

print()

# 4. Evaluar los modelos optimizados

results = {}

for model\_name, model in optimized\_models.items():

print(f"Evaluando {model\_name} (Optimizado)...")

# Predicciones

y\_pred = model.predict(X\_test)

# Calcular métricas

mse = mean\_squared\_error(y\_test, y\_pred)

mae = mean\_absolute\_error(y\_test, y\_pred)

# Validación cruzada

cv\_scores = cross\_val\_score(model, X, y, cv=5, scoring='neg\_mean\_squared\_error')

mean\_cv\_score = -cv\_scores.mean()

# Guardar resultados

results[model\_name] = {

"MSE": mse,

"MAE": mae,

"MSE (CV)": mean\_cv\_score,

"Varianza (CV)": cv\_scores.var()

}

print(f"MSE: {mse:.2f}, MAE: {mae:.2f}, MSE (CV): {mean\_cv\_score:.2f}")

print()

# 5. Curvas de aprendizaje (ejemplo con Ridge Regression optimizado)

train\_sizes, train\_scores, test\_scores = learning\_curve(

optimized\_models["Ridge Regression"], X, y, cv=5, scoring='neg\_mean\_squared\_error', train\_sizes=np.linspace(0.1, 1.0, 10)

)

train\_errors = -np.mean(train\_scores, axis=1)

test\_errors = -np.mean(test\_scores, axis=1)

plt.plot(train\_sizes, train\_errors, label="Training Error")

plt.plot(train\_sizes, test\_errors, label="Validation Error")

plt.xlabel("Tamaño del conjunto de entrenamiento")

plt.ylabel("Error (MSE)")

plt.legend()

plt.title("Curvas de Aprendizaje - Ridge Regression (Optimizado)")

plt.show()

# 6. Resumen de resultados

print("\nResumen de resultados:")

for model\_name, metrics in results.items():

print(f"{model\_name}:")

for metric\_name, value in metrics.items():

print(f" {metric\_name}: {value:.2f}")

print()