IT Academy
by KIBERNUM

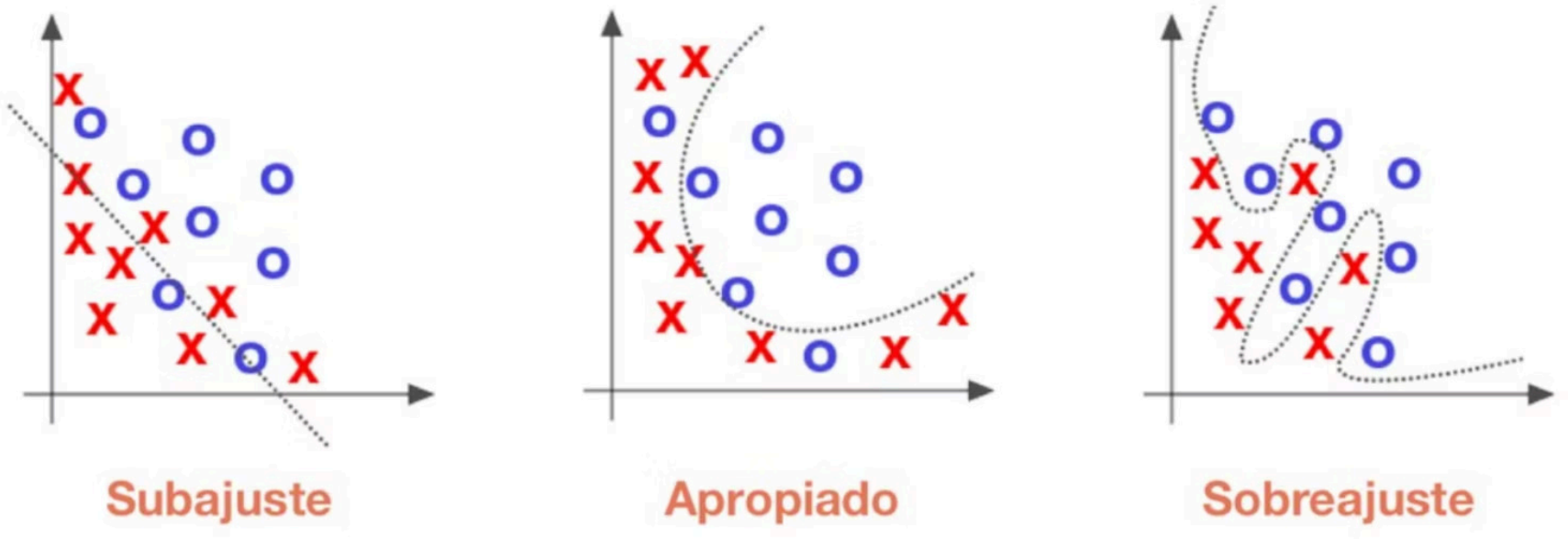
Ajustes de modelos y validación cruzada

Uno de los principales desafíos al construir modelos de Machine Learning es lograr que funcionen bien no solo con los datos de entrenamiento, sino también con datos nuevos y desconocidos. Esta capacidad de "generalización" es lo que distingue a un modelo útil de uno engañosamente perfecto.

R **por Kibernum Capacitación S.A.**

Tipos de error en el aprendizaje automático

Ajustar un modelo significa entrenarlo para que reconozca patrones en los datos. Pero ¿qué tan bien debe aprender? Si aprende muy poco, no captará los patrones importantes. Si aprende demasiado, puede terminar "memorizando" los datos. Esto nos lleva a tres posibles escenarios:



La imagen ilustra tres escenarios comunes al entrenar modelos de Machine Learning:

- En el subajuste, el modelo es demasiado simple y no logra capturar la relación entre los datos.
- En el ajuste adecuado, el modelo generaliza correctamente, identificando patrones sin exagerar.
- En el sobreajuste, el modelo memoriza demasiado los datos de entrenamiento, lo que perjudica su rendimiento con nuevos datos.

Esta visualización es clave para comprender el equilibrio necesario entre precisión y generalización.

Sobreajuste (Overfitting)

Un modelo sobreajustado es como un estudiante que se aprendió de memoria todas las respuestas del ensayo, pero no entendió realmente los temas. En el examen real, donde las preguntas cambian un poco, no sabe qué hacer, porque no aprendió a razonar, solo a repetir.

¿Por qué sucede?

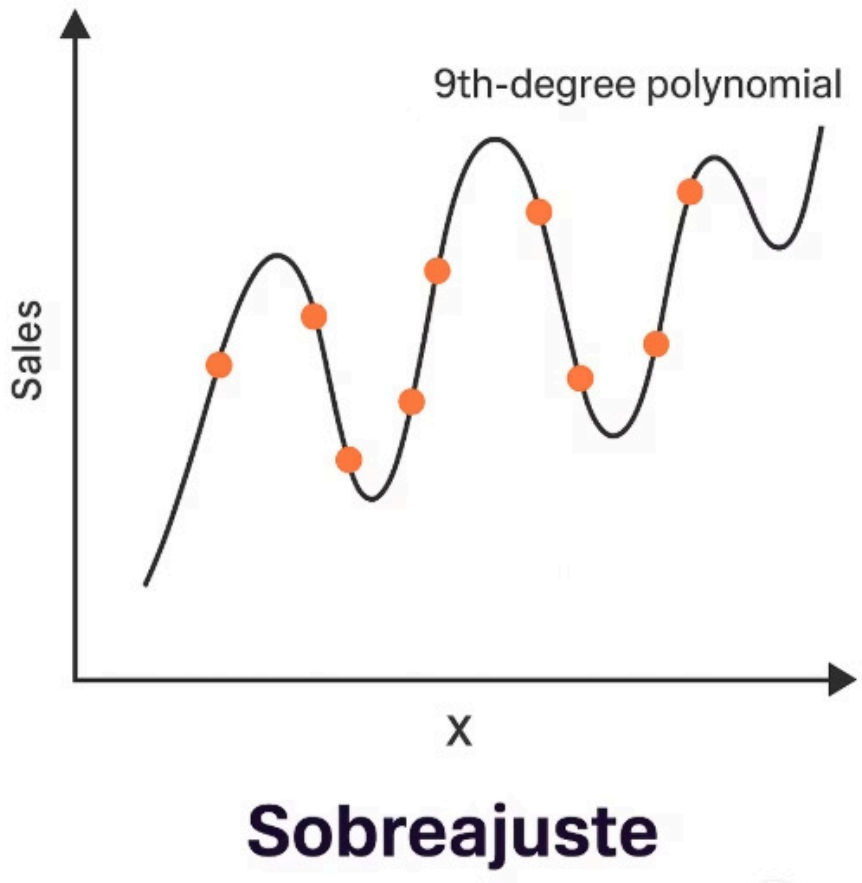
Esto ocurre cuando el modelo es demasiado complejo para la cantidad o tipo de datos que tiene. En vez de aprender los patrones importantes, también aprende el "ruido" o las coincidencias raras del entrenamiento. Termina siendo un modelo que "memoriza" en vez de "comprender".

Consecuencias de un modelo sobreajustado:

- Excelente rendimiento en entrenamiento (¡parece perfecto!).
- Mal rendimiento en datos nuevos (fracasa en la vida real).
- No generaliza, solo repite lo que ya vio.

Imagina que tienes 10 datos sobre ventas y quieres predecir una tendencia. Un modelo equilibrado trazaría una línea o curva que resume bien la tendencia general. Pero si haces que el modelo pase por cada uno de esos 10 puntos exactos, crearás una curva exagerada que no sigue ningún patrón lógico, solo conecta los puntos por obligación.

Resultado: el modelo parece muy preciso, pero si llega un nuevo dato, no sabe qué hacer. Las predicciones se vuelven erráticas, impredecibles y poco confiables.



Subajuste (Underfitting)

Un modelo subajustado es como un estudiante que leyó solo los títulos de los temas, sin profundizar. Cuando le hacen preguntas, responde de forma vaga o demasiado general, sin capturar los detalles que realmente importan.

¿Por qué ocurre?

Sucede cuando el modelo es demasiado simple para la complejidad del problema. No tiene la capacidad suficiente para reconocer ni siquiera los patrones básicos en los datos.

Consecuencias de un modelo subajustado:

- Mal desempeño en entrenamiento (ni siquiera aprende lo que ya vio).
- Mal desempeño en prueba (no generaliza).
- Ignora relaciones clave entre las variables.

Imagina que tienes datos que claramente forman una curva (como una sonrisa), pero decides ajustar una línea recta. El modelo "se esfuerza", pero no puede capturar la forma real. El resultado será un modelo que falla tanto con los datos conocidos como con los nuevos.

Como resultado obtendrás predicciones pobres, sin importar qué datos le des.

Ajuste adecuado: el punto ideal

Este es el escenario ideal. El modelo ha aprendido lo suficiente para capturar los patrones verdaderos del problema, pero no tanto como para memorizar los detalles irrelevantes.

¿Cómo se logra?

- Eligiendo un modelo apropiado para la tarea (ni muy simple ni muy complejo).
- Usando técnicas como validación cruzada para medir su rendimiento de forma equilibrada.
- Reservando un conjunto de prueba independiente.
- Ajustando hiperparámetros para afinar su comportamiento.
- Tener un modelo que rinda bien tanto en entrenamiento como en datos nuevos, logrando así una buena capacidad de generalización.

Un buen modelo no memoriza, comprende. Ni el que "no sabe nada" (subajuste), ni el que "se lo sabe todo de memoria" (sobreajuste) son útiles. El verdadero aprendizaje está en el equilibrio.

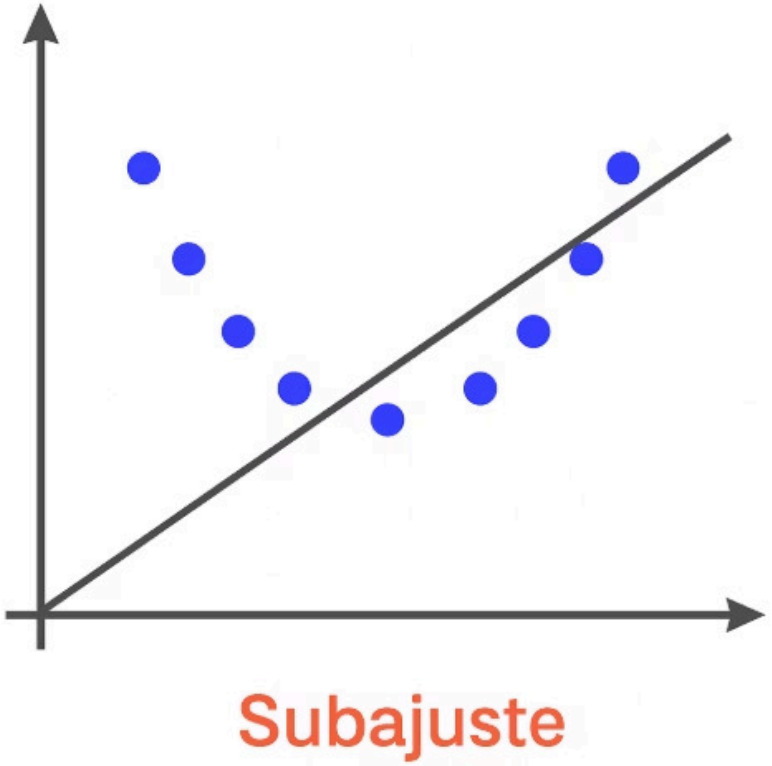


Imagen 3: *Aunque los datos tienen una forma claramente curvada, el modelo intenta ajustarlos con una línea recta. Este ejemplo representa un modelo demasiado simple, incapaz de capturar la estructura real de los datos. El resultado es un modelo que falla tanto en entrenamiento como en predicción.*

Ejercicio guiado: Explorando el ajuste del modelo en función de su complejidad

Objetivo del ejercicio

Comprender cómo la complejidad de un modelo afecta su capacidad para aprender de los datos y generalizar correctamente. A través de visualizaciones comparativas, podrán identificar los efectos del subajuste, ajuste adecuado y sobreajuste, utilizando modelos de regresión polinomial con distintos grados.

Contexto del ejercicio

Vamos a trabajar con un conjunto de datos sintético generado con make_regression, el cual incluye ruido (variabilidad aleatoria) para simular un entorno más realista. Luego, construiremos modelos de regresión polinomial de distintos grados para visualizar cómo cambia su comportamiento al aumentar la complejidad.

Los puntos generados simulan una relación lineal con algo de ruido. Esta visualización inicial será la base para observar cómo distintos modelos intentan ajustarse a estos datos.

Paso 1: Generar y visualizar los datos

```python
# Paso 1: Importamos las librerías necesarias
from sklearn.datasets import make_regression
import matplotlib.pyplot as plt
import numpy as np

# Generamos un conjunto de datos sintético
#  🔹 100 muestras
#  🔹 1 característica (X)
#  🔹 Ruido aleatorio para simular datos reales
X, y = make_regression(n_samples=100, n_features=1, noise=20, random_state=1)

# Visualizamos los datos en un diagrama de dispersión
plt.scatter(X, y, color='blue')
plt.title("Distribución de datos simulados")
plt.xlabel("X")
plt.ylabel("y")
plt.grid(True)
plt.show()
```

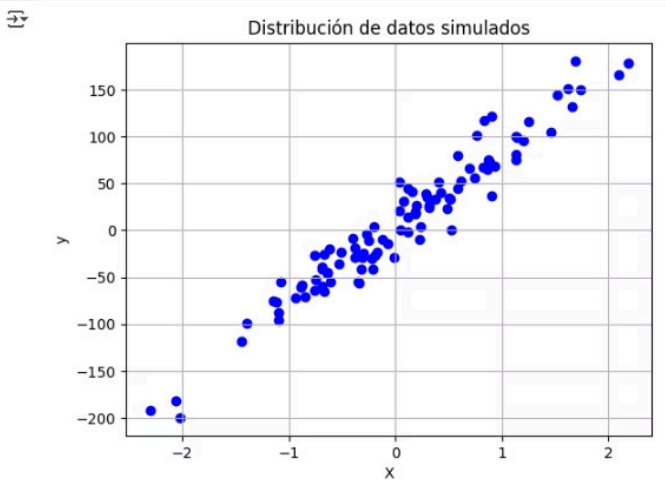


Imagen 4
Los puntos generados simulan una relación lineal con algo de ruido. Esta visualización inicial será la base para observar cómo distintos modelos intentan ajustarse a estos datos.

Paso 2: Entrenar modelos con distintos grados de complejidad

```
[2] # Paso 2: Entrenamos modelos de regresión polinomial con diferentes grados

    from sklearn.preprocessing import PolynomialFeatures
    from sklearn.linear_model import LinearRegression

    # Definimos tres niveles de complejidad: simple (1), medio (4), complejo (15)
    grados = [1, 4, 15]

    # Creamos una figura con 3 gráficos lado a lado
    plt.figure(figsize=(12, 4))

    # Para cada grado, entrenamos y visualizamos el modelo correspondiente
    for i, grado in enumerate(grados):
        # Convertimos X en variables polinomiales según el grado
        poly = PolynomialFeatures(degree=grado)
        X_poly = poly.fit_transform(X)

        # Creamos y entrenamos el modelo
        modelo = LinearRegression().fit(X_poly, y)

        # Realizamos predicciones con los mismos datos
        y_pred = modelo.predict(X_poly)

        # Dibujamos el gráfico: puntos reales (azul) y predicciones (línea roja)
        plt.subplot(1, 3, i+1)
        plt.scatter(X, y, color='blue', s=10)
        plt.plot(X, y_pred, color='red')
        plt.title(f'Modelo de grado {grado}')
        plt.grid(True)
        plt.tight_layout()

    # Mostramos los 3 gráficos juntos
    plt.show()
```

Imagen 5

Imagen 6

Comparamos tres modelos con distinta capacidad de aprendizaje:

Grado 1: Modelo lineal muy simple.

Grado 4: Modelo más flexible, capaz de adaptarse a la forma general.

Grado 15: Modelo muy complejo, capaz de memorizar incluso el ruido.

Observemos las curvas generadas y respondamos en conjunto:

- ¿Cuál de los modelos parece subajustado (no sigue la forma de los datos)?
- ¿Cuál muestra un ajuste adecuado (captura la tendencia general sin exagerar)?
- ¿Cuál parece sobreajustado (se adapta demasiado incluso al ruido)?

Ahora podemos concluir:

Grado 1: modelo demasiado simple → subajuste.

Grado 4: buen equilibrio → ajuste adecuado.

Grado 15: modelo demasiado complejo → sobreajuste.

Sesgo y varianza: el gran dilema

Una vez que comprendemos qué es el sobreajuste y el subajuste, es importante ir más allá y entender qué los causa. Uno de los factores clave que influyen en este equilibrio es la relación entre sesgo y varianza. Esta tensión natural es uno de los conceptos más importantes en el diseño de modelos de Machine Learning y afecta directamente la capacidad del modelo para generalizar.

¿QUÉ ES EL SESGO (BIAS)?

El sesgo se refiere a los errores que comete un modelo por hacer suposiciones simplificadas sobre los datos. Un modelo con alto sesgo no logra representar la complejidad real del problema.

Por ejemplo, un modelo que siempre predice "promedio" sin importar las variables de entrada tiene alto sesgo: ignora diferencias importantes.

Características de un modelo con alto sesgo:

- Aprende poco o nada.
- Es rígido y no se adapta a los datos.
- Tiende al subajuste.

¿QUÉ ES LA VARIANZA?

La varianza mide la sensibilidad del modelo a pequeñas variaciones en los datos de entrenamiento. Un modelo con alta varianza cambia mucho si se entrena con distintos subconjuntos de datos.

Un modelo que se comporta de forma muy distinta dependiendo del conjunto de entrenamiento usado.

Características de un modelo con alta varianza:

- Aprende demasiado.
- Es muy complejo y se adapta incluso al ruido.
- Tiende al sobreajuste.

El trade-off entre sesgo y varianza

No se puede minimizar ambos al mismo tiempo. A medida que reducimos el sesgo (haciendo el modelo más flexible), la varianza tiende a aumentar. Y si reducimos la varianza (haciendo el modelo más rígido), el sesgo aumenta.

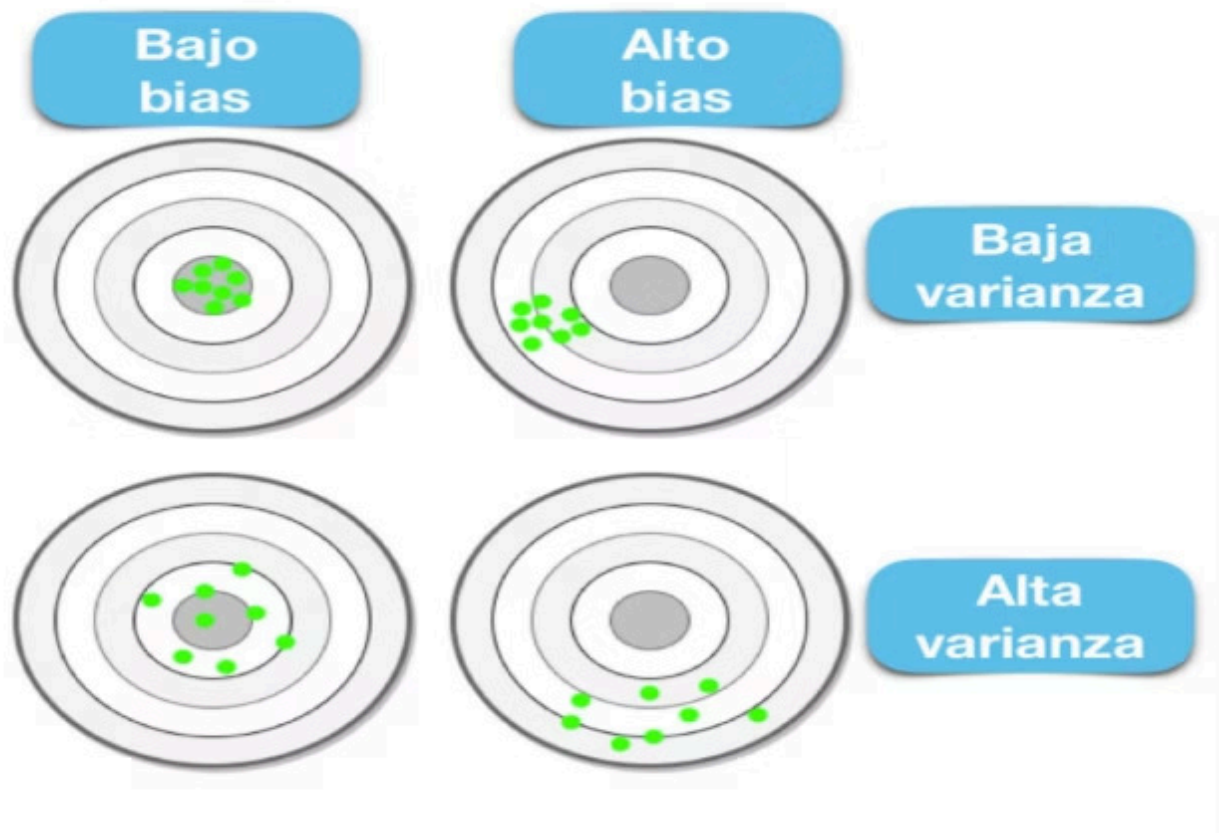
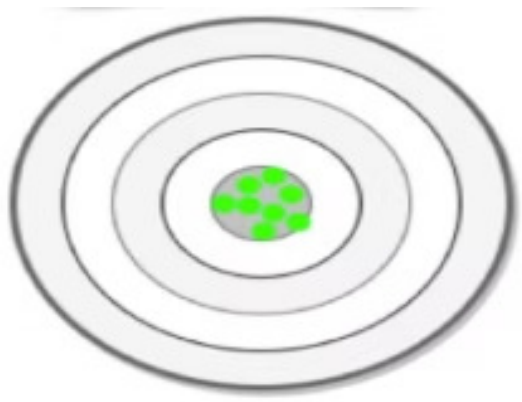


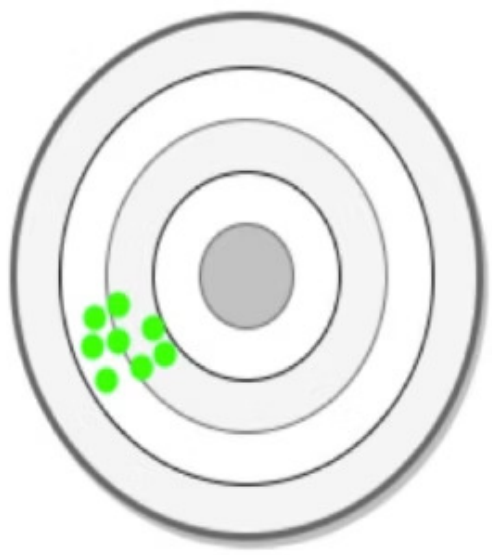
Imagen 7: Visualización del dilema sesgo-varianza

Esta imagen, tomada del sitio **Acturio (2022)**, representa cómo se comportan diferentes modelos de Machine Learning en términos de bias (sesgo) y varianza utilizando una metáfora de tiro al blanco:

Bajo bias, baja varianza: los disparos (predicciones) son precisos y consistentes: el modelo aprende y generaliza bien.

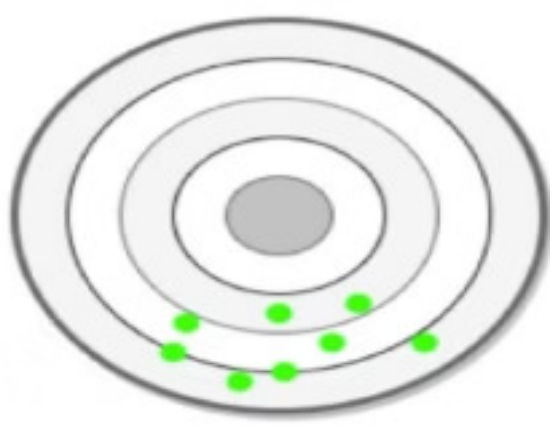


Alto bias, baja varianza: las predicciones son consistentes, pero están lejos del objetivo: el modelo no capta la complejidad del problema (subajuste).



Bajo bias, alta varianza: el modelo aprende, pero es inestable: las predicciones varían demasiado con cambios en los datos (sobreajuste).

Alto bias, alta varianza: predicciones malas e inestables: el peor de los escenarios.



Fuente: **Acturio (2022)**

El objetivo del aprendizaje automático es encontrar el punto óptimo: donde el modelo tenga el menor error de validación posible, logrando un equilibrio entre sesgo y varianza.

Ejercicio práctico: Explorando el sesgo y la varianza en modelos polinomiales

Este ejercicio tiene como objetivo ayudarte a visualizar el comportamiento del sesgo y la varianza al variar la complejidad de un modelo de regresión polinomial. A través de un ejemplo práctico, comprenderás cómo un modelo demasiado simple puede subajustarse (alto sesgo) y cómo uno demasiado complejo puede sobreajustarse (alta varianza).

Paso 1: Importación de librerías y generación de datos

Generamos un dataset simple con 1 variable explicativa (X) y un objetivo (y), agregando ruido para simular condiciones reales. Esto nos permitirá ver cómo distintos modelos se comportan frente a datos con variabilidad.

```
[0]  # Paso 1: Importar las librerías necesarias para el ejercicio
     from sklearn.datasets import make_regression         # Generar datos sintéticos para regresión
     from sklearn.preprocessing import PolynomialFeatures # Para generar variables polinomiales
     from sklearn.linear_model import LinearRegression    # El modelo de regresión lineal
     from sklearn.pipeline import make_pipeline           # Permite combinar transformación + modelo
     from sklearn.model_selection import cross_val_score  # Para aplicar validación cruzada
     import matplotlib.pyplot as plt                      # Para graficar los resultados
     import numpy as np                                   # Para trabajar con arreglos numéricos

     # Creamos un conjunto de datos con 1 sola variable (X) y una respuesta (y)
     # Añadimos ruido (noise=20) para simular condiciones del mundo real
     X, y = make_regression(n_samples=100, n_features=1, noise=20, random_state=0)
```

Imagen 8 :Generamos un dataset simple con 1 variable explicativa (X) y un objetivo (y), agregando ruido para simular condiciones reales. Esto nos permitirá ver cómo distintos modelos se comportan frente a datos con variabilidad

Paso 2: Entrenar modelos con diferentes grados y evaluar su desempeño

```
[3]  # Paso 2: Evaluar modelos de regresión polinomial con distintos grados de complejidad

     grados = range(1, 15)    # Probaremos modelos desde grado 1 (línea recta) hasta grado 14 (muy complejos)
     val_scores = []          # Lista para guardar los errores de validación cruzada

     # Recorremos todos los grados y entrenamos un modelo para cada uno
     for grado in grados:
         # Creamos un pipeline que:
         # 1. Transforma los datos con PolynomialFeatures
         # 2. Aplica un modelo de regresión lineal sobre esos datos transformados
         modelo = make_pipeline(PolynomialFeatures(degree=grado), LinearRegression())

         # Evaluamos el modelo usando validación cruzada (5 particiones del dataset)
         # Esto nos da una idea de qué tan bien generaliza el modelo
         scores = cross_val_score(modelo, X, y, cv=5, scoring='neg_mean_squared_error')

         # Calculamos el promedio de los errores obtenidos y lo almacenamos
         # (negamos el resultado porque sklearn devuelve el error negativo)
         val_scores.append(-scores.mean())
```

Imagen 9:Entrenamos múltiples modelos de regresión polinomial, comenzando con los más simples (grado 1) y avanzando hacia modelos mucho más complejos (hasta grado 14). Para cada modelo, evaluamos su rendimiento utilizando validación cruzada, una técnica que nos permite medir qué tan bien generaliza el modelo a nuevos datos. Esto es clave para entender el equilibrio entre sesgo y varianza: qué tanto aprende el modelo y qué tan estable es cuando cambia el conjunto de entrenamiento.

Paso 3: Visualizar el trade-off entre sesgo y varianza

```
[4]  # Paso 3: Visualizar el error según la complejidad del modelo

     # Graficamos los errores de validación en función del grado del polinomio
     plt.plot(grados, val_scores, label='Error de Validación (CV)', color='red')

     # Etiquetas y título del gráfico
     plt.xlabel("Grado del polinomio")                # Complejidad del modelo
     plt.ylabel("Error cuadrático medio")             # Medida del error
     plt.title("Trade-off entre sesgo y varianza")    # Interpretación del gráfico

     # Estética del gráfico
     plt.legend()
     plt.grid()
     plt.show()
```

Imagen 10: El gráfico muestra cómo varía el error de validación a medida que aumenta la complejidad del modelo:

1. Al principio, el modelo es demasiado simple → alto sesgo, alto error.
2. Luego, el modelo mejora → ajuste adecuado, menor error.
3. Finalmente, el modelo es demasiado complejo → alta varianza, el error vuelve a aumentar por sobreajuste.

Observen que existe un punto óptimo donde el error de validación es más bajo. Ese punto representa el equilibrio entre sesgo y varianza, y es ahí donde el modelo generaliza mejor.

¿CÓMO PODEMOS REDUCIR ESTE TRADE-OFF?

Algunas estrategias comunes para mejorar el equilibrio entre sesgo y varianza incluyen:

Estrategia	Efecto
Aumentar la cantidad de datos	Reduce la varianza
Seleccionar un modelo más simple	Reduce la varianza, pero puede aumentar el sesgo
Aplicar regularización (L1, L2)	Controla la complejidad del modelo
Validación cruzada	Detecta sobreajuste antes de poner en producción

El trade-off entre sesgo y varianza no es un obstáculo, sino una guía. Comprenderlo nos permite ajustar el modelo justo donde es más útil: ni tan simple que ignore los datos, ni tan complejo que los memorice.

Este balance es lo que convierte al Machine Learning en un arte técnico: no buscamos el modelo más complejo ni el más preciso en entrenamiento, sino el más confiable en producción.

Técnicas de validación cruzada

Hasta ahora hemos aprendido que un buen modelo debe generalizar bien a nuevos datos. Para saber si eso ocurre, no basta con ver cómo se comporta en el entrenamiento: necesitamos validarlo con datos que no haya visto antes.

La validación cruzada es una estrategia fundamental para lograr esto. Nos permite estimar de manera confiable el rendimiento del modelo y evitar tanto el sobreajuste como el subajuste.

¿Por qué validar un modelo?

Porque lo importante no es que el modelo recuerde lo que ya vio, sino que sepa cómo actuar frente a nuevos casos.

En esta sección aprenderás cuatro técnicas comunes de validación, sus características y cuándo usarlas.

Retención simple (Holdout)

Es la forma más básica de validación.

¿Cómo funciona?

Se divide el conjunto de datos en dos partes:

- Entrenamiento: para ajustar el modelo (por ejemplo, 80% de los datos).
- Prueba (test): para evaluar el rendimiento (por ejemplo, 20% restantes).

```python
# Paso 1: Importar las librerías necesarias
from sklearn.datasets import make_regression              # Para crear datos simulados
from sklearn.model_selection import train_test_split      # Para dividir el dataset
import pandas as pd                                       # Para mostrar los datos en formato de tabla

# Paso 2: Generar datos sintéticos para regresión
# Creamos 100 muestras con 1 sola característica y algo de ruido (simula datos reales)
X, y = make_regression(n_samples=100, n_features=1, noise=15, random_state=1)

# Paso 3: Dividir los datos en conjuntos de entrenamiento (80%) y prueba (20%)
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42)

# Paso 4: Imprimir el tamaño de cada subconjunto
print("Tamaño de los conjuntos:")
print(f"X_train: {X_train.shape}, y_train: {y_train.shape}")
print(f"X_test:  {X_test.shape}, y_test:  {y_test.shape}")

# Paso 5: Mostrar una muestra de los datos
print("\n Primeras 5 filas del conjunto de entrenamiento:")
df_train = pd.DataFrame({'X': X_train.flatten(), 'y': y_train})
print(df_train.head())

print("\n Primeras 5 filas del conjunto de prueba:")
df_test = pd.DataFrame({'X': X_test.flatten(), 'y': y_test})
print(df_test.head())
```

Ventajas:

- Rápido y fácil de implementar.

Desventajas:

- Inestable: el resultado depende mucho de cómo se dividieron los datos.
- Puede desperdiciar información si el dataset es pequeño.

Validación Cruzada K-Fold

Es una de las técnicas más utilizadas. Más robusta que el holdout.

¿Cómo funciona?

- Divide los datos en K partes iguales (llamadas folds).
- Entrena el modelo K veces, usando K−1 folds como entrenamiento y 1 como prueba.
- Calcula el promedio de los resultados.

Ejemplo clásico: K = 5

Se entrena 5 veces con 80% de los datos y se evalúa con 20% distintos cada vez.

```
[8]  # Importamos las funciones necesarias de scikit-learn
     from sklearn.model_selection import cross_val_score, KFold   # cross_val_score para evaluar el modelo con validación cruzada
     from sklearn.linear_model import LinearRegression             # Usaremos regresión lineal como modelo de ejemplo

     # Creamos una instancia del modelo de regresión lineal
     modelo = LinearRegression()

     # Definimos el esquema de validación cruzada K-Fold
     # n_splits=5 → dividimos el dataset en 5 partes (folds)
     # shuffle=True → mezclamos los datos antes de dividirlos (importante para que no queden ordenados)
     # random_state=1 → asegura que los resultados sean reproducibles (la misma división cada vez)
     kf = KFold(n_splits=5, shuffle=True, random_state=1)

     # Aplicamos validación cruzada al modelo con el esquema definido
     # cv=kf → usamos el objeto KFold como esquema de validación
     # scoring='r2' → usamos el coeficiente de determinación R² como métrica (0 a 1, donde 1 es mejor)
     scores = cross_val_score(modelo, X, y, cv=kf, scoring='r2')

     # Mostramos los resultados obtenidos en cada uno de los 5 folds
     print(f"Resultados por fold: {scores}")

     # Calculamos y mostramos el promedio de R² como una medida general de desempeño
     print(f"Promedio R²: {scores.mean():.2f}")

Resultados por fold: [0.96139046 0.96324158 0.95495953 0.96631891 0.96875135]
Promedio R²: 0.96
```

Imagen 11: Esquema visual del proceso de K-Fold Cross Validation

Ventajas:

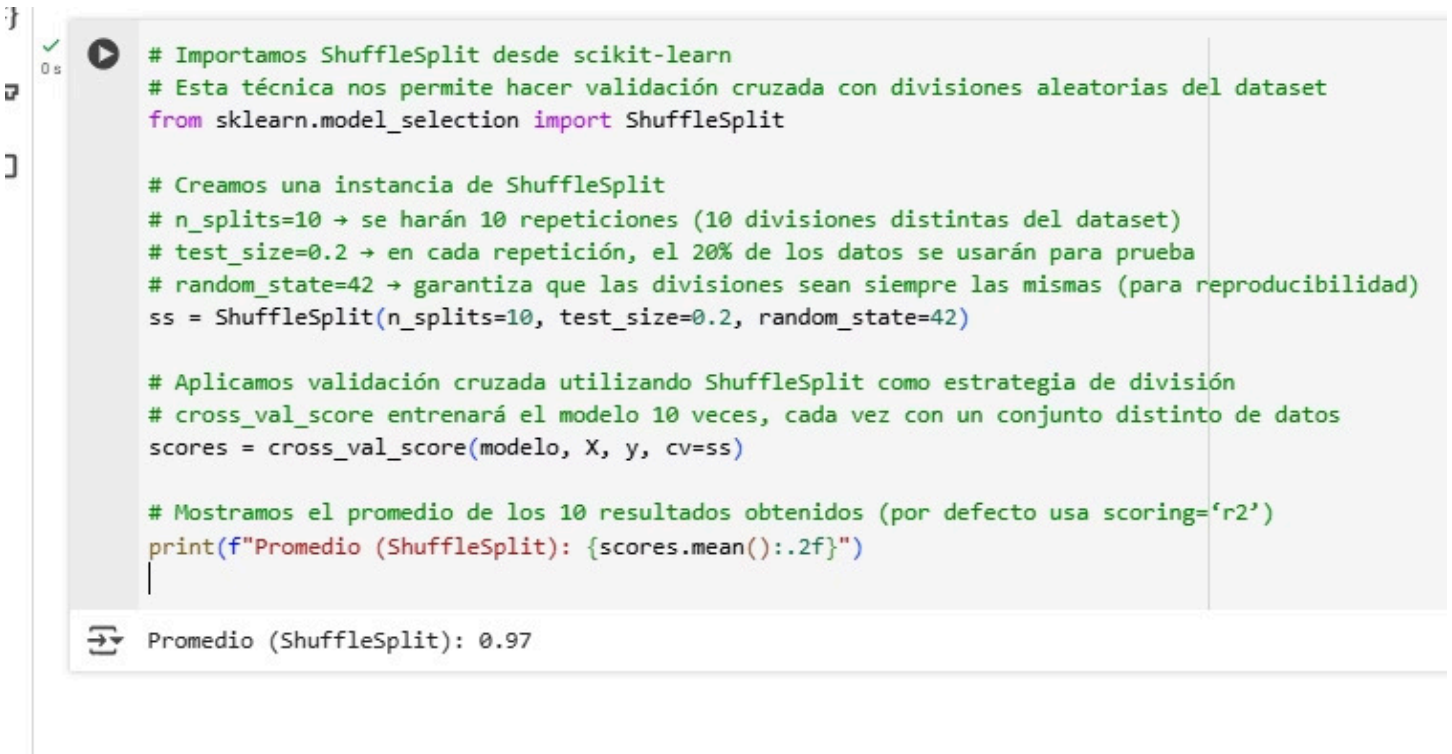
- Usa todos los datos para entrenar y validar.
- Más estable y representativo.
- Cuanto mayor sea K, mayor la precisión… pero también mayor el tiempo de cómputo.

Validación cruzada aleatoria (Shuffle Split)

Es una alternativa flexible entre holdout y K-Fold.

¿Cómo funciona?

- Realiza múltiples divisiones aleatorias del dataset.
- Para cada partición, se evalúa el modelo.
- Se promedian los resultados.



```
# Importamos ShuffleSplit desde scikit-learn
# Esta técnica nos permite hacer validación cruzada con divisiones aleatorias del dataset
from sklearn.model_selection import ShuffleSplit

# Creamos una instancia de ShuffleSplit
# n_splits=10 → se harán 10 repeticiones (10 divisiones distintas del dataset)
# test_size=0.2 → en cada repetición, el 20% de los datos se usarán para prueba
# random_state=42 → garantiza que las divisiones sean siempre las mismas (para reproducibilidad)
ss = ShuffleSplit(n_splits=10, test_size=0.2, random_state=42)

# Aplicamos validación cruzada utilizando ShuffleSplit como estrategia de división
# cross_val_score entrenará el modelo 10 veces, cada vez con un conjunto distinto de datos
scores = cross_val_score(modelo, X, y, cv=ss)

# Mostramos el promedio de los 10 resultados obtenidos (por defecto usa scoring='r2')
print(f"Promedio (ShuffleSplit): {scores.mean():.2f}")
```

```
Promedio (ShuffleSplit): 0.97
```

Similar a hacer muchos holdouts, pero promediando los resultados.

Leave-One-Out (LOOCV)

Leave-One-Out Cross-Validation (LOOCV) es una forma especial de validación cruzada donde se usa una sola observación como conjunto de prueba en cada iteración, y el resto del dataset como conjunto de entrenamiento.

Es considerado un caso extremo de K-Fold, donde K es igual al número total de observaciones.

¿Cómo funciona?

- En cada iteración, se deja un dato fuera para probar.
- El modelo se entrena con todos los demás datos.
- Se repite el proceso una vez por cada observación del dataset.

Ejemplo práctico (con regresión)

Usamos una métrica adecuada: Error Cuadrático Medio (MSE), ya que R^2 no es válido cuando se evalúa con una sola muestra.

[13]

```python
from sklearn.datasets import make_regression
from sklearn.linear_model import LinearRegression
from sklearn.model_selection import LeaveOneOut, cross_val_score

# Generamos un dataset simple
X, y = make_regression(n_samples=20, n_features=1, noise=10, random_state=0)

# Creamos el modelo de regresión lineal
modelo = LinearRegression()

# Creamos el esquema LOOCV
loocv = LeaveOneOut()

# Usamos MSE como métrica (negativo por convención en sklearn)
scores = cross_val_score(modelo, X, y, cv=loocv, scoring='neg_mean_squared_error')

# Mostramos el promedio del error (convertido a positivo)
print(f"Promedio del error (LOOCV - MSE): {-scores.mean():.2f}")
```

```
Promedio del error (LOOCV - MSE): 85.87
```

Ventajas y desventajas de LOOCV

Ventajas de LOOCV

- Utiliza casi todos los datos para entrenar en cada iteración.
- Ideal para datasets pequeños donde cada muestra es valiosa.
- Produce una estimación de rendimiento muy estable (baja varianza).

Desventajas

- Costoso computacionalmente: entrena el modelo tantas veces como observaciones haya.
- En datasets grandes puede volverse lento.

La validación cruzada es la brújula que guía al ingeniero de datos para saber si va en la dirección correcta. No es un paso opcional: es el control de calidad del modelo.

Usar la técnica adecuada según el tamaño del dataset, la complejidad del modelo y los recursos disponibles es clave para diseñar soluciones robustas y confiables.

Implementación práctica con Scikit-learn

Recuerda:

Técnica	¿Cómo funciona?	Ventajas principales	Desventajas principales	Ideal cuando...
Holdout	Divide el dataset una sola vez en entrenamiento y prueba (por ejemplo, 80/20).	Muy rápida y fácil de implementar.	Inestable. Resultados dependen de una única partición.	Se necesita evaluación rápida o un punto de partida.
K-Fold	Divide en K partes iguales, entrena K veces usando K−1 para entrenar y 1 para prueba.	Más estable. Usa todos los datos y minimiza el sesgo.	Puede ser más costoso computacionalmente.	Se busca una evaluación robusta y balanceada.
ShuffleSplit	Genera múltiples divisiones aleatorias con proporciones definidas.	Flexible y aleatorio. Más representativo que un solo Holdout.	No garantiza que todos los datos se usen como test.	Queremos simular muchos escenarios posibles.
LOOCV	Deja un dato fuera para prueba en cada iteración (tantas como muestras).	Usa casi todos los datos para entrenar. Muy preciso.	Muy lento en datasets grandes. Sensible al ruido.	El dataset es pequeño y queremos máxima precisión

Archivos del repositorio para práctica

Antes de avanzar a la siguiente sesión, te invitamos a detenerte un momento y revisar tu trabajo práctico en Jupyter Notebook o Google Colab.

Durante esta sesión, ya implementaste los principales enfoques de validación cruzada utilizando la librería Scikit-learn. Si aún no lo hiciste, ahora es un excelente momento para abrir cada notebook correspondiente y repasar lo que hiciste en clase.

Archivos del repositorio:

3_1_holdout.ipynb → Validación Holdout

3_2_validación_cruzada_k_fold.ipynb → K-Fold

3_3_validacion_cruzada_aleatoria_ShuffleSplit.ipynb → Shuffle Split

3_4_alidacion_cruzada_leave_one_out_LOOCV.ipynb → Leave-One-Out (LOOCV)