¿Qué es la Evaluación de modelos y métricas de rendimiento.?

```
<https://colab.research.google.com/drive/1LE_iqYKa2di0MASjTbpC-
1NwOXdFmOqZ#scrollTo=3WHQfwy8l2G5>
```

La **evaluación de modelos** es el proceso de cuantificar qué tan bien predice un algoritmo de Inteligencia Artificial sobre datos que no ha visto antes. Su objetivo es asegurar que el modelo sea capaz de generalizar y no simplemente de memorizar los datos de entrenamiento (un problema conocido como *overfitting*).

Las **métricas de rendimiento** son las reglas de medida 📏 específicas que usamos para ponerle una "nota" numérica a esa evaluación. Dependiendo del tipo de problema, usaremos diferentes métricas para entender los errores del modelo.

Puede enfocarse desde un punto de vista de:

1. **Métricas para Clasificación**: El modelo predice categorías (ej. ¿es fraude o no?). Se explora la matriz de confusión, precisión y exhaustividad.
Se centra en problemas donde el modelo asigna categorías. Aquí exploraremos por qué la exactitud (**Accuracy**) puede ser engañosa si los datos no están equilibrados. Aparecen herramientas como **Matriz de Confusión**, el **F1-Score** o la **Curva ROC**.

2. **Métricas para Regresión**: El modelo predice valores numéricos continuos (ej. el precio de una casa).
Aparecen conceptos como el **Error Absoluto Medio (MAE)**, el **Error Cuadrático Medio (MSE)** y el **Coeficiente de Determinación (R^2)** para entender cuánto se alejan nuestras predicciones de la realidad.

3. **Estrategias de Validación**: Cómo dividir los datos correctamente usando Python 3 para que la evaluación sea justa (Train/Test Split y Validación Cruzada).
Antes de medir, hay que saber cómo organizar los datos. Veremos la diferencia entre un simple **Train/Test Split** y técnicas más robustas como la **Validación Cruzada (K-Fold Cross Validation)**.

La **evaluación en clasificación** no se trata solo de ver si el modelo "acierta", sino de entender *cómo* y *dónde* se equivoca. 🎯

Métricas para Clasificación

1. Conceptos Base

- **Predicción de categorías:** Es el proceso donde el modelo asigna una etiqueta discreta a una entrada (ej. "Spam" o "No Spam"). A diferencia de la regresión, aquí no buscamos un número, sino una clase. 🏷️

- **Exactitud (Accuracy):** Es el porcentaje total de predicciones correctas. Se calcula como:

$$\text{Accuracy} = \frac{\text{Aciertos Totales}}{\text{Total de Casos}}$$

- **Datos equilibrados (balanceados):** Ocurre cuando las clases que queremos predecir tienen un número similar de ejemplos. Si tienes 500 fotos de gatos 🐈 y 500 de perros 🐕, tus datos están balanceados. Si tienes 990 de gatos y 10 de perros, están **desbalanceados**, y el Accuracy dejará de ser una métrica fiable.

2. Herramientas de Medición

- **Matriz de Confusión:** Es una tabla que muestra los aciertos y errores desglosados en cuatro categorías: **Verdaderos Positivos (TP)**, **Verdaderos Negativos (TN)**, **Falsos Positivos (FP)** y **Falsos Negativos (FN)**. Es el "mapa" de los errores del modelo.

Para entender estos conceptos, imaginemos un **test médico** 🩺 para detectar una enfermedad. En este escenario, ser "Positivo" significa tener la enfermedad y ser "Negativo" significa estar sano.

Concepto	Lo que dice el modelo	Realidad	Resultado
Verdadero Positivo (TP) ✅	Positivo	Positivo	El test detecta correctamente la enfermedad.
Verdadero Negativo (TN) 🛡️	Negativo	Negativo	El test confirma correctamente que la persona está sana.
Falso Positivo (FP) 🔔	Positivo	Negativo	**Falsa alarma:** El test dice que hay enfermedad, pero la persona está sana.
Falso Negativo (FN) ⚠️	Negativo	Positivo	**Error peligroso:** El test dice que la persona está sana, pero en realidad está enferma.

Estos cuatro valores son los "ladrillos" con los que construimos todas las métricas de clasificación. Dependiendo del problema, nos preocupará más un tipo de error que otro.

- **Precisión (Precision):** ¿Qué tan fiable es el modelo cuando dice que algo es positivo? Responde a: *"De todos los que predije como positivos, ¿cuántos lo eran realmente?"* 💎
- **Exhaustividad (Recall/Sensitivity):** ¿Qué capacidad tiene el modelo para encontrar todos los casos positivos? Responde a: *"De todos los casos que eran realmente positivos, ¿cuántos logré detectar?"* 🔍
- **F1-Score:** Es la media armónica entre la Precisión y la Exhaustividad. Es muy útil cuando quieres un equilibrio entre ambas y tienes clases desbalanceadas. ⚖️

La **media armónica** es una métrica. Está diseñada para proporcionar un único valor que equilibre un conjunto de datos desbalanceado.

Fórmula Matemática 🧮

A diferencia del promedio normal (media aritmética), la media armónica se calcula de la siguiente manera:

$$F1 = 2 \cdot \frac{\text{Precisión} \cdot \text{Exhaustividad}}{\text{Precisión} + \text{Exhaustividad}}$$

¿Por qué usamos la media armónica y no la normal? 🤔

La media armónica **penaliza los valores extremos**.

- Si la precisión es 1.0 (perfecta) pero la exhaustividad es 0.0 (pésima), la **media aritmética** te daría un 0.5, lo cual parece aceptable.
- Sin embargo, la **media armónica (F1-Score)** te daría un 0, reflejando que el modelo realmente no es útil porque falla completamente en una de las dos áreas.

Para calcular estas métricas, utilizamos estas fórmulas:

Precisión (Precision) 🎯

La precisión mide qué tan "limpias" son nuestras predicciones positivas. Es la proporción de aciertos positivos sobre **todo lo que el modelo marcó como positivo** (aciertos y errores).

$$Precisión = \frac{TP}{TP + FP}$$

Exhaustividad (Recall) 🔍

La exhaustividad (también llamada sensibilidad) mide la capacidad del modelo para encontrar **todos** los casos positivos reales. Es la proporción de aciertos positivos sobre **el total de casos que realmente eran positivos**.

$$Exhaustividad = \frac{TP}{TP + FN}$$

Código de ejemplo en Python

Ejemplo de **clasificación binaria** (por ejemplo, detectar si un mensaje es "Spam" o "No Spam").

```
from sklearn.metrics import confusion_matrix, accuracy_score,
precision_score, recall_score, f1_score

# 1. Definimos datos de ejemplo
```

```
# y_true: Las etiquetas reales (la verdad)
# y_pred: Lo que nuestro modelo de IA ha predicho
# (0 = No Spam, 1 = Spam)
y_true = [0, 1, 0, 0, 1, 1, 0, 1, 1, 1]
y_pred = [0, 1, 0, 0, 0, 1, 1, 1, 1, 1]

# 2. Calculamos la Matriz de Confusión
# El orden por defecto es: [TN, FP], [FN, TP]
tn, fp, fn, tp = confusion_matrix(y_true, y_pred).ravel()

print(f"--- Matriz de Confusión ---")
print(f"Verdaderos Negativos (TN): {tn}")
print(f"Falsos Positivos (FP): {fp}")
print(f"Falsos Negativos (FN): {fn}")
print(f"Verdaderos Positivos (TP): {tp}\n")

# 3. Calculamos las métricas fundamentales
print(f"--- Métricas de Rendimiento ---")
print(f"Exactitud (Accuracy): {accuracy_score(y_true, y_pred):.2f}")
print(f"Precisión (Precision): {precision_score(y_true, y_pred):.2f}")
print(f"Exhaustividad (Recall): {recall_score(y_true, y_pred):.2f}")
print(f"F1-Score: {f1_score(y_true, y_pred):.2f}")
```

¿Qué está pasando en el código? 🧐

1. `y_true` **vs** `y_pred`: Comparamos la realidad con la predicción. Fíjate que en la lista hay 10 elementos.
2. `ravel()`: Es un pequeño truco de Python para extraer los cuatro valores de la matriz (`tn, fp, fn, tp`) directamente de la tabla que genera `scikit-learn`.
3. **Métricas**: Usamos las funciones integradas que aplican las fórmulas matemáticas que mencionaste antes ($F1$, $Precisión$, etc.).

Matriz de Confusión 🧩

Comparando elemento a elemento `y_true` y `y_pred`:

Índice	Real (y_true)	Predicción (y_pred)	Resultado
0, 2, 3	0	0	**3 TN** (Acierto en Negativo)
1, 5, 7, 8, 9	1	1	**5 TP** (Acierto en Positivo)
6	0	1	**1 FP** (Falsa Alarma)
4	1	0	**1 FN** (Se le escapó un positivo)

- **TP (Verdaderos Positivos):** 5 ✅

- **TN (Verdaderos Negativos):** 3 🛡️
- **FP (Falsos Positivos):** 1 🔔
- **FN (Falsos Negativos):** 1 ⚠️

---

2. Modelo "Perezoso" y Confianza 💤

Un **modelo perezoso** (o *baseline*) es aquel que no aprende patrones, sino que simplemente predice siempre la clase mayoritaria.

Si en tus datos el 60% es "Spam" y el modelo dice siempre "Spam":

- Su **Accuracy** será del 60%.
- Su **Precisión** (confianza) cae drásticamente. ¿Por qué? Porque la precisión mide qué tan seguro puedes estar cuando el modelo dice "1". Si el modelo dice "1" para todo, pierde valor. En este caso, de cada 10 veces que dice "Spam", fallará 4 veces (los ceros reales).

---

3. Informe Automático en Python 🐍

En `scikit-learn`, se pueden obtener todas las métricas (Precisión, Recall, F1-Score y Accuracy) desglosadas por clase con una sola función:

```
from sklearn.metrics import classification_report

print(classification_report(y_true, y_pred))
```

Curva ROC (Receiver Operating Characteristic)

Herramienta visual para evaluar qué tan bien un modelo de clasificación puede distinguir entre dos clases (como "Sano" vs. "Enfermo") a medida que cambiamos el **umbral de decisión**. 📈

En lugar de mirar un solo número, la curva ROC nos muestra el panorama completo al graficar dos métricas enfrentadas estableciendo relaciones (**Tasas**):

1. **Tasa de Verdaderos Positivos (TPR / Sensibilidad):** De todos los positivos reales, ¿cuántos detectamos correctamente?

$$TPR = \frac{TP}{TP + FN}$$

2. **Tasa de Falsos Positivos (FPR):** De todos los negativos reales, ¿cuántos marcamos incorrectamente como positivos?

$$FPR = \frac{FP}{FP + TN}$$

Ejemplo práctico: Radar de Aviones 🛩️

Diseñas un radar para detectar aviones enemigos. El radar recibe señales y debes decidir qué tan fuerte debe ser la señal para activar la alarma (**umbral**):

- **Umbral muy bajo:** El radar es súper sensible. Detectarás todos los aviones (TPR = 1.0), pero también te darán "falsas alarmas" por bandadas de pájaros o nubes (FPR muy alto). Estarías en la esquina superior derecha de la curva. 🐓
- **Umbral muy alto:** El radar es muy estricto. Solo suena si la señal es gigante. No tendrás falsas alarmas (FPR = 0.0), pero se te escaparán aviones reales (TPR bajo). Estarías en la esquina inferior izquierda. 🛡️
- **El Modelo Ideal:** Sería aquel que detecta todos los aviones (TPR = 1.0) sin dar ninguna falsa alarma (FPR = 0.0). En la gráfica, esto es la esquina superior izquierda.

El **AUC (Área Bajo la Curva)** es el número que resume esta gráfica. Un AUC de **1.0** es un modelo perfecto, mientras que un **0.5** es como lanzar una moneda al aire (puro azar). 🎲

1. ¿Por qué el AUC es mejor que el Accuracy? 📉

El **Accuracy** es como un profesor que solo cuenta cuántas respuestas están bien, sin mirar si las preguntas eran fáciles o difíciles. Si en un examen de 100 preguntas, 99 son de "sumar 1+1" y solo 1 es de "cálculo avanzado", alguien que no sepa nada de cálculo pero sí de sumas sacará un 99% de nota. ¿Es un experto? No, solo aprovechó el **desbalance de los datos**.

El **AUC (Area Under the Curve)**, en cambio, mide la capacidad del modelo para **ordenar** las probabilidades. Evalúa si el modelo es capaz de poner a los "positivos" por encima de los "negativos" en una lista de probabilidades, sin importar dónde pongamos el punto de corte (umbral).

- **Accuracy:** Se ve afectado por la proporción de clases.
- **AUC:** Es robusto frente al desbalance porque mira el rendimiento en todos los umbrales posibles.

2. Umbral de Decisión y la Curva ROC 🦸

Por defecto, un modelo suele decir que algo es "Positivo" si su probabilidad es mayor a **0.5**. Pero ese número no es sagrado.

Imagina un detector de incendios 🚒:

- Si bajas el **umbral** a 0.1, el sensor saltará con el mínimo humo (mucha **Exhaustividad/Recall**), pero tendrás muchas falsas alarmas (Falsos Positivos).
- Si subes el **umbral** a 0.9, solo sonará si hay una hoguera en la cocina (mucha **Precisión**), pero quizás se te queme la casa antes de que suene (Falsos Negativos).

3. Código en Python 3

```
import matplotlib.pyplot as plt
from sklearn.datasets import make_classification
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import roc_curve, roc_auc_score

# 1. Generamos datos desbalanceados (90% clase 0, 10% clase 1)
X, y = make_classification(n_samples=1000, n_classes=2, weights=[0.9, 0.1],
random_state=42)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3,
random_state=42)

# 2. Entrenamos un modelo simple
model = LogisticRegression()
model.fit(X_train, y_train)

# 3. Obtenemos las probabilidades (necesarias para la curva ROC)
# Tomamos la columna [:, 1] que es la probabilidad de ser clase "1"
probs = model.predict_proba(X_test)[:, 1]

# 4. Calculamos el AUC y la curva
auc = roc_auc_score(y_test, probs)
fpr, tpr, thresholds = roc_curve(y_test, probs)

# 5. Visualización
plt.figure(figsize=(7, 5))
plt.plot(fpr, tpr, label=f'Modelo (AUC = {auc:.2f})')
plt.plot([0, 1], [0, 1], 'k--', label='Azar (AUC = 0.50)') # Línea de
referencia
plt.xlabel('Tasa de Falsos Positivos (FPR)')
plt.ylabel('Tasa de Verdaderos Positivos (TPR)')
plt.title('Curva ROC')
plt.legend()
plt.grid(True)
plt.show()
```

Planteamiento:

Para el diseño de un sistema para detectar **fraude bancario** (donde el 99.9% de las transacciones son legales y solo el 0.1% son fraude), ¿qué pasaría si se usara un umbral muy alto (0.95)? ¿Estaría el banco más preocupado por molestar a clientes inocentes o por

dejar pasar a un estafador?
¿Qué métrica sería prioritaria en ese caso?

Métricas para Regresión

Pasamos ahora al terreno de la **Regresión** 📈. A diferencia de la clasificación, donde se buscan etiquetas, aquí el objetivo es predecir un valor numérico continuo (como el precio de una vivienda o la temperatura).

En este contexto, la evaluación se basa en medir la **"distancia"** o el error entre el valor real (y) y la predicción del modelo ($\hat{y}$). Vamos a explorar estas métricas clave:

1. Error Absoluto Medio (MAE) 📏

Es la métrica más intuitiva. Simplemente calcula el promedio de las diferencias absolutas entre la realidad y la predicción.

- **Fórmula:** $\text{MAE} = \frac{1}{n}\sum_{i=1}^{n}|y_i - \hat{y}_i|$
- **En lenguaje sencillo:** "En promedio, ¿cuántas unidades se equivoca mi modelo?". Si predices precios de casas y el MAE es 5.000€, significa que tus predicciones suelen errar por esa cantidad.

2. Error Cuadrático Medio (MSE) 🟥

Similar al MAE, pero eleva los errores al cuadrado antes de promediarlos.

- **Fórmula:** $\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$
- **Importancia:** Al elevar al cuadrado, el MSE **penaliza mucho más los errores grandes**. Es muy útil si en tu problema un error pequeño es aceptable, pero uno grande es un desastre.

3. Coeficiente de Determinación (R^2) 📊

A diferencia de las anteriores, esta métrica no está en las unidades de los datos, sino que suele ir de 0 a 1 (o incluso valores negativos).

- **Interpretación:** Indica qué porcentaje de la variación de los datos es "explicado" por el modelo.
  - $R^2 = 1$**:** Predicción perfecta.
  - $R^2 = 0$**:** Tu modelo es tan útil como una línea horizontal que siempre predice el promedio.

Relación con la "Realidad" 🌎

Podemos resumir la relación de estas métricas con la precisión de la siguiente manera:

Métrica	Si las predicciones se ALEJAN 🚀 de la realidad...	Si las predicciones se ACERCAN 🎯 a la realidad...
MAE	El valor **sube** (mayor error promedio).	El valor **baja** (hacia 0).
MSE	El valor **sube drásticamente** (especialmente con valores atípicos).	El valor **baja** rápidamente.
R^2	El valor **baja** (se acerca a 0 o se vuelve negativo).	El valor **sube** (se acerca a 1).

Nota: No sé como formatear la tabla para:
asignar un 20% del ancho de página a la primera columna.
asignar un 40% del ancho de página a la segunda, y , a la tercera y última columna.
agregar saltos de líneas dentro de las celdas de forma que los paréntesis queden en la misma celda pero en diferente línea.
Hacer que se lea R^2 con ese formato de base y superíndice.
etc., etc.
He intentado hacerlo con Obsidian, Gemini, Hojas de Cálculo de Google y Word pero es un lío.
Además si le introduzco a Gemini una entrada en la que le intento especificar lo que quiero, le echa "imaginación"...
Me da que hay demasiada tela....

Cuándo usar **MAE** o **MSE**, frente a un error grande o **outlier** 🚀. Imagina que estamos entrenando un modelo para predecir el precio de 3 casas (en miles de euros).

1. Fórmulas y Escenario

Usaremos estos datos donde la tercera predicción es un "valor extraño" (se equivoca por mucho):

- **Valores reales (y):** $[200, 300, 400]$
- **Predicciones ($\hat{y}$):** $[210, 290, 460]$ (Errores: $+10, -10, +60$)

2. Cálculo de MAE (Error Absoluto Medio) 📏

El MAE trata a todos los errores por igual, de forma lineal.

$$MAE = \frac{1}{n}\sum_{i=1}^{n}|y_i - \hat{y}_i|$$

Sustitución:

$$MAE = \frac{|200-210| + |300-290| + |400-460|}{3} = \frac{10+10+60}{3} = \mathbf{26.6}$$

3. Cálculo de MSE (Error Cuadrático Medio) 🟥

El MSE eleva los errores al cuadrado, lo que amplifica los fallos grandes.

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$$

Sustitución:

$$MSE = \frac{(200-210)^2 + (300-290)^2 + (400-460)^2}{3} = \frac{100 + 100 + 3600}{3} = \mathbf{1266.6}$$

Razonamiento e Interpretación 🧠

- **Efecto del Outlier:** En el MAE, el error de "60" pesa lo que vale. Pero en el MSE, ese mismo error se convierte en **3600**. El MSE "castiga" con mucha más dureza las predicciones que se alejan mucho de la realidad.
- **¿Cuándo usar cuál?:**
- Usa **MAE** si quieres una medida robusta que no se vea alterada drásticamente por valores atípicos (es más "tolerante").
- Usa **MSE** si en tu proyecto un error grande es inaceptable y quieres que el modelo aprenda a evitarlos a toda costa.

Caso práctico: **Sistema de frenado de un coche autónomo** 🚗.
Si el sistema calcula mal la distancia por un margen pequeño, no pasa mucho, pero si se equivoca por mucho, el riesgo es total.

Coeficiente de Determinación o R^2 A diferencia del MAE o el MSE que nos dan errores en unidades específicas, el R^2 nos da una medida relativa de la calidad del modelo. 📊

1. ¿Qué es el R^2 y cuál es su fórmula? 🧐

El R^2 mide qué proporción de la **varianza** total de los datos es explicada por nuestro modelo. Es como comparar nuestro modelo contra un "modelo base" muy simple: uno que siempre predice la **media** ($\bar{y}$) de los datos.

La fórmula es:

$$R^2 = 1 - \frac{SS_{res}}{SS_{tot}}$$

Donde:

- SS_{res} **(Suma de Cuadrados de los Residuos):** Es la suma de los errores de nuestro modelo al cuadrado. $\sum(y_i - \hat{y}_i)^2$. Fíjate que esto es básicamente el **MSE** multiplicado por el número de datos.
- SS_{tot} **(Suma de Cuadrados Totales):** Es la varianza total de los datos respecto a su media. $\sum(y_i - \bar{y})^2$. Representa el error que cometeríamos si no tuviéramos modelo y solo usáramos el promedio.

2. Relación con la "Inteligencia" del Modelo 🧠

El valor de R^2 nos dice cuánta información útil ha capturado el modelo frente a simplemente "adivinar" el promedio.

Valor de R2	Interpretación de la "Inteligencia"	Significado técnico
Cercano a 1	**Modelo Brillante** ✨	El modelo explica casi toda la variabilidad. Las predicciones están muy cerca de los valores reales.
Cercano a 0	**Modelo "Perezoso"** 💤	El modelo no es mejor que predecir siempre la media. No ha aprendido ninguna relación útil entre las variables.
Negativo	**Modelo "Dañino"** ⚠️	¡Increíble pero posible! Significa que el modelo es **peor** que predecir la media. Sus predicciones están tan alejadas que confunden más de lo que ayudan.

3. Ejemplo Numérico y el efecto de los Outliers 🚀

Tenemos 3 datos de ventas: **10, 20, 30**. La media ($\bar{y}$) es **20**.
Calculamos la variabilidad total (SS_{tot}):

$$(10-20)^2 + (20-20)^2 + (30-20)^2 = 100 + 0 + 100 = \mathbf{200}$$

Caso A: Modelo Bueno

Predicciones: **11, 19, 31**.
$SS_{res} = (10-11)^2 + (20-19)^2 + (30-31)^2 = 1 + 1 + 1 = \mathbf{3}$

$$R^2 = 1 - \frac{3}{200} = \mathbf{0.985}$$

Caso B: Modelo con Outlier (Valor extraño)

Supongamos que el modelo predice bien los dos primeros, pero en el tercero falla mucho por un outlier: **11, 19, 60**.
$SS_{res} = (10-11)^2 + (20-19)^2 + (30-60)^2 = 1 + 1 + 900 = \mathbf{902}$

$$R^2 = 1 - \frac{902}{200} = \mathbf{-3.51}$$

Razonamiento: El outlier disparó el error al cuadrado (SS_{res}), haciendo que el numerador sea mucho mayor que el denominador. Como resultado, el R^2 se vuelve negativo, indicando que el modelo ha "perdido el norte".

1. Varianza Total (SS_{tot}) 📉

La **Suma de Cuadrados Totales** (SS_{tot}) representa la variabilidad natural de tus datos. Es el error que cometerías si no usaras ningún modelo y simplemente predijeras siempre la

media ($\bar{y}$).
Fórmula:

$$SS_{tot} = \sum_{i=1}^{n} (y_i - \bar{y})^2$$

Ejemplo Numérico:
Imagina que tenemos estos precios de casas (en miles): $y = [100, 200, 300]$.

1. Calculamos la media: $\bar{y} = \frac{100+200+300}{3} = 200$.
2. Sustituimos en la fórmula:

$$SS_{tot} = (100 - 200)^2 + (200 - 200)^2 + (300 - 200)^2$$

$$SS_{tot} = (-100)^2 + 0^2 + 100^2 = 10,000 + 0 + 10,000 = \mathbf{20,000}$$

2. Interpretación Combinada: R^2 + MAE 🔍

Escenario	R2	MAE	Conclusión
Error Generalizado	Bajo	Alto	El modelo no entiende la tendencia general de los datos. ❌
Presencia de Outliers	Alto	Bajo	El modelo predice bien la mayoría de casos, pero falla estrepitosamente en unos pocos. 🚀

¿Por qué sucede esto? El MAE no se "asusta" con los errores grandes (outliers), pero el R^2 (que usa errores al cuadrado) cae rápidamente cuando hay una predicción muy lejana.

3. Limitaciones y R^2 Ajustado ⚖️

El gran peligro del **Overfitting** (sobreajuste) con el R^2 es que, si añades más variables a tu modelo (aunque sean ruido o datos sin sentido), el R^2 **nunca bajará**. Esto da una falsa sensación de mejora.

Para corregir esto, usamos el R^2 **Ajustado**:

- **¿Qué hace?** Penaliza al modelo por cada variable adicional que no aporta valor real. 📉
- **¿Por qué se usa?** Para saber si una variable nueva realmente mejora la predicción o si solo estamos "memorizando" el ruido.

El R^2 **ajustado** es una versión modificada del coeficiente de determinación que introduce una "penalización" por la complejidad del modelo. A diferencia del R^2 convencional, que siempre aumenta (o se mantiene igual) al añadir nuevas variables —aunque estas sean puro ruido—, el R^2 ajustado solo sube si la nueva variable mejora el modelo más de lo que se esperaría por puro azar.

¿Por qué se usa con muchas variables? ⚖️

En ciencia de datos, existe el riesgo de caer en el **overfitting** (sobreajuste). Si añadimos suficientes variables (por ejemplo, el signo del zodíaco del vendedor para predecir el precio de una casa), el R^2 normal subirá ligeramente por pura coincidencia matemática. El R^2 ajustado detecta este "engaño" y reduce su valor para reflejar que el modelo se está volviendo innecesariamente complejo sin ganar precisión real.

Fórmula Matemática 🧮

$$\bar{R}^2 = 1 - \left[\frac{(1 - R^2)(n - 1)}{n - k - 1} \right]$$

Donde:

- R^2: Coeficiente de determinación normal.
- n: Número de observaciones (tamaño de la muestra).
- k: Número de variables predictoras (características).

Ejemplo Numérico y Sustitución 🔢

Estamos prediciendo el precio de unas viviendas con un dataset pequeño:

- **Muestra (n):** 20 casas.
- **Modelo A (2 variables):** $k = 2$, $R^2 = 0.80$.
- **Modelo B (Añadimos 10 variables inútiles):** $k = 12$, R^2 sube a 0.82 por azar.

Sustitución para el Modelo A:

$$\bar{R}^2 = 1 - \left[\frac{(1 - 0.80)(20 - 1)}{20 - 2 - 1} \right] = 1 - \left[\frac{0.20 \cdot 19}{17} \right] = 1 - 0.223 \approx \mathbf{0.777}$$

Sustitución para el Modelo B:

$$\bar{R}^2 = 1 - \left[\frac{(1 - 0.82)(20 - 1)}{20 - 12 - 1} \right] = 1 - \left[\frac{0.18 \cdot 19}{7} \right] = 1 - 0.488 \approx \mathbf{0.512}$$

Interpretación de los resultados 🧠

Aunque el R^2 del Modelo B es mayor (0.82 frente a 0.80), su R^2 **ajustado se desploma** de 0.777 a 0.512.

Esto nos indica que las 10 variables extra no están aportando valor real; al contrario, están "inflando" artificialmente la métrica de rendimiento y restando fiabilidad al modelo. El R^2 ajustado nos dice que el Modelo A es, en realidad, mucho más "inteligente" y robusto.

Concepto Teórico	Siglas	Función en sklearn.metrics
Error Absoluto Medio	**MAE**	`mean_absolute_error`
Error Cuadrático Medio	**MSE**	`mean_squared_error`

Concepto Teórico	Siglas	Función en sklearn.metrics
Raíz del Error Cuadrático Medio	**RMSE**	`root_mean_squared_error` (o `mean_squared_error` con `squared=False`)
Coeficiente de Determinación	R^2	`r2_score`
Coeficiente de Determinación Ajustado	$\bar{R}^2$	*(No existe función directa, se calcula manualmente)*

Notas 📝

RMSE: Es muy común incluirlo porque, al ser la raíz cuadrada del MSE, devuelve el error a las **unidades originales** de los datos (como euros 💶 o metros 📏), pero manteniendo la penalización a los errores grandes.
R^2 **Ajustado:** En el ecosistema de Machine Learning con `scikit-learn`, se prioriza el rendimiento predictivo. El R^2 ajustado es más frecuente en el análisis estadístico clásico (donde se usa la librería `statsmodels`), por lo que en Python se suele programar la fórmula usando los resultados de `r2_score`.

Ejemplo de script con combinación de cálculo y visualización. Código en **Python 3** que:

1. Genera datos sintéticos (simulando precios de casas según su tamaño).
2. Entrena un modelo simple.
3. Calcula todas las métricas solicitadas.
4. Genera un gráfico explicativo donde se ven los "residuos" (las líneas rojas que representan el error).
   Copiar y pegar este bloque Ejemplo en Notebook (Jupyter/Colab).

Código Python: Métricas de Regresión 📉

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_absolute_error, mean_squared_error,
r2_score

# --- 1. GENERACIÓN DE DATOS (Dataset Inline) ---
# Simulamos datos: X = Tamaño (m2), y = Precio (miles de €)
np.random.seed(42) # Semilla para reproducibilidad
X = 2 * np.random.rand(20, 1) # 20 casas aleatorias
y = 4 + 3 * X + np.random.randn(20, 1) # Precio con un poco de "ruido"
(varianza natural)

# --- 2. ENTRENAMIENTO DEL MODELO ---
model = LinearRegression()
model.fit(X, y)
```

```
y_pred = model.predict(X) # Predicciones del modelo

# --- 3. CÁLCULO DE MÉTRICAS ---

# A) MAE (Error Absoluto Medio)
# Promedio de la diferencia absoluta.
mae = mean_absolute_error(y, y_pred)

# B) MSE (Error Cuadrático Medio)
# Promedio de los errores al cuadrado (penaliza mucho los fallos grandes).
mse = mean_squared_error(y, y_pred)

# C) RMSE (Raíz del Error Cuadrático Medio)
# Devuelve el error a las unidades originales (miles de €).
# Nota: En sklearn versiones nuevas existe root_mean_squared_error,
# pero la forma más compatible es la raíz del MSE.
rmse = np.sqrt(mse)

# D) R2 (Coeficiente de Determinación)
# Qué porcentaje de la varianza explicamos (0 a 1).
r2 = r2_score(y, y_pred)

# E) R2 Ajustado (Cálculo manual)
# Penaliza si añadimos variables inútiles.
n = len(y)        # Número de muestras (20)
p = X.shape[1]    # Número de variables predictoras (1)
r2_ajustado = 1 - (1 - r2) * (n - 1) / (n - p - 1)

# --- 4. IMPRESIÓN DE RESULTADOS ---
print(f"--- 📊 REPORTE DE MÉTRICAS DE REGRESIÓN ---")
print(f"MAE (Error Absoluto):      {mae:.4f} (El error promedio es de
{mae:.2f} miles de €)")
print(f"MSE (Error Cuadrático):    {mse:.4f} (Difícil de interpretar
directamente)")
print(f"RMSE (Raíz del MSE):       {rmse:.4f} (El error estándar es de
{rmse:.2f} miles de €)")
print(f"R² (Score):                {r2:.4f} (El modelo explica el
{r2*100:.1f}% de la varianza)")
print(f"R² Ajustado:               {r2_ajustado:.4f} (Ajuste por complejidad
del modelo)")

# --- 5. VISUALIZACIÓN GRÁFICA ---
plt.figure(figsize=(10, 6))

# a) Dibujar los datos reales
plt.scatter(X, y, color='blue', label='Datos Reales (y)')

# b) Dibujar la línea de regresión (predicción)
plt.plot(X, y_pred, color='green', linewidth=2, label='Modelo / Predicción
(ŷ)')
```

```
# c) Dibujar los ERRORES (Residuos)
# Estas líneas rojas son lo que miden MAE y MSE
for i in range(len(X)):
    plt.plot([X[i], X[i]], [y[i], y_pred[i]], color='red', linestyle='--',
alpha=0.5)

# Decoración del gráfico
plt.title('Regresión Lineal: Visualizando los Errores (Residuos)',
fontsize=14)
plt.xlabel('Tamaño de la casa (X)')
plt.ylabel('Precio (y)')
plt.legend()
plt.grid(True, alpha=0.3)

# Mostrar solo la primera leyenda de "Error" para no saturar
plt.plot([], [], color='red', linestyle='--', label='Error (Residuo)')
plt.legend()

plt.show()
```

Interpretación de este código 🧐

1. **Consola:** Verán los números exactos. Fíjate en la diferencia entre el **MSE** (que sale alto, por ejemplo 0.6 o 0.8) y el **RMSE** (que baja a 0.8 o 0.9). Esto ilustra por qué el RMSE es más fácil de "leer" (está en euros, no en "euros cuadrados").
2. **Gráfico:**

- Los **puntos azules** son la realidad.
  - La **línea verde** es el modelo.
  - Las **líneas rojas discontinuas** son clave: representan la distancia $y - \hat{y}$.
  - El **MAE** es el promedio de la longitud de esas líneas rojas.
  - El **MSE** es el promedio de la longitud de esas líneas elevadas al cuadrado (dando más peso a las líneas largas).

Adición de un **outlier** artificial al código (un punto muy lejano) para que vean en directo cómo el R^2 se desploma y el MSE se dispara.

Un solo punto puede destruir un modelo lo que refleja la importancia de limpiar los datos.

Código listo para copiar y pegar en una **nueva celda** de Google Colab o Jupyter Notebook. Se asume que ya se ha ejecutado el código anterior (variables `X`, `y` y `model` originales).

Código: El Efecto del Outlier (Destruyendo el Modelo) 💥

```
# --- 6. EL EXPERIMENTO DEL OUTLIER (Añadimos un dato "tóxico") ---
```

```
# 1. Crear el Outlier
# Añadimos un punto en X=3 (lejos) con valor Y=-5 (muy negativo, rompiendo
la tendencia positiva)
X_outlier = np.vstack([X, [[3.0]]])
y_outlier = np.vstack([y, [[-5.0]]])

# 2. Entrenar un NUEVO modelo con el dato contaminado
model_bad = LinearRegression()
model_bad.fit(X_outlier, y_outlier)
y_pred_bad = model_bad.predict(X_outlier)

# 3. Recalcular métricas para ver el desastre
mse_bad = mean_squared_error(y_outlier, y_pred_bad)
r2_bad = r2_score(y_outlier, y_pred_bad)

# --- 4. COMPARATIVA DE IMPACTO ---
print(f"--- 🚨 IMPACTO DEL OUTLIER 🚨 ---")
print(f"MSE Original: {mse:.2f}  --->  MSE con Outlier: {mse_bad:.2f} (¡Se
ha disparado!)")
print(f"R² Original:  {r2:.2f}  --->  R² con Outlier:  {r2_bad:.2f} (El
modelo ha empeorado drásticamente)")

# --- 5. VISUALIZACIÓN DEL DAÑO ---
plt.figure(figsize=(10, 6))

# a) Datos originales (Azul) y el Outlier (Rojo Gigante)
plt.scatter(X, y, color='blue', alpha=0.5, label='Datos Normales')
plt.scatter([3.0], [-5.0], color='red', s=200, marker='X', label='OUTLIER
(Dato Extraño)')

# b) Línea del modelo ORIGINAL (Punteada Verde) - Lo que debería ser
X_range = np.linspace(0, 3.5, 100).reshape(-1, 1)
plt.plot(X_range, model.predict(X_range), color='green', linestyle='--',
linewidth=2, label='Modelo Original (Sin Outlier)')

# c) Línea del modelo AFECTADO (Sólida Roja) - Cómo el outlier "tira" de la
línea
plt.plot(X_range, model_bad.predict(X_range), color='red', linewidth=3,
label='Modelo Afectado (Sesgado)')

# Decoración
plt.title('Cómo un solo Outlier "rompe" la Regresión y dispara el MSE',
fontsize=14)
plt.xlabel('Tamaño (X)')
plt.ylabel('Precio (y)')
plt.legend()
plt.grid(True, alpha=0.3)

# Mostrar la distancia del error del outlier (línea vertical negra)
plt.plot([3.0, 3.0], [-5.0, model_bad.predict([[3.0]])[0][0]],
```

```
color='black', linestyle=':', label='Error del Outlier')

plt.show()
```

Salida esperada 👀

1. **Consola: MSE** se multiplica (quizás pase de 0.8 a 15.0 o más) debido a que ese único error se eleva al cuadrado. El R^2 probablemente caiga en picado (incluso podría volverse negativo o muy cercano a 0), indicando que el modelo ya no explica bien la varianza general.
2. **Gráfico:**
   - Línea verde (el modelo bueno) siguiendo a la mayoría de los puntos.
   - **Línea roja** (el modelo nuevo) inclinada hacia abajo a la derecha, "secuestrada" por el punto rojo (la X grande).
   - Esto demuestra gráficamente que el MSE (y la regresión lineal simple) es muy sensible a los valores atípicos.

Estrategias de Validación. 🛡️

En el contexto de Machine Learning con Python, la "validación" no se refiere a comprobar si un dato es un número o una letra (eso es limpieza de datos). Aquí, **validar** significa someter al modelo a un examen para asegurarnos de que funcionará bien en el mundo real y que no ha memorizado las respuestas (overfitting).

1. Train/Test Split (Dividir en Entrenamiento y Prueba) ✂️

Es la estrategia más sencilla y rápida. Consiste en dividir el conjunto de datos.

- **Training Set (Entrenamiento):** Suele ser el 70-80% de los datos. Es el "libro de texto" con el que el modelo estudia.
- **Test Set (Prueba):** Es el 20-30% restante. Es el "examen final". El modelo **nunca** ve estos datos durante el entrenamiento. Solo los usamos al final para evaluarlo.

El riesgo: Depende del azar, si el "Test" contiene solo casos fáciles o si contiene solo casos difíciles, se puede producir una falsa sensación de ajuste o desajuste.

2. Validación Cruzada (K-Fold Cross Validation) 🔄

Esta es una estrategia más robusta y "democrática". En lugar de hacer un solo examen, hacemos K exámenes diferentes.

1. Dividimos los datos en K **partes iguales** (o "folds"). Por ejemplo, $K = 5$.
2. Iteración 1: Usamos la parte 1 para probar y las otras 4 para entrenar.
3. Iteración 2: Usamos la parte 2 para probar y las otras 4 para entrenar.
4. ... Repetimos hasta usar todas las partes como prueba una vez.

5. **Resultado Final:** Promediamos las 5 notas obtenidas.

La ventaja: Eliminamos el factor suerte. Todos los datos son usados para entrenar y para probar en algún momento. El resultado es mucho más fiable.

Tabla Comparativa: Train/Test vs. K-Fold 🥊

Tabla resumen:

Característica	Train/Test Split	K-Fold Cross Validation
Velocidad ⚡	**Muy rápido** (se entrena 1 vez).	**Lento** (se entrena K veces).
Fiabilidad 🛡️	Baja (depende de cómo caiga el corte).	**Alta** (el promedio suaviza la suerte).
Uso de datos 💾	Desperdicia datos (el Test set nunca se usa para aprender).	Eficiente (todos los datos sirven para aprender en algún momento).
Cuándo usarlo	Datasets muy grandes (donde K-Fold sería eterno).	Datasets pequeños o medianos (donde cada dato cuenta).

Ejemplo en Python 3 🐍

Este código ilustra ambas técnicas usando `scikit-learn`.

```
import numpy as np
from sklearn.datasets import make_classification
from sklearn.model_selection import train_test_split, cross_val_score
from sklearn.linear_model import LogisticRegression

# 1. Generamos datos de ejemplo (100 filas)
X, y = make_classification(n_samples=100, random_state=42)

# --- ESTRATEGIA A: TRAIN/TEST SPLIT ---
# Dividimos una sola vez: 80% estudiar, 20% examen
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42)

model_a = LogisticRegression()
model_a.fit(X_train, y_train)
score_a = model_a.score(X_test, y_test)

print(f"--- Estrategia Train/Test ---")
print(f"Nota del examen único: {score_a:.2f} (Puede ser suerte)")

# --- ESTRATEGIA B: CROSS VALIDATION (K-Fold) ---
# K=5: Haremos 5 exámenes diferentes
model_b = LogisticRegression()
# cross_val_score hace todo el trabajo sucio por nosotros
```

```
scores_b = cross_val_score(model_b, X, y, cv=5)

print(f"\n--- Estrategia K-Fold (K=5) ---")
print(f"Notas de los 5 exámenes: {scores_b}")
print(f"Nota PROMEDIO real: {scores_b.mean():.2f} (Más fiable)")
```

Conclusiones 🎓

Si existe la posibilidad y el dataset no es gigantesco (millones de datos), **siempre** intentar usar **Cross Validation** para asegurar un modelo estable.
El *Train/Test Split* prototipar rápido.
K-Fold para validar con mayor fiabilidad.

¿Cómo guardar el modelo una vez validado?

Se ha entrenando y validando el modelo (horas o días). Ha aprendido patrones complejos y tiene un **Accuracy** o un R^2 increíble. Si se cierra el cuaderno de Python (Jupyter/Colab), **toda esa "inteligencia" se pierde** en la memoria RAM y sería necesario re-entrenar desde cero la próxima vez.

Para evitar esto, usamos la **Serialización**. Es el proceso de guardar el modelo en un archivo (como si fuera un documento de Word o una partida guardada de un videojuego 🎮) para poder usarlo después en producción o en otra aplicación.

En **Python 3** y `scikit-learn`, la herramienta estándar y más eficiente es `joblib`.

Código: Guardar y Cargar (receta final) 💾

Se muestra cómo guardar el modelo entrenado de la IA en un archivo `.pkl` y cómo recargar más tarde.

```
import joblib
from sklearn.linear_model import LinearRegression

# 1. Supongamos que este es tu modelo YA ENTRENADO y VALIDADO
# (Usamos un ejemplo simple)
modelo = LinearRegression()
X = [[1], [2], [3]]
y = [2, 4, 6]
modelo.fit(X, y)

print("✅ Modelo entrenado. Predicción para 5: ", modelo.predict([[5]]))

# --- PASO A: GUARDAR EL MODELO (Serialización) ---
# Usamos joblib.dump(objeto, 'nombre_archivo.pkl')
filename = 'mi_super_modelo_v1.pkl'
joblib.dump(modelo, filename)
```

```
print(f"💾 El modelo se ha guardado exitosamente en '{filename}'")
print("... Simulamos que cerramos el programa y pasa el tiempo ...\n")

# --- PASO B: CARGAR EL MODELO (Deserialización) ---
# Ahora puedes estar en otro script, otro día, o en un servidor web.
# No necesitas tener los datos de entrenamiento (X, y) originales, solo el
archivo.

modelo_cargado = joblib.load(filename)

print("📂 Modelo cargado desde el disco.")

# --- PASO C: USARLO EN PRODUCCIÓN ---
# El modelo cargado recuerda todo lo que aprendió.
nueva_prediccion = modelo_cargado.predict([[5]])
print(f"🔮 Predicción del modelo cargado para 5: {nueva_prediccion}")
```

Resumen Final

1. **Clasificación:** *Accuracy* puede ser confuso, mejor *Matriz de Confusión*, *F1-Score* y la *Curva ROC*.
2. **Regresión:** *MAE* (robusto) y *MSE* (sensible a outliers), *R^2* para medir el ajuste frente al promedio.
3. **Validación:** *Train/Test Split* es rápido pero arriesgado, y *Cross Validation (K-Fold)* es la mejor opción si los medios lo permiten.
4. **Persistencia:** `joblib` permite guardar el modelo.