# VISUALIZACIÓN, EDA Y PIPELINE BÁSICO DE MACHINE LEARNING

## 0. Enfoque general de la sesión

Hasta ahora has aprendido a:

- trabajar con datos en Python
- entender estructuras como arrays y DataFrames
- manipular datos **ya disponibles en el entorno de trabajo**

En esta sesión damos un paso clave:

> Aprender a leer los datos antes de modelar.

Por primera vez, los datos con los que trabajaremos:

- no estarán “dados por hecho”
- no estarán limpios
- no estarán pensados para el modelo

Esta sesión no trata de “hacer gráficos bonitos” ni de “probar modelos al azar”.

Trata de **aprender a pensar como alguien que va a construir un modelo de Machine Learning**.

Aquí aprenderás a:

- cargar y comprender datasets reales
- usar la visualización como herramienta de análisis
- detectar problemas reales en los datos
- preparar los datos con intención
- construir un primer modelo *baseline*
- evaluar resultados con criterio

Esta sesión conecta directamente:

- con lo que has visto en datos y Python

- y con lo que verás después en modelos supervisados y redes neuronales

Por eso es una sesión **central** del curso.

---

# 1. DATASETS REALES: CÓMO ENTRAN EN PYTHON (Y QUÉ ES `df`)

En la sesión anterior trabajaste con datos **ya disponibles en memoria** (arrays, DataFrames creados en el propio código o dados por el entorno).

En un proyecto real, casi siempre ocurre lo contrario:

> Los datos existen fuera del programa (archivos, bases de datos, APIs) y primero hay que traerlos a Python.

En esta sesión vamos a trabajar con un archivo CSV. Lo importante aquí no es "aprender CSV", sino entender el **acto fundacional**:

- **cargar un dataset**
- **obtener un DataFrame**
- **acceder a variables (columnas)**
- **verificar rápidamente que lo cargado tiene sentido**

---

## 1.1 Qué es un dataset (en términos prácticos)

Un *dataset* es un conjunto estructurado de datos. En la práctica, lo vas a encontrar como:

- un archivo (`.csv`, `.xlsx`, `.parquet`, etc.)
- con **filas** y **columnas**

Interpretación mínima (útil para Machine Learning):

- **cada fila** suele representar una observación (un cliente, un evento, una transacción...)
- **cada columna** suele representar una variable (edad, plan, uso, estado...)

Esto es importante porque casi todo lo que haremos después (visualización, preparación, modelos) se basa en:

- elegir variables

- entender distribuciones
- analizar diferencias entre grupos
- medir relaciones

---

## 1.2 CSV como formato de intercambio (qué es y qué no es)

Un archivo **CSV** (*Comma-Separated Values*) es un formato de texto que representa tablas.

Características típicas:

- primera fila: nombres de columnas
- filas siguientes: valores
- separador frecuente: coma `,` (a veces `;`, depende del origen)

Lo que **sí** debes asumir:

- es un formato común para intercambiar datos
- se puede abrir con Excel, Google Sheets o un editor de texto
- puede contener errores de tipos (números leídos como texto), fechas ambiguas, etc.

Lo que **no** vamos a hacer aquí (todavía):

- gestionar problemas complejos de `encoding`
- discutir delimitadores raros o ficheros enormes
- "limpiar" el dataset al cargarlo

La meta de este bloque es únicamente:

> cargar y verificar.

---

## 1.3 Cargar un dataset en pandas: `pd.read_csv`

La forma estándar de cargar un CSV en pandas es:

```
import pandas as pd

df = pd.read_csv("datos_ejercicios_sesion3.csv")
```

Idea clave:

- `pd.read_csv(...)` **lee el archivo**
- y devuelve un objeto llamado **DataFrame**
- lo guardamos en una variable (por convención, `df`)

**Nota práctica sobre rutas (sin complicarlas):**

- Si el archivo está en la **misma carpeta** que tu script, basta el nombre.
- Si está en otra carpeta, necesitas la ruta relativa, por ejemplo:

```
df = pd.read_csv("data/datos_ejercicios_sesion3.csv")
```

---

## 1.4 Qué es `df` y por qué se llama así

`df` no es una palabra reservada. Es una convención:

- `df` = **dataframe**

Podrías llamarlo `clientes`, `datos`, `tabla`, etc. pero `df` es frecuente y te lo encontrarás constantemente.

Lo importante es el significado conceptual:

> df es el dataset cargado en memoria, en forma tabular (filas/columnas).

A partir de ahora, cuando veas:

```
df
```

debes pensar:

- "tabla de datos"
- "dataset ya disponible para análisis en Python"

---

## 1.5 Acceder a columnas: `df["columna"]` y qué obtienes

Cada columna del DataFrame es una variable del dataset.

Acceder a una columna se hace así:

```
df["tenure_days"]
```

Eso devuelve una estructura llamada **Series** (una columna con un valor por fila).

Dos comprobaciones útiles (para fijar concepto):

```
type(df)              # DataFrame
type(df["tenure_days"])  # Series
```

**Por qué esto importa:**

- casi todos los análisis univariantes empiezan con “una columna”
- casi todos los gráficos empiezan con “una columna” o “dos columnas”
- la preparación para modelado empieza con “selección de columnas”

---

## 1.6 Verificaciones mínimas obligatorias tras cargar (antes de analizar)

Antes de dibujar una sola gráfica o calcular nada, hay un protocolo mínimo.

Estas comprobaciones no son burocracia: evitan errores de base.

### a) Tamaño del dataset

```
df.shape
```

Interpretación:

- número de filas (observaciones)
- número de columnas (variables)

### b) Vista rápida de filas

```
df.head(5)
```

Objetivo:

- comprobar que los datos “tienen sentido”
- detectar nombres raros de columnas

- ver si hay valores evidentemente anómalos (sentinelas, textos donde esperabas números, etc.)

### c) Nombres de columnas

```
df.columns
```

Esto te permite:

- saber qué variables existen
- referirte a ellas sin “inventar nombres”
- detectar espacios o nombres inconsistentes

### d) Tipos de datos inferidos por pandas

```
df.dtypes
```

Esto es crítico: al cargar, pandas “adivina” tipos y a veces se equivoca.

Señales típicas de problema:

- algo que debería ser numérico aparece como `object`
- fechas aparecen como `object` (texto)
- códigos numéricos son en realidad categorías (p.ej., `region_id`)

### e) Valores ausentes (visión global)

```
df.isna().sum().sort_values(ascending=False)
```

Esto te dice:

- qué columnas tienen `NaN`
- cuántos
- y te orienta sobre si el problema es marginal o estructural

---

## Cierre del bloque 1

En este bloque has establecido el punto de partida realista del curso:

- qué significa trabajar con un dataset externo

- cómo se carga un CSV con pandas
- qué es un DataFrame (`df`) y cómo se accede a una columna
- qué verificaciones mínimas debes hacer **antes** de cualquier análisis

A partir de aquí, ya tiene sentido hablar de visualización y análisis univariante, porque ya no estamos trabajando "con datos abstractos", sino con un dataset real cargado en memoria.

---

### Referencias específicas del bloque 1

- Pandas — `read_csv` (parámetros, comportamiento, ejemplos)
  https://pandas.pydata.org/docs/reference/api/pandas.read_csv.html
- Pandas — Tutorial: DataFrame como estructura tabular (tabla orientada a datos)
  https://pandas.pydata.org/docs/getting_started/intro_tutorials/01_table_oriented.html
- Pandas — Tipos de datos (`dtypes`) y fundamentos operativos
  https://pandas.pydata.org/docs/user_guide/basics.html#dtypes

---

## 2. Visualización de datos como herramienta de análisis (no de presentación)

Antes de entrenar cualquier modelo, es imprescindible responder a una pregunta básica:

> ¿Qué tipo de datos tengo delante y qué estructura presentan?

La visualización es la herramienta principal para responder a esa pregunta.

### 2.1 Visualizar no es "dibujar": es analizar

En este curso, la visualización se entiende como:

- una herramienta para **detectar patrones**
- una forma de **validar suposiciones**
- un medio para **descubrir problemas ocultos**
- una ayuda para **tomar decisiones técnicas**

No se utiliza para:

- embellecer resultados
- comunicar a terceros
- justificar conclusiones ya tomadas

Eso vendrá más adelante. Aquí la visualización es **una herramienta de trabajo**.

---

## 2.2 Qué preguntas debe responder una visualización

Una buena visualización debe ayudarte a responder preguntas como:

- ¿Cómo se distribuyen los valores de una variable?
- ¿Hay valores extremos o anómalos?
- ¿Existen relaciones aparentes entre variables?
- ¿La relación es clara o está dominada por ruido?
- ¿Hay grupos o patrones ocultos?

Si un gráfico no ayuda a responder **ninguna** de estas preguntas, probablemente no es útil para el análisis.

---

## 2.3 Tipos de variables y visualización (introducción conceptual)

Antes de elegir un gráfico, debes tener claro **qué tipo de variable estás analizando**:

- **Variables numéricas continuas**

  (edad, ingresos, tiempo, duración)
- **Variables numéricas discretas**

  (número de eventos, conteos)
- **Variables categóricas**

  (país, tipo de usuario, plan)
- **Variables binarias**

  (sí/no, 0/1)

Cada tipo de variable **admite mejor unos gráficos que otros**.

Elegir mal el gráfico puede ocultar información o inducir a errores.

Este punto se desarrollará progresivamente en los siguientes bloques.

---

## 2.4 Advertencia temprana: visualizar no es concluir

Un principio fundamental que debes interiorizar desde ahora:

> Ver un patrón no implica entender su causa.

La visualización permite:

- observar relaciones aparentes
- generar hipótesis
- detectar irregularidades

Pero **no permite**:

- afirmar causalidad
- garantizar utilidad predictiva
- decidir automáticamente qué variables "sirven"

Este aviso se repetirá a lo largo de la sesión, porque es uno de los errores más frecuentes en proyectos reales.

---

## Referencias específicas del bloque 2

- Python Data Science Handbook — Exploratory Data Analysis
  https://jakevdp.github.io/PythonDataScienceHandbook/
- Matplotlib — Introduction and basic concepts
  https://matplotlib.org/stable/tutorials/introductory/usage.html
- Seaborn — Data visualization and exploratory analysis
  https://seaborn.pydata.org/tutorial.html

---

## Cierre del bloque 2

En este bloque has aprendido que:

- la visualización es una herramienta de análisis, no de presentación
- su objetivo es entender la estructura del dato
- un gráfico debe responder a una pregunta concreta
- observar no equivale a concluir

En el siguiente bloque empezarás a ver **cómo se visualizan variables individuales** y qué tipo de información puedes extraer de ellas antes de pensar en relaciones o modelos.

---

# 3. Análisis univariante: entender cada variable por separado

Antes de analizar relaciones entre variables o entrenar modelos, es imprescindible **entender cada variable de forma aislada**.

Este paso se denomina **análisis univariante** y es una parte esencial del análisis exploratorio de datos.

La idea central es simple:

> No puedes interpretar una relación entre variables si no entiendes primero cómo se comporta cada una por separado.

---

## 3.1 Qué es el análisis univariante y para qué sirve

El análisis univariante se centra en **una sola variable cada vez** y permite responder preguntas como:

- ¿Cómo se distribuyen sus valores?
- ¿Cuál es su rango típico?
- ¿Existen valores extremos?
- ¿Hay asimetría o colas largas?
- ¿Hay valores ausentes o anómalos?

Este análisis no busca aún relaciones ni predicción.

Busca **comprensión básica y detección temprana de problemas**.

---

## 3.2 Variables numéricas: forma de la distribución

Para variables numéricas, el primer objetivo es entender la **forma de la distribución**.

Aspectos clave a observar:

- **Centro**: dónde se concentran los valores
- **Dispersión**: qué tan separados están

- **Asimetría**: si la distribución está sesgada hacia un lado
- **Multimodalidad**: si hay varios “picos” (posibles subgrupos)

Los gráficos más habituales para este análisis son:

- histogramas
- curvas de densidad
- boxplots (que se introducirán después)

En este punto, no importa memorizar gráficos, sino **saber qué información extraer de ellos**.

---

## 3.3 Asimetría y colas largas: un patrón muy común

En datos reales, muchas variables no siguen distribuciones “bonitas” o simétricas.

Ejemplos habituales:

- ingresos
- duración de sesiones
- número de eventos
- tiempos de espera

Estas variables suelen presentar:

- **cola larga a la derecha**
- muchos valores pequeños
- pocos valores muy grandes

Esto es normal y **no es un error del dataset**.

Pero tiene implicaciones importantes más adelante:

- algunos modelos asumen cierta regularidad en los datos
- la presencia de colas largas puede afectar a medias y varianzas
- la interpretación de valores extremos debe hacerse con cuidado

En esta fase **no se corrige nada**: solo se observa y se anota.

---

## 3.4 Outliers: detectar no es eliminar

Un **outlier** es un valor que se aleja claramente del patrón general de la variable.

Es fundamental no cometer este error conceptual:

> Outlier no significa “dato incorrecto”.

Un outlier puede ser:

- un error de medición
- un caso extremo pero real
- una señal importante del fenómeno estudiado

Por eso, en el análisis univariante:

- se **detectan**
- se **documentan**
- pero **no se eliminan automáticamente**

La decisión de tratarlos (o no) se tomará más adelante, con más contexto.

---

## 3.5 Variables discretas y conteos

Algunas variables numéricas representan **conteos**:

- número de accesos
- número de incidencias
- número de acciones realizadas

Estas variables suelen tener características propias:

- muchos valores bajos (incluso ceros)
- algunos valores altos
- distribución irregular

En estos casos, es importante observar:

- si hay concentración en pocos valores
- si existe una masa grande de ceros
- si aparecen valores inesperadamente altos

Esto puede indicar:

- distintos tipos de comportamiento

- usuarios inactivos vs muy activos
- posibles segmentos ocultos

## 3.6 Variables categóricas: frecuencia y equilibrio

Para variables categóricas, el análisis univariante se centra en:

- número de categorías
- frecuencia de cada una
- equilibrio o desequilibrio entre categorías

Preguntas clave:

- ¿Hay categorías muy dominantes?
- ¿Existen categorías muy raras?
- ¿Hay valores mal codificados o inconsistentes?

Un gran desequilibrio en categorías puede tener impacto en:

- visualización posterior
- codificación para modelos
- estabilidad de los resultados

## 3.7 Valores ausentes: detectar antes de modelar

Una parte crítica del análisis univariante es **detectar valores ausentes**.

Aspectos a observar:

- qué variables tienen valores faltantes
- en qué proporción
- si la ausencia parece marginal o relevante

En esta fase:

- no se imputan valores
- no se eliminan filas
- no se toman decisiones automáticas

El objetivo es **saber dónde están los problemas**, no resolverlos todavía.

## 3.8 Qué se espera obtener del análisis univariante

Al finalizar este bloque, deberías ser capaz de:

- describir cómo se distribuye cada variable relevante
- identificar variables con asimetría fuerte
- detectar outliers evidentes
- localizar valores ausentes
- anotar posibles problemas de calidad del dato

Este conocimiento será fundamental para:

- analizar relaciones entre variables
- preparar los datos para modelado
- interpretar correctamente los resultados de un modelo.

---

## 3.9 Cómo realizar un análisis univariante en la práctica (Python)

Hasta ahora hemos descrito **qué observar** en un análisis univariante y **por qué es importante**.

Ahora vamos a ver **cómo se hace realmente en Python**, partiendo de una base ya establecida:

- el dataset ha sido cargado en un `DataFrame` llamado `df`
- sabemos qué columnas existen (`df.columns`)
- sabemos qué representa cada variable a nivel conceptual

> El objetivo de este bloque no es memorizar comandos,
>
> sino **entender qué herramienta usar según la pregunta que queremos responder**.

---

### 3.9.1 Primer vistazo numérico: resumen estadístico

Antes de dibujar cualquier gráfico, conviene obtener una **visión numérica rápida** de una variable.

Para una variable numérica concreta (por ejemplo, duración de uso o número de eventos):

```
df["num_logins_30d"].describe()
```

Este resumen permite detectar rápidamente:

- rango típico de valores
- diferencias entre media y mediana (posible asimetría)
- valores extremos sospechosos
- escalas inesperadas

📌 **Lectura guiada importante**

- si la media ≠ mediana → probablemente hay asimetría
- valores máximos muy alejados → posible cola larga u outliers
- mínimos negativos en variables que no deberían → posible error o sentinela

⚠️ Este resumen **no sustituye a los gráficos**, solo orienta.

---

## 3.9.2 Distribución de variables numéricas: histograma

El histograma es la herramienta básica del análisis univariante para variables numéricas.

Ejemplo:

```
import matplotlib.pyplot as plt

df["num_logins_30d"].hist(bins=30)
plt.xlabel("Número de logins (30 días)")
plt.ylabel("Frecuencia")
plt.title("Distribución de logins en 30 días")
plt.show()
```

Este gráfico permite responder preguntas como:

- ¿dónde se concentran la mayoría de los valores?
- ¿la distribución es simétrica o sesgada?
- ¿hay una cola larga?
- ¿existen valores aislados?

En este punto:

- ❌ no se eliminan valores
- ❌ no se transforman variables

Solo se **observa y se anota**.

---

## 3.9.3 Variables discretas o de conteo

Muchas variables relevantes en ML representan **conteos**:

- número de accesos
- número de incidencias
- número de acciones realizadas

Para este tipo de variables, además del histograma, es útil ver la frecuencia exacta de valores:

```
df["support_tickets_90d"].value_counts().sort_index()
```

Esto permite detectar:

- concentración masiva en valores bajos (especialmente ceros)
- pocos valores muy altos
- comportamiento muy desigual entre observaciones

📌 Este patrón es **normal en datos reales** y suele indicar:

- usuarios poco activos
- unos pocos casos muy intensivos
- posibles segmentos ocultos

---

## 3.9.4 Variables categóricas: frecuencias y equilibrio

Para variables categóricas (por ejemplo, tipo de contrato o región), el análisis univariante se basa en **frecuencias**, no en estadísticas numéricas.

```
df["contract_type"].value_counts()
```

Y, si interesa ver proporciones:

```
df["contract_type"].value_counts(normalize=True)
```

Aquí debes observar:

- categorías dominantes
- categorías muy poco frecuentes
- posibles desequilibrios importantes

📌 Un fuerte desequilibrio puede tener impacto posterior en:

- codificación
- estabilidad del modelo
- interpretación de resultados

---

## 3.9.5 Detección de valores ausentes

Una parte esencial del análisis univariante es **localizar valores ausentes**, tanto por variable como a nivel global.

Por variable:

```
df["income_annual"].isna().sum()
```

Vista general del dataset:

```
df.isna().sum().sort_values(ascending=False)
```

En esta fase:

- ❌ no se imputan valores
- ❌ no se eliminan filas

El objetivo es **saber dónde están los problemas**, no resolverlos todavía.

---

## 3.9.6 Qué NO se hace en el análisis univariante

Conviene dejar esto muy claro:

En el análisis univariante **no** se:

- transforman variables
- eliminan outliers automáticamente
- escalan datos
- preparan variables para modelos

El análisis univariante es **diagnóstico**, no tratamiento.

Tomar decisiones aquí suele llevar a errores difíciles de detectar después.

---

## 3.9.7 Resultado esperado de este paso

Tras un análisis univariante bien hecho, deberías tener:

- una descripción clara de cada variable relevante
- notas explícitas sobre:
  - asimetría
  - colas largas
  - valores extremos
  - valores ausentes
- una base sólida para:
  - analizar relaciones entre variables
  - preparar datos con criterio
  - interpretar resultados de modelos sin ingenuidad

Si este paso se hace mal (o se omite), los errores reaparecen más adelante, amplificados.

---

### Cierre del bloque 3.9

En este bloque has aprendido **cómo ejecutar** un análisis univariante de forma consciente:

- partiendo de un dataset real cargado en `df`
- usando herramientas mínimas pero suficientes
- observando antes de decidir
- separando diagnóstico de tratamiento

En el siguiente bloque, el foco cambiará:

> pasaremos de analizar variables por separado
> a **analizar cómo se relacionan entre sí**.

---

### Referencias específicas del bloque 3.9

- Pandas — `Series.describe`
  https://pandas.pydata.org/docs/reference/api/pandas.Series.describe.html
- Pandas — `value_counts` y análisis de frecuencias
  https://pandas.pydata.org/docs/reference/api/pandas.Series.value_counts.html
- Matplotlib — Histogramas
  https://matplotlib.org/stable/api/_as_gen/matplotlib.pyplot.hist.html

---

# 4. Relaciones entre variables: de la observación a la hipótesis

Hasta ahora has analizado **variables de forma aislada**.

Ese paso es imprescindible, pero insuficiente.

En problemas reales, las preguntas importantes casi nunca son univariantes:

- ¿Cómo cambia una variable cuando cambia otra?
- ¿Existen patrones conjuntos?
- ¿El comportamiento es consistente o está dominado por ruido?

Este bloque introduce un **cambio de foco fundamental**:

> Pasamos de describir variables
> a **explorar cómo se comportan juntas**.

Este paso es crítico porque:

- es la base de la **predicción**
- introduce la noción de **dependencia**
- anticipa qué modelos pueden funcionar (y cuáles no)

---

## 4.1 Advertencia estructural: relación no implica causalidad

Antes de ver ningún gráfico, es imprescindible fijar una advertencia clave:

> Detectar una relación no implica explicar su causa.

La visualización permite:

- observar patrones conjuntos
- detectar dependencias aparentes
- generar hipótesis

Pero **no permite**:

- afirmar causalidad
- establecer dirección del efecto
- descartar variables ocultas

Este aviso no es retórico:

es uno de los errores más comunes en análisis de datos y Machine Learning.

### Mini-ejemplo: relación espuria por variable oculta (tiempo)

```
import numpy as np
import matplotlib.pyplot as plt

rng = np.random.default_rng(0)
t = np.arange(200)

x = t + rng.normal(0,10, size=t.size)
y = t + rng.normal(0,10, size=t.size)

plt.scatter(x, y, alpha=0.7)
plt.xlabel("X")
plt.ylabel("Y")
plt.title("Relación aparente inducida por una variable oculta (tiempo)")
plt.show()
```

**Lectura guiada**

- Qué mirar: nube alineada “convincente”.
- Qué concluir: hay dependencia aparente.
- Qué NO concluir: que X causa Y.

---

## 4.2 El diagrama de dispersión como punto de partida

Cuando se analizan **dos variables numéricas**, el gráfico básico es el

**diagrama de dispersión** (*scatter plot*).

La pregunta que responde es simple:

> ¿Existe algún patrón conjunto entre estas dos variables?

### Mini-ejemplo: uso y duración de sesión

```
import matplotlib.pyplot as plt

plt.scatter(df["num_logins_30d"], df["avg_session_minutes"], alpha=0.6)
plt.xlabel("Logins (30 días)")
plt.ylabel("Duración media de sesión (min)")
plt.title("Uso vs duración de sesión")
plt.show()
```

Este gráfico **no decide nada**, pero permite saber si merece la pena seguir mirando.

---

## 4.3 Cómo leer un scatter plot (lectura guiada)

Un scatter plot **no se mira “en general”**.

Se lee siguiendo una secuencia fija de preguntas:

1. ¿Existe una **tendencia global** (creciente / decreciente)?
2. ¿La relación parece **lineal o no lineal**?
3. ¿El ruido domina o la señal es clara?
4. ¿Hay **outliers** que distorsionan la lectura?
5. ¿Aparecen **subgrupos**?

### Mini-ejemplo con checklist explícito

```
x = df["num_logins_30d"]
y = df["avg_session_minutes"]

plt.scatter(x, y, alpha=0.6)
plt.xlabel("Logins")
plt.ylabel("Minutos")
plt.title("Lectura guiada: tendencia · ruido · outliers")
plt.show()

print("Rangos:", (x.min(), x.max()), (y.min(), y.max()))
print("Posibles outliers:")
print(df.loc[x.nlargest(3).index, ["num_logins_30d","avg_session_minutes"]])
```

**Idea clave**

> Si no puedes responder al menos a 2–3 de estas preguntas,
>
> el gráfico no es informativo para el problema.

---

# 4.4 Tendencia y forma: no todo es lineal

Una relación puede existir sin ser lineal.

Esto es crítico porque muchos modelos **asumen linealidad**, y fallan cuando la relación no cumple ese supuesto.

## Mini-ejemplo: relación no lineal clara

```
import numpy as np

rng = np.random.default_rng(1)
x = np.linspace(-3,3,300)
y = x**2 + rng.normal(0,0.7, size=x.size)

plt.scatter(x, y, alpha=0.6)
plt.xlabel("X")
plt.ylabel("Y")
```

```
plt.title("Relación no lineal: correlación lineal engañosa")
plt.show()
```

**Idea fundacional**

> Puede existir una relación real
>
> y aun así **no ser útil para un modelo lineal**.

---

# 4.5 Ruido: el límite práctico de la predicción

El **ruido** es la variabilidad que no explica la relación principal.

Dos relaciones pueden tener:

- la misma tendencia
- pero distinto nivel de ruido

## Mini-ejemplo: misma tendencia, distinto ruido

```
rng = np.random.default_rng(2)
x = np.linspace(0,100,250)

y_low =2 * x + rng.normal(0,10, size=x.size)
y_high =2 * x + rng.normal(0,60, size=x.size)

plt.scatter(x, y_low, alpha=0.6, label="Bajo ruido")
plt.scatter(x, y_high, alpha=0.6, label="Alto ruido")
plt.legend()
plt.title("Misma tendencia, distinto ruido")
plt.show()
```

**Consecuencia clave**

> Más ruido → menor capacidad predictiva
>
> aunque el patrón sea real.

---

# 4.6 Relación con la variable objetivo: cambio de pregunta

Cuando una de las variables es la **objetivo** (`churned`), la pregunta cambia.

Ya no es:

> ¿Existe relación entre X e Y?

Sino:

> ¿Esta variable se distribuye de forma distinta entre los grupos?

### Mini-ejemplo: comparación por grupos

```
import seaborn as sns

sns.boxplot(data=df, x="churned", y="num_logins_30d")
plt.xlabel("Churn (0=no, 1=sí)")
plt.ylabel("Logins (30 días)")
plt.title("Uso según abandono")
plt.show()
```

Este gráfico:

- no decide nada
- no selecciona variables
- **solo orienta** sobre si hay señal potencial

---

## 4.7 Correlación: qué mide y qué no mide

La correlación cuantifica **relación lineal**, nada más.

- puede ser alta por relaciones espurias
- puede ser baja aunque exista relación no lineal
- no implica causalidad

> La correlación es una pista, no una prueba.

### Mini-ejemplo: correlaciones como mapa, no como juez

```
df_num = df.select_dtypes(include="number")
corr = df_num.corr(numeric_only=True)
```

```
corr["churned"].drop("churned").sort_values(
    key=lambda x: x.abs(), ascending=False
).head(8)
```

**Lectura correcta**

- Sirve para **priorizar exploración**
- No sirve para descartar variables automáticamente

---

## 4.8 Resultado esperado del análisis bivariante

Al finalizar este bloque, **no deberías tener decisiones**, sino algo mucho más valioso:

> una lista corta de hipótesis verificables

### Mini-ejemplo: salida esperada (plantilla)

```
hipotesis = [
"Menor uso (logins) podría asociarse a mayor churn → verificar por grupos",
"Más tickets de soporte podrían asociarse a churn → comprobar ruido",
"Plan y cuota mensual podrían ser redundantes → revisar outliers",
]

for i, h in enumerate(hipotesis,1):
print(f"{i}.{h}")
```

---

## Cierre del bloque 4

En este bloque has aprendido a:

- analizar relaciones entre variables con criterio
- leer gráficos bivariantes de forma estructurada
- distinguir señal, ruido y forma
- formular **hipótesis**, no conclusiones

El siguiente paso natural es integrar todo esto en un proceso coherente:

> aplicar estas ideas dentro de un pipeline completo de análisis exploratorio.

---

## Referencias específicas del bloque 4

- Python Data Science Handbook — Relationships and visualization
  https://jakevdp.github.io/PythonDataScienceHandbook/
- Matplotlib — Scatter plots
  https://matplotlib.org/stable/api/_as_gen/matplotlib.pyplot.scatter.html
- Seaborn — Visualizing statistical relationships
  https://seaborn.pydata.org/tutorial/relational.html

---

# 5. Visualización aplicada: primer pipeline de análisis exploratorio (EDA)

En los bloques anteriores has aprendido:

- cómo cargar un dataset real
- cómo analizar variables individualmente
- cómo explorar relaciones entre pares de variables

Ahora vamos a **integrar todo eso** en un proceso coherente.

> Un pipeline de EDA no es una receta fija.
>
> Es una **secuencia razonada de pasos** que evita improvisaciones.

Este bloque introduce un **pipeline mínimo**, diseñado para:

- no perderse en gráficos irrelevantes
- no tomar decisiones prematuras
- preparar el terreno para el modelado

---

## 5.1 Qué es un pipeline de EDA (y por qué lo necesitas)

Un **pipeline de EDA** es una secuencia ordenada de pasos que se repite, con variaciones, en casi cualquier proyecto real.

En este curso usaremos el siguiente esquema base:

1. **Preparación mínima del dataset**
2. **Visión global de relaciones**

3. **Exploración dirigida**
4. **Análisis univariante focalizado**
5. **Comparación por grupos (si hay objetivo)**

La clave no es memorizar pasos, sino entender la lógica:

> Cada paso reduce el espacio de incertidumbre
> y acota el siguiente.

---

## 5.2 Preparación mínima antes de visualizar (EDA defensivo)

Antes de dibujar cualquier gráfico, se realiza una **preparación mínima**.

Objetivo:

> Evitar autoengañarse con gráficos falsos o distorsionados.

Esta preparación **no es limpieza completa**.

Incluye solo decisiones evidentes y documentables.

---

### 5.2.1 Verificación inicial (recordatorio)

```
df.shape
df.head()
df.dtypes
df.isna().sum().sort_values(ascending=False)
```

Aquí solo comprobamos que:

- el dataset se ha cargado correctamente
- las columnas tienen sentido
- los tipos no son absurdos
- existen valores ausentes

---

### 5.2.2 Conversión explícita de tipos evidentes

Ejemplo típico: fechas leídas como texto.

```
df["signup_date"] = pd.to_datetime(df["signup_date"], errors="coerce")
```

📌 **Idea clave**

No “arreglamos” fechas.

Solo evitamos que se interpreten mal.

---

### 5.2.3 Neutralización de sentinelas evidentes

Si el dataset incluye valores que **sabemos** que no representan el fenómeno:

```
df.loc[df["income_annual"] == -1,"income_annual"] = np.nan
```

Esto no es limpieza avanzada:

> Es evitar que un valor falso domine gráficos y estadísticas.

---

## 5.3 Visión global: detectar estructura antes del detalle

Una vez que el dataset es mínimamente coherente, el siguiente paso es obtener una **visión global**.

Pregunta que guía este paso:

> ¿Hay variables que parecen moverse juntas?

Ejemplo conceptual:

```
df_numeric = df.select_dtypes(include="number")
corr = df_numeric.corr()
```

La correlación aquí se usa como:

- **mapa**
- **orientación**
- **no como criterio de decisión**

---

## 5.4 Exploración dirigida: confirmar o descartar hipótesis

A partir de la visión global, se seleccionan **algunos pares concretos** y se analizan con gráficos bivariantes.

Ejemplo:

```
plt.scatter(df["num_logins_30d"], df["avg_session_minutes"])
plt.xlabel("Logins (30 días)")
plt.ylabel("Duración media")
plt.show()
```

Aquí buscamos confirmar:

- forma de la relación
- nivel de ruido
- presencia de outliers
- posibles subgrupos

Si la relación **no se confirma visualmente**, se descarta sin problema.

---

## 5.5 Análisis univariante focalizado (volver a lo individual)

Tras explorar relaciones, se vuelve a variables individuales, pero **solo a las relevantes**.

Ejemplo:

```
df["support_tickets_90d"].hist(bins=30)
plt.title("Distribución de tickets de soporte")
plt.show()
```

Este paso permite:

- entender mejor ruido y asimetría
- anticipar dificultades para el modelado
- decidir si una variable merece atención posterior

---

## 5.6 Comparación por grupos: cuando existe variable objetivo

Si el dataset incluye una variable objetivo (por ejemplo, `churned`), se comparan distribuciones **por grupo**.

Ejemplo:

```
import seaborn as sns

sns.boxplot(data=df, x="churned", y="num_logins_30d")
plt.title("Uso según abandono")
plt.show()
```

La pregunta ya no es “¿cómo es la variable?”, sino:

> ¿cambia esta variable entre los grupos?

Esto ayuda a detectar:

- señales útiles
- solapamiento fuerte
- límites reales de separación

---

## 5.7 Qué NO se hace en el pipeline de EDA

Este bloque **deliberadamente no**:

- entrena modelos
- elimina variables
- toma decisiones finales
- optimiza nada

El EDA sirve para **entender**, no para cerrar.

---

## 5.8 Resultado esperado del pipeline

Al finalizar este bloque, deberías ser capaz de:

- describir la estructura general del dataset
- identificar relaciones plausibles
- detectar ruido dominante
- justificar qué variables explorar después
- anticipar la dificultad del problema

Esto prepara directamente el siguiente bloque:

> preparar los datos para un primer modelo baseline.

---

### Referencias específicas del bloque 5

- Python Data Science Handbook — Exploratory Data Analysis
  https://jakevdp.github.io/PythonDataScienceHandbook/
- Pandas — Working with missing data
  https://pandas.pydata.org/docs/user_guide/missing_data.html
- Seaborn — Relational plots
  https://seaborn.pydata.org/tutorial/relational.html

---

### Cierre del bloque 5

En este bloque has aprendido a:

- aplicar visualización con intención analítica
- seguir un proceso estructurado
- reducir incertidumbre sin precipitar decisiones
- preparar el terreno para el modelado

El siguiente paso ya no es explorar más, sino **convertir observaciones en decisiones técnicas mínimas**.

---

## 6. Preparación de datos orientada al primer modelo baseline

Después del análisis exploratorio, el siguiente paso natural no es "mejorar los datos", sino **hacerlos utilizables por un modelo de Machine Learning**.

Este bloque introduce una idea central que se mantendrá durante todo el curso:

> Preparar datos no significa arreglarlos,
>
> significa **tomar decisiones técnicas mínimas y justificadas**.

El objetivo aquí **no es obtener el mejor dataset posible**, sino uno:

- coherente,
- reproducible,

- y adecuado para entrenar un **primer modelo simple e interpretable**.

---

## 6.1 Preparar datos no es limpiar datos

En proyectos reales, es muy habitual caer en esta tentación:

> "Antes de modelar, vamos a limpiar todo bien."

Este enfoque suele ser contraproducente al inicio porque:

- introduce decisiones sin referencia,
- consume mucho tiempo,
- y puede ocultar problemas estructurales del dato.

En esta sesión, la preparación de datos tiene un objetivo muy concreto:

👉 **permitir entrenar un primer modelo baseline**

👉 **sin introducir fugas de información (*data leakage*)**

---

## 6.2 Qué necesita un modelo (y qué no necesita todavía)

Un modelo supervisado clásico necesita:

- variables numéricas o codificadas,
- una variable objetivo bien definida,
- valores ausentes tratados explícitamente,
- una estructura de entrada coherente.

No necesita todavía:

- distribuciones "bonitas",
- ausencia total de outliers,
- ingeniería compleja de variables,
- optimización avanzada.

Esta distinción evita sobredimensionar esta fase.

---

## 6.3 Separación temprana de variables explicativas y objetivo

Antes de cualquier transformación, se separan claramente:

- **X** → variables explicativas
- **y** → variable objetivo

Esta separación no es solo técnica, es conceptual:

> La variable objetivo no debe influir
>
> en cómo se preparan las variables explicativas.

Este principio evita una de las formas más sutiles de *data leakage*.

---

## 6.4 Preparación mínima en la práctica: un ejemplo completo y honesto

En este punto vamos a **materializar** las ideas anteriores con un ejemplo completo.

> No buscamos la mejor preparación posible.
>
> Buscamos una preparación **suficiente, reproducible y defendible**
>
> para entrenar un primer modelo baseline.

---

### 6.4.1 Separación explícita de X e y

```
X = df.drop(columns=["churned"])
y = df["churned"]
```

**Por qué se hace aquí y no antes**

- El EDA puede analizar `churned`
- La preparación **no debe usarlo**
- Esta separación fija un límite claro desde el principio

---

### 6.4.2 Identificación explícita de tipos de variables

```
num_cols = X.select_dtypes(include=["int64","float64"]).columns
cat_cols = X.select_dtypes(include=["object","category"]).columns
```

📌 **Idea clave**

No “adivinamos” tipos.

Los **explicitamos** para evitar errores silenciosos.

---

### 6.4.3 Tratamiento mínimo de valores ausentes

```
from sklearn.impute import SimpleImputer

num_imputer = SimpleImputer(strategy="median")
cat_imputer = SimpleImputer(strategy="most_frequent")
```

**Justificación de estas decisiones**

- mediana → robusta frente a outliers
- categoría más frecuente → no introduce estructura nueva

> No afirmamos que sea lo óptimo.
>
> Afirmamos que es **suficiente y reversible**.

---

### 6.4.4 Codificación básica de variables categóricas

```
from sklearn.preprocessing import OneHotEncoder

encoder = OneHotEncoder(handle_unknown="ignore", sparse_output=False)
```

**Por qué** `handle_unknown="ignore"`

En datos reales:

- el conjunto de test puede contener categorías nuevas
- el pipeline **no debe romperse** por ello

---

### 6.4.5 Escalado de variables numéricas

```
from sklearn.preprocessing import StandardScaler

scaler = StandardScaler()
```

📌 **Advertencia explícita**

- no todos los modelos requieren escalado
- se introduce aquí por coherencia y reutilización futura

---

### 6.4.6 Construcción del pipeline mínimo de preparación

```
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline

preprocess = ColumnTransformer(
    transformers=[
        ("num", Pipeline([
            ("imputer", num_imputer),
            ("scaler", scaler)
        ]), num_cols),
        ("cat", Pipeline([
            ("imputer", cat_imputer),
            ("encoder", encoder)
        ]), cat_cols)
    ]
)
```

📌 **Este pipeline es una pieza estructural del curso**

- evita inconsistencias entre entrenamiento y test
- evita fugas de información
- se reutilizará en sesiones futuras
- **no está optimizado**

---

### 6.4.7 Qué hemos conseguido (y qué no)

Con este pipeline:

✔️ los datos pueden entrar en un modelo

✔️ las decisiones son explícitas

✔️ el flujo es reproducible

Pero **no**:

- hemos creado nuevas variables
- hemos comparado estrategias
- hemos buscado rendimiento máximo

Eso vendrá más adelante, con contexto.

---

### 6.4.8 ¿Qué ocurre realmente con mis datos a partir de aquí?

Llegados a este punto, es importante aclarar **qué tenemos y qué no tenemos**.

Partimos de:

```
df
```

Un `DataFrame` con:

- valores ausentes
- variables categóricas
- escalas distintas
- estructura heterogénea

Después hemos definido:

- `X` → variables explicativas (sin tocar)
- `y` → variable objetivo
- `preprocess` → pipeline de preparación

📌 **Clave fundamental**

> Hasta ahora NO hemos modificado el DataFrame original.

El pipeline **no transforma los datos todavía**.

---

### ¿Cuándo se preparan realmente los datos?

La transformación ocurre **cuando el pipeline se ajusta** (*fit*):

```
preprocess.fit(X)
```

O, más habitualmente, cuando se usa dentro de un modelo:

```
pipeline.fit(X_train, y_train)
```

En ese momento, internamente ocurre todo esto:

1. Se calculan estadísticas de imputación (solo con entrenamiento)
2. Se ajusta el escalado
3. Se aprenden las categorías
4. Se construye la representación numérica final

Pero todo eso ocurre **dentro del pipeline**, no en el `DataFrame`.

---

## ¿Puedo ver los datos ya preparados?

Sí, de forma explícita:

```
X_prepared = preprocess.fit_transform(X)
```

Esto devuelve una **matriz numérica**, no un DataFrame:

- sin valores ausentes
- con variables codificadas
- con escalas homogéneas

📌 **Pero atención**

En el flujo normal:

- **no necesitas** trabajar con `X_prepared`
- el modelo recibe estos datos directamente desde el pipeline

Verlos sirve solo para **entender qué está pasando**.

---

## Resumen operativo del flujo

```
DataFrame original (df)
        ↓
Separación X / y
        ↓
Pipeline de preparación (preprocess)
        ↓
Datos numéricos preparados (internos)
        ↓
Modelo
```

Este será **el esquema mental que usarás en todo el curso**.

---

## 6.5 Qué NO se hace todavía en esta fase

De forma deliberada, en este bloque **no** se:

- realiza *feature engineering*,
- prueban múltiples imputaciones,
- ajustan hiperparámetros,
- comparan modelos.

Aquí solo se construye:

> una base técnica honesta para el primer modelo.

---

## 6.6 Resultado esperado del bloque

Al finalizar este bloque, deberías ser capaz de:

- explicar qué decisiones mínimas se han tomado,
- justificar por qué no se ha "limpiado todo",
- entender cómo entran los datos al modelo,
- reconocer este pipeline en soluciones futuras,
- anticipar las limitaciones del baseline.

Esto prepara directamente el siguiente paso del curso.

---

## Referencias específicas del bloque 6


- scikit-learn — Pipelines and preprocessing
  https://scikit-learn.org/stable/modules/compose.html
- scikit-learn — SimpleImputer
  https://scikit-learn.org/stable/modules/impute.html
- scikit-learn — Encoding categorical features
  https://scikit-learn.org/stable/modules/preprocessing.html#encoding-categorical-features


---

### Cierre del bloque 6

En este bloque has aprendido que:

- preparar datos no es optimizar ni limpiar en exceso,
- las decisiones deben ser mínimas y justificadas,
- el pipeline protege contra errores estructurales,
- un buen baseline empieza con una preparación honesta.

A partir de aquí, ya estamos listos para responder a la pregunta inevitable:

> ¿Qué pasa si entrenamos un modelo con estos datos?

---

# 7. Primer modelo baseline: construir, entrenar y entender un modelo simple

Con los datos ya **mínimamente preparados**, llega un momento inevitable:

> ¿Qué ocurre si entrenamos un modelo con estos datos?

Este bloque introduce el **primer modelo del curso**, pero con una advertencia clara:

> Este modelo no está aquí para ganar,
> está aquí para **servir de referencia**.

---

## 7.1 Qué es un modelo baseline (y por qué es imprescindible)

Un **modelo baseline** es:

- simple
- rápido de entrenar
- fácil de interpretar
- con pocas decisiones técnicas

Su función **no es**:

- obtener el mejor rendimiento posible

- competir con modelos complejos
- demostrar “nivel de ML”

Su función es esta:

> Establecer un punto de partida honesto.

Todo lo que venga después solo tiene sentido **comparado con este baseline**.

---

## 7.2 Por qué empezar con un modelo lineal

En problemas de clasificación binaria como el nuestro (`churned` sí / no), un modelo clásico y adecuado como baseline es la **regresión logística**.

Se elige porque:

- es interpretable
- produce probabilidades
- tiene supuestos claros
- se entrena rápidamente

Aunque su nombre incluya “regresión”, se utiliza ampliamente para **clasificación**.

---

## 7.3 Separación entre entrenamiento y evaluación

Antes de entrenar, el dataset se divide en dos partes:

- **entrenamiento** → el modelo aprende aquí
- **test** → el modelo se evalúa aquí

Esto simula una situación real:

> Aprender con datos conocidos
>
> y comprobar el comportamiento con datos no vistos.

📌 El conjunto de test **no se usa para aprender nada**.

---

## 7.4 Integración explícita del pipeline de preparación y el modelo

Aquí ocurre algo fundamental:

**el modelo no recibe el DataFrame crudo**.

Recibe los datos **a través del pipeline de preparación** definido en el bloque 6.

```
from sklearn.pipeline import Pipeline
from sklearn.linear_modelimport LogisticRegression

model = LogisticRegression(max_iter=1000)

pipeline = Pipeline(steps=[
    ("preprocess", preprocess),
    ("model", model)
])
```

📌 **Idea clave**

> El pipeline completo (preparación + modelo)
>
> se comporta como una única unidad.

Esto evita errores estructurales y fugas de información.

---

## 7.5 Entrenamiento del modelo baseline

Entrenar un modelo significa **ajustarlo a los datos de entrenamiento**:

```
pipeline.fit(X_train, y_train)
```

En este paso:

- se ajusta la imputación
- se ajusta el escalado
- se ajusta la codificación
- se ajusta el modelo

Todo ocurre **solo con los datos de entrenamiento**.

---

## 7.6 Qué produce realmente un modelo de clasificación

Un modelo de clasificación binaria **no produce directamente etiquetas**.

Produce **probabilidades**, por ejemplo:

- 0.83 → alta probabilidad de abandono
- 0.47 → probabilidad intermedia
- 0.12 → baja probabilidad

Estas probabilidades se convierten en decisiones usando un **umbral**, que por defecto suele ser 0.5 (pero no es sagrado).

---

## 7.7 Primeras predicciones: qué mirar (y qué no)

Tras entrenar el modelo, podemos obtener predicciones:

```
y_pred_proba = pipeline.predict_proba(X_test)[:,1]
```

En este punto **no buscamos métricas todavía**.

Solo comprobamos cosas básicas:

- ¿el modelo predice siempre lo mismo?
- ¿las probabilidades están todas cerca de 0 o de 1?
- ¿hay variedad razonable?

Este paso sirve para detectar errores graves, no para evaluar rendimiento.

---

## 7.8 Qué NO se hace todavía con este modelo

En este bloque **no**:

- se ajustan hiperparámetros
- se comparan modelos
- se elige umbral óptimo
- se sacan conclusiones finales

Este modelo existe solo para:

> Tener un punto de referencia honesto.

---

## 7.9 Resultado esperado del bloque

Al finalizar este bloque, deberías ser capaz de:

- explicar qué es un modelo baseline
- entender cómo entra el dato al modelo
- describir qué aprende (y qué no)
- aceptar rendimientos modestos
- usar este modelo como referencia futura

Esto prepara directamente para el siguiente paso de la evaluación del modelo.

---

## Referencias específicas del bloque 7

- scikit-learn — Logistic Regression

  https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LogisticRegression.html
- scikit-learn — Pipelines

  https://scikit-learn.org/stable/modules/compose.html
- scikit-learn — Train/test split

  https://scikit-learn.org/stable/modules/cross_validation.html

---

# 8. Evaluación del modelo: medir rendimiento y entender errores

Una vez entrenado el modelo baseline, necesitamos responder a una pregunta esencial:

> ¿Cómo se comporta este modelo cuando se equivoca?

Evaluar un modelo **no es obtener un número**, sino **entender sus errores**.

---

## 8.1 Qué produce realmente un modelo de clasificación

Un modelo de clasificación binaria **no produce directamente "sí" o "no"**.

Produce **probabilidades**:

```
y_pred_proba = pipeline.predict_proba(X_test)[:,1]
```

Esto significa:

- valores cercanos a 1 → alta probabilidad de clase positiva
- valores cercanos a 0 → baja probabilidad

Para tomar decisiones, necesitamos convertir estas probabilidades en etiquetas.

---

## 8.2 De probabilidades a predicciones (umbral por defecto)

Usamos un umbral inicial (convencional) de **0.5**:

```
y_pred = (y_pred_proba >=0.5).astype(int)
```

📌 **Importante**

Este umbral **no es óptimo**, es solo un punto de partida.

---

## 8.3 La matriz de confusión: ver los errores

La **matriz de confusión** muestra cómo se comparan:

- lo que realmente ocurrió (`y_test`)
- lo que el modelo predijo (`y_pred`)

```
from sklearn.metrics import confusion_matrix

cm = confusion_matrix(y_test, y_pred)
cm
```

La matriz tiene esta estructura:

```
            Predicho0   Predicho1
Real0         TN          FP
Real1         FN          TP
```

Donde:

- **TP**: abandonos correctamente detectados
- **FP**: clientes fieles marcados como abandono
- **FN**: abandonos no detectados

- **TN**: clientes fieles bien clasificados

📌 **Regla fundamental**

> Antes de mirar métricas,
>
> entiende esta matriz.

---

## 8.4 Accuracy: la métrica más engañosa

La **accuracy** mide el porcentaje total de aciertos:

```
from sklearn.metrics import accuracy_score

accuracy_score(y_test, y_pred)
```

Es útil cuando:

- las clases están equilibradas
- los errores cuestan lo mismo

Pero puede ser muy engañosa en problemas como churn.

📌 **Ejemplo conceptual**

Si el 80% no abandona,

un modelo que siempre diga “no abandono” tiene 80% de accuracy

y **no sirve para nada**.

---

## 8.5 Precision y recall: errores distintos

Aquí empezamos a entender **qué tipo de error importa más**.

### Recall (sensibilidad)

> De todos los abandonos reales,
>
> ¿cuántos detecta el modelo?

```
from sklearn.metrics import recall_score
```

```
recall_score(y_test, y_pred)
```

### Precision

> De los clientes marcados como abandono,
> ¿cuántos lo son realmente?

```
from sklearn.metrics import precision_score

precision_score(y_test, y_pred)
```

📌 **Trade-off inevitable**

- subir recall suele bajar precision
- subir precision suele bajar recall

No existe una solución perfecta.

---

## 8.6 F1-score: equilibrio entre precision y recall

El **F1-score** combina ambas métricas:

```
from sklearn.metrics import f1_score

f1_score(y_test, y_pred)
```

Es útil cuando:

- las clases están desbalanceadas
- ambas métricas importan

Pero sigue siendo un **resumen**, no una explicación.

---

## 8.7 Evaluar sin fijar umbral: curva ROC

Hasta ahora todo depende del umbral (0.5).

La **curva ROC** evalúa el modelo **para todos los umbrales posibles**.

```
from sklearn.metrics import roc_curve

fpr, tpr, thresholds = roc_curve(y_test, y_pred_proba)
```

- **TPR** → tasa de verdaderos positivos (recall)
- **FPR** → tasa de falsos positivos

Podemos visualizarla:

```
import matplotlib.pyplot as plt

plt.plot(fpr, tpr)
plt.plot([0,1], [0,1], linestyle="--")
plt.xlabel("False Positive Rate")
plt.ylabel("True Positive Rate")
plt.title("Curva ROC")
plt.show()
```

📌 La diagonal representa comportamiento aleatorio.

---

## 8.8 ROC-AUC: capacidad discriminativa

El **ROC-AUC** resume la curva ROC en un solo número:

```
from sklearn.metrics import roc_auc_score

roc_auc_score(y_test, y_pred_proba)
```

Interpretación:

- 0.5 → aleatorio
- 0.7–0.8 → aceptable
- 0.8 → buena separación
- 1.0 → perfecta (raro en datos reales)

📌 **Importante**

ROC-AUC **no dice nada del umbral óptimo**.

Solo mide **capacidad de separación**.

## 8.9 Qué NO se decide todavía

En este bloque **no**:

- se ajusta el umbral final
- se optimizan métricas
- se elige el “mejor” modelo

Aquí solo buscamos **entender el comportamiento del baseline**.

## 8.10 Resultado esperado del bloque

Al finalizar este bloque, deberías ser capaz de:

- construir y leer una matriz de confusión
- calcular e interpretar accuracy, precision, recall y F1
- entender qué mide una curva ROC
- interpretar ROC-AUC sin mitificarlo
- justificar por qué un modelo puede ser limitado

Esto prepara el siguiente paso natural para decidir el umbral de decisión

## Referencias específicas del bloque 8

- scikit-learn — Model evaluation

  https://scikit-learn.org/stable/modules/model_evaluation.html
- scikit-learn — Confusion matrix

  https://scikit-learn.org/stable/modules/generated/sklearn.metrics.confusion_matrix.html
- scikit-learn — ROC and AUC

  https://scikit-learn.org/stable/modules/roc.html

# 9. Umbral de decisión y trade-offs operativos

Hasta ahora hemos evaluado el modelo baseline fijando un umbral convencional de **0.5** para convertir probabilidades en predicciones.

Pero en un problema real aparece una pregunta inevitable:

> ¿A partir de qué probabilidad tomo una decisión?

Este bloque introduce el **umbral de decisión** y muestra por qué **no es una decisión técnica neutra**, sino una decisión con impacto operativo.

---

## 9.1 De probabilidades a decisiones: recordatorio operativo

El modelo produce probabilidades:

```
y_pred_proba = pipeline.predict_proba(X_test)[:,1]
```

Para convertirlas en etiquetas, elegimos un umbral:

```
threshold =0.5
y_pred = (y_pred_proba >= threshold).astype(int)
```

📌 **Idea clave**

> Cambiar el umbral no cambia el modelo.
>
> Cambia **cómo lo usas**.

---

## 9.2 Qué cambia cuando cambias el umbral

Si bajas el umbral (por ejemplo, 0.3):

- detectas más positivos (más “churn”)
- aumentan los verdaderos positivos (TP)
- pero también aumentan los falsos positivos (FP)

Si subes el umbral (por ejemplo, 0.7):

- reduces falsos positivos (FP)
- pero aumentan falsos negativos (FN)

Esto es un trade-off inevitable.

---

## 9.3 Ver el efecto del umbral con matriz de confusión y métricas

Vamos a medir qué ocurre con **tres umbrales** distintos: 0.3, 0.5 y 0.7.

```
from sklearn.metrics import confusion_matrix, precision_score, recall_score, f1_scor
e

def eval_threshold(y_true, y_proba, threshold):
    y_pred = (y_proba >= threshold).astype(int)
    cm = confusion_matrix(y_true, y_pred)
    precision = precision_score(y_true, y_pred, zero_division=0)
    recall = recall_score(y_true, y_pred, zero_division=0)
    f1 = f1_score(y_true, y_pred, zero_division=0)
    return cm, precision, recall, f1

for tin [0.3,0.5,0.7]:
    cm, p, r, f1 = eval_threshold(y_test, y_pred_proba, t)
print(f"Umbral: {t}")
print("Matriz de confusión:\n", cm)
print(f"Precision: {p:.3f} | Recall:{r:.3f} | F1:{f1:.3f}")
print("-" *40)
```

📌 **Cómo leer esto**

- Si al bajar el umbral sube mucho el recall pero cae la precision → estás capturando más abandonos pero con más "falsas alarmas".
- Si al subir el umbral sube la precision pero cae el recall → estás siendo más estricto, pero te dejas abandonos sin detectar.

---

## 9.4 Relación directa entre umbral, precision y recall

Este bloque conecta con lo aprendido en evaluación:

- umbral ↓ → **recall ↑** (normalmente) y **precision ↓**
- umbral ↑ → **precision ↑** (normalmente) y **recall ↓**

No es una ley matemática absoluta, pero es el patrón típico.

---

## 9.5 Umbral y contexto: por qué no existe un “mejor umbral universal”

Elegir un umbral depende del contexto del problema, por ejemplo:

- Si el coste de **no detectar un abandono** es muy alto → priorizas **recall**
- Si el coste de **actuar sobre clientes que no abandonan** es alto → priorizas **precision**
- Si tienes recursos limitados (solo puedes actuar sobre X clientes) → umbral ajustado a capacidad operativa

📌 **Idea clave del curso**

> El modelo puntúa.
>
> El negocio (o el problema real) decide el umbral.

---

## 9.6 Curva Precision–Recall: herramienta para decidir con clases desbalanceadas

En problemas con desbalance (como churn), la curva Precision–Recall suele ser más informativa que ROC.

```
from sklearn.metrics import precision_recall_curve

precisions, recalls, thresholds = precision_recall_curve(y_test, y_pred_proba)
```

Visualización:

```
import matplotlib.pyplot as plt

plt.plot(recalls, precisions)
plt.xlabel("Recall")
plt.ylabel("Precision")
plt.title("Curva Precision–Recall")
plt.show()
```

Cómo usarla:

- eliges un objetivo (p.ej. “quiero recall ≥ 0.8”)

- miras qué precision puedes conseguir con ese recall
- decides si es aceptable operativamente

---

## 9.7 Errores comunes con el umbral (advertencias fundacionales)

Errores que deben evitarse:

- ajustar el umbral usando el conjunto de test como si fuera entrenamiento
- elegir el umbral solo porque maximiza una métrica
- pensar que cambiar el umbral "mejora el modelo"
- olvidar el coste real de FP/FN

📌 Cambiar el umbral **no mejora el modelo**: cambia el punto de operación.

---

## 9.8 Resultado esperado del bloque

Al finalizar este bloque, deberías ser capaz de:

- explicar qué es un umbral y por qué existe
- convertir probabilidades en decisiones con distintos umbrales
- ver cómo cambian FP/FN en la matriz de confusión
- relacionar umbral con precision y recall
- justificar un umbral en función del contexto

Esto prepara para el siguiente paso natural de comparación de modelos y validación cruzada.

---

## Referencias específicas del bloque 9

- scikit-learn — `precision_recall_curve`

  https://scikit-learn.org/stable/modules/generated/sklearn.metrics.precision_recall_curve.html

- scikit-learn — Precision, recall, F1

  https://scikit-learn.org/stable/modules/model_evaluation.html#precision-recall-f-measure-metrics

- scikit-learn — Glossary (threshold, decision function)
  https://scikit-learn.org/stable/glossary.html

---

# 10. Comparación de modelos y validación cruzada: medir estabilidad, no buscar al "ganador"

Hasta ahora hemos trabajado con:

- un modelo baseline
- un único conjunto de entrenamiento y test
- métricas razonables
- decisiones interpretables

Esto es suficiente para **aprender**, pero no para **confiar plenamente** en una conclusión.

Este bloque introduce una idea clave:

> Un resultado aislado no es una estimación fiable del rendimiento real.

---

## 10.1 El problema del "single split"

Cuando separas los datos una sola vez en entrenamiento y test:

- el resultado depende del azar de esa partición
- dos splits distintos pueden dar métricas distintas
- las conclusiones pueden ser frágiles

Esto es especialmente relevante cuando:

- el dataset no es enorme
- hay ruido
- las clases se solapan (como en churn)

📌 **Idea clave**

> Un único test set te dice qué pasó una vez,
> no *cómo se comporta el modelo en general*.

## 10.2 Qué es la validación cruzada (sin fórmulas)

La **validación cruzada** consiste en:

- dividir el conjunto de entrenamiento en *k* partes
- entrenar el modelo *k* veces
- cada vez usar una parte distinta para validar
- obtener *k* estimaciones del rendimiento

El objetivo **no es mejorar el modelo**, sino:

> medir su comportamiento medio y su variabilidad.

## 10.3 Regla metodológica fundamental (no negociable)

Una regla que debe quedar absolutamente clara:

> La validación cruzada se aplica solo sobre el conjunto de entrenamiento.

El conjunto de test:

- no se usa para ajustar
- no se usa para comparar
- se reserva para la evaluación final

Romper esta regla introduce **data leakage**, incluso aunque no sea evidente.

## 10.4 Validación cruzada aplicada al pipeline completo

Cuando trabajamos con pipelines, la validación cruzada se aplica **al pipeline entero**, no solo al modelo.

```
from sklearn.model_selection import cross_val_score

scores = cross_val_score(
    pipeline,
    X_train,
    y_train,
    cv=5,
```

```
    scoring="roc_auc"
)
```

📌 **Qué ocurre aquí internamente**

En cada fold:

- se ajusta el preprocesado
- se entrena el modelo
- se evalúa en datos no vistos

Todo de forma **aislada y correcta**.

---

## 10.5 Interpretar media y variabilidad

La validación cruzada no produce un único número, sino un conjunto de valores.

De ahí extraemos:

```
scores.mean(), scores.std()
```

Interpretación fundacional:

- **media** → rendimiento esperado
- **desviación estándar** → estabilidad

Lectura guiada:

- media aceptable + baja desviación → modelo estable
- media aceptable + alta desviación → modelo sensible al split
- media baja → límite del modelo o del dato

📌 Ignorar la variabilidad es uno de los errores más comunes.

---

## 10.6 Comparar modelos: cuándo tiene sentido

La validación cruzada permite comparar modelos **de forma más justa**, por ejemplo:

- baseline (regresión logística)
- modelo más flexible (árbol de decisión)

Pero la comparación debe centrarse en:

- mejora media real
- estabilidad
- complejidad añadida

Una mejora pequeña puede:

- no ser significativa
- no justificar mayor complejidad
- no ser útil operativamente

📌 **Idea clave**

> Más complejo ≠ mejor
>
> Mejor ≠ útil

---

## 10.7 Qué NO se hace todavía en este bloque

En esta sesión **no**:

- se hace búsqueda exhaustiva de hiperparámetros
- se prueban muchos algoritmos
- se optimiza agresivamente
- se elige el "modelo final"

Este bloque existe para responder a una sola pregunta:

> ¿Hay margen real de mejora respecto al baseline?

---

## 10.8 Resultado esperado del bloque

Al finalizar este bloque, deberías ser capaz de:

- explicar por qué un único split no es suficiente
- entender qué aporta la validación cruzada
- interpretar media y desviación
- comparar modelos con criterio
- aceptar cuando la mejora es marginal o inexistente

---

## 10.9 Cierre de la sesión: qué has aprendido realmente

Con este bloque se cierra la **primera vuelta completa** al ciclo de Machine Learning:

1. entender los datos
2. explorarlos con intención
3. preparar un pipeline honesto
4. entrenar un baseline
5. evaluar errores
6. ajustar decisiones (umbral)
7. medir estabilidad

Esto no es una colección de técnicas.

Es un **modo de pensar** que se repetirá en todo el curso.

---

# Referencias específicas del bloque 10

- scikit-learn — Cross-validation

  https://scikit-learn.org/stable/modules/cross_validation.html

- scikit-learn — `cross_val_score`

  https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.cross_val_score.html

- scikit-learn — Glossary (cross-validation, overfitting)

  https://scikit-learn.org/stable/glossary.html