Datos, NumPy y visualización como base del aprendizaje automático

1. Introducción de la sesión

En la sesión anterior has trabajado con Python como lenguaje: variables, estructuras básicas, funciones y control del flujo. Eso te permite **escribir programas correctos**.

En esta sesión das un paso distinto:

> Empiezas a trabajar con datos.

Trabajar con datos no consiste únicamente en almacenar números o recorrer listas. Implica:

- representar información de forma coherente,
- operar con conjuntos completos de valores,
- interpretar resultados numéricos,
- y detectar errores que no generan excepciones, pero invalidan conclusiones.

A lo largo de esta sesión aprenderás:

- por qué Python “tal cual” no es suficiente para trabajar con datos de forma rigurosa,
- cómo NumPy introduce un modelo de representación adecuado,
- por qué la forma y el tipo de los datos importan tanto como los valores,
- y por qué visualizar es una herramienta de verificación, no un adorno.

Esta sesión **no trata aún de entrenar modelos**, sino de **preparar el terreno correctamente**.

Sin esta base, cualquier modelo posterior será frágil.

---

2. Bloque 1 — De Python “general” a datos estructurados

2.1 El problema: listas y bucles no escalan como modelo mental

Hasta ahora, cuando has querido trabajar con varios valores, has usado listas de Python.

Por ejemplo:

```
values = [1, 2, 3, 4, 5]

result = []
for v in values:
    result.append(v * 2)

print(result)
```

Este código es correcto y funciona.

Sin embargo, plantea varios problemas cuando el objetivo es trabajar con datos de forma sistemática:

- obliga a pensar elemento a elemento,
- mezcla lógica de control con lógica matemática,
- no expresa claramente la intención (“multiplicar todo el conjunto”),
- no escala bien a datos grandes o multidimensionales.

En ciencia de datos e IA, **no se piensa en términos de bucles**, sino en términos de **operaciones sobre conjuntos completos**.

2.2 Pensar en datos no es pensar en elementos

Cuando trabajas con datos, lo habitual es querer expresar ideas como:

- “suma este valor a todas las observaciones”
- “centra esta variable”
- “compara dos conjuntos de datos”

- “aplica la misma operación a toda una matriz”

Expresar estas ideas con listas y bucles:

- es verboso,
- es propenso a errores,
- oculta el significado matemático de la operación.

Necesitas una forma de decirle al código:

> “Esto no es una colección de elementos sueltos,
> es un **objeto matemático** sobre el que quiero operar”.

Aquí es donde aparece NumPy.

2.3 NumPy no es una comodidad, es un cambio de modelo

NumPy introduce una estructura fundamental: el **array numérico**.

Un array NumPy no es una lista mejorada. Es otra cosa:

- todos los valores tienen el mismo tipo,
- la estructura tiene una forma definida,
- las operaciones están pensadas para actuar sobre el conjunto completo.

Veamos una comparación directa.

Con listas:

```
a = [1, 2, 3]
b = [4, 5, 6]

print(a + b)
```

Salida:

```
[1, 2, 3, 4, 5, 6]
```

Esto **no es una suma matemática**, es una concatenación.

Con NumPy:

```
import numpy as np

a = np.array([1, 2, 3])
b = np.array([4, 5, 6])

print(a + b)
```

Salida:

```
[5 7 9]
```

Aquí la operación tiene **significado matemático claro**.

2.4 Operar con conjuntos completos (sin bucles)

Una de las ideas clave que debes interiorizar desde este punto es la siguiente:

> En NumPy, las operaciones se formulan sobre el conjunto,
>
> no sobre cada elemento por separado.

Ejemplo:

```
values = np.array([10, 20, 30, 40])

print("Valores originales:", values)

scaled = values * 0.1
print("Valores escalados:", scaled)
```

No hay bucle.

No hay acumuladores.

No hay lógica auxiliar.

La intención del código es clara: **escalar todo el conjunto**.

Este tipo de expresividad es esencial en IA, porque:

- reduce errores,
- mejora la legibilidad,
- y refleja directamente el razonamiento matemático.

2.5 Primer aviso importante: el código puede funcionar y estar mal

A partir de este punto del curso, empieza a aparecer un tipo de error nuevo:

- el código **no falla**,
- no hay excepciones,
- los resultados “parecen razonables”,
- pero conceptualmente son incorrectos.

Ejemplo sencillo:

```
values = np.array([1, 2, 3])
result = values + [10]

print(result)
```

Este código se ejecuta y devuelve un resultado.

Más adelante aprenderás **por qué funciona** y **cuándo es peligroso**.

Por ahora, quédate con esta idea:

> En trabajo con datos, entender qué significa una operación
>
> es tan importante como que el código se ejecute.

2.6 Qué se fija en este bloque

Al terminar este bloque, debes tener claras estas ideas:

- trabajar con datos no es lo mismo que trabajar con listas,

- los bucles no son el modelo mental adecuado,
- NumPy introduce una representación pensada para operaciones matemáticas,
- el significado de una operación importa tanto como el resultado numérico.

Este bloque prepara el terreno para el siguiente:

> entender qué es realmente un array NumPyy por qué su estructura importa.

Referencias específicas — Bloque 1

- NumPy — What is NumPy?
  https://numpy.org/doc/stable/user/whatisnumpy.html
- NumPy — Quickstart tutorial
  https://numpy.org/doc/stable/user/quickstart.html

---

3. Bloque 2 — NumPy como modelo de representación numérica

En el bloque anterior has visto por qué las listas de Python no son un buen modelo para trabajar con datos de forma rigurosa. En este bloque vamos a ir un paso más allá:

> No solo importa qué valores tienes,
>
> importa **cómo están representados**.

NumPy no se limita a permitir operaciones más cómodas. Introduce un **modelo de representación numérica explícito**, y entenderlo bien es clave para evitar errores más adelante.

3.1 El `ndarray`: algo distinto a una lista (aunque se le parezca)

Cuando creas un array con NumPy, estás creando un objeto de tipo `ndarray`.

```
import numpy as np
```

```
values = np.array([1, 2, 3, 4])

print(type(values))
print(values)
```

Salida:

```
<class 'numpy.ndarray'>
[1 2 3 4]
```

Aunque visualmente se parezca a una lista, un `ndarray` es conceptualmente distinto:

- todos los elementos tienen el mismo tipo,
- los datos se almacenan de forma contigua en memoria,
- la estructura está pensada para operaciones matemáticas eficientes.

Una lista, en cambio, es una colección heterogénea de referencias a objetos.

3.2 Homogeneidad de tipo: una decisión, no una limitación

En NumPy, todos los elementos de un array comparten el mismo tipo (`dtype`).

```
mixed = np.array([1, 2.5, 3])
print(mixed)
print(mixed.dtype)
```

Salida:

```
[1.  2.5 3. ]
float64
```

NumPy ha convertido todos los valores a un tipo común. Esto no es un detalle técnico menor:

- permite operaciones vectorizadas rápidas,
- evita ambigüedades,

- hace explícita la naturaleza numérica del dato.

A partir de ahora, debes asumir que:

> cuando trabajas con NumPy, el tipo es parte del significado del dato.

3.3 Crear arrays: ser explícito importa

Aunque NumPy puede inferir el tipo automáticamente, es buena práctica ser explícito cuando el contexto lo requiere.

```
values = np.array([10, 20, 30], dtype=np.float64)

print(values)
print(values.dtype)
```

Elegir el tipo correcto:

- evita conversiones implícitas,
- previene errores silenciosos,
- y será relevante cuando trabajes con grandes volúmenes de datos.

Más adelante verás que una mala elección de tipo puede afectar incluso al rendimiento y a la estabilidad de modelos.

3.4 La forma (`shape`) como contrato estructural

Todo array NumPy tiene una **forma** (`shape`), que describe cómo están organizados los datos.

```
v = np.array([1, 2, 3, 4])
print(v.shape)
```

Salida:

```
(4,)
```

Este array es un vector unidimensional.

Ahora una matriz:

```
m = np.array([
    [1, 2, 3],
    [4, 5, 6]
])

print(m)
print(m.shape)
```

Salida:

```
[[1 2 3]
 [4 5 6]]
(2, 3)
```

La forma no es un detalle accesorio:

> La forma es un contrato que condiciona
>
> qué operaciones tienen sentido y cuáles no.

3.5 Dimensión y significado

Un mismo conjunto de números puede tener **significados distintos** según su forma.

```
a = np.array([1, 2, 3])
b = np.array([[1, 2, 3]])

print(a.shape)
print(b.shape)
```

Salida:

```
(3,)
(1, 3)
```

Visualmente contienen los mismos valores, pero estructuralmente son distintos.

En IA, esta diferencia puede representar cosas tan distintas como:

- un vector de características,
- una observación única,
- una fila de un dataset.

Por eso, **no basta con mirar los valores**. Hay que mirar la forma.

3.6 Acceder a dimensiones y tamaños

NumPy te permite inspeccionar fácilmente la estructura de un array.

```
print("Dimensiones:", m.ndim)
print("Forma:", m.shape)
print("Número total de elementos:", m.size)
```

Estos atributos son herramientas básicas de verificación y se usarán constantemente a partir de ahora.

3.7 Primer contacto con arrays multidimensionales

NumPy no se limita a vectores y matrices. Puede representar datos en múltiples dimensiones.

```
t = np.array([
    [[1, 2], [3, 4]],
    [[5, 6], [7, 8]]
])

print(t)
print("Forma:", t.shape)
```

Salida:

```
Forma: (2, 2, 2)
```

No es importante todavía saber operar con estos datos, pero sí entender que:

> NumPy está diseñado para representar estructuras numéricas generales,
> no solo tablas bidimensionales.

Esto será clave más adelante, por ejemplo, en redes neuronales.

3.8 Errores típicos al empezar con `ndarray`

Algunos errores comunes que debes evitar:

- asumir que un vector y una matriz fila son lo mismo,
- no comprobar la forma antes de operar,
- confiar solo en el resultado numérico sin revisar la estructura,
- mezclar arrays de distinta forma sin entender cómo encajan.

Estos errores **no siempre generan excepciones**, pero sí resultados incorrectos.

3.9 Qué se fija en este bloque

Al terminar este bloque, debes tener claro que:

- un `ndarray` no es una lista,
- el tipo (`dtype`) forma parte del significado del dato,
- la forma (`shape`) define cómo se interpretan los valores,
- dos arrays con los mismos números pueden representar cosas distintas.

Este bloque prepara directamente el siguiente:

> cómo operar con datos respetando su estructuray por qué NumPy puede hacer cosas “automáticamente” que debes entender.

Referencias específicas — Bloque 2

- NumPy — The ndarray

  https://numpy.org/doc/stable/reference/arrays.ndarray.html
- NumPy — Array creation routines

  https://numpy.org/doc/stable/reference/routines.array-creation.html
- NumPy — Data types

  https://numpy.org/doc/stable/user/basics.types.html

---

4. Bloque 3 — Operar con datos: forma, tipo y significado

En los bloques anteriores has aprendido que, al trabajar con datos, **no basta con tener números**. La forma (`shape`) y el tipo (`dtype`) determinan **qué operaciones tienen sentido** y **cómo deben interpretarse los resultados**.

En este bloque vas a aprender a **operar con arrays NumPy de forma correcta**, entendiendo qué ocurre estructuralmente cuando sumas, multiplicas o combinas datos.

> A partir de aquí, el curso asume que no operas a ciegas:
>
> miras la forma, el tipo y el significado antes de confiar en el resultado.

4.1 Operaciones vectorizadas: el modelo mental correcto

En NumPy, las operaciones están diseñadas para aplicarse **sobre conjuntos completos**, no elemento a elemento mediante bucles.

Ejemplo básico:

```
import numpy as np

a = np.array([1, 2, 3], dtype=np.float64)
b = np.array([4, 5, 6], dtype=np.float64)

print("a:", a)
print("b:", b)
```

```
print("a + b:", a + b)
print("a * b:", a * b)
```

Estas operaciones:

- se aplican posición a posición,
- no requieren bucles,
- expresan directamente una operación matemática.

Este es el **modelo mental correcto** para trabajar con datos en IA.

4.2 Operaciones escalares: claridad y seguridad

Uno de los casos más habituales es operar un conjunto de datos con un escalar.

```
values = np.array([10, 20, 30, 40], dtype=np.float64)

print("Valores originales:", values)
print("Valores escalados:", values * 0.1)
```

Este patrón se usa constantemente para:

- escalado de variables,
- normalización simple,
- conversiones de unidades.

Aquí no hay ambigüedad:

el escalar se aplica a **todos los elementos del array**.

4.3 Broadcasting: cuando NumPy “encaja” las formas

NumPy permite operar arrays de distinta forma mediante un mecanismo llamado **broadcasting**.

Ejemplo:

```
m = np.array([
    [10, 20, 30],
```

```
    [40, 50, 60]
], dtype=np.float64)

offset = np.array([1, 2, 3], dtype=np.float64)

print("m:\n", m)
print("offset:", offset)
print("m + offset:\n", m + offset)
```

Aquí ocurre algo importante:

- `offset` tiene forma `(3,)`
- NumPy lo aplica **por columnas**
- la operación se ejecuta sin error

Este comportamiento es muy potente, pero también puede ser peligroso si no se entiende bien.

4.4 Broadcasting no es magia: la forma manda

El broadcasting **no adivina tu intención**. Solo sigue reglas estructurales.

Observa este caso:

```
row_offset = np.array([1, 2], dtype=np.float64)

print("m + row_offset:\n", m + row_offset)
```

Este código **lanza un error**, porque las formas no son compatibles.

Si quieres aplicar un desplazamiento por filas, debes hacerlo explícito:

```
row_offset = np.array([1, 2], dtype=np.float64).reshape(2, 1)
print("m + row_offset:\n", m + row_offset)
```

Conclusión clave:

> Si una operación depende de la forma,

> **la forma debe ser explícita**.

4.5 El eje (`axis`): sobre qué dimensión operas

Muchas operaciones de NumPy agregan valores: sumas, medias, máximos, etc.

El parámetro `axis` indica **sobre qué dimensión se realiza la operación**.

Ejemplo:

```
print("m:\n", m)
print("Suma total:", m.sum())
print("Suma por columnas:", m.sum(axis=0))
print("Suma por filas:", m.sum(axis=1))
```

Interpretación:

- `axis=0` → recorres filas, agregas columnas
- `axis=1` → recorres columnas, agregas filas

Este concepto es fundamental y reaparecerá constantemente.

4.6 `axis` y significado de los resultados

El mismo cálculo puede tener significados muy distintos según el eje.

```
print("Media por columnas:", m.mean(axis=0))
print("Media por filas:", m.mean(axis=1))
```

En un dataset real, esto puede representar:

- medias por variable (columnas),
- medias por observación (filas).

Confundir el eje **no rompe el código**, pero rompe el significado.

4.7 Errores silenciosos: el verdadero riesgo

Uno de los mayores peligros al trabajar con NumPy es que:

- el código se ejecuta,
- produce números,
- pero el resultado **no significa lo que crees**.

Ejemplo:

```
wrong = m - m.mean(axis=1)
print(wrong)
```

Este código funciona, pero el resultado **no es el centrado correcto por filas**.

La versión correcta es:

```
correct = m - m.mean(axis=1).reshape(-1, 1)
print(correct)
```

La diferencia está únicamente en la forma, pero el significado cambia por completo.

4.8 Estrategia defensiva al operar con datos

A partir de este bloque, se asume como norma que:

- revisas `shape` antes de operar,
- imprimes resultados intermedios cuando hay dudas,
- haces explícitas las dimensiones cuando el significado importa,
- no confías en una operación solo porque "funciona".

Esto no es exceso de precaución: es **trabajo riguroso con datos**.

4.9 Qué se fija en este bloque

Al terminar este bloque, debes tener claro que:

- las operaciones vectorizadas son la base del trabajo con datos,
- el broadcasting es potente pero peligroso,
- `axis` determina el significado de las agregaciones,

- muchos errores en IA son estructurales, no sintácticos.

Este bloque deja el terreno preparado para el siguiente:

> usar la visualización como herramienta para verificarque las operaciones tienen sentido.

Referencias específicas — Bloque 3

- NumPy — Broadcasting

  https://numpy.org/doc/stable/user/basics.broadcasting.html
- NumPy — Universal functions (ufuncs)

  https://numpy.org/doc/stable/reference/ufuncs.html
- NumPy — Reduction operations and axis

  https://numpy.org/doc/stable/reference/generated/numpy.sum.html

---

5. Bloque 4 — Visualización como verificación (antes de modelar)

En los bloques anteriores has aprendido a **representar datos** y a **operar con ellos**. A partir de aquí aparece una idea clave del curso:

> Antes de entrenar cualquier modelo,
>
> debes **ver** si los datos tienen sentido.

La visualización no se introduce como una herramienta decorativa ni como un añadido opcional. Se introduce como un **mecanismo de verificación técnica**, capaz de detectar errores que el código no detecta.

5.1 Por qué visualizar antes de modelar

Un modelo puede entrenarse sin errores y devolver métricas aparentemente razonables, incluso cuando:

- los datos están mal escalados,
- existen valores extremos dominantes,

- no hay relación real entre variables,
- el problema está mal planteado.

La visualización permite detectar estos problemas **antes** de invertir tiempo en entrenar modelos.

A partir de este punto del curso, se asume que:

> entrenar sin haber visualizado
>
> es una **decisión técnica deficiente**.

5.2 Qué valida la visualización que el código no valida

El código puede decirte:

- cuántos valores hay,
- qué tipo tienen,
- si hay valores nulos.

La visualización puede decirte cosas distintas:

- cómo se distribuyen realmente los valores,
- si hay asimetrías o colas largas,
- si existen outliers evidentes,
- si hay estructura o solo ruido.

Estas propiedades **no generan excepciones**, pero invalidan conclusiones si no se detectan.

5.3 Matplotlib como herramienta mínima

En esta sesión se usa Matplotlib como herramienta mínima de visualización. No para dominar la librería, sino para **controlar qué se está representando**.

Ejemplo básico:

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

```
x = np.array([1, 2, 3, 4, 5])
y = np.array([1, 4, 9, 16, 25])

plt.figure()
plt.plot(x, y)
plt.xlabel("x")
plt.ylabel("y")
plt.title("Relación entre x e y")
plt.show()
```

Si no sabes qué representa cada eje, **no sabes qué estás mirando**.

5.4 Distribución de una variable: el primer chequeo obligatorio

Antes de usar una variable numérica, debes ver **cómo se distribuye**.

```
rng = np.random.default_rng(0)
values = rng.normal(loc=0.0, scale=1.0, size=1000)

print("min:", values.min())
print("max:", values.max())
print("mean:", values.mean())

plt.figure()
plt.hist(values, bins=30)
plt.xlabel("valor")
plt.ylabel("frecuencia")
plt.title("Distribución de la variable")
plt.show()
```

Este gráfico permite detectar:

- ausencia de variabilidad,
- sesgo pronunciado,
- rangos inesperados,

- valores extremos.

5.5 Valores extremos: cuándo los números engañan

Un pequeño número de valores extremos puede dominar completamente una variable.

```
values[0] = 10
values[1] = -9

plt.figure()
plt.hist(values, bins=30)
plt.title("Distribución con valores extremos")
plt.show()
```

Aunque las estadísticas sigan siendo “razonables”, la visualización muestra que:

- la masa principal queda aplastada,
- muchos modelos se verán afectados,
- el preprocesamiento será necesario.

5.6 Relación entre variables: ¿hay señal o solo ruido?

Además de ver variables individualmente, debes observar **cómo se relacionan**.

```
x = rng.uniform(0, 10, size=300)
y = 2 * x + rng.normal(0, 2, size=300)

plt.figure()
plt.scatter(x, y)
plt.xlabel("x")
plt.ylabel("y")
plt.title("Relación entre x e y")
plt.show()
```

Este gráfico permite evaluar si:

- existe relación visible,
- la relación es aproximadamente lineal,
- un modelo simple tiene sentido.

5.7 Cuando no hay relación (y eso también es información)

No todos los datasets contienen señal útil.

```
x = rng.normal(0, 1, size=300)
y = rng.normal(0, 1, size=300)

plt.figure()
plt.scatter(x, y)
plt.xlabel("x")
plt.ylabel("y")
plt.title("Ausencia de relación")
plt.show()
```

Si no hay relación visible:

- entrenar un modelo no resolverá el problema,
- el dataset o el objetivo deben replantearse.

5.8 Errores comunes al interpretar gráficos

Al visualizar, debes evitar errores habituales:

- confundir correlación con causalidad,
- ignorar escalas de los ejes,
- ver patrones donde solo hay ruido,
- sacar conclusiones a partir de pocos puntos,
- confiar en un único gráfico.

La visualización **no decide**, orienta.

5.9 Qué se fija en este bloque

Al terminar este bloque, debes asumir que:

- visualizar es parte del trabajo técnico,
- ningún modelo se entrena sin haber visto los datos,
- los gráficos sirven para detectar problemas, no para decorar informes,
- interpretar visualmente requiere pensamiento crítico.

Este bloque cierra la Sesión 2 dejando claro que:

> sin datos bien entendidos, no hay aprendizaje automático fiable.

Referencias específicas — Bloque 4

- Matplotlib — Pyplot tutorial

  https://matplotlib.org/stable/tutorials/introductory/pyplot.html
- Matplotlib — Histograms

  https://matplotlib.org/stable/gallery/statistics/hist.html
- Python Data Science Handbook — Visualization with Matplotlib

  https://jakevdp.github.io/PythonDataScienceHandbook/04.00-Visualization-With-Matplotlib.html

---

6. Cierre de la Sesión 2

En esta sesión has dejado de trabajar con Python como un lenguaje “general” y has empezado a trabajar con **datos como objetos estructurados**.

Has visto que, en el contexto de la Inteligencia Artificial, no basta con que el código funcione. Es necesario entender:

- **qué representan los datos**,
- **cómo están organizados**,
- **qué significado tienen las operaciones**,
- y **qué información se puede extraer antes de modelar**.

A lo largo de la sesión has construido un hilo claro:

- las listas y los bucles no son el modelo mental adecuado para trabajar con datos,
- NumPy introduce una representación numérica coherente y explícita,
- la forma (`shape`) y el tipo (`dtype`) forman parte del significado del dato,
- las operaciones vectorizadas y el uso de `axis` determinan qué se está calculando realmente,
- la visualización permite verificar supuestos que el código no valida.

6.1 Qué se asume a partir de ahora en el curso

A partir de esta sesión, el curso **asume como base** que:

- sabes distinguir entre una lista y un `ndarray`,
- compruebas la forma de los datos antes de operar,
- entiendes que el tipo numérico no es un detalle menor,
- no confías en un resultado solo porque "sale un número",
- visualizas los datos antes de entrenar modelos.

Estos supuestos **no se volverán a justificar desde cero**. Se usarán como punto de partida.

6.2 El cambio de rol: de escribir código a razonar con datos

Hasta ahora, gran parte del aprendizaje consistía en escribir código correcto.

A partir de aquí, el foco cambia:

> El objetivo ya no es solo que el código funcione,
>
> sino que **lo que hace tenga sentido desde el punto de vista del dato**.

Esto implica:

- justificar decisiones,
- detectar errores silenciosos,

- interpretar resultados,
- y ser capaz de explicar por qué un enfoque es razonable o no.

Este cambio de rol es fundamental para todo lo que viene después.

6.3 Por qué esta sesión es clave para el resto del curso

Todo el contenido posterior del curso se apoya directamente en lo que has visto aquí:

- el análisis exploratorio de datos,
- el preprocesamiento,
- la elección de modelos,
- la evaluación de resultados,
- y la interpretación de errores.

Sin una comprensión clara de:

- cómo se representan los datos,
- cómo se operan,
- y cómo se verifican,

el aprendizaje automático se convierte en una sucesión de recetas frágiles.

6.4 Conexión con la siguiente sesión

En esta sesión has trabajado con:

- arrays NumPy explícitos,
- ejemplos controlados,
- visualización manual y deliberada.

En la siguiente sesión darás un paso natural:

> trabajarás con datasets reales,
>
> estructurados en tablas,
>
> y con herramientas que automatizan parte del análisis exploratorio.

Para hacerlo con rigor, necesitarás exactamente lo que has aprendido aquí:

criterio sobre datos, no solo herramientas.

6.5 Idea clave para llevarte de esta sesión

Si tienes que quedarte con una sola idea de esta sesión, es esta:

> En Inteligencia Artificial,
> **los errores más graves no suelen dar excepciones**,
> dan resultados plausibles pero incorrectos.

Por eso:

- entender los datos,
- verificar su estructura,
- y visualizarlos antes de modelar

no es una opción, es una condición necesaria.