


# Sesión 2: Sistemas de Ecuaciones y Transformaciones Matriciales

El Aprendizaje de Máquina (Machine Learning) se basa en gran medida en la manipulación de datos y la construcción de modelos predictivos, procesos que se fundamentan en la álgebra lineal y la resolución de sistemas de ecuaciones.

 **por Kibernet Capacitación S.A.**

# Introducción al Álgebra Lineal en Machine Learning

El álgebra lineal constituye una de las bases fundamentales del Aprendizaje de Máquina. En particular, la resolución de sistemas de ecuaciones lineales permite ajustar modelos como la Regresión Lineal, mientras que las transformaciones matriciales se aplican para manipular y optimizar datos en diferentes contextos (normalización, rotaciones, cambios de base, etc.).

Ahora profundizaremos en los conceptos teóricos y prácticos que permiten comprender cómo se representan, transforman y resuelven problemas de manera matricial, así como su relevancia en la optimización de modelos.

# Sistemas de Ecuaciones Lineales

Estos sistemas aparecen de forma natural en diversos algoritmos de Aprendizaje de Máquina, especialmente en métodos de regresión y en la estimación de parámetros de modelos. Aprenderemos a plantear y resolver sistemas lineales mediante diferentes enfoques, desde la inversa de matrices hasta métodos numéricos especializados. Se revisarán técnicas y buenas prácticas de la biblioteca NumPy en Python para automatizar estos cálculos de manera eficiente.

Un sistema de ecuaciones lineales se puede expresar en forma matricial como:

$$Ax = b$$

donde  $A$  es la matriz de coeficientes,  $x$  el vector de incógnitas y  $b$  el vector de términos independientes.

En Aprendizaje de Máquina, muchos problemas pueden formularse como sistemas de ecuaciones lineales. Por ejemplo, la Regresión Lineal busca encontrar los parámetros que satisfagan la ecuación:

$$X\beta = y$$

donde:

- $X$  es la matriz de características (filas = muestras, columnas = variables),
- $\beta$  es el vector de parámetros,
- $y$  es el vector de valores objetivo.

# Resolución de Sistemas con NumPy

En Python, la librería NumPy (módulo `numpy.linalg`) proporciona funciones para resolver estos sistemas:

1. `np.linalg.solve(A, b)`
  - Diseñada para sistemas cuadrados (misma cantidad de ecuaciones y variables).
  - Retorna la solución exacta si la matriz A es invertible.
2. `np.linalg.lstsq(A, b, rcond=None)`
  - Se utiliza en casos sobre-determinados o sub-determinados.
  - Devuelve la solución en el sentido de mínimos cuadrados, muy útil en problemas de regresión lineal.

Ejemplo:

```
Es posible que el código generado esté sujeto a una licencia | aerpenbeck/binder-numpy-test
import numpy as np

# Ejemplo de sistema 2x2
A = np.array([[2, 1],
              [1, 3]])

b = np.array([8, 13])
solucion = np.linalg.solve(A, b)

print("Solución del sistema:", solucion)
```

⇒ Solución del sistema: [2.2 3.6]

Aplicación en Aprendizaje de Máquina:

En Regresión Lineal, la solución de mediante la fórmula  $\beta = (X^T X)^{-1} X^T y$  o `np.linalg.lstsq` es un caso particular de resolución de sistemas lineales.

# Métodos de Resolución con NumPy

## np.linalg.inv y la ecuación

$$(X^T X)^{-1} X^T y$$

Cuando X tiene más filas que columnas (sobre-determinado) y es de rango completo, se puede usar la aproximación de mínimos cuadrados:

```
Es posible que el código generado esté sujeto a una licencia | aerpenbeck/binder-numpy-test
import numpy as np

# Ejemplo de sistema lineal
X = np.array([[1, 1],
              [1, 2],
              [1, 3],
              [1, 4]])

y = np.array([2, 3, 4, 5])

# Calculo de beta con la fórmula analítica de mínimos cuadrados
beta = np.linalg.inv(X.T @ X) @ X.T @ y

print("Parámetros beta:\n", beta)
```

Parámetros beta:  
[1. 1.]

## np.linalg.solve

Para sistemas cuadrados (misma cantidad de ecuaciones y variables), se puede usar np.linalg.solve(A, b):

```
Es posible que el código generado esté sujeto a una licencia | aerpenbeck/binder-numpy-test
import numpy as np

# Sistema A*x = b

A = np.array([[2, 1],
              [1, 3]])

b = np.array([8, 13])

solucion = np.linalg.solve(A, b)

print("Solución del sistema:\n", solucion)
```

Solución del sistema:  
[2.2 3.6]

Para sistemas sobre-determinados (más ecuaciones que variables) o sub-determinados (menos ecuaciones que variables), np.linalg.lstsq(X, y, rcond=None) encuentra la solución en el sentido de mínimos cuadrados.

```
Es posible que el código generado esté sujeto a una licencia | aerpenbeck/binder-numpy-test
import numpy as np

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

y = np.array([2, 3, 4, 5])

# Usando lstsq
beta_lstsq, residuals, rank, s = np.linalg.lstsq(X, y, rcond=None)

print("Parámetros beta usando lstsq:\n", beta_lstsq)
```

Parámetros beta usando lstsq:  
[1. 1.]

# Valores y Vectores Propios

Los valores y vectores propios proporcionan una descomposición fundamental de las matrices, permitiendo comprender la estructura interna de los datos. Estos conceptos son cruciales para técnicas como el Análisis de Componentes Principales (PCA), que reduce la dimensionalidad de los datos sin perder información esencial.

Los valores propios ( $\lambda$ ) y vectores propios ( $v$ ) de una matriz  $A$  se definen como:

$$Av = \lambda v$$

- El valor propio  $\lambda$  indica el factor de estiramiento o compresión en la dirección de  $v$ .
- El vector propio  $v$  define esa dirección invariante bajo la transformación de  $A$ .

En Machine Learning, se usan principalmente en:

- PCA (Análisis de Componentes Principales): para la reducción de la dimensionalidad.
- Análisis de estabilidad de algoritmos (ej., en optimización).
- Interpretación geométrica de transformaciones.

# Cálculo de Valores y Vectores Propios con NumPy

```
Es posible que el código generado esté sujeto a una licencia | aerpenbeck/binder-numpy-test
import numpy as np

# Definimos una matriz cuadrada
M = np.array([[2, 1],
              [1, 2]])

# Cálculo de valores y vectores propios
valores_propios, vectores_propios = np.linalg.eig(M)

print("Valores propios:\n", valores_propios)
print("Vectores propios:\n", vectores_propios)
```

Valores propios:  
[3. 1.]  
Vectores propios:  
[[ 0.70710678 -0.70710678]  
[ 0.70710678 0.70710678]]

Interpretación Geométrica Básica:

- Cada vector propio define una dirección en la cual la transformación lineal (representada por  $M$ ) actúa como un simple factor de escala  $\lambda$ .
- El valor propio  $\lambda$  indica el factor de estiramiento o compresión en esa dirección.

# Transformaciones Matriciales

Las matrices no solo representan datos, sino que también describen transformaciones en el espacio. Aprenderemos a modelar rotaciones, escalados y cambios de base, y cómo estas transformaciones son utilizadas en la normalización y preprocesamiento de datos.

Una transformación lineal puede verse como la aplicación de una Matriz A sobre un vector x, generando un nuevo vector y:

$$y = Ax$$

En 2D, estas transformaciones pueden corresponder a rotaciones, escalados, reflexiones, etc.

## Representación de Transformaciones con NumPy

Rotación en 2D

La matriz de rotación por un ángulo  $\theta$  en el plano es:

$$R(\theta) = \begin{pmatrix} \cos \theta & -\sin \theta \\ \sin \theta & \cos \theta \end{pmatrix}.$$

```
Es posible que el código generado esté sujeto a una licencia | aerpenbeck/binder-numpy-test
import numpy as np
import matplotlib.pyplot as plt

theta = np.radians(45) # Rotar 45 grados
R = np.array([[np.cos(theta), -np.sin(theta)],
              [np.sin(theta), np.cos(theta)]])

# Un vector inicial
v = np.array([1, 0])

# Transformar el vector
v_rotado = R @ v

print("Vector original:", v)
print("Vector rotado 45°:", v_rotado)
```

Vector original: [1 0]  
Vector rotado 45°: [0.70710678 0.70710678]



# Visualización de Transformaciones

Con Matplotlib se puede visualizar cómo un conjunto de puntos se transforma:

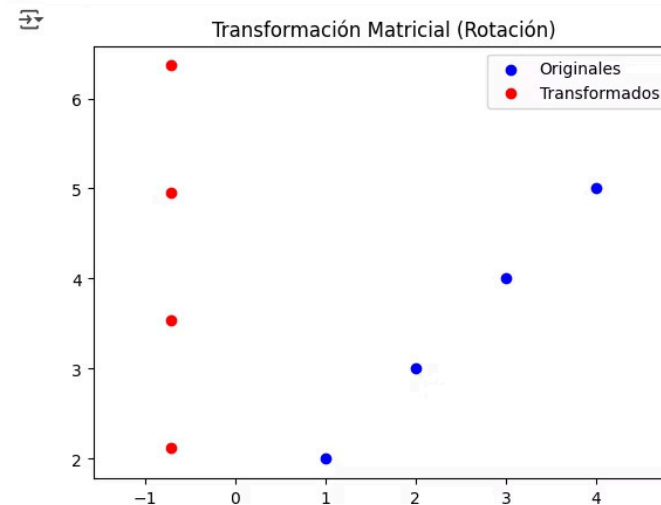
```
Es posible que el código generado esté sujeto a una licencia | aerpenbeck/binder-numpy-test
import numpy as np
import matplotlib.pyplot as plt

theta = np.radians(45) # Rotar 45 grados
R = np.array([[np.cos(theta), -np.sin(theta)],
              [np.sin(theta), np.cos(theta)]])

# Conjunto de puntos
puntos = np.array([[1, 2],
                   [2, 3],
                   [3, 4],
                   [4, 5]]).T # Transpuesta para tener shape (2, n)

puntos_transformados = R @ puntos

# Graficar
plt.scatter(puntos[0, :], puntos[1, :], color='blue', label='Originales')
plt.scatter(puntos_transformados[0, :], puntos_transformados[1, :], color='red', label='Transformados')
plt.legend()
plt.axis('equal')
plt.title('Transformación Matricial (Rotación)')
plt.show()
```



Aplicaciones en Aprendizaje de Máquina:

- Normalización y Estandarización: Transformaciones de datos para mejorar la eficiencia de algoritmos.
- Cambios de Base: PCA, donde se diagonaliza la matriz de covarianza para encontrar las direcciones principales.
- Optimización: Algunas técnicas de optimización aplican transformaciones (precondicionamiento) para mejorar la convergencia de algoritmos de entrenamiento.

# Actividad Práctica Guiada

Objetivo:

- Resolver un sistema de ecuaciones lineales en Python y visualizar una transformación de datos en 2D.

Paso a Paso

1. Resolver un Sistema de Ecuaciones
  - Generar una matriz A de 3x3 y un vector b de 3 elementos.
  - Utilizar `np.linalg.solve(A, b)` para obtener la solución.

```
Es posible que el código generado esté sujeto a una licencia | aerpenbeck/binder-numpy-test
import numpy as np

A = np.array([[3, 2, -1],
              [2, -2, 4],
              [-1, 0.5, -1]])

b = np.array([1, -2, 0])

x = np.linalg.solve(A, b)

print("Solución del sistema:\n", x)
```

⇒ Solución del sistema:  
[ 1. -2. -2.]

2. Transformación 2D

- Crear una matriz de transformación (rotación, escalado o ambas).
- Definir un conjunto de puntos y aplicar la transformación.
- Visualizar el resultado con Matplotlib.

Es posible que el código generado esté sujeto a una licencia | aerpenbeck/binder-numpy-test

```
import matplotlib.pyplot as plt

# Matriz de escalado y rotación
scale = 1.5
theta = np.radians(30)
T = np.array([[scale*np.cos(theta), -scale*np.sin(theta)],
              [scale*np.sin(theta), scale*np.cos(theta)]])

# Puntos originales
pts = np.array([[1, 1],
                [2, 1],
                [2, 2],
                [1, 2]]).T # 2x4

# Transformación
pts_transformed = T @ pts

# Visualización
plt.scatter(pts[0, :], pts[1, :], color='blue', label='Originales')
plt.scatter(pts_transformed[0, :], pts_transformed[1, :], color='red', label='Transformados')
plt.axis('equal')
plt.legend()
plt.title("Transformación 2D (Escalado + Rotación)")
plt.show()
```

