Módulo 1 : NumPy Capstone

Filosofía: Esta es la prueba final. Aquí no se trata de aprender una función, sino de *diseñar una solución* combinando todo lo aprendido (Módulo 0 y Módulo 1) para resolver problemas de data science del mundo real.

1. Desafío: Broadcasting Avanzado (K-Means "desde cero")

Escenario: Tienes 100 puntos de datos en 2D (shape (100, 2)) y 3 "centroides" de clusters (shape (3, 2)).
El Reto: Calcular una matriz de distancias de (100, 3) donde dist[i, j] sea la distancia euclidiana entre el punto i y el centroide j. Luego, asignar cada punto a su centroide más cercano. **Todo esto sin un solo bucle for**.

```
import numpy as np

print("--- 1. Desafío: Broadcasting Avanzado (K-Means) ---")
np.random.seed(42)

# 1. Generar datos
n_puntos = 100
n_features = 2 # 2D
n_clusters = 3

puntos = np.random.rand(n_puntos, n_features) * 10
centroides = np.random.rand(n_clusters, n_features) * 10

print(f"Shape Puntos: {puntos.shape}")
print(f"Shape Centroides: {centroides.shape}")

# 2. El truco: Expandir dimensiones para el Broadcasting
#    Queremos restar (100, 2) de (3, 2) de forma "cruzada".
#    puntos -> (100, 1, 2)
#    centroides -> (1, 3, 2)
puntos_exp = puntos[:, np.newaxis, :]
centroides_exp = centroides[np.newaxis, :, :]

print(f"\nShape Puntos (expandido): {puntos_exp.shape}")
print(f"Shape Centroides (expandido): {centroides_exp.shape}")

# 3. Restar (Broadcasting mágico)
#    NumPy "estira" ambos arrays para que coincidan
```

```
#    El resultado es (100, 3, 2)
#    Donde [i, j, :] es el vector de diferencia entre punto i y centroide j
vectores_distancia = puntos_exp - centroides_exp
print(f"\nShape Vectores Distancia: {vectores_distancia.shape}")

# 4. Calcular distancias euclidianas
#    Dist = sqrt(sum(diff^2))
dist_sq = np.sum(vectores_distancia**2, axis=2) # Sumar en el eje de features (axis=2)
distancias = np.sqrt(dist_sq)
print(f"Shape Matriz de Distancias: {distancias.shape}") # (100, 3)

# 5. Asignar cada punto al cluster más cercano
#    `np.argmin` sobre el eje 1 (los clusters)
asignacion_clusters = np.argmin(distancias, axis=1)

print(f"\nShape Asignaciones: {asignacion_clusters.shape}")
print("\nAsignación de los primeros 10 puntos:")
print(asignacion_clusters[:10]) # [2, 0, 1, 0, 0, 2, 2, 0, 1, 0]

# 6. (Extra) Calcular la media de los puntos por cluster (Módulo 0)
for k in range(n_clusters):
    mask_cluster = (asignacion_clusters == k)
    puntos_en_cluster = puntos[mask_cluster]
    print(f"\nCluster {k}: {len(puntos_en_cluster)} puntos")
    if len(puntos_en_cluster) > 0:
        print(f"  Centroide original: {centroides[k]}")
        print(f"  Centroide (calculado): {np.mean(puntos_en_cluster, axis=0)}")
```

2. Desafío: Apilado hstack (One-Hot Encoding Manual)

Escenario: Tienes un dataset mixto. Una matriz numérica de (10, 3) (edad, altura, peso) y un array categórico (10,) (país: 'ES', 'MX', 'AR').
El Reto: Convertir el array categórico a One-Hot Encoding (OHE) y "pegarlo" (apilarlo horizontalmente) a la matriz numérica para crear una matriz de features final (10, 6).

```
import numpy as np

print("\n--- 2. Desafío: Apilado `hstack` (One-Hot Encoding) ---")
np.random.seed(42)

# 1. Datos de entrada
datos_numericos = np.random.randint(18, 60, size=(10, 3))
datos_categoricos = np.random.choice(['ES', 'MX', 'AR'], size=10)
```

```
print("Datos Numéricos:\n", datos_numericos[:5])
print("\nDatos Categóricos:\n", datos_categoricos[:5])

# 2. Identificar categorías únicas (Módulo 0)
categorias = np.unique(datos_categoricos)
n_categorias = len(categorias)
print(f"\nCategorías: {categorias}") # ['AR', 'ES', 'MX']

# 3. Crear el mapeo (Módulo 0)
cat_to_index = {cat: i for i, cat in enumerate(categorias)}

# 4. Crear la matriz OHE (Fancy Indexing)
indices = np.array([cat_to_index[cat] for cat in datos_categoricos])
matriz_ohe = np.zeros((len(datos_categoricos), n_categorias), dtype=int)
matriz_ohe[np.arange(len(datos_categoricos)), indices] = 1

print("\nMatriz OHE:\n", matriz_ohe[:5])

# 5. La operación "zip" (hstack)
#    np.hstack apila arrays horizontalmente (por columnas)
#    Tienen que tener el mismo número de filas
matriz_final = np.hstack((datos_numericos, matriz_ohe))

print(f"\nShape Matriz Final: {matriz_final.shape}")
print("Matriz Final (head 5):\n", matriz_final[:5])
# (Col 0,1,2 = numéricas; Col 3,4,5 = OHE de AR, ES, MX)
```

3. Desafío: Apilado column_stack (Feature Engineering en TimeSeries)

Escenario: Tienes un array de timestamps (N,) y un array de valores de sensor (N,).
El Reto: Crear una matriz de features (N, 4) donde las columnas sean: [valor, hora_del_dia, dia_de_semana, es_finde]. Esta es la operación "zip" más común en Data Science.

```
import numpy as np

print("\n--- 3. Desafío: Apilado `column_stack` (TimeSeries FE) ---")
np.random.seed(42)

# 1. Generar datos base
timestamps = np.arange('2024-01-01', '2024-01-04', dtype='datetime64[h]')
N = len(timestamps)
```

```
valores = np.random.normal(100, 5, N) + (np.arange(N) * 0.1) # Tendencia

print(f"Generados {N} registros.")
print(f"Valores (head): {valores[:5]}")

# 2. Feature Engineering (Módulo 1)
#    Extraer features temporales
dias_desde_epoch = timestamps.astype('datetime64[D]').astype(int)
dia_semana = (dias_desde_epoch + 3) % 7 # 0=Lunes
hora_dia = timestamps.astype('datetime64[h]').astype(int) % 24
es_finde = (dia_semana >= 5).astype(int) # 1 si finde, 0 si no

# 3. La operación "zip" (column_stack)
#    np.column_stack toma una lista de arrays 1D (N,) y los
#    convierte en columnas de una matriz 2D (N, k)
matriz_features = np.column_stack((
   valores,
   hora_dia,
   dia_semana,
   es_finde
))

print(f"\nShape Matriz Features: {matriz_features.shape}")
print("\nMatriz de Features (head 5):")
print("[Valor, Hora, DiaSem, EsFinde]")
print(matriz_features[:5])
```

4. Desafío: La "Operación Secreta" (np.einsum)

Escenario: Tienes un "batch" de 10 matrices (10, 5, 4) y un "batch" de 10 vectores (10, 4). Quieres hacer 10 productos "matriz-vector" independientes.
El Reto: Hacerlo en una sola línea de código usando np.einsum (Suma de Einstein), la herramienta más densa pero potente de NumPy para álgebra tensorial.

```
import numpy as np

print("\n--- 4. Desafío: `np.einsum` (Batch Matrix-Vector) ---")
np.random.seed(42)

B = 10 # Tamaño del Batch
N = 5
M = 4
```

```
# 1. Datos
batch_matrices = np.random.rand(B, N, M) # (10, 5, 4)
batch_vectores = np.random.rand(B, M)    # (10, 4)

# 2. La solución con `np.einsum`
#    'bnm,bm->bn'
#    b = eje de batch, n = eje de filas, m = eje de columnas
#    Le decimos:
#    1. Coge el array 'bnm' (matrices)
#    2. Coge el array 'bm' (vectores)
#    3. Multiplica y suma sobre el eje 'm' (el eje común)
#    4. Devuélveme un array 'bn'
resultado_einsum = np.einsum('bnm,bm->bn', batch_matrices, batch_vectores)

print(f"Shape Resultado (einsum): {resultado_einsum.shape}") # (10, 5)

# 3. Verificación (con bucle de Python)
#    Así es como lo haríamos "a mano"
resultado_check = np.zeros((B, N))
for i in range(B):
   # (5, 4) @ (4,) -> (5,)
   resultado_check[i] = np.dot(batch_matrices[i], batch_vectores[i])

print(f"Shape Resultado (check): {resultado_check.shape}")

# 4. Comprobar que son idénticos
print(f"\n¿Son idénticos los resultados? {np.allclose(resultado_einsum, resultado_check)}")

# ---
print("\nEjemplo 2 `einsum`: Traza de un batch (suma de diagonales)")
# 'bnn->b' -> Coge un batch de matrices (b,n,n) y suma por la diagonal (n=n)
matrices_cuadradas = np.random.rand(B, 3, 3)
trazas = np.einsum('bnn->b', matrices_cuadradas)
print(f"Trazas (shape {trazas.shape}):\n {trazas}")

# Verificación
trazas_check = np.trace(matrices_cuadradas, axis1=1, axis2=2)
print(f"¿Son idénticas las trazas? {np.allclose(trazas, trazas_check)}")
```

5. Desafío Final: Procesamiento de Imágenes (Batch

Masking)

Escenario: Eres un Data Scientist de IA. Tienes un "batch" de 10 imágenes a color de 128x128 píxeles. Quieres aplicar una máscara (un "filtro" circular) a *todo* el batch de golpe para aumentar tu dataset.
El Reto: Crear un batch 4D (10, 128, 128, 3) de imágenes sintéticas. Luego, crear *una* sola máscara 2D (128, 128) y usar la magia del broadcasting para aplicarla a las 10 imágenes (y sus 3 canales) de una sola vez.

```
import numpy as np
# (matplotlib es solo para la verificación visual, no es parte del reto)
# import matplotlib.pyplot as plt

print("\n--- 5. Desafío Final: Batch de Imágenes y Broadcasting ---")
np.random.seed(42)

# 1. Definir dimensiones
B = 10  # n_imagenes (Batch size)
H = 128 # Altura (Height)
W = 128 # Ancho (Width)
C = 3   # Canales (R, G, B)

# 2. Generar el batch de imágenes sintéticas
#    `uint8` es el tipo de dato ESTÁNDAR para imágenes
#    Crearemos un fondo azul con un cuadrado verde en el centro
images_batch = np.zeros((B, H, W, C), dtype=np.uint8)

# Fondo azul (R=0, G=0, B=255)
images_batch[..., 2] = 255

# Cuadrado verde (R=0, G=255, B=0)
# Usamos slicing para seleccionar el centro de TODAS las imágenes del batch
images_batch[:, 32:96, 32:96, 1] = 255

print(f"Shape del Batch de Imágenes: {images_batch.shape}")
print(f"Tipo de dato: {images_batch.dtype}")

# 3. Crear UNA máscara 2D (la operación "chula")
#    Usamos `np.meshgrid` para crear un "mapa de coordenadas"
x = np.linspace(-1, 1, W) # Coordenadas X de -1 a 1
y = np.linspace(-1, 1, H) # Coordenadas Y de -1 a 1
xx, yy = np.meshgrid(x, y)
```

```
# Calcular la distancia al centro (Teorema de Pitágoras)
# dist = sqrt(x^2 + y^2)
distancia_centro = np.sqrt(xx**2 + yy**2)

# Crear una máscara circular (todo lo que esté a > 1 de radio, es 0)
# (Lo escalamos a float 0.0-1.0 para que sea un "filtro")
mascara_2d = 1.0 - distancia_centro
mascara_2d[mascara_2d < 0] = 0.0 # Clip

print(f"\nShape de la Máscara 2D: {mascara_2d.shape}")
print(f"Tipo de dato: {mascara_2d.dtype}")

# 4. El "Ostión": Aplicar la máscara 2D al Batch 4D
#    Shape de Imágenes: (10, 128, 128, 3)
#    Shape de Máscara:   (128, 128)
#
#    ¡No son compatibles! Necesitamos "alinear los ejes".
#    Queremos que la máscara se aplique a cada canal (C)
#    y a cada imagen (B) por igual.

#    Expandimos la máscara a (1, 128, 128, 1)
mascara_expandida = mascara_2d[np.newaxis, :, :, np.newaxis]
print(f"\nShape Máscara Expandida: {mascara_expandida.shape}")

# 5. La operación de Broadcasting
#    NumPy ve: (10, 128, 128, 3) * (1, 128, 128, 1)
#    "Estira" el eje 0 (batch) de 1 a 10
#    "Estira" el eje 3 (canal) de 1 a 3
#    ...y realiza 10*128*128*3 multiplicaciones.
#    (Convertimos a float para la multiplicación y luego a uint8)

imagenes_filtradas = images_batch.astype(np.float32) * mascara_expandida
imagenes_filtradas = imagenes_filtradas.astype(np.uint8)

print("\n¡Operación de broadcasting completada!")
print(f"Shape final: {imagenes_filtradas.shape}")

# 6. Verificación
#    Comprobamos que las esquinas de la primera imagen están a 0
print("\nVerificación de la Imagen 0:")
print("Esquina Superior Izq (R,G,B):", imagenes_filtradas[0, 0, 0, :])
print("Esquina Inferior Der (R,G,B):", imagenes_filtradas[0, -1, -1, :])
print("Centro de la Imagen (R,G,B):", imagenes_filtradas[0, 64, 64, :]) # Debe ser (0, 255, 0)
```

```
                                        # (porque máscara=1 en el centro)

# --- Visualización (Opcional, si tienes matplotlib) ---
# fig, ax = plt.subplots(1, 2)
# ax[0].imshow(images_batch[0])
# ax[0].set_title("Imagen Original [0]")
# ax[1].imshow(imagenes_filtradas[0])
# ax[1].set_title("Imagen Filtrada [0]")
# plt.show()
```