Módulo 1 (Desafíos): Taller Avanzado NumPy

Filosofía: El poder real no está en saber np.mean(). Está en saber *qué* media calcular. Estos desafíos combinan la generación y manipulación de arrays de NumPy con la lógica de Python (fechas, lambdas) para resolver problemas complejos de análisis.

1. Desafío: Simulación de Monte Carlo (Cálculo de Pi)

Escenario: Usar la fuerza bruta de NumPy para estimar el valor de Pi.
La idea es generar millones de puntos aleatorios (x, y) dentro de un cuadrado de 1x1 y contar cuántos caen dentro de un cuarto de círculo de radio 1.
La proporción (puntos_dentro / puntos_totales) es una aproximación de Pi / 4.

```
import numpy as np

print("--- 1. Desafío: Simulación de Monte Carlo (Pi) ---")
np.random.seed(42)

# Cuantos más puntos, más precisión (y más uso de NumPy)
n_puntos = 10_000_000

# 1. Generar TODOS los puntos (x, y) a la vez.
#    Generamos una matriz de 10M de filas x 2 columnas.
#    Valores uniformes entre 0 y 1.
puntos = np.random.uniform(low=0.0, high=1.0, size=(n_puntos, 2))
print(f"Shape de los puntos: {puntos.shape}")

# 2. Extraer 'x' e 'y' usando slicing de columnas
x = puntos[:, 0]
y = puntos[:, 1]

# 3. Calcular la distancia al origen (Hipotenusa: sqrt(x^2 + y^2))
#    Esta operación está 100% vectorizada.
distancia = np.sqrt(x**2 + y**2)
print(f"Shape de las distancias: {distancia.shape}")

# 4. Crear una máscara booleana: ¿está dentro del círculo? (dist <= 1)
mask_dentro_circulo = (distancia <= 1.0)

# 5. Contar cuántos 'True' hay.
#    np.sum() trata True como 1 y False como 0.
```

```
puntos_dentro = np.sum(mask_dentro_circulo)
print(f"Puntos totales: {n_puntos}")
print(f"Puntos dentro: {puntos_dentro}")

# 6. Calcular Pi
pi_estimado = 4 * (puntos_dentro / n_puntos)
print(f"\nPi Estimado (con {n_puntos} puntos): {pi_estimado}")
print(f"Valor Real de Pi: {np.pi}")
```

2. Desafío: Agregación de TimeSeries con np.datetime64

Escenario: Tienes datos de un sensor cada hora durante un año. Quieres comparar la temperatura media de los "días laborables" (L-V) frente a los "fines de semana" (S-D). Esto requiere manipular el tipo datetime64 de NumPy.

```
import numpy as np

print("\n--- 2. Desafío: Agregación de TimeSeries (datetime64) ---")
np.random.seed(42)

# 1. Generar un array de timestamps (uno por hora, para todo 2024)
#    np.datetime64 es el 'datetime' de NumPy.
#    'h' es el código de "hora".
timestamps = np.arange('2024-01-01', '2025-01-01', dtype='datetime64[h]')
n_horas = len(timestamps)
print(f"Generadas {n_horas} horas de datos.")

# 2. Generar datos de sensor (paseo aleatorio + ruido estacional)
#    Simulamos un ciclo diario (más frío por la noche)
horas_del_dia = np.arange(n_horas) % 24
ciclo_diario = -np.cos(horas_del_dia * (2 * np.pi / 24)) * 5 # Amplitud de 5 grados
paseo = np.random.normal(0, 0.2, n_horas).cumsum()
temperaturas = 15 + paseo + ciclo_diario # Temp base 15
print(f"Shape de temperaturas: {temperaturas.shape}")

# 3. (El truco) Extraer el día de la semana
#    No hay un `.weekday()` vectorizado. Lo hacemos "a lo data scientist":
#    Convertimos a 'días' ('D') y usamos aritmética modular.
#    El "Epoch" de NumPy (1970-01-01) fue un Jueves (día 3).
dias_desde_epoch = timestamps.astype('datetime64[D]').astype(int)
dia_semana = (dias_desde_epoch + 3) % 7 # 0=Lunes, 1=Martes, ... 6=Domingo
```

```
print(f"Primeros días de la semana: {dia_semana[:10]} (0=Lunes)")

# 4. Crear máscaras booleanas
mask_laborables = (dia_semana < 5) # Días 0-4
mask_findesemana = (dia_semana >= 5) # Días 5-6

# 5. Aplicar máscaras y calcular agregados
media_temp_laborables = np.mean(temperaturas[mask_laborables])
media_temp_findesemana = np.mean(temperaturas[mask_findesemana])
max_temp_laborables = np.max(temperaturas[mask_laborables])
max_temp_findesemana = np.max(temperaturas[mask_findesemana])

print(f"\nTemp. Media (L-V): {media_temp_laborables:.2f}°C")
print(f"Temp. Media (S-D): {media_temp_findesemana:.2f}°C")
print(f"Temp. Máx (L-V): {max_temp_laborables:.2f}°C")
print(f"Temp. Máx (S-D): {max_temp_findesemana:.2f}°C")
```

3. Desafío: np.vectorize para aplicar lógica Python (lambda)

Escenario: Tienes un "batch" de 100 imágenes en escala de grises (matrices 2D) y quieres aplicarles una corrección de contraste "sigmoide" (no lineal) definida por una lambda. Esta operación no está vectorizada por defecto. Usaremos np.vectorize para "puentear" Python y NumPy.

```
import numpy as np

print("\n--- 3. Desafío: np.vectorize y lambda ---")
np.random.seed(42)

# 1. Generar un batch de 100 "imágenes" de 32x32
#    Valores de píxeles (enteros de 0 a 255)
#    Shape = (100, 32, 32)
imagenes_batch = np.random.randint(0, 256, size=(100, 32, 32))
print(f"Shape del batch: {imagenes_batch.shape}")

# 2. Definir una función 'lambda' de Python para la corrección
#    Es una curva sigmoide que aumenta el contraste
#    k = controla la pendiente (más contraste)
k = 10.0
sigmoide_lambda = lambda x: 255 / (1 + np.exp(k * (0.5 - x / 255)))
```

```
# 3. Crear una versión "vectorizada" de la lambda
#    Esto crea una 'ufunc' de NumPy a partir de la función de Python
#    Sigue usando un bucle por debajo, pero la sintaxis es limpia
vectorized_sigmoide = np.vectorize(sigmoide_lambda)

# 4. Aplicar la función a *todo* el batch de una sola vez
imagenes_procesadas = vectorized_sigmoide(imagenes_batch)

# 5. Convertir de nuevo a enteros (la función devuelve floats)
imagenes_procesadas = imagenes_procesadas.astype(np.uint8)

# 6. Verificación (ejemplo con la primera imagen)
print("\nImagen 0 (Original) - Primeros 5x5 píxeles:\n", imagenes_batch[0, :5, :5])
print("\nImagen 0 (Procesada) - Primeros 5x5 píxeles:\n", imagenes_procesadas[0, :5, :5])
```

4. Desafío: Limpieza de Datos 3D (Sensores + Outliers)

Escenario: Tienes datos de 10 estaciones meteorológicas (E). Cada una reporta 24 horas (H) de 3 sensores (S): Temp, Humedad, Viento.
La shape es (E, H, S) -> (10, 24, 3).
El problema: Los sensores de viento a veces fallan y reportan -999.
Queremos calcular la media por hora (para todas las estaciones), pero ignorando los fallos.

```
import numpy as np

print("\n--- 4. Desafío: Limpieza de Datos 3D y Ejes (Axis) ---")
np.random.seed(42)
n_estaciones = 10
n_horas = 24
n_sensores = 3 # (Temp, Hum, Viento)

# 1. Generar datos (shape 10, 24, 3)
datos = np.random.rand(n_estaciones, n_horas, n_sensores) * 100
# Escalamos los datos para que sean realistas
datos[:, :, 0] = np.random.normal(15, 5, (n_estaciones, n_horas)) # Temp
datos[:, :, 1] = np.random.uniform(40, 80, (n_estaciones, n_horas)) # Humedad
datos[:, :, 2] = np.random.uniform(0, 50, (n_estaciones, n_horas)) # Viento

# 2. Introducir outliers (-999) solo en el sensor de Viento (col 2)
n_outliers = 30
idx_estaciones = np.random.randint(0, n_estaciones, n_outliers)
```

```
idx_horas = np.random.randint(0, n_horas, n_outliers)
datos[idx_estaciones, idx_horas, 2] = -999 # Acceso 3D

print(f"Shape de los datos: {datos.shape}")

# 3. Reemplazar outliers por `np.nan`
#    (Las funciones "nan-safe" son mejores que el masking)
datos[datos == -999] = np.nan
print(f"NaNs introducidos: {np.isnan(datos).sum()}")

# 4. Calcular la media "por hora"
#    Queremos colapsar las 10 estaciones (axis=0)
#    El resultado debe ser (24, 3) -> (media por hora, para cada sensor)
#    Usamos np.nanmean para ignorar los NaNs
media_horaria = np.nanmean(datos, axis=0)

print(f"\nShape de la media horaria: {media_horaria.shape}")
print("\nMedia por hora (Temp, Hum, Viento):")
# (mostramos las primeras 5 horas)
print(media_horaria[:5, :])
print("\nVerificación de media de Viento (debe ser > 0):")
print(media_horaria[:, 2].mean())
# Si no hubiéramos usado nanmean, los -999 habrían destrozado la media
```

5. Desafío: Vectorización de Texto (Bag-of-Words) y Similaridad de Coseno

Escenario: Tienes un corpus de documentos (reviews). Quieres convertirlos en una matriz numérica (Bag-of-Words) y usarla para encontrar el review más similar a una nueva consulta (query).
Esta es la base de los motores de búsqueda y los sistemas de recomendación.

```
import numpy as np
import string # Para la limpieza de puntuación

print("\n--- 5. Desafío: Vectorización de Texto (NLP con NumPy) ---")

# --- Corpus de entrada (Datos "sucios") ---
corpus = [
    "El rápido zorro marrón salta sobre el perro perezoso.",
    "Nunca salta un perro perezoso.",
    "El zorro rápido y marrón es ágil.",
```

```
    "Un perro ágil no es un perro perezoso."
]

# --- 1. Tokenización y Vocabulario (Lógica de Python - Módulo 0) ---
def tokenize(text):
    """Limpia, pasa a minúsculas y divide el texto en palabras."""
    # Quitar puntuación
    translator = str.maketrans('', '', string.punctuation)
    text_clean = text.translate(translator)
    # Minúsculas y dividir
    return text_clean.lower().split()

tokenized_corpus = [tokenize(doc) for doc in corpus]
print("Corpus tokenizado:\n", tokenized_corpus)

# Crear el vocabulario (set comprehension anidado)
vocab_set = {word for doc in tokenized_corpus for word in doc}
vocab = sorted(list(vocab_set)) # Convertir a lista ordenada
n_vocab = len(vocab)
print(f"\nTamaño del vocabulario: {n_vocab}")

# Crear el mapeo de palabra -> índice
word_to_index = {word: i for i, word in enumerate(vocab)}
print("Mapeo 'zorro' ->", word_to_index.get('zorro'))


# --- 2. Creación de la Matriz BoW (Lógica de NumPy) ---
n_docs = len(corpus)
bow_matrix = np.zeros((n_docs, n_vocab), dtype=np.int32)

for i, doc_tokens in enumerate(tokenized_corpus):
    # Obtener los índices de las palabras en este documento
    indices = [word_to_index[token] for token in doc_tokens if token in word_to_index]

    # (El truco de NumPy) Contar apariciones de cada índice
    # np.unique es perfecto para esto:
    doc_indices, counts = np.unique(indices, return_counts=True)

    # Rellenar la fila de la matriz BoW
    bow_matrix[i, doc_indices] = counts

print("\nMatriz Bag-of-Words (Shape: n_docs, n_vocab):\n", bow_matrix)
print(f"Shape: {bow_matrix.shape}")
```

```
# --- 3. Operación NLP: Similaridad de Coseno ---
# ¿Qué documento se parece más a "un rápido zorro marrón"?

def vectorize_query(query, word_to_index, n_vocab):
    """Convierte una query en un vector BoW usando el vocabulario existente."""
    query_tokens = tokenize(query)
    query_vec = np.zeros(n_vocab, dtype=np.int32)

    indices = [word_to_index[token] for token in query_tokens if token in word_to_index]
    doc_indices, counts = np.unique(indices, return_counts=True)

    query_vec[doc_indices] = counts
    return query_vec

query = "un rápido zorro marrón"
query_vec = vectorize_query(query, word_to_index, n_vocab)
print(f"\nVector de la Query '{query}':\n {query_vec}")

# --- 4. Cálculo Vectorizado de Similaridad ---
# Similaridad(A, B) = (A · B) / (||A|| * ||B||)

# (A · B) -> Producto punto de la matriz (4, 15) por el vector (15,)
# Resultado: (4,) -> un score por documento
dot_products = np.dot(bow_matrix, query_vec)

# ||A|| -> Norma L2 de cada fila de la matriz (axis=1)
norms_matrix = np.linalg.norm(bow_matrix, axis=1)

# ||B|| -> Norma L2 del vector query
norm_query = np.linalg.norm(query_vec)

print("\nProductos Punto:", dot_products)
print("Normas Matriz:", norms_matrix)
print("Norma Query:", norm_query)

# Evitar división por cero (si un doc o la query no tienen palabras del vocab)
denominator = norms_matrix * norm_query
similarities = np.divide(
    dot_products,
    denominator,
    out=np.zeros_like(dot_products, dtype=float), # Pone 0 si hay división por 0
```

```
    where=denominator!=0
)

# --- 5. Resultado ---
print(f"\nScores de Similaridad de Coseno: {similarities}")
best_doc_index = np.argmax(similarities)
print(f"El documento más similar a la query es el índice {best_doc_index}:")
print(f"-> '{corpus[best_doc_index]}'")
```