Coordinación de
Educación Abierta y a Distancia
VICERRECTORADO ACADÉMICO

Unach
UNIVERSIDAD NACIONAL DE CHIMBORAZO Online

PROGRAMACIÓN 2

Actividad Autónoma 2

Unidad 2: Manipulación y Análisis de Datos con NumPy

Tema 2: Indexación, Segmentación y Manipulación Avanzada

FACULTAD DE
Ingeniería

Nombres: Lyndon Andrés Castro Vaca

Fecha: 29-05-2025

Carrera: Ciencia de Datos e Inteligencia Artificial

Periodo académico:

Semestre:

Objetivo de la actividad: El objetivo de esta actividad es fortalecer las competencias en manipulación avanzada de arreglos utilizando la biblioteca NumPy, mediante la aplicación práctica de técnicas como la indexación avanzada, segmentación por slicing y condiciones booleanas, así como operaciones de broadcasting, transformación y combinación de arreglos. A través de la resolución guiada de ejercicios, se busca que el estudiante desarrolle un pensamiento lógico y analítico, mejore su comprensión del procesamiento eficiente de datos y consolide sus habilidades en programación científica con Python.

Recursos o temas que debe haber estudiado antes de hacer la actividad:

- Fundamentos de NumPy: creación de arreglos, tipos de datos y dimensiones.
- Indexación básica y avanzada: acceso a elementos utilizando índices enteros, listas de índices y máscaras booleanas.
- Segmentación (slicing) en arreglos unidimensionales y multidimensionales.
- Uso de funciones de manipulación de arreglos como reshape(), transpose(), concatenate(), split() y stack().
- Principios de broadcasting y cómo se aplican en operaciones entre arreglos de distintas formas.
- Lectura y análisis del documento base U2_S_.pdf con ejemplos y ejercicios de manipulación con NumPy.
- Manejo básico de un entorno de desarrollo como Jupyter Notebook o VS Code para probar y documentar el código.

Formato de entrega: Jupyter Notebook, a la nube de GitHub Classroom.

Instrucciones:

1. Selecciona tres ejercicios del PDF U2_S_.pdf.
2. Resuelve cada uno con:

- Comentarios detallados del proceso.
- Explicación del resultado.
- Prueba con distintos datos de entrada.

3. Incluye tus respuestas en un archivo .ipynb o .py

Ejercicio 1 – Producto punto vs. producto Hadamard Genere dos matrices aleatorias A y B de tamaño 200 × 200:

a. Calcule el producto Hadamard $A * B$.
b. Calcule el producto punto AB.
c. Con %timeit compare el tiempo medio de ambas operaciones y discuta la diferencia teórica

```
1.  import numpy as np
2.  import timeit
3.
4.  # Genero dos matrices aleatorias de tamaño 200x200 para trabajar con ellas
5.  A = np.random.rand(200, 200)
6.  B = np.random.rand(200, 200)
7.
8.  # a) Calculo el producto Hadamard, que es la multiplicación elemento a elemento
9.  hadamard_product = A * B
```

```
10.
11. # b) Calculo el producto punto (multiplicación matricial tradicional)
12. dot_product = np.dot(A, B)
13.
14. # c) Mido el tiempo que tarda cada operación para comparar su eficiencia
15. hadamard_time = timeit.timeit('A * B', globals=globals(), number=10)
16. dot_time = timeit.timeit('np.dot(A, B)', globals=globals(), number=10)
17.
18. print("Tiempo promedio producto Hadamard:", hadamard_time/10, "segundos")
19. print("Tiempo promedio producto punto:", dot_time/10, "segundos")
20.
21. # Explico que el producto Hadamard es más rápido porque solo multiplica elemento a
    elemento,
22. # mientras que el producto punto requiere operaciones más complejas como sumas y
    multiplicaciones de filas por columnas,
23. # lo que implica mayor carga computacional, especialmente en matrices grandes.
24.
25. # Repito la comparación con matrices más pequeñas (10x10) para ver cómo cambia la
    diferencia de tiempos
26. A_small = np.random.rand(10, 10)
27. B_small = np.random.rand(10, 10)
28.
29. hadamard_time_small = timeit.timeit('A_small * B_small', globals=globals(),
    number=100)
30. dot_time_small = timeit.timeit('np.dot(A_small, B_small)', globals=globals(),
    number=100)
31.
32. print("\nPara matrices 10x10:")
33. print("Tiempo Hadamard:", hadamard_time_small/100, "segundos")
34. print("Tiempo producto punto:", dot_time_small/100, "segundos")
35.
```

Prueba de escritorio

```
Tiempo promedio producto Hadamard: 1.5490000077988952e-05 segundos
Tiempo promedio producto punto: 0.00037678000080632046 segundos

Para matrices 10x10:
Tiempo Hadamard: 6.78999931551516e-07 segundos
Tiempo producto punto: 1.2369999603834004e-06 segundos
PS C:\Users\andre\OneDrive\Documentos\carteta Py>
```

Ejercicio 7 – Estimación de parámetros de una normal Simule 10 000 muestras de N(μ = 2,5, σ = 1,3). Estime μ y σ 2 con fórmulas vectorizadas y compare con np.mean y np.var.

```
1. import numpy as np
2.
3. # Defino los parámetros reales de la distribución normal que quiero
   simular
4. mu_real = 2.5
5. sigma_real = 1.3
6.
7. # Genero 10,000 muestras aleatorias usando estos parámetros
8. muestras = np.random.normal(loc=mu_real, scale=sigma_real,
   size=10000)
9.
10.# Estimo la media (mu) de forma manual sumando y dividiendo por la
   cantidad de muestras
11.mu_estimado = np.sum(muestras) / len(muestras)
12.# También uso la función de numpy para comparar
13.mu_np = np.mean(muestras)
14.
15.# Estimo la varianza (sigma^2) de forma manual
16.sigma2_estimado = np.sum((muestras - mu_estimado)**2) /
   len(muestras)
17.# Comparo con la función de numpy
18.sigma2_np = np.var(muestras)
19.
20.# Muestro los resultados para ver cómo se comparan mis estimaciones
   manuales y las de numpy con los valores reales
21.print("Estimación manual de mu:", mu_estimado)
22.print("Estimación con np.mean:", mu_np)
23.print("\nEstimación manual de sigma^2:", sigma2_estimado)
24.print("Estimación con np.var:", sigma2_np)
25.print("\nValores reales: mu =", mu_real, ", sigma^2 =",
   sigma_real**2)
26.
27.# Ahora quiero ver qué pasa si uso menos muestras (solo 100)
28.muestras_peq = np.random.normal(loc=mu_real, scale=sigma_real,
   size=100)
29.mu_peq = np.mean(muestras_peq)
30.sigma2_peq = np.var(muestras_peq)
31.
32.# Imprimo las estimaciones usando pocas muestras para comparar la
   precisión
33.print("\nCon 100 muestras:")
34.print("mu estimado:", mu_peq)
35.print("sigma^2 estimado:", sigma2_peq)
36.
```

Prueba de escritorio

```
Estimación manual de mu: 2.5024598449842133
Estimación con np.mean: 2.5024598449842133

Estimación manual de sigma^2: 1.6999469659076487
Estimación con np.var: 1.6999469659076487

Valores reales: mu = 2.5 , sigma^2 = 1.6900000000000002

Con 100 muestras:
mu estimado: 2.3768424458267585
sigma^2 estimado: 1.458881502230088
```

Ejercicio 9 – Correlación de Pearson y covarianza Simule X ∼ N(0, 1) y Y = 3X + ε con ε ∼ N(0, 0,5 2). Calcule la covarianza y el coeficiente de Pearson sin usar np.corrcoef.

```python
import numpy as np

# Fijo una semilla para que los resultados sean reproducibles
np.random.seed(42)

# Genero 1000 datos de X con distribución normal estándar
X = np.random.normal(0, 1, 1000)
# Genero ruido epsilon con menor desviación estándar para añadir variabilidad a Y
epsilon = np.random.normal(0, 0.5, 1000)
# Defino Y como una relación lineal con X más ruido
Y = 3 * X + epsilon

# Calculo manualmente la media de X y Y para centrar los datos
mean_X = np.mean(X)
mean_Y = np.mean(Y)
# Calculo la covarianza manualmente como promedio del producto de las diferencias respecto a las medias
covarianza = np.sum((X - mean_X) * (Y - mean_Y)) / len(X)

# Calculo manualmente las desviaciones estándar para normalizar la covarianza
std_X = np.std(X)
std_Y = np.std(Y)
# Calculo el coeficiente de correlación de Pearson dividiendo la covarianza por el producto de desviaciones
pearson = covarianza / (std_X * std_Y)

# Verifico mis cálculos manuales con las funciones de numpy para covarianza y correlación
cov_np = np.cov(X, Y, ddof=0)[0, 1]  # ddof=0 para usar N en el denominador, igual que el cálculo manual
pearson_np = np.corrcoef(X, Y)[0, 1]
```

```
28.
29. print("Covarianza manual:", covarianza)
30. print("Covarianza con np.cov:", cov_np)
31. print("\nCoeficiente de Pearson manual:", pearson)
32. print("Coeficiente de Pearson con np.corrcoef:", pearson_np)
33.
34. # Ahora pruebo con otra relación lineal negativa para ver el cambio en la covarianza
    y correlación
35. Y2 = -2 * X + epsilon
36. cov2 = np.cov(X, Y2, ddof=0)[0, 1]
37. pearson2 = np.corrcoef(X, Y2)[0, 1]
38.
39. print("\nCon Y = -2X + ε:")
40. print("Covarianza:", cov2)
41. print("Pearson:", pearson2)
42.
```

Prueba de escritorio

```
Covarianza manual: 2.854004952675637
Covarianza con np.cov: 2.854004952675637

Coeficiente de Pearson manual: 0.9857246307584189
Coeficiente de Pearson con np.corrcoef: 0.9857246307584191

Con Y = -2X + ε:
Covarianza: -1.9355199959819502
Pearson: -0.9697172287650991
PS C:\Users\andre\OneDrive\Documentos\carteta Py>
```