Class 2: Advanced NumPy and Data Manipulation

Objectives

- Understand multidimensional NumPy arrays.
- Learn NumPy operations: indexing, slicing, reshaping, filtering, vectorization, broadcasting.
- Explore NumPy axis for aggregation.
- Use NumPy random functions and other useful methods.

1 Multidimensional NumPy Arrays

NumPy arrays can represent mathematical objects:

- **Scalar (0D)**: Single value.
- **Vector (1D)**: List of values.
- **Matrix (2D)**: Table of values.
- **Tensor (xD, x > 2)**: Higher-dimensional arrays.

Code Example

```
import numpy as np
# Scalar: single value, 0 dimensions, used for constants
scalar = np.array(47)  # Output: 47, ndim: 0

# Vector: 1D array, used for sequences or features
vector = np.array([1, 2, 3, 4, 5])  # Output: [1 2 3 4 5], ndim: 1

# Matrix: 2D array, used for tables or images
matrix = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
# Output: [[1 2 3], [4 5 6], [7 8 9]], ndim: 2
```

2 NumPy Operations

NumPy supports efficient operations for data manipulation.

Indexing

Access elements using non-negative (0, 1, ...) or negative (-1, -2, ...) indices.

```
# 1D array indexing: access single elements
my_1d_array = np.array([1, 2, 3, 4, 5])
print(my_1d_array[2])   # Output: 3 (third element)
print(my_1d_array[-3])  # Output: 3 (third from end)

# 2D array indexing: access elements with [row, col]
my_2d_array = np.array([[1, 2, 3, 4], [5, 8, 9, 5], [7, 1, 2, 7]])
print(my_2d_array[1, 1])  # Output: 8 (row 2, column 2)
print(my_2d_array[1, -3]) # Output: 8 (row 2, third from end)
```

Slicing

Extract subarrays using `[start:stop:step]` (end index excluded).

```
# 1D slicing: extract parts of an array
print(my_1d_array[::])      # Output: [1 2 3 4 5] (full array)
print(my_1d_array[2:5])     # Output: [3 4 5] (elements 3 to 5)
print(my_1d_array[1:5:2])   # Output: [2 4] (every second element)

# 2D slicing: extract submatrices
print(my_2d_array[0:2, 1:3]) # Output: [[2 3], [8 9]] (rows 1-2, cols 2-3)
print(my_2d_array[1:3, :])   # Output: [[5 8 9 5], [7 1 2 7]] (rows 2-3)
```

Reshaping

Change array shape while preserving elements.

```
# Reshape 2D to 3D: number of elements must match
my_2d_array = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9], [10, 11, 12], [13,
    14, 15]])
my_2d_array_reshaped = my_2d_array.reshape(1, 3, 5)
# Output: [[[1 2 3 4 5], [6 7 8 9 10], [11 12 13 14 15]]], shape: (1, 3, 5)

# Reshape with -1: automatically calculate one dimension
my_np_array = np.zeros(shape=(3, 4, 2, 2))
my_np_array_reshaped = my_np_array.reshape(2, -1, 12)  # -1 infers size
# Output: shape: (2, 2, 12)
```

Filtering

Select elements using boolean masks or indices.

```
# Boolean filtering: select elements based on conditions
my_1d_array = np.array([1, 2, 3, 4, 5, 6])
filter_mask = [True, False, True, False, True, False]
print(my_1d_array[filter_mask])  # Output: [1 3 5]

# Index-based filtering: select specific indices
indices = [0, 2, 5]
print(my_1d_array[indices])  # Output: [1 3 6]
```

Vectorization

Perform operations on entire arrays without loops.

```
# Vectorized operation: apply operations to all elements
my_1d_array = np.array([1, 2, 3, 4, 5])
print(my_1d_array % 2 == 1)  # Output: [ True False  True False  True]
print(my_1d_array + 5)       # Output: [6 7 8 9 10]
```

Broadcasting

Align arrays of different shapes for operations.

```
# Broadcasting: multiply 1D array with 2D array
w = np.array([1, 2])
x = np.array([[3, 4], [5, 6], [7, -8], [-5, -3]])
print(w * x)  # Output: [[ 3  8], [ 5 12], [ 7 -16], [-5 -6]]
```

3 NumPy Axis

Axis specifies the dimension for operations:

- `axis=0`: Operate along rows (column-wise).
- `axis=1`: Operate along columns (row-wise).

```
# Axis-based aggregation
my_np_array = np.array([[1, 5, -2], [0, 1, 9]])
print(np.sum(my_np_array))        # Output: 14 (sum all elements)
print(np.sum(my_np_array, axis=0)) # Output: [1 6 7] (column sums)
print(np.sum(my_np_array, axis=1)) # Output: [4 10] (row sums)

# 3D array axis
my_np_3d = np.zeros(shape=(3, 2, 4))
print(np.sum(my_np_3d, axis=0).shape) # Output: (2, 4)
print(np.sum(my_np_3d, axis=1).shape) # Output: (3, 4)
print(np.sum(my_np_3d, axis=2).shape) # Output: (3, 2)
```

4 NumPy Random

Generate random numbers for simulations or testing.

```
# Random numbers
print(np.random.rand())          # Output: random float [0, 1)
print(np.random.rand(2, 5))      # Output: 2x5 array of random floats
print(np.random.randint(100))     # Output: random int [0, 100)
print(np.random.randint(100, size=(5,))) # Output: 5 random ints

# Shuffle array: randomize order in-place
my_array = np.random.randint(100, size=(5,))
np.random.shuffle(my_array)
print(my_array)  # Output: shuffled array
```

5 Useful Methods

Additional NumPy functions for data manipulation.

```
# Sorting
probabilities = np.array([0.3, 0.1, 0.4, 0.2])
print(np.sort(probabilities))          # Output: [0.1 0.2 0.3 0.4]
print(np.sort(probabilities)[::-1])    # Output: [0.4 0.3 0.2 0.1]

# Argmax/Argmin: find index of max/min value
print(np.argmax(probabilities))        # Output: 2 (index of 0.4)
print(np.argmin(probabilities))        # Output: 1 (index of 0.1)

# Split array into subarrays
my_np_array = np.array([1, 5, 1, 1, 5, 7])
new_np_array = np.split(my_np_array, 3) # Split into 3 equal parts
print(new_np_array)                    # Output: [array([1, 5]), array([1,
    1]), array([5, 7])]

# Dot product: for vector operations
my_np_array_1 = np.array([1, 2, 3])
my_np_array_2 = np.array([3, 4, 3])
print(np.dot(my_np_array_1, my_np_array_2)) # Output: 14 (1*3 + 2*4 + 3*3)

# Matrix transpose: swap rows and columns
my_2d_array_1 = np.array([[2, 1, 3], [7, 5, 6]])
print(my_2d_array_1.T)                 # Output: [[2 7], [1 5], [3 6]]

# Statistical measures
print(np.mean(my_np_array_1))          # Output: 2.0
print(np.median(my_np_array_1))        # Output: 2.0
print(np.std(my_np_array_1))           # Output: 0.816496580927726
```

Summary

Advanced NumPy techniques enable efficient data manipulation. Practice multidimensional arrays, operations, axis-based aggregation, and random number generation to prepare for data science tasks.