NumPy for Data Science - Part 2

Type	Data science masterclass

I. Linear Algebra with NumPy

1. Vectors and Matrix Multiplication

Vectors

- **Definition**: 1D arrays representing magnitude and direction.
- **Operations**:

```
import numpy as np
v = np.array([2, 5, -1])      # Vector of shape (3,)
w = np.array([3, 0, 4])
dot_product = np.dot(v, w)    # 2*3 + 5*0 + (-1)*4 = 2
cross_product = np.cross(v, w) # Cross product (3D only)
```

Matrices

- **Definition**: 2D arrays representing linear transformations.
- **Types**:

```
A = np.array([[1, 2], [3, 4]])  # 2×2 matrix
B = np.array([[5, 6], [7, 8]])
```

Matrix Multiplication

- **Rules**: Columns of first matrix = Rows of second matrix.
- **Methods**:

```
# Method 1: np.dot()
C = np.dot(A, B)
```

```
# Method 2: @ operator
C = A @ B

# Element-wise multiplication (not matrix multiplication!)
C = A * B
```

- **Shapes**:
  - `A.shape = (m, n)`, `B.shape = (n, p)` → Result shape = `(m, p)`.

Broadcasting

- Vectors automatically cast into row/column vectors:

  ```
  v = np.array([1, 2])
  A = np.array([[3, 4], [5, 6]])
  result = A @ v  # Treated as column vector (2×1)
  ```

---

2. Matrix Transposition and Inversion

Transposition

- Swap rows and columns:

  ```
  A = np.array([[1, 2], [3, 4], [5, 6]])  # 3×2
  A_T = A.T                               # 2×3
  ```

- **Properties**:
  - `(A.T).T = A`
  - `(AB)^T = B^T A^T`.

Inversion

- **Definition**: Matrix
- A^{-1} such that $A * A^{-1} = I$
- **Requirements**:

    - Square matrix.
    - Non-zero determinant (`np.linalg.det(A) ≠ 0`).

    ```
    A = np.array([[3, 1], [1, 2]])
    A_inv = np.linalg.inv(A)  # Inverse matrix
    identity = A @ A_inv      # Approximates [[1, 0], [0, 1]]
    ```

- **Pseudo-Inverse**: For non-square matrices (uses SVD):

    ```
    A_pinv = np.linalg.pinv(A)  # Moore-Penrose pseudo-inverse
    ```

---

3. Solving Linear Equations (`np.linalg.solve()`)

- **Problem**: Find x in Ax=b
- **Example**:

    ```
    A = np.array([[2, 1], [1, 3]])  # Coefficient matrix
    b = np.array([8, 11])          # Constants
    x = np.linalg.solve(A, b)      # Solution: x = [3., 2.]
    ```

- **Conditions**:
    - A must be square and invertible.
    - If A is singular (non-invertible), use least squares:

        ```
        x, residuals, rank, _ = np.linalg.lstsq(A, b, rcond=None)
        ```

- **Numerical Stability**:
    - Avoid inverting matrices directly. Use `solve()` for better accuracy.

---

4. Eigenvalues and Eigenvectors

Definitions

- **Eigenvalue** λ: Scalar such that $Av = \lambda v$.

- **Eigenvector (v)**: Non-zero vector transformed by A only by scaling.

Computation

```
A = np.array([[4, 2], [1, 3]])
eigenvalues, eigenvectors = np.linalg.eig(A)
```

- **Output**:
  - `eigenvalues`: 1D array of eigenvalues (e.g., `[5., 2.]`).
  - `eigenvectors`: 2D array where **columns** are eigenvectors.

Symmetric Matrices

- Use `np.linalg.eigh()` for faster computation:

  ```
  eigenvalues, eigenvectors = np.linalg.eigh(A)  # A must be symmetric
  ```

Applications

- Principal Component Analysis (PCA).
- Stability analysis in differential equations.

---

5. Additional Tools

Matrix Rank

- Number of linearly independent rows/columns:

  ```
  rank = np.linalg.matrix_rank(A)
  ```

Determinant

- Scalar value encoding matrix properties:

  ```
  det = np.linalg.det(A)  # Determinant of A
  ```

Norms

- Vector/matrix "size":

```
v_norm = np.linalg.norm(v, ord=2)  # L2 norm (default)
matrix_norm = np.linalg.norm(A, ord='fro')  # Frobenius norm
```

Common Errors & Tips

1. **Shape Mismatch**:
   - Ensure matrices align for multiplication (e.g., `(3,2) @ (2,4)` works).
2. **Singular Matrix**:
   - Check determinant or condition number:

   ```
   cond_number = np.linalg.cond(A)  # Large value ≈ singular
   ```
3. **Efficiency**:
   - Prefer `solve()` over `inv()` for \(Ax = b \).

II. Statistical Operations in NumPy

NumPy offers a powerful suite of statistical functions that are essential for data analysis and scientific computing.

Descriptive Statistics

Descriptive statistics summarize and describe the main features of a dataset.

Mean (Average)

- **Function**: `np.mean()`
- **Description**: Calculates the arithmetic mean along the specified axis.
- **Usage**:

```
import numpy as np

data = np.array([1, 2, 3, 4, 5])
mean_value = np.mean(data)
# Output: 3.0
```

- **Parameters**:
  - `a`: Input array.
  - `axis`: Axis along which the means are computed. Default is to compute the mean of the flattened array.

Median

- **Function**: `np.median()`
- **Description**: Computes the median of the data along the specified axis.
- **Usage**:

```
median_value = np.median(data)
# Output: 3.0
```

- **Note**: The median is less affected by outliers compared to the mean.

Mode

- **Note**: NumPy does not have a built-in `mode` function.
- **Alternative**: Use SciPy's `stats.mode()` function.

```
from scipy import stats

mode_value, count = stats.mode(data)
# Output: mode_value = array([1]), count = array([1])
```

- **Usage**: Useful for finding the most frequent value in a dataset.

Standard Deviation

- **Function**: `np.std()`
- **Description**: Computes the standard deviation, a measure of the spread of data.
- **Usage**:

```
std_dev = np.std(data)
# Output: 1.41421356
```

- **Formula:**

  $\sigma = \mathrm{sqrt}((1/N) * \Sigma (x_i - \mu)^2)$

- Where μ is the mean, x_i are the data points, and N is the number of data points

Variance

- **Function**: `np.var()`
- **Description**: Computes the variance, representing the average squared deviation from the mean.
- **Usage**:

```
variance = np.var(data)
# Output: 2.0
```

Aggregation Functions

Aggregation functions compute a single value from an array.

Minimum and Maximum

- **`np.min()`**: Returns the smallest value.

```
min_value = np.min(data)
```

```
# Output: 1
```

- `np.max()`: Returns the largest value.

```
max_value = np.max(data)
# Output: 5
```

Sum

- **Function**: `np.sum()`
- **Description**: Adds up all elements in the array.
- **Usage**:

```
total = np.sum(data)
# Output: 15
```

Product

- **Function**: `np.prod()`
- **Description**: Calculates the product of all elements.
- **Usage**:

```
product = np.prod(data)
# Output: 120
```

Cumulative Sum and Product

- `np.cumsum()`: Cumulative sum of elements.

```
cumsum = np.cumsum(data)
# Output: array([ 1,  3,  6, 10, 15])
```

- `np.cumprod()`: Cumulative product of elements.

```
cumprod = np.cumprod(data)
# Output: array([  1,   2,   6,  24, 120])
```

Covariance and Correlation Matrices

Understanding relationships between variables is crucial in statistical analysis.

Covariance Matrix

- **Function**: `np.cov()`
- **Description**: Computes the covariance matrix to assess how two variables vary together.
- **Usage**:

```
x = np.array([1, 2, 3])
y = np.array([4, 5, 6])

covariance_matrix = np.cov(x, y)
# Output:
# array([[1., 1.],
#        [1., 1.]])
```

- **Interpretation**:
  - Positive values indicate a positive relationship.
  - Negative values indicate an inverse relationship.

Correlation Matrix

- **Function**: `np.corrcoef()`
- **Description**: Computes the Pearson correlation coefficient matrix.
- **Usage**:

```
correlation_matrix = np.corrcoef(x, y)
# Output:
```

```
# array([[1., 1.],
#        [1., 1.]])
```

- **Interpretation**:
  - Values range from -1 to 1.
  - **1**: Perfect positive correlation.
  - **0**: No correlation.
  - **1**: Perfect negative correlation.

Example

```
import matplotlib.pyplot as plt

# Sample data
x = np.random.rand(100)
y = 2 * x + np.random.normal(0, 0.1, 100)

# Calculate correlation coefficient
corr_coeff = np.corrcoef(x, y)[0, 1]
print(f"Correlation Coefficient: {corr_coeff}")

# Scatter plot
plt.scatter(x, y)
plt.title('Scatter Plot of x and y')
plt.xlabel('x')
plt.ylabel('y')
plt.show()
```

Random Number Generation for Statistical Modeling

NumPy's random module provides functions to generate random numbers for simulations and modeling.

Basic Random Numbers

- **Uniform Distribution**
  - **Function:** `np.random.rand()`
  - **Description**: Generates random numbers between 0 and 1.
  - **Usage**:

    ```
    random_numbers = np.random.rand(5)
    # Output: array([0.37454012, 0.95071431, 0.73199394, 0.59865848, 0.15601864])
    ```

  - **Shape**: Specify dimensions as arguments, e.g., `np.random.rand(3, 2)`.
- **Normal Distribution**
  - **Function:** `np.random.randn()`
  - **Description**: Generates samples from a standard normal distribution (mean=0, std=1).
  - **Usage**:

    ```
    normal_samples = np.random.randn(5)
    # Output: array([-0.1382643 ,  0.64768854,  1.52302986, -0.23415337, -0.23413696])
    ```

Random Integers

- **Function:** `np.random.randint()`
- **Description**: Returns random integers from `low` (inclusive) to `high` (exclusive).
- **Usage**:

  ```
  random_ints = np.random.randint(1, 10, size=5)
  # Output: array([2, 9, 6, 3, 5])
  ```

Setting the Seed

- **Function:** `np.random.seed()`

- **Description**: Sets the seed for random number generation to ensure reproducibility.
- **Usage**:

```
np.random.seed(42)
```

Custom Distributions

- **Uniform Distribution**
    - **Function**: `np.random.uniform(low, high, size)`
    - **Usage**:

    ```
    uniform_samples = np.random.uniform(-1, 1, size=5)
    # Output: array([-0.25091976,  0.90142861,  0.46398788,  0.19731697,
    -0.68796272])
    ```

- **Normal Distribution**
    - **Function**: `np.random.normal(loc, scale, size)`
    - **Parameters**:
        - `loc`: Mean of the distribution.
        - `scale`: Standard deviation.
        - `size`: Output shape.
    - **Usage**:

    ```
    normal_samples = np.random.normal(loc=0, scale=1, size=5)
    # Output: array([ 0.64768854, 1.52302986, -0.23415337, -0.2341369
    6, 1.57921282])
    ```

Random Choice

- **Function**: `np.random.choice()`
- **Description**: Generates a random sample from a given 1-D array.

- **Usage**:

```
elements = np.array([10, 20, 30, 40, 50])
choices = np.random.choice(elements, size=3, replace=False)
# Output: array([30, 40, 20])
```

Additional Statistical Functions

Percentiles

- **Function**: `np.percentile()`
- **Description**: Computes the q-th percentile of the data.
- **Usage**:

```
percentile_25 = np.percentile(data, 25)
# Output: 2.0
percentile_75 = np.percentile(data, 75)
# Output: 4.0
```

Histogram Data

- **Function**: `np.histogram()`
- **Description**: Computes the histogram of a set of data.
- **Usage**:

```
counts, bin_edges = np.histogram(data, bins=5)
# counts: array of counts in each bin
# bin_edges: array of bin edges
```

Correlation Function

- **Function**: `np.correlate()`
- **Description**: Computes the cross-correlation of two 1-D sequences.

- **Usage**:

  ```
  signal1 = np.array([1, 2, 3])
  signal2 = np.array([0, 1, 0.5])
  correlation = np.correlate(signal1, signal2, mode='full')
  # Output: array([0.5, 2.5, 4.0, 3.5, 0.0])
  ```

Working with Multidimensional Arrays

NumPy functions can operate along specified axes in multi-dimensional arrays.

Axis Parameters

- **Axis 0**: Performs operation down each column.
- **Axis 1**: Performs operation across each row.

Examples

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

# Sum across columns
sum_columns = np.sum(matrix, axis=0)
# Output: array([4, 6])

# Sum across rows
sum_rows = np.sum(matrix, axis=1)
# Output: array([3, 7])
```

Practical Applications

Z-Score Normalization

Standardizing data to have a mean of 0 and a standard deviation of 1.

- **Formula**:

$z = \frac{x-\mu}{\sigma}$

- **Implementation**:

  ```
  z_scores = (data - np.mean(data)) / np.std(data)
  ```

Data Simulation

Generate datasets for testing statistical models.

```
# Simulate 1000 data points from a normal distribution
simulated_data = np.random.normal(loc=50, scale=5, size=1000)

# Calculate descriptive statistics
mean_simulated = np.mean(simulated_data)
std_simulated = np.std(simulated_data)

print(f"Mean: {mean_simulated}, Standard Deviation: {std_simulated}")
```

Best Practices and Tips

- **Reproducibility**: Always set a random seed when generating random data for experiments.

  ```
  np.random.seed(0)
  ```

- **Vectorization**: Utilize NumPy's vectorized operations for efficiency instead of Python loops.
- **Data Types**: Ensure data types are consistent to avoid unexpected results, especially when performing operations that are sensitive to integer division.
- **Axis Awareness**: Be mindful of the `axis` parameter to control the direction of operations in multi-dimensional arrays.
- **Chain Functions**: Combine operations where possible for concise code.

```
# Calculate the mean of squared data
mean_squared = np.mean(np.square(data))
```

III. Broadcasting and Vectorization

1. Understanding Broadcasting Rules

Broadcasting allows NumPy to perform operations on arrays of different shapes without creating unnecessary copies of data. This leads to more efficient computations.

Rules of Broadcasting:

1. **Match dimensions from the right**: NumPy compares shapes element-wise from the rightmost dimension.
2. **Size 1 can be expanded**: If a dimension is `1`, it is stretched to match the larger dimension.
3. **Missing dimensions are treated as `1`**: If one array has fewer dimensions, it is prepended with `1`s to match the shape of the larger array.

Example:

```
import numpy as np

A = np.array([[1, 2, 3], [4, 5, 6]])  # Shape: (2,3)
B = np.array([[1], [2]])             # Shape: (2,1)

result = A + B  # B is broadcasted to (2,3)

print(result)
```

Output:

```
[[2 3 4]
 [6 7 8]]
```

Here, `B` with shape `(2,1)` is broadcasted to `(2,3)` before addition.

---

2. Using Vectorized Operations for Efficiency

Vectorization refers to applying operations to entire arrays without explicit loops. It leverages **NumPy's optimized C and Fortran backend**, making computations significantly faster.

Why Use Vectorized Operations?

- Eliminates explicit `for` loops
- Reduces execution time
- Makes code more readable and concise

Example: Multiplying two arrays without loops

```
import numpy as np

A = np.array([1, 2, 3, 4])
B = np.array([10, 20, 30, 40])

# Vectorized multiplication
result = A * B

print(result)
```

Output:

```
[ 10  40  90 160]
```

This avoids a `for` loop and is much faster.

---

3. Example: Optimizing Loops with NumPy

Suppose we want to compute the square of an array's elements.

Using Loops (Slow approach)

```
arr = [1, 2, 3, 4, 5]
squared = []

for x in arr:
    squared.append(x ** 2)

print(squared)
```

Time Complexity: `O(n)` (Python loops are slow due to interpreted execution)

Using NumPy Vectorization (Fast approach)

```
import numpy as np

arr = np.array([1, 2, 3, 4, 5])

squared = arr ** 2  # Element-wise squaring

print(squared)
```

Optimized execution runs in the **C backend**, making it significantly faster.

Key Takeaways

- **Broadcasting** allows operations on different shaped arrays without copying data
- **Vectorization** eliminates loops, making operations faster and memory-efficient
- **NumPy functions** leverage optimized backend implementations for performance

---

Bonus: Check Broadcasting Compatibility

You can use `np.broadcast_shapes()` in NumPy 1.20+ to check if two shapes can be broadcasted.

```
import numpy as np

np.broadcast_shapes((2, 1), (3,))  # Returns (2,3)
```

IV. Advanced NumPy Features

1. Structured Arrays and Record Arrays

Structured arrays allow storing heterogeneous data types (e.g., integers, floats, strings) in a single NumPy array. They work like a table where each element has named fields, similar to a database record or a Pandas DataFrame.

Creating a Structured Array

```
import numpy as np

# Define a structured data type
dtype = [('name', 'U10'), ('age', 'i4'), ('weight', 'f4')]

# Create an array with structured data
data = np.array([('Alice', 25, 55.5), ('Bob', 30, 72.3)], dtype=dtype)

print(data)
print(data['name'])  # Access a specific field
```

Output:

```
[('Alice', 25, 55.5) ('Bob', 30, 72.3)]
```

```
['Alice' 'Bob']
```

Each record consists of a name (string), age (integer), and weight (float).

Modifying Fields in a Structured Array

```
data['age'] += 1  # Increment all ages by 1
print(data)
```

Record Arrays

A `record array` is similar to a structured array but allows access to fields using attribute notation (`data.name` instead of `data['name']`).

```
record_data = np.rec.array(data)
print(record_data.name)  # Access like an attribute
```

---

2. Memory Layout and Performance Optimization

Understanding NumPy's memory layout is essential for optimizing performance, especially for large datasets.

C vs. Fortran Order

- **C-order (Row-major):** Elements in the same row are stored contiguously in memory.
- **Fortran-order (Column-major):** Elements in the same column are stored contiguously in memory.

```
A = np.array([[1, 2, 3], [4, 5, 6]], order='C')  # Row-major
B = np.array([[1, 2, 3], [4, 5, 6]], order='F')  # Column-major

print(A.flags['C_CONTIGUOUS'])  # True
print(B.flags['F_CONTIGUOUS'])  # True
```

Checking and Changing Memory Layout

```
A = np.array([[1, 2], [3, 4]])
print(A.strides)  # Stride information (how many bytes to move to the next ele
ment)

# Convert to column-major (Fortran order)
A_f = np.asfortranarray(A)
print(A_f.flags['F_CONTIGUOUS'])  # True
```

Optimizing Memory Access with `np.copy()` and `np.ravel()`

```
B = A.ravel(order='F')  # Flatten using Fortran order
print(B)
```

Flattening in the right order improves performance in certain operations.

---

3. Using `np.frombuffer()` , `np.fromfile()` , and `np.tofile()`

These functions allow efficient reading and writing of binary data.

Using `np.frombuffer()`

Reads raw bytes and interprets them as an array without making a copy. Useful for working with shared memory.

```
import numpy as np

buffer = b'\x01\x02\x03\x04'
arr = np.frombuffer(buffer, dtype=np.uint8)
print(arr)  # [1 2 3 4]
```

Since no copy is made, modifying `arr` may change the original buffer.

Using `np.fromfile()`

Reads data directly from a binary or text file into a NumPy array.

```
arr = np.fromfile('data.bin', dtype=np.float32)
print(arr)
```

To read text files:

```
arr = np.fromfile('data.txt', dtype=np.float32, sep=' ')
```

Using `np.tofile()`

Writes an array to a file in binary format.

```
arr = np.array([1.5, 2.5, 3.5], dtype=np.float32)
arr.tofile('output.bin')
```

To save in text format:

```
arr.tofile('output.txt', sep=' ')
```

Key Takeaways

- **Structured arrays** store mixed data types efficiently.
- **Memory layout (C vs. Fortran order)** affects performance in computations.
- `np.frombuffer()` reads raw memory without copying.
- `np.fromfile()` **and** `np.tofile()` enable efficient binary and text file handling.

These features are essential for high-performance numerical computing in Python.

V. Working with Datasets

1. NumPy Arrays for Large Data Manipulation

NumPy arrays are ideal for working with large datasets due to their efficiency in terms of memory usage and speed. By storing data in contiguous memory blocks, NumPy performs operations faster than lists or other Python structures.

Advantages of NumPy for Large Data

- **Memory Efficiency:** NumPy uses a fixed-size type for each element, unlike Python lists, which store data as objects with extra overhead.
- **Vectorized Operations:** Operations are applied to entire arrays, avoiding slow loops and improving performance.
- **Multidimensional Arrays:** Efficient handling of multi-dimensional datasets (e.g., matrices or tensors).

Example: Large Dataset Manipulation

```
import numpy as np

# Create a large NumPy array
data = np.random.rand(1000000)

# Apply a vectorized operation (e.g., square all elements)
result = data ** 2

print(result[:10])  # Display the first 10 elements
```

This approach is much faster than manually iterating over each element in a large dataset.

---

2. Handling Missing Data (NaN, inf)

Missing or invalid data is common when working with real-world datasets. NumPy provides tools to handle `NaN` (Not a Number) and `inf` (infinity) values.

Detecting Missing Data

```
import numpy as np

# Create an array with NaN and inf
data = np.array([1.0, 2.0, np.nan, np.inf, 5.0])

# Detect NaN values
is_nan = np.isnan(data)
print("NaN values:", is_nan)

# Detect inf values
is_inf = np.isinf(data)
print("Inf values:", is_inf)
```

Handling NaN Values

- **Replacing NaN with a specific value**

```
data[np.isnan(data)] = 0  # Replace NaN with 0
```

- **Removing NaN values**

```
data_clean = data[~np.isnan(data)]  # Remove NaN values
print(data_clean)
```

Handling Inf Values

- **Replacing inf with a specific value**

```
data[np.isinf(data)] = 1000  # Replace inf with 1000
```

---

3. Working with Time Series Data (Using `np.datetime64`)

Time series data is widely used in fields like finance, economics, and sensor data. NumPy provides the `np.datetime64` object to handle dates and times efficiently.

Creating `np.datetime64` Objects

```
import numpy as np

# Create datetime objects
date1 = np.datetime64('2025-02-06')
date2 = np.datetime64('2025-03-01')

print(date1, date2)
```

Arithmetic with `np.datetime64`

You can perform arithmetic operations with `np.datetime64` objects to compute date differences or add time spans.

```
# Difference between dates
date_diff = date2 - date1
print(f"Days difference: {date_diff}")

# Add days to a date
date3 = date1 + np.timedelta64(10, 'D')  # Add 10 days
print("New date:", date3)
```

Using Time Units with `np.timedelta64`

The `np.timedelta64` object represents differences in time (e.g., days, hours, minutes). This can be useful for manipulating time series data.

```
# Adding hours, minutes, or seconds
date4 = date1 + np.timedelta64(5, 'h')  # Add 5 hours
print("New date after adding hours:", date4)
```

Generating Time Ranges

NumPy can generate time ranges for creating time series data.

```
# Generate a date range with daily frequency
dates = np.arange('2025-01-01', '2025-01-10', dtype='datetime64[D]')
print(dates)
```

Key Takeaways

- **NumPy arrays** are essential for efficient handling of large datasets, allowing vectorized operations and efficient memory use.
- **Missing data** (NaN, inf) can be handled with specialized functions like `np.isnan()`, `np.isinf()`, and replaced or removed as necessary.
- **`np.datetime64`** and **`np.timedelta64`** provide powerful tools for working with time series data, supporting arithmetic and date generation.

These techniques are fundamental for managing and processing real-world datasets in data science and related fields.