Python for Machine Learning

A Comprehensive Introduction

Sayan Chaki

March 12, 2025

NumPy Basics and Applications

- NumPy is the foundational library for numerical and scientific computing in Python.
- Unlike regular Python lists, NumPy arrays are more efficient for numerical operations because they are implemented in C, offering significant performance improvements.
- NumPy provides a powerful data structure called an **ndarray** (N-dimensional array) to handle large datasets.
- NumPy is used extensively for manipulating images, signal processing, scientific computing, and machine learning.
- **Difference between Lists and NumPy Arrays:**
  - Python Lists:
    - Can hold elements of mixed data types (e.g., integers, floats, strings).
    - Slower for numerical operations as they are not optimized for this purpose.
  - NumPy Arrays:
    - Homogeneous data types (all elements must be of the same type).
    - Optimized for numerical operations with vectorized operations.
    - Allow for broadcasting, element-wise operations, and are memory efficient.

Creating and Accessing NumPy Arrays

- NumPy arrays can be created from Python lists or using NumPy functions.
- Arrays can have any number of dimensions.
- NumPy supports slicing and indexing for element access.

```python
import numpy as np

# Creating NumPy arrays from Python lists
arr1 = np.array([1, 2, 3, 4])  # 1D array
arr2 = np.array([[1, 2], [3, 4]])  # 2D array

# Accessing elements using index
print(arr1[0])  # Output: 1 (First element of 1D array)
print(arr2[1, 1])  # Output: 4 (Element at second row,
    second column of 2D array)

# Slicing arrays
print(arr1[1:3])  # Output: [2 3]
print(arr2[:, 1])  # Output: [2 4] (All rows, second column)
```

- **Indexing**: Use square brackets for indexing and slicing, similar to Python lists.

Array Operations and Broadcasting

- NumPy allows vectorized operations for efficient array manipulation.
- Broadcasting makes it easy to perform operations on arrays of different shapes.

```
# Element-wise operations
arr1 = np.array([1, 2, 3, 4])
arr2 = np.array([10, 20, 30, 40])

# Adding two arrays
arr3 = arr1 + arr2  # Output: [11 22 33 44]
print(arr3)

# Scalar operations (multiplying by a scalar)
arr4 = arr1 * 2  # Output: [2 4 6 8]
print(arr4)

# Broadcasting with arrays of different shapes
arr5 = np.array([1, 2, 3])  # Shape (3,)
arr6 = np.array([[1], [2], [3]])  # Shape (3, 1)

arr7 = arr5 + arr6  # Broadcasting to shape (3, 3)
print(arr7)
```

Array Aggregations and Statistical Operations

- NumPy provides efficient functions to perform aggregations such as sum, mean, min, and max.
- Statistical operations can be computed along specific axes or on the entire array.

```
# Aggregation functions
arr = np.array([1, 2, 3, 4, 5])

# Sum of all elements
sum_arr = np.sum(arr)  # Output: 15
print(sum_arr)

# Mean of all elements
mean_arr = np.mean(arr)  # Output: 3.0
print(mean_arr)

# Min and Max of the array
min_arr = np.min(arr)  # Output: 1
max_arr = np.max(arr)  # Output: 5
print(min_arr, max_arr)

```

Reshaping and Transposing Arrays

- Reshaping arrays allows changing the dimensions while maintaining the same number of elements.
- Transposing swaps the axes of a 2D array.

```
# Reshaping an array
arr = np.array([1, 2, 3, 4, 5, 6])

# Reshaping into a 2x3 array
reshaped_arr = arr.reshape(2, 3)
print(reshaped_arr)
# Output: [[1 2 3]
#          [4 5 6]]

# Transposing an array (for 2D arrays)
transposed_arr = reshaped_arr.T
print(transposed_arr)
# Output: [[1 4]
#          [2 5]
#          [3 6]]
```

- **Reshaping**: The ‘reshape()‘ function is used to change the dimensions of an array

NumPy Random Functions

- NumPy includes a ‘random‘ module to generate random numbers and sample from distributions.
- This is useful in simulations, data sampling, and testing.

```python
# Generating random numbers
rand_arr = np.random.rand(2, 3)  # Random numbers between 0 and 1
print(rand_arr)

# Generating random integers
rand_int_arr = np.random.randint(0, 10, size=(2, 3))  # Integers between 0 and 10
print(rand_int_arr)

# Random numbers from a normal distribution
normal_arr = np.random.randn(3, 3)  # Normal distribution with mean 0 and std 1
print(normal_arr)
```

- **Random Numbers**: Functions like ‘np.random.rand()‘, ‘np.random.randint()‘, and ‘np.random.randn()‘ help generate random values for simulations or data generation.

Concatenation and Stacking of Arrays

- NumPy allows joining arrays using concatenation or stacking.
- This is useful for combining datasets or splitting arrays into smaller pieces.

```
# Concatenating arrays
arr1 = np.array([1, 2, 3])
arr2 = np.array([4, 5, 6])

# Concatenation along a single axis (1D arrays)
concatenated_arr = np.concatenate((arr1, arr2))
print(concatenated_arr)  # Output: [1 2 3 4 5 6]

# Stacking arrays vertically (axis=0)
stacked_arr = np.vstack((arr1, arr2))
print(stacked_arr)
# Output: [[1 2 3]
#          [4 5 6]]

# Stacking arrays horizontally (axis=1)
stacked_arr_h = np.hstack((arr1, arr2))
print(stacked_arr_h)  # Output: [1 2 3 4 5 6]
```

NumPy in Image Processing

- NumPy is widely used for processing images in computer vision tasks.
- Images are represented as multi-dimensional arrays (2D for grayscale, 3D for RGB).
- NumPy arrays enable fast manipulation of pixel data, such as resizing, transforming, and applying filters.

```python
import numpy as np
import matplotlib.pyplot as plt
from skimage import io

# Load an image into a NumPy array
image = io.imread('image.jpg')

# Access pixel data and manipulate
grayscale_image = np.mean(image, axis=2)  # Convert to
    grayscale by averaging RGB channels

# Show the original and grayscale images
plt.subplot(1, 2, 1)
plt.imshow(image)
plt.title('Original Image')
```

NumPy for Other Data Types

- NumPy is also used for handling other types of data like time series, sensor data, and signal processing.
- Its powerful indexing and slicing features make it ideal for handling structured data in an efficient manner.
- NumPy can be used in machine learning for tasks like feature extraction, data transformation, and handling large datasets.

```python
# Example: Time series data processing
time = np.linspace(0, 10, 100)  # Generate time values from
    0 to 10
signal = np.sin(time)  # Generate sine wave

# Plotting the signal
plt.plot(time, signal)
plt.title('Sine Wave Signal')
plt.xlabel('Time')
plt.ylabel('Amplitude')
plt.show()
```

- Time Series Data: NumPy is perfect for handling continuous data like time series and signals (e.g., sine waves).

Linear Regression: Introduction

- **Linear Regression** is used to model the relationship between a dependent variable (y) and one or more independent variables (x).
- In simple linear regression, the relationship is modeled as:

$$y = \beta_0 + \beta_1 x + \epsilon$$

where:
  - y is the dependent variable (output),
  - x is the independent variable (input),
  - β_0 is the intercept,
  - β_1 is the slope (coefficient),
  - ϵ is the error term (residuals).

The Least Squares Method

- The objective is to find the best-fit line that minimizes the error term (residuals).
- We use the **Least Squares Method** to minimize the sum of squared residuals:

$$\beta_1 = \frac{n \sum_{i=1}^{n} x_i y_i - \sum_{i=1}^{n} x_i \sum_{i=1}^{n} y_i}{n \sum_{i=1}^{n} x_i^2 - (\sum_{i=1}^{n} x_i)^2}$$

- The intercept β_0 is calculated as:

$$\beta_0 = \frac{\sum_{i=1}^{n} y_i - \beta_1 \sum_{i=1}^{n} x_i}{n}$$

Conclusion

- Linear regression is a simple yet powerful technique for modeling relationships between variables.
- Using the basic equations for the slope and intercept, we can easily compute the best-fit line.
- NumPy allows us to perform the necessary calculations efficiently, and we can visualize the results using Matplotlib.
- This approach can be extended to multiple linear regression by adding more features (independent variables).