Machine Learning with Python

Ismail JAMIAI (email: ijamiai@uae.ac.ma)

November 23, 2024

1 Chapter 2: Working with Vectors, Matrices, and Arrays in NumPy

1.1 2.0. Introduction

NumPy is a foundational tool of the Python machine learning stack. NumPy allows for efficient operations on the data structures often used in machine learning: vectors, matrices, and tensors. While NumPy isn't the focus of this course, it will show up frequently in the following chapters. This chapter covers the most common NumPy operations we're likely to run into while working on machine learning workflows.

1.2 2.1. Creating a Vector

1.2.1 Problem

You need to create a vector

1.2.2 Solution

Use NumPy to create a one-dimensional array:

```python
# Load library
import numpy as np

# Create a vector as a row
vector_row = np.array([1, 2, 3])

# Create a vector as a column
vector_column = np.array([[1],
                          [2],
                          [3]])

# View the vectors
print(vector_row)
print(vector_column)
```

1.2.3 Discussion

NumPy's main data structure is the multidimensional array. A vector is just an array with a single dimension. To create a vector, we simply create a one-dimensional array. Just like vectors, these arrays can be represented horizontally (i.e., rows) or vertically (i.e., columns).

1.3 2.2. Creating a Matrix

1.3.1 Problem

You need to create a matrix.

1.3.2 Solution

Use NumPy to create a two-dimensional array:

```python
# Load library
import numpy as np

# Create a matrix
matrix = np.array([[1, 2],
                   [1, 2],
                      [1, 2]])

# View matrix
print(matrix)
```

1.3.3 Discussion

To create a matrix we can use a NumPy two-dimensional array. In our solution, the matrix contains three rows and two columns (a column of 1s and a column of 2s).

NumPy actually has a dedicated matrix data structure:

```
import numpy as np
matrix_object = np.mat([[1, 2],
                        [1, 2],
                        [1, 2]])
  print(matrix_object)
```

However, the matrix data structure is not recommended for two reasons. First, arrays are the defacto standard data structure of NumPy. Second, the vast majority of NumPy operations return arrays, not matrix objects.

1.4 2.3. Creating a Sparse Matrix

1.4.1 Problem

Given data with very few nonzero values, you want to efficiently represent it.

1.4.2 Solution

Create a sparse matrix:

```
#  Load  libraries
import numpy as np
from scipy import sparse

# Create a matrix
```

```
matrix = np.array([[0, 0],
                   [0, 1],
                   [3, 0]])

# Create compressed sparse row (CSR) matrix
matrix_sparse = sparse.csr_matrix(matrix)

# View sparse matrix
print(matrix_sparse)
```

1.4.3 Discussion

A frequent situation in machine learning is having a huge amount of data; however, most of the elements in the data are zeros. For example, imagine a matrix where the columns are every movie on Netflix, the rows are every Netflix user, and the values are how many times a user has watched that particular movie. This matrix would have tens of thousands of columns and millions of rows! However, since most users do not watch most movies, the vast majority of elements would be zero.

A sparse matrix is a matrix in which most elements are 0. Sparse matrices store only nonzero elements and assume all other values will be zero, leading to significant computational savings. In our solution, we created a NumPy array with two nonzero values, then converted it into a sparse matrix. If we view the sparse matrix we can see that only the nonzero values are stored.

There are a number of types of sparse matrices. However, in compressed sparse row (CSR) matrices, (1, 1) and (2, 0) represent the (zero-indexed) indices of the nonzero values 1 and 3, respectively. For example, the element 1 is in the second row and second column. We can see the advantage of sparse matrices if we create a much larger matrix with many more zero elements and then compare this larger matrix with our original sparse matrix:

```
#  Load  libraries
import numpy as np
from scipy import sparse
```

```
# Create larger matrix
matrix_large = np.array([[0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
                         [0, 1, 0, 0, 0, 0, 0, 0, 0, 0],
                         [3, 0, 0, 0, 0, 0, 0, 0, 0, 0]])

# Create compressed sparse row (CSR) matrix
matrix_large_sparse = sparse.csr_matrix(matrix_large)

# View original sparse matrix
print(matrix_large_sparse)
```

As we can see, despite the fact that we added many more zero elements in the larger matrix, its sparse representation is exactly the same as our original sparse matrix. That is, the addition of zero elements did not change the size of the sparse matrix.

As mentioned, there are many different types of sparse matrices, such as compressed sparse column, list of lists, and dictionary of keys. While an explanation of the different types and their implications is outside the scope of this book, it is worth noting that while there is no "best" sparse matrix type, there are meaningful differences among them, and we should be conscious about why we are choosing one type over another.

1.5 2.4. Preallocating NumPy Arrays

1.5.1 Problem

You need to preallocate arrays of a given size with some value.

1.5.2 Solution

NumPy has functions for generating vectors and matrices of any size using 0s, 1s, or values of your choice:

```
  # Load library
  import numpy as np

  # Generate a vector of shape (1,5) containing all zeros
```

```
vector = np.zeros(shape=5)

# View the matrix
print(vector)

# Generate a matrix of shape (3,3) containing all ones
matrix = np.full(shape=(3,3), fill_value=1)

# View the vector
print(matrix)
```

1.5.3 Discussion

Generating arrays prefilled with data is useful for a number of purposes, such as making code more performant or using synthetic data to test algorithms. In many programming languages, preallocating an array of default values (such as 0s) is considered common practice.

1.6 2.5. Selecting Elements

1.6.1 Problem

You need to select one or more elements in a vector or matrix

1.6.2 Solution

NumPy arrays make it easy to select elements in vectors or matrices:

```
# Load library
import numpy as np

# Create row vector
vector = np.array([1, 2, 3, 4, 5, 6])

# Create matrix
matrix = np.array([[1, 2, 3],
                   [4, 5, 6],
                   [7, 8, 9]])
```

```python
# Select third element of vector
print(vector[2])

# Select all elements of a vector
#print(vector[:])
# Select everything up to and including the third element
#print(vector[:3])
# Select everything after the third element
#print(vector[3:])
# Select the last element
#print(vector[-1])
# Reverse the vector
#print(vector[::-1])

# Select second row, second column
print(matrix[1,1])
# Select the first two rows and all columns of a matrix
#print(matrix[:2,:])
# Select all rows and the second column
#print(matrix[:,1:2])
```

1.6.3 Discussion

Like most things in Python, NumPy arrays are zero-indexed, meaning that the index of the first element is 0, not 1. With that caveat, NumPy offers a wide variety of methods for selecting (i.e., indexing and slicing) elements or groups of elements in arrays:

1.7 2.6. Describing a Matrix

1.7.1 Problem

You want to describe the shape, size, and dimensions of a matrix.

1.7.2 Solution

Use the shape, size, and ndim attributes of a NumPy object:

```
# Load library
import numpy as np

# Create matrix
matrix = np.array([[1, 2, 3, 4],
                   [5, 6, 7, 8],
                   [9, 10, 11, 12]])

# View number of rows and columns
print(matrix.shape)

# View number of elements (rows * columns)
print(matrix.size)

# View number of dimensions
print(matrix.ndim)
```

1.7.3 Discussion

This might seem basic (and it is); however, time and again it will be valuable to check the shape and size of an array both for further calculations and simply as a gut check after an operation.

1.8 2.7. Applying Functions over Each Element

1.8.1 Problem

You want to apply some function to all elements in an array.

1.8.2 Solution

Use the NumPy vectorize method:

```
# Load library
import numpy as np

# Create matrix
matrix = np.array([[1, 2, 3],
                   [4, 5, 6],
                      [7, 8, 9]])

#print(matrix + 100)

# Create function that adds 100 to something
add_100 = lambda i: i + 100

# Create vectorized function
vectorized_add_100 = np.vectorize(add_100)

# Apply function to all elements in matrix
print(vectorized_add_100(matrix))
```

1.8.3 Discussion

The NumPy **vectorize** method converts a function into a function that can apply to all elements in an array or slice of an array. It's worth noting that **vectorize** is essentially a **for** loop over the elements and does not increase performance. Furthermore, NumPy arrays allow us to perform operations between arrays even if their dimensions are not the same (a process called broadcasting). For example, we can create a much simpler version of our solution using broadcasting:

Broadcasting does not work for all shapes and situations, but it is a common way of applying simple operations over all elements of a NumPy array.

1.9 2.8. Finding the Maximum and Minimum Values

1.9.1 Problem

You need to find the maximum or minimum value in an array.

1.9.2 Solution

Use NumPy's max and min methods:

```
# Load library
import numpy as np

# Create matrix
matrix = np.array([[1, 2, 3],
                   [4, 5, 6],
                   [7, 8, 9]])
# Return maximum element
print(np.max(matrix))

# Return minimum element
print(np.min(matrix))
```

1.9.3 Discussion

Often we want to know the maximum and minimum value in an array or subset of an array. This can be accomplished with the max and min methods. Using the axis parameter, we can also apply the operation along a certain axis:

```
# Load library
import numpy as np

# Create matrix
matrix = np.array([[1, 2, 3],
                   [4, 5, 6],
                   [7, 8, 9]])
# Find maximum element in each column
print(np.max(matrix, axis=0))

# Find maximum element in each row
print(np.max(matrix, axis=1))
```

1.10 2.9. Calculating the Average, Variance, and Standard Deviation

1.10.1 Problem

You want to calculate some descriptive statistics about an array:

1.10.2 Solution

Use NumPy's mean, var, and std:

```python
# Load library
import numpy as np

# Create matrix
matrix = np.array([[1, 2, 3],
                   [4, 5, 6],
                   [7, 8, 9]])

# Return mean
print(np.mean(matrix))

# Return variance
print(np.var(matrix))

# Return standard deviation
print(np.std(matrix))

# Find the mean value in each column
#print(np.mean(matrix, axis=0))
```

1.10.3 Discussion

Just like with max and min, we can easily get descriptive statistics about the whole matrix or do calculations along a single axis:

1.11 2.10. Reshaping Arrays

1.11.1 Problem

You want to change the shape (number of rows and columns) of an array without changing the element values.

1.11.2 Solution

Use NumPy's reshape:

```python
# Load library
import numpy as np

# Create 4x3 matrix
matrix = np.array([[1, 2, 3],
                   [4, 5, 6],
                   [7, 8, 9],
                   [10, 11, 12]])

# Reshape matrix into 2x6 matrix
print(matrix.reshape(2, 6))

#print(matrix.size)

#print(matrix.reshape(1, -1))

#print(matrix.reshape(12))
```

1.11.3 Discussion

Reshape allows us to restructure an array so that we maintain the same data but organize it as a different number of rows and columns. The only requirement is that the shape of the original and new matrix contain the same number of elements (i.e., are the same size). We can see the size of a matrix using size:

One useful argument in reshape is -1, which effectively means "as many as needed," so reshape(1, -1) means one row and as many columns as needed:

Finally, if we provide one integer, reshape will return a one-dimensional array of that length:

1.12 2.11. Transposing a Vector or Matrix

1.12.1 Problem

You need to transpose a vector or matrix.

1.12.2 Solution

Use the **T** method:

```python
# Load library
import numpy as np

# Create matrix
matrix = np.array([[1, 2, 3],
                   [4, 5, 6],
                   [7, 8, 9]])

# Transpose matrix
print(matrix.T)

# Transpose vector
#print(np.array([1, 2, 3, 4, 5, 6]).T)

# Transpose row vector
#print(np.array([[1, 2, 3, 4, 5, 6]]).T)
```

1.12.3 Discussion

Transposing is a common operation in linear algebra where the column and row indices of each element are swapped. A nuanced point typically overlooked outside of a linear algebra class is that, technically, a vector can't be transposed because it's just a collection of values:

However, it is common to refer to transposing a vector as converting a row vector to a column vector (notice the second pair of brackets) or vice versa:

1.13 2.12. Flattening a Matrix

1.13.1 Problem

You need to transform a matrix into a one-dimensional array.

1.13.2 Solution

Use the **flatten** method:

```
# Load library
import numpy as np

# Create matrix
matrix = np.array([[1, 2, 3],
                   [4, 5, 6],
                   [7, 8, 9]])

# Flatten matrix
print(matrix.flatten())
#print(matrix.reshape(1, -1))
```

1.13.3 Discussion

flatten is a simple method to transform a matrix into a one-dimensional array. Alternatively, we can use reshape to create a row vector:

Another common way to flatten arrays is the **ravel** method. Unlike **flatten**, which returns a copy of the original array, **ravel** operates on the original object itself and is therefore slightly faster. It also lets us **flatten** lists of arrays, which we can't do with the flatten method. This operation is useful for flattening very large arrays and speeding up code:

```
# Load library
import numpy as np

# Create one matrix
matrix_a = np.array([[1, 2],
                     [3, 4]])

# Create a second matrix
matrix_b = np.array([[5, 6],
                     [7, 8]])

# Create a list of matrices
matrix_list = [matrix_a, matrix_b]
```

```
# Flatten the entire list of matrices
print(np.ravel(matrix_list))
```

1.14 2.13. Finding the Rank of a Matrix

1.14.1 Problem

You need to know the rank of a matrix.

1.14.2 Solution

Use NumPy's linear algebra method $\mathbf{matrix_{rank}}$:

```
    # Load library
    import numpy as np

    # Create matrix
    matrix = np.array([[1, 1, 1],
                       [1, 1, 10],
                       [1, 1, 15]])

    # Return matrix rank
    print(np.linalg.matrix_rank(matrix))
```

1.14.3 Discussion

The rank of a matrix is the dimensions of the vector space spanned by its columns or rows. Finding the rank of a matrix is easy in NumPy thanks to $\mathbf{matrix_{rank}}$.

1.15 2.14. Getting the Diagonal of a Matrix

1.15.1 Problem

You need to get the diagonal elements of a matrix.

1.15.2 Solution

Use NumPy's **diagonal**:

```
# Load library
import numpy as np

# Create matrix
matrix = np.array([[1, 2, 3],
                   [2, 4, 6],
                   [3, 8, 9]])

# Return diagonal elements
print(matrix.diagonal())

# Return diagonal one above the main diagonal
#print(matrix.diagonal(offset=1))

# Return diagonal one below the main diagonal
#print(matrix.diagonal(offset=-1))
```

1.15.3 Discussion

NumPy makes getting the diagonal elements of a matrix easy with **diagonal**. It is also possible to get a diagonal off the main diagonal by using the **offset** parameter:

1.16 2.15. Calculating the Trace of a Matrix

1.16.1 Problem

You need to calculate the trace of a matrix.

1.16.2 Solution

Use **trace**:

```
# Load library
import numpy as np
```

```
# Create matrix
matrix = np.array([[1, 2, 3],
                   [2, 4, 6],
                   [3, 8, 9]])

# Return trace
print(matrix.trace())

# Return diagonal and sum elements
#print(sum(matrix.diagonal()))
```

1.16.3 Discussion

The trace of a matrix is the sum of the diagonal elements and is often used under the hood in machine learning methods. Given a NumPy multidimensional array, we can calculate the trace using trace. Alternatively, we can return the diagonal of a matrix and calculate its sum:

1.17 2.16. Calculating Dot Products

1.17.1 Problem

You need to calculate the dot product of two vectors.

1.17.2 Solution

Use NumPy's **dot** function:

```
# Load library
import numpy as np

# Create two vectors
vector_a = np.array([1, 2, 3])
vector_b = np.array([4, 5, 6])

# Calculate dot product
print(np.dot(vector_a, vector_b))
#print(vector_a @ vector_b)
```

1.17.3 Discussion

The dot product of two vectors, a and b, is defined as:

$$< a, b >= \sum_{i=1}^{n} a_i b_i$$

where a_i is the ith element of vector a and b_i is the ith element of vector b. We can use NumPy's **dot** function to calculate the dot product. Alternatively, in Python 3.5+ we can use the new @ operator:

1.18 2.17. Adding and Subtracting Matrices

1.18.1 Problem

You want to add or subtract two matrices.

1.18.2 Solution

Use NumPy's add and **subtract**

```
# Load library
import numpy as np

# Create matrix
matrix_a = np.array([[1, 1, 1],
                     [1, 1, 1],
                     [1, 1, 2]])

# Create matrix
matrix_b = np.array([[1, 3, 1],
                     [1, 3, 1],
                     [1, 3, 8]])

# Add two matrices
print(np.add(matrix_a, matrix_b))

# Subtract two matrices
print(np.subtract(matrix_a, matrix_b))
```

```
# Add two matrices
#print(matrix_a + matrix_b)
```

1.18.3 Discussion

Alternatively, we can simply use the + and − operators:

1.19 2.18. Multiplying Matrices

1.19.1 Problem

You want to multiply two matrices.

1.19.2 Solution

Use NumPy's **dot**:

```
# Load library
import numpy as np

# Create matrix
matrix_a = np.array([[1, 1],
                     [1, 2]])

# Create matrix
matrix_b = np.array([[1, 3],
                     [1, 2]])

# Multiply two matrices
print(np.dot(matrix_a, matrix_b))

# Multiply two matrices
#print(matrix_a @ matrix_b)

# Multiply two matrices element-wise
#print(matrix_a * matrix_b)
```

1.19.3 Discussion

Alternatively, in Python 3.5+ we can use the @ operator:
If we want to do element-wise multiplication, we can use the $*$ operator:

1.20 2.19. Inverting a Matrix

1.20.1 Problem

You want to calculate the inverse of a square matrix.

1.20.2 Solution

Use NumPy's linear algebra **inv** method:

```python
# Load library
import numpy as np

# Create matrix
matrix = np.array([[1, 4],
                   [2, 5]])

# Calculate inverse of matrix
print(np.linalg.inv(matrix))

# Multiply matrix and its inverse
#print(matrix @ np.linalg.inv(matrix))
```

1.20.3 Discussion

The inverse of a square matrix, A, is a second matrix, A^{-1}, such that:

$$AA^{-1} = I$$

where I is the identity matrix. In NumPy we can use **linalg.inv** to calculate A^{-1} if it exists. To see this in action, we can multiply a matrix by its inverse, and the result is the identity matrix:

1.21 2.20. Generating Random Values

1.21.1 Problem

You want to generate pseudorandom values.

1.21.2 Solution

Use NumPy's random:

```
# Load library
import numpy as np

# Set seed
np.random.seed(0)

# Generate three random floats between 0.0 and 1.0
print(np.random.random(3))

# Generate three random integers between 0 and 10
#print(np.random.randint(0, 11, 3))

# Draw three numbers from a normal distribution with mean 0.0
# and standard deviation of 1.0
#print(np.random.normal(0.0, 1.0, 3))

# Draw three numbers from a logistic distribution with mean 0.0 and
↪  scale of 1.0
#print(np.random.logistic(0.0, 1.0, 3))

# Draw three numbers greater than or equal to 1.0 and less than 2.0
#print(np.random.uniform(1.0, 2.0, 3))
```

1.21.3 Discussion

NumPy offers a wide variety of means to generate random numbers—many more than can be covered here. In our solution we generated floats; however, it is also common to generate integers:

Alternatively, we can generate numbers by drawing them from a distribution (note this is not technically random):

Finally, sometimes it can be useful to return the same random numbers multiple times to get predictable, repeatable results. We can do this by setting the "seed" (an integer) of the pseudorandom generator. Random processes with the same seed will always produce the same output. We will use seeds throughout this book so that the code you see in the book and the code you run on your computer produces the same results.