Chapter 1. Vectors, Matrices, and Arrays

1.0 Introduction

NumPy is the foundation 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 is not the focus of this book, it will show up frequently throughout the following chapters. This chapter covers the most common NumPy operations we are likely to run into while working on machine learning workflows.

Problem

You need to create a vector.

Solution

Use NumPy to create a one-dimensional array:

```
# 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]])
```

Discussion

NumPy's main data structure is the multidimensional array. 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).

- Vectors, Math is Fun
- Euclidean vector, Wikipedia

1.2 Creating a Matrix

Problem

You need to create a matrix.

Solution

Use NumPy to create a two-dimensional array:

```
# Load library
import numpy as np

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

two columns (a column of 1s and a column of 2s).

NumPy actually has a dedicated matrix data structure:

```
matrix_object = np.mat([[1, 2],
                        [1, 2],
                        [1, 2]])
```

```
matrix([[1, 2],
        [1, 2],
        [1, 2]])
```

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

See Also

- Matrix, Wikipedia
- Matrix, Wolfram MathWorld

Problem

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

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)
```

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

Sparse matrices only store 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:

```
# View sparse matrix
print(matrix_sparse)
```

```
  (1, 1)    1
  (2, 0)    3
```

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 non-zero 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:

```
# 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]])
```

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

```
  (1, 1)    1
  (2, 0)    3
```

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

```
  (1, 1)    1
  (2, 0)    3
```

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 between them and we should be conscious about why we are choosing one type over another.

- Sparse matrices, SciPy documentation
- 101 Ways to Store a Sparse Matrix

1.4 Selecting Elements

Problem

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

Solution

NumPy's arrays make that easy:

```
# 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],
```

```
vector[2]
```

```
3
```

```
# Select second row, second column
matrix[1,1]
```

```
5
```

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:

```
# Select all elements of a vector
vector[:]
```

```
array([1, 2, 3, 4, 5, 6])
```

```
array([1, 2, 3])
```

```
# Select everything after the third element
vector[3:]
```

```
array([4, 5, 6])
```

```
# Select the last element
vector[-1]
```

```
6
```

```
# Select the first two rows and all columns of a matrix
matrix[:2,:]
```

```
array([[1, 2, 3],
       [4, 5, 6]])
```

```
array([[2],
       [5],
       [8]])
```

1.5 Describing a Matrix

Problem

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

Solution

Use `shape`, `size`, and `ndim`:

```
# Load library
import numpy as np

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

```
(3, 4)
```

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

```
12
```

```
# View number of dimensions
matrix.ndim
```

```
2
```

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 some operation.

Problem

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

Solution

Use NumPy's `vectorize`:

```
# Load library
import numpy as np

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

# 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
vectorized_add_100(matrix)
```

Discussion

NumPy's `vectorize` class 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:

```
# Add 100 to all elements
matrix + 100
```

```
array([[101, 102, 103],
       [104, 105, 106],
       [107, 108, 109]])
```

1.7 Finding the Maximum and Minimum Values

Problem

Solution

Use NumPy's `max` and `min`:

```
# Load library
import numpy as np

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

# Return maximum element
np.max(matrix)
```

```
9
```

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

```
1
```

with the `max` and `min` methods. Using the `axis` parameter we can also apply the operation along a certain axis:

```
# Find maximum element in each column
np.max(matrix, axis=0)
```

```
array([7, 8, 9])
```

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

```
array([3, 6, 9])
```

1.8 Calculating the Average, Variance, and Standard Deviation

Problem

You want to calculate some descriptive statistics about an array.

```
# Load library
import numpy as np

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

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

```
5.0
```

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

```
6.666666666666667
```

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

Discussion

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

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

```
array([ 4.,  5.,  6.])
```

1.9 Reshaping Arrays

Problem

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

Solution

Use NumPy's `reshape`:

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

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

```
array([[ 1,  2,  3,  4,  5,  6],
       [ 7,  8,  9, 10, 11, 12]])
```

Discussion

`reshape` allows us to restructure an array so that we maintain the same data but it is organized 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., the same size). We can see the size of a matrix using `size`:

```
matrix.size
```

```
12
```

```
matrix.reshape(1, -1)
```

```
array([[ 1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12]])
```

Finally, if we provide one integer, `reshape` will return a 1D array of that length:

```
matrix.reshape(12)
```

```
array([ 1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12])
```

1.10 Transposing a Vector or Matrix

Problem

You need to transpose a vector or matrix.

Solution

```
# Load library
import numpy as np

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

# Transpose matrix
matrix.T
```

```
array([[1, 4, 7],
       [2, 5, 8],
       [3, 6, 9]])
```

Discussion

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

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

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:

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

```
array([[1],
       [2],
       [3],
       [4],
       [5],
       [6]])
```

1.11 Flattening a Matrix

Problem

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

Solution

```
# Load library
import numpy as np

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

# Flatten matrix
matrix.flatten()
```

```
array([1, 2, 3, 4, 5, 6, 7, 8, 9])
```

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:

```
matrix.reshape(1, -1)
```

```
array([[1, 2, 3, 4, 5, 6, 7, 8, 9]])
```

Problem

You need to know the rank of a matrix.

Solution

Use NumPy's linear algebra method `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
np.linalg.matrix_rank(matrix)
```

```
2
```

Discussion

See Also

- The Rank of a Matrix, CliffsNotes

1.13 Calculating the Determinant

Problem

You need to know the determinant of a matrix.

Solution

Use NumPy's linear algebra method `det`:

```
# Load library
import numpy as np

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

```
0.0
```

Discussion

It can sometimes be useful to calculate the determinant of a matrix. NumPy makes this easy with `det`.

See Also

- The determinant | Essence of linear algebra, chapter 5, 3Blue1Brown
- Determinant, Wolfram MathWorld

1.14 Getting the Diagonal of a Matrix

Problem

You need to get the diagonal elements of a matrix.

```
# Load library
import numpy as np

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

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

```
array([1, 4, 9])
```

Discussion

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

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

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

```
array([2, 8])
```

1.15 Calculating the Trace of a Matrix

Problem

You need to calculate the trace of a matrix.

Solution

Use `trace`:

```
# Load library
import numpy as np

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

```
matrix.trace()
```

```
14
```

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`. We can also return the diagonal of a matrix and calculate its sum:

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

```
14
```

See Also

- The Trace of a Square Matrix

Problem

You need to find the eigenvalues and eigenvectors of a square matrix.

Solution

Use NumPy's `linalg.eig`:

```
# Load library
import numpy as np

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

# Calculate eigenvalues and eigenvectors
eigenvalues, eigenvectors = np.linalg.eig(matrix)
```

```
# View eigenvalues
eigenvalues
```

```
# View eigenvectors
eigenvectors
```

```
array([[-0.17622017, -0.96677403, -0.53373322],
       [-0.435951  ,  0.2053623 , -0.64324848],
       [-0.88254925,  0.15223105,  0.54896288]])
```

Discussion

Eigenvectors are widely used in machine learning libraries. Intuitively, given a linear transformation represented by a matrix, A, eigenvectors are vectors that, when that transformation is applied, change only in scale (not direction). More formally:

$$Av = \lambda v$$

where A is a square matrix, λ contains the eigenvalues and v contains the eigenvectors. In NumPy's linear algebra toolset, `eig` lets us calculate the eigenvalues, and eigenvectors of any square matrix.

See Also

- Eigenvectors and Eigenvalues Explained Visually, Setosa.io

1.17 Calculating Dot Products

Problem

You need to calculate the dot product of two vectors.

Solution

Use NumPy's `dot`:

```
# 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
np.dot(vector_a, vector_b)
```

```
32
```

$$\sum_{i=1}^{n} a_i b_i$$

where *ai* is the *i*th element of vector *a*. We can use NumPy's `dot` function to calculate the dot product. Alternatively, in Python 3.5+ we can use the new `@` operator:

```
# Calculate dot product
vector_a @ vector_b
```

```
32
```

See Also

- Vector dot product and vector length, Khan Academy
- Dot Product, Paul's Online Math Notes

1.18 Adding and Subtracting Matrices

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
np.add(matrix_a, matrix_b)
```

```
array([[ 2,  4,  2],
       [ 2,  4,  2],
       [ 2,  4, 10]])
```

```
array([[ 0, -2,  0],
       [ 0, -2,  0],
       [ 0, -2, -6]])
```

Discussion

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

```
# Add two matrices
matrix_a + matrix_b
```

```
array([[ 2,  4,  2],
       [ 2,  4,  2],
       [ 2,  4, 10]])
```

1.19 Multiplying Matrices

Problem

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
np.dot(matrix_a, matrix_b)
```

```
array([[2, 5],
       [3, 7]])
```

Discussion

Alternatively, in Python 3.5+ we can use the @ operator:

```
array([[2, 5],
       [3, 7]])
```

If we want to do element-wise multiplication, we can use the * operator:

```
# Multiply two matrices element-wise
matrix_a * matrix_b
```

```
array([[1, 3],
       [1, 4]])
```

See Also

- Array vs. Matrix Operations, MathWorks

1.20 Inverting a Matrix

Problem

Solution

Use NumPy's linear algebra `inv` method:

```
# Load library
import numpy as np

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

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

```
array([[-1.66666667,  1.33333333],
       [ 0.66666667, -0.33333333]])
```

Discussion

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

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

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

```
array([[ 1.,  0.],
       [ 0.,  1.]])
```

See Also

- Inverse of a Matrix

1.21 Generating Random Values

Problem

You want to generate pseudorandom values.

Solution

Use NumPy's `random`:

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

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

```
array([ 0.5488135 ,  0.71518937,  0.60276338])
```

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:

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

```
array([3, 7, 9])
```

Alternatively, we can generate numbers by drawing them from a distribution:

```
array([-1.42232584,  1.52006949, -0.29139398])
```

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

```
array([-0.98118713, -0.08939902,  1.46416405])
```

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

```
array([ 1.47997717,  1.3927848 ,  1.83607876])
```

Finally, it can sometimes 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.