Machine Learning Lab

Exercise 1

Vectors, Matrices, and Arrays

1.0 Introduction

Numpy is the foundation of the Python machine learning stack. It allows for efficient operations on the data structures often used in machine learning: vectors, matricies, and tensors.

This exercise covers the most common NumPy operations we are likely to run into

1.1 Creating a Vector

Problem

You need to create a vector

Solution

Use Numpy to create a one-dimensional array

In [1]:
```
# load library
import numpy as np

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

# create a vetor as a column
vector_column = np.array([[1],
                          [2],
                          [3]])
```

Discussion

Numpy's main data structure is the multidimensional array

See Also

- Vectors, Math is Fun (https://www.mathsisfun.com/algebra/vectors.html (https://www.mathsisfun.com/algebra/vectors.html))
- Euclidian vector, Wikipedia (https://en.wikipedia.org/wiki/Euclidean_vector (https://en.wikipedia.org/wiki/Euclidean_vector))

1.2 Creating a Matrix

Problem

You need to create a matrix.

Solution

Use Numpy to create a two-dimensional array:

In [2]:
```
# load library
import numpy as np

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

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:

In [10]:
```
matrix_object = np.mat([[1, 2],
                        [1, 2],
                        [1, 2]])
```

However the matrix data structure is not recommended for two reaons. 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 (https://en.wikipedia.org/wiki/Matrix_(mathematics (https://en.wikipedia.org/wiki/Matrix_(mathematics))
- Matrix, Wolfram MathWorld (http://mathworld.wolfram.com/Matrix.html (http://mathworld.wolfram.com/Matrix.html))

1.3 Creating a Sparse Matrix

Problem

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

Solution

Create a sparse matrix:

In [6]:
```
# 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 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.

Sparse matricies 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:

In [7]:
```
# 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:

In [9]:
```
# 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_sparse)
```

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

In [10]:
```
# 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 exercise, 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.

See Also

- Sparse matrices, SciPy documentation (https://docs.scipy.org/doc/scipy/reference/sparse.html (https://docs.scipy.org/doc/scipy/reference/sparse.html))
- 101 Ways to Store a Sparse Matrix (https://medium.com/@jmaxg3/101-ways-to-store-a-sparse-matrix-c7f2bf15a229 (https://medium.com/@jmaxg3/101-ways-to-store-a-sparse-matrix-c7f2bf15a229))

1.4 Selected Elements

Problem

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

Solution

NumPy's arrays make that easy

In [14]:
```
# 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]])

# select the third element of vector
vector[2]
```

Out[14]: 3

In [15]:
```
# select second row, second column
matrix[1,1]
```

Out[15]: 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:

```
In [16]: # Select all elements of a vector
         vector[:]
```

```
Out[16]: array([1, 2, 3, 4, 5, 6])
```

```
In [18]: # select everything up to and including the third element
         vector[:3]
```

```
Out[18]: array([1, 2, 3])
```

```
In [19]: # select the last element
         vector[-1]
```

```
Out[19]: 6
```

```
In [20]: # select the first two rows and all columns of a matrix
         matrix[:2, :]
```

```
Out[20]: array([[1, 2, 3],
                [4, 5, 6]])
```

```
In [22]: # select all rows and the second column
         matrix[:,1:2]
```

```
Out[22]: 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:

```
In [24]: # 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
         matrix.shape
```

```
Out[24]: (3, 4)
```

```
In [25]: # view number of elements (rows * columns)
         matrix.size
```

```
Out[25]: 12
```

```
In [26]: # view number of dimensions
         matrix.ndim
```

```
Out[26]: 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

1.6 Applying Operations to Elements

Problem

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

Solutions

Use NumPy's vectorize:

In [27]:
```
# load library
import numpy as np

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

# create function that adds 1000 to something
add_1000 = lambda i: i + 1000

# create vectorized function
vectorized_add_1000 = np.vectorize(add_1000)

# apply function to all elementsin matrix
vectorized_add_1000(matrix)
```

Out[27]:
```
array([[1001, 1002, 1003],
       [1004, 1005, 1006],
       [1007, 1008, 1009]])
```

Discusion

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:

In [28]:
```
# add 1000 to all elements
matrix + 1000
```

Out[28]:
```
array([[1001, 1002, 1003],
       [1004, 1005, 1006],
       [1007, 1008, 1009]])
```

Finding Maximum and Minimum Values

Problem

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

Solution

Use NumPy's max and min:

In [29]:
```
# load library
import numpy as np

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

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

Out[29]: 9

In [30]:
```
# return minimum element
np.min(matrix)
```

Out[30]: 1

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:

In [31]:
```
# find maximum element in each column
np.max(matrix, axis=0)
```

Out[31]: array([7, 8, 9])

In [32]:
```
# find maximum element in each row
np.max(matrix, axis=1)
```

Out[32]: array([3, 6, 9])

1.8 Calculating the Average, Variance, and Standard Deviation

Problem

You want to calculate some descriptive statistics about an array.

Solution

Use NumPy's mean, var, and std:

In [33]:
```
# 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)
```

Out[33]: 5.0

In [34]:
```
# return variance
np.var(matrix)
```

Out[34]: 6.666666666666667

In [35]:
```
# return standard deviation
np.std(matrix)
```

Out[35]: 2.581988897471611

Discussion

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

```
In [36]: # find the mean value in each column
         np.mean(matrix, axis=0)
```

```
Out[36]: 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:

```
In [37]: # 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
         matrix.reshape(2, 6)
```

```
Out[37]: 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:

```
In [38]: matrix.size
```

```
Out[38]: 12
```

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:

```
In [39]: matrix.reshape(1, -1)
```

```
Out[39]: 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:

```
In [40]: matrix.reshape(12)
```

```
Out[40]: 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

Use the T method:

In [41]:
```
# load library
import numpy as np

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

# transpose matrix
matrix.T
```

Out[41]:
```
array([[1, 4, 7],
       [2, 5, 8],
       [3, 6, 9]])
```

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:

In [42]:
```
# transpose vector
np.array([1, 2, 3, 4, 5, 6]).T
```

Out[42]:
```
array([1, 2, 3, 4, 5, 6])
```

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:

In [43]:
```
# transpose row vector
np.array([[1, 2, 3, 4, 5, 6]]).T
```

Out[43]:
```
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

Use flatten:

In [44]:
```
# load library
import numpy as np

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

# flatten matrix
matrix.flatten()
```

Out[44]:
```
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:

In [45]:
```
matrix.reshape(1, -1)
```

Out[45]:
```
array([[1, 2, 3, 4, 5, 6, 7, 8, 9]])
```

1.12 Finding the Rank of a Matrix

Problem

You need to know the rank of a matrix

Solution

Use NumPy's linear algebra method matrix_rank:

In [46]:
```
# 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)
```

Out[46]:
```
2
```

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 matrix_rank.

See Also

- The Rank of a Matrix, CliffsNotes (https://www.cliffsnotes.com/study-guides/algebra/linear-algebra/real-euclidean-vector-spaces/the-rank-of-a-matrix (https://www.cliffsnotes.com/study-guides/algebra/linear-algebra/real-euclidean-vector-spaces/the-rank-of-a-matrix))

1.13 Calculating the Determinant

Problem

You need to know the determinant of a matrix

Solution

Use NumPy's linear algebra method det:

In [48]:
```
# load library
import numpy as np

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

# return the determinant of matrix
np.linalg.det(matrix)
```

Out[48]:
```
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, Blue1Brown (https://www.youtube.com/watch?v=Ip3X9LOh2dk (https://www.youtube.com/watch?v=Ip3X9LOh2dk))
- Determinant, Wolfram MathWorld (http://mathworld.wolfram.com/Determinant.html (http://mathworld.wolfram.com/Determinant.html))

1.14 Getting the Diagonal of a Matrix

Problem

You need to get the diagonal elements of matrix.

Solution

Use diagonal:

In [49]:
```
# 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()
```

Out[49]:
```
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:

In [50]:
```
# return diagonal one above the main diagonal
matrix.diagonal(offset=1)
```

Out[50]:
```
array([2, 6])
```

In [52]:
```
# return diagonal one below the main diagonal
matrix.diagonal(offset=-1)
```

Out[52]:
```
array([2, 8])
```

1.15 Calculating the Trace of a Matrix

Problem

You need to calculate the trace of a matrix

Solution

Use trace:

In [53]:
```
# load library
import numpy as np

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

# return trace
matrix.trace()
```

Out[53]:
```
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:

In [54]:
```
# return diagonal and sum elements
sum(matrix.diagonal())
```

Out[54]:
```
14
```

See Also

- The Trace of a Square Matrix (http://mathonline.wikidot.com/the-trace-of-a-square-matrix (http://mathonline.wikidot.com/the-trace-of-a-square-matrix))

1.16 Finding Eigenvalues and Eigenvectors

Problem

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

Solution

Use NumPy's linalg.eig:

In [56]:
```
# 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
```

Out[56]:
```
array([13.55075847,  0.74003145, -3.29078992])
```

In [57]:
```
# view eigenvectors
eigenvectors
```

Out[57]:
```
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 (http://setosa.io/ev/eigenvectors-and-eigenvalues/ (http://setosa.io/ev/eigenvectors-and-eigenvalues/))
- Eigenvectors and eigenvalues | Essence of linear algebra, 3Blue1Brown (https://www.youtube.com/watch?v=PFDu9oVAE-g (https://www.youtube.com/watch?v=PFDu9oVAE-g))

1.17 Calculating Dot Products

Problem

You need to calculate the dot product of two vectors.

Solution

```
In [60]: # 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)

Out[60]: 32
```

Discussion

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

$$\sum(a_i * b_i)$$

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

```
In [61]: # calculate dot product
         vector_a @ vector_b

Out[61]: 32
```

See Also

- Vector dot product and vector length, Khan Academy (https://www.khanacademy.org/math/linear-algebra/vectors-and-spaces/dot-cross-products/v/vector-dot-product-and-vector-length (https://www.khanacademy.org/math/linear-algebra/vectors-and-spaces/dot-cross-products/v/vector-dot-product-and-vector-length))
- Dot Product, Paul's Online Math Notes (http://tutorial.math.lamar.edu/Classes/CalcII/DotProduct.aspx (http://tutorial.math.lamar.edu/Classes/CalcII/DotProduct.aspx))

1.18 Adding and Subtracting Matricies

Problem

You want to add or subtract two matricies

Solution

Use NumPy's add and subtract:

In [76]:
```
# load library
import numpy as np

# create matricies
matrix_a = np.array([[1, 1, 1],
                     [1, 1, 1],
                     [1, 1, 2]])

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

# add two matricies
np.add(matrix_a, matrix_b)
```

Out[76]:
```
array([[ 2,  4,  2],
       [ 2,  4,  2],
       [ 2,  4, 10]])
```

In [77]:
```
# subtract two matrices
np.subtract(matrix_a, matrix_b)
```

Out[77]:
```
array([[ 0, -2,  0],
       [ 0, -2,  0],
       [ 0, -2, -6]])
```

Discussion

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

In [78]:
```
# add two matricies
matrix_a + matrix_b
```

Out[78]:
```
array([[ 2,  4,  2],
       [ 2,  4,  2],
       [ 2,  4, 10]])
```

1.19 Multiplying Matricies

Problem

You want to multiply two matrices.

Solution

Use NumPy's dot:

In [79]:
```
# load library
import numpy as np

# create matrices
matrix_a = np.array([[1, 1],
                     [1, 2]])

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

# multiply two matrices
np.dot(matrix_a, matrix_b)
```

Out[79]:
```
array([[2, 5],
       [3, 7]])
```

Discussion

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

In [80]:
```
# multiply two matrices
matrix_a @ matrix_b
```

Out[80]:
```
array([[2, 5],
       [3, 7]])
```

See Also

- Array vs Matrix Operations, MathWorks (https://www.mathworks.com/help/matlab/matlab_prog/array-vs-matrix-operations.html?requestedDomain=true (https://www.mathworks.com/help/matlab/matlab_prog/array-vs-matrix-operations.html?requestedDomain=true))

1.20 Inverting a Matrix

Problem

You want to calculate the inverse of a square matrix.

Solution

Use NumPy's linear algebra inv method:

In [64]:
```
# load library
import numpy as np

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

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

Out[64]:
```
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:

$$A * A^{-1} = I$$

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:

In [65]:
```
matrix @ np.linalg.inv(matrix)
```

Out[65]:
```
array([[1., 0.],
       [0., 1.]])
```

See Also

- Inverse of a Matrix (http://www.mathwords.com/i/inverse_of_a_matrix.htm (http://www.mathwords.com/i/inverse_of_a_matrix.htm))

1.21 Generating Random Values

Problem

You want to generate pseudorandom values.

Solution

Use NumPy's random:

In [69]:
```
# load library
import numpy as np

# set seed
np.random.seed(0)

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

Out[69]:
```
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:

In [70]:
```
# genereate three random integers between 1 and 10
np.random.randint(0, 11, 3)
```

Out[70]:
```
array([3, 7, 9])
```

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

In [71]:
```
# draw three numbers from a normal distribution with mean 0.0
# and standard deviation of 1.0
np.random.normal(0.0, 1.0, 3)
```

Out[71]:
```
array([-1.42232584,  1.52006949, -0.29139398])
```

In [73]:
```
# draw three numbers from a logistic distribution with mean 0.0 and scale of 1.0
np.random.logistic(0.0, 1.0, 3)
```

Out[73]:
```
array([-0.98118713, -0.08939902,  1.46416405])
```

In [75]:
```
# draw three numbers greater than or equal to 1.0 and less than 2.0
np.random.uniform(1.0, 2.0, 3)
```

Out[75]:
```
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 so that the code you see in these lab exercises and the code you run on your computer produces the same results.

In [ ]:
```

```