Introduction to Machine Learning

COMP70050 Autumn Term 2021/2022

Introduction to NumPy

Hello! It's me, Josiah, and I will be your guide through this tutorial.

In this tutorial, we will look at `NumPy`, a Python package that is used for scientific computing.

I have prepared this module as an introduction. I will not be able to cover every single feature of NumPy (and won't), so please do refer to the official documentation to explore all the nitty gritty details that NumPy offers. I will only focus on features that you will most likely use to get you through most tasks.

NumPy

`NumPy` is a Python scientific computing package, which allows you to do large-scale multi-dimensional array operations easily and efficiently, and provides many Mathematical and Scientific libraries to help you perform scientific computations easily.

If you are an experienced MATLAB user

For the MATLAB/Octave users among you, everything will seem familiar. NumPy tries to emulate the features in MATLAB, but in a Pythonic way. The two main things you have to remember as MATLAB users are:

- indices start at `0` in `NumPy`, not `1` as in MATLAB
- use square brackets `[]` to access elements instead of paranthesis `()`.

You can refer to this quick tutorial that gives you a summary of how to do MATLAB-y things in NumPy. Then feel free to just get cracking and experimenting!

Importing NumPy

To use NumPy:

```
import numpy as np
```

It is common to use `np` as a shorthand. We will assume this import and use `np` in all the examples that will follow.

If you get a `ModuleNotFoundError`, then you will have to first install NumPy.

```
$ pip3 install numpy
```

Next >>

Introduction to Machine Learning

COMP70050 Autumn Term 2021/2022

np.ndarray

The main 'pillar' of NumPy is the `np.ndarray` object (or `np.array` as an alias), which stands for *N-dimensional arrays*.

Unlike a Python `list`, elements in an `np.ndarray` **must** be of the same type.

`np.ndarray`s are also more efficient, and these have smaller memory consumption and better runtime compared to Python `list`s. So definitely use `np.ndarray` over `list` any time for large scale array/matrix operations.

Creating a NumPy array

For one dimensions, you can pass a Python `list` (or in fact any sequence type) to the constructor of `np.array()`.

```
x = np.array([1, 2, 3])
print(x)          ## [1, 2, 3]
print(type(x))    ## <class 'numpy.ndarray'>
```

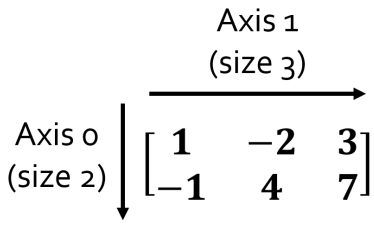
For higher dimensions, use a nested `list`. The following code gives you a 2D matrix.

```
y = np.array([[1, 2, 3], [-1, 4, 7]])
print(y)
## [[ 1  2  3]
##  [-1  4  7]]
```

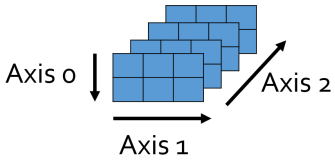
In NumPy, a dimension is called an **axis** (plural: **axes**).

`np.array([1, 2, 3])` has one axis (axis 0) with 3 elements.

`np.array([[1, 2, 3], [-1, 4, 7]])` has two axes. Axis 0 has 2 elements (rows), axis 1 has 3 elements (columns).



In higher dimensions, it is easier to think of the arrays as arrays inside arrays.



`np.ndarray` attributes

Here are some of the most important attributes of the `np.ndarray` object:

- `arr_name.ndim` – the number of axes (dimensions) of the array
- `arr_name.shape` – a `tuple` of integers representing the dimensions of the array (the number of elements in each axis)
- `arr_name.size` – the total number of elements of the array (i.e. the product of the elements of `arr_name.shape`)
- `arr_name.dtype` – a *data type* object describing the type of the elements in the array

```
x = np.array([[[0, 1, 2, 3],
               [4, 5, 6, 7]],
              [[0, 1, 2, 3],
               [4, 5, 6, 7]],
              [[0 ,1 ,2, 3],
               [4, 5, 6, 7]]])
print(x.ndim)     ## 3
print(x.shape)    ## (3, 2, 4)
print(x.size)     ## 24
print(x.dtype)    ## int64
```

The `dtype` of an `ndarray` is usually inferred from the type of elements in the sequence that you passed to the constructor.

Users can also explicitly specify the `dtype` in the constructor. This can either be standard Python types (e.g. `int` or `float`) or `np.dtype` (e.g. `np.int32`, `np.float64`)

```
x = np.array([0, 1, 2, 3])
print(x.dtype)    ## int64
x = np.array([0.0, 1.1, 2, 3])
print(x.dtype)    ## float64
x = np.array(["a", "b", "c", "d"])
print(x.dtype)    ## <U1 (unicode string)
x = np.array([0, 1, 2, 3], dtype=float)
print(x.dtype)    ## float64
x = np.array([0, 1, 2, 3], dtype=np.int32)
print(x.dtype)    ## int32
x = np.array([0, 1, 2, 3], dtype=np.uint32)
print(x.dtype)    ## uint32
x = np.array([0, 1, 2, 3], dtype=np.float64)
print(x.dtype)    ## float64
```

`int32` refers to a 32-bit integer, while `int64` refers to a 64-bit integer. Simply put, you can represent larger numbers with a larger number of bits.

`uint32` is an *unsigned* integer (non-negative integer).Useful for when you are not expecting a number to be negative (e.g. counts).

The official documentation provides a complete list of attributes for `np.ndarray`.

<< Previous Next >>

Introduction to Machine Learning

COMP70050 Autumn Term 2021/2022

Creating arrays

Numpy provides you with different convenient functions to create special types of arrays.

These are mainly useful if you need to initialise an array and fill it up later.

You can create an array filled with zeros, ones, or a number of your choice.

```
x = np.zeros(3)
## [0. 0. 0.]

x = np.ones(3)
## [1. 1. 1.]

x = np.full(3, 10)
## [10 10 10]

x = np.zeros((2, 3))  # 2 rows, 3 columns
## [[0. 0. 0.]
##  [0. 0. 0.]]

x = np.ones((2, 3))
## [[1. 1. 1.]
##  [1. 1. 1.]]

x = np.full((2, 3), 9.)
## [[9. 9. 9.]
##  [9. 9. 9.]]
```

Or filled with random numbers.

```
x = np.empty((2,4))
## [[6.95333209e-310 1.93101617e-312 6.95333207e-
310 6.91856305e-310]
##  [6.95333208e-310 2.12199579e-314 6.95333207e-
310 6.95333213e-310]]
```

You can also create an identity matrix with either `np.eye()` or `np.identity()`.

```
x = np.eye(3)
## [[1. 0. 0.]
##  [0. 1. 0.]
##  [0. 0. 1.]]
```

There is also a function `np.arange(start, stop, step)` that is equivalent to Python's `range()`.

```
x = np.arange(3, 11, 2)    ## [3 5 7 9]
```

If you use `np.arange()` with **non-integer** arguments, the results may not be consistent due to floating point precision issues. In such cases, it will be better to use `np.linspace()` to create an array with values that are spaced linearly in a specified interval.

```
# Generate an array with 5 elements, equally
spaced between 0 to 10
x = np.linspace(0, 10, num=5)  ## [ 0.   2.5  5.
7.5 10. ]
```

You can use `np.zeros_like(arr)`, `np.ones_like(arr)`, `np.full_like(arr)`, and `np.empty_like(arr)` to create arrays of zeros, ones and random numbers of the same shape AND type as `arr` (`arr` can be an `np.ndarray` or a Python `list`).

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

y = np.zeros_like(x)
print(y)
## [[0 0 0]
##  [0 0 0]]
print(y.shape)
## (2, 3)
```

I have only highlighted some of the array creation functions that will likely be more useful. There are many other array creation functions that I will not cover (like creating from a file or a string). I will leave it to you to explore on your own by reading the documentation.

Introduction to Machine Learning

COMP70050 Autumn Term 2021/2022

Arithmetic operations

In basic Python, if you have two lists, and you want to add up the elements across both lists, you will need to use a loop.

```
x = [1, 2, 3]
y = [4, 5, 6]
z = []
for (i, j) in zip(x, y):
    z.append(i + j)
print(z)    ## [5, 7, 9]
```

In Numpy, you can easily perform such element-wise operations more efficiently and more compactly. No loops required! This process is called **vectorisation** (or a vectorised operation).

```
x = np.array([1, 2, 3])
y = np.array([4, 5, 6])
z = x + y
print(z)    ## [5 7 9]
```

Here are some examples:

```
x = np.array([1, 2, 3, 4])
y = np.array([0, 1, 2, 3])
print(x + 2)      ## [3 4 5 6]
print(x - y)      ## [1 1 1 1]
print(x < 3)      ## [ True  True False False]
print(x + y > 5)  ## [False False False  True]

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

# multiplying an array with a scalar
print(a * 3)
## [[ 3  6]
##  [ 9 12]]
```

```
# elementwise multiplication
print(a * b)
## [[ 2  6]
##  [12 20]]

# elementwise division
c = np.array([1, 2])
d = np.array([3, 4])
print(c / d)
## [0.33333333 0.5       ]
```

Matrix multiplication can be performed using the `@` operator or `np.matmul()`. Make sure you note the difference between `a*b` and `a@b`!

```
print(a@b)
## [[10 13]
##  [22 29]]

print(np.matmul(a, b))
## [[10 13]
##  [22 29]]
```

To compute the inner product of two vectors (1D arrays), use `np.dot()` (dot product).

```
x = np.array([1, 2, 3])
y = np.array([2, 3, 4])
print(np.dot(x, y))   ## 20   == (1*2)+(2*3)+(3*4)
```

<< Previous

Next >>

Introduction to Machine Learning

COMP70050 Autumn Term 2021/2022

Indexing and slicing

Now, let's talk about how to access elements in a NumPy array.

For a 1D array, you access it just like a Python `list`, with a square bracket `[]`. Slicing works too.

```
x = np.array([1,2,3,4,5])
print(x[0])    ## 1
print(x[-1])   ## 5
print(x[2:4]) ## [3 4]
print(x[:3])   ## [1 2 3]
```

For multidimensional arrays, you access indices of different axes/dimensions by separating the indices with a comma. Slicing will also work.

```
y = np.array([[1, 2, 3, 5], [-1, 4, 7, 9]])
print(y[0,1])     ## 2  (row 0, col 1)
print(y[-1,-2])   ## 7  (last row, second-to-last
col)
print(y[1:3, :-1]) ## [[-1  4  7]] (figure this
out yourself!)
print(y[:,:])     ## What does this do?
print(y[0])       ## And this one?
```

While you can also use `y[0][1]` as in a Python `list`, this is much slower than `y[0,1]`. So use `y[0,1]` if you want your code to be more efficient!

Integer array indexing

You can also access arbitrary groups of items in an `np.ndarray`, using a list of indices.

```
print(y[[1, 0], [3, 2]])  ## [9 3]  (row 1, col
3; and row 0, col 2)
```

Boolean indexing

You can also access only elements in an array with a boolean `np.ndarray` as its indices.

```
x = np.array([1, 2, 3, 4, 5])
condition = np.array([True, False, False, True,
True])
print(x[condition])    # [1 4 5]  (only keep
elements that are True)
```

This is mainly useful for filtering your arrays. Possibly one of the most useful features that you may end up using a lot!

```
y = np.array([[1, 2, 3, 5], [-1, 4, 7, 9]])

print(y[y < 4])
## [1 2 3 -1] (Keep only elements that are <4)

print(y[(y < 4) & (y > 1)])
## [2 3] (Keep elements that are <4 and >1).
## (Note the paranthesis - you will get errors
otherwise because of operator precedence)

print(y[np.logical_and(y < 4, y > 1)])
## Same as above
```

More advanced indexing techniques are covered in the official documentation.

<< Previous Next >>

Introduction to Machine Learning

COMP70050 Autumn Term 2021/2022

Reshaping arrays

You can also easily reshape and manipulate arrays in NumPy.

To **flatten** an array into a 1D array, use `.flatten()`.

```
x = np.array([[0, 1, 2, 3],
              [4, 5, 6, 7],
              [-6, -5, -4, -3]
             ])
print(x.flatten())
## [ 0  1  2  3  4  5  6  7 -7 -6 -5 -4]
```

There is also a similar method called `.ravel()`. While `.flatten()` returns an array with a *copy* of the elements, `.ravel()` returns an array with a reference to the elements. So modifying an element in an array generated by `.ravel()` will also modify the element in the original array. Test this out yourself and compare the difference!

```
w = np.array([[1, 2, 3], [4, 5, 6]])
v = w.flatten()
v[0] = 10
print(v)
print(w)

v = w.ravel()
v[0] = 10
print(v)
print(w)
```

To **transpose** an array, use `.transpose()` or `.T`

```
print(x.transpose())
## [[ 0  4 -7]
##  [ 1  5 -6]
##  [ 2  6 -5]
##  [ 3  7 -4]]
```

```
print(x.T)
## Same as above
```

To **reshape** an array to a different shape, use `.reshape()`. Obviously, you have to keep the total number of elements the same as in the old array.

```
print(x)
## [[ 0  1  2  3]
##  [ 4  5  6  7]
##  [-7 -6 -5 -4]]

print(x.reshape((4,3)))
## [[ 0  1  2]
##  [ 3  4  5]
##  [ 6  7 -7]
##  [-6 -5 -4]]

print(x.reshape((6,2)))
## [[ 0  1]
##  [ 2  3]
##  [ 4  5]
##  [ 6  7]
##  [-7 -6]
##  [-5 -4]]

print(x.reshape((2,6)))
## [[ 0  1  2  3  4  5]
##  [ 6  7 -7 -6 -5 -4]]

# 2D to 3D? No problemo!
print(x.reshape((2,3,2)))
## [[[ 0  1]
##   [ 2  3]
##   [ 4  5]]
##
##  [[ 6  7]
##   [-7 -6]
##   [-5 -4]]]

# You can use -1 to let NumPy automatically infer
the size of the remaining axis
print(x.reshape((2, -1)))
## [[ 0  1  2  3  4  5]
##  [ 6  7 -7 -6 -5 -4]]
```

By default NumPy arrays are reshaped in **row-major order**, so left-

to-right, top-to-bottom for 2D. You can change this to **column-major order** (like in MATLAB) by assigning a keyword argument `order='F'` to `.reshape()` (`F` stands for Fortran-like order).

```
print(x)
## [[ 0  1  2  3]
##  [ 4  5  6  7]
##  [-7 -6 -5 -4]]

print(x.reshape((4,3)))
## [[ 0  1  2]
##  [ 3  4  5]
##  [ 6  7 -7]
##  [-6 -5 -4]]

print(x.reshape((4,3), order='F'))
## [[ 0  5 -5]
##  [ 4 -6  3]
##  [-7  2  7]
##  [ 1  6 -4]]
```

There is an equivalent method `.resize()`, but this modifies the array directly, rather than returning a new array.

```
print(x)
## [[ 0  1  2  3]
##  [ 4  5  6  7]
##  [-7 -6 -5 -4]]

x.resize((4,3))

print(x)
## [[ 0  1  2]
##  [ 3  4  5]
##  [ 6  7 -7]
##  [-6 -5 -4]]
```

Adding a new axis to a NumPy array

You may at some point need to convert 1D arrays to a higher dimensional array.

If you need to add a new axis/dimension to an existing NumPy array, use `np.newaxis`.

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

```
print(x.shape)     ## (6, )

y = x[np.newaxis, :]
print(y.shape)     ## (1, 6)
```

This is useful if you need to convert a NumPy array to a row vector or a column vector.

```
row_vector = x[np.newaxis, :]
print(row_vector.shape)    ## (1, 6)

col_vector = x[:, np.newaxis]
print(col_vector.shape)    ## (6, 1)
```

Alternatively, you can use `np.expand_dims()` to achieve the same thing.

```
row_vector = np.expand_dims(x, axis=0)
print(row_vector.shape)    ## (1, 6)

col_vector = np.expand_dims(x, axis=1)
print(col_vector.shape)    ## (6, 1)
```

Removing single-dimensional entries from a NumPy array

Perhaps at some point in your machine learning experiments, you may end up with an array of size `(3,1,2)`.

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

print(x)
## [[[1 2]]
##  [[3 4]]
##  [[5 6]]]
## Note the extra square brackets!

print(x.shape)
## (3, 1, 2)
```

Depending on what you are doing, the dimension with a single element may not be of use to you. You can convert it to an array of size `(3,2)` by `.squeeze()`-ing out that singleton dimension.

```
y = x.squeeze()
print(y)
```

```
## [[1 2]
##  [3 4]
##  [5 6]]
## Compare this to the one above

print(y.shape)
## (3, 2)
```

`.squeeze()` will remove all singleton dimensions. If you only want to remove only a specific dimension, use the optional keyword argument `axis`.

```
x = np.array([[[0], [0], [0]]])
print(x)
## [[[0]
##  [0]
##  [0]]]

print(x.shape)
## (1, 3, 1)

y = x.squeeze(axis=0)
print(y)
## [[0]
##  [0]
##  [0]]

print(y.shape)
## (3, 1)

y = x.squeeze(axis=2)
print(y)
## [[0 0 0]]

print(y.shape)
## (1, 3)
```

Note: Both `x.squeeze()` and `np.squeeze(x)` can be used. I use `x.squeeze()` because I prefer OOP (and it's shorter!)

<< Previous

Next >>

Introduction to Machine Learning

COMP70050 Autumn Term 2021/2022

Splitting arrays

You can split an array **evenly** into multiple sub-arrays with `np.split()`, giving the number of splits and also the axis as input arguments.

```
x = np.arange(1, 19).reshape(2, 9)
print(x)
## [[ 1  2  3  4  5  6  7  8  9]
##  [10 11 12 13 14 15 16 17 18]]

## split on axis 1 (columns) into 3 even-sized
sub-arrays
y = np.split(x, 3, axis=1)
print(y[0])
## [[ 1  2  3]
##  [10 11 12]]

print(y[1])
## [[ 4  5  6]
##  [13 14 15]]

print(y[2])
## [[ 7  8  9]
##  [16 17 18]]

## split on axis 0 (rows) into 3 even-sized sub-
arrays
y = np.split(x, 3, axis=0)
print(y[0])
## [[1 2 3 4 5 6 7 8 9]]

print(y[1])
## [[10 11 12 13 14 15 16 17 18]]

print(y[2])
## []
```

A similar function `np.array_split()` also allows you to split an array without needing to be strictly even. For example, you can divide 10 columns into 3 sub-arrays.

You can also specify *where* to split, by giving a list or tuple as the second argument (instead of an integer).

```
## split on axis 1 (columns) at columns 3 and 5.
y = np.split(x, [3, 5], axis=1)
print(y[0])
## [[ 1  2  3]
##  [10 11 12]]

print(y[1])
## [[ 4  5]
##  [13 14]]

print(y[2])
## [[ 6  7  8  9]
##  [15 16 17 18]]
```

There are also special functions that split at specific axes.

- `np.vsplit(arr, section)`: same as `np.split(arr, section, axis=0)` [Vertical split]
- `np.hsplit(arr, section)`: same as `np.split(arr, section, axis=1)` [Horizontal split]
- `np.dsplit(arr, section)`: same as `np.split(arr, section, axis=2)` [Depth split]

<< Previous Next >>

Introduction to Machine Learning

COMP70050 Autumn Term 2021/2022

Stacking arrays

You can also concatenate and stack multiple existing arrays. Explore the output in the examples below. I think they should be pretty self-explanatory.

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

print(np.concatenate((a, b), axis=0))
## [[1 2]
##  [3 4]
##  [5 6]]

print(np.concatenate((a, b.T), axis=1))
## [[1 2 5]
##  [3 4 6]]
```

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

print(np.stack((c, d), axis=0))
## [[1 2 3]
##  [4 5 6]]

print(np.stack((c, d), axis=1))
## [[1 4]
##  [2 5]
##  [3 6]]
```

Again, there are specialised versions of `np.stack()`:

- `np.vstack(tuple)`: same as `np.stack(tuple, axis=0)`
- `np.hstack(tuple)`: same as `np.stack(tuple, axis=1)`
- `np.dstack(tuple)`: same as `np.stack(tuple,`

```
    axis=2)
```

Repeating an array

You can also repeat an array easily with `np.repeat()`.

```
print(np.repeat(0, 2))
## [0 0]

x = np.array([[1, 2],[3, 4]])
print(x)
## [[1 2]
##  [3 4]]

print(np.repeat(x, 2))
## [1 1 2 2 3 3 4 4]

print(np.repeat(x, 2, axis=0))
## [[1 2]
##  [1 2]
##  [3 4]
##  [3 4]]

print(np.repeat(x, 2, axis=1))
## [[1 1 2 2]
##  [3 3 4 4]]

# You can also specify the number of repeats
individually
print(np.repeat(x, [1, 2], axis=0))
## [[1 2]
## [3 4]
## [3 4]]

print(np.repeat(x, [3, 2], axis=0))
## [[1 2]
##  [1 2]
##  [1 2]
##  [3 4]
##  [3 4]]
```

<< Previous

Next >>

Introduction to Machine Learning

COMP70050 Autumn Term 2021/2022

Mathematical functions

NumPy provides many mathematical functions that can be used in a vectorised manner.

Min and Max

You can use the `ndarray` methods `.max()` or `.min()` to compute the maximum/minimum values in an array.

`np.max(arr)` and `np.min(arr)` also works.

```
a = np.array([[4,7,3],[1,2,5]])

# Compute overall min/max
print(a.min())   ## 1
print(a.max())   ## 7

# Compute min/max across rows
print(a.min(axis=0))   ## [1 2 3]
print(a.max(axis=0))   ## [4 7 5]

# Compute min/max across columns
print(a.min(axis=1))   ## [3 1]
print(a.max(axis=1))   ## [7 5]
```

To get the **indices** for the min/max values, use `.argmax()` and `.argmin()`.

```
a = np.array([[4,7,3],[1,2,5]])

# Get (flattened) indices of overall min/max
print(a.argmin())   ## 3 (index of "1")
print(a.argmax())   ## 1 (index of "7")

# Get row indices of min/max across rows
print(a.argmin(axis=0))   ## [1 1 0]
print(a.argmax(axis=0))   ## [0 0 1]
```

```
# Get column indices of min/max across columns
print(a.argmin(axis=1))   ## [2 0]
print(a.argmax(axis=1))   ## [1 2]
```

Statistical methods for arrays

- `arr.mean(axis=None)`
    - Compute the mean of `arr`.
    - If `axis` is not provided, compute the mean of the flattened array
    - If `axis` is provided, compute the means across the axis.
- `arr.std(axis=None)`
    - Compute the standard deviation of `arr`
    - If `axis` is not provided, compute the standard deviation of the flattened array
    - If `axis` is provided, compute the standard deviations across the axis.
- `arr.var(axis=None)`
    - Compute the variance of `arr`
    - If `axis` is not provided, compute the variance of the flattened array
    - If `axis` is provided, compute the variances across the axis.

Sum and Cumulative Sum

Like `.min()` and `.max()`, `.sum()` can be used to sum up a flattened array, or compute the sum across a specified axis. Explore the output for below yourself and try to make sense of it! Draw out the 3D array and the axis direction on a piece of paper if you are confused.

```
a = np.arange(24).reshape((2,3,4))
print(a)
## [[[ 0  1  2  3]
##   [ 4  5  6  7]
##   [ 8  9 10 11]]
##
##  [[12 13 14 15]
```

```
##    [16 17 18 19]
##    [20 21 22 23]]]

print(a.sum())
## 276

print(a.sum(axis=0))
## [[12 14 16 18]
##  [20 22 24 26]
##  [28 30 32 34]]

print(a.sum(axis=1))
## [[12 15 18 21]
##  [48 51 54 57]]

print(a.sum(axis=2))
## [[ 6 22 38]
##  [54 70 86]]
```

`.cumsum()` computes the **cumulative** sum of an array.

```
a = np.array([1, 2, 3, 4, 5])
print(a.cumsum())   ### [1 3 6 10 15]
```

Like `.sum()`, you can compute the cumulative sum across a specific axis.

```
a = np.array([[1,2,3], [4,5,6]])
print(a)
## [[1 2 3]
##  [4 5 6]]

print(a.cumsum())
## [ 1  3  6 10 15 21]

print(a.cumsum(axis=0))
## [[1 2 3]
##  [5 7 9]]

print(a.cumsum(axis=1))
## [[ 1  3  6]
##  [ 4  9 15]]
```

Product and Cumulative Product

Just like `.sum()` and `.cumsum()`, `.prod()` amd `.cumsum()`

computes the product and cumulative product of an array respectively.

I will not provide examples for these as they are essentially similar to the above (just multiply instead of adding).

Others

These functions can be applied to arrays in an elementwise fashion.

- `np.floor(arr)`: Floor function
- `np.round(arr)`: Round function
- `np.exp(arr)`: Exponential function
- `np.sqrt(arr)`: Square root
- `np.sin(arr)`: Sin function
- `np.cos(arr)`: Cos function

<< Previous Next >>

Introduction to Machine Learning

COMP70050 Autumn Term 2021/2022

Indexing functions

Truth value functions

`np.all(arr)` checks whether **all** elements in `arr` are `True`.

`np.any(arr)` checks whether at least one element in `arr` is `True`.

Note: NaN, positive and negative infinity are considered `True` because they are non-zero.

```
print(np.all(np.array([True, True, True])))
## True
print(np.all(np.array([True, True, False])))
## False
print(np.any(np.array([True, False, False])))
## True
print(np.any(np.array([False, False, False])))
## False
```

For multidimensional arrays, you can also check across a specific axis.

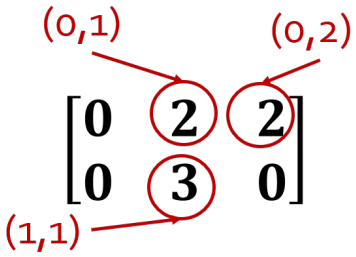
```
x = np.array([[True, True, False, False], [True,
False, True, False]])
print(np.all(x, axis=0))   ## [ True False False
False]
print(np.any(x, axis=0))   ## [ True  True  True
False]
print(np.all(x, axis=1))   ## [False False]
print(np.any(x, axis=1))   ## [True True]
```

Searching functions

`np.nonzero()` returns the indices of all elements that are not zero

(or not `False`).

```
x = np.array([[0,2,2], [0,3,0]])
print(np.nonzero(x))     ## (array([0, 0, 1]),
array([1, 2, 1]))
print(x[np.nonzero(x)]) ## [2 2 3]
```



Unique

To find a set of unique elements in an array, use (you guessed it!) `np.unique()`.

```
x = np.array([12, 15, 13, 15, 16, 17, 13, 13, 18,
13, 19, 18, 11, 16, 15])

print(np.unique(x))
## [11 12 13 15 16 17 18 19]
```

You can also return the index of the first occurrence of each of the unique elements in the array. For example, the number `11` is first encountered at index `12`.

```
(unique_x, unique_indices) = np.unique(x,
return_index=True)

print(unique_x)
## [11 12 13 15 16 17 18 19]

print(unique_indices)
## [12  0  2  1  4  5  8 10]
```

You can also count the number of times each unique element occurs in the array.

```
(unique_x, unique_counts) = np.unique(x,
return_counts=True)

print(unique_x)
## [11 12 13 15 16 17 18 19]
```

```
print(unique_counts)
## [1 1 4 3 2 1 2 1]
```

Sorting functions

You can sort an array using `np.sort()`.

You can also sort across rows or columns by specifying the axis.

```
a = np.array([3, 1, 2])
print(np.sort(a))  ## [1 2 3]

b = np.array([[6, 4], [1, 0], [2, 7]])

# Sort rows
print(np.sort(b, axis=0))
## [[1 0]
##  [2 4]
##  [6 7]]

# Sort columns
print(np.sort(b, axis=1))
## [[4 6]
##  [0 1]
##  [2 7]]
```

To get the **indices** after a sort, use `np.argsort()`.

```
print(np.argsort(a))  ## [1 2 0]

# Get indices of rows after sorting
print(np.argsort(b, axis=0))
## [[1 1]
##  [2 0]
##  [0 2]]

# Get indices of columns after sortig
print(np.argsort(b, axis=1))
## [[1 0]
##  [1 0]
##  [0 1]]
```

Diagonals

If you need to obtain the diagonal of an array, you might find the `np.diagonal(arr)` function or `arr.diagonal()` method useful.

```
x = np.array([[10, 4, 2], [6, 9, 3], [1, 5, 8]])
print(x.diagonal())           ## [10  9  8]
print(x.diagonal(offset=1))   ## [4 3]
print(x.diagonal(offset=-1)) ## [6 5]
```

<< Previous Next >>

Introduction to Machine Learning

COMP70050 Autumn Term 2021/2022

Miscellaneous

A few extra NumPy tips that do not fit elsewhere!

Loading and saving arrays

You can save and load numpy arrays onto/from the disk.

The first option is to save them as binary files. There are actually Python `pickle` files.

- `*.npy` files are usually for one array.
- `*.npz` files are for multiple arrays.

```
a = np.array([1, 2, 3, 4, 5, 6])
np.save("myfile", a)
a_fromdisk = np.load("myfile.npy")
print(a_fromdisk)

b = np.array([[1,2],[3,4]])
np.savez("multiple", a=a, b=b)
c = np.load("multiple.npz")
print(c["a"])
print(c["b"])
```

The second option is to save them as text files.

```
np.savetxt("myfile.csv", a)
a_fromtext = np.loadtxt("myfile.csv")
print(a_fromtext)
```

Generating random numbers

To generate random numbers in NumPy, you should first import the Default Random Number Generator provided by NumPy.

```
from numpy.random import default_rng
```

You should then initialise the random number generator with a **seed number** (just choose any number).

Tip: Random numbers are not really random in computers. They are **pseudo-random** because you can reproduce the same 'random' sequence with a fixed seed number. This is important in your scientific experiments so that you can **reproduce** your experimental results!

```
seed = 7777
rg = default_rng(seed)
```

You can then use the random generator instance to generate random numbers or arrays.

```
x = rg.random((5,3))
print(x)
## [[0.94192575 0.59573376 0.58880255]
##  [0.71400906 0.24578678 0.63006854]
##  [0.26233112 0.39487634 0.65615324]
##  [0.85818932 0.20564809 0.9837931 ]
##  [0.18541645 0.67159972 0.99064663]]

# Generate random integers from 1 (inclusive) to
10 (exclusive)
y = rg.integers(1, 10, size=(5,3))
print(y)
## [[9 8 7]
##  [7 2 8]
##  [9 9 4]
##  [1 9 6]
##  [6 3 3]]
```

Printing more values

NumPy only prints values at the start and end of the array if the array is too large.

```
print(np.arange(2000)) # [0 1 2 ... 1997 1998
1999]
```

To print more values, use `set_printoptions()`

```
np.set_printoptions(threshold=10000) # defualt
threshold is 1000
```

```
print(np.arange(2000))   # should print all
numbers now
```

<< Previous Next >>

Introduction to Machine Learning

COMP70050 Autumn Term 2021/2022

Practical examples

I think that is more than enough NumPy for you to digest for now!

Let's try to make things more interesting with some practical examples of how you might end up using all these fancy NumPy features in Machine Learning.

Computing accuracy for evaluating classification

Let us look at one example. You have seen/will see me discussing a bit on Machine Learning evaluation in your Introduction to Machine Learning lecture next week.

For a classification task, we might have to compute an evaluation metric called \(accuracy\).

\[accuracy = \frac{| correct |}{| instances |}\]

Let's say we have built a classifier for three classes: `apple`, `orange`, `pear`. And our classifier has predicted the output for a set of test instances. Say we have compared the output of each test instance with the correct label, and have put them in a **confusion matrix** below.

	apple	**orange**	**pear**
apple	18	3	9
orange	1	30	4
pear	7	5	23

Let's assume the rows represent the correct label and the columns represent the predicted classes.

So according to the confusion matrix, 18 instances of apples are correctly predicted as apples by your classifier. 3 instances of apples

are misclassified as oranges, and 9 instances of apples are misclassified as pears.

Similarly, 1 orange was misclassified as apple, 4 misclassfied as pears, and the remaining 30 are correctly classified.

The confusion matrix can be represented as a NumPy array as below:

```
x = np.array([[18, 3, 9], [1, 30, 4], [7, 5, 23]])
```

We can compute the number of instances for each class from the matrix.

```
class_distribution = np.sum(x, axis=1)  ## [30 35 35]
```

We can also compute the total number of test instances.

```
total_instances = np.sum(x)  ## 100
```

We can also compute the proportion of instances per class (in percentage)

```
class_percentage = class_distribution / total_instances  ## [0.3  0.35 0.35]

# or...
class_percentage = np.sum(x, axis=1) / np.sum(x)

# or with a more OOP notation...
class_percentage = x.sum(axis=1) / x.sum()
```

Now, to compute the accuracy, we need to know how many instances are correctly predicted overall. We can take advantage of the fact that the correct predictions are in the **diagonal** of the confusion matrix, i.e. 18, 30 and 23. So we extract the diagonal from the matrix.

```
correct_perclass = np.diagonal(x) ## [18 30 23]
```

We can then sum up the number of correct predictions per class to get the total number of correct predictions

```
total_correct = np.sum(correct_perclass)  ## 71
```

The two steps above can be combined into a single line if you do so

desire.

```
total_correct = np.sum(np.diagonal(x))   ## 71

## or if you prefer an object-oriented aproach...
total_correct = x.diagonal().sum()
```

Now, you can compute the accuracy!

```
accuracy = total_correct / total_instances
```

Of course, this could have been all done in a single line if you like.

```
accuracy = np.sum(np.diagonal(x)) / np.sum(x)
```

Computing Mean Squared Error for evaluating regression

Another good example would be to compute the mean squared error for regression, as briefly shown in the Introduction to Machine Learning lectures. Fortunately, the official NumPy tutorial already has this example covered, so there is no point for me to reproduce it just for the sake of it (lazy!). So please just look at the example on the official tutorial to see how you would implement this.

Introduction to Machine Learning

COMP70050 Autumn Term 2021/2022

NumPy exercises

Now that you have spent so much time reading, it is time to get your hands dirty with NumPy!

Many thanks to the following people for contributing to these exercises: Joe Stacey

Task 1

Multiply each element of `np.arange(10)` by `2`

Task 2

Try doing the following using NumPy:

- Create an identity matrix
- Find the square root of each element in an array
- Square each element in an array
- Take logs of each element in an array
- Create an array of zeros
- Create an array of ones
- Give the value of \(\pi\)
- Transpose an array
- Round down a float
- Find the median of floats in a NumPy array
- Add a new element to a NumPy array

Task 3

Create a \(1 \times 40\) 1D array, and then convert this into a \(8 \times 5\) 2D NumPy array.

Task 4

Create a \(1 \times 2000\) NumPy array with the step-wise sequence from 1 to 2000, and then convert this into a \(40 \times 50\) 2D NumPy array.

Then

- Index this array correctly to return the value 435
- Index this array correctly to return the value 951

Task 5

What happens when we use `.ravel()`?

- For example: How is `np.arange(20).reshape(20, 1).ravel()` different to `np.arange(20).reshape(20, 1)`?
- Try using .shape to help investigate

Task 6

We can use `.all()` to check whether two NumPy arrays are identical. Which of the following do you expect to return `True`? Check your answers using Python.

- `(np.arange(20).reshape(20,1) == np.arange(20)).all()`
- `(np.arange(20).reshape(20,1).ravel() == np.arange(20)).all()`
- `(np.arange(20).reshape(20,1,1).ravel() == np.arange(20)).all()`
- `(np.arange(20).reshape(20,) == np.arange(20)).all()`
- `(np.arange(20).reshape(20,) == np.arange(20).ravel()).all()`

Task 7

Normalise the vector reprsented by `np.array([3, 5, 1, 2, 4])` to find the unit vector in the same direction.