CS4618: Artificial Intelligence I

Vectors and Matrices

Derek Bridge
School of Computer Science and Information Technology
University College Cork

Initialization

In [1]:

```
%load_ext autoreload
%autoreload 2
%matplotlib inline
```

In [2]:

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
```

In [3]:

```
import numpy.linalg as npla

from math import sqrt
```

Doing Things with Data

- All of these are about doing things with data:
  - data science, data analytics, machine learning, statistics, statistical machine learning, statistical inference, data mining, knowledge discovery, pattern recognition, …
- These fields have been given impetus by:
  - availability of lots of data (sometimes 'big data'), partly due to sensors, the Internet, …
  - availability of hardware for high volume storage and processing, including GPUs, cloud computing, …
- We use techniques discovered by these fields for tasks in AI such as prediction (regression, classification), clustering, speech recognition, machine translation, …
- But, first, some background maths!

Matrices

- A **matrix** is a rectangular array, in our case of real numbers
- In general, we use bold capital letters, e.g. $\boldsymbol{A}$, for matrices, e.g.
$$\boldsymbol{A} = \begin{bmatrix} 2 & 4 & 0 \\ 1 & 3 & 2 \end{bmatrix}$$
- **Dimension**: A matrix with m rows and n columns is an $m \times n$ **matrix**
  - What are m and n for $\boldsymbol{A}$?
- We refer to an **element** of a matrix either using subscripts or indexes
  - $\boldsymbol{A}_{i,j}$ or $\boldsymbol{A}[i, j]$ is the element in the ith row and jth column
  - We will index from 1
    - However, we will sometimes use position 0 for 'technical' purposes
    - And we must be aware that Python numpy arrays and matrices are 0-indexed
  - So what are $\boldsymbol{A}_{2,1}$, $\boldsymbol{A}_{1,2}$, $\boldsymbol{A}_{0,0}$ and $\boldsymbol{A}_{3,2}$?

Vectors

- A **vector** is a matrix that has only one column, i.e. a $m \times 1$ matrix
- A vector with m rows is called a m**-dimensional** vector
- In general, we use bold lowercase letters for vectors, e.g.
$$\boldsymbol{x} = \begin{bmatrix} 2 \\ 4 \\ 3 \end{bmatrix}$$
- Sometimes this is called a **column vector**
- Then, by contrast, a **row vector** is a matrix that has only one row, i.e. a $1 \times n$ matrix, e.g.
$$[2, 4, 3]$$
- Unless stated otherwise, a vector should be assumed to be a column vector.
- We can refer to an element using a single subscript, again most of the time indexed from 1
  - So what is $\boldsymbol{x}_1$?

Vectors and Matrices in Python

- Of the many ways of representing vectors and matrices in Python, we will use two:
  - pandas library:
    - for vectors: `Series`, a kind of one-dimensional array
    - for matrices: `DataFrames`, which are tabular data structures of rows and (named) columns
  - numpy library
    - numpy arrays, which can be one-dimensional, two-dimensional, or have more dimensions

  The scikit-learn library expects its data to arrive as numpy arrays

Using numpy arrays

In [4]:

```
# Vectors
# We will use a numpy 1d array, which we can create from a list
# But, done this way, there is no way for us to distinguish between column- and
 row-vectors
x = np.array([2, 4, 3])

# Matrices
# We will use a numpy 2d array, which we can create from a list of lists
A = np.array([[2, 4, 0], [1, 3, 2]])
```

We can see their dimensions:

In [5]:

```
x.shape
```

Out[5]:

```
(3,)
```

In [6]:

```
A.shape
```

Out[6]:

```
(2, 3)
```

You can think of (3,) as saying it's not a nested list: it has 3 numbers in it.

We can make it into a nested list using the reshape method, and then its shape is (3,1):

In [7]:

```
X = x.reshape((3,1))
X
```

Out[7]:

```
array([[2],
       [4],
       [3]])
```

In [8]:

```
X.shape
```

Out[8]:

```
(3, 1)
```

Scalar Addition and Scalar Multiplication

- In this context, 'scalar' simply means a number
- Scalar addition and multiplication both work **elementwise**, i.e.:
  - In scalar addition, we add the number to each element in the matrix
  - In scalar multiplication, we multiply each element in the matrix by the number
- E.g.

$$\boldsymbol{A} = \begin{bmatrix} 2 & 4 & 0 \\ 1 & 3 & 2 \end{bmatrix} \qquad 2 + \boldsymbol{A} = \begin{bmatrix} 4 & 6 & 2 \\ 3 & 5 & 4 \end{bmatrix} \qquad 2\boldsymbol{A} = \begin{bmatrix} 4 & 8 & 0 \\ 2 & 6 & 4 \end{bmatrix}$$

Scalar Addition and Scalar Mutliplication in numpy

- numpy arrays enable operations like these using the normal addition, subtraction, multiplication and division operators and without writing for loops

In [9]:

```
A = np.array([[2, 4, 0], [1, 3, 2]])
```

In [10]:

```
2 + A
```

Out[10]:

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

In [11]:

```
2 * A
```

Out[11]:

```
array([[4, 8, 0],
       [2, 6, 4]])
```

- Other Python operators also work

In [12]:

```
A**2
```

Out[12]:

```
array([[ 4, 16,  0],
       [ 1,  9,  4]])
```

Matrix Addition and Hadamard Product

- Matrx addition and Hadamard product require two matrices that have *the same dimensions*
- They are also defined elementwise: by adding or multiplying *corresponding* elements
- E.g.
  $\boldsymbol{A} = \begin{bmatrix} 2 & 4 & 0 \\ 1 & 3 & 2 \end{bmatrix} \quad \boldsymbol{B} = \begin{bmatrix} 1 & 0 & 5 \\ 2 & 3 & 2 \end{bmatrix} \quad \boldsymbol{A} + \boldsymbol{B} = \begin{bmatrix} 3 & 4 & 5 \\ 3 & 6 & 4 \end{bmatrix} \quad \boldsymbol{A} * \boldsymbol{B} = \begin{bmatrix} 2 & 0 & 0 \\ 2 & 9 & 4 \end{bmatrix}$
- In maths, Hadamard product is more often written with a dot ($\cdot$ or $\circ$), but we will use $*$

Matrix Addition and Hadamard Product in numpy

- We don't need to write any loops, just use + and *

In [13]:

```
A = np.array([[2, 4, 0], [1, 3, 2]])
B = np.array([[1, 0, 5], [2, 3, 2]])
```

In [14]:

```
A + B
```

Out[14]:

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

In [15]:

```
A * B
```

Out[15]:

```
array([[2, 0, 0],
       [2, 9, 4]])
```

Broadcasting in numpy

- In maths, matrix addition and Hadamard product require the matrices to have the same dimensions
- But, in numpy, things are less rigid, e.g.:

In [16]:

```
x = np.array([2, 4, 3])
A = np.array([[2, 4, 0], [1, 3, 2]])

A + x
```

Out[16]:

```
array([[4, 8, 3],
       [3, 7, 5]])
```

- Conceptually, the smaller array is copied enough times to make its dimensions compatible with the larger array
  - But it isn't *literally* copied and, in many cases, is substantially faster that writing your own loops
  - This is called **broadcasting**

numpy's Rules for Broadcasting

- The rules for broadcasting are: the dimensions of the two arrays are compared, starting from the trailing dimensions, and are compatible when
  - they are equal, or
  - one of them is 1
- In the example above $\boldsymbol{A}$ was 2×3 and $\boldsymbol{x}$ was 3
- Hence, which of these will work, and which will give errors?

In [ ]:

```
A = np.ones((5, 4))
x = np.ones(4)

A + x
```

In [ ]:

```
A = np.ones((5,4))
B = np.ones((5,1))

A + B
```

In [ ]:

```
A = np.ones((5,4))
x = np.ones(5)

A + x
```

In [ ]:

```
A = np.ones((5, 4))
x = np.ones(5)
B = x.reshape((5,1))

A + B
```

In [ ]:

```
A = np.ones((2, 1))
B = np.ones((4, 3))

A * B
```

Matrix Multiplication

- We can compute $\boldsymbol{AB}$, the result of multiplying matrices $\boldsymbol{A}$ and $\boldsymbol{B}$, provided the number of columns of $\boldsymbol{A}$ equals the number of rows of $\boldsymbol{B}$
  - If $\boldsymbol{A}$ is a $m \times p$ matrix and $\boldsymbol{B}$ is a $p \times n$ matrix, then we can compute $C = \boldsymbol{AB}$
  - $\boldsymbol{C}$ will be a $m \times n$ matrix
- $\boldsymbol{C}_{i,j}$ is obtained by multiplying elements of the ith row of $\boldsymbol{A}$ by corresponding elements of the jth column of $\boldsymbol{B}$ and summing:

$$\boldsymbol{C}_{i,j} = \sum_{k=1}^{p} \boldsymbol{A}_{i,k}\boldsymbol{B}_{k,j}$$

- E.g.

$$\boldsymbol{A} = \begin{bmatrix} 2 & 4 & 0 \\ 1 & 3 & 2 \end{bmatrix} \qquad \boldsymbol{B} = \begin{bmatrix} 3 & 1 & 2 \\ 2 & 3 & 1 \\ 1 & 3 & 3 \end{bmatrix} \qquad \boldsymbol{AB} = \begin{bmatrix} 14 & 14 & 8 \\ 11 & 16 & 11 \end{bmatrix}$$

- Since vectors are just one-column vectors, matrix multiplication can apply — provided the dimensions are OK, e.g.

$$\boldsymbol{A} = \begin{bmatrix} 2 & 4 & 0 \\ 1 & 3 & 2 \end{bmatrix} \qquad \boldsymbol{x} = \begin{bmatrix} 2 \\ 3 \\ 1 \end{bmatrix} \qquad \boldsymbol{y} = \begin{bmatrix} 2 \\ 3 \end{bmatrix} \qquad \boldsymbol{Ax} = \begin{bmatrix} 16 \\ 13 \end{bmatrix} \qquad \boldsymbol{Ay} \text{ is undefined}$$

Matrix Multiplication in numpy

- numpy offers `dot` as a function or method for matrix multiplication:

In [20]:

```
A = np.array([[2, 4, 0], [1, 3, 2]])
B = np.array([[3, 1, 2], [2, 3, 1], [1, 3, 3]])

# Multiplication as a function
# np.dot(A, B)

# Multiplication as a method
A.dot(B)
```

Out[20]:

```
array([[14, 14,  8],
       [11, 16, 11]])
```

- Remember, matrix multplication in numpy is done with `dot`, not *
- Brodcasting does not apply to matrix multiplication, since it's not an elementwise operation

Transpose

- The **transpose** of $m \times n$ matrix $\boldsymbol{A}$, written $\boldsymbol{A}^T$, is the $n \times m$ matrix in which the first row of $\boldsymbol{A}$ becomes the first column of $\boldsymbol{A}^T$, the second row of $\boldsymbol{A}$ becomes the second column of $\boldsymbol{B}$, and so on:
  - $\boldsymbol{A}^T_{i,j} = \boldsymbol{A}_{j,i}$
- E.g.

$$\boldsymbol{A} = \begin{bmatrix} 2 & 4 & 0 \\ 1 & 3 & 2 \end{bmatrix} \qquad \boldsymbol{A}^T = \begin{bmatrix} 2 & 1 \\ 4 & 3 \\ 0 & 2 \end{bmatrix}$$

- As a special case, if $\boldsymbol{x}$ is a m-dimensional column vector ($m \times 1$), then $\boldsymbol{x}^T$ is a m-dimensional row vector ($1 \times m$), e.g.

$$\boldsymbol{x} = \begin{bmatrix} 2 \\ 4 \\ 3 \end{bmatrix} \qquad \boldsymbol{x}^T = [2, 4, 3]$$

Transpose in numpy

- numpy arrays offer easy ways to compute their transpose: either the `transpose` method or the `T` attribute:

In [21]:

```
A = np.array([[2, 4, 0], [1, 3, 2]])

# Transpose as a method
# A.transpose()

# Tranpose as an attribute
A.T
```

Out[21]:

```
array([[2, 1],
       [4, 3],
       [0, 2]])
```

Identity Matrices

- The $n \times n$ **identity matrix**, $\boldsymbol{I}_n$, contains zeros except for entries on the main diagonal (from top left to bottom right):
  - $\boldsymbol{I}_n[i, i] = 1$ for $i = 1, \ldots, n$ and $\boldsymbol{I}_n[i, j] = 0$ for $i \neq j$
- E.g.:

$$\boldsymbol{I}_3 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

- If $\boldsymbol{A}$ is an $m \times n$ matrix then, $\boldsymbol{A}\boldsymbol{I}_n = \boldsymbol{I}_m\boldsymbol{A} = \boldsymbol{A}$

Identity Matrices in numpy

- Create identity matrices using the `identity` function:

In [22]:

```
np.identity(3)
```

Out[22]:

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

Inverses

- If $\boldsymbol{A}$ is a $n \times n$ matrix, then its **inverse**, $\boldsymbol{A}^{-1}$ (*if it has one*) is also a $n \times n$ matrix such that $\boldsymbol{A}\boldsymbol{A}^{-1} = \boldsymbol{I}_n$.
- E.g.
$$\boldsymbol{A} = \begin{bmatrix} 1 & 0 & 2 \\ 2 & -1 & 3 \\ 4 & 1 & 8 \end{bmatrix} \qquad \boldsymbol{A}^{-1} = \begin{bmatrix} -11 & 2 & 2 \\ -4 & 0 & 1 \\ 6 & -1 & -1 \end{bmatrix}$$
- Some $n \times n$ matrices do not have inverses, e.g.
$$\begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$
In these cases, provided the matrix is square, you can compute a **pseudo-inverse**, which you can use for *some* of the same purposes instead

Inverses in numpy

- numpy.linalg offers function `inv` for computing inverses, but also function `pinv` for computing the Moore-Penrose pseudo-inverse:

In [23]:

```
A = np.array([[1, 0, 2], [2, -1, 3], [4, 1, 8]])

npla.inv(A)
```

Out[23]:

```
array([[-11.,   2.,   2.],
       [ -4.,  -0.,   1.],
       [  6.,  -1.,  -1.]])
```

In [24]:

```
npla.pinv(A)
```

Out[24]:

```
array([[ -1.10000000e+01,   2.00000000e+00,   2.00000000e+00],
       [ -4.00000000e+00,   1.42108547e-14,   1.00000000e+00],
       [  6.00000000e+00,  -1.00000000e+00,  -1.00000000e+00]])
```

In [25]:

```
B = np.ones((3,3))

npla.inv(B) # raises an exception
```

```
---------------------------------------------------------------------------
LinAlgError                               Traceback (most recent call last)
<ipython-input-25-68bf09415942> in <module>()
      1 B = np.ones((3,3))
      2 
----> 3 npla.inv(B) # raises an exception

/home/dgb/anaconda3/lib/python3.5/site-packages/numpy/linalg/linalg.py in inv(a)
    524     signature = 'D->D' if isComplexType(t) else 'd->d'
    525     extobj = get_linalg_error_extobj(_raise_linalgerror_singular)
--> 526     ainv = _umath_linalg.inv(a, signature=signature, extobj=extobj)
    527     return wrap(ainv.astype(result_t, copy=False))
    528 

/home/dgb/anaconda3/lib/python3.5/site-packages/numpy/linalg/linalg.py in _raise_linalgerror_singular(err, flag)
     88 
     89 def _raise_linalgerror_singular(err, flag):
---> 90     raise LinAlgError("Singular matrix")
     91 
     92 def _raise_linalgerror_nonposdef(err, flag):

LinAlgError: Singular matrix
```

In [26]:

```
npla.pinv(B)
```

Out[26]:

```
array([[ 0.11111111,  0.11111111,  0.11111111],
       [ 0.11111111,  0.11111111,  0.11111111],
       [ 0.11111111,  0.11111111,  0.11111111]])
```

Some numpy Methods

- numpy offers methods for calculations that, in other languages, would require you to write loops
- E.g. `sum`, `mean`, `min`, `max`, `argmin`, `argmax`, …

In [27]:

```
x = np.array([2, 4, 3])
A = np.array([[2, 4, 0], [1, 3, 2]])
```