Matricies and Linear Algebra using SymPy and Numpy

Dr Rob Collins

Version 8, 23rd August 2023

Introduction

This Machine Learning course is intended to be generally non-mathematical. Nevertheless, no advanced study of Machine Learning can be realistically considered complete without a good grounding in Probability and Linear Algebra.

For students who are less familiar with these subjects I recommend the following as useful references:

Deisenroth, M.P., Faisal, A.A. and Ong, C.S (2021) "Mathematics for Machine Learning", Cambridge University Press

Strang, G. (2005) "Linear Algebra and its Applications", TBS

Walpole, R.E., Myers, R.H., Myers, S.L. and Ye, K. (2012) "Probability and Statistics for Engineers and Scientists", Prentice Hall

For many years Prof. Gilbert Strang presented a 'famous' course in Linear Algebra at MIT. The course is available free as part of MITs 'Open Courseware' programme. If you wish to become proficient in Machine Learning, that course is a strong recommendation: https://ocw.mit.edu/courses/18-06-linear-algebra-spring-2010/ . My own experience was that both Strang's book and the lecture course videos are easier to follow if you have access to a computer-tool to experiment with some of the Algebra. The SymPy tool used in this section provides exactly such a tool and may be a significant aid to study.

The following tutorial provides what I regard as an absolute minimum for required for a first course in Machine Learning. The material in this tutorial will be a requirement in order to read and understand most text-books, papers and even websites in the field of Machine Learning.

Instructions for Students

This workbook is different to some of the others in this series. As a student you are required to 'experiment' (modify and execute) many of the code blocks. You should do this until you have a through understanding of what they are doing and how they work.

Student activities are indicated in the workbook below with a '**Student Task**' indicator.

In working through the following notebook, please do the following:

1. Create an empty Jupyter notebook on your own machine
2. Enter all of the **Python code** from this notebook into **code blocks** in your notebook
3. Execute **and experiment with** each of the code blocks to develop and check your understanding
4. You **do not need to** replicate all of the explanatory / tutorial text in text (markdown) blocks
5. You **may** add your own comments and description into text (markdown) blocks if it helps you remember what the commands do
6. You **may** add further code blocks to experiment with the commands and try out other things
7. Enter and run as many of the code blocks as you can within the time available during class
8. **After class**, enter and run any remaining code blocks that you have not been able to complete in class

The numbers shown in the 'In [n]' text on your Jupyter notebook are likely to be different to the ones shown here because they are updated each time a block is executed.

Two Libraries for Linear Algebra - Sympy and Numpy

This tutorial introduces two libraries for linear algebra

- Sympy
- Numpy

The first is a *symbolic* mathematics library - and this make it easier to read and understand linear algebra as it might appear in a mathematics text-book (e.g. Strang).

The second is a very fast and efficient library used for numerical computation. It implements many more linear algebra operations than Sympy and is much more widely used in Machine Learning.

So Sympy is the best tool to help you learn linear algebra wheras you are much more likely to use Numpy for real Machine Learning projects.

1 Algebraic Equations represented in Matricies

Linear Algebra is concerned with the use of equations such as:

$1x + 2y = 3$

$4x + 5y = 6$

In the context of Machine Learning you will often see such equations represented in the form of Matricies:

$\begin{bmatrix}1 & 2 \\ 4 & 5 \end{bmatrix}\begin{bmatrix}x \\ y\end{bmatrix}=\begin{bmatrix}3\\ 6 \end{bmatrix}$

A this point you should check your understanding of how the above equations are represented in matrix form

2 Matricies using the Sympy Library

2.1 Defining Matricies in Sympy

For example, we can represent the following two matricies:

x = $\begin{bmatrix}1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$

.. and

y = $\begin{bmatrix}7 & 8 \\ 9 & 3 \\ 2 & 1 \end{bmatrix}$

Let's start by using Sympy to define some matricies which we can use in later examples

In [1]:
```
from sympy import *

x = Matrix([[1,2,3], [4,5,6]])
x
```
Out[1]: $\displaystyle \left[\begin{matrix}1 & 2 & 3\\4 & 5 & 6\end{matrix}\right]$

In [2]:
```
y = Matrix([[7,8],[9,3],[2,1]])
y
```
Out[2]: $\displaystyle \left[\begin{matrix}7 & 8\\9 & 3\\2 & 1\end{matrix}\right]$

In [3]:
```
p = Matrix([[4,7,1],[4,3,2]])
p
```
Out[3]: $\displaystyle \left[\begin{matrix}4 & 7 & 1\\4 & 3 & 2\end{matrix}\right]$

2.2 Matrix Addition and Subtraction

Two matricies can be added only if they are the same shape. Addition of matricies, simply means adding each element in one array to the corresponding element in the second array:

In [4]:
```
x + p
```
Out[4]: $\displaystyle \left[\begin{matrix}5 & 9 & 4\\8 & 8 & 8\end{matrix}\right]$

The same goes for subtraction. Matricies have to be the same shape, and they are subtracted on an element-by-element basis:

In [5]:
```
x - p
```
Out[5]: $\displaystyle \left[\begin{matrix}-3 & -5 & 2\\0 & 2 & 4\end{matrix}\right]$

2.3 Matrix multiplication by a constant

Matricies can be mutiplied by a constant. Each element is multiplied by the constant:

In [6]:
```
2 * x
```
Out[6]: $\displaystyle \left[\begin{matrix}2 & 4 & 6\\8 & 10 & 12\end{matrix}\right]$

2.4 Multiplying pairs of Matricies

Matricies can be multiplied together but they have to be the correct shape. The number of columns in x has to match the number of rows in y. Notice also the order of multiplication and the shape of the result

Remember that x has a shape of 2x3:

In [7]:
```
x
```
Out[7]: $\displaystyle \left[\begin{matrix}1 & 2 & 3\\4 & 5 & 6\end{matrix}\right]$

y has a shape of 3x2

In [8]:
```
y
```
Out[8]: $\displaystyle \left[\begin{matrix}7 & 8\\9 & 3\\2 & 1\end{matrix}\right]$

So the result will be a 2x2 matrix:

In [9]:
```
x * y
```
Out[9]: $\displaystyle \left[\begin{matrix}31 & 17\\85 & 53\end{matrix}\right]$

And the multiplication won't work with if the matricies are of the wrong shape:

In [26]:
```
k = Matrix([[1,2],[3,4]])
k
```
Out[26]: $\displaystyle \left[\begin{matrix}1 & 2\\3 & 4\end{matrix}\right]$

The next cell will give an error:

In [ ]:
```
x * k
```

2.5 Dot-Product (Inner Product) of Vectors

It is frequently usefull in Machine Learning to compute the 'dot-product' between two vectors. The dot-product measures the projection (shadow) of one vector onto another. The result is a scaller quantity.

In [11]:
```
v1 = Matrix([1,2,3,4,5])
v2 = Matrix([3,2,1,7,6])
v1
```
Out[11]: $\displaystyle \left[\begin{matrix}1\\2\\3\\4\\5\end{matrix}\right]$

In [12]:
```
v2
```
Out[12]: $\displaystyle \left[\begin{matrix}3\\2\\1\\7\\6\end{matrix}\right]$

In [13]:
```
v1.dot(v2)
```
Out[13]: $\displaystyle 68$

Which, because of symmetry is exactly the same as v2.v1:

In [14]:
```
v2.dot(v1)
```
Out[14]: $\displaystyle 68$

2.6 Cross-product of Vectors

The other, common, vector operation is the cross-product. This essentially provides a measure of the area or volume 'enclosed' or 'marked out' by the vectors. It returns a vector result such that the magnitude of the vector is the area / volume and the direction of the vector is normal to the volume / area defined.

Sympy will only perform this operation with vectors of maximum length 3

In [15]:
```
v3 = Matrix([1,2,3])
v4 = Matrix([3,2,1])
v3
```
Out[15]: $\displaystyle \left[\begin{matrix}1\\2\\3\end{matrix}\right]$

In [16]:
```
v4
```
Out[16]: $\displaystyle \left[\begin{matrix}3\\2\\1\end{matrix}\right]$

In [17]:
```
v3.cross(v4)
```
Out[17]: $\displaystyle \left[\begin{matrix}-4\\8\\-4\end{matrix}\right]$

2.7 Vector Norm - Length of a Vector

In machine learning we will often want to compute the length of a vector. There are actualy many (an infinante number!) of ways of doing this .. but the most common is computed in exactly the same way as in normal geometry: the square-root of the sum of the squares of the coordinates.

Interestingly, Sympy shows the result symbolically rather than as a real number .. but then again, that is exactly what Sympy was designed to do!

In [18]:
```
v1.norm()
```
Out[18]: $\displaystyle \sqrt{55}$

In [19]:
```
v2.norm()
```
Out[19]: $\displaystyle 3 \sqrt{11}$

2.7 Determant and Inverse of a Square Matrix

If you are not familiar with the definition, use and computation of determinants then you should check out one of the references (such as Strang) for an explanation. Essentially, square matricies can represent a transformation in m-dimensional space. The determinant indicates the scalling of an object (represented by a matrix / vector) in that space from before to after the transformation. If the determinant is zero, that means that the object is squished to zero volume (length) by the transformation. One implication of this is that there is no inverse defined for a matrx with determinant of zero : once an object is squished flat we can't un-squish it again.

Sympy enables easy computation of the determinant of a square matrix.

In [20]:
```
A = Matrix([[1, 2, 3], [3, 6, 2], [2, 0, 1]] )
A
```
Out[20]: $\displaystyle \left[\begin{matrix}1 & 2 & 3\\3 & 6 & 2\\2 & 0 & 1\end{matrix}\right]$

In [21]:
```
A.det()
```
Out[21]: $\displaystyle -28$

A has a non-zero determinant and therefore the matrix has an inverse:

In [22]:
```
A.inv()
```
Out[22]: $\displaystyle \left[\begin{matrix}- \frac{3}{14} & \frac{1}{14} & \frac{1}{2}\\- \frac{1}{28} & \frac{5}{28} & - \frac{1}{4}\\\frac{3}{7} & - \frac{1}{7} & 0\end{matrix}\right]$

However, if we started with a matrix with a determinant of zero:

In [23]:
```
Z = Matrix([[1,2,3],[4,5,6],[7,8,9]])
Z
```
Out[23]: $\displaystyle \left[\begin{matrix}1 & 2 & 3\\4 & 5 & 6\\7 & 8 & 9\end{matrix}\right]$

In [24]:
```
Z.det()
```
Out[24]: $\displaystyle 0$

.. the inverse of Z is not defined (the following will show an error)

In [ ]:
```
Z.inv()
```

2.8 Other Matrix Operations from Linear Algebra

We will often want to think about matricies as concise representations of a system of linear equations. In that case there are several operations that will help us think about and compute properties of that system:

- Determinant - (as above) - is the matrix invertable / system of equations 'soluble')
- Rank - How many rows / columns in the matrix are independent
- Column space - The sub-space which is 'accessible' ('marked out') by the vectors in the matrix (which may not be a full m-dimensional space, even though the matrix has dimension m x n)
- Null space - For a matrix 'A' - the set of all vectors 'x' that satisfy Ax = 0

The last of these are somewhat advanced concepts. If you are not familiar with these, then you are directed towards Gilbert Strangs "Linear Algebra" for an excellent explanation.

Consider this matrix, representing three equations with 4 unknowns:

In [30]:
```
A = Matrix([[1,3,3,2], [2,6,9,7], [-1,-3,3,4]])
A
```
Out[30]: $\displaystyle \left[\begin{matrix}1 & 3 & 3 & 2\\2 & 6 & 9 & 7\\-1 & -3 & 3 & 4\end{matrix}\right]$

But there are only actually 2 independent rows / columns here:

In [31]:
```
A.rank()
```
Out[31]: 2

That is illustrated by using Gaussian Elimination to transform the matrix into reduced row eschalon form:

In [32]:
```
U = A.rref()
U[0]
```
Out[32]: $\displaystyle \left[\begin{matrix}1 & 3 & 0 & -1\\0 & 0 & 1 & 1\\0 & 0 & 0 & 0\end{matrix}\right]$

(The slightly annoying '[0]' here is because the sympy 'rref' function actually returns a vector result. The zeroth element of that vector is the reduced eschalon form. The second element indicates the 'pivot' columns. The pivot columns are those with that mark out the column space of the matrix. Another way of saying this, is that they are the columns that have non-zero numbers along the 'stair-case diagonal' running from top-left to bottom-right of the reduced matrix.

In [33]:
```
U[1]
```
Out[33]: (0, 2)

And here is that column space (again, I will show each vector in the result separately):

In [34]:
```
C = A.columnspace()
C[0]
```
Out[34]: $\displaystyle \left[\begin{matrix}1\\2\\-1\end{matrix}\right]$

In [35]:
```
C[1]
```
Out[35]: $\displaystyle \left[\begin{matrix}3\\9\\3\end{matrix}\right]$

In this case, all solutions to the original system of equations exist on a 2-dimensional (flat) plane that cuts across 3-dimensional space. The two vectors above define that plane. Every point on the plane is some linear combination of the above two vectors.

The 'nullspace' of the matrix is the set of vectors defining the sub-space that would be mapped onto zero by the matrix. All linear combinations of vectors in this space will be transformed onto zero by the matrix.

In [36]:
```
N = A.nullspace()
N[0]
```
Out[36]: $\displaystyle \left[\begin{matrix}-3\\1\\0\\0\end{matrix}\right]$

In [37]:
```
N[1]
```
Out[37]: $\displaystyle \left[\begin{matrix}1\\0\\-1\\1\end{matrix}\right]$

Which can be demonstrated with some examples:

In [38]:
```
A * N[0]
```
Out[38]: $\displaystyle \left[\begin{matrix}0\\0\\0\end{matrix}\right]$

In [39]:
```
A * (2 * N[0])
```
Out[39]: $\displaystyle \left[\begin{matrix}0\\0\\0\end{matrix}\right]$

In [40]:
```
A * (5 * N[0] + 7 * N[1])
```
Out[40]: $\displaystyle \left[\begin{matrix}0\\0\\0\end{matrix}\right]$

Whereas all other vectors do not have this proprty:

In [41]:
```
NZ = Matrix([-4,1,0,0])
NZ
```
Out[41]: $\displaystyle \left[\begin{matrix}-4\\1\\0\\0\end{matrix}\right]$

In [42]:
```
A * NZ
```
Out[42]: $\displaystyle \left[\begin{matrix}-1\\-2\\1\end{matrix}\right]$

3. Matricies in Numpy

The above should help you understand some of the operations of matricies and Linear Algebra. In practice however, sympy is much less frequently used in Machine Learning than Numpy. In general, in ML we need to be doing fast, numerical computation rather than slower, symbolic manipulation.

In this section I review the use of 'Numpy' - a very common library for numeric calculations in the field of Machine Learning.

In [ ]:
```
import numpy as np

x = np.array([[1,2,2],[4,5,6]])
y = np.array([[7,8],[9,3], [2,1]])

print(x, "\n")
print(y)
```

Note that numpy's data type we are using to represent Matricies is actually called an 'array'

Numpy provides a variety of functions for manipulating matricies, including:

3.1 Matrix Addition

Here I will define a new matrix 'p' to make the addition more interesting.

Note that addition is only define for matricies of the same 'shape'. I.e. with equivalent numbers of rows and columns.

Addition of matricies is particularly easy to read and write due to the re-use of the '+' operator:

In [ ]:
```
p = np.array([[4,7,1],[4,3,2]])
print("x =\n", x)
print("\np =\n", p)
print("\nx +  p =\n",x + p)
```

3.2 Matrix Subtraction

Analagously, the '-' operator performs element-by-element matrix (array) subraction:

In [ ]:
```
print("x =\n", x)
print("\np =\n", p)
print("\nx -  p =\n",x - p)
```

3.3 Matrix multiplication by a Constant

If you simply want to multiply each element of a matrix by a contstant, then that can be achieved using the "*" operator:

In [ ]:
```
print (5 * x)
```

Otherwise, if you want to multiple two matricies together, this can be achieved with the following. Notice, as the convention with matrix multiplication that the rank of the matricies has to match: the number of columns in x has to match the number of rows in y. Notice also the order of multiplication and the shape of the result:

In [ ]:
```
print("x =\n", x)
print("\ny =\n", y)
print("\nmatmul(x,y) =")
print(np.matmul(x,y))
```

or .. using the '@' operator rather than the more familiar '*' operator for matrix muliplication

In [ ]:
```
print(x @ y)
```

3.4 Dot-product (Sometimes called 'inner product')

Numpy also provides a function to calculate the 'dot-product' between two 1-dimensional arrays (also frequently refered to as 'vectors'). The result of the numpy dot-product between two vectors is a number (scaler).

The dot-product measures the 'projection' of one vector onto another. That is, it provides a measure of similarity in the 'direction' of vectors. If vectors are orthogonal (at right-angles) to each other, then the scaler product is defined as zero. Otherwise, the scaler product indicates the size of the 'shadow' one vector would cast on the other.

In text books this operation is often shown as $x.y^{T}$

When thinking about vectors in space it is more frequently indicated as |x|.|y|.cos(theta), where theta is the angle between the lines.

Both of these provide the same result.

The scaler product can be calculated in numpy using 'dot':

In [ ]:
```
v1 = np.array([1,2,3])
v2 = np.array([4,5,6])
print (np.dot(v1,v2))
```

Finally, numpy also provides an 'inner' function. For 1-D vectors the result of 'inner' is identical to the 'dot' product. In higher dimensions these operators perform different operations - however those are beyond the scope of this course.

In [ ]:
```
print (np.inner(v1,v2))
```

3.5 Matrix Division by a constant

We divide each element of a matrix by a constant like this:

In [ ]:
```
print(np.divide(x,2))
```

By analogy with the above, you can also use the '/' operator to divide a matrix by a constant:

In [ ]:
```
print(x/2)
```

We can do an element-by-element divide between two matricies of the same shape:

In [ ]:
```
print (np.divide(x,p))
```

And again, use the '/' operator if more convenient:

In [ ]:
```
print (x/p)
```

3.6 Determinant of a matrix

The Determinant of a matrix is a measure of the 'scaling' that the matrix generates when it is applied to a series of vectors. This is very well explained here: https://towardsdatascience.com/what-really-is-a-matrix-determinant-89c09884164c

Determinants can be calculated for square matricies.

Numpy provides an easy-to-use function to calculate determinants:

In [ ]:
```
s = np.array([[1,2,3], [14,25,36],[74,38,29]])
print(np.linalg.det(s))
```

3.7 Inverse of a matrix

Finally, numpy provides a function to determine the inverse of a matrix:

In [ ]:
```
s_inv = np.linalg.inv(s)
print(s_inv)
```

Matricies, of course, have the property that if they they are multiplied by their inverse the result is the Identify matrix (1's on the diagonal, 0's elsewhere):

In [ ]:
```
print( np.matmul(s,s_inv))
```

Which is a close appoximation to:

$\begin{bmatrix}1 & 0 & 0 \\ 0 & 1 & 0\\ 0 & 0 & 1 \end{bmatrix}$

3.8 Solutions to Linear Equations

As mentioned in the first section, series of linear equations can be represented as matricies.

$1x + 2y = 1$

$3x + 5y = 2$

In the context of Machine Learning you will often see such equations represented in the form of Matricies:

$\begin{bmatrix}1 & 2 \\ 3 & 5 \end{bmatrix}\begin{bmatrix}x \\ y\end{bmatrix}=\begin{bmatrix}1\\ 2 \end{bmatrix}$

Such systems of linear equations may be soluble using numpy. By 'soluble' I mean that we can find values for x and y for which the system of equations is consistent.

In [ ]:
```
M1 = np.array([[1, 2], [3, 5]])

M2 = np.array([1, 2])

A = np.linalg.solve(M1, M2)

print(A)
```

In this case, subsititing a value of '-1' for x, and '1' for y, produces a constent set of equations:

In [ ]:
```
print(1 * -1 + 2 * 1)
```

In [ ]:
```
print (3 * -1 + 5 * 1)
```

Of course, in some cases there will be no solution to a set of linear equation. For example:

$1x + 2y = 1$

$2x + 4y = 3$

represented as:

$\begin{bmatrix}1 & 2 \\ 2 & 4 \end{bmatrix}\begin{bmatrix}x \\ y\end{bmatrix}=\begin{bmatrix}1\\ 3 \end{bmatrix}$

The following should produce a 'Singular Matrix' error, since there are no values for which the equations hold true:

In [ ]:
```
M1 = np.array([[1, 2], [2, 4]])

M2 = np.array([1, 3])

A = np.linalg.solve(M1, M2)

print(A)
```

3.9 Matrix (or vector) Norm

It is often useful to be able to compute the 'length' of a vector. There are many ways to do this (technically there are an infinte number of ways of doing it!). However, the most common methods for computing lengths (distances) in Machine learning are the L1 and L2 norms.

The L2 norm is what, in standard geometry we call a "Euclidean distance" (or "Euclidean norm") and is simply the square root or the sum of the squares of the dimensions. Here is the simplest example .. a 3,4,5 triangle:

In [ ]:
```
myVector = [3,4]
print(np.linalg.norm(myVector))
```

Numpy extends this into higher dimensions, although calculated in an exactly analagous manner:

In [ ]:
```
myVector = [1,2,3]
print(np.linalg.norm(myVector))
```

Numpy can also calculate the 'L1' norm as follows. This is sometimes refered to as the 'Manhattan distance'. The L1 is simply the sum of each vector dimension:

In [ ]:
```
print(np.linalg.norm(myVector, ord = 1))
```

In [ ]: