Linear Algebra

- Vectors and Matrices play central role in IDA.
- Matrices - obvious way to store tabular data.
- Foundation of linear algebra, which is the language of all data analytics algorithms (ml, ai, etc.)

Vectors and Matrices

Vector - 1D array of values:

$$v = \begin{bmatrix} x_1 \\ x_2 \\ \dots \\ x_n \end{bmatrix}$$

Matrix - 2D array of values:

$$A = \begin{bmatrix} x_{11} & x_{12} & \dots & x_{1m} \\ x_{21} & x_{22} & \dots & x_{2m} \\ \dots & \dots & \dots & \dots \\ x_{n1} & x_{n2} & \dots & x_{nm} \end{bmatrix}$$

There are also higher order generalizations of matrices (called *tensors*), which represented 3D or higher arrays of values. These are in fairly common use in modern data science, though typically (but certainly not always), tensors are just used in the "multi-dimensional array" sense, not in their true linear algebra sense. Tensors as linear operators that act on e.g. matrices or other higher-order tensors, are slightly less common most basic data science.

Row and column ordering

Matrices can be laid out in memory by row or by column

$$A = \begin{bmatrix} 100 & 80 \\ 60 & 80 \\ 100 & 100 \end{bmatrix}$$

Row major ordering: 100, 80, 60, 80, 100, 100

Column major ordering: 100, 60, 100, 80, 80, 100

Row major ordering is default for C 2D arrays (and default for Numpy)

Basics and operations

Addition: Two matrices A and B, $[n \times m)$, $A + B = C$ is calculated: $C_{ij} = A_{ij} + B_{ij}$

Transpose: For a matrix A, $(n \times m)$, it's transpose $C = A^T$ is calculated: $C_{ij} = A_{ji}$

Multiplication: For a matrix A ($(n \times m)$) and B ($(m \times p)$) product $C = AB$ is calculated:

$$C_{ij} = \sum_{k=1}^{m} A_{ik} B_{kj}$$

Matrix multiplication is associative ($(AB)C = A(BC)$), distributive $A(B+C) = AB + AC$, but not commutative ($AB \neq BA$).

Identity matrix: The identity matrix I is a square matrix with ones on the diagonal an zeros everywhere else:

$$I = \begin{pmatrix} 1 & 0 & \dots & 0 \\ 0 & 1 & \dots & 0 \\ \dots & \dots & \dots & \dots \\ 0 & 0 & \dots & 1 \end{pmatrix}$$

For any matrix A: $AI = IA = A$

Matix inverse: For a square matrix A, the matrix inverse A^{-1} is the unique matrix such that

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

The matrix inverse need not exist for all square matrices.

Some equations: $(AB)^T = B^T A^T$, $(AB)^{-1} = B^{-1} A^{-1}$

Scalar product of vectors: $x * y = x^T y = \sum_{i=1}^{n} x_i y_i$;

Vector Norm: $||x||_2 = \sqrt{\sum_{i=1}^{n} x_i^2}$

Numpy arrays

Creating an array via `numpy.array` command returns a type `ndarray`. You can also create arrays of zeroes, ones or random numbers (in this case, the `np.randon.randn` create a matrix with standard random normal entries, while `np.random.rand` creates uniform random entries).

```
import numpy as np

b = np.array([-13,9])
A = np.array([[4,-5], [-2,3]])
print(b, "\n")
print(A, "\n")

print(np.ones(40), "\n")           # 1D array of ones
print(np.zeros(4), "\n")          # 1D array of zeros
```

```
print(np.random.rand(40), "\n")          # 1D array of random normal
numbers

print(np.ones((3,4)), "\n")        # 2D array of ones
print(np.zeros((3,4)), "\n")       # 2D array of zeros
print(np.random.randn(3,4))        # 2D array of random normal numbers
```

```
[-13   9]

[[ 4 -5]
 [-2  3]]

[1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1.
1.
 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1.]

[0. 0. 0. 0.]

[0.06728787 0.91425788 0.47196836 0.53282662 0.21708857 0.29636847
 0.12074778 0.00857913 0.04491886 0.25491678 0.83884836 0.50523009
 0.93696112 0.96442306 0.06720672 0.6044625  0.41814113 0.2931281
 0.13785456 0.52992055 0.68001117 0.50593221 0.95358541 0.79283673
 0.24040528 0.83234537 0.87054646 0.23969951 0.71883544 0.82815598
 0.75085477 0.60536751 0.1016104  0.08980058 0.09551223 0.6233089
 0.55297487 0.20505456 0.95430942 0.61408969]

[[1. 1. 1. 1.]
 [1. 1. 1. 1.]
 [1. 1. 1. 1.]]

[[0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]]

[[ 1.53580788 -0.840737    2.27044104  1.9489999 ]
 [-1.17807911  0.21101295  0.22909662  0.4165834 ]
 [-0.41543971 -0.38854401  0.50554714 -1.37578374]]
```

You can also create the indentity matrix using the `np.eye()` command, and a diagonal matrix with the `np.diag()` command.

```
print(np.eye(3),"\n")                    # create array for 3x3
identity matrix
print(np.diag(np.random.randn(3)),"\n")   # create diagonal array
```

```
[[1. 0. 0.]
 [0. 1. 0.]
 [0. 0. 1.]]

[[1 0 0]
```

```
 [0 2 0]
 [0 0 3]]
```

Indexing into numpy arrays

```
A = np.array([[1,2,3], [4,5,6], [7,8,9], [10, 11, 12]])
print(A, "\n")
print(A[1,1],"\n")           # select singe entry
print(A[1,:],"\n")           # select entire row
print(A[1:3, :], "\n")       # slice indexing
```

```
[[ 1  2  3]
 [ 4  5  6]
 [ 7  8  9]
 [10 11 12]]

5

[4 5 6]

[[4 5 6]
 [7 8 9]]
```

```
print(A[1:2,1:2], "\n")  # Select A[1,1] as a singleton 2D array
print(A[[1,2,3],:], "\n")  # select rows 1, 2, and 3
print(A[[2,1,2],:], "\n")  # select rows 2, 1, and 2 again
print(A[[False, True, False, True],:], "\n")  # Select 1st and 3rd
rows
```

```
[[5]]

[[ 4  5  6]
 [ 7  8  9]
 [10 11 12]]

[[7 8 9]
 [4 5 6]
 [7 8 9]]

[[ 4  5  6]
 [10 11 12]]
```

Basic operations on arrays

Arrays can be added/subtracted, multiplied/divided, and transposed, but these **are not all the same** as their linear algebra counterparts. Array multiplication and division are done **elementwise**, they are **not** matrix multiplication or anything related to matrix inversion.

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

print(A+B, "\n") # add A and B elementwise (same as "standard" matrix addition)
print(A-B, "\n") # subtract B from A elementwise (same as "standard" matrix subtraction)
print(A*B, "\n") # elementwise multiplication, _not_ matrix multiplication
print(A/B, "\n") # elementwise division, _not_ matrix inversion
```

```
[[ 2  3  4]
 [ 5  7  7]
 [10  9 12]
 [11 15 13]]

[[ 0  1  2]
 [ 3  3  5]
 [ 4  7  6]
 [ 9  7 11]]

[[ 1  2  3]
 [ 4 10  6]
 [21  8 27]
 [10 44 12]]

[[ 1.          2.          3.        ]
 [ 4.          2.5         6.        ]
 [ 2.33333333  8.          3.        ]
 [10.          2.75       12.        ]]
```

You can transpose arrays, but note this only has meaning for 2D (or higher) arrays. Transposing a 1D array doesn't do anything, since Numpy has no notion of column vectors vs. row vectors for 1D arrays.

```
print(A, "\n")
print(A.T, "\n")
```

```
[[ 1  2  3]
 [ 4  5  6]
 [ 7  8  9]
 [10 11 12]]

[[ 1  4  7 10]
 [ 2  5  8 11]
 [ 3  6  9 12]]
```

Numpy broadcasting

Things start to get very fun when you add/substract/multiply/divide array of different sizes. Rather than throw an error, Numpy will try to make sense of your operation using the Numpy broadcasting rules. This is an advanced topic, which often really throws off newcomers to Numpy, but with a bit of practice the rules become quite intuitive.

```
A = np.ones((4,3))          # A is 4x3
x = np.array([[1,2,3]])      # x is 1x3

print(A, '\n')
print(x, '\n')

print(A*x)                            # repeat x along dimension 4
(repeat four times), and multiply A
```

```
[[1. 1. 1.]
 [1. 1. 1.]
 [1. 1. 1.]
 [1. 1. 1.]]

[[1 2 3]]

[[1. 2. 3.]
 [1. 2. 3.]
 [1. 2. 3.]
 [1. 2. 3.]]
```

Linear Algebra Operations

Starting with Python 3, there is now a matrix multiplication operator `@` defined between numpy arrays (previously one had to use the more cumbersome `np.dot()` function to accomplish the same thing).

```
A = np.random.randn(5,4)
C = np.random.randn(4,3)
x = np.random.randn(4)
y = np.random.randn(5)
z = np.random.randn(4)

print(A @ C, "\n")       # matrix-matrix multiply (returns 2D array)
print(A @ x, "\n")       # matrix-vector multiply (returns 1D array)
print(x @ z)       # inner product (scalar) - be careful about return
type, not an ndarray!
```

```
[[ 0.54038532  0.09443322  0.71702601]
 [ 0.28061454 -2.52658994 -0.17971708]
 [-0.94982818 -1.05939834 -0.96819654]
 [ 2.34049754 -0.11445262  2.30829404]]
```

```
 [ 0.65761741  0.61212264  1.80323206]]

[ 0.21057349 -1.2984257   0.01726926 -0.78865254  1.16874288]

1.0760795880621499
```

By default the @ operator will be applied left-to-right, which may result is very inefficient orders for the matrix multiplication.

```
A = np.random.randn(1000,1000)
B = np.random.randn(1000,2000)
x = np.random.randn(2000)

%timeit A @ B @ x
# print(A @ B @ x)

119 ms ± 13.1 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

This performs the matrix products $(AB)x$, which computes the inefficient matrix multiplication first. If we want to compute the product in the much more efficient order $A(Bx)$, we would use the command

```
%timeit A @ (B @ x)

3.08 ms ± 355 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

Complexity of operations

Assume A, B [$n \times n$], x, y - vectors [n].

Matrix-matrix product AB: $O(n^3)$

Matrix-vector product Ax: $O(n^2)$

Vector-vector inner product $x^T y$: $O(n)$

Matrix inverse/solve: A^{-1}: $O(n^3)$

Finally, Numpy includes the routine `np.linalg.inv()` for computing the matrix inverse A^{-1} and `np.linalg.solve()` for computing the matrix solve $A^{-1}b$.

```
b = np.array([-13,9])
A = np.array([[4,-5], [-2,3]])

print(np.linalg.inv(A), "\n")   # explicitly form inverse
print(np.linalg.solve(A,b))     # compute solution A^{-1}b

[[1.5 2.5]
 [1.  2. ]]
```

```
[3. 5.]
```

Sparse Matrices

Many matrices are sparse (contain mostly zero entries, with only a few non-zero entries)

Examples: matrices formed by real-world graphs, document-word count matrices (more on both of these later)

Storing all these zeros in a standard matrix format can be a huge waste of computation and memory

Sparse matrix libraries provide an efficient means for handling these sparse matrices, storing and operating only on non-zero entries

There are several different ways of storing sparse matrices, each optimized for different operations

Coordinate (COO) format: store each entry as a tuple (row_index, col_index, value)

$$A = \begin{bmatrix} 0 & 0 & 3 & 0 \\ 2 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \\ 4 & 0 & 1 & 0 \end{bmatrix}$$

Instead: $values = [2,4,1,3,1,1]$ $row-indices = [1,3,2,0,3,1]$
$column-indices = [0,0,1,2,2,3]$

Python sparse matrix libraries

https://docs.scipy.org/doc/scipy/reference/sparse.html

```
import scipy.sparse as sp


values = [2, 4, 1, 3, 1, 1]
row_indices = [1, 3, 2, 0, 3, 1]
column_indices = [0, 0, 1, 2, 2, 3]
A = sp.coo_matrix((values, (row_indices, column_indices)),
shape=(4,4))
print(A.todense())
```

```
[[0 0 3 0]
 [2 0 0 1]
 [0 1 0 0]
 [4 0 1 0]]
```