Python for Scientific Data Analysis

NumPy/SciPy

Section 5: Basic Linear Algebra with NumPy and SciPy

Linear Algebra is a cornerstone of computational mathematics. The simple linear algebra you probably learned in high school or undergrad consists of small vectors and matrices. But the real power with linear algebra rests in very large matrices and vectors. Since you cannot realistically solve these by hand, computers come in. Hence, the importance of numerical linear algebra.

Both NumPy and SciPy have numerous linear algebra routines. In this section, we will focus on the simplest ones with NumPy, describing how to do basic linear algebra operations. Similarly, in the next section, we will discuss the simplest modeling techniques with SciPy.

Later in the course, after we have learned *Matplotlib*, *Pandas*, and *AstroPy* , we will build upon this foundation to describe more advanced linear algebra operations with both NumPy and SciPy and advanced modeling with SciPy. But for now, this short introduction to linear algebra and modeling will allow us to do useful things with packages.

Creating Vectors and Matrices with NumPy

Creating vectors and matrices for linear algebra operations is straightforward with NumPy. In fact, you have done them already. E.g.

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

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

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

a_column_vector.shape
#(6, 1)

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

Note that the first example -- *a*vector_ -- is orientationless: it is neither a row nor column vector but just a 1D list of numbers in NumPy. Orientation in NumPy is given by brackets `[ ]`. The outermost brackets group all of the numbers together into one object. Then, each additional set of brackets indicates a row: so a row vector has all numbers in one row, while a column vector has multiple rows, with each row containing one number. A matrix then will have multiple rows with each row containing more than one number.

Array arithmetic rules -- e.g. adding, subtracting -- follow those described in Section 2.

Basic Linear Algebra Operations

Vector Norms

The magnitude or *norm* of a vector is the distance from head to tail of the vector. It is computed using the standard Euclidean distance formula: the square-root of the sum of squared vector elements:

|| v || = $\sqrt{\sum_{i=1}^{n} v_i^2}$

In Python this can be computed as `np.linalg.norm(v)`.

Sometimes we don't want the vector as-is but want a corresponding *unit vector*, where ||v|| = 1. To produce this unit vector you have to take the input vector and divide by its norm.

Matrix Determinant

The *determinant* is a key property associated with a square matrix. It has a special importance when we later get to inverting matrices or do eigendecomposition of them. The determinant of matrix A is usually noted as $det(A)$ or |A|.

As you might remember, it's pretty easy to calculate determinants for small matrices, e.g. 2x2:

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} = ad - bc$$

For much larger matrices this gets quickly unwieldy, although you may remember how to use cofactor expansion or Gauss-Jordan elimination to compute the determinants of slightly larger matrices.

Thankfully, we can just ask NumPy:

`matrix_det=np.linalg.det([matrix])`.

Or SciPy

`matrix_det=scipy.linalg.det([matrix])`.

Vector and Matrix Multiplication

The real power with NumPy's linear algebra routines come from multiplying large matrices and vectors, which we obviously cannot do by hand. Routines for how to do this are below ...

Dot Product

The `dot` operation does vector-vector, matrix-vector, and matrix-matrix multiplication of two arrays: `np.dot([first array],[second array])`. E.g. `np.dot(a,b)`.

Mathematically, what this operation does for each is:

The vector-vector operation, resulting in scalar s from vectors x and y:

$s = \sum_i x_i y_i$

The matrix-vector operation, resulting in vector y from matrix A and vector x:

$y_i = \sum_j A_{ij} x_j$

The vector-matrix operation, resulting in vector y from matrix A and vector x:

$y_j = \sum_i x_i A_{ij}$

The matrix-matrix operation, resulting in matrix C from matrices A and B:

$C_{ij} = \sum_k A_{ik} B_{kj}$

Again, in all cases, the operation is `np.dot([first array],[second array])`

Np.Matmul

NumPy provides another, simpler routine for *matrix* multiplication, called (unsurprisingly) `np.matmul([first matrix,second matrix])`.

E.g. in our previous example of matrices A and B this matrix multiplicaiton results in matrix C:

$C_{ij} = \sum_k A_{ik} B_{kj}$

For this case, the call is `np.matmul(A,B)`.

Now, there will be plenty of other matrix operations of greater sophistication and power. But we will get to these later.

Outer Product

Here's another useful operation: the *outer product*. The outer product of two vectors creates a matrix. In order to make this work, one vector must be a column vector and another must be a row vector.

e.g.

$$\begin{bmatrix} a \\ b \\ c \end{bmatrix} \begin{bmatrix} d & e \end{bmatrix} = \begin{bmatrix} ad & ae \\ bd & be \\ cd & ce \end{bmatrix}$$

In code, the outer product is computed as `np.outer(a,b)`. Below is an example:

```
v1=np.array([[1,2,3,4]])
#array([[1, 2, 3, 4]])

v2=np.array([[5],[6],[7]])*0.5
#array([[2.5],
#       [3. ],
#       [3.5]])

mat1=np.outer(v1,v2)
#array([[ 2.5,  3. ,  3.5],
#       [ 5. ,  6. ,  7. ],
#       [ 7.5,  9. , 10.5],
#       [10. , 12. , 14. ]])
```

Note also, the `np.dot` function can *also* compute an outer product PROVIDED that one vector is a column vector and another is a row vector.

Solving Basic Linear Equations

Now, we can take the power of numerical linear algebra to the next level by using it to solve systems of linear equations. ***Many*** coding problems I encounter are really just dressed up versions of a simple task to solve linear equations.

For instance, consider the following set of linear equations:

$$x_1 + 4x_2 = 1$$
$$3x_1 + 4x_2 = -4$$

Now, you could just solve this by hand fairly fast ($x_1 = -2.5, x_2 = 0.875$). But why do that when NumPy and SciPy will solve it for you & do it faster?

Recast the system of linear equations as a matrix equation of the form $\mathbf{Ax} = \mathbf{b}$. Here, $\mathbf{A}$ equals the coefficients in front of the variables: $\begin{bmatrix} 1 & 4 \\ 3 & 4 \end{bmatrix}$. And $\mathbf{b}$ equals the right-hand side of the equation: a column vector: $\begin{bmatrix} 1 \\ -4 \end{bmatrix}$. The unknown $\mathbf{x}$ equals the two variables x_1 and x_2.

In Python this is written as:

```
 a=np.array([[1,4,],[3,4]]) #a two-by-two matrix
 b=np.array([[1],[-4]]) #a column matrix
```

We can then solve this system of linear equations by matrix *inversion*. Thankfully, NumPy and SciPy have a LOT of tools to do this in various ways. For now, we will focus on the most obvious, brute-force way: a simple inversion:

$$\mathbf{x}=\mathbf{A}^{-1}\mathbf{b}$$

Here are various ways to solve it in NumPy:

`np.linalg.solve`

```
a=np.array([[1,4,],[3,4]]) #a two-by-two matrix
b=np.array([[1],[-4]]) #a column matrix

x=np.linalg.solve(a,b)
#array([[-2.5  ],
     [ 0.875]])
```

Using `np.linalg.inv` (the inversion function) and `np.dot`

```
a=np.array([[1,4,],[3,4]]) #a two-by-two matrix
b=np.array([[1],[-4]]) #a column matrix

np.linalg.inv(a).dot(b)
#array([[-2.5  ],
     [ 0.875]])
```

Using SciPy's `linalg.solve`

```
from scipy import linalg
linalg.solve(a,b)
#array([[-2.5  ],
     [ 0.875]])
```

Using SciPy's `linalg.inv` and `np.dot`

```
from scipy import linalg
a=np.array([[1,4,],[3,4]]) #a two-by-two matrix
b=np.array([[1],[-4]]) #a column matrix

linalg.inv(a).dot(b)
#array([[-2.5  ],
     [ 0.875]])
```

Singular Value Decomposition

Singular value decomposition is another ultra-powerful linear algebra operation. For my research, it becomes especially powerful in cases where the data are noisy, such that a full matrix inversion tends to amplify noise (which, you know, is bad). Because we live in the real world, *all* data are noisy to some level, sometimes at a level that matters. Hence SVD.

SVD decomposes a matrix of M rows and N into the product of three matrices: the left singular vector U, the singular values Σ, and a right singular vector V^T:

$$\mathbf{A} = \mathbf{U}\ \Sigma\ \mathbf{V}^T$$

Note that the dimensions of each of these matrices is different: U has dimensions of MxM (i.e. a square matrix equal to the number of rows), the singular values Σ have MxN dimensions, and the right singular vector V^T has NxN dimensions.

In Numpy, the function to do SVD is `np.linalg.svd`, which returns a 3-element tuple of NumPy arrays. E.g.

```
aaa=np.array([[1,2,3,4],[5,6,7,8],[9,10,11,12],[13,14,15,16],[17,18,19,20
]])

aaa.shape
#(5,4)

U,s,Vt=np.linalg.svd(aaa)

S=np.zeros(np.shape(aaa)) ##THIS IS IMPORTANT!
np.fill_diagonal(S,s)

#Array values

#U
#array([[-0.09654784, -0.76855612,  0.60361934,  0.11849959,  0.14697464]
,
       [-0.24551564, -0.4896142 , -0.58187858, -0.58473646,  0.13964448],
       [-0.39448345, -0.21067228, -0.45158207,  0.7625265 , -0.12094203],
       [-0.54345125,  0.06826963,  0.23432255, -0.24484197, -0.76494794],
       [-0.69241905,  0.34721155,  0.19551877, -0.05144765,  0.59927085]]
)

#S
#array([[3.86226568e+01, 0.00000000e+00, 0.00000000e+00, 0.00000000e+00],
       [0.00000000e+00, 2.07132307e+00, 0.00000000e+00, 0.00000000e+00],
       [0.00000000e+00, 0.00000000e+00, 7.60977226e-16, 0.00000000e+00],
       [0.00000000e+00, 0.00000000e+00, 0.00000000e+00, 3.86063773e-16]])

#Vt
#array([[-0.44301884, -0.47987252, -0.51672621, -0.55357989],
       [ 0.70974242,  0.26404992, -0.18164258, -0.62733508],
       [-0.03307776, -0.36339946,  0.82603221, -0.42955499],
       [ 0.54672284, -0.75361849, -0.13293153,  0.33982718]])



#dimensionality

U.shape
#(5,5)

S.shape
#(5,4)

Vt.shape
#(4,4)
```

In the above example, note the cautions about the singular value matrix, s. What np.linalg.svd actually returns for s is a 1-D vector of singular values (my guess is that this is for speed/memory issues).

So, caution: Σ is a diagonal matrix with these values, but the matrix itself is 2-D (the off-diagonal elements are zero). To get the full middle matrix you have to create another array equal in dimensionality to the original matrix, set all elements originally to zero, and then fill the diagonals with s.

We are going to stop at this point with SVDs, but do know they will become useful when trying to solve linear equations/least squares problems.

Eigendecomposition

Another ultra-powerful linear algebra tool in NumPy and SciPy is ***eigendecomposition***. Don't know what this is? Sure you've heard of it, because it is closely related to both SVD and to this magical thing called *principal component analysis* which we will get to later in the course.

Eigendecomposition works for square matrices. Like SVD, it often crops up with correlation or covariance matrices (more on that later). Every square matrix of dimensions MxM has M eigenvalues (scalars) and M corresponding eigenvectors. Eigendecomposition then computes these scalar-vector pairs.

The main eigenvalue equation is very simple:

$\mathbf{A}\mathbf{v}=\lambda\mathbf{v}$. To be clear this is not saying that the matrix A is the same thing as a scalar λ. Rather, the effect or consequence of the matrix operating on the vector is the same as the effect of the scalar operating on that same vector.

So how can we find eigenvalues? Thankfully, NumPy makes this very easy with `np.linalg.eig`. E.g.

```
matrix=np.array([[1,2],[3,4]])
evals,evecs=np.linalg.eig(matrix)

evals
#array([-0.37228132,  5.37228132])

evecs
#array([[-0.82456484, -0.41597356],
       [ 0.56576746, -0.90937671]])
```

And we can do the same thing with SciPy

```
from scipy import linalg
matrix=np.array([[1,2],[3,4]])
evals,evecs=linalg.eig(matrix)

evals
#array([-0.37228132,  5.37228132])

evecs
#array([[-0.82456484, -0.41597356],
       [ 0.56576746, -0.90937671]])
```

Okay fine, we can compute eigenvalues and eigenvectors. So? We gain a bit more insight if we rearrange the eigenvalue equation:

$\mathbf{A}\mathbf{v}=\lambda\mathbf{v}$

$\mathbf{A}\mathbf{v}-\lambda\mathbf{v}=0$

$(\mathbf{A}-\lambda\mathbf{I})\mathbf{v}=0$

Note the imposition of the identity matrix $\mathbf{I}$ (A matrix minus a scalar is not a thing/wrong dimensionality: we have to operate element-wise).

And then ...

$\tilde{\mathbf{A}}$ = $\mathbf{A}$-$\lambda\mathbf{I}$, so

$\tilde{\mathbf{A}}\mathbf{v}$ = 0

In other words, the eigenvector is in the nullspace of the matrix shifted by its eigenvalue. A matrix shifted by its eigenvalue is singular, because only singular matrices have non-trivial (e.g. v=0) null space. Because singular matrices have determinants of zero, |$\mathbf{A}$-$\lambda\mathbf{I}$| =0. I.e. we shift a matrix by its (unknown) eigenvalue λ, set its determinant to zero and solve for λ. For the case of a simple 2x2 matrix with elements $\begin{bmatrix} a & b \\ c & d \end{bmatrix}$, shifting by eigenvalues λ leads to the equation:

$\begin{bmatrix} a & b \\ c & d \end{bmatrix}$ - λ $\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$, or $\lambda^2 - (a+d)\lambda + (ad - bc)$ = 0. I.e. this is a second order polynomial equation with thus two solutions. A 3x3 matrix? The leading term is λ^3 and so on. In other words, the characteristic polynomial of an MxM matrix will have λ^M term and a MxM matrix will have M eigenvalues.

Now, so far this seems a lot like SVD. We have decomposed a matrix and get back eigenvalues and eigenvectors. The power with both SVD and eigendecomposition comes later because we can make Python *truncate* these matrices to filter out noise (or in more jargony language, filter out the components that do not explain much of the variance we see). Look for a section on PCA to describe this.