EECS 189 Project T Final: NumPy

Abhinav G., Shrey V.,
Department of Computer Science
University of California, Berkeley
Berkeley, CA 94704
`(abhinavg, shreyv) @berkeley.edu`

In the machine learning world today, it is almost impossible to imagine big companies write their ML code without the use of important packages like NumPy. NumPy provides users with quick, optimized linear algebra datatypes and methods, and is renowned for its simplicity and easiness to use. With optimizations for various types of hardware, CPUs, and GPUs, as well as highly optimized code for linear algebra operations, NumPy is unmatched in its utility in today's world. With its blazing fast operations and its versatility, it is important for any student entering the world of computing for linear algebra and machine learning to understand NumPy.

1 Linear Algebra Background

To understand the note as well as the associated jupyter notebook assignment, we assume students to have the basic linear algebra background provided from EECS16A. Here, we briefly go over a recap of the background information.

1.1 Vectors and Matrices

A vector is a collection of numbers that represent a value. Vectors contain several "elements" which are the individual entries of the vector. Here's an example of a vector representing "3 apples, 1 tomato, and 2 bananas":

$[\, 3 \quad 1 \quad 2 \,]$

A matrix, on the other hand, represents a collection of vectors. Frequently, we use matrices to represent data. For instance, in a store, we might have the following orders from different customers:
John: 1 apple, 1 banana, 1 orange
Linda: 2 apples, 3 bananas, 4 oranges
Mary: 3 apples, 3 bananas, 4 oranges
We would represent this in matrix form as

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

With a recollection of these ideas from 16A, let's move on to slightly more complicated material.

1.2 Vector Operations

Vectors can undergo operations of addition, subtraction, scalar multiplication, and dot products.
Addition and subtraction is simple: The resultant vector is simply the sum or difference of the respective components.
Scalar multiplication, on the other hand, multiplies all of the vector's elements by the scalar constant.
Dot products multiply components element-wise, and sum up all the products.

1.3 Matrix Operations

Matrices may be added, subtracted, multiplied, and inverted. Addition and subtraction is the simplest of the operations, requiring equal dimensional matrices, and simply adding/subtracting element-wise.

Multiplication, on the other hand, is more involved. We begin with matrix-vector multiplication. If we hope to multiply a matrix and vector, then we would take the dot product of each row of the matrix and the vector, giving the resultant vector's components in order.

Matrix-matrix multiplication, is very similar. We can treat matrix-matrix multiplication as a horizontally stacked multiplication of matrix-vectors. We first generate the matrix-vector product of the matrix and the first column vector of the second matrix, and keep going through all of the vectors of the second matrix, horizontally stacking the products.

The inverse of a matrix is A is the corresponding matrix A* that yields identity when multiplied with A. A*A = AA* = I. A must be a square matrix to have a corresponding inverse.

1.4 Eigenvectors and Eigenvalues

The eigenvectors of a matrix are the vectors x such that $Ax = \lambda x$, where λ is a constant. These vectors are particularly important because they represent the axes along a linear transformation. Why is this so? This is because the product of a matrix times the eigenvector remains in the same direction as the eigenvector.

1.5 Diagonalization

With the knowledge of eigenvalues and eigenvectors, we can move to one of the most important topics of linear algebra, diagonalization. Every matrix A can be diagonalized into the product of matrices VDV^{-1}, where D is a diagonal matrix.

In fact, we can diagonalize a matrix simply by finding its eigenvalues and vectors. We find the eigenvalues of a matrix A, and these eigenvalues are placed on the diagonal of D. After this, the i^{th} eigenvector is the i^{th} column of V, and has a corresponding eigenvalue which is the i^{th} entry of D.

1.6 Least Squares

When we try to solve the system Xw = y, we generally exploit the use of matrix inverses to obtain the solution $w = X^{-1}y$. However, in the real world, our observations are frequently noisy, and solutions cannot be exactly obtained. We must instead obtain the closest estimate to the solution.

One notion of the "closest" solution is the solution with minimum L-2 norm distance of the y-predictions using the proposed solution to the actual measured y values. The way to obtain this solution is through the technique of least squares.

We first note that the y values that we can hope to obtain have to lie in the column space of the X matrix. This is because Xw is simply a linear combination of the columns of X, and this is how our prediction is obtained. We now reach the following intuition: what is the closest point in X's column space to y? From EECS16A, we know the closest point is the orthogonal projection of y onto the column space of X.

From this intuition, how do we obtain a closed bound solution? If the solution perpendicularly projects y onto X, then
$X^T(y - Xw^*) = 0$. Here, w^* is the least-squares solution. With some algebra manipulation, we find $w^* = (X^TX)^{-1}X^Ty$

2 Use of NumPy in Linear Algebra

Now that we have gone through a brief linear algebra review of the concepts taught in 16A, we can move to the use of NumPy in practice.

2.1 The Basics

NumPy, first and foremost, is used to create vectors and arrays for quick operations. This is done using the following syntax:

```
import numpy as np
a = [1,2,3,4] # basic python list
np_a = np.array(a) # NumPy vector for a

mat = [[1,2,3,4], [4,5,6,7]] # basic python 2D List
np_mat = np.array(mat) # NumPy matrix for mat

np_mat.shape # shape of mat, returns (2,4)
```

With these vectors and matrices, we are now ready to do complex operations. Let's consider matrix products, sums, differences, inverses, as well as vector dot products.

```
import numpy as np
a = [1,2,3,4] # basic python list
np_a = np.array(a) # NumPy vector for a

mat = [[1,2,3,4], [4,5,6,7]] # basic python 2D List
np_mat = np.array(mat) # NumPy matrix for mat

mat2 = [[1,2],[3,4],[5,6],[7,8]]

sum_matrices = np_mat + np_mat
difference_matrices = np_mat - np_map
product = np_mat @ np_mat

invertable matrix = np.array([[1,2],[2,3]])
inverse = np.linalg.inv(invertable_matrix)

dot_product = np.dot(a, a)
```

2.2 Special Matrices

In linear algebra, there are certain matrices that are particularly special, having useful properties that have connections to probability, machine learning, and statistics. The most important of these matrices are the identity matrix, the zero matrix, and the set of diagonal matrices. Here, we will provide you with examples for how to build these special matrices with NumPy functions.

```
import numpy as np
identity = np.eye(5) # 5x5 Identity Matrix

diagonal_matrix = np.diag([1,3,4,5])

zero_matrix = np.zeros((5,10)) # 5x10 zero matrix
```

2.3 Matrix Multiplication and np.einsum

Matrix multiplication is always going to be used in many machine learning algorithms. In Python, the simplest way to multiply two matrices A and B as AB is `A @ B`. There are several other methods that may become more handy in complex problems. The most generic matrix manipulation function is `np.einsum`.

The first argument of np.einsum is always a string specifying the mathematical operation being done on an array or matrix. Suppose you have matrix A with dimension 2x3 and matrix B with dimension 3x4. If we let `i = range(2), j = range(3), k = range(4)`, then the matrix multiplication specifying AB (a 2x4 matrix) is `np.einsum('ij, jk -> ik', A, B)`. In other words, we are taking element (i, j) of A and element (j, k) of B and combining them to create element (i, k) of matrix AB. Below is a brief example illustrating how to compute a transpose of a matrix with `np.einsum`.

```
import numpy as np
A = np.array([3, 4, 7, 10, 500, 20]).reshape(2, 3)
A_transpose = np.einsum('ji', A) # will output the 3x2 transpose of A
```

2.4 Eigenvalues, Diagonalization, Inverses

In order to do important operations that pertain to eigenvalues, eigenvectors, and inverses, NumPy has functions explictly optimized to satisfy this purpose. To diagonalize, as explained earlier in the note, simply combine the eigenvalues and vectors into the appropriate matrices.

```
import numpy as np
a = np.array([[5,1],[2,3]])
a_inv = np.linalg.inv(a) # inverts a

eigenvalues, eigenvectors = np.linalg.eig(a) # note that eigenvectors are stored
    columnwise!
```

2.5 NumPy Least Squares

NumPy also has built-in functionality for least squares with the given X and y matrices. Least squares, as explained earlier, will allow users to generate estimates for coefficients to solve Xw = y.

```
import numpy as np
X = np.array([[1,2,32], [2,5,12], [2,2,2], [6,7,1]])
y = np.array([4,34,2,1])
w = np.linalg.lstsq(X,y)[0] # first term in the lstsq function will give us the
    coefficients
```

3 Final Notes

With the power of NumPy at your hand, computations for graphs and problems of tens of hundreds, even thousands, of degrees, are made extremely fast. Having the knowledge of NumPy is a prerequisite to entering the machine learning world, and is an excellent way to quickly test ideas and operations on large data.