SSE 691: Intro to Data Science Project 2

Jonathan Alfred
9-17-2016

Contents

Introduction

In this project, important concepts of linear algebra were explored. These concepts were applied to data science and important functionalities were both created and tested in python, utilizing lists as vectors and matrices. NumPy was also explored, as linear algebra concepts are already built into this library.

Linear Algebra

This section gives an overview of linear algebra and how linear algebra applies to data science.

What is Linear Algebra?

Linear algebra is the branch of mathematics that deals with vector spaces. Vector spaces are collections of vectors. These are essential lines with a magnitude and direction. These can be used to represent forces and motion, but can be used to represent less physical objects, such as a collection of data points representing a person. Vector operations, such vector addition and scalar multiplication (Figure 1), must meet several requirements, called axioms. These requirements are for things like Commutativity of addition, where the order of addition of vectors does not matter.

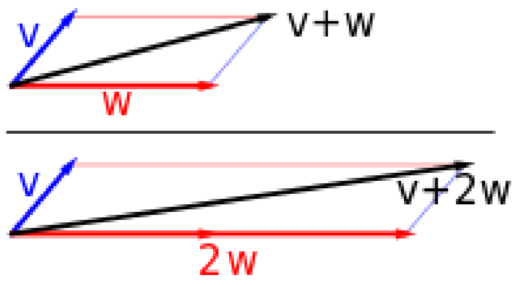

Figure 1 - Vector addition and scalar muliplication

The study of linear algebra first came to be around the late 17th and early 18th centuries. Several of the fundamental topics, such as matrices did not emerge until the 19th century. Linear algebra looks into interactions of vectors within vector space, looking to find common properties among all vector spaces and studying lines, planes and subplanes. Its applications are very diverse, ranging from analytical geometry to social sciences.

Linear Algebra in Data Science

Linear algebra is a fundamental pillar of data science. Basic concepts, such as matrices and vectors, are used in practically everything. Mathematical modeling makes great use of vectors and matrices. An example of this would be a model of a rocket. Vectors would be used for the motion of the rocket, along with the derivations of its velocity and acceleration. These would be placed in a 3x3 matrix, used within a variety of different equations to produce a predicted path of a rocket along a path. An image of the predicted path can be seen in Figure 2.

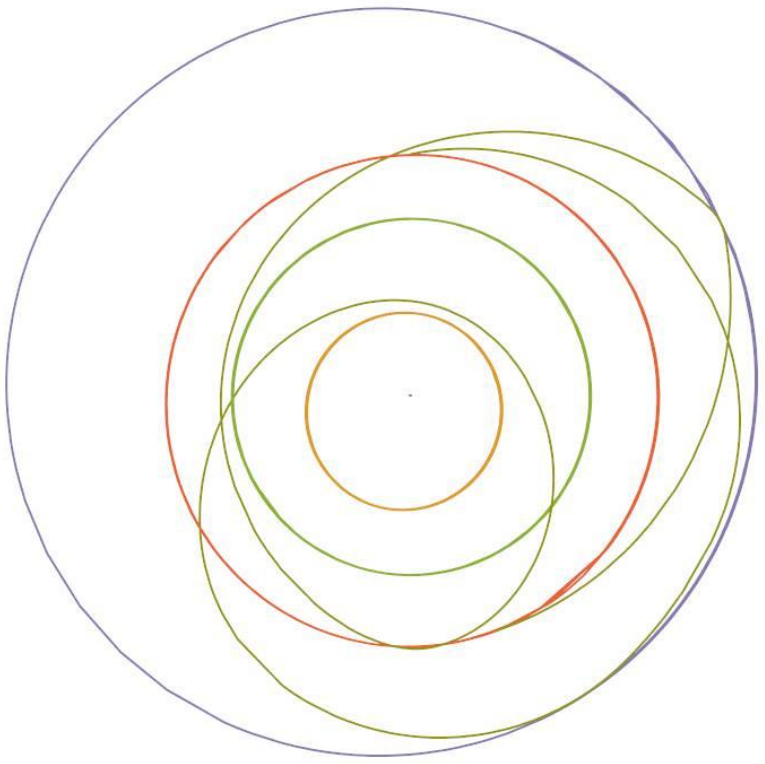

Figure 2 - Mathematica model illustrating the Rich Purnell Manuever see in the Martian. The orbits of Earth, Mercury, Venus, and Mars are all included.

Linear algebra is used in machine learning. Many different matrix operations are used in neural networks. Learning concepts make use of vectors to change the weights within the neural network. Errors seen in the output are corrected using different types of learning iterations to compute a correction to weights within the neural network, making use of vector manipulations and matrix operations.

Important Concepts

This section goes over the two very key topics used in linear algebra: Vectors and Matrices

Vectors

Chapter 4 goes over many of the different operations used with vectors. It showed just how the different operations were coded into python. Examples were provided to code these operations for lists, though it was sure to note that lists were not optimal for resources and that these operations were not efficient, but simply for to serve as an introduction to linear algebra techniques.

The operations that were described in the chapter are seen below:

- Vector Addition. Vectors are added component wise and returned as vector of the same dimensions as the added vectors. The two vectors being added together must be of the same dimension, or the addition will fail. Our code does not account for this.

```
def vector_add(v,w):
    return [v_i + w_i for v_i, w_i in zip (v,w)]
```

- Vector Subtraction. This is similar to vector addition, in that the vectors are manipulated component-wise, but are subtracted instead of added. As with addition, the vectors must be of the same dimensions or the subtraction will fail.

```
def vector_subtract(v,w):
    return [v_i -w_i for v_i, w_i in zip (v,w)]
```

- Vector Summation. This is vector addition over more than just two vectors. It returns one single vector with the same dimensions as the added vectors.

```
def vector_sum(vectors):
    return reduce(vector_add, vectors)
```

- Scalar multiplication. A scalar number is multiplied by all components of the vector. Each component is multiplied and returned a vector of the same size as the original vector.

```
def scalar_multiply(c, v):
    return [c*v_i for v_i in v]
```

- Vector Mean. This takes a list of vectors, component wise adds them, and then takes the average of the vectors, returning that to the user.

```
def vector_mean(vectors):
    n = len(vectors)
    return scalar_multiply(1/n, vector_sum(vectors))
```

- Dot Product. The dot product is the component wise multiplication of two vectors. After the multiplication, all the components are multiplied together.

```
def dot(v, w):
    return sum(v_i*w_i for v_i, w_i in zip(v, w))
```

- Sum of Squares. Sum of squares is the dot product of a vector with itself. This squares all components of the vector, before summing the values up.

```
def sum_of_squares(v):
    return dot(v, v)
```

- Magnitude. The square root of the sum of squares is the magnitude of a vector. This is the length of the vector as an absolute value.

```
def magnitude(v):
    return math.sqrt(sum_of_squares(v))
```

- Distance. This refers to the distance between the two ends of a vector. There were two methodologies used, both returning the same value.

```
def squared_distance(v, w):
    return sum_of_squares(vector_subtract(v,w))

def distance_v1(v, w):
    return math.sqrt(squared_distance(v,w))

def distance_v2(v, w):
    return magnitude(vector_subtract(v,w))
```

Each of these functions were run with example code and produced the following output.

```
Vector Addition: [1, 4] + [3, 2] = [4, 6]
Vector Subtraction: [-1, 3] - [2, -2] = [-3, 5]
Vector Sum:
 Vector List -> [[1, 4], [3, 2], [-1, 3], [2, -2], [-3, -4]]
 Answer -> [2, 3]
Scalar Multiply: [2, -2] * 4 =  [8, -8]
Vector Mean:
 Vector List -> [[1, 4], [3, 2], [-1, 3], [2, -2], [-3, -4]]
 Answer -> [0.4, 0.6000000000000001]
Dot Product: [-3, -4] . [1, 4] = -19
Sum of Squares: for [3, 2] -> 13
Magnitude: for [-1, 3] -> 3.1622776601683795
Distance Squared: for [2, -2] and [-3, -4] -> 29
Distance: for [1, 4] and [3, 2]
 Version 1 -> 2.8284271247461903
 Version 2 -> 2.8284271247461903
```

Matrices

Matrices are two-dimensional collections of numbers. They are rectangular, meaning that each row has an equal number of values and each column has an equal number of values. An example of a 2 x 3 matrix is seen below:

```
A = [[1,2,3],
     [4,5,6]]
```

In this class, matrices will be used for a few reasons. The first is to represent large data sets consisting of multiple vectors. An example of this would be a data set consisting heights, weights, and ages for 1000 people. Another reason would be to represent linear functions that map k-dimensional vectors to n-dimensional vectors. This will be explored later on as the techniques learned grow to involve such functionality. The last reason for matrices is to represent binary relationships. This can be something as simple as showing whether or not someone knows someone else. If the person does know another, then the value would be a one. Otherwise, it would be a zero. True/false relationships can be represented very easily with little overhead.

NumPy

In this chapter, all of the code was written utilizing lists. While this is good for expositional work, to show the reader how many of the different functions are coded, in practice, lists are inefficient. They require large amounts of overhead and operate slower due to the possibility of having mixed data types.

As we progress through the book, we will begin to make more use of the NumPy array. These take up significantly less data than a Python list. Accessing data items with the NumPy array is much quicker as well. This is due to Python Lists being a collection of pointers, where as an array is a grouped memory chunk without a reference to another pointer. NumPy arrays are also made of one single data type, rather than multiple data types.

Discussion

Linear algebra is a topic that I have worked in heavily. In prior semesters, multiple classes that I have taken, such as Scientific Modeling with Mathematica and several control classes, required linear algebra. I may not have even realized at the time that the work was linear algebra. MatLab made use of arrays fairly often when doing signal analysis and matrices of equations as well. Some of the topics that I am learning currently, such as Deep Q Learning, make use of linear algebra as well.

This has shown me several of the fundamentals of linear algebra. It has been a good refresher for techniques that I have used before. Showing me how it was coded as well has been rather interesting, as many techniques I would not have thought to use were implemented.

Indirect Logs

Date	Minutes	Activity Description
9/15/2016	180	Went through chapter 4.
9/16/2016	180	Continued through chapter 4. Began report.
9/17/2016	120	Continued work on report
9/18/2016	120	Finished Report.
Total	600	

References

[1] "Linear Algebra". *Wikipedia*. N.p., 2016. Web. 17 Sept. 2016.

[2] "Vector Space". *Wikipedia*. N.p., 2016. Web. 17 Sept. 2016.

[3] Why NumPy instead of Python lists? (n.d.). Retrieved September 18, 2016, from

http://stackoverflow.com/questions/993984/why-numpy-instead-of-python-lists

Image Credits

Figure Number	Credit
Figure 1	https://en.wikipedia.org/wiki/Vector_space
Figure 2	Jonathan Alfred

Code

```
def vector_add(v,w):
    return [v_i + w_i for v_i, w_i in zip (v,w)]

def vector_subtract(v,w):
    return [v_i -w_i for v_i, w_i in zip (v,w)]

from functools import reduce

def vector_sum(vectors):
    return reduce(vector_add, vectors)
```

```
def scalar_multiply(c, v):
    return [c*v_i for v_i in v]

def vector_mean(vectors):
    n = len(vectors)
    return scalar_multiply(1/n, vector_sum(vectors))

def dot(v, w):
    return sum(v_i*w_i for v_i, w_i in zip(v, w))

def sum_of_squares(v):
    return dot(v, v)

import math

def magnitude(v):
    return math.sqrt(sum_of_squares(v))

def squared_distance(v, w):
    return sum_of_squares(vector_subtract(v,w))

def distance_v1(v, w):
    return math.sqrt(squared_distance(v,w))

def distance_v2(v, w):
    return magnitude(vector_subtract(v,w))

#just using two dimensional vectors. Note this is just to show functionality of
things to do with linear algebra.
vector_1 = [1, 4]
vector_2 = [3, 2]
vector_3 = [-1, 3]
vector_4 = [2, -2]
vector_5 = [-3, -4]
vector_list = [vector_1, vector_2, vector_3, vector_4, vector_5]

print("Vector Addition: {0} + {1} = {2}".format(vector_1, vector_2,
vector_add(vector_1, vector_2)))
print("Vector Subtraction: {0} - {1} = {2}".format(vector_3, vector_4,
vector_subtract(vector_3, vector_4)))
print("Vector Sum: \n Vector List -> {} \n Answer -> {}".format(vector_list,
vector_sum(vector_list)))
print("Scalar Multiply: {} * {} =  {}".format(vector_4, 4, scalar_multiply(4,
vector_4)))
print("Vector Mean: \n Vector List -> {} \n Answer -> {}".format(vector_list,
vector_mean(vector_list)))
print("Dot Product: {} . {} = {}".format(vector_5, vector_1, dot(vector_5,
vector_1)))
print("Sum of Squares: for {} -> {}".format(vector_2, sum_of_squares(vector_2)))
print("Magnitude: for {} -> {}".format(vector_3, magnitude(vector_3)))
print("Distance Squared: for {} and {} -> {}".format(vector_4, vector_5,
squared_distance(vector_4, vector_5)))
print("Distance: for {} and {}\n Version 1 -> {} \n Version 2 -> {}".format(vector_1,
                                                                            vector_2,
distance_v1(vector_1, vector_2),

distance_v2(vector_1, vector_2)))

def shape(A):
    num_rows = len(A)
    num_cols = len(A[0])
    return num_rows, num_cols

def get_row(A, i):
    return A[i]

def get_column(A, j):
    return [A_i[j] for A_i in A]

def make_matrix(num_rows, num_cols, entry_fn):
    return [[entry_fn(i, j)
            for j in range(num_cols)]
           for i in range(num_rows)]

def is_diagonal(i, j):
    return 1 if i == j else 0

identity_matrix = make_matrix(5,5, is_diagonal)
print(identity_matrix)
```