Linear Algebra

Linear algebra is the branch of mathematics that deals with *vector spaces*. Although I can't hope to teach you linear algebra in a brief chapter, it underpins a large number of data science concepts and techniques, which means I owe it to you to at least try. What we learn in this chapter we'll use heavily throughout the rest of the book.

Vectors

Abstractly, *vectors* are objects that can be added together (to form new vectors) and that can be multiplied by *scalars* (i.e., numbers), also to form new vectors.

Concretely (for us), vectors are points in some finite-dimensional space. Although you might not think of your data as vectors, they are a good way to represent numeric data.

For example, if you have the heights, weights, and ages of a large number of people, you can treat your data as three-dimensional vectors `(height, weight, age)`. If you're teaching a class with four exams, you can treat student grades as four-dimensional vectors `(exam1, exam2, exam3, exam4)`.

The simplest from-scratch approach is to represent vectors as lists of numbers. A list of three numbers corresponds to a vector in three-dimensional space, and vice versa:

In [15]:
```
height_weight_age = [
    70,  # inches,
    170, # pounds,
    40,  # years
]
grades = [
    95, # exam1
    80, # exam2
    75, # exam3
    62, # exam4
]
print( height_weight_age )
print( grades )
```

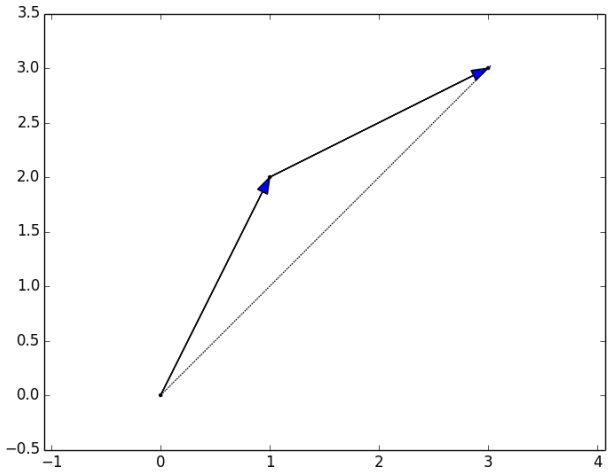
```
[70, 170, 40]
[95, 80, 75, 62]
```

One problem with this approach is that we will want to perform *arithmetic* on vectors. Because Python lists aren't vectors (and hence provide no facilities for vector arithmetic), we'll need to build these arithmetic tools ourselves. So let's start with that.

To begin with, we'll frequently need to add two vectors. Vectors add *componentwise*. This means that if two vectors v and w are the same length, their sum is just the vector whose first element is `v[0] + w[0]`, whose second element is `v[1] + w[1]`, and so on. (If they're not the same length, then we're not allowed to add them.)

For example, adding the vectors `[1, 2]` and `[2, 1]` results in `[1 + 2, 2 + 1]` or `[3, 3]`, as shown in Figure 4-1.

Figure 4-1: ***Adding two vectors***



We can easily implement this by zip-ing the vectors together and using a list comprehension to add the corresponding elements:

In [16]:
```
def vector_add(v, w):
    """adds corresponding elements"""
    return [ v_i + w_i for v_i, w_i in zip(v, w) ]
vector_add([1, 2, 3], [4, 5, 6])
```

Out[16]:
```
[5, 7, 9]
```

Similarly, to subtract two vectors we just subtract corresponding elements:

In [17]:
```
def vector_subtract(v, w):
    """subtracts corresponding elements"""
    return [ v_i - w_i for v_i, w_i in zip(v, w) ]
vector_subtract([1, 2, 3], [4, 5, 6])
```

Out[17]:
```
[-3, -3, -3]
```

We'll also sometimes want to componentwise sum a list of vectors. That is, create a new vector whose first element is the sum of all the first elements, whose second element is the sum of all the second elements, and so on. The easiest way to do this is by adding one vector at a time:

In [18]:
```
def vector_sum(vectors):
    """sums all corresponding elements"""
    result = vectors[0]                          # start with the fi
    for vector in vectors[1:]:                   # then loop over th
        result = vector_add(result, vector)      # and add them to t
    return result
vector_sum([[0, 1, 2], [1, 2, 3], [2, 3, 4], [3, 4, 5]])
```

Out[18]: [6, 10, 14]

If you think about it, we are just `reduce` -ing the list of vectors using `vector_add` , which means we can rewrite this more briefly using higher-order functions:

In [19]:
```
from functools import reduce
def vector_sum(vectors):
    return reduce(vector_add, vectors)
vector_sum([[0, 1, 2], [1, 2, 3], [2, 3, 4], [3, 4, 5]])
```

Out[19]: [6, 10, 14]

or even:

In [20]:
```
from functools import partial
vector_sum = partial(reduce, vector_add)
vector_sum([[0, 1, 2], [1, 2, 3], [2, 3, 4], [3, 4, 5]])
```

Out[20]: [6, 10, 14]

although this last one is probably more clever than helpful.

We'll also need to be able to multiply a vector by a scalar, which we do simply by multiplying each element of the vector by that number:

In [21]:
```
def scalar_multiply(c, v):
    """c is a number, v is a vector"""
    return [c * v_i for v_i in v]
scalar_multiply(7, [ 2, 4, 8, 5, 3 ])
```

Out[21]: [14, 28, 56, 35, 21]

This allows us to compute the componentwise means of a list of (same-sized) vectors:

In [22]:
```
def vector_mean(vectors):
    """compute the vector whose ith element is the mean of the
    ith elements of the input vectors"""
    n = len(vectors)
    return scalar_multiply(1/n, vector_sum(vectors))
vector_mean([[0, 1, 2], [1, 2, 3], [2, 3, 4], [3, 4, 5]])
```

Out[22]: [1.5, 2.5, 3.5]

A less obvious tool is the *dot product*. The dot product of two vectors is the sum of their

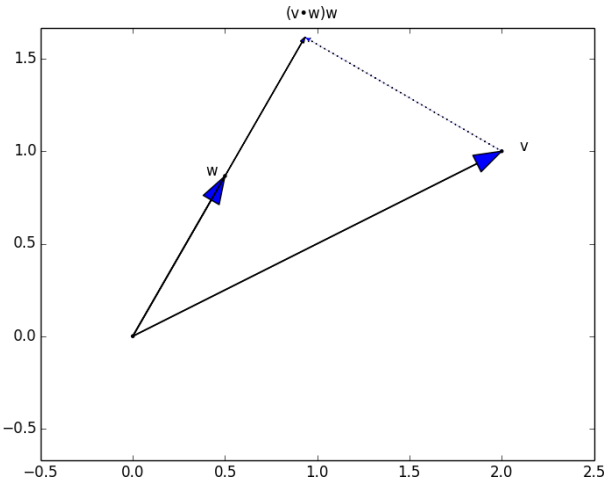
componentwise products:

In [23]:
```
def dot(v, w):
    """v_1 * w_1 + ... + v_n * w_n"""
    return sum(v_i * w_i for v_i, w_i in zip(v, w))
dot([ 1, 3, 5, 7 ], [ 2, 4, 6, 8 ])
```

Out[23]: 100

The dot product measures how far the vector v extends in the w direction. For example, if `w = [1, 0]` then `dot(v, w)` is just the first component of v. Another way of saying this is that it's the length of the vector you'd get if you *projected* v onto w (Figure 4-2).

Figure 4-2: ***The dot product as vector projection***



Using this, it's easy to compute a vector's *sum of squares* :

In [24]:
```
def sum_of_squares(v):
    """v_1 * v_1 + ... + v_n * v_n"""
    return dot(v, v)
sum_of_squares([ 1, 2, 3, 4, 5 ])
```

Out[24]: 55

Which we can use to compute its *magnitude* (or length):

In [25]:
```
import math
def magnitude(v):
    return math.sqrt(sum_of_squares(v))         # math.sqrt is squar
magnitude([1, 2, 3, 4, 5])
```

Out[25]: 7.416198487095663

We now have all the pieces we need to compute the distance between two vectors, defined as:

$$\sqrt{(v_1 - w_1)^2 + \cdots + (v_n - w_n)^2}$$

In [26]:
```
def squared_distance(v, w):
    """(v_1 - w_1) ** 2 + ... + (v_n - w_n) ** 2"""
    return sum_of_squares(vector_subtract(v, w))
def distance(v, w):
    return math.sqrt(squared_distance(v, w))
distance([1, 2, 3, 4], [5, 6, 7, 8])
```

Out[26]: 8.0

Which is possibly clearer if we write it as (the equivalent):

In [27]:
```
def distance(v, w):
    return magnitude(vector_subtract(v, w))
distance([1, 2, 3, 4], [5, 6, 7, 8])
```

Out[27]: 8.0

That should be plenty to get us started. We'll be using these functions heavily throughout the book.

> ***Note***
> Using lists as vectors is great for exposition but terrible for performance.
> In production code, you would want to use the NumPy library, which includes a high-performance array class with all sorts of arithmetic operations included.

Matrices

A *matrix* is a two-dimensional collection of numbers. We will represent matrices as `lists` of `lists`, with each inner list having the same size and representing a row of the matrix. If A is a matrix, then `A[i][j]` is the element in the ith row and the jth column. Per mathematical convention, we will typically use capital letters to represent matrices. For example:

In [28]:
```
A = [[1, 2, 3],   # A has 2 rows and 3 columns
     [4, 5, 6]]
B = [[1, 2],      # B has 3 rows and 2 columns
     [3, 4],
     [5, 6]]
print( A )
print( B )
```

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

> ***Note***
> In mathematics, you would usually name the first row of the matrix "row 1" and the first column "column 1." Because we're representing matrices with Python

> lists, which are zero-indexed, we'll call the first row of a matrix "row 0" and the first column "column 0."

Given this list-of-lists representation, the matrix A has `len(A)` rows and `len(A[0])` columns, which we consider its shape:

In [29]:
```
def shape(A):
    num_rows = len(A)
    num_cols = len(A[0]) if A else 0        # number of elements in
    return num_rows, num_cols
shape([[1, 2, 3],
       [4, 5, 6]])
```

Out[29]:
```
(2, 3)
```

If a matrix has n rows and k columns, we will refer to it as a $n \times k$ matrix. We can (and sometimes will) think of each row of a $n \times k$ matrix as a vector of length k, and each column as a vector of length n:

In [31]:
```
def get_row(A, i):
    return A[i]                 # A[i] is already the ith row
def get_column(A, j):
    return [A_i[j]              # jth element of row A_i
            for A_i in A]       # for each row A_i
A = [[1, 2, 3],
     [4, 5, 6],
     [7, 8, 9]]
print( get_row(A, 0) )
print( get_column(A, 2) )
```

```
[1, 2, 3]
[3, 6, 9]
```

We'll also want to be able to create a matrix given its shape and a function for generating its elements. We can do this using a nested list comprehension:

In [33]:
```
def make_matrix(num_rows, num_cols, entry_fn):
    """returns a num_rows x num_cols matrix
    whose (i,j)th entry is entry_fn(i, j)"""
    return [[entry_fn(i, j)             # given i, create a list
             for j in range(num_cols)]  #    [entry_fn(i, 0), ... ]
            for i in range(num_rows)]   # create one list for each i
```

Given this function, you could make a 5×5 *identity matrix* (with `1`s on the diagonal and `0`s elsewhere) with:

In [34]:
```
def is_diagonal(i, j):
    """1's on the 'diagonal', 0's everywhere else"""
    return 1 if i == j else 0
identity_matrix = make_matrix(5, 5, is_diagonal)
identity_matrix
```

Out[34]:
```
[[1, 0, 0, 0, 0],
 [0, 1, 0, 0, 0],
 [0, 0, 1, 0, 0],
 [0, 0, 0, 1, 0],
 [0, 0, 0, 0, 1]]
```

Output:
```
[[1, 0, 0, 0, 0],
 [0, 1, 0, 0, 0],
 [0, 0, 1, 0, 0],
 [0, 0, 0, 1, 0],
 [0, 0, 0, 0, 1]]
```

Matrices will be important to us for several reasons.

First, we can use a matrix to represent a data set consisting of multiple vectors, simply by considering each vector as a row of the matrix. For example, if you had the heights, weights, and ages of 1,000 people you could put them in a $1,000 \times 3$ matrix:

In [35]:
```
data = [
    [70, 170, 40],
    [65, 120, 26],
    [77, 250, 19],
    # ....
]
```

Second, as we'll see later, we can use an $n \times k$ matrix to represent a linear function that maps k-dimensional vectors to n-dimensional vectors. Several of our techniques and concepts will involve such functions.

Third, matrices can be used to represent binary relationships. In Chapter 1, we represented the edges of a network as a collection of pairs `(i, j)`. An alternative representation would be to create a matrix A such that `A[i][j]` is `1` if nodes i and j are connected and `0` otherwise.

Recall that before we had:

In [36]:
```
friendships = [
    (0, 1), (0, 2), (1, 2), (1, 3), (2, 3), (3, 4),
    (4, 5), (5, 6), (5, 7), (6, 8), (7, 8), (8, 9),
]
friendships
```

Out[36]:
```
[(0, 1),
 (0, 2),
 (1, 2),
 (1, 3),
 (2, 3),
 (3, 4),
 (4, 5),
 (5, 6),
 (5, 7),
 (6, 8),
 (7, 8),
 (8, 9)]
```

We could also represent this as:

In [37]:
```
    #     user 0    1    2    3    4    5    6    7    8    9
    #
friendships = [[0,   1,   1,   0,   0,   0,   0,   0,   0,   0],   #
               [1,   0,   1,   1,   0,   0,   0,   0,   0,   0],   #
               [1,   1,   0,   1,   0,   0,   0,   0,   0,   0],   #
               [0,   1,   1,   0,   1,   0,   0,   0,   0,   0],   #
               [0,   0,   0,   1,   0,   1,   0,   0,   0,   0],   #
               [0,   0,   0,   0,   1,   0,   1,   1,   0,   0],   #
               [0,   0,   0,   0,   0,   1,   0,   0,   1,   0],   #
               [0,   0,   0,   0,   0,   1,   0,   0,   1,   0],   #
               [0,   0,   0,   0,   0,   0,   1,   1,   0,   1],   #
               [0,   0,   0,   0,   0,   0,   0,   0,   1,   0]]   #
friendships
```

Out[37]:
```
[[0, 1, 1, 0, 0, 0, 0, 0, 0, 0],
 [1, 0, 1, 1, 0, 0, 0, 0, 0, 0],
 [1, 1, 0, 1, 0, 0, 0, 0, 0, 0],
 [0, 1, 1, 0, 1, 0, 0, 0, 0, 0],
 [0, 0, 0, 1, 0, 1, 0, 0, 0, 0],
 [0, 0, 0, 0, 1, 0, 1, 1, 0, 0],
 [0, 0, 0, 0, 0, 1, 0, 0, 1, 0],
 [0, 0, 0, 0, 0, 1, 0, 0, 1, 0],
 [0, 0, 0, 0, 0, 0, 1, 1, 0, 1],
 [0, 0, 0, 0, 0, 0, 0, 0, 1, 0]]
```

If there are very few connections, this is a much more inefficient representation, since you end up having to store a lot of zeroes. However, with the matrix representation it is much quicker to check whether two nodes are connected—you just have to do a matrix lookup instead of (potentially) inspecting every edge:

In [38]:
```
print( friendships[0][2] == 1 )         # True, 0 and 2 are friends
print( friendships[0][8] == 1 )         # False, 0 and 8 are not fri
```

```
True
False
```

Similarly, to find the connections a node has, you only need to inspect the column (or the row) corresponding to that node:

In [39]:
```
friends_of_five = [
    i                                               # only need
    for i, is_friend in enumerate(friendships[5]) # to look at
    if is_friend                                    # one row
]
friends_of_five
```

Out[39]:
```
[4, 6, 7]
```

Previously we added a list of connections to each node object to speed up this process, but for a large, evolving graph that would probably be too expensive and difficult to maintain.

We'll revisit matrices throughout the book.

For Further Exploration

For Further Exploration

- Linear algebra is widely used by data scientists (frequently implicitly, and not infrequently by people who don't understand it). It wouldn't be a bad idea to read a textbook. You can find several freely available online:" -- Linear Algebra, from UC Davis -- Linear Algebra, from Saint Michael's College -- If you are feeling adventurous, Linear Algebra Done Wrong is a more advanced introduction
- All of the machinery we built here you get for free if you use NumPy. (You get a lot more too.)