Álgebra Linear

Vetores

Resumidamente, os vetores são objetos que podem ser adicionados em conjunto (para formar novos vetores) e que podem ser multiplicados por escalares (isto é, números), também para formar novos vetores. Concretamente (para nós), os vetores são pontos em algum espaço de dimensões finitas. Embora você não pense em seus dados como vetores, eles são uma boa maneira de representar dados numéricos.
Por exemplo, se você tem alturas, pesos e idades de um grande número de pessoas, você pode tratar seus dados como vetores tridimensionais (altura, peso e idade). Se você leciona uma disciplina com quatro exames, você pode tratar notas de alunos como vetores de quatro dimensões (exame1, exame2, exame3, exame4).

In [12]:

```
import math
from functools import reduce
```

In [13]:

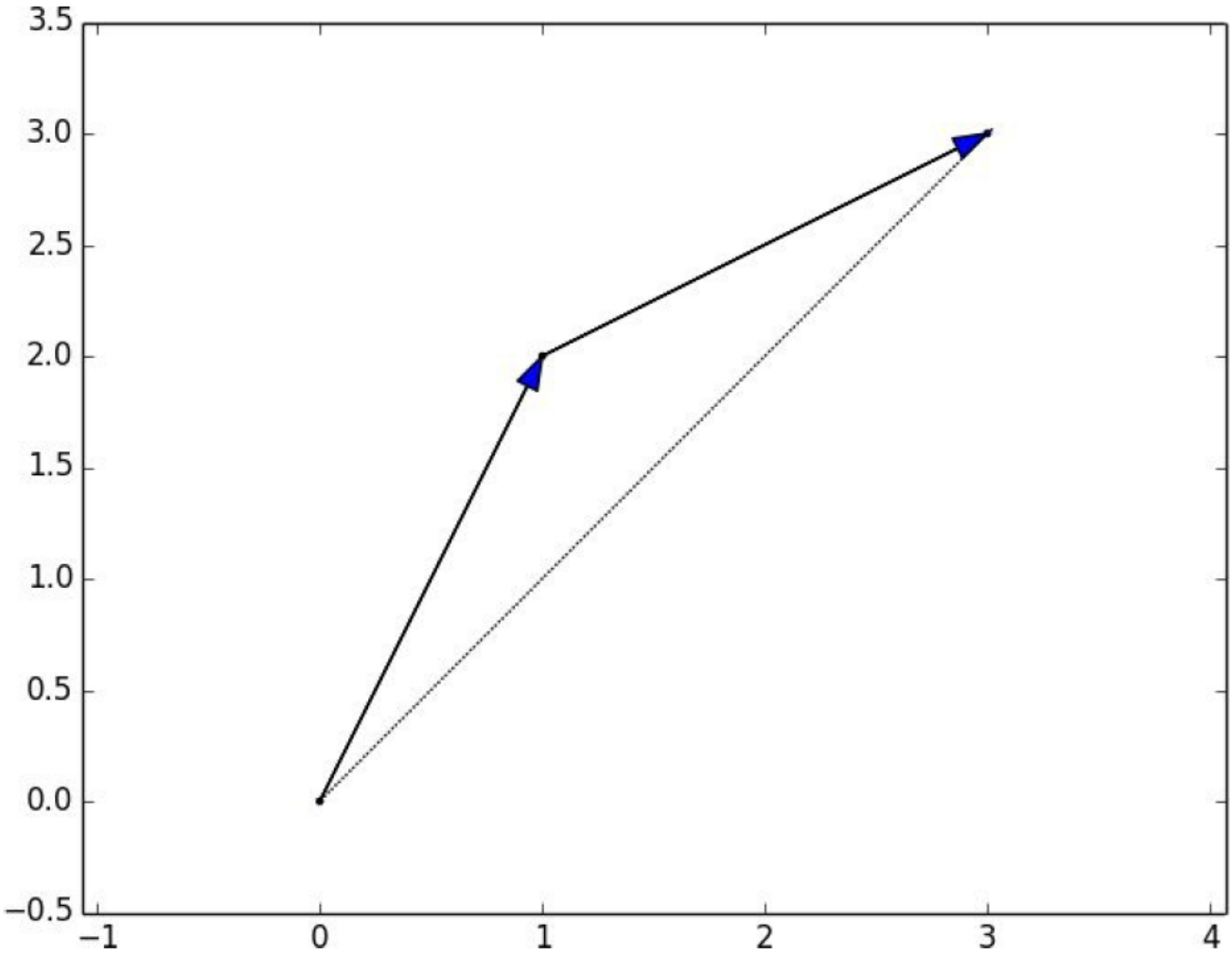
```
height_weight_age = [175, # cm,
                     90, # kg,
                     40] # anos
print(height_weight_age)

grades = [95, # exam1
          80, # exam2
          75, # exam3
          62 ] # exam4
print(grades)
```

```
[175, 90, 40]
[95, 80, 75, 62]
```

Soma de Vetores

Frequentemente precisamos somar dois vetores. Isso significa que, se dois vetores v e w forem do mesmo comprimento, sua soma é apenas o vetor cujo primeiro elemento é v[0] + w[0], cujo segundo elemento é v[1] + w[1], e assim em diante. (Se eles não tiverem o mesmo comprimento, então não podemos somá-los.)

Por exemplo, adicionar os vetores [1, 2] e [2, 1] resulta em [1 + 2, 2 + 1] ou [3, 3], como mostrado na figura abaixo:



Podemos implementar isso facilmente ao compactar os vetores juntos e usar uma list comprehension para adicionar os elementos correspondentes:

In [14]:

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

In [15]:

```
print(vector_add([1, 2, 3], [3, 2, 1]))
```

```
[4, 4, 4]
```

Da mesma forma, para subtrair dois vetores, subtraimos elementos correspondentes:

In [16]:

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

In [17]:

```
print(vector_subtract([1, 2, 3], [3, 2, 1]))
```

```
[-2, 0, 2]
```

Também às vezes queremos constituir um componente de uma lista de vetores, ou seja, criar um novo vetor cujo primeiro elemento seja a soma de todos os primeiros elementos, o segundo elemento é a soma de todos os segundos elementos, e assim por diante. A maneira mais fácil de fazer isso é adicionando um vetor por vez:

In [18]:

```
def vector_sum(vectors):
    """sums all corresponding elements"""
    result = vectors[0] # start with the first vector
    for vector in vectors[1:]: # then loop over the others
        result = vector_add(result, vector) # and add them to the result
    return result
```

In [19]:

```
print(vector_sum([[1, 2, 3],
                  [3, 2, 1]]))
```

```
[4, 4, 4]
```

Se você pensar sobre isso notará que estamos apenas reduzindo a lista de vetores usando vector_add, o que significa que podemos reescrever isso usando outras funções:

In [20]:

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

In [21]:

```
print(vector_sum([[1, 2, 3], [3, 2, 1]]))
```

```
[4, 4, 4]
```

Escalar

Também precisamos poder multiplicar um vetor por um escalar, o que fazemos simplesmente multiplicando cada elemento do vetor por esse número:

In [22]:

```
def scalar_multiply(c, v):
    """c is a number, v is a vector"""
    return [c * v_i for v_i in v]
```

In [23]:

```
print(scalar_multiply(3, [3, 2, 1]))
```

```
[9, 6, 3]
```

Isso nos permite calcular a média dos componentes de uma lista de vetores do mesmo tamanho:

In [24]:

```
def vector_mean(vectors):
    """compute the vector whose ith element is the mean of the ith elements of t
he input vectors"""
    n = len(vectors)
    return scalar_multiply(1/n, vector_sum(vectors))
```

In [25]:

```
print(vector_mean([[1, 2, 3], [3, 2, 1]]))
```

```
[2.0, 2.0, 2.0]
```

Produto Escalar

Uma ferramenta menos óbvia é o produto escalar. O produto escalar de dois vetores é a soma dos produtos de seus componentes:

In [26]:

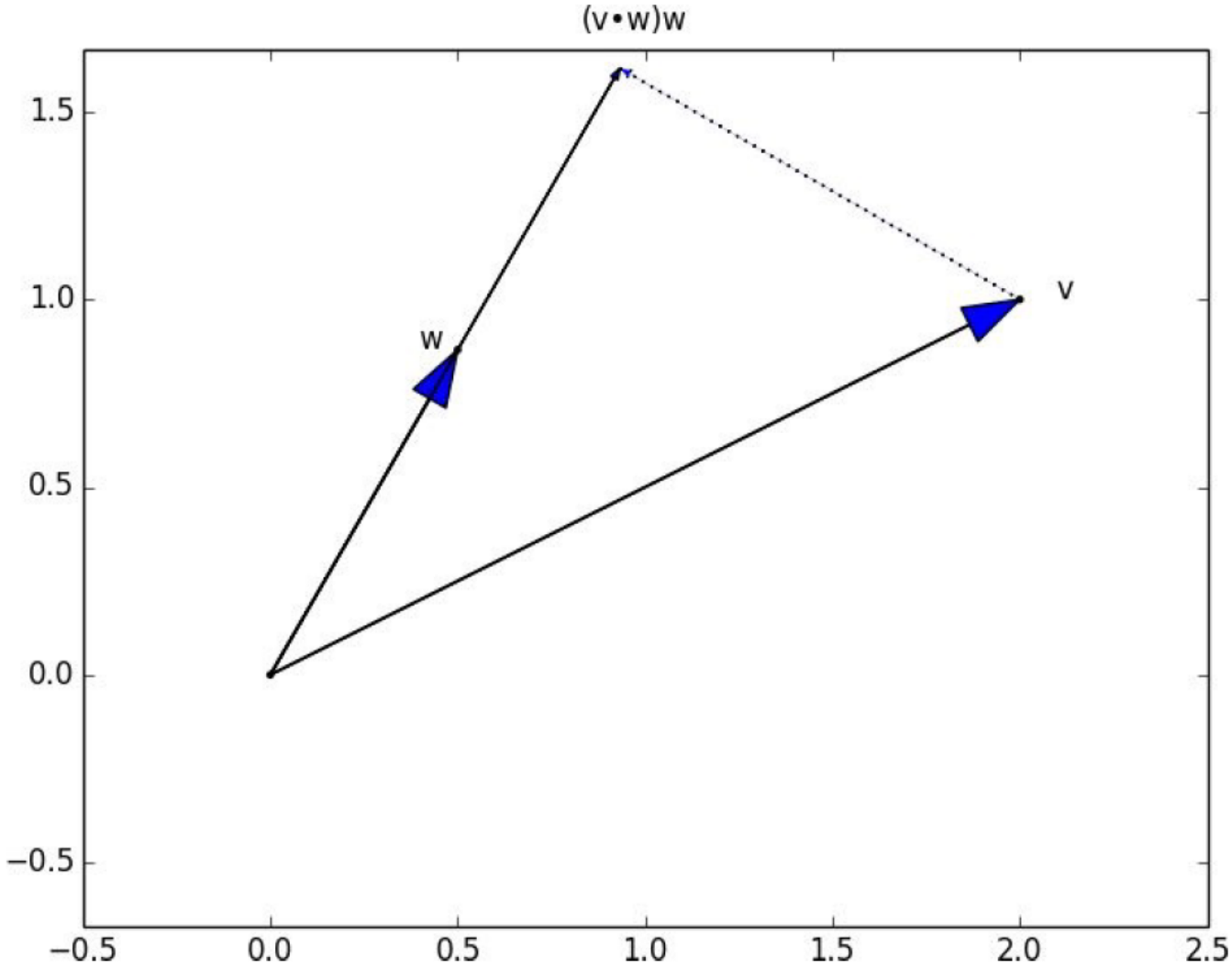
```
def dot(v, w):
    """v_1 * w_1 + ... + v_n * w_n"""
    return sum(v_i * w_i
           for v_i, w_i in zip(v, w))
```

In [27]:

```
print(dot([1, 2, 3], [3, 2, 1]))
```

```
10
```

O produto escalar mede o quão longe o vetor v se estende na direção w. Outra maneira de dizer isso é que é o comprimento do vetor que você obtém se você projetar v em w.



Usando a função de produto escalar, é fácil calcular a soma dos quadrados de um vetor:

In [28]:

```
def sum_of_squares(v):
    """v_1 * v_1 + ... + v_n * v_n"""
    return dot(v, v)
```

In [29]:

```
print(sum_of_squares([1, 2, 3]))
```

```
14
```

O que podemos usar para calcular sua magnitude (ou comprimento):

In [30]:

```
def magnitude(v):
    return math.sqrt(sum_of_squares(v)) # math.sqrt is square root function
```

In [31]:

```
print(magnitude([1, 2, 3]))
```

```
3.7416573867739413
```

Agora temos todas as peças que precisamos para calcular a distância entre dois vetores, definidos como:

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

In [32]:

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

Matrizes

Vamos representar matrizes como listas de listas, com cada lista interna com o mesmo tamanho e representando uma linha da matriz. Se A é uma matriz, então, A[i][j] é o elemento na *inésima* linha e na *jésima* coluna. Por convenção matemática, normalmente usaremos letras MAIÚSCULAS para representar matrizes.

In [33]:

```
A = [[1, 2, 3], # A has 2 rows and 3 columns
     [4, 5, 6]]
print("Matriz A = ", A)

B = [[1, 2], # B has 3 rows and 2 columns
     [3, 4],
     [5, 6]]
print("Matriz B = ", B)
```

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

Dada esta representação de listas de listas, a matriz A tem linhas len(A) e colunas len(A[0]), que consideramos sua forma:

In [34]:

```
def shape(A):
    num_rows = len(A)
    num_cols = len(A[0]) if A else 0 # number of elements in first row
    return num_rows, num_cols
```

In [35]:

```
print(shape([[1, 2, 3], [3, 2, 1]]))
```

```
(2, 3)
```

Se uma matriz tiver n linhas e k colunas, nós nos referiremos a ela como uma matriz "n X k". Podemos (e às vezes pensamos) em cada linha de uma matriz "n X k" como um vetor de comprimento "k", e cada coluna como vetor de comprimento "n":

In [36]:

```
def get_row(A, i):
    return A[i] # A[i] is 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
```

In [37]:

```
print(get_row([[1, 2, 3], [3, 2, 1]], 1))
```

```
[3, 2, 1]
```

In [38]:

```
print(get_column([[1, 2, 3], [3, 2, 1]], 2))
```

```
[3, 1]
```

Também queremos criar uma matriz dada a sua forma e uma função para gerar seus elementos. Podemos fazer isso usando uma list comprehension aninhada:

In [39]:

```
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
```

Dada esta função, você pode fazer uma matriz de identidade 5 × 5 (com 1s na diagonal e 0s em outro lugar) com:

In [40]:

```
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)
print(identity_matrix)
```

```
[[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]]
```

Matrizes serão importantes para nós por vários motivos. Podemos usar uma matriz para representar um conjunto de dados que consiste de vários vetores, simplesmente considerando cada vetor como uma linha da matriz. Por exemplo, se você tivesse as alturas, pesos e idades de 1.000 pessoas, você poderia colocá-las em uma matriz 1.000X3:

data =
[[70, 170, 40],
[65, 120, 26],
[77, 250, 19],

....

]

Matrizes podem ser usadas para representar relacionamentos binários. Na primeira aula, representamos as conexões de uma rede como uma coleção de pares (i, j). Uma representação alternativa seria criar uma matriz A tal que A[i][j] seja 1 se os nós i e j estiverem conectados e 0 caso contrário. Lembre-se que antes de termos:

In [41]:

```
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)]
```

In [42]:

```
print(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)]
```

Também podemos representar como:

In [43]:

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

In [44]:

```
print(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]]
```

In [ ]:

```

```