Professor Rodrigo Bossini
https://sites.google.com/site/professorrodrigobossini/

Data Science com Python 3
Álgebra Linear

A fim de realizar cálculos estatísticos importantes para a ciência de dados, iremos precisar de alguns conceitos fundamentais da álgebra linear. Por isso, iremos revisitá-los neste material.

Passo 1 (Vetores) Informalmente, um vetor é uma coleção de pontos. **Observação**, **amostra** entre outros são nomes também utilizados para se referir a vetores. Em nossos exemplos, um usuário da rede social poderia ser um vetor. Como veremos, vetores podem ter a eles aplicados diversas operações de interesse que resultam em novos vetores, valores escalares etc.

1.1 Veja alguns exemplos:
Uma pessoa com altura, peso e idade é um vetor tridimensional.
Um aluno que fez quatro testes é um vetor com quatro dimensões (cada nota é uma dimensão).

1.2 Para representar vetores, iremos utilizar listas simples. Veja a Listagem 1.2.1.

Listagem 1.2.1

```
#vetor com três dimensões
heigth_weight_age = [
    170, #peso
    70, #peso
    40 #idade
]
#vetor com quatro dimensões
grades =[
    95, #nota1
    80, #nota2
    75, #nota3
    62 #nota4
]
```

Listas em Python não têm definidas por padrão as operações comuns a vetores. Por isso, iremos escrever funções que as realizam.

Passo 1.2 (Soma de dois vetores) A soma de dois vetores tem como resultado um terceiro vetor. Os vetores são somados componente a componente: resultante[i] = a[i] + b[i] (i = 0, …, n-1).

Nota: Dois vetores com números de componentes diferentes não podem ser somados.

A Listagem 1.2.1 mostra uma função que faz a soma de dois vetores, devolvendo o vetor resultante ao final.

Listagem 1.2.1

```
def vector_add (v, w):
    #zip devolve uma lista de tuplas
    return [v_i + w_i for v_i, w_i in zip (v, w)]
```

A Listagem 1.2.2 mostra um exemplo de uso da função vector_add.

Listagem 1.2.2

```
def vector_add_test():
    v = [1, 7]
    w = [5, 4]
    r = vector_add(v, w)
    print (r)
```

Passo 1.3 (Subtração de dois vetores) A operação de subtração é análoga à de soma. Veja a Listagem 1.3.1.

Listagem 1.3.1

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

A Listagem 1.3.2 mostra uma função de teste para a função vector_subtract.

Listagem 1.3.2

```
def vector_subtract_test ():
    v = [5, 4]
    w = [2, 1]
    r = vector_subtract(v, w)
    print (r)
```

Passo 1.4 (Soma de uma lista de vetores) Também pode ser útil somar uma lista de vetores, obtendo como resultado um vetor cuja primeira componente seja a soma de todas as primeiras componentes, a segunda componente seja a soma de todas as segundas componentes e assim por diante. O código da Listagem 1.4.1 implementa uma função que faz essa soma. A Listagem 1.4.2 a coloca em teste.

Listagem 1.4.2

```
def vector_sum (vectors):
    result = vectors[0]
    for vector in vectors[1:]:
        result = vector_add(result, vector)
    return result
```

Listagem 1.4.3

```
def vector_sum_test ():
    v1 = [1, 2, 5, 6]
    v2 = [4, 3, 2, 1]
    v3 = [1, 1, 1, 1]
    r = vector_sum([v1, v2, v3])
    print (r)
```

Passo 1.5 (Multiplicação de vetor por escalar) Um escalar é um número simples, que representa uma grandeza não vetorial. Por exemplo, o número 5 é um escalar. Uma operação importante é a multiplicação de vetor por escalar, que tem como resultado um r tal que

r[i] = v[i] * escalar (i = 0, …, n -1).

Ou seja, para cada posição i do vetor, simplesmente fazemos uma multiplicação pelo escalar. A função da Listagem 1.5.1 implementa a multiplicação de um vetor e um escalar, ambos recebidos por parâmetro.

Listagem 1.5.1

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

A Listagem 1.5.2 exibe um teste para a multiplicação de vetor por escalar.

Listagem 1.5.2

```
def scalar_multiply_test ():
    c = 5
    v = [4, 1, 2, 5]
    r = scalar_multiply(c, v)
    print (r)
```

Passo 1.6 (Média de vetores) As funções vector_sum e scalar_multiply viabilizam o cálculo da média de vetores. Para uma coleção de vetores v1, v2, vn, o vetor resultante r é definido como

r[i] = (v1[i] + v2[i] + … + vn[i])/n

Veja a função da Listagem 1.6.1 e seu teste na Listagem 1.6.2.

Listagem 1.6.1

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

Listagem 1.6.2

```
def vector_mean_test ():
    v1 = [1, 2, 5, 6]
    v2 = [4, 3, 2, 1]
    v3 = [1, 1, 1, 1]
    r = vector_mean([v1, v2, v3])
    print(r)
```

Passo 1.7 (Produto escalar (em inglês: dot product) O produto escalar (não confundir com multiplicação de vetor por escalar) consiste na multiplicação entre dois vetores e tem como resultado um escalar (daí o nome). Os dois vetores são multiplicação componente a componente e cada resultado é somado. Formalmente:

$$escalar = \sum_i v[i] + w[i]$$

A função da Listagem 1.7.1 implementa o produto escalar entre dois vetores. Veja um teste para ela na Listagem 1.7.2.

Listagem 1.7.1

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

Listagem 1.7.2

```
def dot_test ():
    v = [1, 2]
    w = [3, 4]
    r = dot (v, w)
    print (r)
```

Passo 1.8 (Soma dos quadrados) A soma dos quadrados de um vetor pode ser facilmente calculada utilizando a função dot. Veja as listagens 1.8.1 (implementação) e 1.8.2 (teste).

Listagem 1.8.1

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

Listagem 1.8.2

```
def sum_of_squares_test():
    v = [1, 2, 3]
    r = sum_of_squares (v)
    print (r)
```

Passo 1.9 (Magnitude) Quando elevamos as componentes de um vetor ao quadrado, alteramos sua unidade de medida. Extraindo a raiz quadrada do resultado voltamos à unidade original. O resultado obtido é chamado de magnitude do vetor. Veja as listagens 1.9.1 (implementação) e 1.9.2 (teste).

Listagem 1.9.1

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

Listagem 1.9.2

```
def magnitude_test ():
    v = [1, 2, 3]
    r = magnitude(v)
    print(r)
```

Passo 1.10 (Distância Euclidiana) Uma pergunta natural envolve a similaridade de observações ou amostras representadas como vetores. Diversas formas de cálculos são possíveis. Uma muito comum é a distância Euclidiana. Veja a sua definição:

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

Veja sua implementação na Listagem 1.10.1. A Listagem 1.10.2 mostra uma implementação alternativa que usa funções que definimos anteriormente.

Listagem 1.10.1

```
def squared_distance (v, w):
    return sum_of_squares((vector_subtract(v, w)))
def distance (v, w):
    return math.sqrt(squared_distance(v, w))
```

Listagem 1.10.2

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

A Listagem 1.10.3 mostra o quão “similares” (ou o quão distantes estão) são alguns usuários ficticios, considerando idade, peso e altura. Intuitivamente, quanto menor o resultado obtido, mais similares são os usuários.

Listagem 1.10.3

```
def distance_test ():
    #idade, peso e altura, nessa ordem
    u1 = [27, 80, 180]
    u2 = [58, 100, 198]
    u3 = [29, 79, 179]
    print (f'u1 vs u2: {distance(u1, u2)}')
    print (f'u1 vs u3: {distance(u1, u3)}')
    print(f'u2 vs u3: {distance(u2, u3)}')
```

Passo 1.11 (Covariância) Suspeita-se que a quantidade de tempo gasto na rede social por um usuário está relacionada com o número de amigos que ele possui. As estruturas de dados da Listagem 1.11.1 lhe foram entregues. A primeira mostra a quantidade de amigos que cada usuário tem. A segunda, a quantidade de minutos passados em um dia por cada um deles.

Listagem 1.11.1

```
def qtde_amigos_minutos_passados ():
    #primeira lista: número de amigos
    #segunda lista: qtde de minutos passados por dia (em média)
    return ([1, 10, 50, 2, 150], [5, 200, 350, 17, 1])
```

Se por um lado a **variância** calcula como uma **única variável** desvia de sua média, a **covariância** verifica como **duas variáveis** variam em conjunto de suas médias. Para calcular a covariância entre dois vetores, precisamos da função da Listagem 1.11.2. Ela calcula um vetor cujos componentes são as diferenças dos componentes do vetor original em relação à sua média.

Listagem 1.11.2

```
def variance (v):
    mean = sum(v) / len(v)
    print (mean)
    return [v_i - mean for v_i in v]
```

A função da Listagem 1.11.3 mostra a um teste para a variância.

Listagem 1.11.3

```
def variance_test():
    v = [1, 2, 3]
    r = variance(v)
    print (r)
```

A covariância é implementada pela função da Listagem 1.11.4.

Listagem 1.11.4

```
def covariance (x, y):
    n = len (x)
    return dot(variance(x), variance(y)) / (n - 1)
```

Lembre-se de que a função **dot** resume os produtos dos pares correspondentes dos elementos. Para uma determinada coordenada i, se é verdade que

$$x[i] > mean(x) \text{ e } y[i] > mean(y) \text{ ou } x[i] < mean(x) \text{ e } y[i] < mean(y)$$

então um número positivo entra para a soma calculada por **dot**. Por outro lado, quando

$$x[i] > mean(x) \text{ e } y[i] < mean(y) \text{ ou } y[i] > mean(y) \text{ e } x[i] < mean(x)$$

um número negativo entra para a soma calculada por **dot.**

Note que
- uma covariância **positiva** “**grande**” indica que x tende a ser grande quando y é grande e pequeno quando y é pequeno. Ou seja, x e y são proporcionais.
- uma covariância **negativa “grande”** indica que x tende a ser grande quando y é pequeno e pequeno quando y é grande. Ou seja, x e y são inversamente proporcionais.
- uma covariãncia igual ou muito próxima a zero indica que x e y não têm relação.
Veja as tabelas 1.11.1, 1.11.2 e 1.11.3 para exemplos ilustrativos.

Tabela 1.11.1 – Covariância positiva
x e y são proporcionais

x	**y**	**mean(x)**	**mean(y)**	**(x[i] - mean(x)) x y[i] - mean(y))**
3	1	6	4	-3 x -3 = 9
12	7			6 x 3 = 18
3	4			- 3 x 0 = 0
Covariância total: (9 + 18) / (3 - 1) = 27 / 2 = 13.5				

Tabela 1.11.2 – Covariância negativa
x e y são inversamente proporcionais

x	**y**	**mean(x)**	**mean(y)**	**(x[i] - mean(x)) x y[i] - mean(y))**
-1	8	6	4	-7 x 4 = -28
8	2			2 x -2 = -4
11	2			5 x -2 = -10
Covariãncia total: (-28 + -4 + -10) / (3 - 1) = - 42 / 2 = -21				

Tabela 1.11.3 - Covariância próxima de zero
x e y são inversamente proporcionais

x	**y**	**mean(x)**	**mean(y)**	**(x[i] - mean(x)) x y[i] - mean(y))**
8	2	6	4	2 x -2 = -4
6	6			0 x 2 = 0
4	4			-2 x 0 = 0
Covariância total: (-4)/(3 - 1) = -4/2 = -2				

A Listagem 1.11.5 mostra a função de covariância operando sobre os valores das tabelas de 1.11.1 a 1.11.3.

Listagem 1.11.5

```
def covariance_tests():
    #teste 1
    x = [3, 12, 3]
    y = [1, 7, 4]
    print(f'covariance: {covariance(x, y)}')
    #teste 2
    x = [-1, 8, 11]
    y = [8, 2, 2]
    print (f'covariance: {covariance(x, y)}')
    #teste 3
    x = [8, 6, 4]
    y = [2, 6, 4]
    print (f'covariance: {covariance(x, y)}')
```

Passo 1.12 (Correlação) A covariância tem, ao menos, duas características indesejáveis:

- A unidade dos dados pode ser de difícil interpretação. Por exemplo, estamos lidando com multiplicações entre minutos por dia e números de amigos. O valor resultante de 5 amigos por minutos por dia está em que unidade?

- Se todos os usuários da lista tiverem o dobro de amigos (com o mesmo número de minutos) a covariância seria muito maior mas não necessariamente indicaria uma relação mais forte entre as variáveis.

Por essas razões é comum considerar-se uma medida chamada correlação. Ela é o resultado da divisão da covariância pelo desvio padrão de cada vetor envolvido. O resultado está sempre entre -1 e 1:

-1 indica anticorrelação perfeita
1 indica correlação perfeita
0 indica inexistência de correlação
valores muito próximos de 0 indicam correlações (quando positivos) ou anticorrelações (quando negativos) muito fracas.

A função da Listagem 1.12.1 mostra a implementação da da correlação.

Listagem 1.12.1

```
def correlation (x, y):
    desvio_padrao_x = math.sqrt(sum_of_squares(variance(x)) / (len(x) - 1))
    desvio_padrao_y = math.sqrt(sum_of_squares(variance(y)) / (len(y) - 1))
    if desvio_padrao_x > 0 and desvio_padrao_y > 0:
        return covariance(x, y) / desvio_padrao_y / desvio_padrao_x
    else:
        return 0
```

A Listagem 1.12.2 mostra testes de correlação com os dados das tabelas de 1.11.1 a 1.11.3.

Listagem 1.12.2

```
def correlation_tests_with_table_data():
    # teste 1
    x = [3, 12, 3]
    y = [1, 7, 4]
    print(f'correlation: {correlation(x, y)}')
    # teste 2
    x = [-1, 8, 11]
    y = [8, 2, 2]
    print(f'correlation: {correlation(x, y)}')
    # teste 3
    x = [8, 6, 4]
    y = [2, 6, 4]
    print(f'correlation: {correlation(x, y)}')
```

Passo 1.13 (Outlier (valor discrepante) Vamos verificar a correlação entre as listas pertencentes à tupla devolvida pela função **qtde_amigos_minutos_passados**. Note que, intuitivamente, há correlação entre cada coordenada, a menos da última que parece incondizente com o esperado. Veja a Listagem 1.13.1.

Listagem 1.13.1

```
def correlation_test_with_outlier ():
    data = qtde_amigos_minutos_passados()
    resultado = correlation(data[0], data[1])
    print (resultado)
```

Depois de inspecionar os dados, você percebeu que o valor discrepante se referia a uma conta de teste desativada. Após uma limpeza nos dados realize o teste da Listagem 1.13.2 e veja o resultado.

Listagem 1.13.2

```
def qtde_amigos_minutos_passados_sem_outlier ():
    #primeira lista: número de amigos
    #segunda lista: qtde de minutos passados por dia (em média)
    return ([1, 10, 50, 2], [5, 200, 350, 17])
def correlation_test_removing_outlier ():
    data = qtde_amigos_minutos_passados_sem_outlier()
    resultado = correlation(data[0], data[1])
    print (resultado)
```

Nota: Utilizamos listas para ilustrar didaticamente algumas das principais operações que nos serão de interesse. Do ponto de vista de desempenho, essa é uma péssima ideia. Na prática, é de interesse usar a biblioteca NumPy, que tem estruturas de dados e funções apropriadas para isso.

Bibliografia

GRUS, J. **Data Science from Scratch: First Principles with Python.** 1st ed. O'Reilly Media, 2015.