El algoritmo estándar para multiplicar matrices por vectores (un tema del curso "Álgebra Lineal Numérica")

Egor Maximenko

http://www.egormaximenko.com

Instituto Politécnico Nacional
Escuela Superior de Física y Matemáticas
Ciudad de México

1 de marzo de 2021

Objetivo:

- programar el algoritmo estándar de multiplicación de matrices por vectores;
- analizar la complejidad del algoritmo;
- medir el tiempo de ejecución de los programas.

Prerrequisitos:

- programación con funciones, arreglos y ciclos;
- construcción de matrices y vectores pseudoaleatorios;
- ciclos “for” anidados;
- el algoritmo para calcular el producto punto de dos vectores.

Problema de programación

Escribir una función de dos argumentos A, b que calcule y devuelva el producto Ab, donde A es una matriz y b es un vector.

> Entrada: una matriz A y un vector b.
>
> Propiedades de la entrada: se supone que el número de columnas de A coincide con la longitud de b.
>
> Salida: el producto Ab.

Vamos a resolver este problema de varias maneras equivalentes.
Primero, recordemos la fórmula para el producto Ab.

Ejemplo (repaso)

Sean $A \in \mathcal{M}_{2\times 3}(\mathbb{R})$, $b \in \mathbb{R}^3$. Denotemos Ab por c:

$$\begin{bmatrix} c_1 \\ c_2 \end{bmatrix} = \begin{bmatrix} A_{1,1} & A_{1,2} & A_{1,3} \\ A_{2,1} & A_{2,2} & A_{2,3} \end{bmatrix} \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix}.$$

Ejemplo (repaso)

Sean $A \in \mathcal{M}_{2\times 3}(\mathbb{R})$, $b \in \mathbb{R}^3$. Denotemos Ab por c:

$$\begin{bmatrix} c_1 \\ c_2 \end{bmatrix} = \begin{bmatrix} A_{1,1} & A_{1,2} & A_{1,3} \\ A_{2,1} & A_{2,2} & A_{2,3} \end{bmatrix} \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix}.$$

Cada componente del vector c es el producto punto de un renglón de A por la columna b:

Ejemplo (repaso)

Sean $A \in \mathcal{M}_{2\times 3}(\mathbb{R})$, $b \in \mathbb{R}^3$. Denotemos Ab por c:

$$\begin{bmatrix} c_1 \\ c_2 \end{bmatrix} = \begin{bmatrix} A_{1,1} & A_{1,2} & A_{1,3} \\ A_{2,1} & A_{2,2} & A_{2,3} \end{bmatrix} \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix}.$$

Cada componente del vector c es el producto punto de un renglón de A por la columna b:

$$c_1 = A_{1,*}b = A_{1,1}b_1 + A_{1,2}b_2 + A_{1,3}b_3 = \sum_{k=1}^{3} A_{1,k}b_k;$$

Ejemplo (repaso)

Sean $A \in \mathcal{M}_{2\times 3}(\mathbb{R})$, $b \in \mathbb{R}^3$. Denotemos Ab por c:

$$\begin{bmatrix} c_1 \\ c_2 \end{bmatrix} = \begin{bmatrix} A_{1,1} & A_{1,2} & A_{1,3} \\ A_{2,1} & A_{2,2} & A_{2,3} \end{bmatrix} \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix}.$$

Cada componente del vector c es el producto punto de un renglón de A por la columna b:

$$c_1 = A_{1,*}b = A_{1,1}b_1 + A_{1,2}b_2 + A_{1,3}b_3 = \sum_{k=1}^{3} A_{1,k}b_k;$$

$$c_2 = A_{2,*}b = A_{2,1}b_1 + A_{2,2}b_2 + A_{2,3}b_3 = \sum_{k=1}^{3} A_{2,k}b_k.$$

La fórmula general (repaso)

Sea $A \in \mathcal{M}_{m\times n}(\mathbb{R})$ y sea $b \in \mathbb{R}^n$. Denotemos Ab por c.
Entonces $c \in \mathbb{R}^m$. La componente c_j, para cada j en $\{1, \ldots, m\}$, es

$$(Ab)_j = A_{j,*}b = \sum_{j=1}^{n} A_{j,k}b_k.$$

La fórmula general (repaso)

Sea $A \in \mathcal{M}_{m\times n}(\mathbb{R})$ y sea $b \in \mathbb{R}^n$. Denotemos Ab por c.
Entonces $c \in \mathbb{R}^m$. La componente c_j, para cada j en $\{1, \ldots, m\}$, es

$$(Ab)_j = A_{j,*}b = \sum_{j=1}^{n} A_{j,k}b_k.$$

El producto $A_{j,*}b$ se puede considerar como el producto punto de dos vectores.

La fórmula general (repaso)

Sea $A \in \mathcal{M}_{m\times n}(\mathbb{R})$ y sea $b \in \mathbb{R}^n$. Denotemos Ab por c.
Entonces $c \in \mathbb{R}^m$. La componente c_j, para cada j en $\{1, \ldots, m\}$, es

$$(Ab)_j = A_{j,*}b = \sum_{j=1}^{n} A_{j,k}b_k.$$

El producto $A_{j,*}b$ se puede considerar como el producto punto de dos vectores.

(En caso de matrices y vectores complejos, la situación es un poco diferente.)

La función que calcula el producto punto (repaso)

```
def dotproduct(x, y):
    n = len(x)
    s = 0
    for k in range(n):
        s += x[k] * y[k]
    return s
```

Esta función tiene un ciclo. Dentro del ciclo el número de operaciones es constante.

Se ejecutan n iteraciones del ciclo.

El número total de operaciones es aproximadamente Cn^1, donde C es una constante.

Decimos que la función `dotproduct` es de nivel 1.

Solución que utiliza nuestro producto punto

```
def mulmatvec1r(A, b):
    m = A.shape[0] # el numero de renglones de A
    c = zeros(m)
    for j in range(m):
        c[j] = dotproduct(A[j, :], b)
    return c
```

El número 1 en el título de la función significa que usamos una función de nivel 1.

La letra *r* significa que trabajamos con la matriz A por renglones.

Solución que utiliza el producto punto de la librería

Podemos sustituir nuestra función `dotproduct` por la función `dot` del paquete `numpy`.

```
def mulmatvec1r_dot(A, b):
    m = A.shape[0]
    c = zeros(m)
    for j in range(m):
        c[j] = dot(A[j, :], b)
    return c
```

Solución con dos ciclos anidados, versión previa

En vez de llamar la función `dotproduct`, sustituimos su código.

Por supuesto, sustituimos `x[k]` por `A[j, k]`, `y[k]` por `b[k]`.

```
def mulmatvec0r_temporary(A, b):
    (m, n) = A.shape
    c = zeros(m)
    for j in range(m):
        s = 0
        for k in range(n):
            s += A[j, k] * b[k]
        c[j] = s
    return c
```

Solución con dos ciclos anidados, versión final

Podemos acumular la suma en `c[j]`, sin usar la variable auxiliar `s`.

```
def mulmatvec0r(A, b):
    (m, n) = A.shape
    c = zeros(m)
    for j in range(m):
        for k in range(n):
            c[j] += A[j, k] * b[k]
    return c
```

El número 0 en el título de la función significa que no utilizamos ningunas funciones auxiliares; trabajamos solamente con operaciones de nivel 0.

Análisis de la complejidad

Todas las versiones del algoritmo `mulmatvec` tienen la misma complejidad computacional.

Vamos a contar el número de las operaciones de multiplicación (de números reales) y el número de las operaciones de adición (de números reales).

Cada componente de la matriz A participa en una operación de multiplicación.
El número de las multiplicaciones (de números) es mn,
y el número de las adiciones también es mn.

El número de incrementos de los números enteros en los ciclos también es mn.

La complejidad total es de orden $\Theta(mn)$.

Análisis de complejidad

En particular, si $m = n$, entonces el número de las multiplicaciones es

$$n^2,$$

y el orden de la complejidad es $\Theta(n^2)$.

En nuestro contexto (algoritmos con matrices) es cómodo decir que el algoritmo es de nivel 2.

Pruebas de la constistencia para datos pequeños

Calculamos Ab usando varias versiones del algoritmo.

```
def test0_mulmatvec():
    A = array([[4, 0.5, -1], [2, 1, -1.5]])
    b = array([0.5, 2, 4])
    c0 = A @ b
    c1 = mulmatvec1r(A, b)
    c2 = mulmatvec1r_dot(A, b)
    c3 = mulmatvec0r(A, b)
    return column_stack([c0, c1, c2, c3]) # juntar en una matriz


print(test0_mulmatvec())
```

Pruebas de la consistencia para tamaños arbitrarios

```
def maxerrorvec(vecs):
    return max(linalg.norm(u - vecs[0]) for u in vecs)

def test1_mulmatvec(m, n):
    A = random.rand(m, n); b = random.rand(n)
    c0 = A @ b
    c1 = mulmatvec1r(A, b)
    c2 = mulmatvec1r_dot(A, b)
    c3 = mulmatvec0r(A, b)
    return maxerrorvec([c0, c1, c2, c3])

print(test1_mulmatvec(5, 3))
```

Pruebas de la velocidad

```
import time

def test2_mulmatvec(n):
    A = random.rand(n, n); b = random.rand(n)
    t0 = time.time(); c0 = A @ b
    t1 = time.time(); c1 = mulmatvec1r(A, b)
    t2 = time.time(); c2 = mulmatvec1r_dot(A, b)
    t3 = time.time(); c3 = mulmatvec0r(A, b)
    t4 = time.time()
    return [t1 - t0, t2 - t1, t3 - t2, t4 - t3]

print(test2_mulmatvec(1000))
```

Resultados de las pruebas de velocidad

	$n = 1024$	$n = 2048$	$n = 4096$
`A @ b`	$5.3 \cdot 10^{-4}$	$2.1 \cdot 10^{-3}$	$8.4 \cdot 10^{-3}$
`mulmatvec1r`	0.30	1.2	4.8
`mulmatvec1r_dot`	$1.9 \cdot 10^{-3}$	$5.8 \cdot 10^{-3}$	$2.0 \cdot 10^{-2}$
`mulmatvec0r`	0.43	1.8	7.0

Resultados de las pruebas de velocidad

	$n = 1024$	$n = 2048$	$n = 4096$
`A @ b`	$5.3 \cdot 10^{-4}$	$2.1 \cdot 10^{-3}$	$8.4 \cdot 10^{-3}$
`mulmatvec1r`	0.30	1.2	4.8
`mulmatvec1r_dot`	$1.9 \cdot 10^{-3}$	$5.8 \cdot 10^{-3}$	$2.0 \cdot 10^{-2}$
`mulmatvec0r`	0.43	1.8	7.0

Conclusión 1: las funciones de la librería son mucho más rápidas que las nuestras, porque las librerías están escritas en lenguajes compilables (C, Fortran), mientras que Python es un lenguaje interpretado.

Conclusión 2: al multiplicar n por 2, el tiempo se multiplica por 4.