INSTITUTO TECNOLÓGICO DE BUENOS AIRES

93.54 - MÉTODOS NUMÉRICOS

Trabajo Práctico 2: Cuadrados Mínimos

Grupo 8

CRESPO, Elisabet del Pilar	59098
MARTORELL, Ariel	56209
VEKSELMAN, Alan	59378

Profesores

FIERENS, Pablo Ignacio
GRISALES CAMPEÓN, Juan Pablo

Fecha de entrega: 14/04/2021

1. Introducción

En este trabajo se propuso implementar un algoritmo que resuelva el problema de cuadrados mínimos con la forma siguiente:

$$A \cdot \vec{x} = \vec{b} \tag{1}$$

Con $A \in R^{mxn}$, $m \geq n$, y $b \in R^{mx1}$.

La idea es minimizar el error cuadrático, es decir, minimizar la siguiente expresión:

$$||A.\vec{x} - \vec{b}||_2 \tag{2}$$

2. Cuadrados mínimos con factorización de Cholesky

El algoritmo de Cholesky es una serie de pasos para convertir a una matriz A en un producto de dos matrices, una traspuesta de la otra, y que a su vez sean triangular inferior y superior, respectivamente. Esta factorización se basa en la hipótesis de que la matriz A es positiva semidefinida, es decir, que todos sus autovalores son reales, distintos y mayores a cero. Si se da esta condición, entonces se podrá expresar:

$$A = G \cdot G^T \tag{3}$$

Donde G es una matriz triangular inferior, es decir, todos los componentes que estén por encima de su diagonal principal serán iguales a cero. De esta manera, partiendo del problema de 1, ahora se puede plantear al sistema de la siguiente forma, multiplicando previamente por A^T de ambos lados:

$$G \cdot G^T \cdot \vec{x} = A^T \cdot \vec{b} \tag{4}$$

Puede notarse que ahora el sistema se puede resolver como dos sistemas triangulares, siendo éstos:

$$G \cdot \vec{w} = A^T \cdot \vec{b} \tag{5}$$

$$G^T \cdot \vec{x} = \vec{w} \tag{6}$$

Este algoritmo presenta una clara ventaja frente a otros similares para resolver problemas de error cuadrático mínimo, y es que, al ser $A^T \cdot A = G \cdot G^T$, entonces la factorización de $A^T \cdot A$ no requiere el almacenamiento de dos matrices, sino sólo una. Además, su complejidad computacional es menor que la de otros algoritmos como LU, QR o SVD, ya que es del orden de $\frac{n^3}{3}$.

La principal desventaja de este algoritmo es que su funcionamiento requiere, en principio, la multiplicación previa por $A^T \cdot A$. Ésta, además, debe ser una matriz semidefinida positiva, lo cual no es siempre el caso. Por último, es numéricamente inestable, lo cual causa errores más relevantes que otros algoritmos a medida que crece el número de condición de A.

3. Implementación del algoritmo

Para obtener una resolución de cuadrados mínimos utilizando la descomposición Cholesky, lo primero que se hizo fue obtener la matriz $B = A^T \cdot A$, y luego se procedió a obtener la matriz triangular inferior G aplicando el algoritmo de Cholesky.

Primero se obtuvo el tamaño de las filas y columnas de la matriz B, representada en este caso con un arreglo de *NumPy*. Utilizando la propiedad *.shape* de dicha clase se obtuvo el tamaño de las filas y columnas de la matriz, y se creó un arreglo auxiliar G del mismo tamaño que la matriz B. En caso de no ser cuadrada, el algoritmo causará una excepción, avisando de tal hecho. Al comprobar que la matriz es cuadrada, se procede a buscar los valores de cada elemento de la matriz G. Para esto, se hace un ciclo *for* que va desde 0 hasta el tamaño de las filas (se puede usar el tamaño de las filas o de las columnas indistintamente, ya que es una matriz cuadrada), en donde se calculan los elementos de la diagonal de la siguiente forma:

$$G_{ii} = \sqrt{B_{ii} - \sum_{k=0}^{i-2} G_{ik}^2} \tag{7}$$

Los elementos que no perteneces a la diagonal se obtienen utilizando la siguiente fórmula:

$$G_{ij} = \frac{B_{ij} - \sum_{k=0}^{j-2} G_{ik} * G_{jk}}{G_{jj}} \tag{8}$$

Antes de seguir con la implementación de todo el código en cuestión, se procederá a describir el funcionamiento de la función *solve_triangular(A,b,lower)*, cuyo resultado devuelve un vector con todas las soluciones a la ecuación $A \cdot \vec{x} = \vec{b}$, donde A es una matriz triangular inferior o superior. Esta función tiene como parámetros de entrada la matriz A, el vector $\vec{b}$ y un *bool* que indica si dicha matriz es triangular superior o inferior.

Para obtener dicho vector se implementó un ciclo *for* que va desde 0 hasta el alto (o ancho) de la matriz, utilizando la siguiente fórmula en el caso que sea una matriz triangular superior:

$$X_n = \frac{b_n - \sum_{i=N-n}^{N-1} A_{ni} * x_i}{A_{nn}}, \tag{9}$$

y en caso de ser una matriz triangular inferior:

$$X_n = \frac{b_n - \sum_{i=0}^{n-1} A_{ni} * x_i}{A_{nn}}, \tag{10}$$

Para finalizar, en la función *leastsq* primero se chequea que el parámetro b de entrada sea un vector columna y también se comprueba que la cantidad de filas de A sea menor a su cantidad de columnas, ya que, si esto pasara, se tendrían más incógnitas que ecuaciones, y no se podría aplicar Cholesky. Una vez pasados estos controles, se procede a generar la matriz G, utilizando la función *cholesky*, siendo el parámetro de entrada $A^T \cdot A$. Luego se llama a la función *solve_triangular(G,* $A^T \cdot b$*, lower=True)*, se lo almacena en la variable w y por ultimo se devuelve como resultado *solve_triangular(G,* $A^T \cdot w$*, lower=False)*, con el que se obtiene la solución al sistema propuesto con el método de cuadrados mínimos.

4. Verificación del algoritmo

Para verificar el algoritmo implementado se procedió a realizar una función *test*, con la que se realizarán 150 pruebas para comparar el algoritmo propio con el algoritmo implementado en la librería *SciPy*. Para esto se crean, en primer lugar, el alto y el ancho (h y w), que serán números enteros aleatorios entre 100 y 200, y entre 2 y 100, respectivamente. Luego se crean la matriz A, de números flotantes aleatorios entre -100 y 100, de tamaño hxw, y el vector y, de números aleatorios de punto flotante entre -100 y 100, de dimensión $hx1$. Se procede a resolver primero con la función de *SciPy*, y después con la función que se implementó, para luego comparar ambos resultados. Si alguna de estas comparaciones no es concordante, se imprimirá un mensaje que diga *Failed* y se saldrá de la función. En caso de terminar exitosamente todas las comparaciones, se imprimirá un mensaje final de *Ok!*.

5. Anexo

5.1. Cholesky

```
import numpy as np

def cholesky(A : np.array) -> np.array:
    h, w = A.shape
    G = np.zeros((w, h))

    if h == w:
        if np.all(np.linalg.eig(A)[0] > 0):
            for i in range(h):
                G[i, i] = np.sqrt(A[i, i] -  G[i, :i].dot(G[i, :i]))
                G[i+1:w, i] = (A[i, i+1:w] - G[i+1:w, :i].dot(G[i, :i])) / G[i, i]
        else:
            raise ValueError('Input error: Matrix must be positive semidefinite')
    else:
        raise ValueError('Input error: Matrix must be square')
    return G
```

5.2. Sistema triangular

```
def solve_triangular(G : np.array, y : np.array, lower = False) -> np.array:
    h, w = G.shape
    res = np.zeros((h, 1))

    if h == w:
        for i in np.arange(h)[::(-1) ** (not lower)]:
            res[i] = (y[i] - np.dot(G[i], res)) / G[i, i]

    return res
```

5.3. Cuadrados Mínimos

```
def leastsq(A : np.array, b : np.array) -> np.array:
    if len(b.shape) == 1 or not b.shape[1] == 1:
        raise ValueError('Input error: Independent terms must be a column vector')

    elif A.shape[0] < A.shape[1]:
        raise ValueError("Input error: Cholesky can't solve systems with n > m")

    G = cholesky(A.T.dot(A))
    w = solve_triangular(G, A.T.dot(b), lower=True)
    return solve_triangular(G.T, w, lower=False)
```

5.4. Verificación

```
from scipy.linalg import lstsq

def test() -> None:

    num_tests = 150

    for i in range(num_tests):
        h = np.random.randint(100, 201)
        w = np.random.randint(2, 101)

        A = -100 + 200 * np.random.rand(h, w)
        y = -100 + 200 * np.random.rand(h, 1)

        # Verificación con SciPy
        x2 = lstsq(A, y)[0]

        # Implementación
        x1 = leastsq(A, y)

        if not np.allclose(x1, x2):
            print('Failed')
            return

    print('Ok!')
```