

Trabajo en *Python*

Jose Joaquín Rodríguez García y Araceli Teruel Domenech

Universidad Politécnica de Valencia
Master en Big Data Analytics

Septiembre 2017

1. Primera tarea

Comparación de los tiempos de ejecución entre código puro de *Python* y código usando la librería *Numpy*.

1.1. Primer apartado

1.1.1. Enunciado

Ordenar una lista:

- (a) Usando bucles de *Python*.
- (b) Usando un bucle de *Python* y la función *argmin* de la librería *Numpy*.
- (c) Usando la función de ordenación de la librería *Numpy*.

1.1.2. Resolución

Usando vectores de un número mayor o igual a 100 elementos, se ve en la figura 1 que el *script* más eficiente en tiempo de ejecución es el que proporciona la librería *Numpy* como función, seguido del *script* que usa bucles de *Python* y la función *argmin* de la librería *Numpy* y por último lo es el *script* que usa solamente bucles de *Python*.

Por último, comentar que al aumentar el número de elementos de un vector a ordenar, por encima de 200, el *script* que hace uso de bucles de *Python* ve incrementado el tiempo

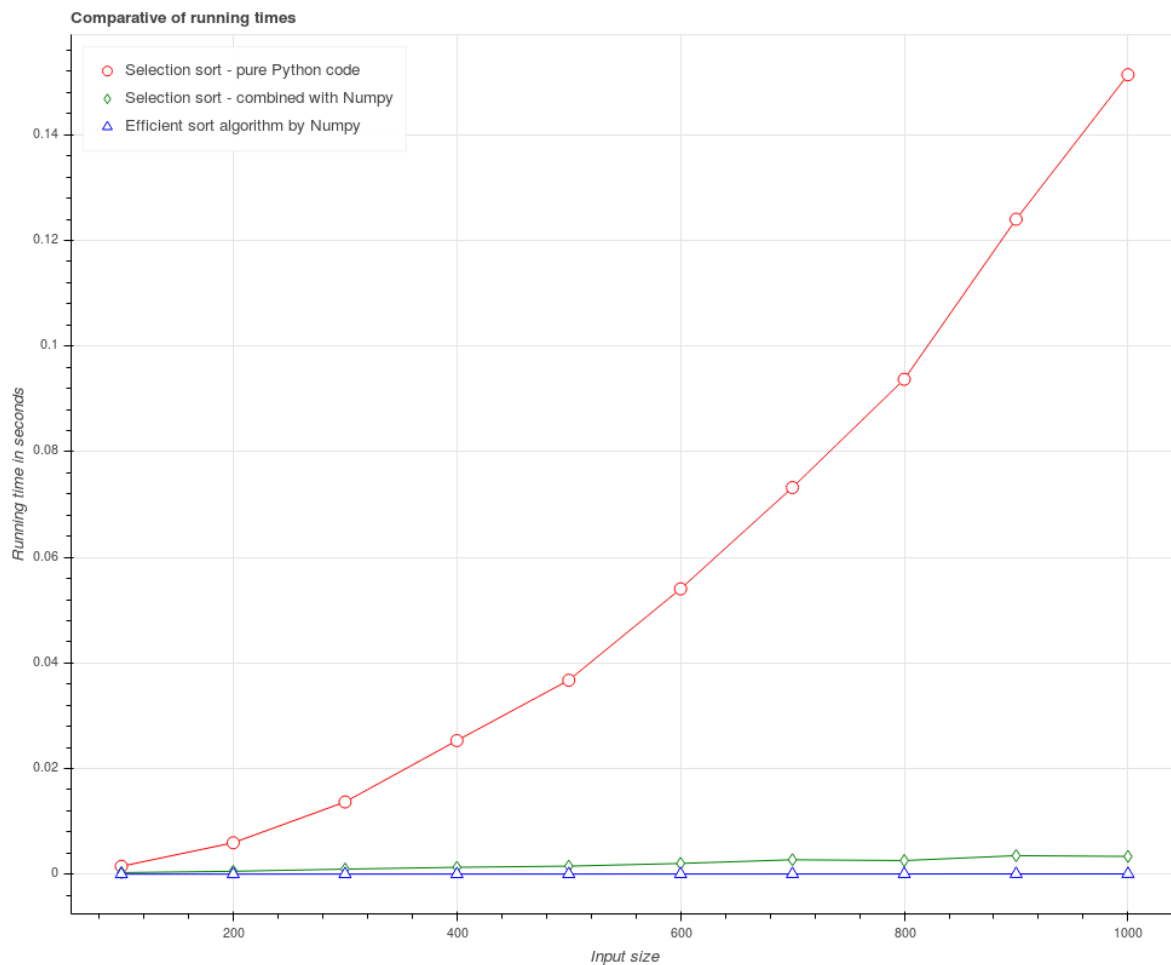


Figura 1: Comparativa de los tiempos de ejecución de los tres scripts.

que necesita para la tarea de forma exponencial. Lo cual refleja que este algoritmo es muy ineficiente en términos de tiempos de computo, para ordenar una gran cantidad de elementos.

1.2. Segundo apartado

1.2.1. Enunciado

Producto escalar entre dos vectores:

- (a) Usando listas y bucles de *Python*.
- (b) Usando bucles de *Python* y vectores de la librería *Numpy*.

(c) Usando el producto matricial de *Python* y vectores de la librería *Numpy*.

1.2.2. Resolución

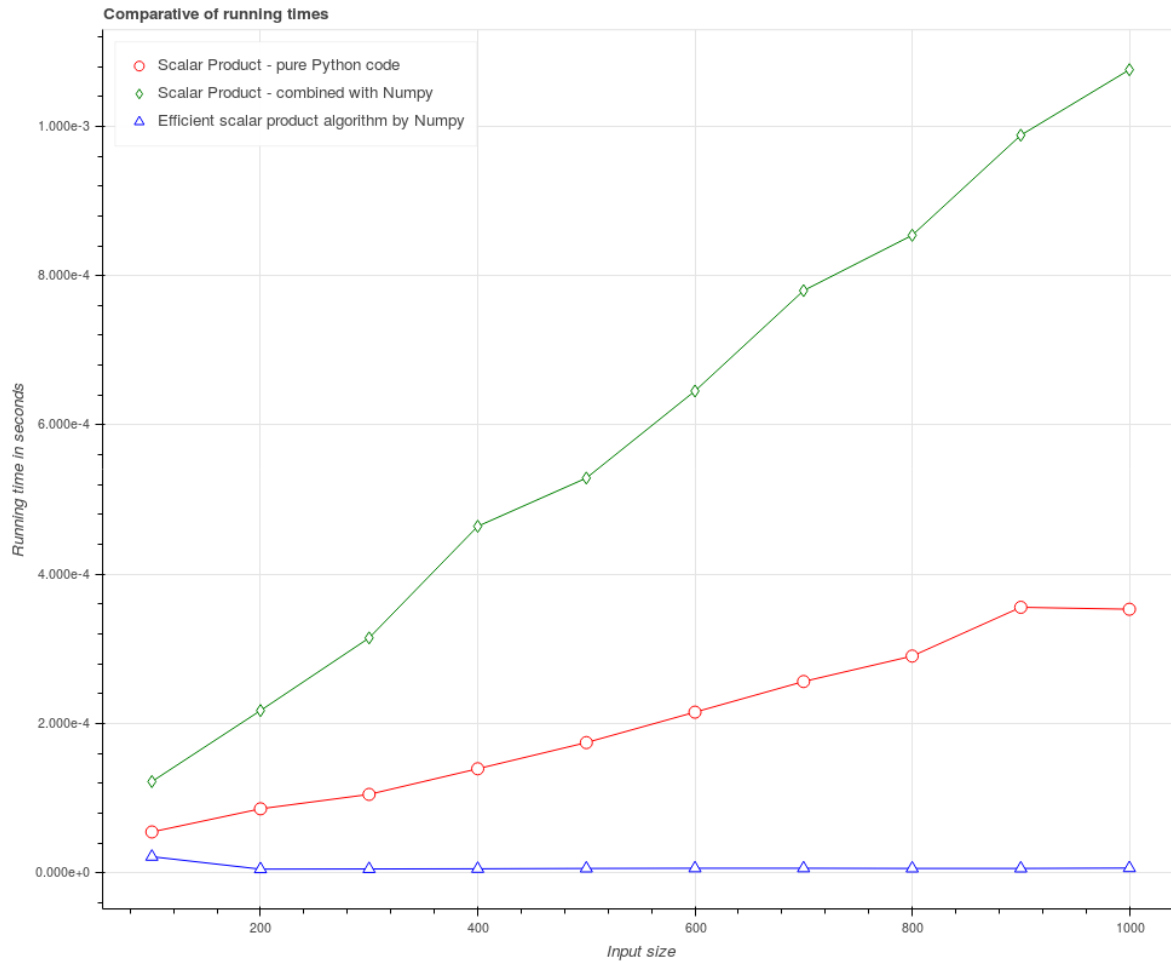


Figura 2: Comparativa de los tiempos de ejecución de los tres scripts.

Para el producto escalar de vectores con un número mayor o igual a 100 elementos vemos en la figura 2 que el *script* más eficiente en tiempo de ejecución es el que hace uso del tipo de vector de la librería *Numpy* combinado con la operación de producto matricial implementado en dicha librería, el siguiente más eficiente el *script* que hace uso del vector como una lista de *Python* y usa un bucle para recorrer los elementos de las listas y por último el menos eficiente es el que hace uso del vector como tipo de vector de la librería *Numpy* y usa un bucle *for* para recorrer los elementos de estos vectores.

Se observa que el aumento del tiempo necesario para realizar el producto escalar, a

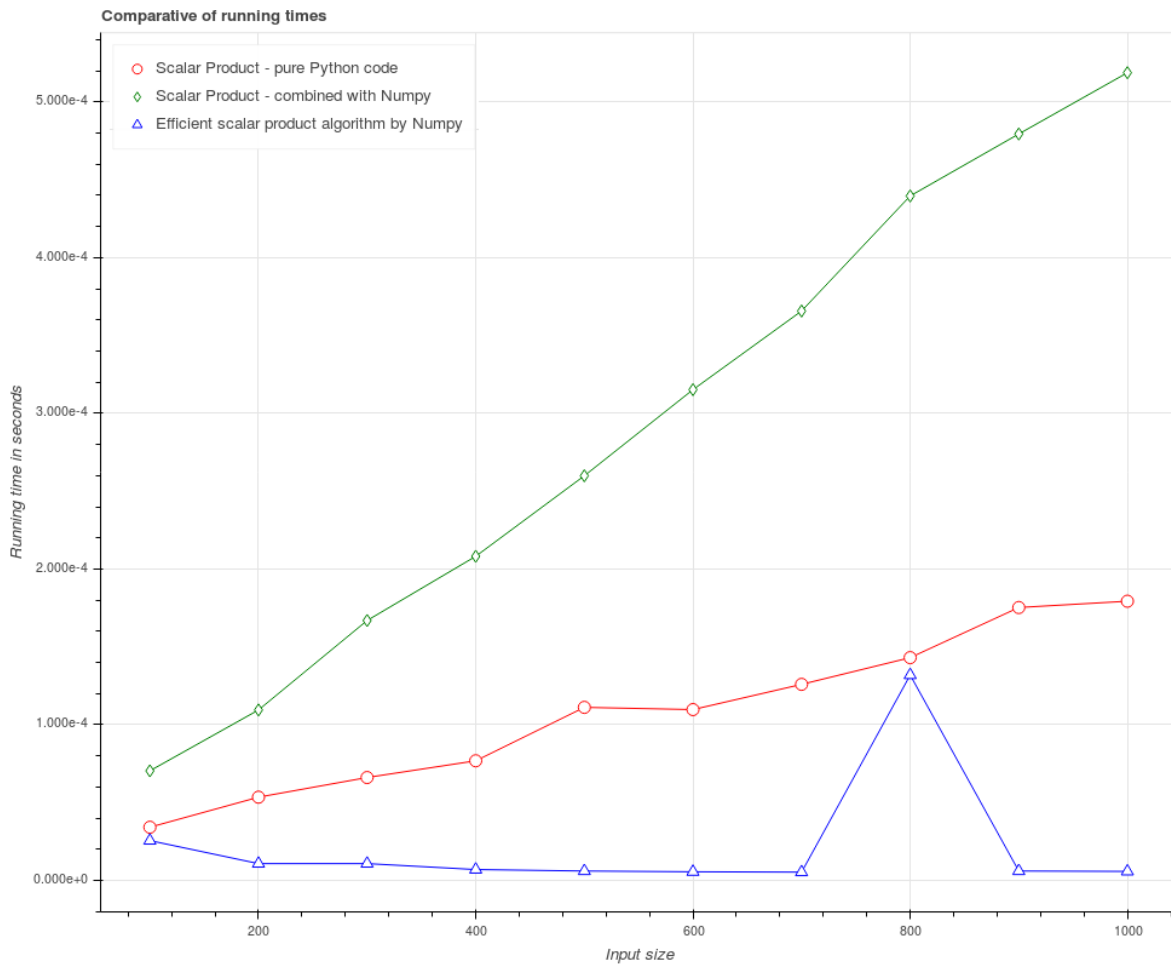


Figura 3: Pico en el tiempo de ejecución del script (c).

medida que se aumenta el número de elementos de los vectores, tanto con el *script* que usa código puro de *Python* combinado con *Numpy*, como con el *script* que usa código puro de *Python* sigue una tendencia lineal, con la diferencia de que la pendiente del primero de los *scripts* comentados es más pronunciada que la del segundo *script*. Mientras que el *script* que usa sólo código *Python* se mantiene más o menos¹ constante rozando el tiempo de cómputo 0.

¹Haciendo varias pruebas en algunos casos se ven picos aislados, veasé por ejemplo la figura 3, aunque estos picos siguen estando por debajo del tiempo que requieren los otros dos *scripts* para llevar a cabo la operación.

1.3. Tercer apartado

1.3.1. Enunciado

Producto matricial entre un vector y una matriz:

- (a) Usando listas y bucles de *Python*.
- (b) Usando bucles de *Python* con vectores y matrices de la librería *Numpy*.
- (c) Usando el producto matricial de *Python* con vectores y matrices de la librería *Numpy*.

1.3.2. Resolución

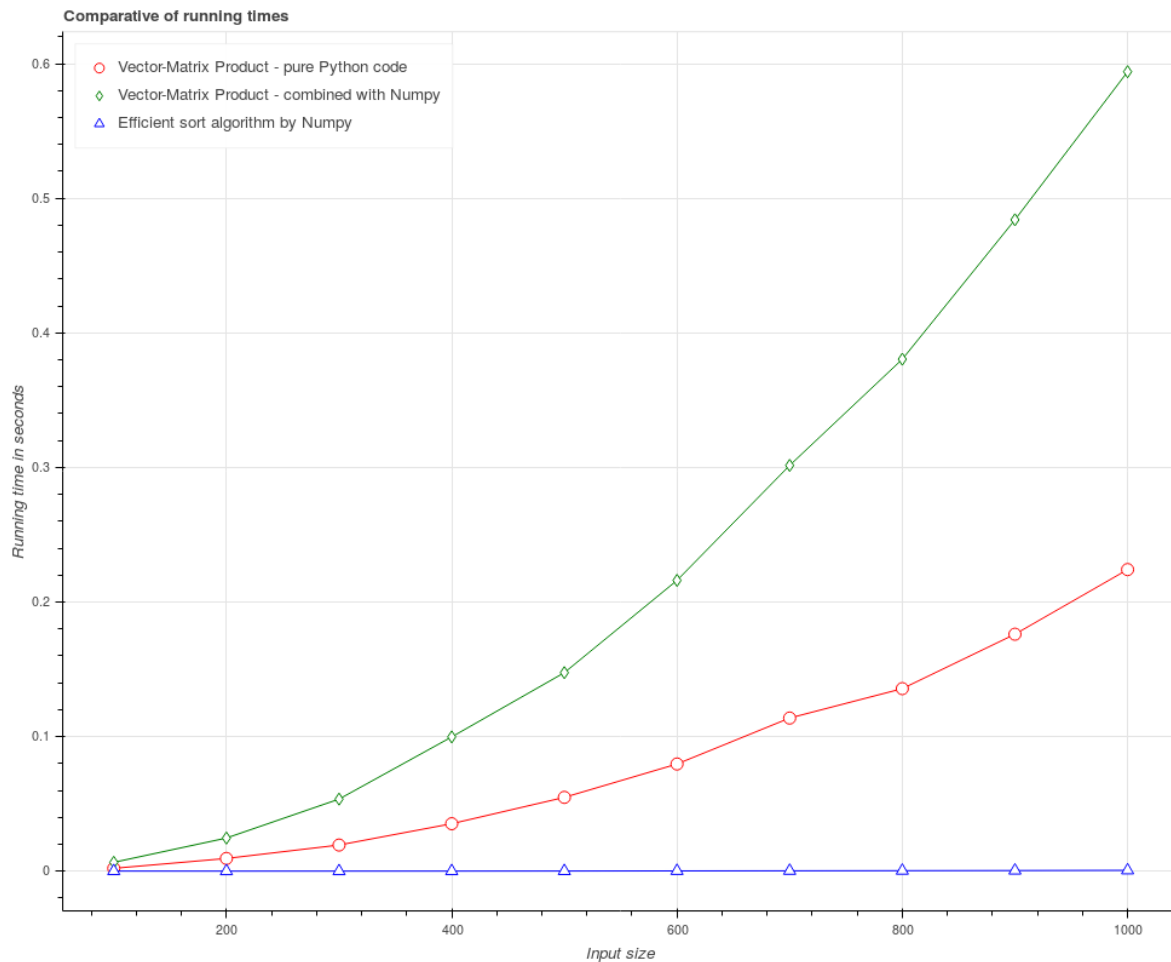


Figura 4: Comparativa de los tiempos de ejecución de los tres scripts.

Para el producto matricial de una matriz por un vector o viceversa tomando el numero n de elementos del vector como 100 o más y las dimensiones de la matriz como $n \times n$, vemos en la figura 4 que el *script* más eficiente en tiempo de ejecución es el que hace uso del tipo de vector de la librería *Numpy* combinado con la operación de su mismo producto matricial, el siguiente más eficiente el *script* que hace uso del vector y matriz como listas de *Python* y usa un bucle para recorrer los elementos de las listas y por último el menos eficiente es el que hace uso del vector y matriz como tipo de vector o matriz (arrays) de la librería *Numpy* y usa un bucle for para recorrer los elementos de estos vectores.

Se observa que el aumento del tiempo necesario para realizar el producto escalar, a medida que se aumenta el valor n de elementos del vector y dimensiones de la matriz $n \times n$, tanto con el *script* que usa código *Python* combinado con la librería *Numpy*, como con el *script* que usa código puro de *Python*, sigue una tendencia exponencial, con la diferencia de que la base del exponente de la tendencia del primero de los *scripts* comentados es mayor que la base del exponente de la tendencia del segundo, siendo la tendencia del segundo *script* comentado casi lineal. Mientras que el *script* que usa simplemente código de la librería de *Numpy* se mantiene constante rozando el tiempo de cómputo 0.

1.4. Cuarto apartado

1.4.1. Enunciado

Producto matricial entre un vector y una matriz:

- (a) Usando listas y bucles de *Python*.
- (b) Usando bucles de *Python* y matrices de la librería *Numpy*.
- (c) Usando el producto matricial de *Python* y matrices de la librería *Numpy*.

1.4.2. Resolución

Para el producto matricial de dos matrices cuadradas $n \times n$, siendo n mayor o igual a 100, vemos en la figura 5 que el *script* más eficiente en tiempo de ejecución es el que hace uso del tipo de matriz de la librería *Numpy* combinado con la operación de producto matricial de esta misma librería, el siguiente más eficiente es el *script* que trata a ambas matrices como listas de *Python* y usa un bucle para recorrer los elementos de las listas y por último el menos eficiente es el que hace uso de ambas matrices como tipo de matrices

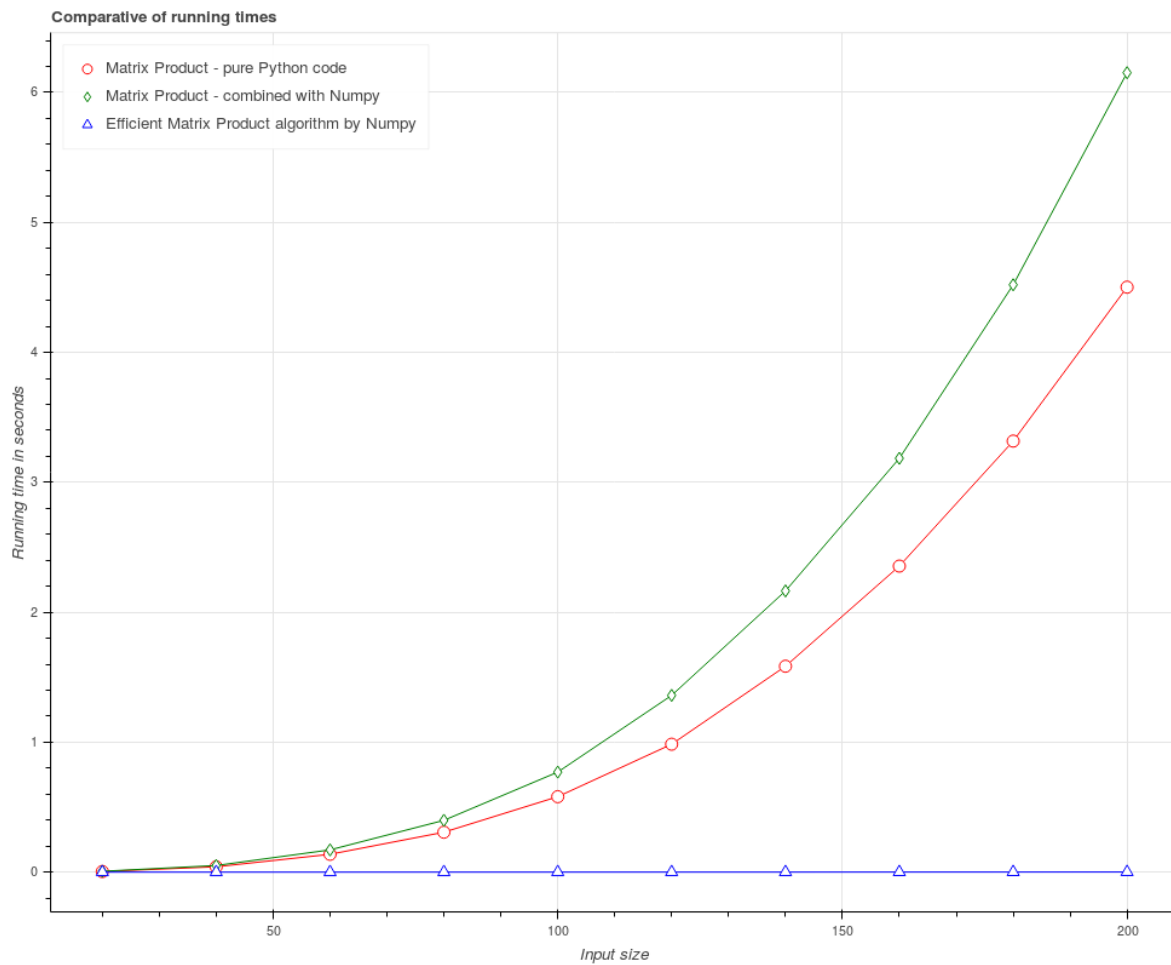


Figura 5: Comparativa de los tiempos de ejecución de los tres scripts.

de la librería *Numpy* y usa un bucle for para recorrer los elementos de estas matrices. En la Figura 6 podemos ver, que esto sigue siendo así aunque el tamaño del vector sea pequeño, aunque, al contrario que en los otros casos, la diferencia es bastante menor.

Se observa que el aumento del tiempo necesario para realizar el producto matricial, a medida que se aumenta el valor n de las dimensiones de las matrices, tanto con el *script* que combina la librería *Python* con código puro, como con el *script* que usa sólo código puro de *Python*, sigue una tendencia exponencial, con la diferencia de que la base del exponente de la tendencia del primero de los *scripts* comentados es mayor que la base del exponente de la tendencia del segundo. Mientras que el *script* que usa la librería de *Numpy* tanto para la definición de las matrices como para el producto de ellas se mantiene constante rozando el tiempo de cómputo 0.

1.5. Conclusión

Como hemos podido comprobar, siempre que usemos la librería *Numpy* para todo el cálculo será mucho más eficiente en tiempo de ejecución. Pero si vamos a usar código puro de *Python* para realizar algún cálculo, es más eficaz, en cuanto a tiempo de computo, el definir las variables que vayamos a usar usando la librería *Numpy*, ya que la diferencia es bastante notable en todos los casos. Por último comentar que el código puro de *Python* para un número reducido de elementos (un número menor o igual a 100) no presenta diferencias significativas, cuanto a tiempo de computo, con el resto de los *scripts* usados.

2. Segunda tarea

2.1. Enunciado

Escribir un código en *Python* que:

1. Cargue la matriz A y el vector b de un archivo del disco.
2. Resolver el sistema $Ax = b$, es decir, obtener el vector x .
3. Mostrar los resultados por pantalla para comprobar que son realmente una solución.

2.2. Resolución

Comenzamos el ejercicio llamando a la función *subprocess.call* para que cada vez que ejecutemos el ejercicio, nos genere un sistema de cinco ecuaciones con cinco incógnitas. Al ejecutarlo nos genera la **Ecuación (1)**:

$$\begin{pmatrix} 18 & 12 & 15 & 6 & 1 \\ 3 & 9 & 7 & 6 & 0 \\ 0 & 0 & 15 & 7 & 19 \\ 3 & 2 & 13 & 17 & 7 \\ 4 & 17 & 18 & 14 & 19 \end{pmatrix} \times \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{pmatrix} = \begin{pmatrix} 575 \\ 264 \\ 411 \\ 490 \\ 754 \end{pmatrix}, \quad (1)$$

donde la solución a dicha ecuación es **(2)**:

$$\begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{pmatrix} = \begin{pmatrix} 15 \\ 9 \\ 6 \\ 16 \\ 11 \end{pmatrix}. \quad (2)$$

Una vez tenemos cargados los datos, los convertimos a *vectores n-dimensionales* de la librería de *Numpy* y finalmente resolvemos la ecuación con la función *numpy.linalg.solve*. Todo este proceso es resumido en la figura 6.

```
In [1]: import numpy
import MyEquation
import subprocess

#Creamos el fichero system.txt llamando al siguiente archivo
subprocess.call(["python", "equation-system-generator.py"])
#Leemos el archivo que nos devolverá la matriz A y el vector b
A,b = MyEquation.load_equation_system( 'system.txt' )
print("Matriz A: ")
print(A)
print("Vector b: ", b)
#numpy.linalg.solve --> Nos devuelve la solución de una ecuación lineal de matrices de l
x = numpy.linalg.solve(A, b)
print( "Solution (x) del sistema Ax=b: ", x )
print("Comprobación de la solución con la operación Ax (A.dot(x)): ", A.dot(x))

Matriz A:
[[ 18.  12.  15.   6.   1.]
 [  3.   9.   7.   6.   0.]
 [  0.   0.  15.   7.  19.]
 [  3.   2.  13.  17.   7.]
 [  4.  17.  18.  14.  19.]]
Vector b: [ 575.  264.  411.  490.  754.]
Solution (x) del sistema Ax=b: [ 15.   9.   6.  16.  11.]
Comprobación de la solución con la operación Ax (A.dot(x)): [ 575.  264.  411.  490.  754.]
```

Figura 6: *script modificado con las impresiones por pantalla de los resultados obtenidos.*