Universidad Complutense de Madrid

Facultad de Informática

Aprendizaje Automático y Big Data

Memoria Práctica 0.

Profesor:

- Alberto Díaz Esteban.

Alumnos:

- Marina de la Cruz López.

- Diego Alejandro Rodríguez Pereira.

Introducción

Para esta práctica hemos implementado dos versiones de un algoritmo de integración numérica basado en el método de Monte Carlo en el lenguaje de Python y utilizando la herramienta *JupyterNotebook* para la realización del código. La primera de ellas es una versión iterativa mientras que la segunda es una versión vectorial. Una vez implementados los códigos de las dos versiones hemos comparado resultados y tiempos de ejecución obtenidos en ambas versiones.

Versión Iterativa

La primera versión es una solución iterativa que realiza *num\_puntos* iteraciones para calcular el resultado.

Es un código sencillo realizado puramente con funciones iterativas como el *for loop*. Debido a esto, y a que no conocíamos muy bien la librería de *numpy*, también nos fue más fácil y rápido implementar este código que en la segunda versión.

Posteriormente, editamos esta versión para que la primera parte, en la que obtenemos el máximo de la función f(x), usase *np.linspace* para generar los puntos aleatorios y *np.max* para obtener el máximo, dejando un único *for loop* el que calcula la cantidad de puntos *y* debajo de f(*x*) para (*x,y)* aleatorios

Versión Vectorial

La segunda versión es una solución que utiliza operaciones entre vectores en vez de bucles.

Para esta versión es necesario importar la librería de *numpy* para poder acceder a las funciones vectoriales que ella dispone. Ya que en esta implementación hemos sustituido todos los bucles presentes en la versión anterior por funciones vectoriales de *numpy*.

La primera parte es igual que la versión editada de la versión iterativa, con *np.linspace* y *np.max,* y cambiamos el bucle *for* por *np.sum* que cuenta la cantidad de elementos del array con una máscara, que en nuestro caso es la condición *y* < f(*x*), siendo f(*x*) e *y* dos arrays.

A la hora de implementar ha sido la versión en la que hemos tardado más, ya que no teníamos mucha práctica con la utilización de funciones de la librería de *numpy*. A pesar de esto, y como se podrá observar en los resultados obtenidos, vale mucho la pena aprender a manejar bien esta librería ya que nos será de muy utilidad especialmente cuando se trate de manejar una gran cantidad de datos.

Dentro de los errores que cometimos se encuentra en que la función que utilizamos como parámetro tenía ciertos problemas. Específicamente al usar funciones de *math*, como, por ejemplo: *math.log*, *math.sin*, *math.tan*, ya que al parecer son funciones que solo se pueden ejecutar en un único elemento y no en un *array*. Por lo que tuvimos que utilizar funciones de *numpy*, como por ejemplo *numpy.log* o *numpy.sin*. Pero una vez utilizadas las funciones anteriores el programa funciona correctamente.

Cabe destacar, que, gracias a esta práctica, sobre todo a esta implementación, que hemos logrado aprender muchas funciones de la librería de *numpy* que no conocíamos. Entre las más útiles se encuentran, por ejemplo, *linspace*.

Gráficas

Hemos decidido mostrar las gráficas con los puntos aleatorios generados y la función para comprobar que se estaban generando de forma correcta en ambas versiones del algoritmo.

Imagen que contiene Flecha

Descripción generada automáticamente

Figura 1

Para ello hemos hecho uso de las funciones de *matplotlib.pyplot* *scatter*, para mostrar los puntos aleatorios y *plot* para mostrar la gráfica.

Resultados Obtenidos

Para comprobar que los resultados obtenidos en ambas implementaciones son correctos, hemos utilizado la función *spicy.integrate.quad* de Python.

Una vez ejecutados las tres funciones comparamos sus resultados. Los tres arrojaron resultados similares, algo que era esperar, ya que el método Monte Carlo se basa en las probabilidades, y para obtener resultados precisos habría que generar una cantidad significante de números aleatorios. Dicho esto, como los resultados fueron los correctos, pudimos comprobar que la implementación de ambas versiones también es correcta.

|  |
| --- |
| scipy.integrate.quad(np.sin,0,3)  (1.9899924966004452, 2.2093354885571127e-14)  integra\_mc(np.sin, 0,3, 10000)  1.9928999806751968  integra\_mc\_vec(np.sin, 0,3, 10000)  1.9895999807071962 |

Después probamos a ejecutar las implementaciones con otras funciones positivas.

|  |
| --- |
| Logaritmo  scipy.integrate.quad(np.log,1,3)  (1.2958368660043291, 1.43866792479611e-14)  integra\_mc(np.log, 1,3)  1.306030288768649  integra\_mc\_vec(np.log, 1, 3)  1.2928469413046317  Tangente  scipy.integrate.quad(np.tan,0,1.5)  (2.648783653978435, 8.334415026552347e-12)  integra\_mc(np.tan, 0,1.5, 40000)  2.6397858141105455  integra\_mc\_vec(np.tan, 0, 1.5, 40000)  2.6514194855669624 |

Dejando a un lado los resultados de las funciones, pasamos a los resultados obtenidos de los tiempos de ejecución de ambas implementaciones.

Los tiempos de ejecución obtenidos en la primera implementación son mucho mayores a los de la segunda implementación. Teniendo la primera un crecimiento lineal con respecto al número de elementos. Mientras que la segunda implementación tiene un crecimiento prácticamente constante respecto al número de elementos. Esto se puede observar en la *figura2.*

Estos tiempos se deben a que la librería de *numpy* es mucho más eficiente y rápida al hacer los cálculos vectoriales que realizar simples iteraciones con bucles sobre los elementos.

Podemos concluir que la libería de *numpy* viene de gran ayuda especialmente cuando se trata de manejar arrays, por lo que es preferible usar esta opción para que nuestros programas y algoritmos terminen en un menor tiempo.

Chart, scatter chart

Description automatically generated

Figura 2

Conclusiones

La biblioteca de *Numpy* hace que trabajar con las estructuras de *arrays* sea más fácil y sencillo. Por lo que además de obtener un código más compacto y legible proporciona una mayor velocidad de ejecución que al utilizar las listas propias de Python.

Por lo que consideramos aprender las funciones de esta librería, así como nosotros hicimos en esta práctica, y utilizarlos especialmente cuando se trate de manejar una gran cantidad de datos.

Código

import time

import numpy as np

import random as rn

import scipy.integrate

import matplotlib.pyplot as plt

def integra\_mc(fun, a, b, num\_puntos=10000):

#Generación de números aleatorios equiespaciados

#Posteriormente se obtienen los puntos de la coordenada Y

puntosX = np.linspace(a, b, num = num\_puntos)

puntosY = fun(puntosX)

#Calculamos el máximo valor de Y

MAX = np.max(puntosY)

#Bucle donde sacamos la cantidad de puntos y debajo de f(x) con x e y aleatorios.

num\_debajo = 0

for i in range(0, num\_puntos):

x = rn.uniform(a, b)

y = rn.uniform(0, MAX)

if(y < fun(x)):

num\_debajo = num\_debajo+1

return (num\_debajo/num\_puntos)\*(b-a)\*MAX

def integra\_mc\_vec(fun, a, b, num\_puntos=10000):

#Generación de números aleatorios equiespaciados

#Posteriormente se obtienen los puntos de la coordenada Y

x = np.linspace(a, b, num = num\_puntos)

puntosY = fun(x)

#Calculamos el máximo valor de Y

MAX = np.max(puntosY)

#Generamos los puntos y de forma aleatoria

y = np.random.uniform(0, MAX, num\_puntos)

#Hacemos la suma de los que comprueban la condición

num\_debajo = np.sum(y < puntosY)

return (num\_debajo/num\_puntos)\*(b-a)\*MAX

def compara\_tiempos(fun):

sizes = np.linspace(100, 10000000, 20)

times\_iter = []

times\_vec = []

for size in sizes:

a = 0

b = 3

tic = time.process\_time()

res\_iter = integra\_mc(fun, a, b, int(size))

toc = time.process\_time()

times\_iter += [1000 \* (toc - tic)]

tic = time.process\_time()

res\_vec = integra\_mc\_vec(fun, a, b, int(size))

toc = time.process\_time()

times\_vec += [1000 \* (toc - tic)]

p1 = len(sizes)

p2 = len(times\_iter)

plt.figure()

plt.scatter(sizes, times\_iter, c='red', label='bucle')

plt.scatter(sizes, times\_vec, c='blue', label='vector')

plt.legend()

plt.savefig('compara\_tiempos\_dot.png')