Aprendizaje Automático y minería de datos: Práctica 0

Autores: *Tatiana Duarte Balvís, Miguel Mur Cortés*

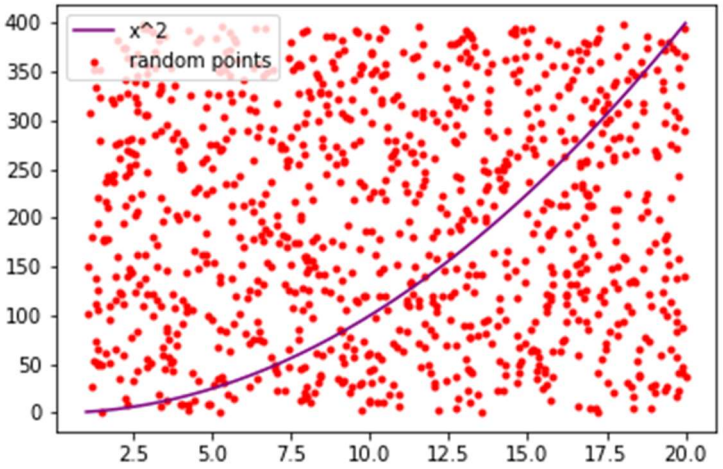
1. Objetivo:

El objetivo de la práctica es implementar un algoritmo de integración numérica basado en el método de Monte Carlo para resolver integrales definidas. Este método se basa en "disparar" números aleatorios comprendidos en el cuadrado formado en ancho por el rango de la integral y en alto por el valor máximo del valor que alcanza la función.

El porcentaje de números que caen en el área comprendida entre el eje X y la función se utiliza para calcular la integral de este modo:

$$I \approx \frac{N_{debajo}}{N_{total}} (b-a)M$$

Además de este problema, se pide resolverlo de dos maneras: Una iterativa y la otra usando las operaciones de vectores de numpy. De esta forma, se podrá apreciar cual de las dos aproximaciones es más eficiente.



2. Proceso:

Parámetros:

- Fun: Función a integrar. En nuestro caso es una función cuadrática.
- A: Límite inferior de la integral.
- B: Límite superior de la integral.
- Num_Puntos: Número de “disparos” a simular por el método de Monte Carlo.

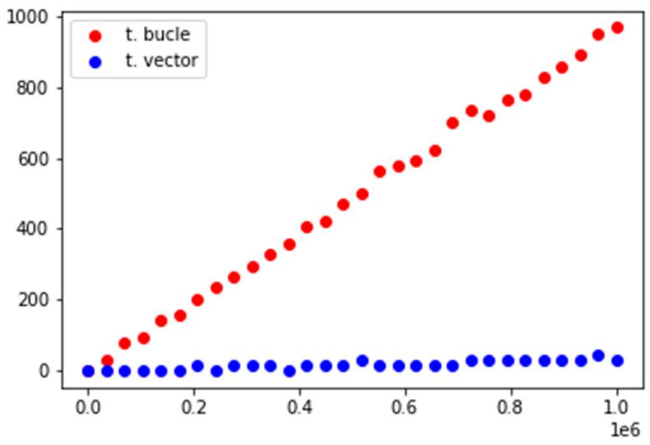
Para la resolución de este problema, hemos implementado estas funciones:

- **Integra_mc_iterativo(fun, a, b, num_puntos=10000) :** Resuelve la integral mediante el método de Monte Carlo usando un bucle.
- **Integra_mc_vectorizado(fun, a, b, num_puntos=10000):** Resuelve la integral mediante el método de Monte Carlo usando operaciones de vectores de la librería NumPy.
- **Presenta_grafica_integral(fun, a, b, num_puntos=10000):** Dibuja una gráfica que representa los “disparos” y la función utilizada.
- **Presenta_grafica_tiempo(fun, a, b, num_puntos=30):** Dibuja una gráfica que compara el tiempo de ejecución de ambas aproximaciones según el número de disparos a realizar.
- **Compara_tiempos():** Función principal que calcula la integral y sus tiempos de ejecución, además de pintar las gráficas.

3. Conclusiones:

En esta gráfica, el eje X describe el número de “disparos” generados para el cálculo de la integral según el método de Monte Carlo. El eje Y describe el tiempo de ejecución de cada método en ms.

Como se puede apreciar, el tiempo que tarda la aproximación iterativa en realizar la simulación es mucho más elevado que la aproximación por vectores. Esta diferencia crece mucho más cuanto mayores sean los “disparos” a simular.
Por tanto, podemos concluir que trabajar con operaciones sobre vectores de la librería NumPy es mucho más eficiente que usar bucles.



4. Código:

```
import time
import numpy as np
import matplotlib.pyplot as plt

def cuadratica(x):
    return x * x

def presenta_grafica_tiempo(fun, a, b, num_puntos=30):
    intervalo = np.linspace(100, 1000000, num_puntos)

    time_it = []
    time_vec = []

    for i in intervalo:
        time_it += [integra_mc_iterativo(fun, a, b, int(i))[1]]
        time_vec += [integra_mc_vectorizado(fun, a, b, int(i))[1]]

    plt.figure()
    plt.scatter(intervalo, time_it, c='red', label='t. bucle')
    plt.scatter(intervalo, time_vec, c='blue', label='t. vector')
    plt.legend()
    plt.savefig('time.png')

def presenta_grafica_integral(fun, a, b, num_puntos=10000):
    intervalo = np.linspace(a, b, num_puntos)
    M = np.max(fun(intervalo))
    plt.figure()
    plt.plot(intervalo, fun(intervalo), '-', c='purple', label='x^2')
    ran_x = np.random.rand(num_puntos) * (b-a) + a
    ran_y = np.random.rand(num_puntos) * M
    plt.scatter(ran_x, ran_y, marker='.', c='red', label='random points')
    plt.legend()
    plt.savefig('integral.png')

def integra_mc_iterativo(fun, a, b, num_puntos=10000):
    tic = time.process_time()
    #Calculo de M
    intervalo = np.linspace(a, b, num_puntos)
    M = np.max(fun(intervalo))
    #Calculamos num_puntos aleatorios
    num_debajo = 0
    for i in range(num_puntos):
        ran_x = np.random.rand() * (b-a) + a
        ran_y = np.random.rand() * M
        if(fun(ran_x) > ran_y):
            num_debajo += 1
```

```
    #Calculo de la integral
    integral = (num_debajo/num_puntos) * (b-a) * M
    toc = time.process_time()
    return (integral, (toc-tic) * 1000)

def integra_mc_vectorizado(fun, a, b, num_puntos=10000):
    tic = time.process_time()
    #Calculo de M
    intervalo = np.linspace(a, b, num_puntos)
    M = np.max(fun(intervalo))
    #Calculamos num_puntos aleatorios
    ran_x = np.random.rand(num_puntos) * (b-a) + a
    ran_y = np.random.rand(num_puntos) * M
    num_debajo = np.sum(fun(ran_x) > ran_y)
    #Calculo de la integral
    integral = (num_debajo/num_puntos) * (b-a) * M
    toc = time.process_time()
    return (integral, (toc-tic)*1000)

def compara_tiempos():
    resultado_iterativo = integra_mc_iterativo(cuadratica, 1, 20, 1000000)
    resultado_vectorizado = integra_mc_vectorizado(cuadratica, 1, 20, 1000000)
    print("Tiempo para proceso iterativo en ms:",resultado_iterativo[1])
    print("Resultado de la integral:",resultado_iterativo[0])
    print("Tiempo para proceso vectorizado en ms:",resultado_vectorizado[1])
    print("Resultado de la integral:",resultado_vectorizado[0])
    presenta_grafica_integral(cuadratica, 1, 20, 1000)
    presenta_grafica_tiempo(cuadratica, 1, 20, 30)

compara_tiempos()
```