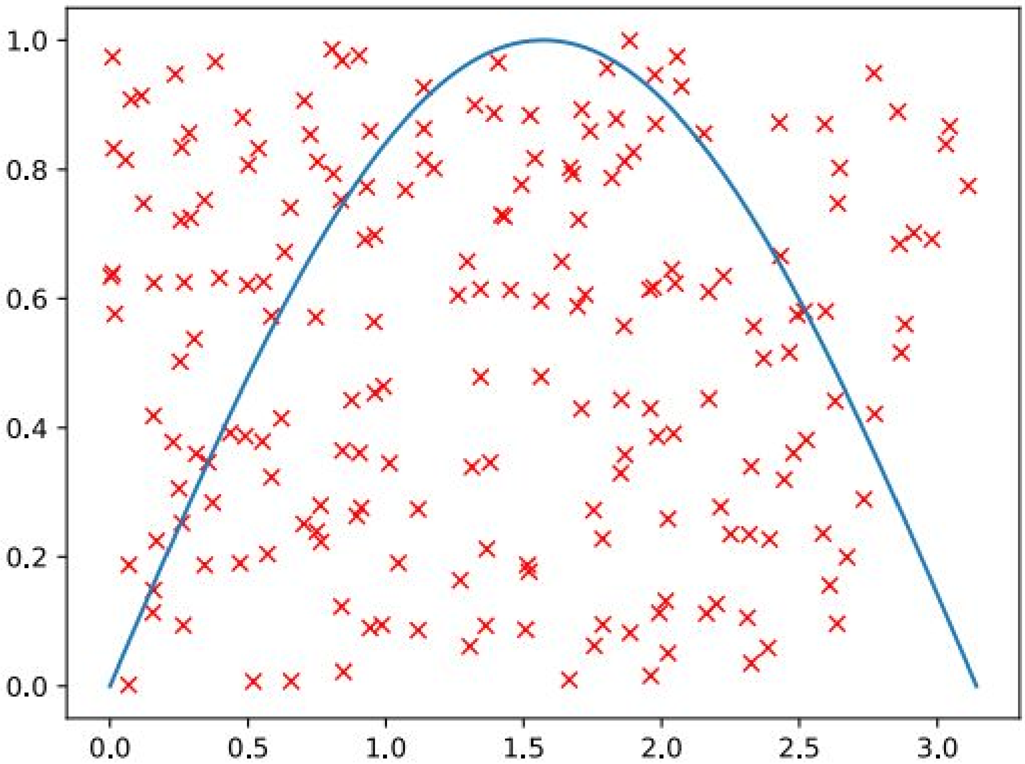
ALGORITMO CÁLCULO INTEGRAL DEFINIDA : MÉTODO DE MONTECARLO



Realizado por: Tomás Golomb Durán y
Montserrat Sacie Alcázar
Universidad Complutense de Madrid
Asignatura: Aprendizaje Automático y Big Data
Profesor: Pedro Antonio González Calero
Curso: 2018-2019

En esta práctica hemos implementado el método de MonteCarlo para calcular integrales definidas en un intervalo [a, b] de funciones positivas.Para ello usamos el lenguaje python y las librerías numpy, random, math, time y matplotlib.

```
import matplotlib.pyplot as plt
from matplotlib.backends.backend_agg import FigureCanvasAgg
from matplotlib.figure import Figure
import random as rd
import numpy as np
import math
import time
```

Función integra_mc_bucle (1)

En la primera implementación del algoritmo (versión iterativa) calculamos *numPuntos* aleatorios *x* y comprobamos cual es el máximo de todos ellos en cada iteración, almacenando el máximo en una variable *m*.

A continuación generamos de nuevo los puntos *x* aleatorios, ya que no hemos usado vector para almacenarlos, y generamos las imágenes aleatorias de esos puntos *y*. En cada iteración del bucle comprobamos si la *y* generada se encuentra por debajo de la imagen real *fun* de la *x* aleatoria y en ese caso incrementamos una unidad la variable contador *hit.*

Esta función devuelve una tupla con el resultado de la integral definida de fun en el intervalo *a* y *b*; y el tiempo de ejecución del método.

```
def integra_mc_bucle(fun, a, b, num_puntos = 100000):
        hit = 0
        m = 0
        tic = time.time()
        for i in range(0, num_puntos):
                randX = rd.uniform(a, b)
                m = max(m, fun(randX))
        for i in range(0, num_puntos):
                randX = rd.uniform(a, b)
                randY = rd.uniform(0, m)
                if (fun(randX) >= randY):
                        hit += 1
        toc = time.time()
        return ((hit / float(num_puntos))*(b-a)*m, (toc - tic)*1000)
```

Función integra_mc_vector (2)

En este caso pretendemos usar vectores de numpy y sus operaciones para resolver el mismo algoritmo de forma más eficiente en tiempo que *integra_mc_bucle*.

En un vector *xvect* guardamos las *x* generadas aleatoriamente y calculamos la imagen de cada elemento sustituyéndolo en la función *fun*. Las imágenes f(x) se guardan en el vector *fxvect*.

Con la función *np.max* aplicada sobre *fxvect* calculamos la máxima imagen de los puntos generados, *m*.
Por otro lado, guardamos en el vector *yvect* las *y* aleatorias.
Para comprobar qué *y* aleatorias quedan debajo de la función, comparamos *fxvect* e *yvect* (f(x) debe ser mayor o igual que y) y guardamos los booleanos en *rvect*. Con *rvect* ya podemos sumar los elementos para obtener el contador de puntos que quedan dentro (true = 1).

De nuevo devolvemos una tupla con el resultado de la integral y el tiempo de ejecución.

```
def integra_mc_vector(fun, a, b, num_puntos=1000):
        tic = time.time()
        xvect = np.linspace(a, b, num_puntos)
        fxvect = map(fun, xvect)
        m = np.max(fxvect)
        yvect = np.linspace(0, m, num_puntos)
        rvect = np.less_equal(yvect, fxvect)
        ptsDentro = np.sum(rvect)
        toc = time.time()
        return ((ptsDentro / float(num_puntos))*(b-a)*m, (toc - tic)*1000)
```

Nuestro objetivo además del cálculo de la integral, es comprobar que las operaciones vectorizadas son más eficientes que las iterativas. Para ello definimos la función *comparaTiempos()* en la que llamamos a las funciones (1) y (2) con diferentes números de puntos (guardados en varray) que se pasan como parámetro a las funciones. Como ejemplo a = 0, b = math.pi

Para cada cálculo del bucle *for* guardamos los tiempos de ejecución en los vectores time_bucle y time_vector.

```
def comparaTiempos():
        varray = np.linspace(100, 1000000, 20)
        time_bucle = []
        time_vector = []
        ticGrande = time.time()
        for nums in varray:
                par = integra_mc_bucle(fLineal, 0, math.pi, int(nums))
                print(par[0])
                time_bucle += [par[1]]

                par2 = integra_mc_vector(fLineal, 0, math.pi, int(nums))
                print(par2[0])
                time_vector += [par2[1]]
        plt.figure()
        plt.xlabel('Numeros generados aleatoriamente')
        plt.ylabel('Tiempo de ejecucion')
        plt.scatter(varray, time_bucle, c='red', label='Bucle')
        plt.scatter(varray, time_vector, c='blue', label='Vector')
        plt.legend()

        plt.savefig('comparacion.png')
        tocGrande = time.time()
        print (tocGrande - ticGrande)
```

Finalmente dibujamos en una gráfica (usando librería matplot) estos dos vectores para compararlos visualmente y confirmar nuestra hipótesis de eficiencia.