Clase 4 - Numpy y Matplotlib

March 25, 2017

1 Introducción

En la clase de hoy trabajaremos con los paquetes de cálculo científico y graficación `numpy` y `matplotlib`, ambos miembros de la *suite* scipy. Con estos dos paquetes (junto con `scipy.linalg`) podemos reemplazar muchas de las utilidades que programas como Matlab nos dan: graficación, cálculos numéricos con matrices...

```
In [2]: import numpy as np
```

2 Numpy

2.1 Arreglos de Numpy.

Así como los arreglos y las matrices son el objeto esencial en matlab, los `numpy.array` y las `numpy.matrix` son los objetos esenciales para los cálculos en Numpy. Una de las diferencias principales entre los arreglos de numpy y las listas es que, mientras las listas pueden almacenar objetos de diferentes tipos, los arreglos solo admiten elementos de un tipo a la vez (es decir, no se pueden tener arreglos como [1, False, [1,2,3]].

```
In [3]: X = np.array([1,2,3])
        print(X)
        A = np.matrix('1,2; 3,4')
        print(A)
```

```
[1 2 3]
[[1 2]
 [3 4]]
```

Los arreglos y las matrices se pueden sumar, y ahora la suma no representa concatenación sino suma componente a componente:

```
In [4]: X = np.array([1,2,3])
        Y = np.array([2,4,6])
        print(X + Y)
```

```
[3 6 9]
```

```
In [5]: A = np.matrix([[1,2], [3,4]])
        B = np.matrix('0, 1; 1, 1')
        print(A + B)
        print(type(A+B))

[[1 3]
 [4 5]]
<class 'numpy.matrixlib.defmatrix.matrix'>
```

2.2 Funciones para crear arreglos

A la hora de crear arreglos grandes, podemos recurrir a las siguientes funciones (que resultan ser muy parecidas a las usadas en Matlab):

- `numpy.arange(inicio, fin, paso)`
- `numpy.linspace(inicio, fin, cantidad)`, para crear un arreglo con puntos distribuidos de forma lineal entre `inicio` y `fin`.

```
In [6]: np.arange(1,10)

Out[6]: array([1, 2, 3, 4, 5, 6, 7, 8, 9])

In [7]: np.linspace(1,10,5)

Out[7]: array([  1.  ,   3.25,   5.5 ,   7.75,  10.  ])
```

A veces se necesitan arreglos con números aleatorios, para eso podemos usar - `numpy.random.random(tamaño)` para obtener números entre 0 y 1 escogidos de forma uniforme al azar. - `numpy.random.randint(menor, mayor, tamaño))` para obtener números enteros entre el menor y el mayor - 1.

```
In [8]: np.random.random(10)

Out[8]: array([ 0.65331648,  0.47867165,  0.20051284,  0.80913836,  0.73259759,
                0.132641  ,  0.55043593,  0.14667835,  0.57072164,  0.09161002])

In [9]: np.random.random((2,2))

Out[9]: array([[ 0.00090092,  0.22721791],
               [ 0.8447204 ,  0.35449551]])

In [10]: np.random.randint(1, 10, 10)

Out[10]: array([8, 2, 3, 3, 3, 1, 8, 6, 8, 7])

In [11]: np.random.randint(1, 10, (2,2))

Out[11]: array([[5, 3],
                [7, 4]])
```

2.3 Slicing de arreglos

Además del slicing usual para listas, Numpy nos permite hacer slicing de los arreglos de una forma más inteligente.

```
In [12]: X = np.array([1,2,3])
         print(X[:2])
         print(X % 2 == 0)

[1 2]
[False  True False]
```

Podemos sacar de un arreglo todos los valores que cumplan cierto valor. Por ejemplo:

```
In [13]: X[X % 2 == 0]

Out[13]: array([2])
```

Es decir, primero sacamos una *máscara* (en este caso `X % 2 == 0`) que consiste en un arreglo booleano (en este caso `[False True False]`). Al pasarle esta máscara a X estamos pidiendo que devuelva solamente los valores `True`.

2.4 Ejercicio: estimar π usando números aleatorios

Podemos estimar π usando solo números enteros: la probabilidad de que dos número enteros seleccionados al azar sean primos relativos (i.e. su máximo común divisor sea 1) es $6/\pi^2$. Usando este hecho podemos estimar π de la siguiente forma:

2.4.1 1. Implementamos una función máximo común divisor:

Usando el algoritmo de euclides:

```
In [14]: def gcd(a,b):
             while b != 0:
                 a, b = b, a%b
             return a
```

Podríamos también usar una que ya esté implementada, como `math.gcd`.

2.4.2 2. Creamos arreglos con números aleatorios enteros de tamaño n.

```
In [15]: n = 1000
         X = np.random.randint(1, 1000, n)
         Y = np.random.randint(1, 1000, n)
```

2.4.3 3. Iteramos sobre ambos, y contamos cuántos son primos relativos usando un diccionario.

```
In [16]: diccionario_de_freq = {'primos relativos': 0, 'no primos relativos': 0}
         for k in range(n):
             if gcd(X[k], Y[k]) == 1:
                 diccionario_de_freq['primos relativos'] += 1
             else:
                 diccionario_de_freq['no primos relativos'] += 1
```

```
In [17]: print(diccionario_de_freq)
```

```
{'primos relativos': 637, 'no primos relativos': 363}
```

2.4.4 4. Calculamos la probabilidad usando la regla de Laplace.

```
In [18]: prob = diccionario_de_freq['primos relativos']/(diccionario_de_freq['primo
         print(prob)
```

```
0.637
```

si la probabilidad p es igual a $6/\pi^2$, despejamos y llegamos a que $\pi = \sqrt{6/p}$

```
In [19]: aprox_pi = np.sqrt(6/prob)
         print(aprox_pi)
```

```
3.06906374588
```

2.4.5 5. Podemos juntar todo en una función:

```
In [20]: def aproximar_pi(n):
             X = np.random.randint(1, 1000, n)
             Y = np.random.randint(1, 1000, n)
             dict_de_freq = {'pr': 0, 'npr': 0}
             for k in range(n):
                 if gcd(X[k], Y[k]) == 1:
                     dict_de_freq['pr'] += 1
                 else:
                     dict_de_freq['npr'] += 1
             prob = dict_de_freq['pr']/n
             return np.sqrt(6/prob)
```

```
In [21]: aproximar_pi(1000000)
```

```
Out[21]: 3.1379302549752111
```

3 Matplotlib

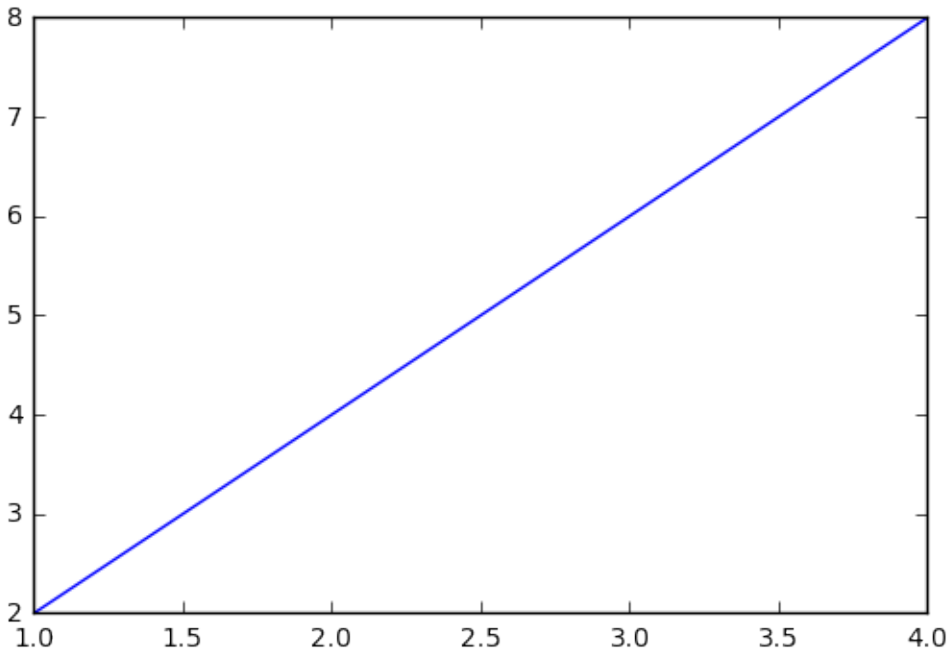
Mientras que `numpy` sirve para hacer los cálculos numéricos, la librería `matplotlib` sirve para realizar gráficos. La forma usual de importarla es la siguiente:

```
In [22]: import matplotlib.pyplot as plt
```

Podemos graficar objetos en listas o arreglos (o, en general, secuenciables numéricos):

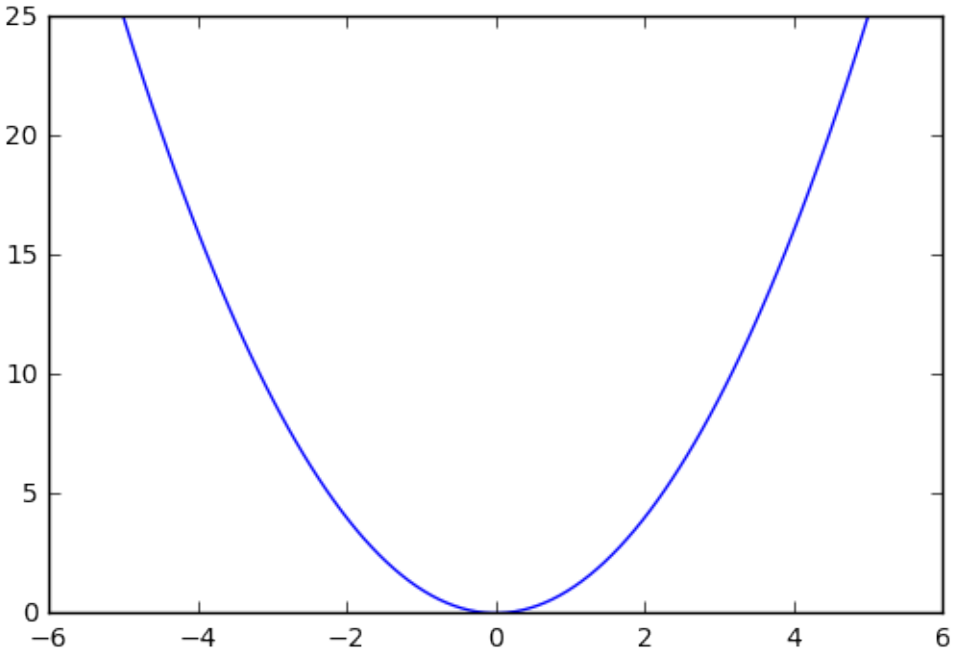
3.1 `plt.plot` y `plt.figure`

```
In [23]: plt.plot([1,2,3,4], [2,4,6,8])
         plt.show()
```



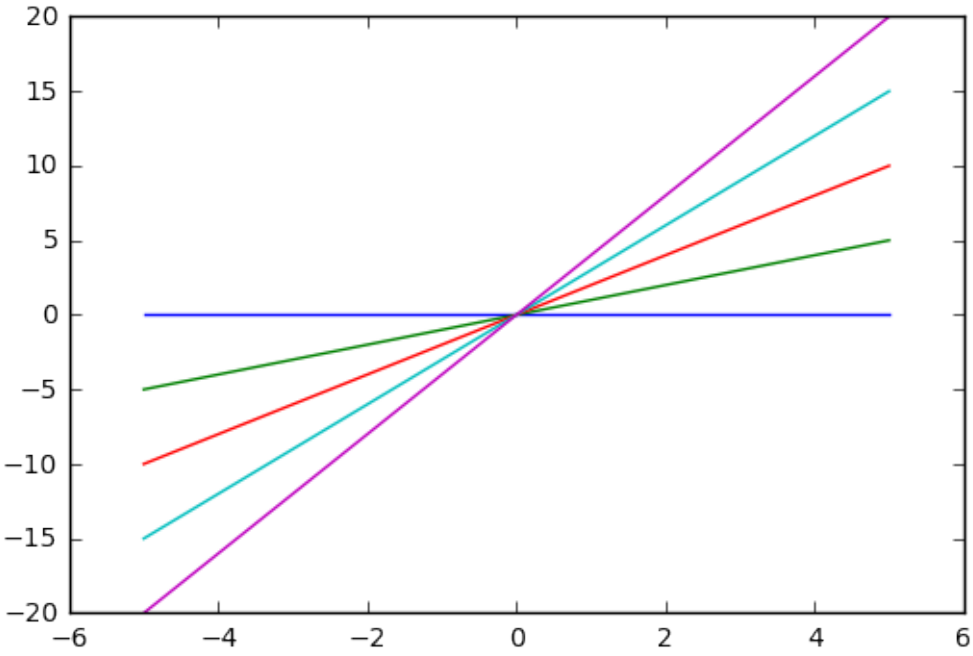
Podemos aprovechar la manipulación de arreglos de numpy para hacer funciones más elaboradas:

```
In [24]: X = np.linspace(-5, 5, 100)
         Y = X ** 2
         plt.plot(X,Y)
         plt.show()
```



Para dibujar más de una curva en una misma figura, iniciamos la figura usando `plt.figure()` y graficamos usando `plt.plot(...)`.

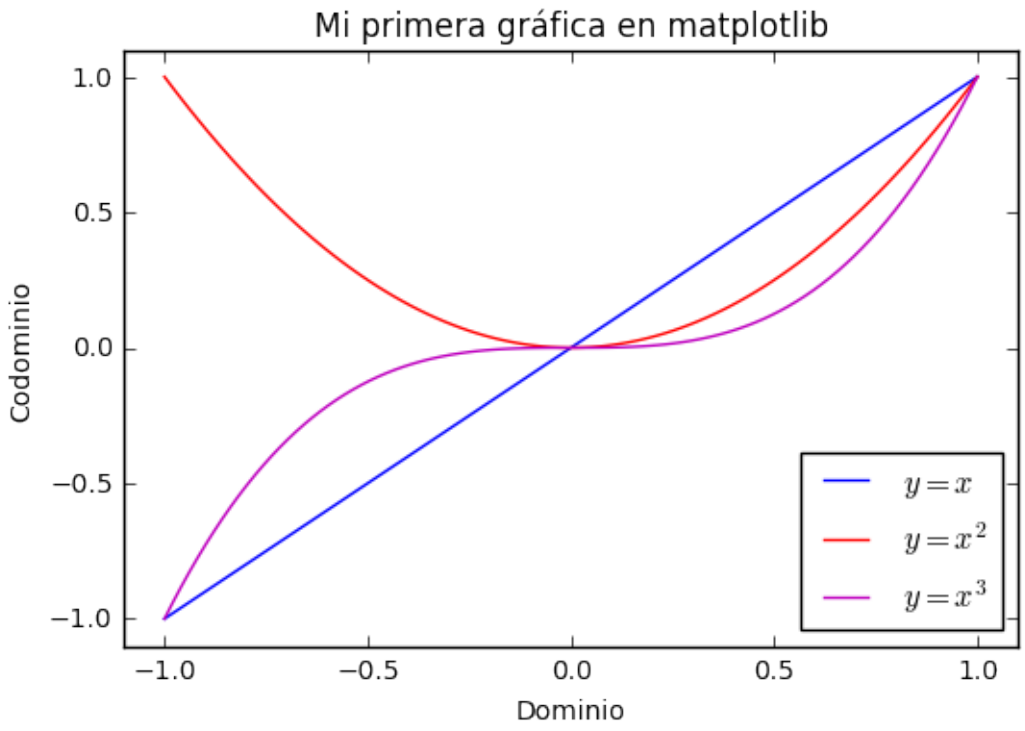
```
In [25]: X = np.linspace(-5, 5, 100)
         plt.figure()
         for k in range(5):
             plt.plot(X, k*X)
         plt.show()
         plt.close() # es una buena práctica cerrar las figuras.
```



3.2 Más características

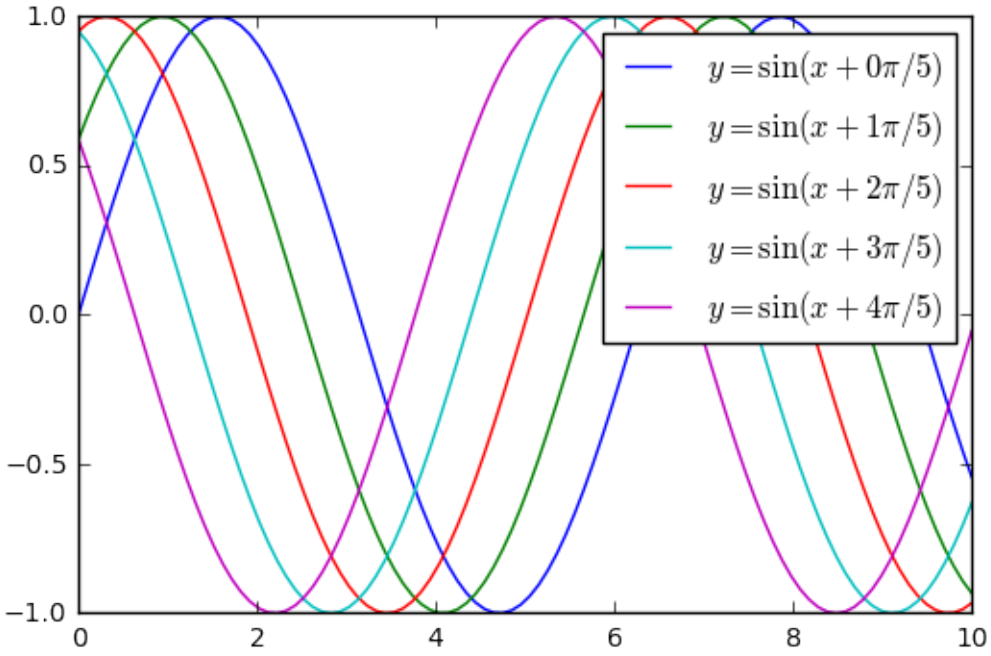
Podemos añadirle más características a las gráficas que realizamos, como título, leyenda, etiquetas para los ejes, límites para los ejes, color de relleno entre los ejes... por ejemplo:

```
In [26]: X = np.linspace(-1, 1, 100)
         plt.figure()
         plt.plot(X, X, 'b', label=r'$y = x$') # r de raw
         plt.plot(X, X ** 2, 'r', label=r'$y = x^2$')
         plt.plot(X, X ** 3, 'm', label=r'$y = x^3$')
         plt.legend(loc='best')
         plt.title('Mi primera gráfica en matplotlib')
         plt.xlim([-1.1, 1.1])
         plt.ylim([-1.1, 1.1])
         plt.xlabel('Dominio')
         plt.ylabel('Codominio')
         plt.show()
         plt.close()
```



Podríamos combinar graficación con nuestro conocimiento sobre formateo de strings:

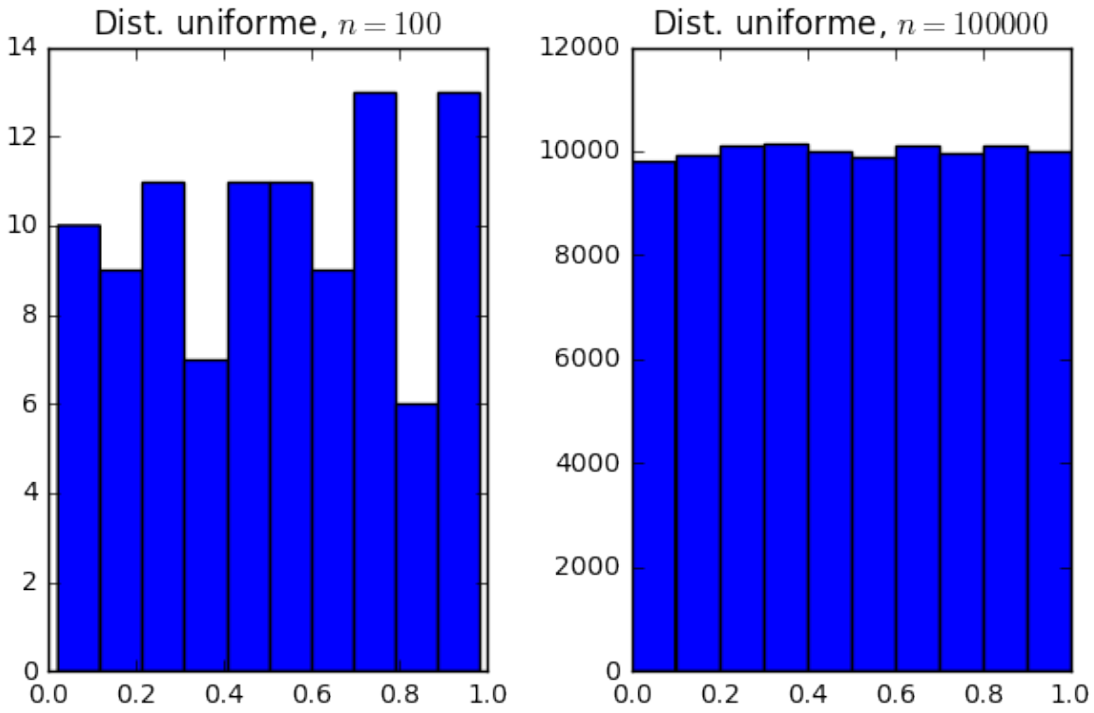
```
In [27]: X = np.linspace(0, 10, 100)
         plt.figure()
         for k in range(5):
             plt.plot(X, np.sin(X + k*np.pi/5), label=r'$y = \sin(x + {}\pi/5)$'.fo
         plt.legend(loc='upper right')
         plt.xlim([0,10])
         plt.show()
         plt.close()
```



3.3 Subplots y guardar imágenes

Además, y como en matlab, podemos crear subplots:

```
In [28]: X1 = np.random.random(100)
         X2 = np.random.random(100000)
         plt.figure()
         plt.subplot(121) #filas, columnas y posición
         plt.hist(X1)
         plt.title(r'Dist. uniforme, $n=100$')
         plt.subplot(122)
         plt.hist(X2)
         plt.title(r'Dist. uniforme, $n=100000$')
         plt.tight_layout() # para que se vea más lindo.
         plt.show()
         plt.close()
```



Por último, podemos guardar las imágenes que creamos usando `plt.savefig(nombre, formato, dpi,...)`

```
In [29]: X1 = np.random.normal(0, 1, 100000)
         X2 = np.random.gamma(1, 1, (100000,))
         plt.figure()
         plt.subplot(121)
         plt.hist(X1)
         plt.title('Dist. Normal')
         plt.subplot(122)
         plt.hist(X2)
         plt.title('Dist. Gamma')
         plt.tight_layout()
         plt.savefig('distribuciones.png', format='png')
         plt.close()
```

3.4 Ejercicio: hacer un diagrama de barras con un conteo de palabras

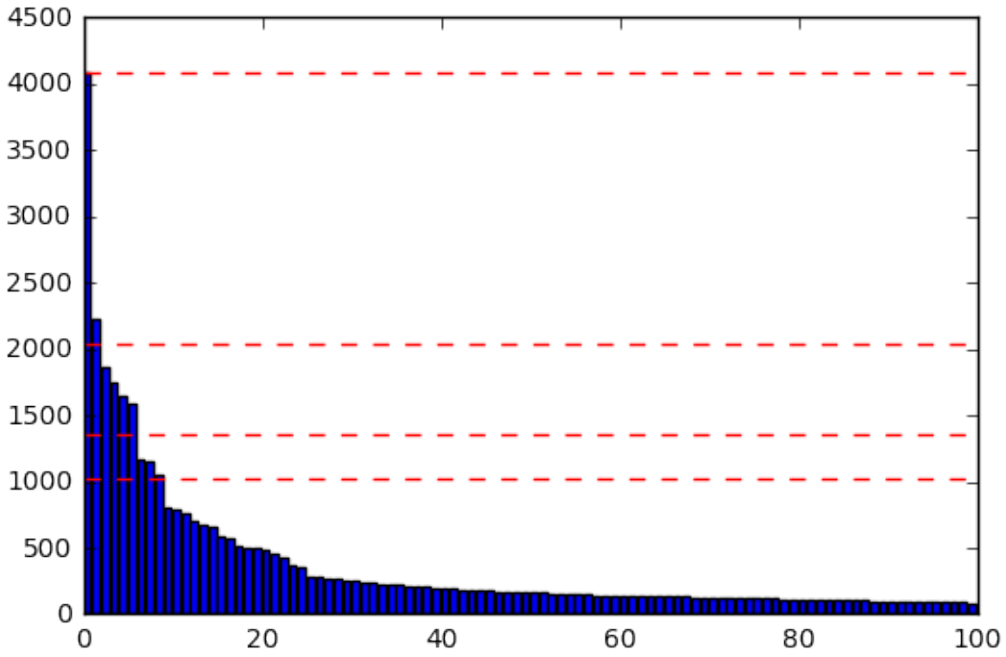
Como vimos en los ejercicios de la clase anterior, la distribución de frecuencias de palabras en textos parece obedecer la llamada "ley de Zipf", que afirma que la distribución se comporta muy parecido a una distribución de Pareto.

```
In [41]: import csv
```

```
In [42]: with open('tabla_palabras_DUBLINERS.csv') as dublinenses_archivo:
             dublinenses_csv = csv.reader(dublinenses_archivo, delimiter = ',')
```

```
            lista_de_palabras = []
            lista_de_valores = []
            for fila in dublinenses_csv:
                lista_de_palabras.append(fila[0])
                lista_de_valores.append(int(fila[1]))
```

```
In [43]: biggest_value = lista_de_valores[0]
         plt.bar(range(len(lista_de_palabras[:100])), lista_de_valores[:100])
         plt.axhline(biggest_value, color='red', ls='--')
         plt.axhline(biggest_value/2, color='red', ls='--')
         plt.axhline(biggest_value/3, color='red', ls='--')
         plt.axhline(biggest_value/4, color='red', ls='--')
         plt.show()
```



Con esto, tenemos lo suficiente para crear una función que guarde la imagen del análisis de Zipf:

```
In [44]: def imagenZipf(path_del_archivo_csv, titulo='Análisis de Zipf', nombre_arc
             with open(path_del_archivo_csv) as archivo:
                 archivo_csv = csv.reader(archivo, delimiter = ',')
                 lista_de_palabras = []
                 lista_de_valores = []
                 for fila in archivo_csv:
                     lista_de_palabras.append(fila[0])
                     lista_de_valores.append(int(fila[1]))
             valor_mas_grande = lista_de_valores[0]
```

```
        plt.bar(range(len(lista_de_palabras[:100])), lista_de_valores[:100])
        plt.axhline(valor_mas_grande, color='red', ls='--')
        plt.axhline(valor_mas_grande/2, color='red', ls='--')
        plt.axhline(valor_mas_grande/3, color='red', ls='--')
        plt.axhline(valor_mas_grande/4, color='red', ls='--')
        plt.title(titulo)
        plt.xlabel('Índice de la palabra')
        plt.ylabel('Frecuencia')
        plt.savefig(nombre_archivo, format='png')
        plt.close()

In [45]: imagenZipf('tabla_palabras_THUSSPAKEZARATHUSTRA.csv', 'Análisis de Zipf -T
```