FACULTAD DE INFORMÁTICA DE BARCELONA

Estimación de la cardinalidad

Informe de la práctica

Sergio Martínez López y Cristian Dávila Carmona

13/01/2014

Índice

Introducción

En el área del Análisis de Flujos de Datos (Data Stream Analysis), el objetivo es desarrollar algoritmos que extraigan información útil de una colección de datos de gran volumen. Uno de los problemas de esta área es la de poder determinar el número de elementos distintos de una secuencia, es decir, la cardinalidad de ésta. Para ello, se necesitan algoritmos que funcionen en estos escenarios específicos, donde podemos tener diferentes limitaciones computacionales (memoria, tiempo, etc).

El objetivo de la práctica es investigar y escoger un algoritmo capaz de realizar una estimación de la cardinalidad de una secuencia e implementarlo. Además, a partir de éste se realizarán diversos experimentos para visualizar como se comporta el algoritmo en diversos escenarios, documentando los resultados obtenidos y las conclusiones extraídas a partir de éstos.

Estimación de la cardinalidad

La estimación de la cardinalidad consiste en indicar una estimación del número de elementos diferentes que contiene un conjunto de datos. Es decir, al trabajar con grandes volúmenes de datos podemos no tener la memoria suficiente para calcular el número exacto de elementos diferentes, por lo que debemos utilizar estimaciones que se acerquen al valor real. Por lo tanto, se necesitan algoritmos capaces de realizar estimaciones de estos valores en estos escenarios.

Objetivo

El principal objetivo que se tiene que tener en cuenta en estos escenarios es encontrar un algoritmo que, utilizando una memoria auxiliar pequeña, nos haga una estimación de la cardinalidad con el mínimo error relativo posible. Es decir, estos algoritmos mejoran la estimación de la cardinalidad a medida que aumenta la memoria auxiliar utilizada, no obstante, deben realizar estimaciones aceptables aunque la memoria auxiliar sea extremadamente pequeña. Por lo tanto, que la estimación tenga el mínimo error posible es tan necesario como que el algoritmo necesite utilizar la mínima memoria auxiliar posible.

Investigación

A la hora de realizar la investigación bibliográfica de este tema, hemos utilizado como herramienta principal Internet. Principalmente, hemos dado prioridad a los artículos publicados por profesores de universidades o empleados de importantes empresas en el sector, como pueden ser Google, etc. Además, el método de búsqueda no ha sido simplemente mediante el buscador de la anterior empresa, sino que también se ha ido visitando las diferentes referencias que tenían los artículos revisados, proporcionando fuentes de información fiable.

Inicialmente, el primer algoritmo de estimación de la cardinalidad que encontramos era el algoritmo LogLog. No obstante, al investigar un poco más sobre este algoritmo nos encontramos con que tenía dos optimizaciones: el SuperLogLog y el HyperLogLog. Por lo tanto, decidimos utilizar una de las dos optimizaciones y nos decantamos por el HyperLogLog, ya que era el más óptimo y, también, era el algoritmo del que más información encontrábamos.

Por otro lado, cabe destacar también que la investigación no solamente se ha centrado en la estimación de la cardinalidad, sino que también se ha investigado información relacionada con las funciones de hash universales. Más concretamente, el objetivo ha sido localizar números primos cercanos al máximo número de bits con el que trabajaríamos (en nuestro caso, 32 bits) y, también, métodos para tratar con strings de longitudes variables.

Algoritmo

El algoritmo que hemos implementado finalmente es el algoritmo HyperLogLog. Este algoritmo tiene un error relativo de $\pm 1.04/\sqrt{m}$, que en comparación con el algoritmo LogLog ($\pm\ 1.30/\sqrt{m}$), produce una estimación mejor de la cardinalidad utilizando la misma cantidad de memoria auxiliar.

Por otro lado, en comparación con otros algoritmos, utiliza una cantidad de memoria auxiliar menor. La principal diferencia se haya en el tamaño de cada registro de la tabla a utilizar, mientras que la mayoría de algoritmos utilizan registros de 32 bits, el HyperLogLog solamente necesita 5 bits por registro. Esto es debido a que la estimación del algoritmo se realiza a partir del número inicial de bits a 0 de los *hashvalues*, siendo extraños los *hashvalues* con una gran cantidad de ceros iniciales (que indican una mayor cardinalidad). Por lo tanto, al utilizar *hashvalues* del tamaño de 32 bits, no necesitamos más de 5 bits para representar la cantidad de ceros iniciales (en el caso de trabajar con valores de 64 bits, se necesitarían registros del tamaño de 6 bits al menos).

Seguidamante, hay una descripción del HyperLogLog en alto nivel extraída de [1]:

Require:

Let $h : D \rightarrow \{0,1\}^{32}$ hash data from domain D.
Let $m = 2^p$ with $p \in [4..16]$.

Phase 0: Initialization.

Define $\alpha_{16} = 0.673$, $\alpha_{32} = 0.697$, $\alpha_{64} = 0.709$,
$\alpha_m = 0.7213/(1 + \frac{1.079}{m})$ for $m \geq 128$.
Initialize m registers $M[0]$ to $M[m-1]$ to 0.

Phase 0: Aggregation.

for all $v \in S$ **do**
$x := h(v)$
$idx := \langle x_{31}, \dots, x_{32-p} \rangle_2$
$w := \langle x_{31-p}, \dots, x_0 \rangle_2$
$M[idx] := \max\{M[idx], \varrho(w)\}$
end for

Phase 2: Result computation.
$E := \alpha_m m^2 \cdot (\sum_{j=0}^{m-1} 2^{-M[j]})^{-1}$
if $E \leq \frac{5}{2}m$ **then**
Let V be the number of registers equal to 0.
if $V \neq 0$ **then**
return LinearCounting(m, V)
else
return E
end if
else if $E \leq \frac{1}{30}2^{32}$ **then**
return E
else
return $-2^{32} \log(1 - \frac{E}{2^{32}})$
end if

Define LinearCounting(m, V)
return $m \log(\frac{m}{V})$

Observando el código en alto nivel anterior, podemos ver que el algoritmo se separa en tres fases diferentes: inicialización, inserción de los elementos y computación de la estimación. Además, se necesita una función de hash que devuelva una secuencia de 32 bits y que el tamaño de la tabla (m) sea una potencia de 2.

En la primera parte, la de inicialización, como su propio nombre indica se inicializan diversas variables que serán utilizadas. Más concretamente, se inicializa la variable α en función del tamaño de la tabla y, además, se inicializan todos los registros de la tabla (M) a 0.

Seguidamente, se procede a la inserción de todos los elementos del flujo de datos. Para cada elemento de éste, se le aplica la función de hash que utilicemos (en nuestro caso, utilizamos una función de hash universal, la cuál se puede observar en el apéndice "Código programa Cardest") y, a partir del *hashvalue* obtenido, coge los primeros *p* bits de éste como índice de la

posición en la tabla M y, al resto de bits, se le aplica una función que devuelve en que posición está el primer bit a 1. Finalmente, si la posición del primer bit a 1 del *hashvalue* es mayor que la que estaba guardada en la posición que señala el índice obtenido, entonces se sobrescribe para guardar el máximo valor.

Para finalizar, se produce el cálculo de la estimación. La estimación se calcula a partir de la fórmula que se describe en el pseudocódigo: $E := \alpha_m m^2 \cdot (\sum_{j=0}^{m-1} 2^{-M[j]})^{-1}$. Además, a diferencia del algoritmo LogLog, el HyperLogLog cuenta con varias optimizaciones según el rango de la cardinalidad, ya que son necesarias algunas correciones cuando este rango es pequeño o, al contrario, es muy grande (cuando la cardinalidad se acerca a 2^{32}, el número de colisiones de la tabla de hash suele aumentar).

Primer experimento

Objetivo

El objetivo de este script es crear una tabla por cada dataset, introduciendo en la tabla los valores de las K ejecuciones. Concretamente, los valores que queremos obtener de cada ejecución son la estimación obtenida, el error relativo cometido y el tiempo de la ejecución.

Además, este script también creará una tabla resumen con el valor medio de las estimaciones obtenidas, el error estándar y el tiempo medio por elemento de cada dataset.

Algoritmo

El script está implementado en python. Se encarga de realizar K ejecuciones sobre cada uno de los 9 datasets y formar una tabla para cada uno de ellos con la información obtenida de sus respectivas ejecuciones.

Inicialmente, la función que se encarga de procesar el dataset crea un nuevo fichero, de extensión 'csv', para introducir los datos e inicializa a 0 las variables necesarias para obtener los rangos de errores.

Seguidamente, por cada iteración K, el script ejecuta el programa "cardest" y obtiene la salida. La salida es de vuelta como un string y es necesario que sea tratada en una función aparte, donde lo procesa y separa la estimación y el número de elementos en 2 variables distintas. Cuando tiene las dos variables, procede a incrementar la variable 'estimacionAcumulada' (utilizada para calcular la estimación media), y la 'estimacionAcumuladaCuadrado' (utilizada en el calculo del error estandar). Por otro lado, para obtener el tiempo de ejecución obtiene el tiempo antes y después de la ejecución del programa. Después de la ejecución, calcula el tiempo total en milisegundos, e incrementa la variable 'tiempoAcomulador', utilizada posteriormente para calcula el tiempo medio de ejecución. Además, también calcula el error relativo e introduce en el fichero una nueva fila con el número de la iteración, la estimación obtenida, el error relativo de la estimación y el tiempo de ejecución. Por último, incrementa la variable del intervalo correspondiente según el error relativo (0%-1%, 1%-5%, 5%-10%, etc).

Una vez se han realizado las K ejecuciones, el script cierra el archivo de la tabla, calcula la estimación media dividiendo el valor de la variable 'estimacionAcumulada' entre K, calcula el

error estándar, el tiempo medio de ejecución en milisegundos y el tiempo medio por elemento en microsegundos. Antes de finalizar con el dataset, imprime por pantalla la cantidad de ejecuciones que ha habido en cada intervalo del error relativo definido y, finalmente, introduce los valores obtenidos (estimación media, etc) en una lista llamada 'tablaResumen', utilizada al final para crear una tabla resumen de los diferentes datasets.

Un último apunte a destacar es que el programa "cardest" que calcula la estimación basa los números random en el tiempo del procesador. Para garantizar que entre dos ejecuciones los números aleatorios son realmente aleatorios y distintos entre ellos, el script realiza una pausa de 1 segundo entre ejecución y ejecución, garantizando así la aleatoriedad.

Resultados

A partir de los datos extraídos de las estimaciones de los nueve datasets, podemos observar que la mayoría de estimaciones se encuentran alrededor de la cardinalidad exacta de cada dataset. No obstante, todos los datasets suelen tener algunos picos en los que el error es notóriamente mayor, pero éstos son solo una minoría del total de ejecuciones realizadas. Por otro lado, teniendo en cuenta que la memoria auxiliar es de 256 bytes (m = 256), y que la mayoría de los errores relativos se mantienen entre el 0%-10%, podemos asegurar que las estimaciones se encuentran en un intervalo aceptable y, por lo tanto, el algoritmo responde bastante bien ante las diferentes secuencias sobre las que se ha ejecutado. Cabe destacar principalmente los resultados del octavo dataset, ya que teniendo una cardinalidad real cercana a los 500000 elementos se consigue una estimación muy aproximada utilizando sólo 256 bytes de memoria, por lo que podemos apreciar el verdadero potencial de este tipo de algoritmos.

Respecto al error relativo de las estimaciones, anteriormente hemos mencionado que el error relativo del algoritmo HyperLogLog es de $\pm 1.04/\sqrt{m}$. A partir de esto, obtenemos que el error relativo debería ser 6.5%, el cual algunas de las muestras obtenidas lo superan. No obstante, la gran mayoría de muestras obtenidas son inferiores a este error relativo, por lo que pensamos que el algoritmo tiende a dar estimaciones con errores relativos menores al señalado.

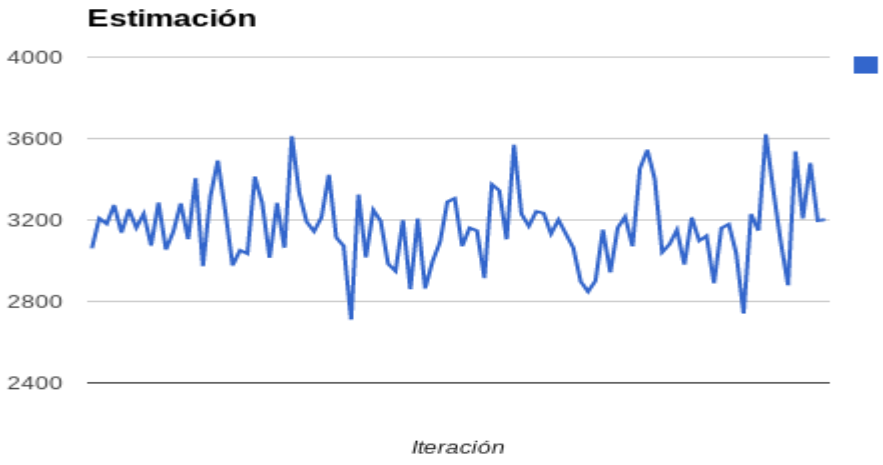
Por otro lado, podemos observar que el tiempo se mantiene constante a lo largo de las 100 ejecuciones por cada dataset, con solo algunos pequeños picos sin gran importancia. Más destacables son las diferencias de tiempo entre los datasets, ya se pueden observar pequeñas

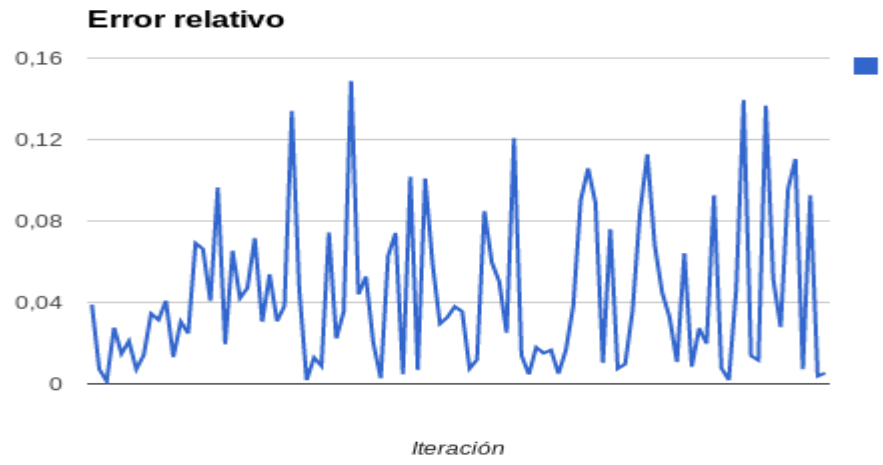
diferencias en el tiempo entre ellos. No obstante, estas diferencias se deben basar en la cantidad de elementos por entrada que tiene cada dataset y por el tamaño de estos elementos, ya que debido a diversos factores como, por ejemplo, la función de hash, podría tener un coste computacional más alto a la hora de tener que tratar con strings de una longitud mayor.

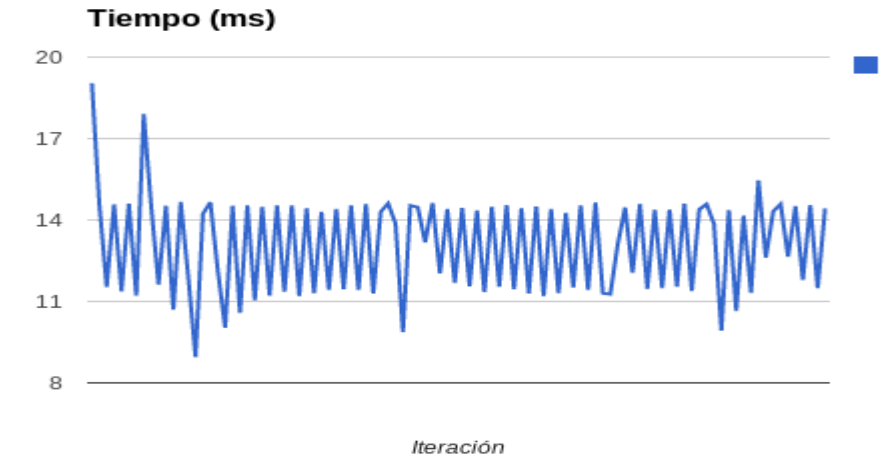
Finalmente, a partir de la tabla resumen, podemos comprobar que el error estándar obtenido es bastante similar en los diferentes datasets, y varía entre el 5.55% hasta el 7,15%, centrándose sobre todo en el 6,5%. Este error obtenido es bastante aceptable con la memoria auxiliar utilizada y es muy significativo que su valor se mantenga bastante próximo entre los diferentes datasets.

1	Dataset	Estimacion media	Estimacion exacta	Error estandard	tiempo medio (ms)	tiempo medio elemento (micro secs)
2	D1	3166.7	3185	0.0566	13.1051	0.7611
3	D2	23421.78	23134	0.0554	116.8979	0.3042
4	D3	5889	5893	0.0656	27.1965	0.4395
5	D4	5675.54	5760	0.0656	18.2294	0.6686
6	D5	9550.41	9517	0.0715	55.2176	0.3318
7	D6	6281.76	6319	0.062	43.6907	0.3506
8	D7	8977.44	8995	0.0634	53.4495	0.3399
9	D8	491057.01	496317	0.0684	4970.3271	0.7048
10	D9	17670.76	17620	0.0648	174.7894	0.3038

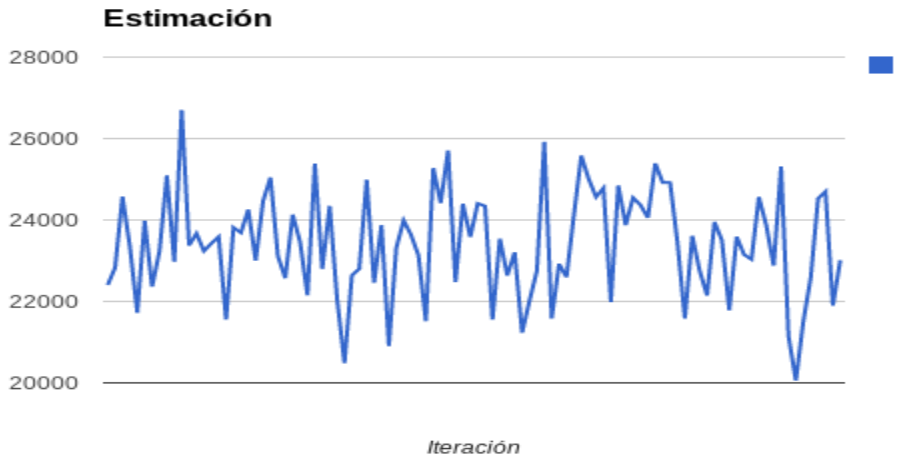
Dataset 1

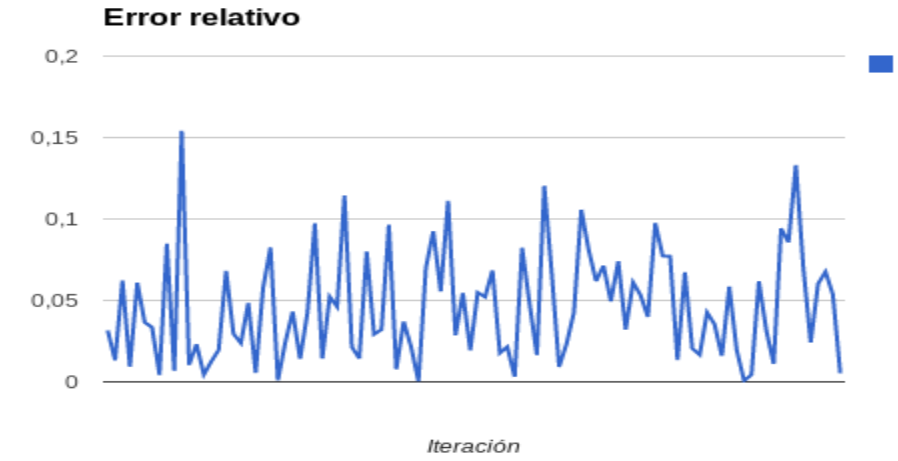

Estimación
4000
3600
3200
2800
2400
Iteración

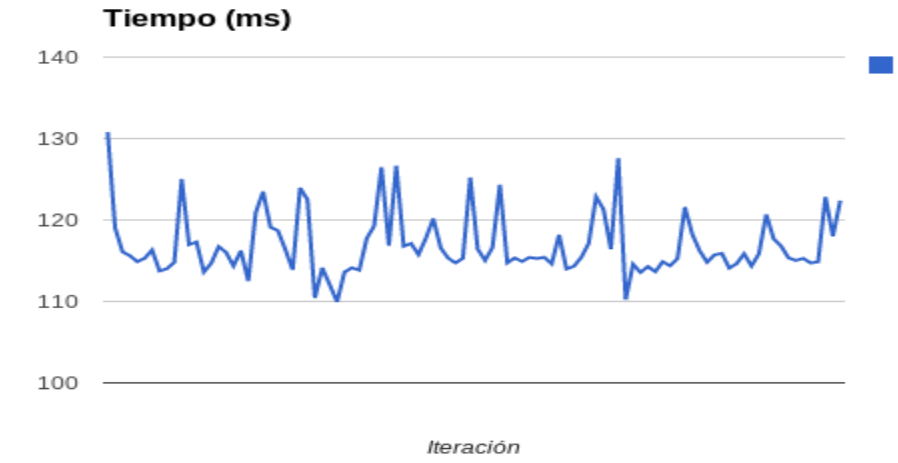

Error relativo
0,16
0,12
0,08
0,04
0
Iteración


Tiempo (ms)
20
17
14
11
8
Iteración

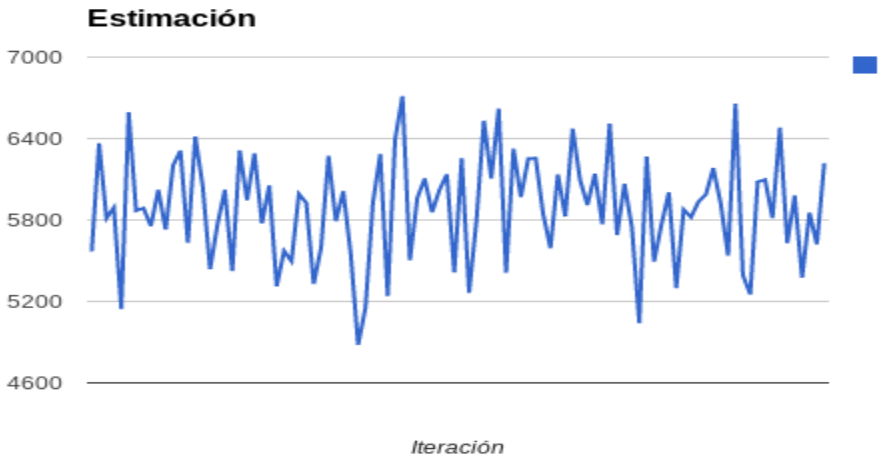
Dataset 2

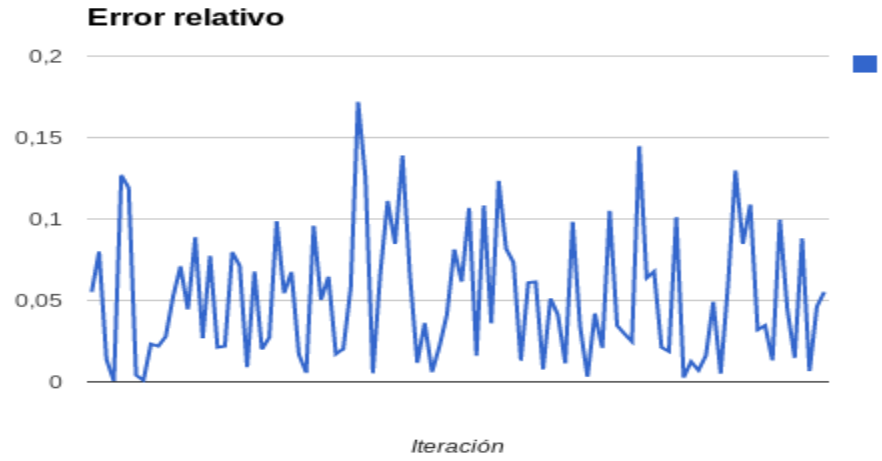

Estimación
28000
26000
24000
22000
20000
Iteración

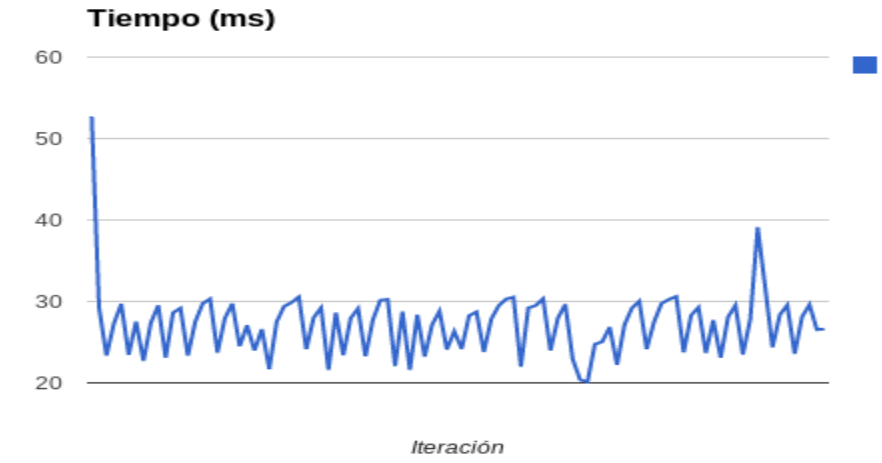

Error relativo
0,2
0,15
0,1
0,05
0
Iteración


Tiempo (ms)
140
130
120
110
100
Iteración

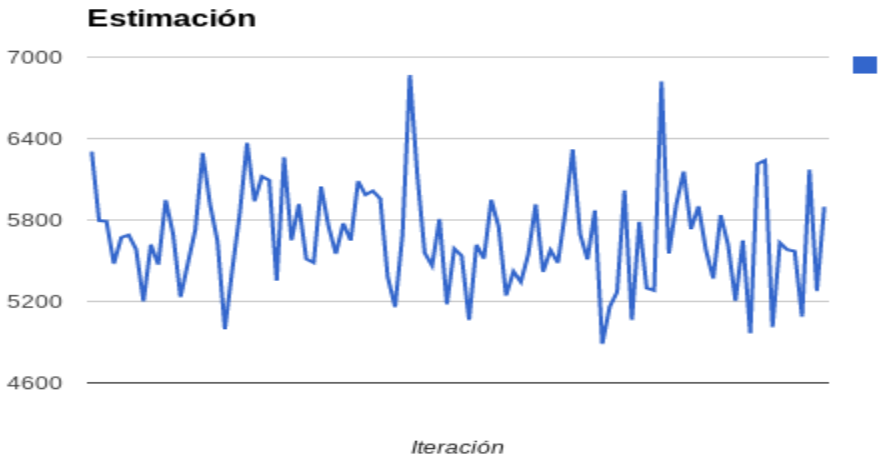
Dataset 3

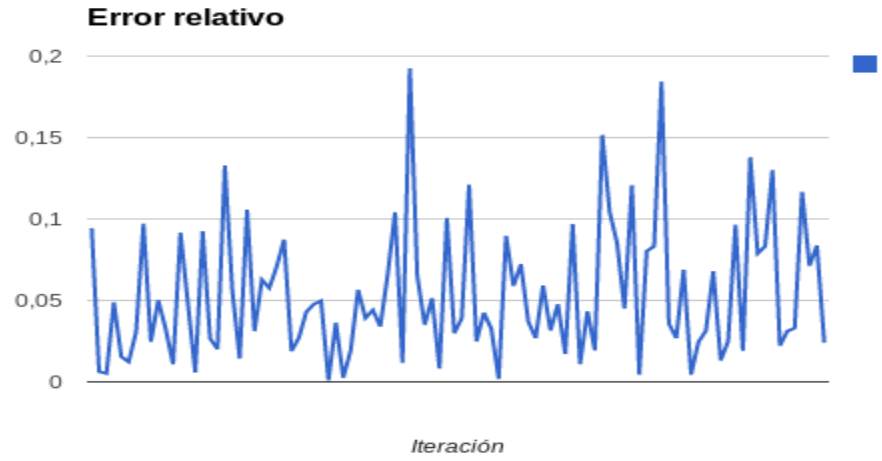

Estimación
7000
6400
5800
5200
4600
Iteración

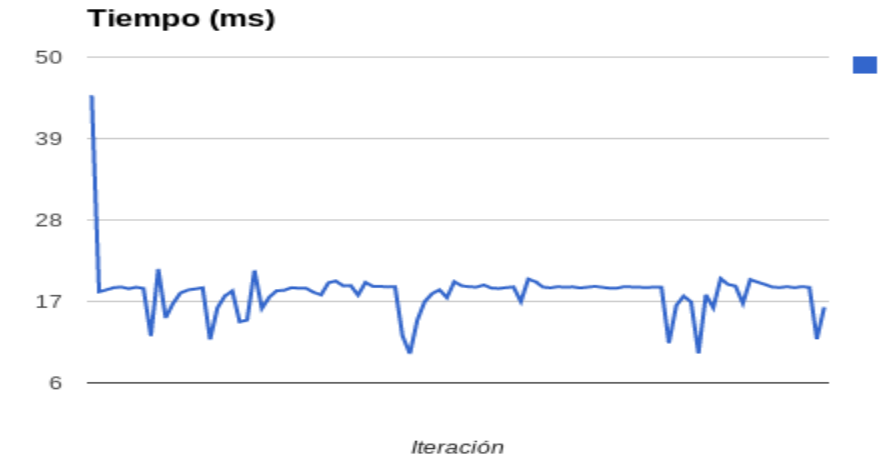

Error relativo
0,2
0,15
0,1
0,05
0
Iteración


Tiempo (ms)
60
50
40
30
20
Iteración

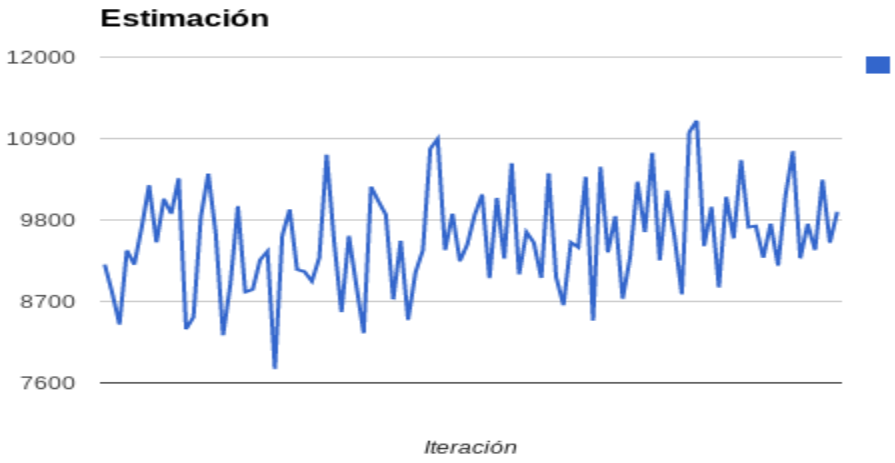
Dataset 4

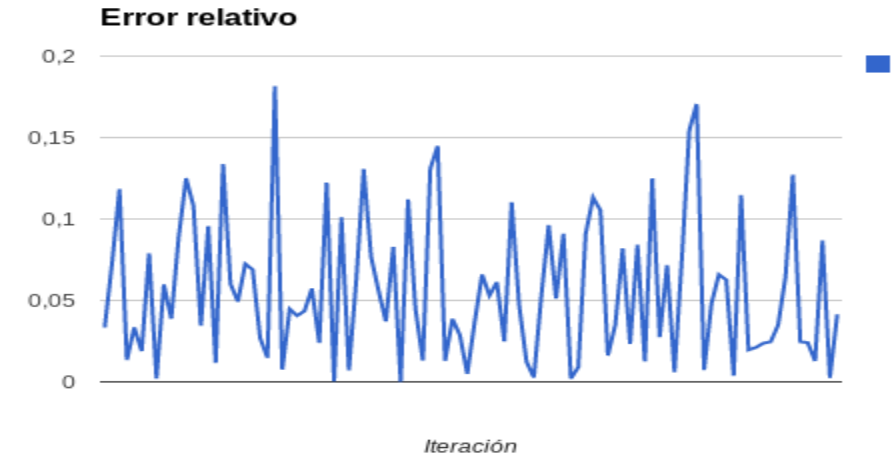

Estimación
7000
6400
5800
5200
4600
Iteración

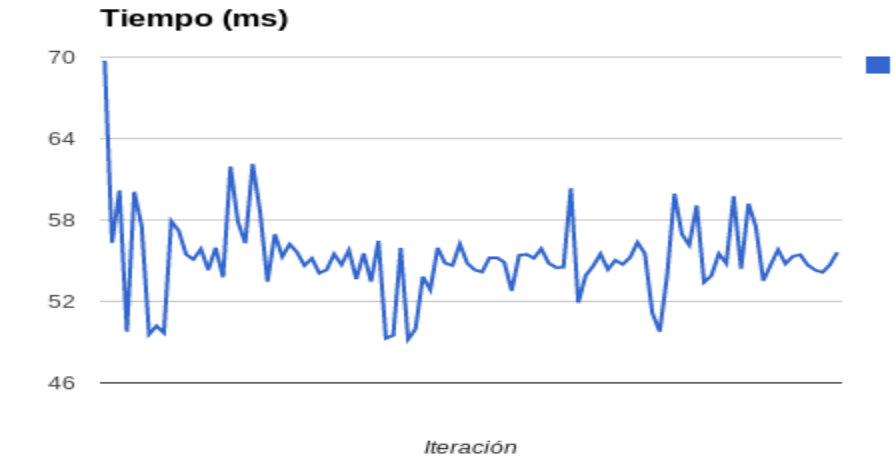

Error relativo
0,2
0,15
0,1
0,05
0
Iteración


Tiempo (ms)
50
39
28
17
6
Iteración

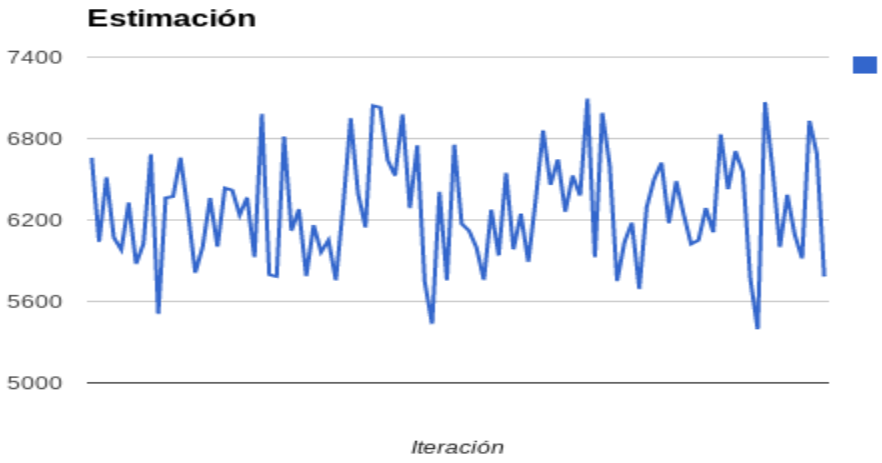
Dataset 5

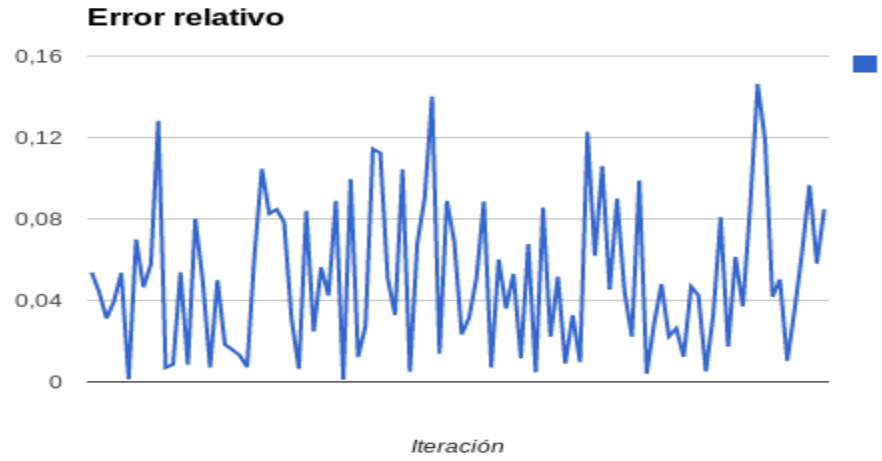

Estimación
12000
10900
9800
8700
7600
Iteración

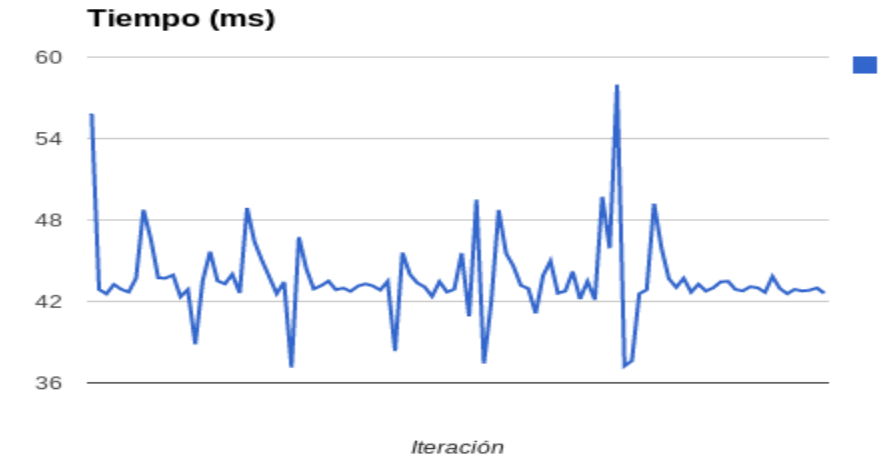

Error relativo
0,2
0,15
0,1
0,05
0
Iteración


Tiempo (ms)
70
64
58
52
46
Iteración

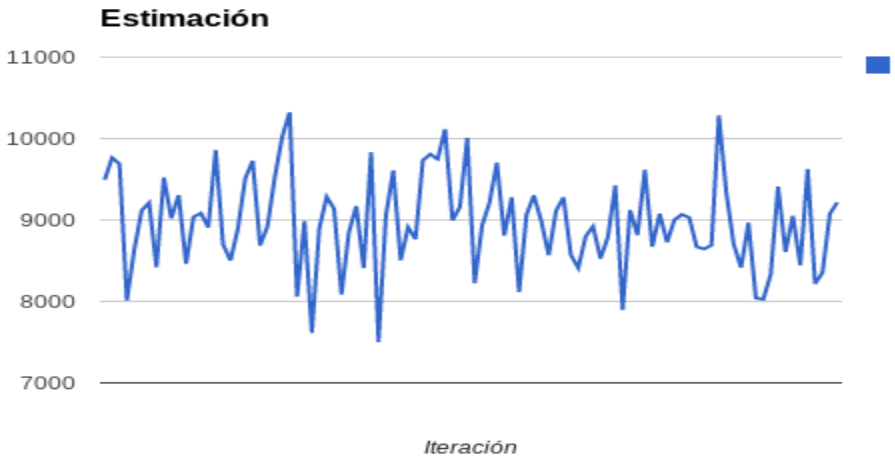
Dataset 6

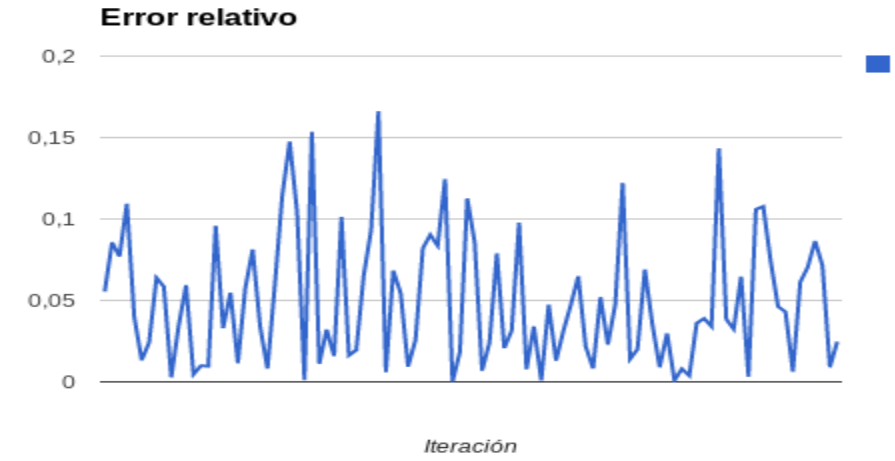

Estimación
7400
6800
6200
5600
5000
Iteración

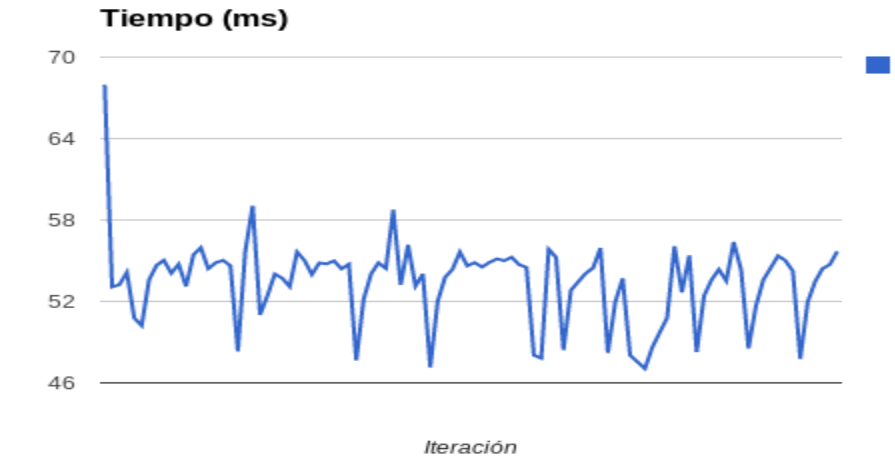

Error relativo
0,16
0,12
0,08
0,04
0
Iteración


Tiempo (ms)
60
54
48
42
36
Iteración

Dataset 7

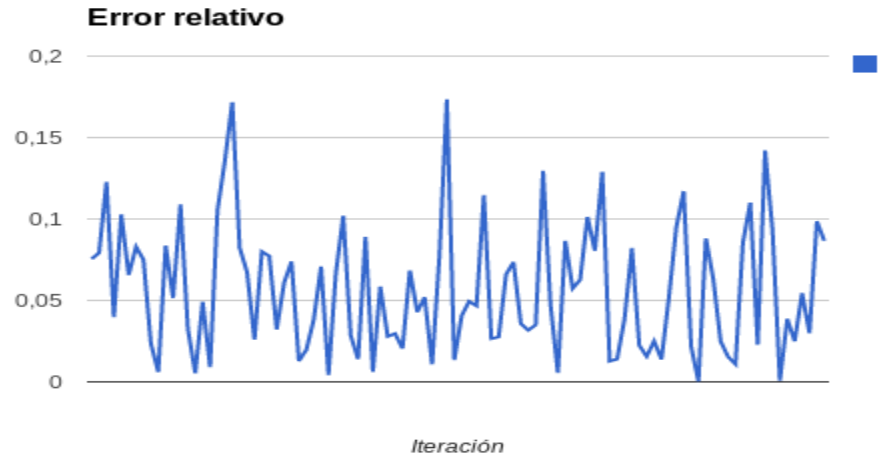

Estimación
11000
10000
9000
8000
7000
Iteración

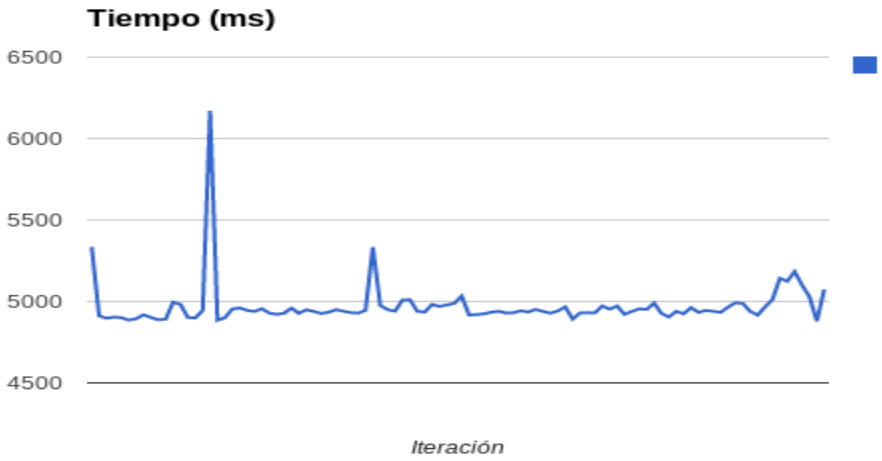

Error relativo
0,2
0,15
0,1
0,05
0
Iteración


Tiempo (ms)
70
64
58
52
46
Iteración

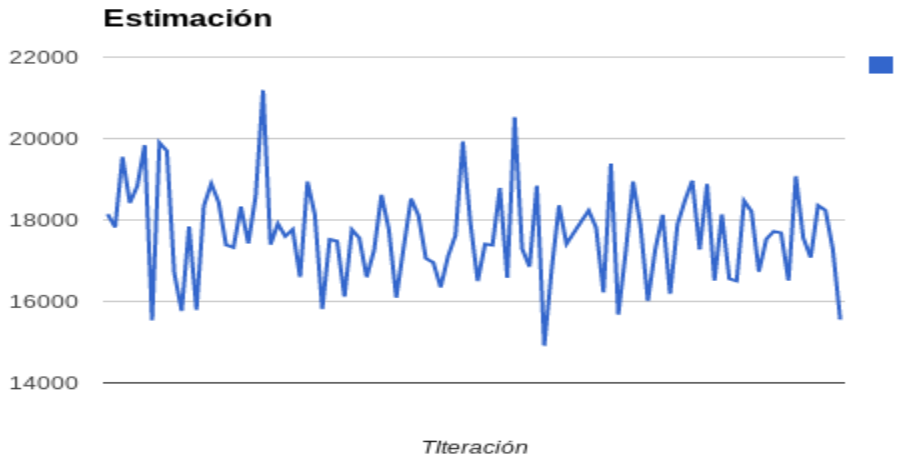
Dataset 8

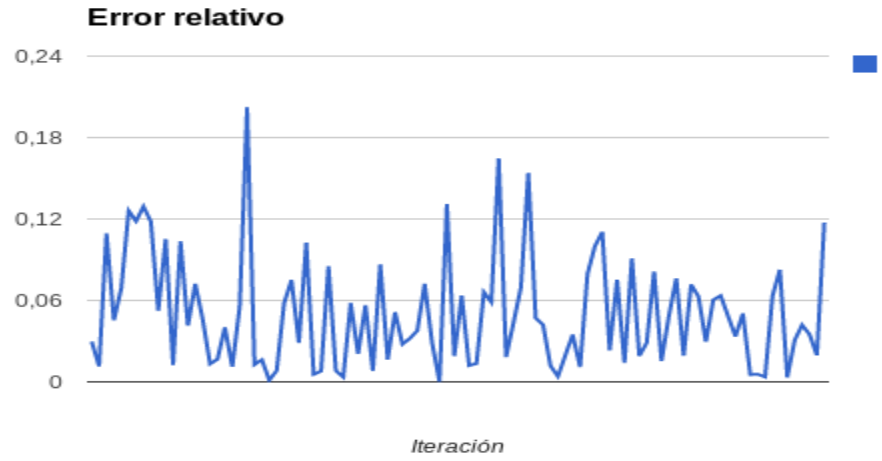

Estimación
600000
550000
500000
450000
400000
Iteración

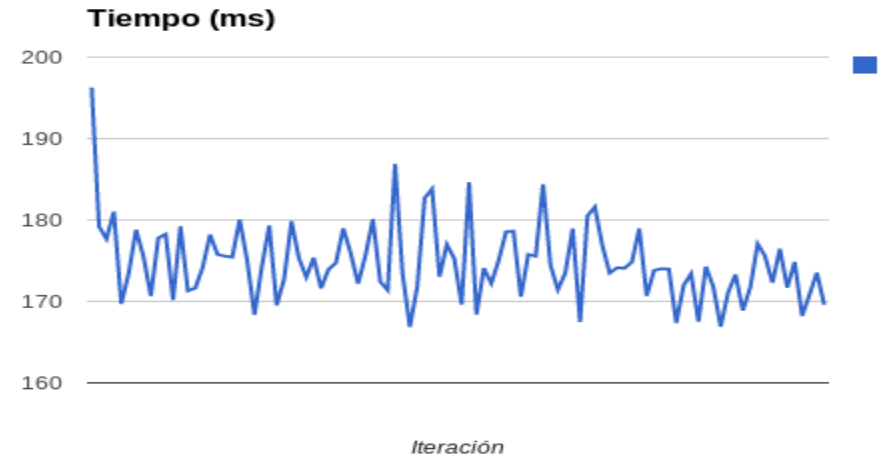

Error relativo
0,2
0,15
0,1
0,05
0
Iteración


Tiempo (ms)
6500
6000
5500
5000
4500
Iteración

Dataset 9


Estimación
22000
20000
18000
16000
14000
TIteración


Error relativo
0,24
0,18
0,12
0,06
0
Iteración


Tiempo (ms)
200
190
180
170
160
Iteración

Segundo experimento

Objetivo

El objetivo de este script es crear una tabla con el valor medio de las estimaciones, el error estándar, el tiempo medio de ejecución y el tiempo medio por elemento del primer dataset. No obstante, a diferencia del experimento anterior, el objetivo de éste es la de visualizar como se comportan los valores anteriores cuando se varía el tamaño de la memoria auxiliar. Es decir, se escogerá un tamaño inicial de la memoría y se irá incrementando éste, realizando un número determinado de ejecuciones por cada tamaño (hasta 10 tamaños diferentes).

Algoritmo

El funcionamiento de este script es prácticamente idéntico al primero. No obstante, en este script las ejecuciones se realizan siempre sobre el mismo dataset (el primero), con la única diferencia de que el tamaño de la memoria aumenta después de K ejecuciones con el mismo tamaño.

Inicialmente, el tamaño de la memoria es de 128 bytes (2^7), y en cada aumento se doblará el tamaño de ésta hasta alcanzar un límite, que serán 65536 bytes (2^{16}). Por lo tanto, se realizarán K ejecuciones con 10 tamaños diferentes de la memoria auxiliar.

Por otro lado, a diferencia del primer script, en este experimento no se guardan los mismos valores (tablas de las ejecuciones, etc), sino que solamente crea un fichero, de extensión 'csv', donde guarda los valores de la estimación media, el error estándar, el tiempo medio de ejecución y el tiempo medio por elemento en función del tamaño de la memoria auxiliar.

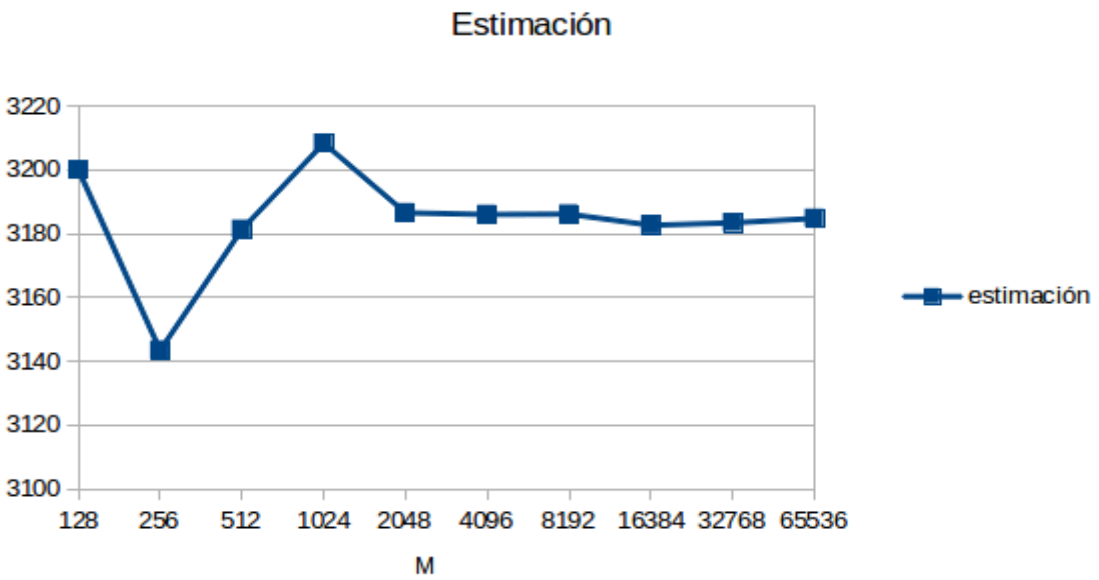
Resultados

Como se ha visto en el anterior experimento, el algoritmo logra unas estimaciones medias muy cercanas a los valores reales. No obstante, en este experimento podemos ver que al aumentar el tamaño de la memoria auxiliar las estimaciones medias tienen un error estándar cada vez menor, ya que al aumentar la memoria obtenemos unas estimaciones cada vez más exactas en comparación a los valores reales. Esto puede ser debido a que el número de colisiones en la tabla de hash sea menor, ya que al haber más espacio la probabilidad de colisión desciende y,

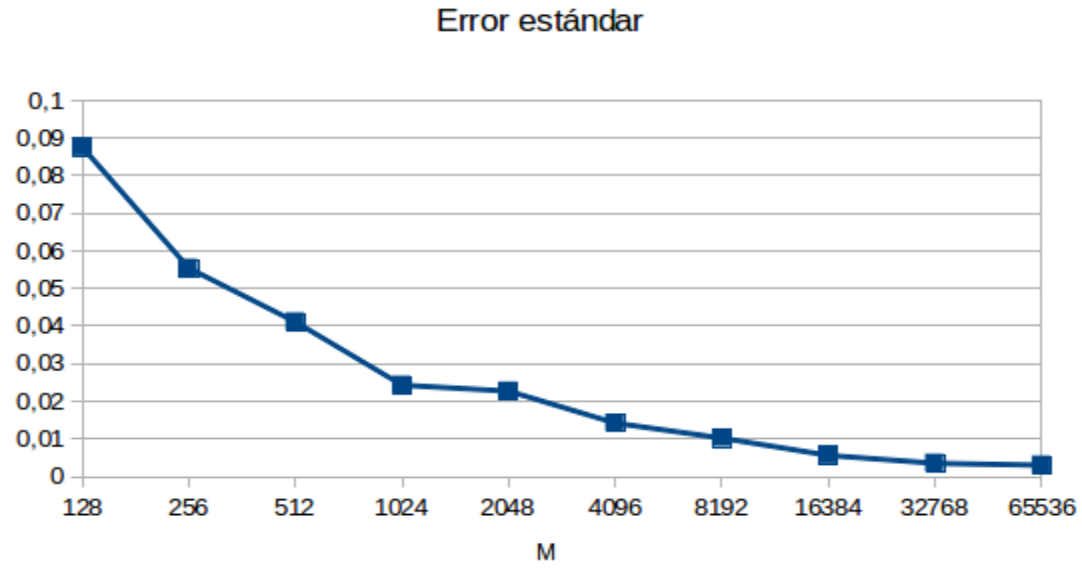
además, podemos guardar más valores para la estimación de la cardinalidad y lograr una mayor exactitud al tener guardadas más muestras de los elementos y no tener que descartarlas.

1	M	Estimacior	Error estandard	tiempo medio (ms)	tiempo medio elemento (micro secs)
2	128	3200.01	0.0876	12.204	0.7088
3	256	3143.4	0.0554	12.693	0.7372
4	512	3181.31	0.0411	12.887	0.7484
5	1024	3208.33	0.0243	11.999	0.6968
6	2048	3186.51	0.0227	12.685	0.7367
7	4096	3185.96	0.0142	13.1	0.7608
8	8192	3186.09	0.0102	12.823	0.7447
9	16384	3182.58	0.0056	13.192	0.7661
10	32768	3183.38	0.0035	13.863	0.8051
11	65536	3184.74	0.003	15.434	0.8963

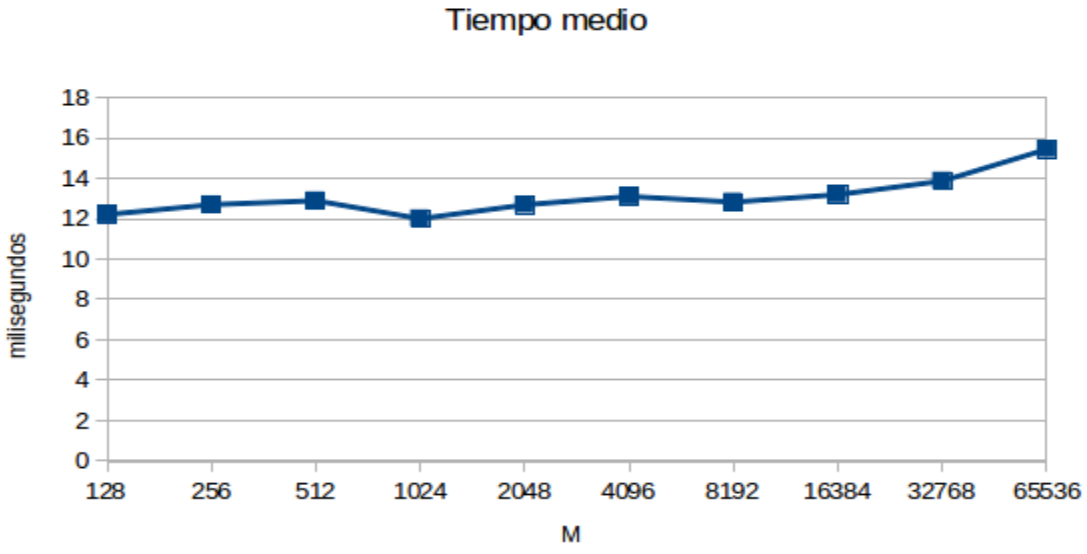
Como ejemplo de lo anteriormente descrito, observando la tabla se puede visualizar como al utilizar un tamaño mayor de memoria auxiliar la estimación se aproxima al valor exacto. Concretamente, el valor exacto de la cardinalidad de este dataset es de 3185, por lo que tenemos que a partir de una memoria auxiliar de 2048 posiciones la estimación es prácticamente exacta, dando un error estándar cada vez más bajo.



En la gráfica del error estándar, podemos confirmar que la estimación tiende al valor de la cardinalidad exacta. El error estándar conseguido con una memoria auxiliar de 65536 bytes es de 0.003, es decir, del 0.03%, mientras que con una memoria de 128 bytes es del 8.76%. Es muy significativa esta reducción, porque nos indica que la desviación obtenida al calcular la cardinalidad es mínima, y que el algoritmo obtiene excelentes resultados con memorias de un tamaño mayor.



Podemos observar en el siguiente gráfico que el tiempo medio de ejecución no varía significativamente según el tamaño de la memoria auxiliar. No obstante, si que podemos observar un ligero incremento del tiempo de ejecución al aumentar el tamaño de la memoria.



Conclusiones

A partir de la información buscada y de los resultados de los experimentos efectuados, hemos podido observar la utilidad que tienen este tipo de algoritmos en unos escenarios específicos. Es decir, son algoritmos necesarios cuando la cantidad de memoria auxiliar es crítica, por lo que podemos deducir que lo que realmente se busca de estos algoritmos es que sepan trabajar con este tipo de limitaciones.

Personalmente, hemos elegido utilizar el HyperLogLog precisamente porque se adapta muy bien a este tipo de escenarios, realizando unas estimaciones muy buenas con una cantidad de memoria mucho menor que otros algoritmos. Concretamente, es este aspecto del algoritmo el que más nos ha llamado la atención, ya que hemos encontrado algunos que tenían una mejor precisión en la estimación pero utilizaban una cantidad de memoria mayor.

Todo esto se puede observar a partir de los resultados del primer experimento, donde se puede apreciar que para diferentes conjuntos de datos el error estándar obtenido es bastante similar. Por lo tanto, a pesar de las diferencias de tamaño tanto de las secuencias como de los elementos de éstas, las estimaciones son muy próximas a la cardinalidad exacta de cada secuencia.

Por otro lado, viendo los resultados del segundo experimento, podemos concluir que al aumentar la memoria auxiliar, la calidad de las estimaciones también aumenta. No obstante, la principal característica de este algoritmo es la capacidad de poder obtener unas estimaciones con un error relativo aceptable sin necesitar casi memoria auxiliar, por lo que su verdadero potencial se exprime cuando tenemos una memoria mucho menor a la cardinalidad real.

Finalmente, otro aspecto a destacar de este tipo de algoritmos es a partir de qué información realizan las estimaciones. A raíz de la investigación y de la implementación del algoritmo, hemos podido ver como este tipo de algoritmos funcionan a partir de la aleatoriedad proporcionada por la función de hash, ya que extraen la información de la cardinalidad del conjunto de datos a partir del *hashvalue* obtenido. Más concretamente, en nuestro caso utilizaban la cantidad de ceros iniciales del *hashvalue* para estimar la cardinalidad del conjunto. Gracias a esto, la información que se requiere guardar se puede codificar con 5 bits, ya que la máxima cantidad de ceros posibles en el *hashvalue* es de 32.

Bibliografía

[1] Stefan Heule, Marc Nunkesser y Alexander Hall.
HyperLogLog in Practice: Algorithmic Engineering of a State of the Art Cardinality Estimation Algorithm.
(**http://stefanheule.com/papers/edbt2013-hyperloglog.pdf**)

[2] Philippe Flajolet, Éric Fusy, Oliver Gandouet y Frédéric Meunier.
HyperLogLog: the analysis of a near-optimal cardinality estimation algorithm.
(**http://algo.inria.fr/flajolet/Publications/FlFuGaMe07.pdf**)

[3] Marianne Durand y Philippe Flajolet.
Loglog Counting of Large Cardinalities.
(**http://algo.inria.fr/flajolet/Publications/DuFl03-LNCS.pdf**)

[4] Daniel M. Kane, Jelani Nelson y David P. Woodruff.
An Optimal Algorithm for the Distinct Elements Problem.
(http://researcher.watson.ibm.com/researcher/files/us-dpwoodru/knw11.pdf)

[5] Josh Poley (Microsoft Corporation).
Magic Numbers: Integers.
(**http://msdn.microsoft.com/en-us/library/ee621251.aspx**)

[6] Wikipedia.
Universal Hashing (Hashing strings).
(http://en.wikipedia.org/wiki/Universal_hashing#Hashing_strings)

[7] Daniel Lemire y Owen Kaser.
Strongly universal string hashing is fast.
(http://arxiv.org/pdf/1202.4961.pdf)

[8] Kevin Wayne.
Randomized Algorithms slides.
(http://www.cs.princeton.edu/~wayne/kleinberg-tardos/pdf/13RandomizedAlgorithms.pdf)

[9] Mikkel Thorup y Yin Zhang.
Tabulation Based 4-Universal Hashing with Applications to Second Moment Estimation.

(http://www.cs.utexas.edu/~yzhang/papers/hash-soda04.pdf)

[10] Python Software Foundation.

Python v2.7.6 documentation.

(http://docs.python.org/2/)

Apéndices

Código del programa *cardest*

```
#include <iostream>
#include <vector>
#include <string.h>
#include <stdlib.h>
#include <stdio.h>
#include <time.h>
#include <math.h>
using namespace std;

// Durante todo el algoritmo trabajaremos con numeros naturales
typedef unsigned char u_char;
typedef unsigned int u_int;

// Constante que se aplica varias veces a la hora de realizar la estimacion
static const double pow_2_32 = 4294967296.0;

// Numero primo mas grande de 32 bits
u_int p = 4294967291;
// Tabla para guardar el numero de "leading zeros"
vector<u_char> M;
// Vector para guardar los numeros aleatorios generados para la funcion de hash
vector<u_int> a;

/*
 * La funcion initialization se encarga de inicializar las variables
 * necesarias para el algoritmo. Concretamente, realiza cuatro
 * inicializaciones diferentes:
 *      1. Inicializa la variable alpha segun el valor de m
 *      2. Inicializa las dos tablas con las posiciones correspondientes.
 *      3. Inicializa la variable b con el numero de bits necesario para
 *         codificar el total de posiciones de la tabla M.
 *      4. Genera 30 numeros aleatorios que son guardados en el vector
 *         a para su utilizacion en la funcion de hash.
 */
void initialization(int m, double &alpha, int &b) {
    // Inicializacion de alpha
    switch(m) {
        case 16:
            alpha = 0.673;
            break;
        case 32:
            alpha = 0.697;
            break;
        case 64:
            alpha = 0.709;
            break;
        default:
            alpha = 0.7213/(1.0 + 1.079/m);
    }

    // Inicializacion de M
    M = vector<u_char> (m, 0);

    // Inicializacion de a
```

```
    a = vector<u_int> (30);

    // Inicializacion de b
    b = 0;
    while(m > 1) {
        b++;
        m = m >> 1;
    }

    // La semilla de la funcion random se obtiene a partir del tiempo del procesador,
    // asi obtenemos numeros aleatorios diferentes por cada ejecucion
    srand(time(NULL));

    // Generacion de los 30 numeros aleatorios (cualquier string de la entrada
    // tiene un tama単o como maximo de 30 caracteres)
    for(int i = 0; i < 30; i++) {
        a[i] = rand();
    }
}

/*
 * La funcion universal_hash recibe como parametro un string y devuelve un hashvalue
 * de 32 bits. Para que la funcion de hash sea universal se utiliza un vector que
 * contiene 30 numeros generados aleatoriamente, uno por cada caracter posible que
puede
 * tener el string recibido por parametro. Ademas, por este motivo, tambien se utiliza
 * un numero primo a la hora de realizar el modulo sobre el hashvalue. De esta manera,
 * se consigue una probabilidad de colision de 1/m, que es considerablemente peque単a.
 */
u_int universal_hash(string key) {
    u_int hash_value = 0;

    // Se calcula el hashvalue a partir de la suma de las multiplicaciones entre
    // las mismas posiciones de los vectores a y key
    for(int i = 0; i < key.size(); i++)
        hash_value += ((u_int) key[i])*a[i];

    // Se le aplica al hashvalue el modulo del numero primo mas grande de 32 bits
    hash_value = hash_value % p;

    return hash_value;
}

/*
 * La funcion leading_zeros recibe como parametro un valor de 32 bits y la posicion
 * inicial desde donde hacer la comprobacion. Por lo tanto, a partir de estos
elementos,
 * la funcion devuelve la posicion del primer bit a 1 cogiendo como posicion inicial
 * la recibida por parametro.
 */
u_char leading_zeros(u_int w, int b) {
    // La variable zeros se inicializa con 1 porque hay que devolver
    // la posicion del primer bit a 1.
    u_char zeros = 1;
    int i = 31-b;

    bool finish = false;
    while(i >= 0 && !finish) {
        if((w >> i) & 1)
            // Si el bit a comprobar esta a 1, se finalizara el bucle
            finish = true;
```

```
        else
            // En caso contrario se ha hayado un cero mas, por lo tanto, se suma uno a
la variable
            zeros++;

        i--;
    }

    return zeros;
}

/*
 * La funcion aggregation se encarga de la insercion de los elementos. Por cada
elemento,
 * se efectuan las siguientes acciones:
 *      1. Se le aplica la funcion de hash al elemento para obtener el hashvalue.
 *      2. A partir del hashvalue, se obtiene la posicion del elemento en la tabla
 *         utilizando los primeros b bits de este.
 *      3. Seguidamente, con el resto de bits se le aplica la funcion leading_zeros
 *         para obtener la posicion del primer bit a 1.
 *      4. A partir del resultado anterior, si este es mayor que el que esta guardado
 *         en su posicion de la tabla, se sobrescribe dicha posicion.
 *      5. Se suma 1 al contador que cuenta el total de palabras del conjunto.
 */
u_int aggregation(int b) {
    // Inicializacion de las variables
    u_int x, idx, w, count;
    string word;

    count = 0;

    // Por cada linea de la entrada se obtiene un nuevo elemento
    while(getline(cin, word)) {
        // Aplicacion de la funcion de hash al elemento
        x = universal_hash(word);
        // Obtencion de la posicion en la tabla M
        idx = x >> (32 - b);

        // Aplicacion de la funcion leading_zeros
        u_char lz = leading_zeros(x, b);

        if(M[idx] < lz) {
            // Si el numero de leading_zeros es mayor que el valor guardado, se
sobrescribe
            M[idx] = lz;
        }

        // Se cuenta una palabra mas al total de palabras
        count++;
    }

    return count;
}

/*
 * La funcion linear_counting
 */
double linear_counting(int m, int v) {
    return (m * log((double) m/(double) v));
}
```

```
/*
 * La funcion computation se encarga de realizar la estimacion de la
 * cardinalidad a partir de los valores guardados en la tabla M. Ademas,
 * una vez calculada la estimacion, si esta se encuentra en un rango
 * muy peque単o o, al contrario, se encuentra en un rango muy grande,se efectua
 * una correccion de esta estimacion.
 */
double computation(int m, double alpha) {
    double E;

    // Calculo de un componente de la estimacion
    double sum = 0.0;
    for(int i = 0; i < m; i++) {
        sum += 1.0/pow(2.0, M[i]);
    }

    // Calculo de la estimacion
    E = (alpha*m*m)/sum;

    if(E <= (5*m)/2) {                    // Small range correction
        // Se cuentan el numero de registros de la tabla M que sean iguales a 0,
        // es decir, que no han sido modificados
        int v = 0;
        for(int i = 0; i < m; i++) {
            if(M[i] == 0)
                v++;
        }

        if(v > 0) {
            // Si existe algun registro a 0 en la tabla M,
            // se utiliza la estimacion de la funcion linear_counting
            return linear_counting(m, v);
        } else {
            // En caso contrario, se utiliza la estimacion inicial
            return E;
        }
    } else if(E <= (pow_2_32/30.0)) {    // Intermediate range (no correction)
        return E;
    } else {                             // Large range correction
        return ((-pow_2_32) * log(1 - E/pow_2_32));
    }
}

int main(int argc, char * argv[]) {
    // Inicializacion de m, con un valor por defecto de 256
    int m = 256;
    int b;
    double alpha;

    // Comprobacion del flag -M
    if(argc == 3) {
        if(strcmp(argv[1], "-M") == 0) {
            // Si se ha utilizado el flag, la variable m obtiene el valor
            // que ha sido pasado por parametro
            m = atoi(argv[2]);
        }
    }

    initialization(m, alpha, b);

    u_int N = aggregation(b);
```

```
    u_int E = computation(m, alpha);

    cout << E << " " << N << endl;
}
```

Código del primer script

```
import os
import subprocess
import math
import time
import csv
import sys
from time import sleep



global iterador



def convertirEnLista(argument):                 # busca los dos valores del string que se le da, utilizando como
  i=0;                                  # marca el espacio
  lista = []
  while (argument[i] != ' '):                 # busca la posicion de la marca
     i = i+1
  aux = len(argument)                       # largo del argumento
  element1 = argument[0:i]                    # obtiene el primer elemento
  element2 = argument[i+1:aux-1]                  # obtiene el segundo elemento
  element1 = float(element1)                  # el element1 es un string, lo pasamos a int
  element2 = float(element2)                  # el element2 es un string, lo pasamos a int
  return [element1,element2]


def procesarDataset(datasetTarget, k, n, N):

  tiempoAcomulado = 0
  # definimos la variable tiempoAcomulado, que se utilizara para sumar los tiempos

  estimacionAcumulada = 0
  # definimos la variable estimacionAcumulada, que se utilizara para sumar las estimaciones

  estimacionAcumuladaCuadrado = 0
  # definimos la variable estimacionAcumuladaCuadrado, que se utilizara para sumar el cuardado de las
estimaciones

  datasetfile  = open('Tablas Datasets/'+datasetTarget+"_Tabla.csv", "wb")
  # abrimos o creamos un fichero csv, donde escribiremos la tabla de resultados

  datasetWriter = csv.writer(datasetfile)              # creamos un writer para el fichero

  datasetWriter.writerows([['i','Estimacion','Error relativo','tiempo (ms)']])
  # escribimos la cabecera de la tabla

  # inicializamos las variables que usaremos para contar el numero de datasets que esta en cada franja de %
de errores
  errorRelativoMenor1 = 0
  errorRelativoMenor1y5 = 0
  errorRelativoMenor5y10 = 0
```

```
  errorRelativoMenor10y15 = 0
  errorRelativoMenor15y25 = 0
  errorRelativoMenor25y40 = 0
  errorRelativoMenor40y55 = 0
  errorRelativoMenor55y70 = 0
  errorRelativoMenor70y85 = 0
  errorRelativoMenor85y100 = 0
  i = 1
  while (i <= k):
     sleep(1)
     # entre ejecucion y ejecucion, hacemos que se pause durante 1 segundo, para asegurar que los numeros
random
     # escogidos en el algoritmo sea realmente aleatorios (dichos numeros se basan en el tiempo del
procesador,
     # por eso al dejar el margen de 1 segundo, nos aseguramos de que la semilla del generador ha variado)


     ejecucion = "./exe < all/"+datasetTarget+".dat"     # comando para ejecutar

     tiempoIni = time.time()
     resultado = subprocess.check_output(ejecucion, shell=True)     # ejecutamos el programa pasandole el
dataset
     tiempoFin = time.time()

     tiempoTotal = tiempoFin - tiempoIni              # calculamos el tiempo de ejecucion
     tiempoTotal = tiempoTotal*1000                  # pasamos a milisegundos
     tiempoTotal = round(tiempoTotal, 4)              # truncamos a 4 digitos por detras de la coma


     tiempoAcomulado += tiempoTotal                  # calculamos el tiempo acomulado de ejecuciones en ms
     tiempoAcomulado = round(tiempoAcomulado, 4)       # truncamos a 4 digitos por detras de la coma

     [estimacion,numeroTotalElementosLeidos] = convertirEnLista(resultado)
     # obtenemos la estimacion y el numeroTotalElementosLeidos

     estimacionAcumulada += estimacion                # sumamos a la estimacionAcumulada
     estimacionAcumuladaCuadrado += estimacion*estimacion    # sumamos el cuadrado de la estimacion a
su acumulada

     errorRelativo = float(abs((1.0*n)-(1.0*estimacion))/(1.0*n))
     errorRelativo = round(errorRelativo, 4)
     # calculamos el error relativo y truncamos a 4 digitos por detras de la coma

     info = [i,estimacion,errorRelativo,tiempoTotal]

     # preparamos una lista, llamada 'info' para escribirla en la tabla

     datasetWriter.writerows([info])                 #escribimos la fila en la tabla
     if errorRelativo < 0.01 :
        errorRelativoMenor1 = errorRelativoMenor1+1
     elif errorRelativo < 0.05 :
        errorRelativoMenor1y5 = errorRelativoMenor1y5+1
     elif errorRelativo < 0.1:
        errorRelativoMenor5y10 = errorRelativoMenor5y10+1
     elif errorRelativo < 0.15 :
```

```
            errorRelativoMenor10y15 = errorRelativoMenor10y15+1
        elif errorRelativo < 0.25 :
            errorRelativoMenor15y25 = errorRelativoMenor15y25+1
        elif errorRelativo < 0.40 :
            errorRelativoMenor25y40 = errorRelativoMenor25y40+1
        elif errorRelativo < 0.55 :
            errorRelativoMenor40y55 = errorRelativoMenor40y55+1
        elif errorRelativo < 0.70 :
            errorRelativoMenor55y70 = errorRelativoMenor55y70+1
        elif errorRelativo < 0.85 :
            errorRelativoMenor70y85 = errorRelativoMenor70y85+1
        elif errorRelativo < 1.0 :
            errorRelativoMenor85y100 = errorRelativoMenor85y100+1

        i = i+1

    datasetfile.close()                         # cerramos el fichero csv

    estimacionMedia = round(estimacionAcumulada/k, 4)
    # calculamos la estimacion media y truncamos a 4 digitos por detras de la coma

    errorEstandard = round(math.sqrt((estimacionAcumuladaCuadrado/k)-
(estimacionMedia*estimacionMedia))/n, 4)
    # calculamos el error estandard y truncamos a 4 digitos por detras de la coma

    tiempoMedio = tiempoAcomulado/(k*1.0)              # calculamos el tiempo medio (en milisegundos)
    tiempoMedio = round(tiempoMedio, 4)               # truncamos a 4 digitos por detras de la coma

    tiempoMedioElemento = float(tiempoMedio*1000/(N*1.0))    # calculamos el tiempo medio por elemento (en
microsegundos)
    tiempoMedioElemento = round(tiempoMedioElemento, 4)     # truncamos a 4 digitos por detras de la coma


    print 'Error relativo de la estimacion menor del 1%: ' , errorRelativoMenor1
    print 'Error relativo de la estimacion entre el 1% y el 5%: ' , errorRelativoMenor1y5
    print 'Error relativo de la estimacion entre el 5% y el 10%: ' , errorRelativoMenor5y10
    print 'Error relativo de la estimacion entre el 10% y el 15% ' , errorRelativoMenor10y15
    print 'Error relativo de la estimacion entre el 15% y el 25% ' , errorRelativoMenor15y25
    print 'Error relativo de la estimacion entre el 25% y el 40% ' , errorRelativoMenor25y40
    print 'Error relativo de la estimacion entre el 40% y el 55% ' , errorRelativoMenor40y55
    print 'Error relativo de la estimacion entre el 55% y el 70% ' , errorRelativoMenor55y70
    print 'Error relativo de la estimacion entre el 70% y el 85% ' , errorRelativoMenor70y85
    print 'Error relativo de la estimacion entre el 85% y el 100% ' , errorRelativoMenor85y100

    print ' '


    global tablaResumen

tablaResumen.append([datasetTarget,estimacionMedia,errorEstandard,tiempoMedio,tiempoMedioElemento
])



# Main
```

```
if len(sys.argv) == 2:
      k = sys.argv[1]                    # definimos el numero de iteraciones
      k = int(k)
else:
  print "Parametros incorrectos! Introduzca un valor de K"
  sys.exit()


print '-- Inicio del Script --\n'

tiempoIniEje = time.time()

compilar = "g++ -o exe cardest.cc"

error = subprocess.call(compilar, shell=True)          # compilamos el programa
if error == 0:
  print '--Compilacion completada correctamente--'
else:
  print '--Compilacion fallida--'
  sys.exit()

try:
  os.makedirs('./Tablas Datasets')          # creamos la carpeta donde pondremos las tablas
except OSError:
  pass                              # si llegamos al error, no hacemos nada (existe la carpeta)

print '--Creacion de carpeta destino de tablas completada--\n'

tablaResumen = []

n = [3185,23134,5893,5760,9517,6319,8995,496317,17620]
# cardinalidad exacta del dataset

N = [17219,384222,61882,27266,166428,124623,157262,7052495,575284]
# numero total de elementos de los datasets

i = 1                               # la 'i' se utiliza para iterar entre los datasets
while (i<10):
  print ' '
  print 'Inicio de la ejecucion del dataset',i
  data = 'D'+str(i)                     # creamos el nombre del dataset
  procesarDataset(data, k, n[i-1], N[i-1])          # llamamos a la funcion que procesara dicho dataset
  print 'Dataset',i,' ejecutado correctamente\n'
  i = i+1


datasetfile  = open('Tablas Datasets/Tabla_Resumen.csv', "wb")
# abrimos o creamos un fichero csv, donde escribiremos la tabla de resumen

datasetWriter = csv.writer(datasetfile)             # creamos un writer para el fichero
datasetWriter.writerows([['Dataset','Estimacion','Error estandard','tiempo medio (ms)', 'tiempo medio
elemento (micro secs)']])

for row in tablaResumen:                        # recorremos la tabla de resumen
```

```
    datasetWriter.writerows([row])              # escribimos la informacion del dataset
datasetfile.close()                     # cerramos el fichero


tiempoFinEje = time.time()
tiempoTotalEje = tiempoFinEje-tiempoIniEje
print 'Tiempo total de ejecucion del script: ', int(tiempoTotalEje/60), ' minutos y ',int(tiempoTotalEje%60), ' segundos'
```

Código del segundo script

```
import os
import subprocess
import math
import time
import csv
import sys
from time import sleep


def convertirEnLista(argument):
  i=0;
  lista = []
  while (argument[i] != ' '):
     i = i+1
  aux = len(argument)
  element1 = argument[0:i]
  element2 = argument[i+1:aux-1]
  [element1,element2]
  return [element1,element2]


def procesarDataset(datasetTarget, k, n, N, M, datasetWriter):
  tiempoAcomulado = 0
  # definimos la variable tiempoAcomulado, que se utilizara para sumar los tiempos

  estimacionAcumulada = 0
  # definimos la variable estimacionAcumulada, que se utilizara para sumar las estimaciones

  estimacionAcumuladaCuadrado = 0
  # definimos la variable estimacionAcumuladaCuadrado, que se utilizara para sumar el cuardado de las
estimaciones

  i = 1
  while (i <= k):
     sleep(1)
     # entre ejecucion y ejecucion, hacemos que se pause durante 1 segundo, para asegurar que los
numeros random
     # escogidos en el algoritmo sea realmente aleatorios (dichos numeros se basan en el tiempo del
procesador,
     # por eso al dejar el margen de 1 segundo, nos aseguramos de que la semilla del generador ha variado)

     ejecucion = "./exe -M "+str(M)+" < all/D1.dat"                # comando para ejecutar

     tiempoIni = time.time()

     resultado = subprocess.check_output(ejecucion, shell=True)     # ejecutamos el programa pasandole el
dataset
     tiempoFin = time.time()

     tiempoTotal = tiempoFin - tiempoIni             # calculamos el tiempo de ejecucion

     tiempoTotal = round(tiempoTotal, 4)             # truncamos a 4 digitos por detras de la coma
```

```
        tiempoTotal = tiempoTotal*1000                  # pasamos a milisegundos
        tiempoAcomulado += tiempoTotal                  # calculamos el tiempo acomulado de ejecuciones en ms

        tiempoAcomulado = round(tiempoAcomulado, 4)     # truncamos a 4 digitos por detras de la coma

        [estimacion,numeroTotalElementosLeidos] = convertirEnLista(resultado)
        # obtenemos la estimacion y el numeroTotalElementosLeidos

        estimacion = float(estimacion)                  # la estimacion esta en string, la pasamos a int

        estimacionAcumulada += estimacion               # sumamos a la estimacionAcumulada

        estimacionAcumuladaCuadrado += estimacion*estimacion
         # sumamos el cuadrado de la estimacion a su acumulada

        errorRelativo = float(abs((1.0*n)-(1.0*estimacion))/(1.0*n))

        errorRelativo = round(errorRelativo, 4)
        # calculamos el error relativo

        i = i+1


    estimacionMedia = round(estimacionAcumulada/k, 4)
    # calculamos la estimacion media y truncamos a 4 digitos por detras de la coma


    errorEstandard = round(math.sqrt((estimacionAcumuladaCuadrado/k)-
(estimacionMedia*estimacionMedia))/n, 4)
    # calculamos el error estandard y truncamos a 4 digitos por detras de la coma

    tiempoMedio = tiempoAcomulado/(k*1.0)               # calculamos el tiempo medio (en milisegundos)

    tiempoMedio = round(tiempoMedio, 4)                 # truncamos a 4 digitos por detras de la coma

    tiempoMedioElemento = float(tiempoMedio*1000/(N*1.0))   # calculamos el tiempo medio por elemento (en
microsegundos)

    tiempoMedioElemento = round(tiempoMedioElemento, 4)     # truncamos a 4 digitos por detras de la coma


    info = [M,estimacionMedia,errorEstandard,tiempoMedio,tiempoMedioElemento]
    # preparamos una lista, llamada 'info' para escribirla en la tabla


    datasetWriter.writerows([info])                 #escribimos la fila en la tabla


# Main
if len(sys.argv) == 2:
        k = sys.argv[1]                         # definimos el numero de iteraciones
        k = int(k)
else:
    print "Parametros incorrectos! Introduzca un valor de K"
```

```
  sys.exit()


print '-- Inicio del Script --\n'

tiempoIniEje = time.time()

compilar = "g++ -o exe cardest.cc"
subprocess.call(compilar, shell=True)
datasetTarget = 'dataset1'
datasetfile  = open(datasetTarget+"_Tabla_base_M.csv", "wb")
# abrimos o creamos un fichero csv, donde escribiremos la tabla de resultados

datasetWriter = csv.writer(datasetfile)              # creamos un writer para el fichero

datasetWriter.writerows([['M','Estimacion','Error estandard','tiempo medio (ms)', 'tiempo medio elemento
(micro secs)']])
# escribimos la cabecera de la tabla


n = 3185             # cardinalidad exacta del dataset
N = 17219             # numero total de elementos del dataset
M = 128              # valor inicial de M
while (M <= 65536):    # 131072 = 2^16
  print 'Inicio de la ejecucion con M = ',M
  procesarDataset(datasetTarget, k, n, N, M, datasetWriter)
  M = M*2
  print 'Fin de la ejecucion'
  print ''

datasetfile.close()                          # cerramos el fichero csv


tiempoFinEje = time.time()
tiempoTotalEje = tiempoFinEje-tiempoIniEje
print 'Tiempo total de ejecucion del script: ', int(tiempoTotalEje/60), ' minutos y ',int(tiempoTotalEje%60), '
segundos'
```