Apunts del Taller de Nous Usos de la Informàtica

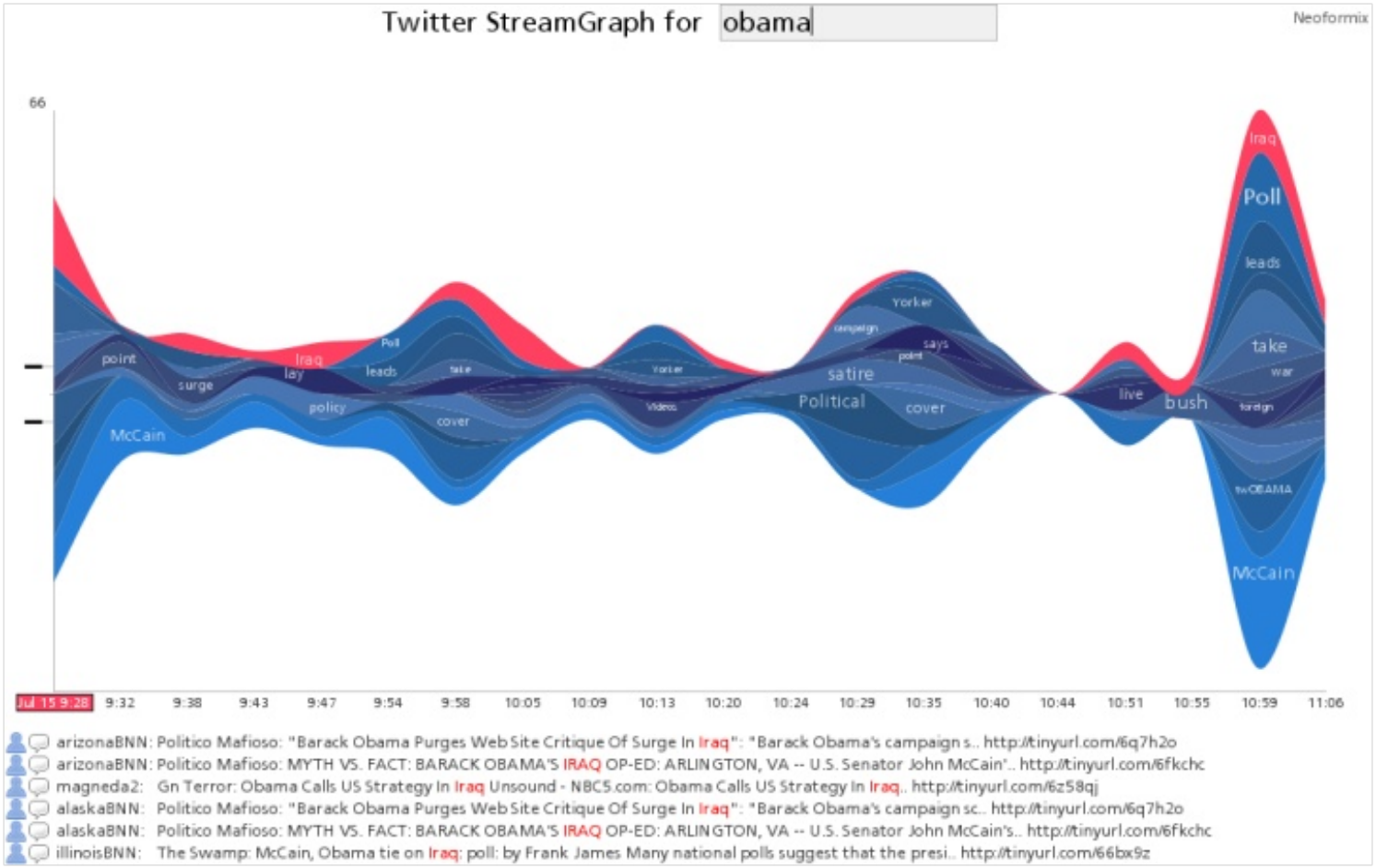
Jordi Vitrià

Universitat de Barcelona

10 de setembre de 2019

UNIVERSITAT DE BARCELONA

Lliçó: Processament de Dades Seqüencials



L'escenari

- Cada cop hi ha més fonts d'informació que ens subministren dades seqüencials en temps real:
  - Mesures d'un gestor de xarxa o de tràfic (seguretat) .
  - Trucades en una gran organització.
  - Transaccions en cadenes de botigues.
  - Operacions en caixers automàtics.
  - Registres log d'un servidor web.
  - Dades de xarxes de sensors.
  - Transaccions de tarjes de crèdit
  - Etc.

L'escenari

- I aquestes fonts d'informació generen quantitats ingents d'informació:
  - Els cercadors d'Internet fan centenars de milions de cerques per dia.
  - Wal*Mart, l'empresa minorista més gran del món, fa més de 20 milions de transaccions diaries.
  - Al món es fan més de 1.000 milions de transaccions amb tarja de crèdit al mes.
  - Al món s'envien 30.000 milions de e-mails diaris i 1000 milions de SMS.
  - Els grans routers d'Internet gestionen 1.000 milions de paquets per hora.

L'escenari

- Per gestionar aquests escenaris ens calen algorismes que estiguin adaptats a:
  - Gestionar volums de dades de l'ordre del TB.
  - Processar les dades en temps real.
  - Processar dades que no seran mai vistes per un humà.

Objectiu d'un sistema de processament de dades seqüencials

L'objectiu genèric d'un sistema de processament de dades seqüencials és explorar les dades (per extreure patrons interessants), donar resposta a les possibles preguntes que pot fer un usuari i calcular estadístics de les dades.

Exemple: Network Data Processing

- Traffic estimation
  - How many bytes were sent between a pair of IP addresses?
  - What fraction network IP addresses are active?
  - List the top 100 IP addresses in terms of traffic
- Traffic analysis
  - What is the average duration of an IP session?
  - What is the median of the number of bytes in each IP session?
- Fraud detection
  - List all sessions that transmitted more than 1000 bytes
  - Identify all sessions whose duration was more than twice the normal.
- Security/Denial of Service
  - List all IP addresses that have witnessed a sudden spike in traffic
  - Identify IP addresses involved in more than 1000 sessions.

Exemple: Sensor Networks

- Cluster Analysis
  - Identification of Profiles: Urban, Rural, Industrial, etc.
- Predictive Analysis
  - Predict the value measured by each sensor for different time horizons.
  - Prediction of picks on the demand.
- Monitoring Evolution
  - Change Detection
    - Detect changes in the behaviour of sensors;
    - Detect Failures and Abnormal Activities;
  - Extreme Values, Anomaly and Outlier Detection
    - Identification of picks on the demand;
    - Identification of critical points in load evolution;

Exemple real

Cloud services teams at X collect billions of metrics daily from millions of heterogeneous mobile devices. Each metric is associated with metadata such as the location, application version, and os version, while analysts issue queries comparing quantiles across different metadata values. For instance, comparing the 99th percentile response latency across different application versions can reveal applications bugs and performance regressions.

Monitorització de xarxes socials

Powered by Bluefin Labs

Social TV Leaderboard

Data for shows on:
Mon, Oct 17, 2011

Top Social TV Shows (Among All Shows)

1 NFL: Dolphins vs. Jets ESPN

161K	103K	8.4
Total Comments	Unique Authors	Response Level

2 The Bad Girls Club

41K	27K	7.8
Total Comments	Unique Authors	Response Level

3 Dancing With the Stars abc

37K	24K	7.7
Total Comments	Unique Authors	Response Level

Social TV data powered by Bluefin Labs. For more data, please visit www.bluefinlabs.com.

Característiques del problema

Des d'un punt de vista computacional, aquestes tasques tenen aquestes característiques en comú:

- Els registres s'han de computar en un **temps constant**.
- La **memòria usada ha de ser constant**.
- Les dades només **s'inspeccionen una vegada**.
- Els resultats s'han de poder **donar en qualsevol moment**.
- S'han de poder gestionar **fonts de dades que canvien amb el temps**.

Algorismes per processar seqüències

- En aquesta lliçó tractarem els següents problemes:
  - Com *mostrejar* una seqüència de manera que podem respondre aproximadament preguntes que requereixen el coneixement de tota la seqüència.
  - La *detecció robusta de canvis* en una seqüència.
  - El *filtratge* de seqüències, o com detectar els items que formen part d'un determinat conjunt quan el conjunt és tant gran que no el podem tenir a memòria.
  - Com *contar el nombre d'elements diferents* que apareixen a una seqüència.

Mostreig d'una seqüència: preliminars

- Suposem que volem estar en disposició de contestar una sèrie de preguntes *ad-hoc* diàries (que no coneixem a priori) sobre una seqüència, però per problemes d'espai només podem emmagatzemar un 10% de la seqüència
- Per exemple: Quina part de les preguntes que han fet els usuaris del meu sistema s'han repetit com a mínim dues vegades per part del mateix usuari durant el darrer mes?
- La versió ingènua consistiria en mostrejar 1 una de cada 10 preguntes i fer càlculs sobre aquesta mostra per contestar les preguntes. Però...

Mostreig d'una seqüència: preliminars

- L'esquema ingenu és incorrecte:
  - Imaginem que la seqüència representa les transaccions que han fet els usuaris amb el nostre sistema, i que un usuari ha fet s preguntes sobre un determinat tema una vegada, d preguntes dues vegades i cap pregunta més de dues vegades durant l'últim mes.
  - Si fem el mostrig ingenu:
    - De les primeres, n'haurem mostrejat $s/10$.
    - De les preguntes que apareixen dues vegades a la seqüència original, n'haurem mostrejat $d/100$ parelles i $18d/100$ apareixeran un sol cop a la mostra.
  - La resposta correcta a la nostra pregunta era $d/(s+d)$, però nosaltres donaríem $d/(10s+19d)$!

Mostreig d'una seqüència: preliminars

- Per contestar les preguntes sobre el comportament dels usuaris hem de mostrejar **totes** les preguntes que fan el 10% dels usuaris. Però mostrejar el 10% dels usuaris ens porta a un altre problema... Com creem la mostra del 10% d'usuaris si no sabem a priori quins són?
- Solució 1: Quan arriba una pregunta, primer *mirem si l'usuari és conegut*. Si ho és, mirem si haviem decidit mostrejar-lo o no i ho apliquem. Si no hi és, generem un nombre aleatori entre 0 i 9 i si surt 0, l'afegim. En cas contrari, decidim no mostrejar-lo i ho apuntem.
- Però... Això és simple d'implementar si tenim la llista d'usuaris a memòria, però i si no hi cap?

Mostreig d'una seqüència de llargada indefinida

En termes més generals podem plantejar aquest altre problema:

- Diguem que tenim una gran seqüència de dades de llargada desconeguda, que només es pot llegir una vegada. El problema és crear un algorisme que retorni una mostra de k elements de manera que tots els ítems tinguin la mateixa probabilitat d'estar-hi representats.
- La forma fàcil, si tenim una llista dels possibles elements de la seqüència, N, és generar enters aleatoris entre 0 i $N-1$ cada cop que arriba un nou ítem i escollir els ítems que cauen a les k primeres posicions, però no és el cas (ni tant sols sabem N).

Mostreig d'una seqüència de llargada indefinida.

- El 1985 es va proposar un mètode satisfactori: el *reservoir sampling*.
- Aquest algorisme manté una mostra aleatòria uniforme de mida fixa, k.
- L'algorisme inicialitza la mostra amb els k primers elements de la seqüència.
- A continuació, cada vegada que arriba un nou element $k+i$ l'afegeix amb probabilitat $p = k/(k+i)$ a la mostra, eliminant un element de la mostra de forma aleatòria per fer-li lloc.
- Es pot demostrar per inducció que després de i elements, la probabilitat de que un element estigui a la mostra és k/i.

Reservoir sampling.

```
def reservoir(s,k):
    import random
    r = [0]*k
    for i in range(k):
        r[i]=s[i]
    for i in range(k,len(s)-1):
        j=random.randint(0,i)
        if j<k:
            r[j] = s[i]
    return r

>>> k=10000
... s=range(1000000)
... res=reservoir(s,k)
... print(reduce(lambda x, y: x + y, res) / len(res))
501025
```

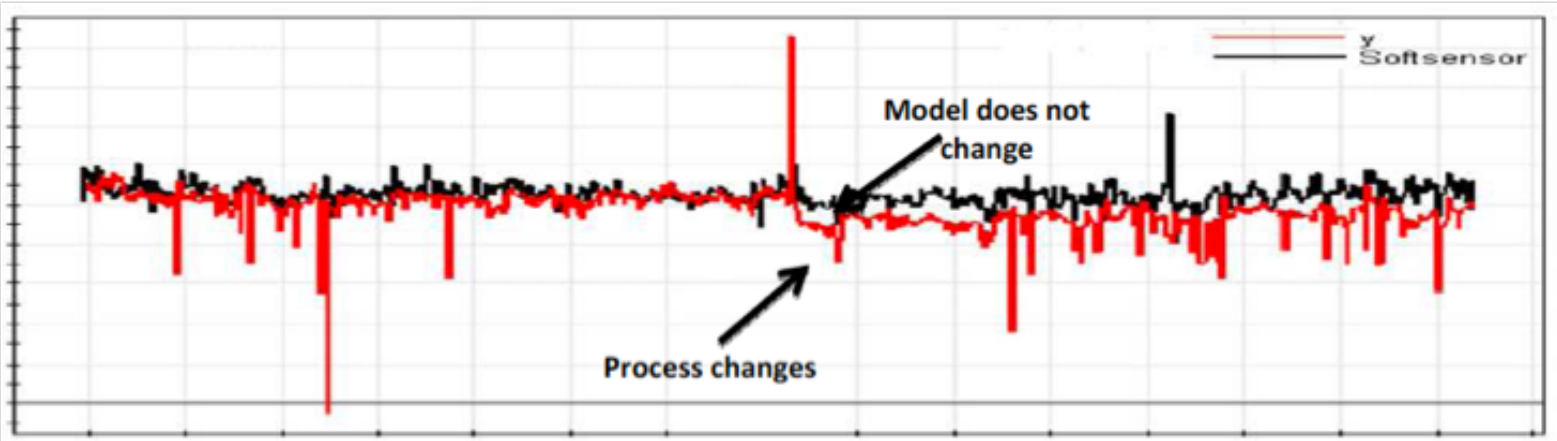
Detecció de canvis.

- Hi ha dos tipus de canvis que val la pena detectar sobre una seqüència dades:
  - La *deriva de concepte*, que representa qualsevol canvi suau en la distribució que genera les dades.
  - La *detecciò d' anomalies*, que representa un canvi sobtat i no esperat en la distribució que genera les dades.

Detecció de canvis.

- Exemple: En un sistema de producció complex els canvis poden tenir molts motius:
  - Hi ha un canvi de qualitat en la materia primera.
  - S'espatlla un sensor.
  - Hi ha operaris nous.
  - Hi ha canvis en els mètodes de producció.
- El que volem és adaptar-nos al canvi el més aviat millor.

Detecció de canvis



source: Evonik Industries

ADWIN

Un cas simple però interessant i útil és determinar quan dues parts complementaries d'una finestra W tenen una mitja diferent.
En aquest cas, senyalarem en aquell instant de temps un *canvi*.
L'algorisme ADWIN és una bona solució per aquest problema:

Algorithm 1 ADWIN

1: $W = []$
2: **for** all $x_i \in S$ **do**
3: $\quad W \leftarrow W \cup \{x_i\}$
4: $\quad$ **while** hi hagi una partició de W en dues subfinestres $W = W_0 + W_1$ que compleixi $|\mu_{W_0} - \mu_{W_1}| \geq \epsilon_{cut}$ **do**
5: $\quad\quad$ Elimina W_0: $W \leftarrow W_1$.
6: $\quad\quad$ Retornar μ_{W_0}.
7: $\quad$ **end while**
8: **end for**

Detecció de canvis.

- Quan trobem una partició $|\mu_{W_0} - \mu_{W_1}| > \epsilon_{cut}$ hem detectat un canvi a la seqüència i retornem la mitja del segment més antic.
- La gràcia d'aquest algorisme rau en la definició de ϵ_{cut}.
- Sigui n la mida de W i n_0, n_1 les mides de W_0, W_1 respectivament. Siguin μ_{W_0}, μ_{W_1} les mitjes empíriques a W_0, W_1 respectivament. Llavors definim:

$$\epsilon_{cut} = \sqrt{\frac{1}{2m} \cdot \ln \frac{4}{\delta'}} \tag{1}$$

on $m = 1/(1/n_0 + 1/n_1)$ i $\delta' = \delta/n$, on δ és el nivell de confiança amb el que volem prendre la decisió.

Detecció de canvis.

- Aquesta valor de tall està calculat segons la cota de Hoeffding, que ens assegura que l'evidència sobre la diferència entre les mitjes és suficient segons el nombre de punts de cada subfinestra.
- Aquesta formula assumeix que les dades estan normalitzades al rang [0,1]. Per tant, abans d'aplicar l'algorisme AdWin cal normalitzar les dades abans!

Detecció de canvis

El procés quan arriba un punt consisteix en anar buidant la part més antiga de la finestra sempre i quan quedi alguna partició que compleixi el criteri:

$$W = 101010110111111$$
$$\Downarrow$$
$$W = \{1\} + \{01010110111111\}$$
$$W = \{10\} + \{1010110111111\}$$
$$W = \{101\} + \{010110111111\}$$
$$\ldots$$
$$W = \{101010110\} + \{111111\}$$
$$\Downarrow$$
$$W = \{1\} + \{11111\}$$
$$W = \{11\} + \{1111\}$$
$$\ldots$$

Filtratge de seqüències

- El problema del filtratge de seqüències és determinar si un determinat element de la seqüència s_i (p.e. una adreça de mail) forma part d'un conjunt E o no.
- Si la cardinalitat del conjunt E és petita podem mantenir una llista d'elements del conjunt a la memòria i fer un test de pertinença de l'element: $s_i \in E$?
- Si la cardinalitat és molt gran (p.e. 10^9 possibles elements), podem tenir problemes de memòria o de temps de processament.

Filtratge de seqüències

- Els *filtres de Bloom* ens poden ajudar en aquests casos.
- Si tenim una memòria de 8GB disponible, podem usar aquesta memòria com un *bit-array* de 8×10^9 bits.
- El filtre de Bloom és una funció *hash* que envia l'element s_i a un determinat bit de la memòria.
- La memòria es pot inicialitzar de manera que els bits que corresponen als elements d'E estiguin a 1 i els altres a 0.

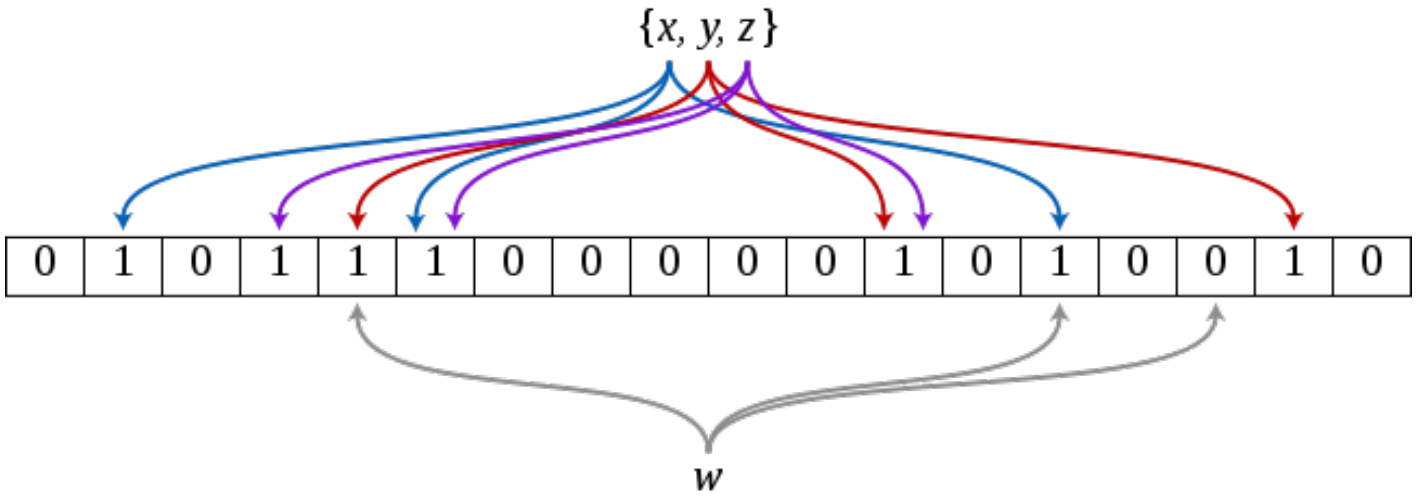
Filtratge de seqüències

- Si suposem que tenim 10^9 elements al conjunt podem assegurar que *com a màxim* $\frac{1}{8}$ part dels bits estaran a 1 (algunes dades diferents del conjunt aniran a la mateixa cel.la i col.lisionaran, però això no és un problema).
- També podem assegurar que només $\frac{1}{8}$ part dels elements que no són del conjunt cauran en una cel.la amb un 1, i per tant en podrem eliminar $\frac{7}{8}$ parts.
- Què fem amb els errors (falsos positius)?

Filtratge de seqüències

- Si fem una *cascada de filtres de Bloom* (formada per funcions hash independents), a cada etapa eliminarem $\frac{7}{8}$ parts dels que han passat la fase anterior. En poques etapes reduïm els falsos positius a nivells molt baixos.
- Per crear un bit-array per una cascada de k elements, simplement posem un 1 a totes les posicions on ens envien les k funcions hash que apliquem.

Filtratge de seqüències



Exemple de funcionament d'una cascada de 3 filtres de Bloom per tres elements del conjunt $\{x, y, z\}$ i per un element que no és del conjunt $\{w\}$.

Filtratge de seqüències

- Si suposem que tenim una bit-array de n bits, m elements en el conjunt i k funcions hash, la *probabilitat d'un fals positiu* és de l'ordre de $(1 - e^{-km/n})^k$.
- En el cas de l'exemple comentat anteriorment (amb un bit-array de 8×10^9 bits i un conjunt de 10^9 elements) amb una funció hash tenim una probabilitat 0.1175 de falsos positius i amb dues funcions hash és 0.0493.

Filtratge de seqüències

- Les propietats dels filtres de Bloom són:
  - La quantitat d'espai necessari per emmagatzemar el filtre és petit comparat amb les dades que representen el conjunt.
  - El temps necessari per comprovar la pertinença d'un element al conjunt és independent del del nombre d'elements del conjunt.
  - No hi ha falsos negatius.
  - Pot haver-hi falsos positius, però la seva freqüència es pot controlar. De fet, hi ha un compromís entre l'eficiència en espai/temps i la freqüència de falsos positius.

Contatge d'elements diferents

- El problema del contatge d'elements es pot definir com determinar el nombre d'elements diferents que han sortit a una seqüència des del principi.
- Un exemple pràctic és el de l'estimació del nombre d'usuaris únics en un servei web.
- L'aproximació ingènua a aquest problema és fer una llista de tots els elements que hem vist. Quan n'arriba un, determinem si està a la llista. Si hi és, no fem res. Si no hi és, l'afegim.
- L'aproximació ingènua té un problema: si hi ha massa elements diferents la *memòria necessària creix sense límit* i el *temps de càlcul augmenta progressivament*.

Contatge d'elements diferents

- L'algorisme de *Flajolet-Martin* proposa una solució a aquest problema basada en la següent idea: es pot estimar el nombre d'elements diferents fent un hashing dels elements a un string de bits suficientment llarg.
- L'string ha de tenir una longitud k suficient de manera que hi hagi més valors possibles de l'string que elements en el conjunt (p.e. amb 64 bits n'hi ha prou pel cas de contar URLs).
- Suposem que tenim una funció hash que produeix un string de k bits per cada element. Llavors, com més elements veiem, més valors diferents de la funció hash veurem.
- A mesura que anem veient més valors diferents de la funció hash tindrem més valors amb propietats específiques, com per exemple que acaben en un substring de 0's (que anomenem *longitud de la cua*).

Contatge d'elements diferents

- Si R és la longitud de cua més llarga que hem vist fins a un determinat moment en la seqüència, es pot demostrar que el nombre 2^R és un bon estimador del nombre d'elements diferents:
  - La probabilitat que s_i acabi en r 0's és 2^{-r}.
  - Si tenim m elements a la seqüència, la probabilitat de que cap d'ells tingui una longitud de cua de almenys r és $(1-2^{-r})^m$, que és igual a $((1-2^{-r})^{2^r})^{m2^{-r}}$.
  - Si r és prou gran, això és aproximadament $e^{-m2^{-r}}$.

Filtratge de seqüències

- Per tant, si m és més gran que 2^r la probabilitat de que un element tingui una longitud de cua de almenys r s'aproxima a 1; si m és més petit que 2^r la probabilitat de que un element tingui una longitud de cua de almenys r s'aproxima a 0.
- Per tant podem concloure que l'estimador de m, 2^R (R és la longitud de cua més gran possible a la seqüència) no és ni massa gran ni massa petit.

Sotware

Bounter -- Counter for large datasets

build passing release v1.0.1 Mailing List gitter join chat → Follow 3k

Bounter is a Python library, written in C, for extremely fast probabilistic counting of item frequencies in massive datasets, using only a small fixed memory footprint.

Why Bounter?

Bounter lets you count how many times an item appears, similar to Python's built-in `dict` or `Counter`:

```
from bounter import bounter

counts = bounter(size_mb=1024)  # use at most 1 GB of RAM
counts.update([u'a', 'few', u'words', u'a', u'few', u'times'])  # count item frequencies

print(counts[u'few'])  # query the counts
2
```

Sotware

1. Cardinality estimation: "How many unique items are there?"

```
from bounter import bounter

counts = bounter(need_counts=False)
counts.update(['a', 'b', 'c', 'a', 'b'])

print(counts.cardinality())  # cardinality estimation
3
print(counts.total())  # efficiently accumulates counts across all items
5
```

Sotware

2. Item frequencies: "How many times did this item appear?"

```
from bounter import bounter

counts = bounter(need_iteration=False, size_mb=200)
counts.update(['a', 'b', 'c', 'a', 'b'])
print(counts.total(), counts.cardinality())  # total and cardinality still work
(5L, 3L)

print(counts['a'])  # supports asking for counts of individual items
2
```

Sotware

3. Full item iteration: "What are the items and their frequencies?"

```
from bounter import bounter

counts = bounter(size_mb=200)  # default version, unless you specify need_items or need_counts
counts.update(['a', 'b', 'c', 'a', 'b'])
print(counts.total(), counts.cardinality())  # total and cardinality still work
(5L, 3)
print(counts['a'])  # individual item frequency still works
2

print(list(counts))  # iterator returns keys, just like Counter
[u'b', u'a', u'c']
print(list(counts.iteritems()))  # supports iterating over key-count pairs, etc.
[(u'b', 2L), (u'a', 2L), (u'c', 1L)]
```

Sotware

Example on the English Wikipedia

Let's count the frequencies of all bigrams in the English Wikipedia corpus:

```
with smart_open('wikipedia_tokens.txt.gz') as wiki:
    for line in wiki:
        words = line.decode().split()
        bigrams = zip(words, words[1:])
        counter.update(u' '.join(pair) for pair in bigrams)

print(counter[u'czech republic'])
42099
```

The Wikipedia dataset contained 7,661,318 distinct words across 1,860,927,726 total words, and 179,413,989 distinct bigrams across 1,857,420,106 total bigrams. Storing them in a naive built-in `dict` would consume over 31 GB RAM.