Getting stuff done with Big Data

Lecture Three: Randomised Algorithms

Miles Osborne

School of Informatics
University of Edinburgh
miles@inf.ed.ac.uk

February 11, 2012

Motivation

Computational efficiency will always be a concern:

- The more efficient we are, the more/ bigger problems we can tackle.
- Greater efficiency means cheaper running costs.
- Using more data can mean better results
- Even with a cluster, we may have to compete with other services.
- Storage / processing times may grow very quickly with Big Data
- For mobiles, we may have limited resources.

We will always want to tackle problems that don't fit into our machines

Motivation

Example

Suppose we want to count the numbers of times each word pair occurs in a large number of documents. How can we do this in a space efficient way?

- An *exact* approach would try to guess the maximum count per pair (say 2^{32}).
  - Allocate a 32-bit counter to each pair of words.
  - Update this counter every time we see the corresponding pair.
- Allocating this space in advance is very wasteful.

Can we do better?

Motivation

We won't see all possible pairs:

- Some pairs will be seen many times (eg function words).
- Some pairs will be seen a few times (eg a rare word and a functiom word).
- Most pairs will never be seen at all. (pairs of rare words).

We can use a sparse representation to only store pair-counts we have seen so far.
Can we do even better?

Motivation

Say we don't need exact counts:

- We may only care about ranking pairs of words by frequency.
- We may only want the top-n most frequent pairs.

By storing *approximate* counts, we can save on space.

> *If we can tolerate errors / inexact results, then randomised approaches will provably be more space/ time efficient (etc) than exact methods.*

Motivation

Randomised algorithms:

- Replace an exact method with one that makes mistakes.
- These mistakes (error rate, ϵ) can be quantified.
- Depending upon the application, the errors may vary:
  - When storing items, we might think we stored items that we never inserted (false positive).
  - When processing items, our approach might fail to find a solution at all.
- Typically, there is a trade-off between the error rate and performance level.

Randomised approaches are often the most efficient approach to many classes of problems

Probabilistic Counting

Returning to our counting problem:

- We want to allocate the smallest amount of space possible to our counters.
- Errors here might involve mis-counting.
- We could sample the data and only count (say) every 1 item in ten.
  - Down-sampling may miss rare events.
- *Probabilistic counting* is a randomised counting approach.

Probabilistic Counting

Central idea:

- Only store exponents (saves on space)
- Only approximate counts (makes errors)

True Count	Approximate Count
1	1
2 – 10	2
10 – 100	3

Probabilistic Counting

How it works:

- Every time we see an instance, instead of always updating the counter f by one, only update it by 1 with probability 2^{-f}.
- To update the counter, the test is whether some random number (sampled uniformly between 0 and 1) is less than 2^{-f}
- We now need only spend $\log(\log(f))$ bits per counter, instead of $\log(f)$ bits.

This counts in log-space.

Probabilistic Counting

Example

Suppose we count the letter *a* in some stream:

Stream	Random Number	Decision
		Counter is 0 initially
a	0.3	$2^0 = 1.0$, update, new counter is 1
aa	0.7	$2^{-1} = 0.5$, fail, no update
aaa	0.3	$2^{-1} = 0.5$, update, new counter is 2
aaaa	0.1	$2^{-2} = 0.25$, update, new counter is 3

Probabilistic Counting

Example

In general we will be counting many objects

Instances	True count	Approximate counts
1000	1	1
1000	4	2

Space used (32 bits per exact counter, 2 bits per approx counter):

True count	Approximate count
2000 * 32	2000 * 2

Probabilistic Counting

- At times we can mis-estimate counts by an order of magnitude or more!
  - We may under-count or over-count
- Using a smaller base (less than 2) reduces errors (there are more update chances, but we can count to less)

Hashing

Many randomised approaches rest upon *hashing*:

- Hashing can be used to reduce space requirements (see Bloom Filters).
- ... can be used for a speed-up (see Distance metrics)
- ... and also for streaming algorithms.

A hash function maps items from a range $1 \dots m$ to $1 \dots n$, where $n << m$

Hashing

A good hash function $h(X)$ has few collisions:

- $P(h(x) = h(y)) = \frac{1}{n}$ (ie the chance of any two items having the same address is the chance of visiting any address with an equal chance)

Good hash functions should be quick to evaluate, since we may be hashing millions of times.

Hashing

Universal Hashing is often used:

- Pick random numbers a and b.
- Pick some large prime p at random:

$$h(x) = ((ax + b)\%p)\ \%n$$

- This uses a modulus operator.
- Each time we pick a new set of random numbers, we get a new hash function.

Universal Hashing closely satisfies the requirements of good hashing.

Hashing

Quiz

Suppose you want to hash a string. How can you do it with Universal Hashing?

Finger Printing

At times, we need to store some object, but we want to do it compactly:

- A *fingerprint* is the hash address of some object.
- The larger n is, the more bits we use.
- The smaller n is, the greater the chance of making a mistake (a collision).
- We only store the fingerprint of objects.
  - Item comparison is fast: just use fingerprints.
  - Item storage can be compact: just store fingerprints.

Finger Printing

Example

String	Finger print (bit pattern)
adssdsds	111
dsfdfda	010
wewdsws	110

Using one bit, we collide twice; three bits there are no collisions

Mid Summary

- Motivated the need for randomised algorithms.
- Introduced a set of techniques.

Bloom Filters: Motivation

Often we need to store items

- Ngrams for language models
- Translation tables
- Model parameters
- Etc

Bloom Filters: Motivation

Two storage problems:

- Membership task: Did we store some item?
- Key-value task: Return the value associated with some key.

We will focus upon the Membership task

Bloom Filters: Motivation

We can work-out worst-case space requirements

- Suppose we have n possible items we need to store
  - For example, all possible word pairs
- To store a set of word pairs of size s:
  - Work-out how many possible subsets of size s there are.
  - Allocate a code-word to each distinct subset.
  - Storing our subset means assigning a code word to that subset and storing the code-word.
- This takes $\log \binom{n}{s}$ bits per set.

As the underlying universe increases in size, there are more possible subsets and so we need to use more space for each item.

Bloom Filters: Motivation

Suppose we can make mistakes:

- We might say we stored an item we never inserted into the table.
  - This is a *False Positive*.
- We might fail to recover some item we inserted into the table.
  - This is a *False Negative*.

Bloom Filter

A Bloom Filter is a randomised data-structure which supports membership queries, with the possibility of False Positives.

- Extremely simple.
- Based upon a bit vector.
- . . . and a set of k hash functions (Universal Hashing) indexing bit addresses.
- Used in mainstream Computer Science:
  - Routing in networks.
  - Detecting intruders.
  - Managing caches.
  - etc

Also used to represent large language models in Machine Translation

Bloom Filters

Suppose we want to store items: A, B, C:

0	1	2	3	4	5	6
0	0	0	0	0	0	0

The BF is initially empty.

Bloom Filters

Storing A:

0	1	2	3	4	5	6
1	0	0	1	0	0	0

(Using two hash functions)

Bloom Filters

Storing B:

0	1	2	3	4	5	6
1	0	0	1	0	0	1

Bloom Filters

Did we store A?

0	1	2	3	4	5	6
1	0	0	1	0	0	1

We hash again and find that all the hashed bits are set:
$\rightarrow$ **true positive**

Bloom Filters

Did we store C?

0	1	2	3	4	5	6
1	0	0	1	0	0	1

We hash again and find that bit 2 is not set:
→ **true negative**

Bloom Filters

Did we store D?

0	1	2	3	4	5	6
1	0	0	1	0	0	1

We hash again and find that bits 0 and 6 are set:
$\rightarrow$ false positive

Bloom Filters

The error rate depends upon

- The number items in the table.
- The size of the table.

If we insert more items into a table of fixed size, then the error rate must increase.

Bloom Filters

For a given number of entries s and a table of size m bits:

- We need to use k hash functions:

$$k = \frac{m}{s} \ln 2$$

- The error rate of our table is:

$$\epsilon = 0.5^k$$

Bloom Filters

Bloom Filters have curious properties:

- They never fill-up.
- We can always recognise items we inserted into the table.
- It is very hard to reverse engineer a BF
  - Interesting privacy implications.

Example: Querying 4 billion Strings

We created two BFs to represent 4B strings:

- Table One: $700M$ of space, using 1 hash function.
  - **50%** error rate
- Table Two: $2GB$ of space, using 3 hash functions.
  - **11%** error rate

24 GB to represent the strings exactly using gzip

Example: Querying 4 billion Strings

700M Filter:

Ngram	Inserted into the table?
serve as the instruments	Yes
serve as there insurer	No
sarkozy sarkozy sarkozy	No
ZZZZX zxzxzx rareta	No
mein name ish trudyyyy	No
bvcxc can't sphelle	No
duo core quad core pentium	No
serve the instructional institution	No
the vodka is strong	No
the meat has gone bad	No

Example: Querying 4 billion Strings

2GB Filter:

Ngram	Inserted into the table?
serve as the instruments	Yes
serve as there insurer	No
sarkozy sarkozy sarkozy	No
ZZZZX zxzxzx rareta	No
mein name ish trudyyyy	No
bvcxc can't sphelle	No
duo core quad core pentium	No
serve the instructional institution	No
the vodka is strong	No
the meat has gone bad	No

Example: Language Models

In Machine Translation we might want to use trillions of words of text:

- It takes 1.5k machines one day (using Map Reduce) to count all ngrams in this data
- As more and more data is used, translation performance continues to increase
- The language model itself cannot be stored on a single machine

Example: Language Models

Using a randomised representation:

- We can achieve a one-order of magnitude space reduction over an exact representation
- Translation can tolerate errors in the language model

The very largest language models are randomised and distributed over multiple machines

Distance Functions: Motivation

A *distance function* measures how 'close' two items are to each other:

- All Facebook friends who share similar interests.
- Web pages that are similar to a search query.
- Images that look like houses
- Documents that are near-duplicates of each other.

All of these tasks use *distance functions*

Distance Functions: Motivation

Suppose you want to find all duplicate and near-duplicate Web pages:

- Vast numbers of Web pages are copied / edited.
- Web size estimate 2008*: more than 1 trillion web pages

* http://googleblog.blogspot.com/2008/07/we-knew-web-was-big.html

Distance Functions: Motivation

A naive approach compares each page to every other page

- A randomised approach can do it using sorting

Basic idea:

- Construct a special fingerprinting scheme.
- Sort items by their fingerprint.
- Items that share the same fingerprint are likely to be similar to each other.

Randomised Distance Metric

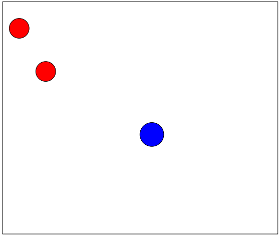
Represent items as vectors:

- Each component might be the presence of a word
- Vector representations are common in Search etc

Assign a fingerprint as follows:

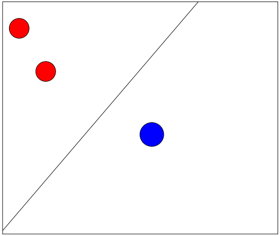
- Randomly construct a hyperplane.
- Assign a zero or one depending on which side of the hyperplane the vector is placed

Randomised Distance Metric



Two red points are close to each other

Randomised Distance Metric



Red points in same plane

Randomised Distance Metric

Errors:

- Random hyperplanes might misclassify an item.
- We can repeat the whole process and amplify the success probability.

Randomised Distance Metric: Locality Sensitive Hashing

Many language tasks involve finding similar items:

- Search (documents similar to a query)
- First Story Detection (very new documents)

LSH is a fast cosine-search based upon a randomised distance metric

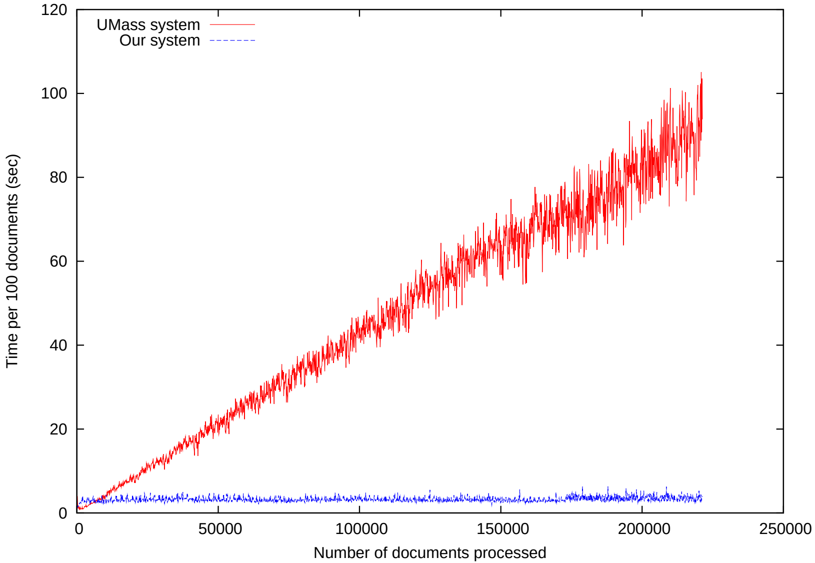
Randomised Distance Metric: Locality Sensitive Hashing

We have used LSH to find breaking news in Twitter

- More than 1 million new posts a day
- Breaking news: a new Tweet that is different from Tweets seen so far

Exact approaches take linear time in terms of the number of documents seen so far

Randomised Distance Metric: Locality Sensitive Hashing



Summary

- Introduced Bloom Filters.
- Introduced Randomised Distance Functions.