# Machine learning using more data than is healthy: building blocks

Miles Osborne

July 2013

# Class Structure

Part One:

- Concept of randomisation.
- Building blocks for tackling massive amounts of data.

Part Two:

- Working with streams.
- Methods for tackling streams.
- Case study: Event detection in Twitter.

# Working with lots of data

Sometimes we have too much data:

- Trillions of words of Web data
- Billions of financial transactions
- Lots of phone calls *cough*
- . . .

How can we build good models using this much data?

# Working with lots of data

We can use a cluster:

- This will cost us money proportional to our usage.
- Storage / processing times may grow very quickly with Big Data
  - No amount of resources might be available to solve the problem.
- Jervons Paradox:
  - As we become more efficient, demand for resources grows even faster.
  - More resources can make us less efficient overall.
- Even if we have the resources, we might want to minimise costs.

# Working with lots of data

An alternative idea is to rethink how we problem solve:

- We would like to use all of the data.
- But at times, we might be able to accept small errors.
  - At scale errors happen all the time.
- Can we trade an exact approach with one that makes errors, but is faster / more compact etc?

# Example

Suppose we want to count the numbers of times some computer sends a packet to another computer. Say there are 100 million possible computers. How can we do this in a space efficient way?

(We might want to predict which machines I will visit)

# Example: Exact Methods

- An exact approach would try to guess the maximum count per pair (say $2^{32}$).
  - Allocate a 32-bit counter to each pair of computers.
  - Update this counter every time we see the corresponding packet.
- Can we do better?

# Example: Exact Methods

We won't see all possible pairs:

- Some pairs will be seen many times (eg each time I ping Google).
- Some pairs will be seen a few times (eg some random Web site I visit).
- Most pairs will never be seen at all. (I don't visit most sites).

We can use a sparse representation to only store pairs we see so far.
Can we do even better?

# Example: Randomised Methods

Say we don't need exact counts:

- We may only care about ranking pairs of computers by frequency.
  - We do not need the actual counts.
- We may only want the top-$n$ most frequent pairs.
  - We don't care about the other pairs.

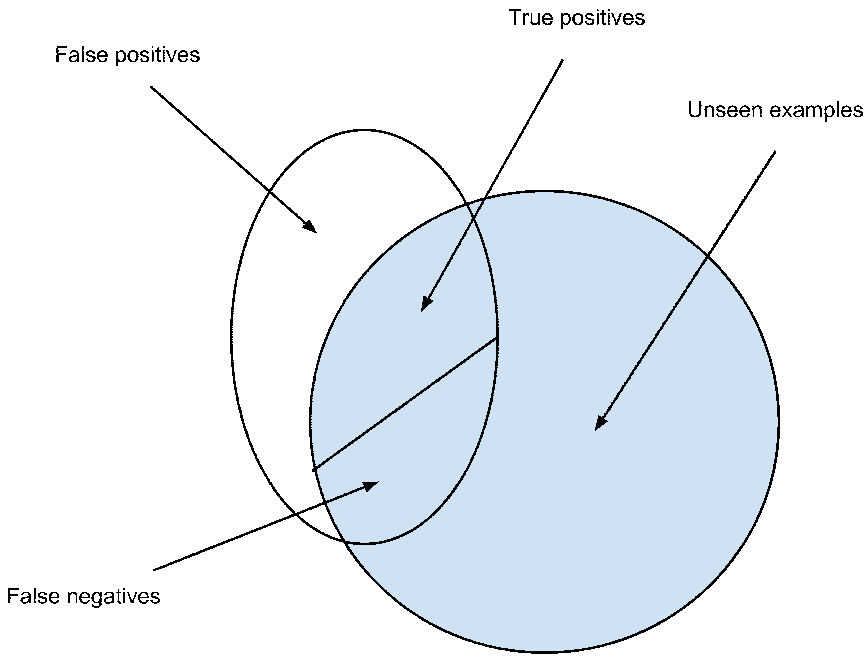
If we can tolerate errors / inexact results then we can be provably more efficient (etc) than exact methods.

# Randomisation

Randomised algorithms:

- Replace an exact method with one that makes mistakes.
- These mistakes (error rate, $\varepsilon$) can be quantified.
- Depending upon the application, the errors may vary:
  - When storing items, we might think we stored items that we never inserted .
  - 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.

# Types of Error



# Machine Learning and Randomisation

Many (all?) ML tasks can be randomised:

- Supervised learning.
- Unsupervised learning.

Usually we replace the representation, but at times we can change the algorithm.

# Example: Randomising Classification

Typically we build a model using features:

- Words, translation pairs, images etc

and counts:

- How often we see some feature in the data

And the model itself generally is a large vector of real values.

# Example: Randomising Classification

Options:

- Represent features in an inexact way (Hashing)
- Represent counts in an inexact way (Probabilistic counting)
- Represent weights inexactly (Quantising)
- Implement feature-weight mapping in an inexact way (Bloom Filter).

# Example: Randomising Classification

Task: case restoration

the cat sat on the mat $\longrightarrow$ **The** cat sat on the mat

on behalf of the chairman, mr. jimi m. hendrix, welcome to the home of the worlds local bank, hsbc $\longrightarrow$
**On** behalf of the **Chairman**, **Mr**. **Jimi M. Hendrix**, welcome to the home of the worlds local bank, **HSBC**

# Example: Randomising Classification

Labels:

- **CA**: all letters uppercase
- **LC**: all letters lowercase
- **MC**: mixed case word
- **UC**: first letter uppercase
- **DIGIT**: a number

Features: Words, prefixes, postfixes etc

# Example: Randomising Classification

Take a large amount of text data and apply rules to created labelled data:

Steve Scrutton is a social work manager $\longrightarrow$
steve/UC scrutton/UC is/LC a/LC social/LC work/LC manager/LC

# Example: Randomising Classification

Represent features inexactly:

- Each feature has an associated weight.
- We maintain a feature-weight mapping.

steve/UC scrutton/UC is/LC . . .

| Feature | Quantised value |
|---|---|
| $\langle$steve, uc$\rangle$ | 132 |
| $\langle$stev, uc$\rangle$ | 223 |
| $\langle$eve, uc$\rangle$ | 344 |
| . . . | . . . |

# Example: Randomising Classification

Replace feature-weight mapping with a randomised representation:

- We do not explicitly store the features-weight mapping.
- We supply a procedure which can tell if some feature we seen in the training data and supplies the associated weight (Bloom Filter).
- Mistakes:
  - We might think one feature was the same as another feature.
  - We might recover a weight that was too high or too low.

The randomised representation will be a lot smaller than the exact version.

# Example: Randomising Classification

Results

- Maximum entropy classifier.
- Same performance as an exact version.
- One third of the space requirements.
- Manipulating error rate has a direct impact on performance.
  - A higher error rate means less space used, but worse performance.

# Building Blocks

- Hashing
- Finger printing
- Probabilistic counting
- Count-min sketch
- Bloom Filter and variants
- Locality Sensitive Hashing

# 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 Locality Sensitive Hashing)
- . . . and also for streaming algorithms.

A hash function maps items from a range $1 \ldots m$ to $1 \ldots n$, 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.

# Hashing

Example:

| Hash Function | a | b |
|---|---|---|
| H1 | 12 | 232 |
| H2 | 44 | 2121 |
| H3 | 76 | 1 |

p = 5783287, n = 1000

| Value | H1 | H2 | H3 |
|---|---|---|---|
| 12 | 376 | 649 | 913 |
| 13 | 388 | 693 | 989 |
| 3443 | 548 | 613 | 669 |

# Hashing and Features

A simple trick to reduce the space requirements for Machine Learning:

- Do not store features.
- Create a hash function which maps features into identifiers.
  - Some features will collide.
- Do ML as normal.

Useful when the space of features grows over time.

# Exotic Hashing

There is a lot of work on hashing:

- Rolling hashing: incrementally computes a hash value from a previously computed value.
- Hashing that is non-uniform.
- Super fast hashing:
  - hash(const char* str) {return (*size_t*)str >> precision}

  http://stackoverflow.com/questions/628790/have-a-good-hash-function-for-a-c-hash-table

# 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

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

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

# Probabilistic Counting

Counting is a central Machine Learning task:

- Feature expectations.
- Empirical frequencies.
- Numerical optimisation.

Counts are often Zipfian, so we should be able to save on space.

# Probabilistic Counting

Central idea:

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

| True Count | Approximate Count (in logs) |
|---|---|
| 1 | 1 |
| 2 – 10 | 2 |
| 10 – 100 | 3 |

# Probabilistic Counting

How it works:

- Every time we see an instance, instead of updating the counter $f$ by one, 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

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

| Stream | Random Number | Counter |
|---|---|---|
| | | 0 |
| $a$ | 0.3 | $2^0 = 1$, so new counter is $1$ |
| $aa$ | 0.7 | $2^{-1} = 0.5$ (this time we fail) |
| $aaa$ | 0.3 | $2^{-1} = 0.5$; this time we update |
| $aaaa$ | 0.1 | $2^{-2} = 0.25$ (etc) |

# Probabilistic Counting

In general we will be counting many objects:

| Instances | True count | Approximate counts |
|---|---|---|
| 1000 | 1 | 1 |
| 1000 | 4 | 2 |

Total space: (1000 * 1) + (1000 * 4) = 5000 v (1000 * 1) + (1000 * 2) = 3000

# Probabilistic Counting

- After $n$ increments, then $E[2^f - 1]$ gives us the approximated count (on average, we can recover the true frequency).
- 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)

# Count-Min Sketch

Probabilistic counting allocates a counter to everything:

- We may only care about the most frequent items.
- Allocating a counter to everything would then be indulgent.

The count-min sketch is a probabilistic data structure which can summarise ("sketch") a stream in sublinear space.

# Count-Min Sketch

The CM sketch consists of:

- A set of arrays of counters.
- Each array has a different hash function.

# Count-Min Sketch

To count an item:

- Visit each array in turn, using each associated hash function.
- Update hashed counter.

To recover the frequency:

- Visit each each in turn and find the hashed value.
- Return the minimum value.

The CM sketch can be used for many other applications.

# CM Sketch Example

Initial sketch:

| | | | |
|---|---|---|---|
| Array 1 | 0 | 0 | 0 |
| Array 2 | 0 | 0 | 0 |
| Array 3 | 0 | 0 | 0 |

And Hash functions $h1$, $h2$ and $h3$.

# CM Sketch Example

Update for $a$:

| | | | |
|---|---|---|---|
| Array 1 | 0 | 1 | 0 |
| Array 2 | 0 | 0 | 0 |
| Array 3 | 0 | 0 | 0 |

# CM Sketch Example

Update for $a$:

| | | | |
|---|---|---|---|
| Array 1 | 0 | 1 | 0 |
| Array 2 | 1 | 0 | 0 |
| Array 3 | 0 | 0 | 0 |

# CM Sketch Example

Update for $a$:

| | | | |
|---|---|---|---|
| Array 1 | 0 | 1 | 0 |
| Array 2 | 1 | 0 | 0 |
| Array 3 | 1 | 0 | 0 |

# CM Sketch Example

Update for $b$:

| | | | |
|---|---|---|---|
| Array 1 | 1 | 1 | 0 |
| Array 2 | 2 | 0 | 0 |
| Array 3 | 1 | 0 | 1 |

# CM Sketch Example

Count for $a$:

| | | | |
|---|---|---|---|
| Array 1 | 1 | 2 | 0 |
| Array 1 | 3 | 0 | 0 |
| Array 1 | 2 | 0 | 1 |

Minimum of 2, 3 and 2 = 2 (correct)

# CM Sketch Example

Count for $b$:

| | | | |
|---|---|---|---|
| Array 1 | 1 | 2 | 0 |
| Array 2 | 3 | 0 | 0 |
| Array 3 | 2 | 0 | 1 |

Minimum of 2, 3 and 1 = 1 (correct)

# Count-Min Sketch

Properties:

- The more counters we have in an array, the lower the error rate ($\varepsilon$).
- The more arrays we have, the more likely the error rate will hold ($\sigma$).
- The recovered estimate is guaranteed to be $\varepsilon$ close to the true estimate, with probability $\sigma$.
- The amount of space used is fixed.
  - For a fixed CM sketch, the error must increase as we count more items.
  - Probabilistic counting grows linearly with the stream size.

# Bloom Filters: Motivation

Often we need to store items

- Credit card numbers
- Images
- Precomputed values
- 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 IP addresses
- To store a set of IP addresses 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.

# 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

# 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:
$\longrightarrow$ 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:
$\longrightarrow$ 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:
$\longrightarrow$ 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:

$$\varepsilon = 0.5^k$$

# 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 |

# Using BFs to store Key-Value Pairs

We often want to store key-value pairs:

- Strings and counts
- Features and identifiers
- State transitions
- Etc

How can we extend a BF to do this efficiently?

# Using BFs to store Key-Value Pairs

A basic BF can only answer a membership query:

- Did we store some item?

Two strategies:

- Encode values using multiple insertions
- (Extend BF to actually store k-v pairs)

# Using BFs to store Key-Value Pairs

Suppose we wish to associate the value 3 with a string s:

- Insert $\langle S, 1 \rangle$
- Insert $\langle S, 2 \rangle$
- Insert $\langle S, 3 \rangle$

If the counts are heavily skewed this can be efficient (most counts are 1 or 2).

# Using BFs to store Key-Value Pairs

Query:

- Did we insert $\langle S, 1 \rangle$? Yes
- Did we insert $\langle S, 2 \rangle$? Yes
- Did we insert $\langle S, 3 \rangle$? Yes
- Did we insert $\langle S, 4 \rangle$? No (hopefully!)

BFs are used to represent large Language Models.

- Reminder: A LM is a large string-probability look-up table used for fluency modelling in translation, speech etc.

# Baseline Language Model Results

| Order | Number of *n*-grams | Space | Space (gzipped) | BLEU |
|---|---|---|---|---|
| 3 | 5.9 M | 174 Mb | 51 Mb | 28.54 |
| 4 | 14.1 M | 477 Mb | 129 Mb | 28.99 |
| 5 | 24.2 M | 924 Mb | 238 Mb | 29.07 |

French-English translation task, 1K test sentence pairs, 500 dev pairs
The baseline LM is based on the SRILM (exact)

# Language Model Results: Log Freq BFs

| Order | Space | BLEU |
|---|---|---|
| 3 | 10 Mb | 28.47 |
| 4 | 20 Mb | 28.63 |
| 5 | 50 Mb | 29.31 |

# Locality Sensitive Hashing: 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

# Locality Sensitive Hashing: 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

# Locality Sensitive Hashing: Motivation

A naive approach compares each page to every other page

- This takes O($n^2$) time.
- A randomised approach can do it using sorting.
  - This takes O($n \log n$) time.

Hashing to the rescue!

# Locality Sensitive Hashing

Basic idea:

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

This can be easily parallelised using Map Reduce

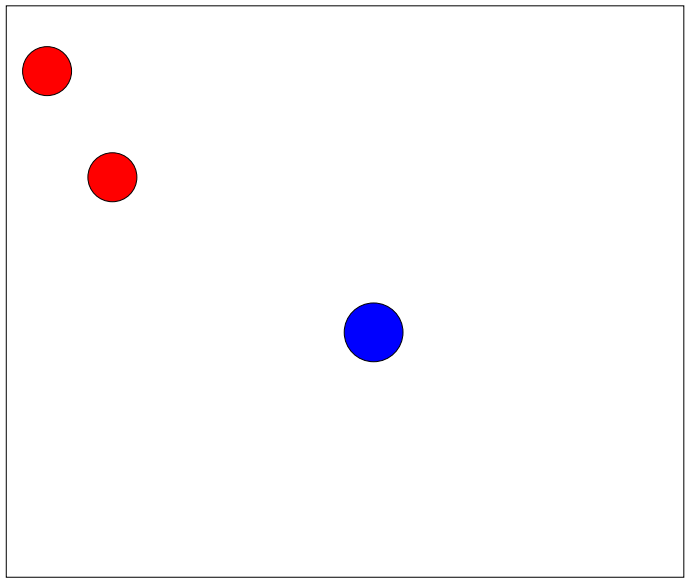
# Locality Sensitive Hashing

Finger printing:

- Represent items as vectors:
  - Each component might be the presence of a word
  - Vector representations are common in Search etc
- Assign a special finger print as follows:
  - Randomly construct a hyperplane over vector space.
  - Assign a zero or one depending on which side of the hyperplane the vector is placed

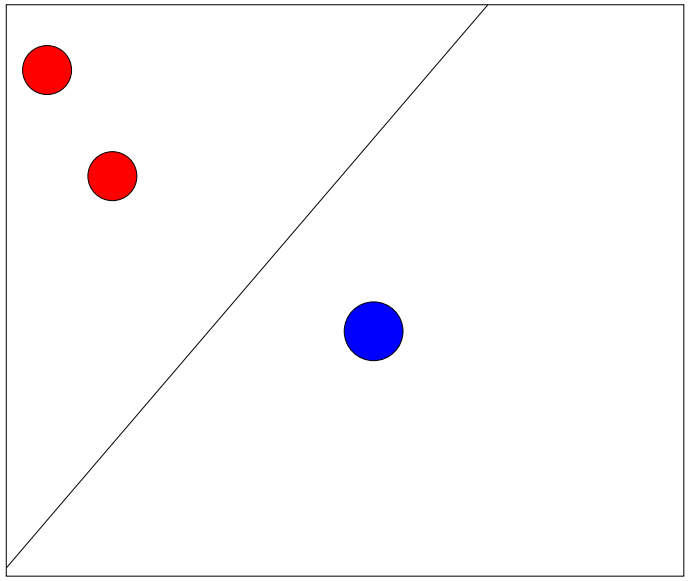
This is called Locality Sensitive Hashing

# Randomised Distance Metric



Two red points are close to each other

# Randomised Distance Metric



Red points in same plane

$$h_i(x) = \text{sign}(x \cdot r_i) > 1$$

# Locality Sensitive Hashing

Errors:

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

LSH is used in deduplication, search and event detection.

# Part 1 Summary

- Randomised methods allow us to tackle really big problems.
  - They typically have a term which trades performance against various kinds of errors.
- We looked at a few building blocks.
- Tackling really Big Data needs randomised methods.