BLOOM FILTERS - RECAP
Bloom filters

• Interface to a Bloom filter
  – BloomFilter(int maxSize, double p);
  – void bf.add(String s); // insert s
  – bool bd.contains(String s);
    • // If s was added return true;
    • // else with probability at least 1-p return false;
    • // else with probability at most p return true;

– I.e., a noisy “set” where you can test membership (and that’s it)
Bloom filters

bf.add(“fred flintstone”):

```
0 0 0 0 0 0 0 0 0 0 0 0
```

set several “random” bits

bf.add(“barney rubble”):

```
1 1 1 0 0 0 1 0 1 0 0 0
```
Bloom filters

bf.contains ("fred flintstone"): return min of “random” bits

bf.contains ("barney rubble"):
Bloom filters

\[
\begin{array}{cccccccc}
1 & 1 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 \\
\end{array}
\]

\text{bf.contains(“wilma flintstone”):}

\[
\begin{array}{cccccccc}
1 & 1 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 \\
\end{array}
\]

\text{bf.contains(“wilma flintstone”):}

\[
\begin{array}{cccccccc}
1 & 1 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 \\
\end{array}
\]

a false positive
BLOOM FILTERS VS COUNT-MIN SKETCHES
Bloom filters – a variant

split the bit vector into $k$ ranges, one for each hash function

\[
\begin{array}{cccccccc}
0 & 0 & 0 & | & 0 & 0 & 0 & | & 0 & 0 & 0 & 0
\end{array}
\]

**bf.add(“fred flintstone”):**

\[
\begin{array}{cccccccc}
0 & 1 & 0 & | & 1 & 0 & 0 & | & 0 & 1 & 0
\end{array}
\]

set one random bit in each subrange

**bf.add(“barney rubble”):**

\[
\begin{array}{cccccccc}
1 & 1 & 0 & | & 1 & 0 & 0 & | & 1 & 1 & 0
\end{array}
\]
Bloom filters – a variant

split the bit vector into k ranges, one for each hash function

```
0 0 0 | 0 0 0 0 | 0 0 0 0
```

bf.contains(“fred flintstore”):

return AND of all hashed bits

```
1 1 0 | 1 0 0 0 | 1 1 1 0
```

bf.contains(“pebbles”):

```
1 1 0 | 1 0 0 0 | 1 1 1 0
```
Bloom filters – a variant

split the bit vector into k ranges, one for each hash function

| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

bf.contains(“pebbles”):

h1

h2

h3

a false positive!
Count-min sketches

split a real vector into k ranges, one for each hash function

\[\begin{array}{ccccccccc}
0 & 0 & 0 & | & 0 & 0 & 0 & | & 0 & 0 & 0 & 0
\end{array}\]

cm.inc("fred flintstone", 3):

\[
\begin{array}{cccccc}
0 & 3 & 0 & | & 3 & 0 & 0 & | & 0 & 3 & 0
\end{array}
\]

add the value to each hash location

cm.inc("barney rubble", 5):

\[
\begin{array}{cccccc}
5 & 3 & 0 & | & 8 & 0 & 0 & | & 5 & 3 & 0
\end{array}
\]
Count-min sketches

split a real vector into k ranges, one for each hash function

```
0 0 0 | 0 0 0 0 | 0 0 0 0
```

cm.get(“fred flintstone”): 3

```
5 3 0 | 8 0 0 | 5 3 0
```

take min when retrieving a value

cm.get(“barney rubble”): 5

```
5 3 0 | 8 0 0 | 5 3 0
```
Count-min sketches

split a \textcolor{red}{real} vector into \( k \) ranges, one for each hash function

\begin{align*}
0 &\quad 0 &\quad 0 &\quad 0 &\quad 0 &\quad 0 &\quad 0 &\quad 0 &\quad 0 &\quad 0 \\
\end{align*}

\texttt{cm.get(“barney rubble“): 5}

\begin{align*}
5 &\quad 3 &\quad 0 &\quad 8 &\quad 0 &\quad 0 &\quad 5 &\quad 3 &\quad 0 \\
\end{align*}

\texttt{cm.add(“pebbles“, 2)}:

\begin{align*}
7 &\quad 3 &\quad 0 &\quad 10 &\quad 0 &\quad 0 &\quad 5 &\quad 5 &\quad 0 \\
\end{align*}
Count-min sketches

Equivalently, use a matrix, and each hash leads to a different row.

\[
\begin{align*}
&\text{cm.inc("fred flintstone", 3):} \\
&\begin{array}{ccc}
0 & 3 & 0 \\
3 & 0 & 0 \\
0 & 3 & 0 \\
\end{array}
\end{align*}
\]

\[
\begin{align*}
&\text{cm.inc("barney rubble", 5):} \\
&\begin{array}{ccc}
5 & 3 & 0 \\
8 & 0 & 0 \\
5 & 3 & 0 \\
\end{array}
\end{align*}
\]
LOCALITY SENSITIVE HASHING (LSH)
LSH: key ideas

• Goal:
  – map feature vector \( x \) to bit vector \( b_x \)
  – ensure that \( b_x \) preserves “similarity”
Random Projections
Random projections
Random projections

To make those points “close” we need to project to a direction orthogonal to the line between them.
Random projections

Any other direction will keep the distant points distant.

So if I pick a random \( r \) and \( r.x \) and \( r.x' \) are closer than \( \gamma \) then probably \( x \) and \( x' \) were close to start with.
**LSH: key ideas**

- **Goal:**
  - map feature vector \( \mathbf{x} \) to bit vector \( \mathbf{b}_\mathbf{x} \)
  - ensure that \( \mathbf{b}_\mathbf{x} \) preserves “similarity”

- **Basic idea:** use *random projections* of \( \mathbf{x} \)
  - Repeat many times:
    - Pick a random hyperplane \( \mathbf{r} \) by picking random weights for each feature (say from a Gaussian)
    - Compute the inner product of \( \mathbf{r} \) with \( \mathbf{x} \)
    - Record if \( \mathbf{x} \) is “close to” \( \mathbf{r} \) (\( \mathbf{r}.\mathbf{x} \geq 0 \))
      - the next bit in \( \mathbf{b}_\mathbf{x} \)
    - Theory says that if \( \mathbf{x}' \) and \( \mathbf{x} \) have small cosine distance then \( \mathbf{b}_\mathbf{x} \) and \( \mathbf{b}_\mathbf{x}' \) will have small Hamming distance
[Slides: Ben van Durme]
Hamming Distance := $h = 1$
Signature Length := $b = 6$

$$\cos(\theta) \approx \cos\left(\frac{h}{b} \pi\right)$$
$$= \cos\left(\frac{1}{6} \pi\right)$$
32 bit signatures

Approximate Cosine

True Cosine

Cheap

256 bit signatures

Approximate Cosine

True Cosine

Accurate

[Slides: Ben van Durme]
LSH applications

• Compact storage of data
  – and we can still compute similarities

• LSH also gives very fast approximations:
  – approx nearest neighbor method
    • just look at other items with $b\mathbf{x}' = b\mathbf{x}$
    • also very fast nearest-neighbor methods for Hamming distance

  – very fast clustering
    • cluster = all things with same $b\mathbf{x}$ vector
Online LSH and Pooling
LSH algorithm

• Naïve algorithm:
  – Initialization:
    • For i=1 to outputBits:
      – For each feature f:
        » Draw \( r(f,i) \sim \text{Normal}(0,1) \)
  – Given an instance \( \mathbf{x} \)
    • For i=1 to outputBits:
      \[
      \text{LSH}[i] = \begin{cases} 
      \sum (\mathbf{x}[f] \cdot r[i,f] \text{ for } f \text{ with non-zero weight in } \mathbf{x}) > 0 \? 1 : 0 
      \end{cases}
      \]
    • Return the bit-vector LSH
LSH algorithm

• But: storing the $k$ classifiers is expensive in high dimensions
  – For each of 256 bits, a dense vector of weights for every feature in the vocabulary
• Storing seeds and random number generators:
  – Possible but somewhat fragile
LSH: “pooling” (van Durme)

• Better algorithm:
  – Initialization:
    • Create a pool:
      – Pick a random seed $s$
      – For $i=1$ to $poolSize$:
        » Draw $pool[i] \sim \text{Normal}(0,1)$
    • For $i=1$ to $outputBits$:
      – Devise a random hash function $hash(i,f)$:
        » E.g.: $hash(i,f) = \text{hashcode}(f) \oplus \text{randomBitString}[i]$
  – Given an instance $x$
    • For $i=1$ to $outputBits$:
      $LSH[i] = \text{sum}(x[f] \times pool[hash(i,f) \mod poolSize] \text{ for } f \in x) > 0 \ ? 1 : 0$
    • Return the bit-vector $LSH$
The Pooling Trick

![Graph showing the relationship between Mean Absolute Error and Pool Size. The error decreases significantly as the pool size increases from $10^1$ to $10^5$.](image)
LSH: key ideas: pooling

• Advantages:
  – with pooling, this is a compact re-encoding of the data
    • you don’t need to store the r’s, just the pool
Locality Sensitive Hashing (LSH) in an On-line Setting
LSH: key ideas: online computation

- Common task: distributional clustering
  - for a word $w$, $v(w)$ is sparse vector of words that co-occur with $w$
  - cluster the $v(w)$’s

...guards at Pentonville prison in North London discovered that an escape attempt...

An American Werewolf in London is to be remade by the son of the original director...

...UK pop up shop on Monmouth Street in London today and on Friday the brand...

$v(London)$: Pentonville, prison, in, North, .... and, on Friday
LSH: key ideas: online computation

- Common task: distributional clustering
  - for a word \( w \), \( \mathbf{v}(w) \) is sparse vector of words that co-occur with \( w \)
  - cluster the \( \mathbf{v}(w) \)’s

\underline{London} is similar to:
\underline{Milan}.97, \underline{Madrid}.96, \underline{Stockholm}.96, \underline{Manila}.95, \underline{Moscow}.95

\underline{in} is similar to:
\underline{during}.99, \underline{on}.98, \underline{beneath}.98, \underline{from}.98, \underline{onto}.97

\underline{sold} is similar to:
\underline{deployed}.84, \underline{presented}.83, \underline{sacrificed}.82, \underline{held}.82, \underline{installed}.82
LSH: key ideas: online computation

• Common task: distributional clustering
  – for a word $w$, $v(w)$ is **sparse** vector of words that co-occur with $w$
  – cluster the $v(w)$’s

\( \vec{v} \in \mathbb{R}^d \)  
\( v \) is context vector; \( d \) is vocab size

\( \vec{r}_i \sim N(0, 1)^d \)  
\( r_i \) is \( i \)-th random projection

\[
h_i(\vec{v}) = \begin{cases} 
1 & \text{if } \vec{v} \cdot \vec{r}_i \geq 0, \\
0 & \text{otherwise.}
\end{cases}
\]

\( h_i(v) \) \( i \)-th bit of LSH encoding

if \( \vec{v} = \sum_j \vec{v}_j \)
then \( \vec{v} \cdot \vec{r}_i = \sum_j \vec{v}_j \cdot \vec{r}_i \)

because context vector is sum of mention contexts

these come one by one as we stream thru the corpus

\[
h_{it}(\vec{v}) = \begin{cases} 
1 & \text{if } \sum_{j}^t \vec{v}_j \cdot \vec{r}_i \geq 0, \\
0 & \text{otherwise.}
\end{cases}
\]

Online
Algorithm 1 Streaming LSH Algorithm

Parameters:
\( m \) : size of pool
\( d \) : number of bits (size of resultant signature)
\( s \) : a random seed
\( h_1, \ldots, h_d \) : hash functions mapping \( \langle s, f_i \rangle \) to \( \{0, \ldots, m - 1\} \)

Initialization:
1: Initialize floating point array \( P[0, \ldots, m - 1] \)
2: Initialize \( H \), a hashtable mapping words to floating point arrays of size \( d \)
3: for \( i := 0 \ldots m - 1 \) do
4: \( P[i] := \) random sample from \( N(0, 1) \), using \( s \) as seed

Online:
1: for each word \( w \) in the stream do
2: for each feature \( f_i \) associated with \( w \) do
3: for \( j := 1 \ldots d \) do
4: \( H[w][j] := H[w][j] + P[h_j(s, f_i)] \)

Signature Computation:
1: for each \( w \in H \) do
2: for \( i := 1 \ldots d \) do
3: if \( H[w][i] > 0 \) then
4: \( S[w][i] := 1 \)
5: else
6: \( S[w][i] := 0 \)
Experiment

• Corpus: 700M+ tokens, 1.1M distinct bigrams
• For each, build a feature vector of words that co-occur near it, using on-line LSH
• Check results with 50,000 actual vectors
Experiment
Closest based on true cosine

London

Milan.97, Madrid.96, Stockholm.96, Manila.95, Moscow.95

ASHER0, Champaign0, MANS0, NOBLE0, come0

Prague1, Vienna1, suburban1, synchronism1, Copenhagen2

Frankfurt4, Prague4, Taszar5, Brussels6, Copenhagen6

Prague12, Stockholm12, Frankfurt14, Madrid14, Manila14

Stockholm20, Milan22, Madrid24, Taipei24, Frankfurt25
Closest based on true cosine

London
Milan.97, Madrid.96, Stockholm.96, Manila.95, Moscow.95
ASHER0, Champaign0, MANS0, NOBLE0, come0
Prague1, Vienna1, suburban1, synchronism1, Copenhagen2

Closest based on 32 bit sig.’s

Cheap
Points to review

• APIs for:
  – Bloom filters, CM sketch, LSH

• Key applications of:
  – Very compact noisy sets
  – Efficient counters accurate for *large* counts
  – Fast approximate cosine distance

• Key ideas:
  – Uses of hashing that allow collisions
  – Random projection
  – Multiple hashes to control $\Pr(\text{collision})$
  – Pooling to compress a lot of random draws
A DEEP-LEARNING VARIANT OF LSH
DeepHash: Getting Regularization, Depth and Fine-Tuning Right

Jie Lin*,1,3, Olivier Morère*,1,2,3, Vijay Chandrasekhar1,3, Antoine Veillard2,3, Hanlin Goh1,3
I2R1, UPMC2, IPAL3

ICMR 2017
DeepHash

Image

Compact Bit Vector
64-1000 bits long
DeepHash

Image

Compact Bit Vector
64-1000 bits long

4k floats

LSH, …
DeepHash

Image

Deep Restricted Boltzmann Machine

4k floats

Compact Bit Vector
64-1000 bits long
An autoencoder

Deep Restricted Boltzmann Machine

A Deep Restricted Boltzmann Machine is closely related to an autoencoder

compact representation of \( x \)

input \( x \)

output \( y = x \)
DeepHash

A deeper autoencoder

Deep Restricted Boltzmann Machine is closely related to a deep autoencoder

Deep Restricted Boltzmann Machine

compact representation of $x$

input $x$

output $y=x$
DeepHash

Deep Restricted Boltzmann Machine

input $x$

compact representation of $x$

output $y = x$

Deep Restricted Boltzmann Machine is closely related to a deep autoencoder but the RBM is symmetric: weights $W$ from layer $j$ to $j+1$ are the transpose of weights from $j+1$ back to $j$.

RBM is also stochastic: compute $Pr(\text{hidden}|\text{visible})$, sample from that distribution, compute $Pr(\text{visible}|\text{hidden})$, sample, ...
DeepHash

This model is trained to compress image features, then reconstruct the image features from the reconstructions.

Another trick: regularize so that the representations are dense (about 50-50 “on” and “off” for an image) and each bit has 50-50 chance of being “on”

And then more training....
Training on matching vs non-matching pairs

Loss pushes the representation for “matching” pairs together and representation for non-matching pairs apart.
Training on matching vs non-matching pairs

Deep Siamese Network

Training Phase 2: Fine-Tuning

Matching & non-matching pairs

W₁ → W₂ → ... → Wₐ

Loss₁ → Loss₂ → ... → Lossₐ

margin-based matching loss
Training on matching vs non-matching pairs

Matching pairs were photos of the same “landmark”
DeepHash: Getting Regularization, Depth and Fine-Tuning Right

Jie Lin*,1,3, Olivier Morère*,1,2,3, Vijay Chandrasekhar1,3, Antoine Veillard2,3, Hanlin Goh1,3
I2R1, UPMC2, IPAL3

ICML 2017
DeepHash: Getting Regularization, Depth and Fine-Tuning Right

Jie Lin*1,3, Olivier Morère*1,2,3, Vijay Chandrasekhar1,3, Antoine Veillard2,3, Hanlin Goh1,3
I2R1, UPMC2, IPAL3

ICML 2017
old-school feature vector representation
CNN hidden layer representation