int getRandomNumber()
{
    return 4;  // chosen by fair dice roll.
    // guaranteed to be random.
}
BLOOM FILTERS/COUNTMIN SKETCHES - 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"): h1 h2 h3
0 1 1 0 0 0 0 0 1 0 0 0
```

```
bf.add("barney rubble"): h1 h2 h3
1 1 1 0 0 0 1 0 1 0 0 0
```

set several “random” bits
Bloom filters

```
1 1 1 0 0 1 0 1 0 0 0
```

`bf.contains ("fred flintstone"):`

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

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

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

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

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

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

\textbf{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

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

bf.add(“fred flintstone”):

set one random bit in each subrange

0 1 0 | 1 0 0 0 | 0 1 0

bf.add(“barney rubble”):

1 1 0 | 1 0 0 0 | 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 0 0

bf.contains(“fred flintstore”):

h1  h2  h3

1 1 0 | 1 0 0 | 1 1 0

return AND of all hashed bits

bf.contains(“pebbles”):

h1  h2  h3

1 1 0 | 1 0 0 | 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”):

1 1 0 | 1 0 0 | 1 1 0

a false positive!
Count-min sketches

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

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

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

cm.inc(“fred flintstone”, 3):

add the value to each hash location

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

cm.inc(“barney rubble”, 5):

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

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

cm.get("fred flintstone"): 3

cm.get("barney rubble"): 5
Count-min sketches

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

```plaintext
0 0 0 0 0 0 0 0 0
```

cm.get("barney rubble"): 5

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

cm.add("pebbles", 2):

```
7 3 0 10 0 0 5 5 0
```
Count-min sketches

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

\[
\begin{array}{ccc}
0 & 3 & 0 \\
3 & 0 & 0 \\
0 & 3 & 0 \\
\end{array}
\]

\[
\begin{array}{ccc}
5 & 3 & 0 \\
8 & 0 & 0 \\
5 & 3 & 0 \\
\end{array}
\]

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

cm.inc("barney rubble", 5):
Applications of Sketches
Applications of count-min sketches

• Keeping lots of lots of running totals
  – naïve Bayes event counters
  – word co-occurrence counters for semantic orientation, highly related words, ....
  – anywhere, in place of a large sparse feature vector
Applications of Bloom filters

• Replace a sparse set input to a deep-learning model
• Replace a sparse set output to a deep-learning model
• ...

Some uses of Bloom filters

• An example application
  – Finding items in “sharded” data
    • Easy if you know the sharding rule
    • Harder if you don’t (like Google n-grams)
Some Uses of Bloom filters

• An example application
  – Finding items in “sharded” data
    • Easy if you know the sharding rule
    • Harder if you don’t (like Google n-grams)

• Simple idea:
  – Build a BF of the contents of each shard
  – To look for key, load in the BF’s one by one, and search only the shards that probably contain key
  – Analysis: you won’t miss anything, you might look in some extra shards
  – You’ll hit $O(1)$ extra shards if you set $p=1/\#\text{shards}$
Some Uses of Bloom filters

• An example application
  – discarding singleton features from a classifier
• Scan through data once and check each w:
  – if bf1.contains(w): bf2.add(w)
  – else bf1.add(w)
• Now:
  – bf1.contains(w) ⇔ w appears >= once
  – bf2.contains(w) ⇔ w appears >= 2x
• Then train, ignoring words not in bf2
Some Uses of Bloom filters

• An example application
  – discarding rare features from a classifier
  – seldom hurts much, can speed up experiments
• Scan through data once and check each w:
  – if bf1.contains(w):
    • if bf2.contains(w): bf3.add(w)
    • else bf2.add(w)
  – else bf1.add(w)
• Now:
  – bf2.contains(w) ⇔ w appears >= 2x
  – bf3.contains(w) ⇔ w appears >= 3x
• Then train, ignoring words not in bf3
Some Uses of Bloom filters

• An example application
  – discarding rare features from a classifier
  – seldom hurts much, can speed up experiments
• Scan through data once and check each \( w \):
  – if \( \text{bf1.contains}(w) \):
    • if \( \text{bf2.contains}(w) \): \( \text{bf3.add}(w) \)
    • else \( \text{bf2.add}(w) \)
  – else \( \text{bf1.add}(w) \)
• Now:
  – \( \text{bf2.contains}(w) \) \( \iff \) \( w \) appears \( \geq 2x \)
  – \( \text{bf3.contains}(w) \) \( \iff \) \( w \) appears \( \geq 3x \)
• Then train, ignoring words not in \( \text{bf3} \)
LOCALITY SENSITIVE HASHING (LSH)
LSH: key ideas

• Goal:
  – map feature vector $\mathbf{x}$ to bit vector $\mathbf{b_x}$
  – ensure that $\mathbf{b_x}$ preserves “similarity”
**LSH: key ideas**

- **Goal:**
  - map feature vector $\mathbf{x}$ to bit vector $\mathbf{b_x}$
  - ensure that $\mathbf{b_x}$ preserves “similarity”

- **Different similarities require different hash schemes**
  - cosine distance: random projections
  - dot product: winner-take-all hash
  - Jaccard similarity: min-hash (Broder, 1997)
  - L2 distance: LSH with $p$-stable distributions
  - spherical LSH: ...
  - ...
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 $\mathbf{b}_x$
  – ensure that $\mathbf{b}_x$ 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” $\mathbf{r}$ ($\mathbf{r} \cdot \mathbf{x} \geq 0$)
      – the next bit in $\mathbf{b}_x$
    • Theory says that is $\mathbf{x}'$ and $\mathbf{x}$ have small cosine distance then $\mathbf{b}_x$ and $\mathbf{b}_x'$ will have small Hamming distance
Hamming Distance := \( h = 1 \)
Signature Length := \( b = 6 \)

\[ \cos(\theta) \approx \cos\left(\frac{h}{b}\pi\right) \]
\[ = \cos\left(\frac{1}{6}\pi\right) \]

[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 $bx’=bx$
    • also very fast nearest-neighbor methods for Hamming distance
  – very fast clustering
    • cluster = all things with same $bx$ vector
Online LSH and Pooling
Online Generation of Locality Sensitive Hash Signatures

Benjamin Van Durme and Ashwin Lall
LSH algorithm

- Naïve algorithm:
  - Initialization:
    - For \( i = 1 \) to \( \text{outputBits} \):
      - For each feature \( f \):
        » Draw \( r(f,i) \sim \text{Normal}(0,1) \)
  - Given an instance \( x \)
    - For \( i = 1 \) to \( \text{outputBits} \):
      \[
      \begin{align*}
      \text{LSH}[i] &= \text{\sum}(x[f] \cdot r[i,f] \text{ for } f \text{ with non-zero weight in } x) > 0 \\
      \text{? 1 : 0}
      \end{align*}
      \]
    - 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 $\text{hash}(i,f)$:
  - Given an instance $\mathbf{x}$
    - For $i=1$ to $outputBits$:
      $\text{LSH}[i] = \text{sum}(\mathbf{x}[f] \times \text{pool[\text{hash}(i,f) \mod poolSize]} \text{ for } f \text{ in } \mathbf{x}) > 0$ ?
      $1 : 0$
    - Return the bit-vector LSH
The Pooling Trick

![Graph showing mean absolute error vs pool size]
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…

\( v(\text{London}) \): 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 – or here, compute similarities

\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 $w$, $\mathbf{v}(w)$ is **sparse** vector of words that co-occur with $w$
  – cluster the $\mathbf{v}(w)$’s


$v(w)$ is very similar to a word embedding (eg, from word2vec or GloVE)
LSH: key ideas: online computation

• Construct LSH(\(v(w)\)) for each word \(w\) in the corpus
  – stream through the corpus once
  – use minimal amount of memory
  – ideally no more than size of output

…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…

\(v(\text{London})\): Pentonville, prison, in, North, …. and, on Friday
\(\text{LSH}(v(\text{London})): 0101101010111011101\)
\(\text{LSH}(v(\text{Monmouth})): 0111101000110010111\)

...
\( \vec{v} \in \mathbb{R}^d \) \hspace{1cm} v \text{ is context vector; } d \text{ is vocab size}

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

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

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

**Online**

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

Parameters:
m : size of pool
d : number of bits (size of resultant signature)
s : a random seed
$h_1, ..., h_d$ : 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: \hspace{1em} $P[i] :=$ random sample from $N(0, 1)$, using $s$ as seed

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

Signature Computation:
1: for each $w \in H$ do
2: \hspace{1em} for $i := 1 \ldots d$ do
3: \hspace{2em} if $H[w][i] > 0$ then
4: \hspace{3em} $S[w][i] := 1$
5: \hspace{2em} else
6: \hspace{3em} $S[w][i] := 0$

weight of $f_i$ in $r_j$

j-th hash of $f_i$
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
<table>
<thead>
<tr>
<th>London</th>
<th>ASHER, Champaign, MANS, NOBLE, come, Prague, Vienna, suburban, synchronism, Copenhagen, Frankfurt, Prague, Taszar, Brussels, Copenhagen, Prague, Stockholm, Frankfurt, Madrid, Manila, Stockholm, Milan, Madrid, Taipei, Frankfurt</th>
</tr>
</thead>
<tbody>
<tr>
<td>Milan.97</td>
<td>Madrid.96, Stockholm.96, Manila.95, Moscow.95</td>
</tr>
</tbody>
</table>

Closest based on true cosine
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(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
DeepHash

Image

Deep Restricted Boltzmann Machine

4k floats

Compact Bit Vector
64-1000 bits long
DeepHash

An autoencoder

Deep Restricted Boltzmann Machine

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 is closely related to a deep autoencoder for compact representation of $x$. 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.

A 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

Learned deep RBM

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

Training Phase 2: Fine-Tuning

Deep Siamese Network

Matching & non-matching pairs

Loss\_1 \quad Loss\_2 \quad Loss\_L

W_1 \quad W_2 \quad \ldots \quad W_L

Transfer model

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 More*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

[Image of a cat]

Trained DeepHash Model

Image Descriptor Hashing (Testing)

Compact Binary Hash

threshold

64-1K bits

ICML 2017
old-school feature vector representation
CNN encoding of image

CNN hidden layer representation
Picking random classifiers (with LSH) is surprisingly close in performance to DeepHash --- when the similarity metric you start with is good