
No quiz today!

# Randomized Algorithms - 2

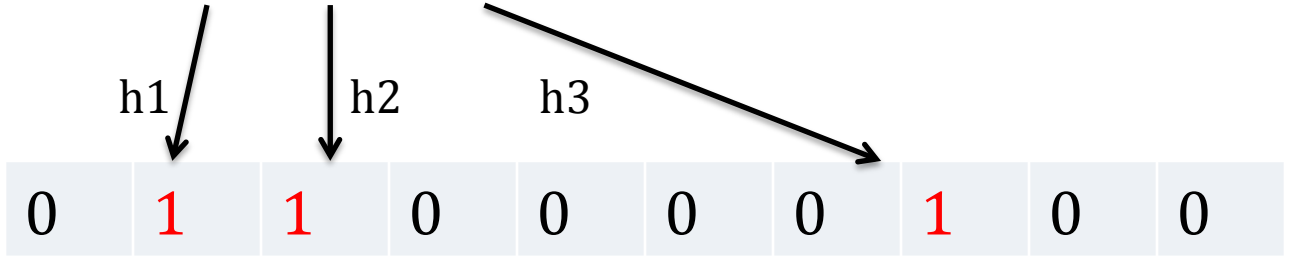

# BLOOM FILTERS - RECAP

# Bloom filters

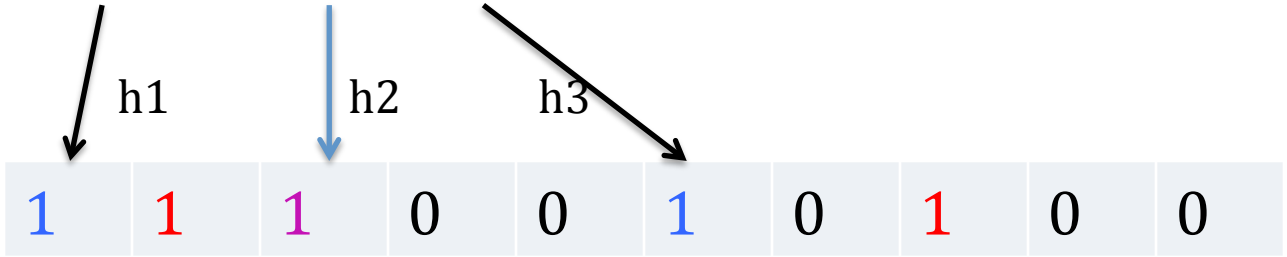

- Interface to a Bloom filter
  - BloomFilter(int maxSize, double p);
  - void bf.add(String s); // insert s
  - bool bd.contains(String s);
    - // If s was added return true;
    - // else with probability at least *1-p* return false;
    - // else with probability at most *p* return true;
  - I.e., a noisy "set" where you can test membership (and that's it)

# Bloom filters


bf.add("fred flintstone"):

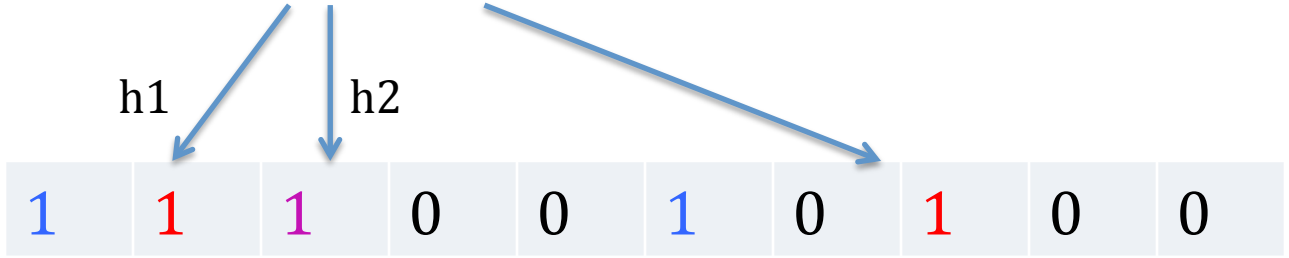



bf.add("barney rubble"):

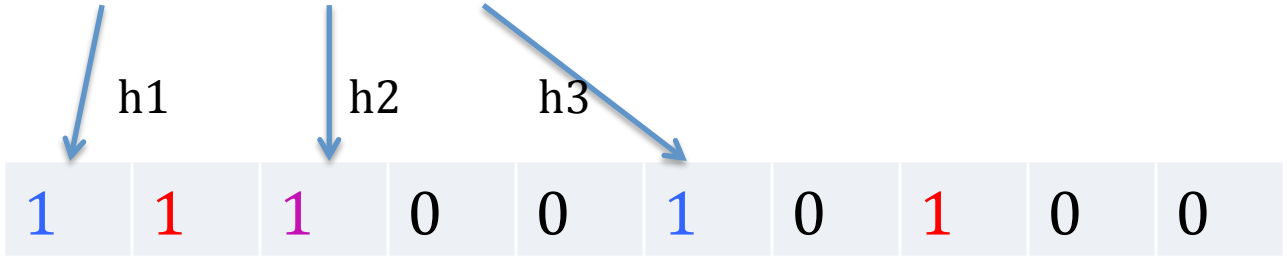


# Bloom filters


bf.contains ("fred flintstone"):

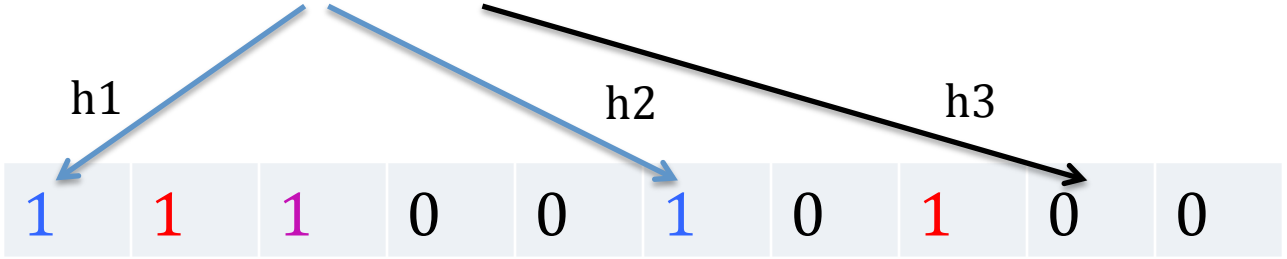



bf.contains("barney rubble"):

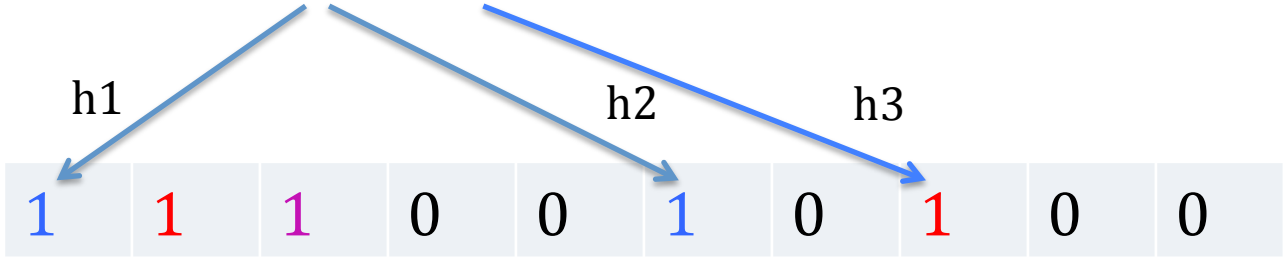


# Bloom filters


bf.contains("wilma flintstone"):




bf.contains("wilma flintstone"):



# Bloom filters - recap


- An implementation
  - Allocate M bits, bit[0]…,bit[1-M]
  - Pick K hash functions hash(1,2),hash(2,s),….
    - E.g: hash(i,s) = hash(s+ randomString[i])
  - To add string s:
    - For i=1 to k, set bit[hash(i,s)] = 1
  - To check contains(s):
    - For i=1 to k, test bit[hash(i,s)]
    - Return “true” if they’re all set; otherwise, return “false”
  - M and K
    - set carefully to obtain right false positive rate
- Sample code and sample applications…

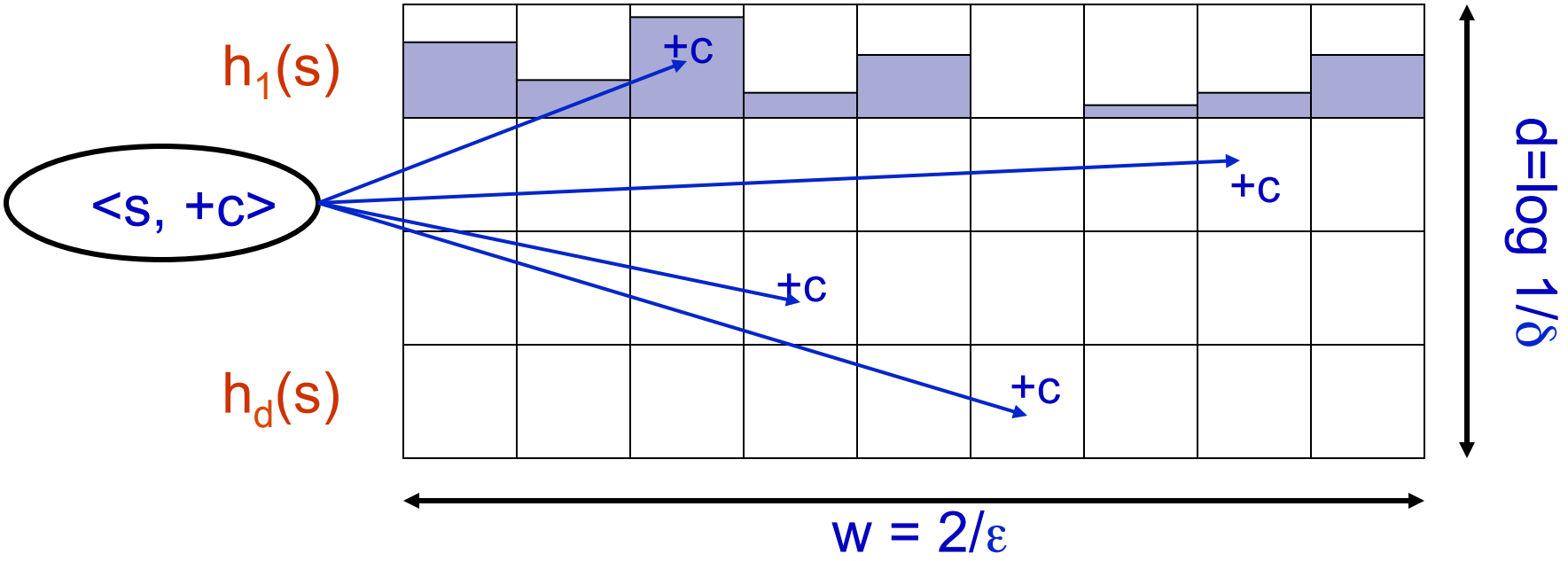

# THE COUNT-MIN SKETCH: RECAP

# Bloom Filter→Count-min sketch


- An implementation
  - Allocate a matrix *CM* with *d* rows, *w* columns
  - Pick *d* hash functions $h_1(s), h_2(s), \ldots$
  - To increment counter *A[s]* for *s* by *c*
    - For *i*=1 to *d*, set *CM[i, hash(i,s)] += c*
  - To retrieve value of *A[s]*:
    - For *i*=1 to *d*, retrieve *M[i, hash(i,s)]*
    - Return **minimum** of these values
  - Similar idea as Bloom filter:
    - if there are *d* collisions, you return a value that's too large; otherwise, you return the correct value.

Question: what does this look like if d=1?

# CM Sketch Structure



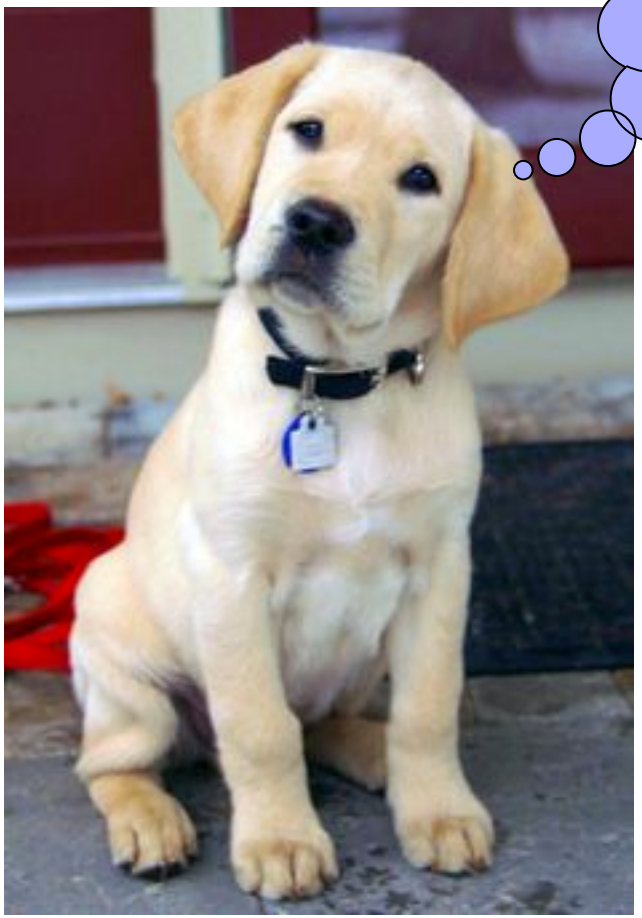

- Each string is mapped to one bucket per row
- Estimate A[j] by taking $\min_k \{ CM[k,h_k(j)] \}$
- Errors are always over-estimates
- Analysis: d=log 1/δ, w=2/ε ➔ error is usually less than $\varepsilon||A||_1$

i.e. with prob > 1-δ

# CM Sketch Guarantees


- *[Cormode, Muthukrishnan'04]* CM sketch guarantees approximation error on point queries less than $\varepsilon||A||_1$ in space $O(1/\varepsilon \log 1/\delta)$
- CM sketches are also accurate for *skewed* values---i.e., only a few entries *s* with large *A[s]*
  - "Finding heavy hitters"
- Application:
  - finding counts for words *x,y* that *frequently* co-occur ➔ compute "semantic orientation" of words
  - some others later on
- A disadvantage:
  - CM is harder to tune than Bloom filters


# LOCALITY SENSITIVE HASHING (LSH)

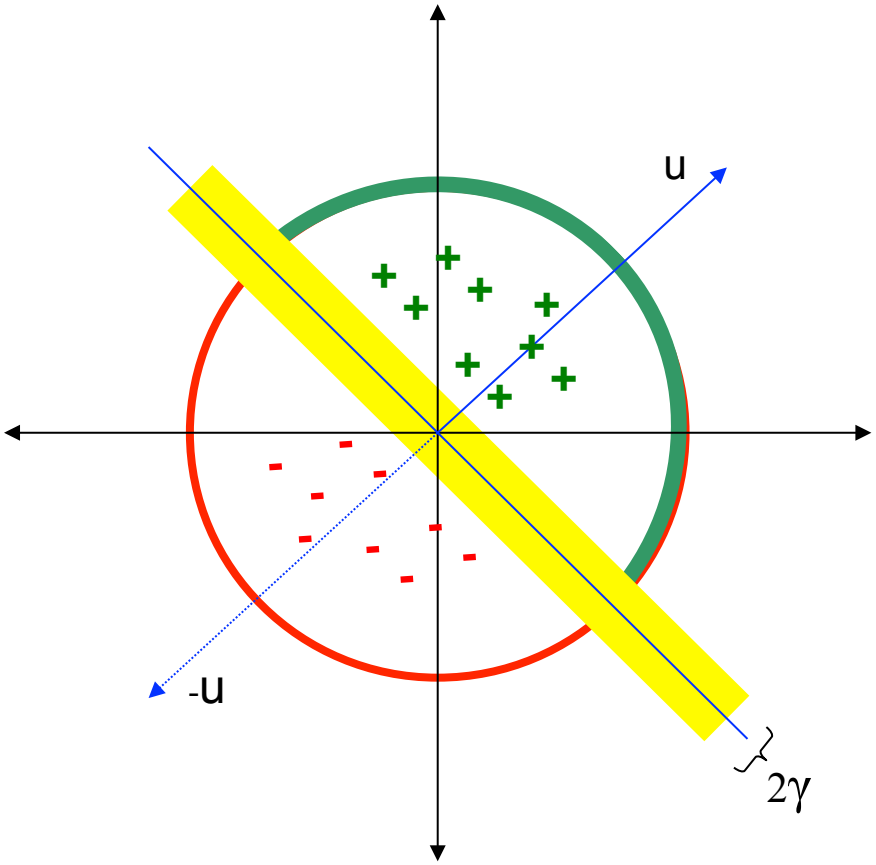

Wait, did he say locality sensitive hash browns?

# LSH: key ideas

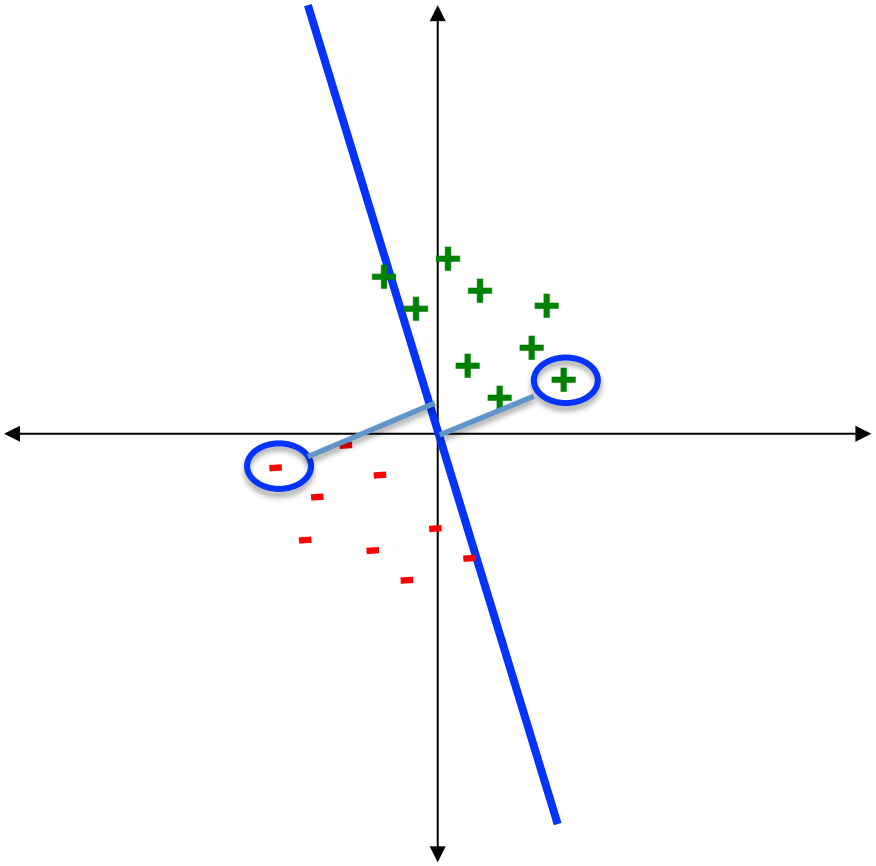

- Goal:
  - map feature vector **x** to bit vector **bx**
  - ensure that **bx** preserves "similarity"

# Random Projections

# Random projections

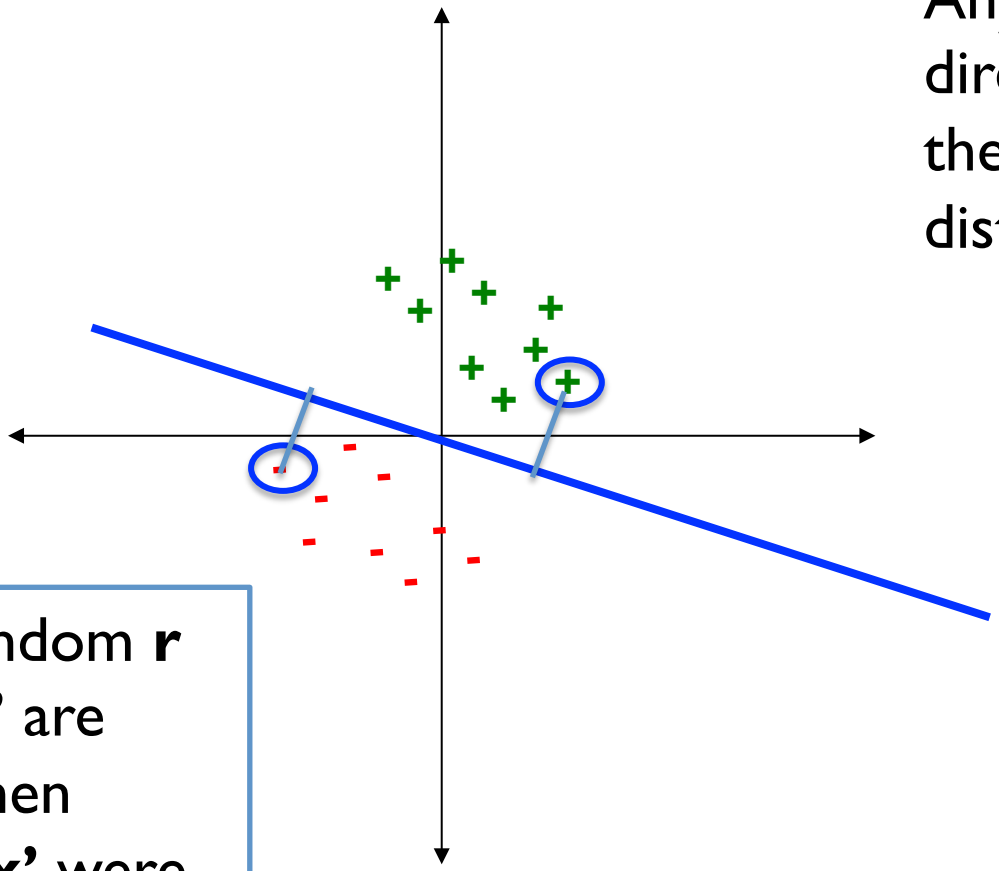


# Random projections



To make those points "close" we need to project to a direction orthogonal to the line between them

# Random projections

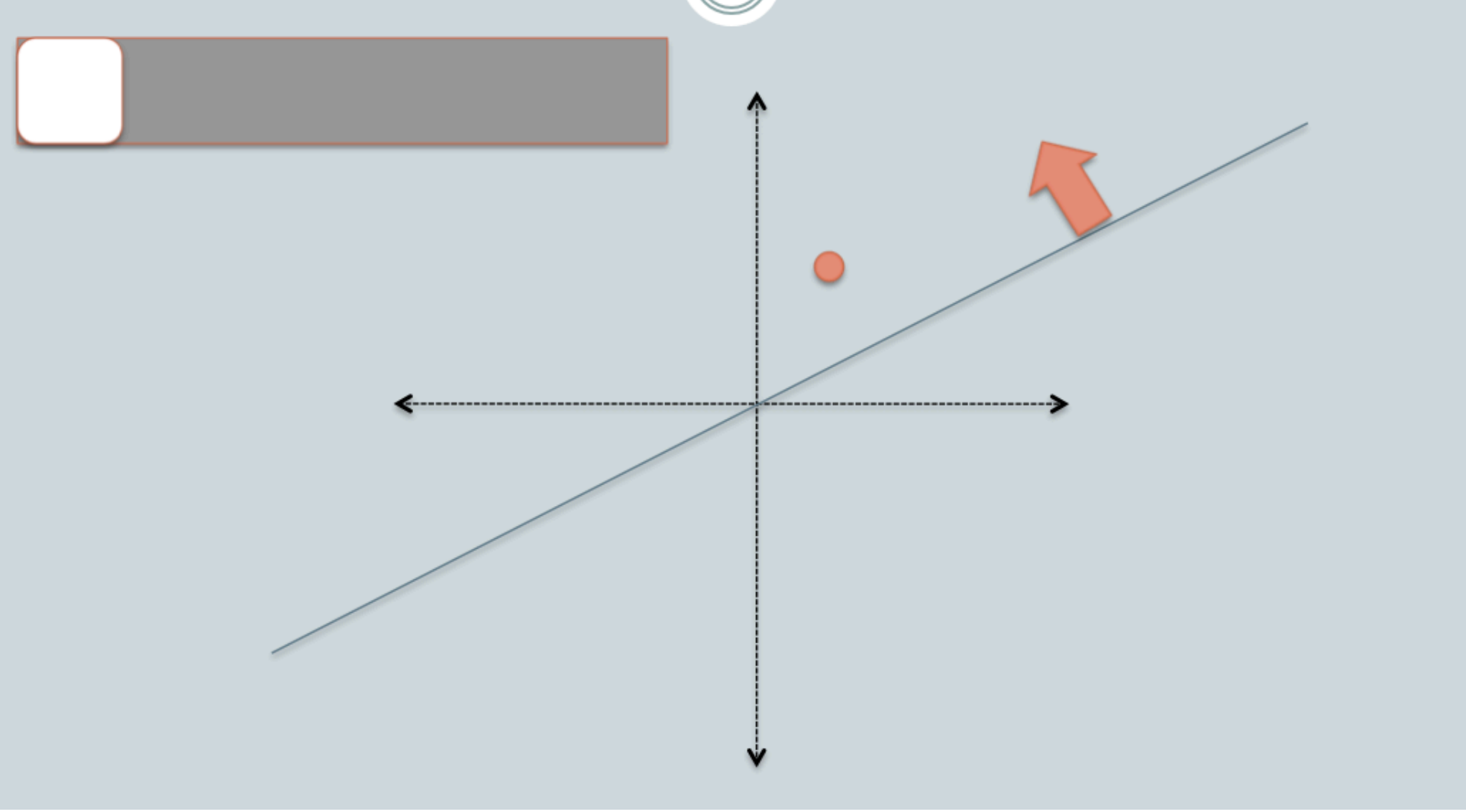


Any other direction will keep the distant points distant.

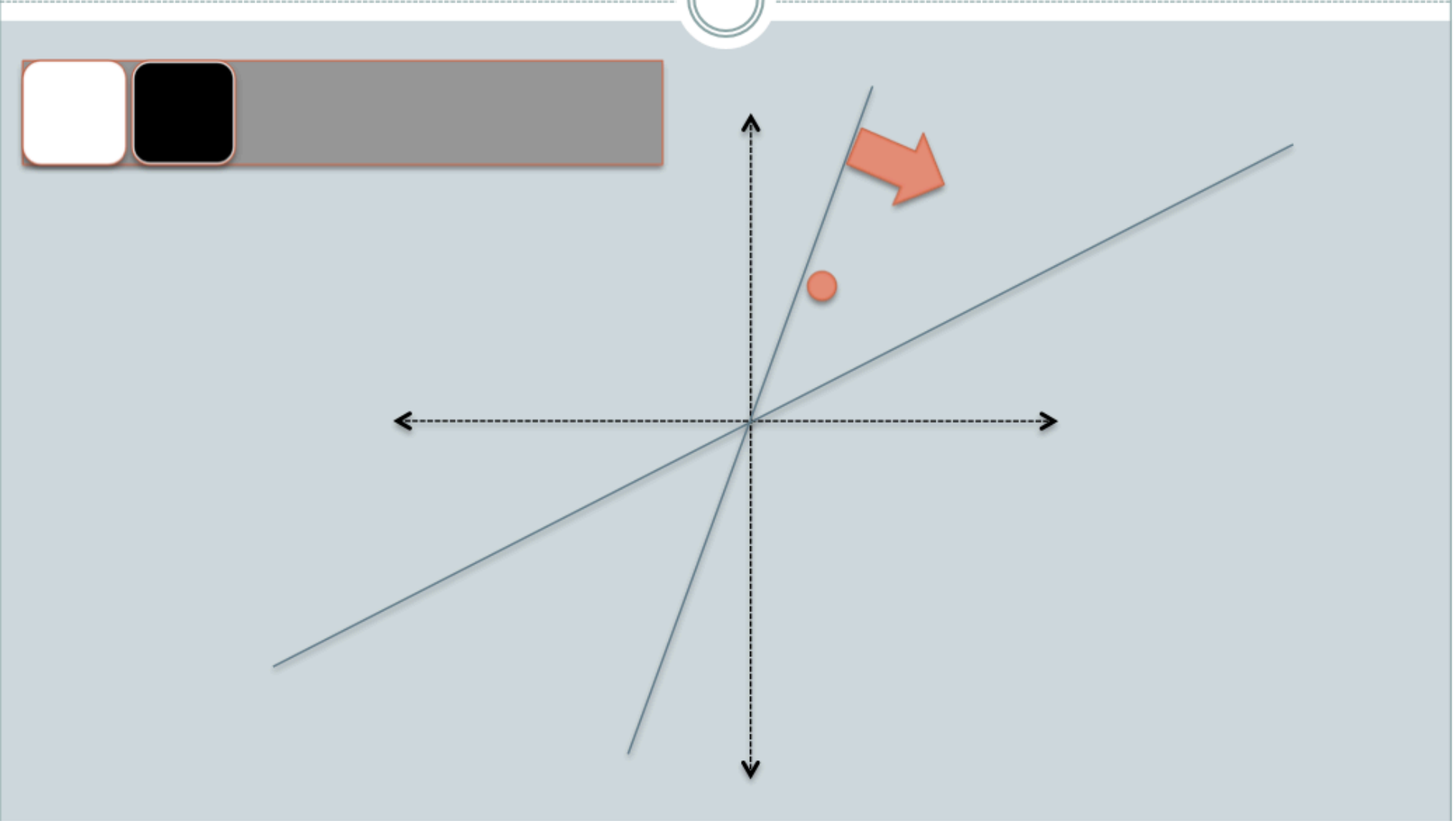
So if I pick a random **r** and **r.x** and **r.x'** are closer than γ then *probably* **x** and **x'** were close to start with.

# LSH: key ideas

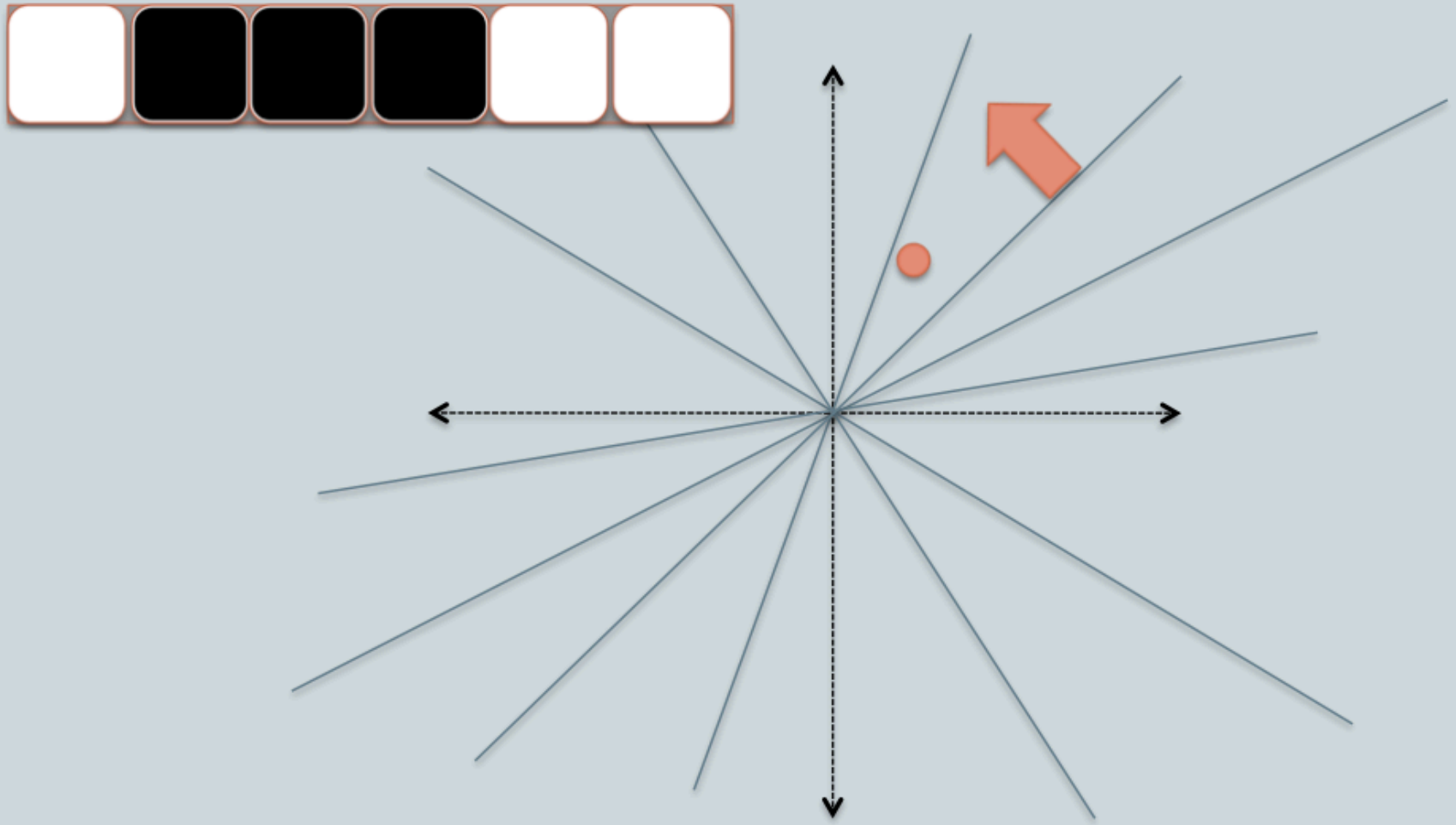

- Goal:
  - map feature vector **x** to bit vector **bx**
  - ensure that **bx** preserves "similarity"
- Basic idea: use *random projections* of **x**
  - Repeat many times:
    - Pick a random hyperplane **r** by picking random weights for each feature (say from a Gaussian)
    - Compute the inner product of **r** with **x**
    - Record if **x** is "close to" **r** (**r.x**>=0)
      - the next bit in **bx**
    - Theory says that is **x'** and **x** have small cosine distance then **bx** and **bx'** will have small Hamming distance



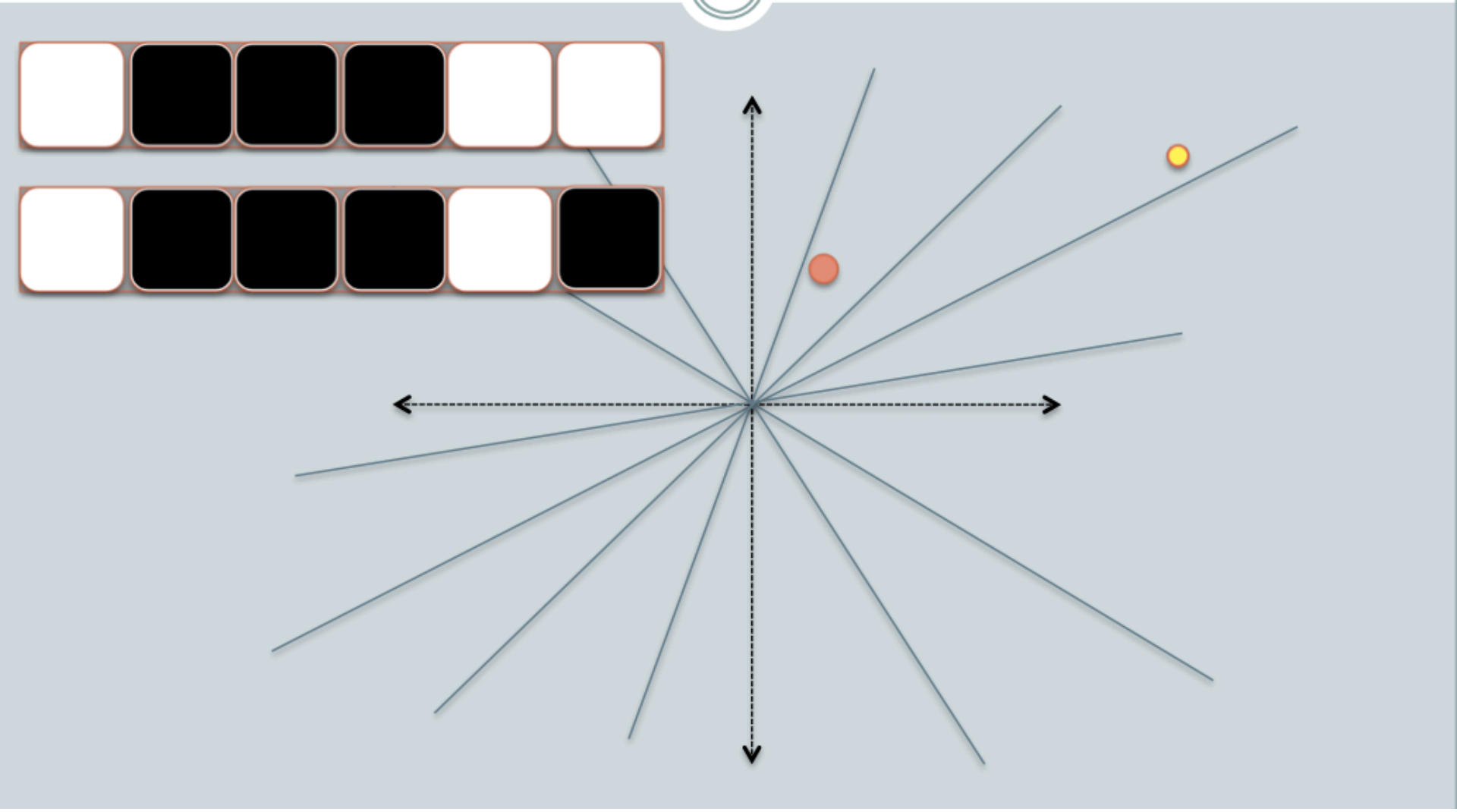
[Slides: Ben van Durme]



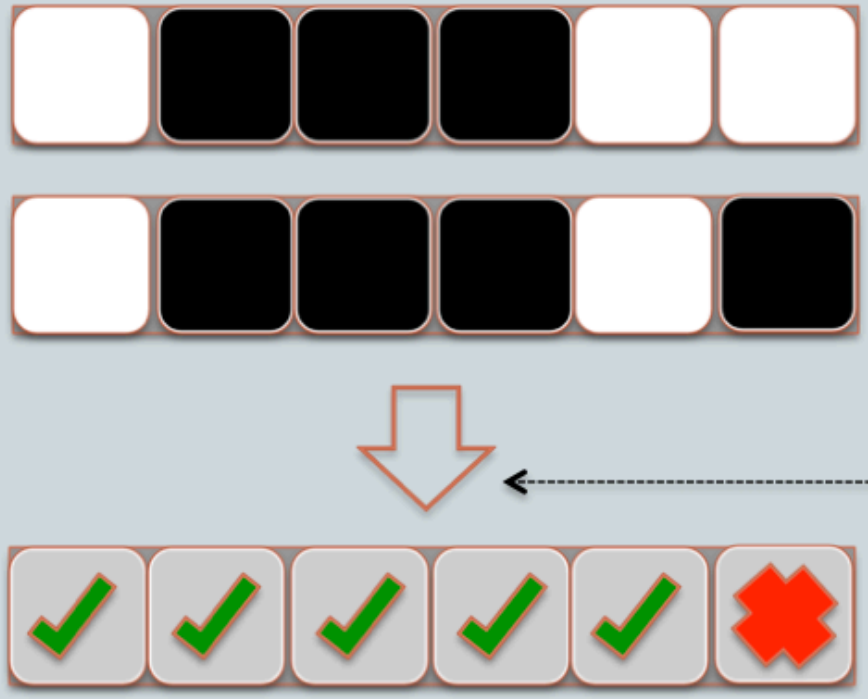
[Slides: Ben van Durme]



[Slides: Ben van Durme]



[Slides: Ben van Durme]



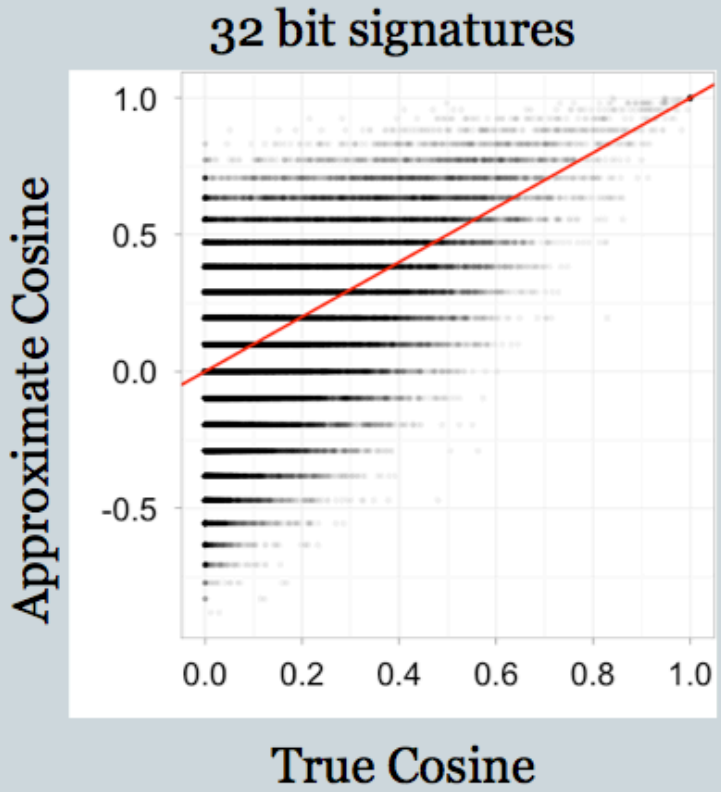
Hamming Distance $:= h = 1$

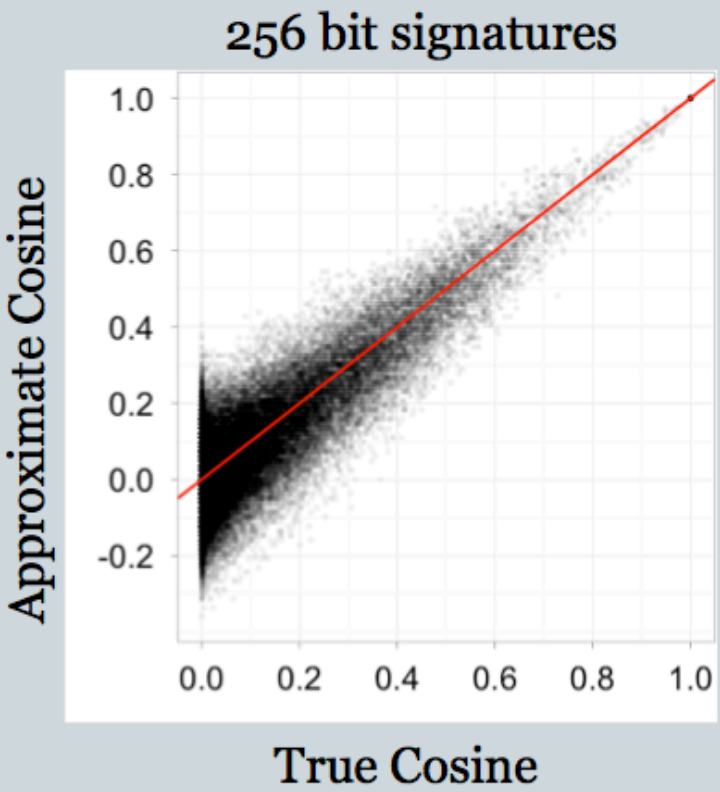
Signature Length $:= b = 6$

$$\cos(\theta) \approx \cos\left(\frac{h}{b}\pi\right)$$

[Slides: Ben van Durme]

$$= \cos\left(\frac{1}{6}\pi\right)$$




**Cheap**

**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 **bx'=bx**
    - also very fast nearest-neighbor methods for Hamming distance
  - very fast clustering
    - cluster = all things with same **bx** vector

# Online Generation of Locality Sensitive Hash Signatures

Benjamin Van Durme and Ashwin Lall

human language technology
center of excellence
JOHNS HOPKINS
UNIVERSITY

DENISON
UNIVERSITY

# LSH algorithm

- Naïve algorithm:
  - Initialization:
    - For i=1 to outputBits:
      - For each feature *f:*
        - » Draw r(f,i) ~ Normal(0,1)
  - Given an instance **x**
    - For i=1 to outputBits:

      LSH[i] =

      sum(**x**[*f*]*r[i,*f*] for *f* with non-zero weight in **x**) > 0 ? 1 : 0
    - 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

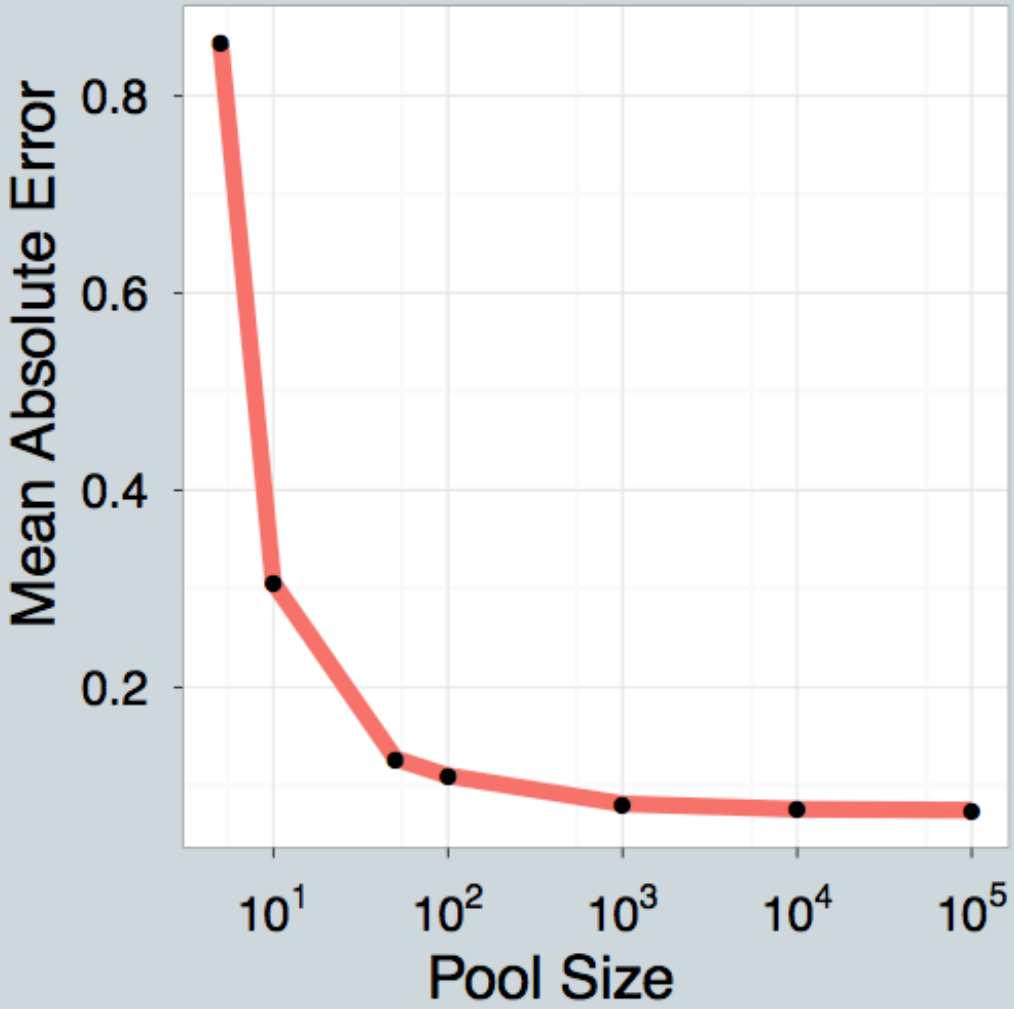
# LSH: "pooling" (van Durme)

- Better algorithm:
  - Initialization:
    - Create a pool:
      - Pick a random seed *s*
      - For i=1 to poolSize:
        - » Draw pool[i] ~ Normal(0,1)
    - For i=1 to outputBits:
      - Devise a random hash function hash(i,*f):*
        - » E.g.: hash(i,f) = hashcode(f) XOR randomBitString[i]
  - Given an instance **x**
    - For i=1 to outputBits:

      LSH[i] = sum(

      **x**[f] * pool[hash(i,f) % poolSize] for *f* in **x**) > 0 ? 1 : 0
    - Return the bit-vector LSH

# The Pooling Trick



# 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 $w$, $\boldsymbol{x}(w)$ is sparse vector of words that co-occur with $w$
  - cluster the $w$'s

$$\vec{v} \in \mathbb{R}^d$$

$$\vec{r}_i \sim N(0,1)^d$$

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

$$\text{if } \vec{v} = \Sigma_j \vec{v}_j$$
$$\text{then } \vec{v} \cdot \vec{r}_i = \Sigma_j \vec{v}_j \cdot \vec{r}_i$$

Break into local products

**Online**

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


## 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:

```
Initialize floating point array P[0, ..., m − 1]
Initialize H, a hashtable mapping words to floating point
    arrays of size d
for i := 0 ... m − 1 do
    P[i] := random sample from N(0, 1), using s as seed
```

ONLINE:

```
for each word w in the stream do
    for each feature f_i associated with w do
        for j := 1 ... d do
            H[w][j] := H[w][j] + P[h_j(s, f_i)]
```

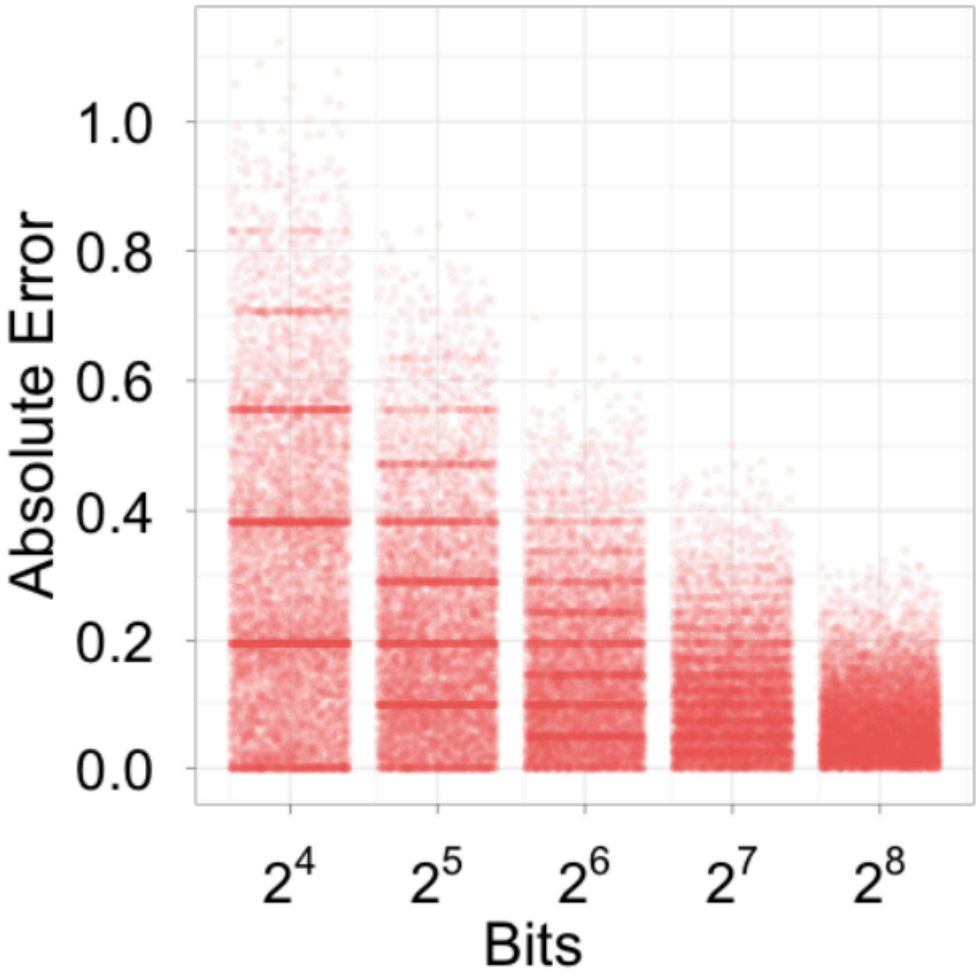
SIGNATURECOMPUTATION:

```
for each w ∈ H do
    for i := 1 ... d do
        if H[w][i] > 0 then
            S[w][i] := 1
        else
            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**$_{.95}$

ASHER$_0$, Champaign$_0$, MANS$_0$, NOBLE$_0$, come$_0$

Prague$_1$, Vienna$_1$, suburban$_1$, synchronism$_1$, Copenhagen$_2$

**London**

**Milan**$_{.97}$, **Madrid**$_{.96}$, **Stockholm**$_{.96}$, **Manila**$_{.95}$, **Moscow**$_{.95}$

ASHER$_0$, Champaign$_0$, MANS$_0$, NOBLE$_0$, come$_0$

Prague$_1$, Vienna$_1$, suburban$_1$, synchronism$_1$, Copenhagen$_2$

Frankfurt$_4$, Prague$_4$, Taszar$_5$, Brussels$_6$, Copenhagen$_6$

Prague$_{12}$, Stockholm$_{12}$, Frankfurt$_{14}$, Madrid$_{14}$, Manila$_{14}$

Stockholm$_{20}$, Milan$_{22}$, Madrid$_{24}$, Taipei$_{24}$, Frankfurt$_{25}$

Closest based on 32 bit sig.'s

**Cheap**

# Locality Sensitive Hashing (LSH) and Pooling Random Values

# LSH algorithm

- Naïve algorithm:
  - Initialization:
    - For i=1 to outputBits:
      - For each feature *f:*
        - » Draw r(f,i) ~ Normal(0,1)
  - Given an instance **x**
    - For i=1 to outputBits:

      LSH[i] =

      sum(**x**[*f*]*r[i,*f*] for *f* with non-zero weight in **x**) > 0 ? 1 : 0
    - 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

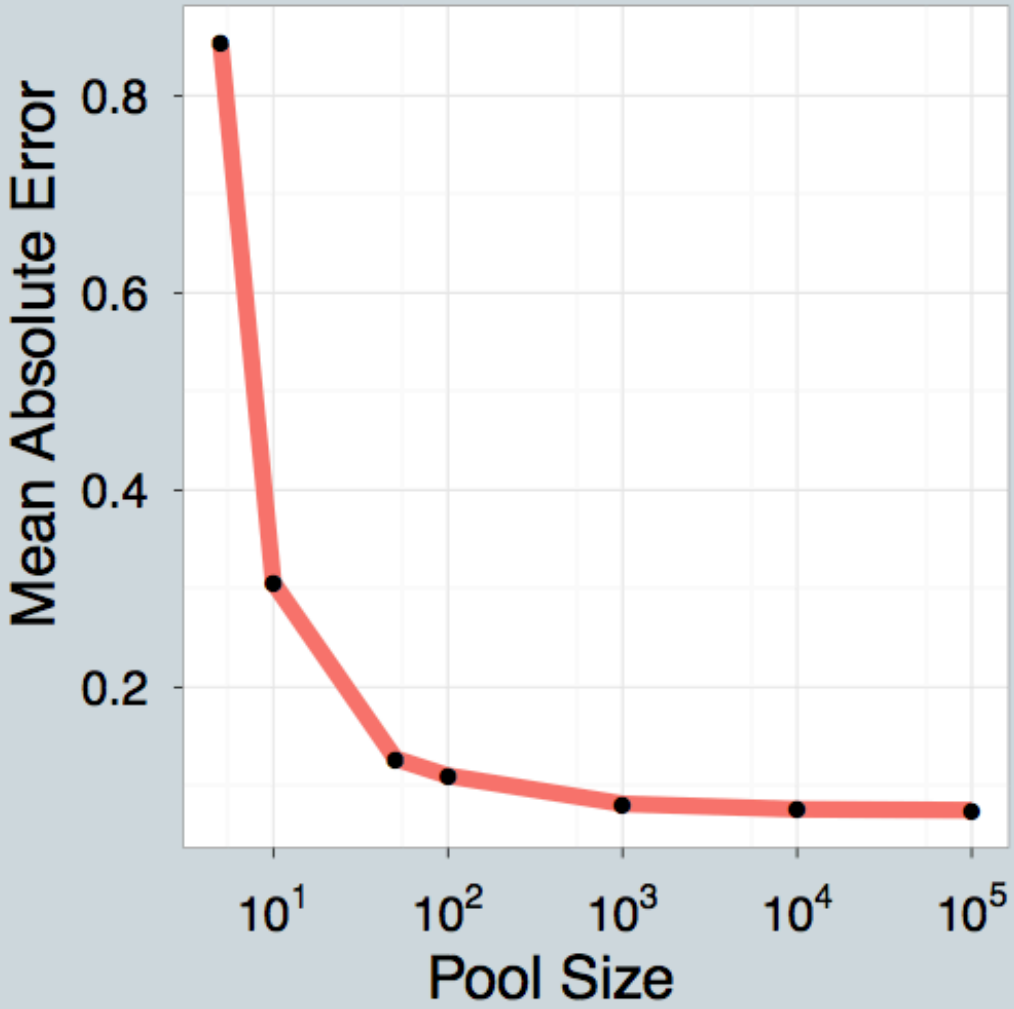
# LSH: "pooling" (van Durme)

- Better algorithm:
  - Initialization:
    - Create a pool:
      - Pick a random seed *s*
      - For i=1 to poolSize:
        - » Draw pool[i] ~ Normal(0,1)
    - For i=1 to outputBits:
      - Devise a random hash function hash(i,*f):*
        - » E.g.: hash(i,f) = hashcode(f) XOR randomBitString[i]
  - Given an instance **x**
    - For i=1 to outputBits:

      LSH[i] = sum(

      **x**[f] * pool[hash(i,f) % poolSize] for *f* in **x**) > 0 ? 1 : 0
    - Return the bit-vector LSH

# The Pooling Trick



# 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 $w$, $\boldsymbol{x}(w)$ is sparse vector of words that co-occur with $w$
  - cluster the $w$'s

$$\vec{v} \in \mathbb{R}^d$$

$$\vec{r}_i \sim N(0,1)^d$$

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

$$\text{if } \vec{v} = \Sigma_j \vec{v}_j$$
$$\text{then } \vec{v} \cdot \vec{r}_i = \Sigma_j \vec{v}_j \cdot \vec{r}_i$$

Break into local products

Online

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


## 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: $\quad P[i] :=$ random sample from $N(0, 1)$, using $s$ as seed

ONLINE:

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

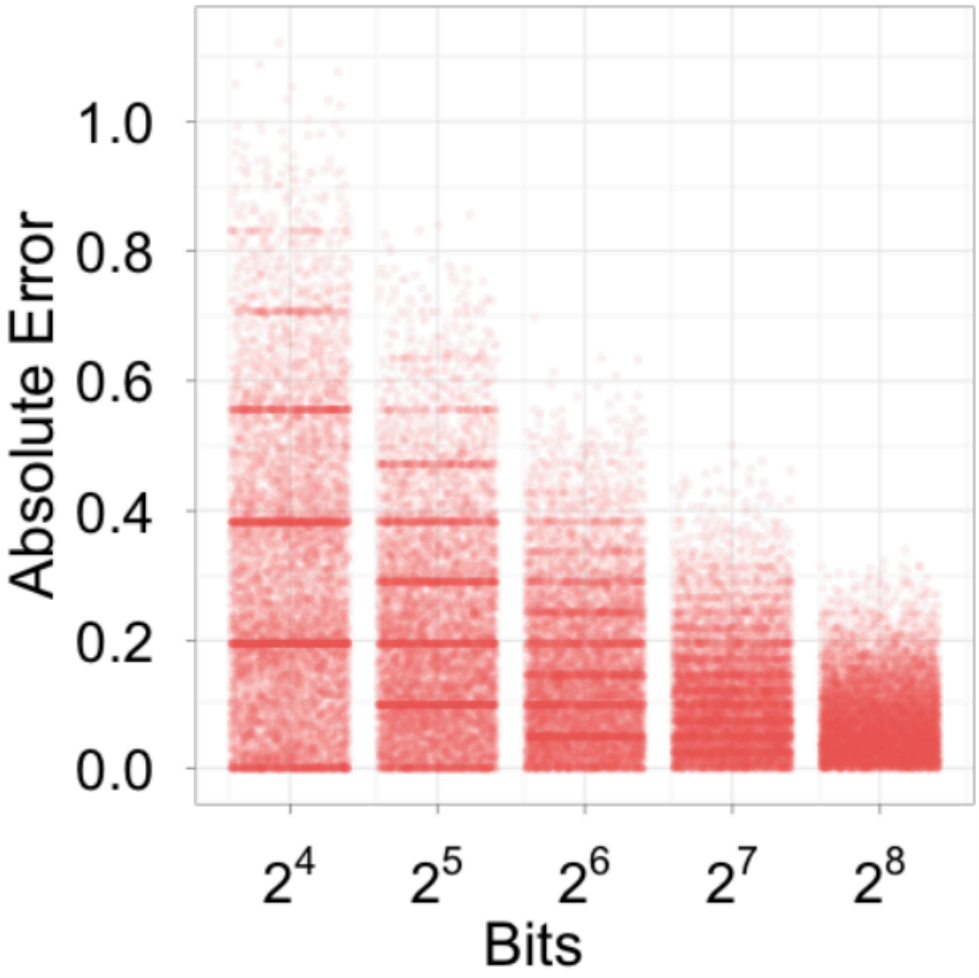
SIGNATURECOMPUTATION:

1: **for** each $w \in H$ **do**
2: $\quad$ **for** $i := 1 \ldots d$ **do**
3: $\quad\quad$ **if** $H[w][i] > 0$ **then**
4: $\quad\quad\quad S[w][i] := 1$
5: $\quad\quad$ **else**
6: $\quad\quad\quad S[w][i] := 0$

# 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**$_{.95}$
ASHER$_0$, Champaign$_0$, MANS$_0$, NOBLE$_0$, come$_0$
Prague$_1$, Vienna$_1$, suburban$_1$, synchronism$_1$, Copenhagen$_2$

**London**

**Milan**$_{.97}$, **Madrid**$_{.96}$, **Stockholm**$_{.96}$, **Manila**$_{.95}$, **Moscow**$_{.95}$
ASHER$_0$, Champaign$_0$, MANS$_0$, NOBLE$_0$, come$_0$
Prague$_1$, Vienna$_1$, suburban$_1$, synchronism$_1$, Copenhagen$_2$
Frankfurt$_4$, Prague$_4$, Taszar$_5$, Brussels$_6$, Copenhagen$_6$
Prague$_{12}$, Stockholm$_{12}$, Frankfurt$_{14}$, Madrid$_{14}$, Manila$_{14}$
Stockholm$_{20}$, Milan$_{22}$, Madrid$_{24}$, Taipei$_{24}$, Frankfurt$_{25}$

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(collision)
  - Pooling to compress a lot of random draws