Randomized Algorithms

William Cohen

Outline

- Randomized methods: today
  - SGD with the hash trick (recap)
  - Bloom filters
- Later:
  - count-min sketches
  - locality sensitive hashing

THE HASH TRICK: A REVIEW

Hash Trick - Insights

- Save memory: don’t store hash keys
- Allow collisions
  - even though it distorts your data some
- Let the learner (downstream) take up the slack

Learning as optimization for regularized logistic regression

- Algorithm: $$w^j = w^j + \lambda(y - p)x^j - \lambda 2\mu w^j$$
- Initialize arrays *W, A* of size *R* and set *k=0*
- For each iteration t=1,…T
  - For each example $(\mathbf{x}_i, y_i)$
    - Let V be hash table so that $V[h] = \sum_{j:hash(j)\%R==h} x_i^j$
    - $p_i = \ldots$; $k{+}{+}$
    - For each hash value h: $V[h]>0$:
      - » $W[h] \mathrel{*}= (1 - \lambda 2\mu)^{k-A[j]}$
      - » $W[h] = W[h] + \lambda(y_i - p^i)V[h]$
      - » $A[h] = k$

Learning as optimization for regularized logistic regression

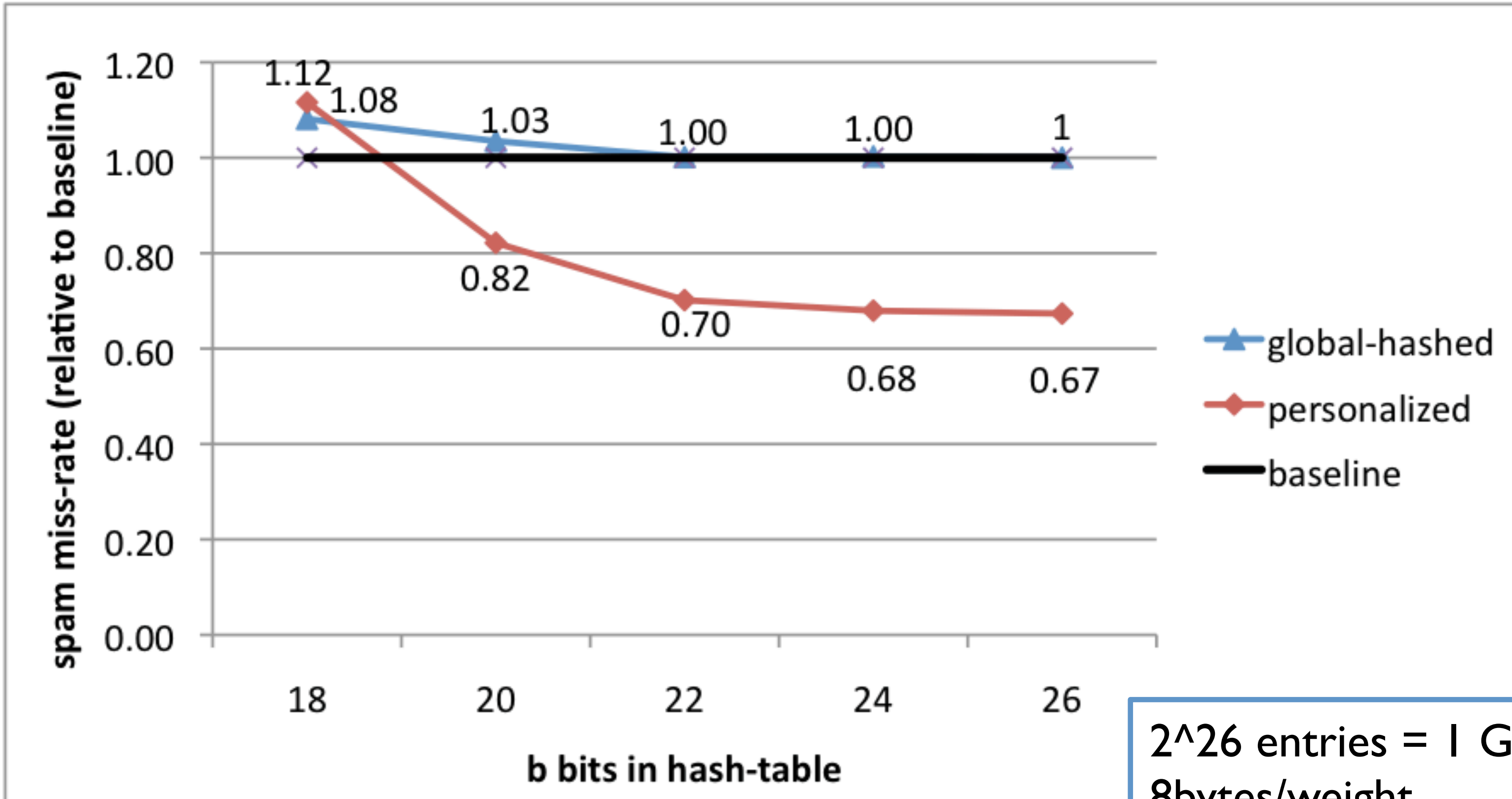
- Initialize arrays *W, A* of size *R* and set *k=0*
- For each iteration t=1,…T
  - For each example $(\mathbf{x}_i, y_i)$
    - *k++;* let *V* be a new array of size R; let *tmp=0*
    - For each j: $\mathbf{x}_i^j > 0$: $V[hash(j)\%R] += \mathbf{x}_i^j$
    - Let *ip=0*
    - For each *h: V[h]>0:*
      - $W[h] \;{*}{=}\; (1 - \lambda 2\mu)^{k-A[h]}$
      - *ip+= V[h]*W[h]*
      - *A[h] = k*
    - *p = 1/(1+exp(-ip))*
    - For each *h: V[h]>0:*
      - $W[h] = W[h] + \lambda(y_i - p^i)V[h]$

$$V[h] = \sum_{j:hash(j)\%R==h} x_i^j$$

regularize W[h]'s

$$p \equiv \frac{1}{1+e^{-\mathbf{V}\cdot\mathbf{w}}}$$

$$w^j = w^j + \lambda(y-p)x^j - \lambda 2\mu w^j$$



2^26 entries = 1 Gb @ 8bytes/weight

Figure 2. The decrease of uncaught spam over the baseline classifier averaged over all users. The classification threshold was chosen to keep the not-spam misclassification fixed at 1%. The hashed global classifier (*global-hashed*) converges relatively soon, showing that the distortion error ϵ_d vanishes. The personalized classifier results in an average improvement of up to 30%.

Data Sets	**#Train**	**#Test**	**#Labels**
RCV1	781,265	23,149	2
Dmoz L2	4,466,703	138,146	575
Dmoz L3	4,460,273	137,924	7,100

Table 1: Text data sets. #X denotes the number of observations in X.

	HLF (2^{28})		**HLF** (2^{24})		**HF**		**no hash**	**U base**	**P base**
	error	mem	error	mem	error	mem	mem	error	error
L2	30.12	2G	30.71	0.125G	31.28	2.25G (2^{19})	7.85G	99.83	85.05
L3	52.10	2G	53.36	0.125G	51.47	1.73G (2^{15})	96.95G	99.99	86.83

Table 5: Misclassification and memory footprint of hashing and baseline methods on DMOZ. HLF: joint hashing of labels and features. HF: hash features only. no hash: direct model (not implemented as too large, hence only memory estimates—we have 1,832,704 unique words). U base: baseline of uniform classifier. P base: baseline of majority vote. mem: memory used for the model. Note: the memory footprint in HLF is essentially independent of the number of classes used.

MOTIVATING BLOOM FILTERS

A variant of feature hashing

- Hash each feature *multiple times* with different hash functions
- Now, each *w* has *k* chances to *not* collide with another useful *w'*
- An easy way to get multiple hash functions
  - Generate some random strings $s_1,\ldots,s_L$
  - Let the k-th hash function for *w* be the ordinary hash of concatenation $w \bullet s_k$

$$V[h] = \sum_{k} \sum_{j:hash(j \cdot s_k)\%R=h} x_i^j$$

A variant of feature hashing

a!=**b** are binary vectors

V(**a**)	1	0	1	0
V(**b**)	1	0	1	0

V(**a**)	1	0	1	0
	0	1	1	0
V(**b**)	1	0	1	0
	1	1	0	0

1	1	2	0
1	1	1	0

- An easy way to get multiple hash functions
  - Generate some random strings $s_1,\ldots,s_L$
  - Let the k-th hash function for w be the ordinary hash of concatenation $w \bullet s_k$

$$V[h] = \sum_k \sum_{j:hash(j \cdot s_k)\%R=h} x_i^j$$

A variant of feature hashing

- Why would this work?

$$V[h] = \sum_{k} \sum_{j:hash(j \cdot s_k)\%R=h} x_i^j$$

- Claim: with 100,000 features and 100,000,000 buckets:
  - k=1 ➔ Pr(any feature duplication) ≈1
  - k=2 ➔ Pr(any feature duplication) ≈0.4
  - k=3 ➔ Pr(any feature duplication) ≈0.01

Hash Trick - Insights

- Save memory: don’t store hash keys
- Allow collisions
  - even though it distorts your data some
- Let the learner (downstream) take up the slack

- Here’s another famous trick that exploits these insights….

BLOOM FILTERS

Bloom filters

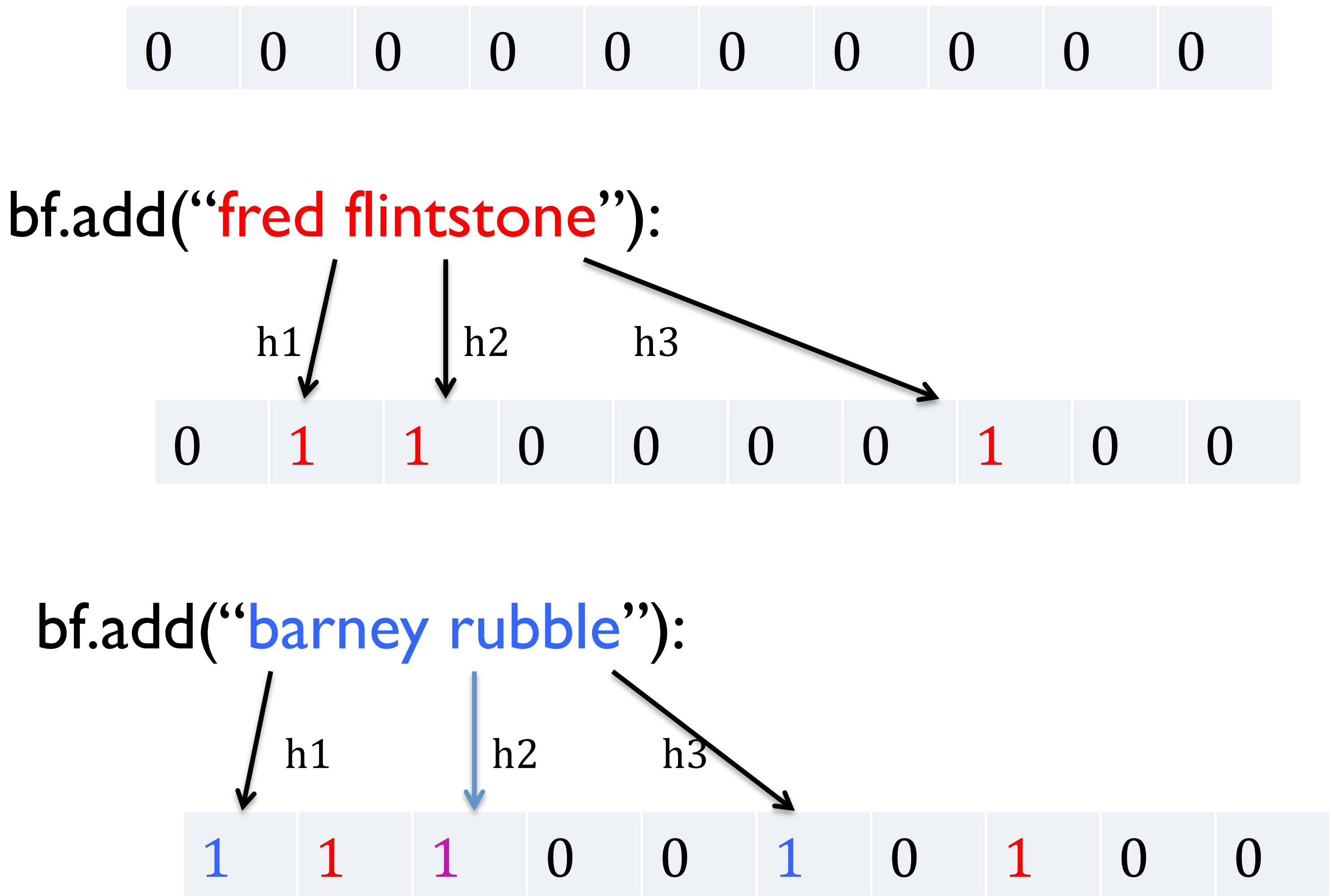
- Interface to a Bloom filter
  - BloomFilter(int maxSize, double p);
  - void bf.add(String s); // insert s
  - bool bd.contains(String s);
    - // If s was added return true;
    - // else with probability at least *1-p* return false;
    - // else with probability at most *p* return true;

  - I.e., a noisy "set" where you can test membership (and that's it)

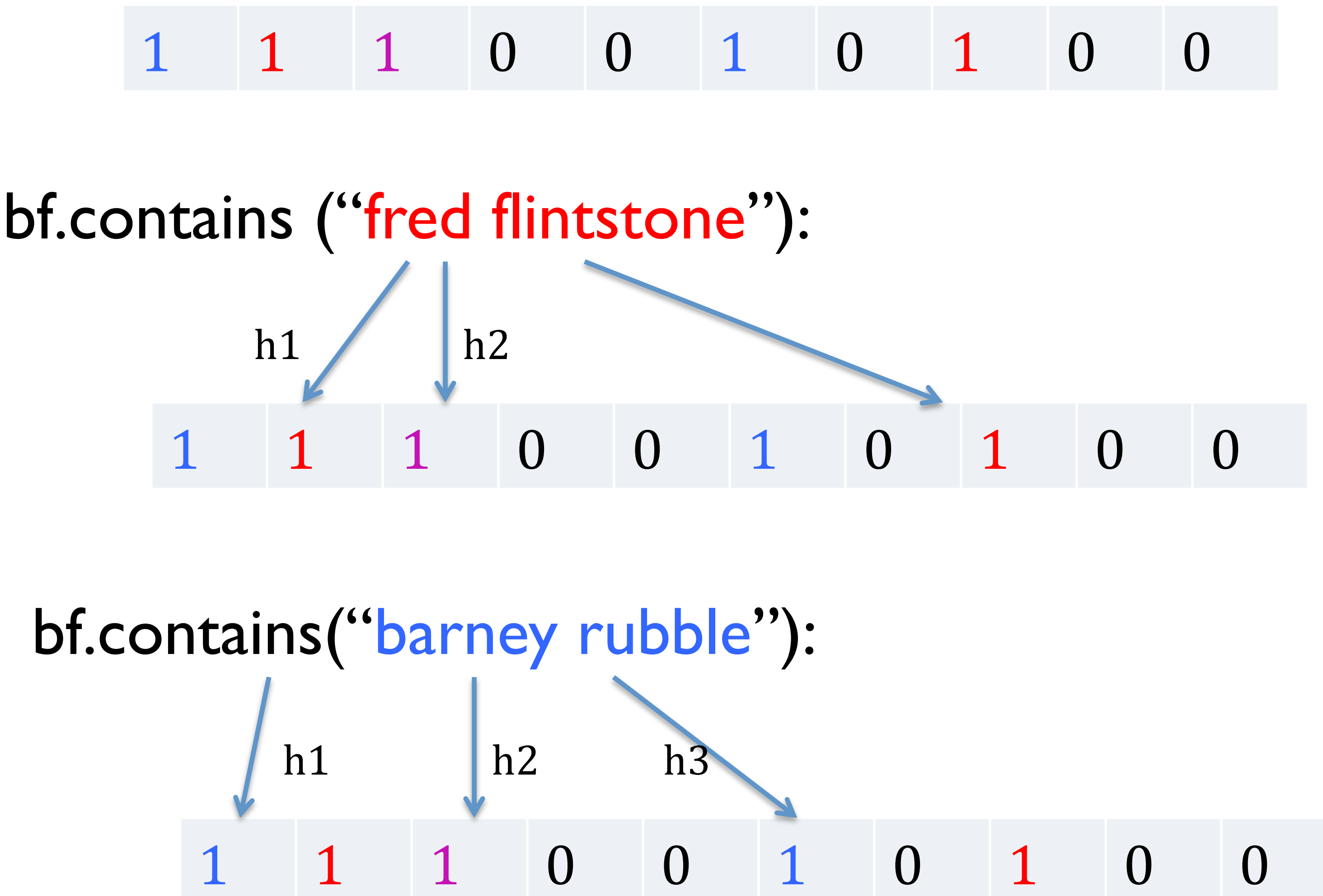
Bloom filters

- 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"
  - We'll discuss how to set M and K soon, but for now:
    - Let M = 1.5*maxSize *// less than two bits per item!*
    - Let K = 2*log(1/p) *// about right with this M*

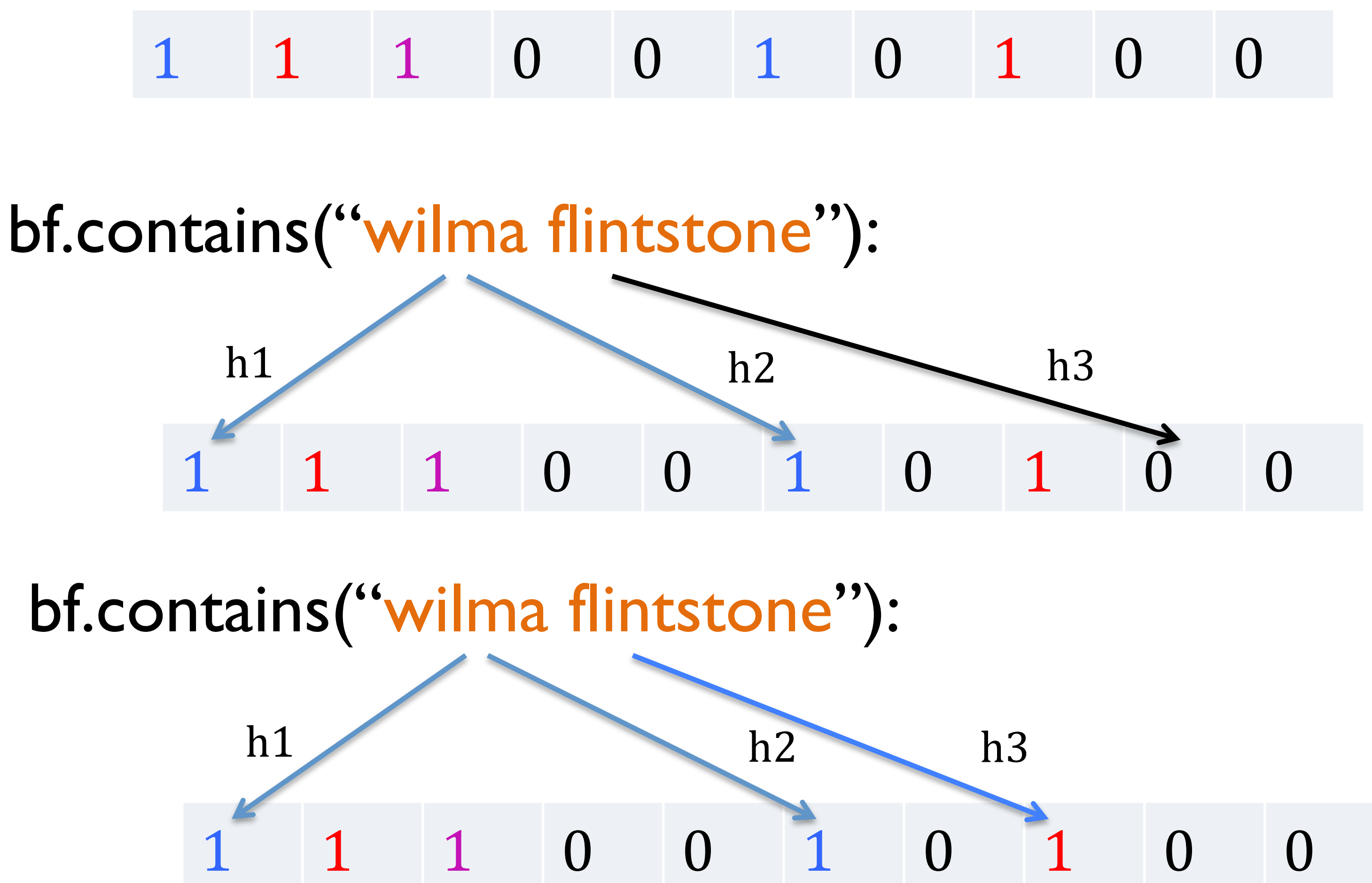
Bloom filters



Bloom filters



Bloom filters



Bloom filters: analysis

- Analysis (*m* bits, *k* hashers):
  - Assume hash(i,s) is a random function
  - Look at Pr(bit j is unset after *n* add's):

$$\left(1-\frac{1}{m}\right)^{kn}$$

  - … and Pr(collision) = Pr(not all *k* bits set)

$$f(m,n,k) = \left(1-\left[1-\frac{1}{m}\right]^{kn}\right)^{k} \approx \left(1-e^{-kn/m}\right)^{k}$$

  - …. fix *m* and *n* and minimize *k:*

$$k = \frac{m}{n}\ln 2 \approx 0.7\frac{m}{n}$$

Bloom filters

$$\left(1 - \left[1 - \frac{1}{m}\right]^{kn}\right)^k \approx \left(1 - e^{-kn/m}\right)^k$$

- Analysis:
  - Plug optimal k=m/n*ln(2) back into Pr(collision):

f(m,n) = $p = \left(1 - e^{-(m/n \ln 2)n/m}\right)^{(m/n \ln 2)}$

  - Now we can fix any two of *p, n, m* and solve for the 3rd:

E.g., the value for *m* in terms of *n* and *p:*

$$m = -\frac{n \ln p}{(\ln 2)^2}.$$

Bloom filters

- Interface to a Bloom filter
  - BloomFilter(int maxSize /* n */, 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: demo

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)

```
furter:google_ngram wcohen$ ls -alh *2gram* | tail
-rw-rw-rw-  1 13527  _lpoperator   264M Sep 17  2011 googlebooks-eng-all-2gram-20090715-90.csv.zip
-rw-rw-rw-  1 13527  _lpoperator   264M Sep 17  2011 googlebooks-eng-all-2gram-20090715-91.csv.zip
-rw-rw-rw-  1 13527  _lpoperator   264M Sep 17  2011 googlebooks-eng-all-2gram-20090715-92.csv.zip
-rw-rw-rw-  1 13527  _lpoperator   264M Sep 17  2011 googlebooks-eng-all-2gram-20090715-93.csv.zip
-rw-rw-rw-  1 13527  _lpoperator   264M Sep 17  2011 googlebooks-eng-all-2gram-20090715-94.csv.zip
-rw-rw-rw-  1 13527  _lpoperator   263M Sep 17  2011 googlebooks-eng-all-2gram-20090715-95.csv.zip
-rw-rw-rw-  1 13527  _lpoperator   264M Sep 17  2011 googlebooks-eng-all-2gram-20090715-96.csv.zip
-rw-rw-rw-  1 13527  _lpoperator   264M Sep 17  2011 googlebooks-eng-all-2gram-20090715-97.csv.zip
-rw-rw-rw-  1 13527  _lpoperator   264M Sep 17  2011 googlebooks-eng-all-2gram-20090715-98.csv.zip
-rw-rw-rw-  1 13527  _lpoperator   264M Sep 17  2011 googlebooks-eng-all-2gram-20090715-99.csv.zip
```

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/#shards

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

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

THE COUNT-MIN SKETCH

A variant of feature hashing

- Hash each feature *multiple times* with different hash functions
- Now, each *w* has *k* chances to *not* collide with another useful *w'*
- Get multiple hash functions as in Bloom filters

- *Part Bloom filter, part hash kernel*
  - *but predates either, called "count-min sketch" -- Cormode and Muthukrishnan*

Bloom filters

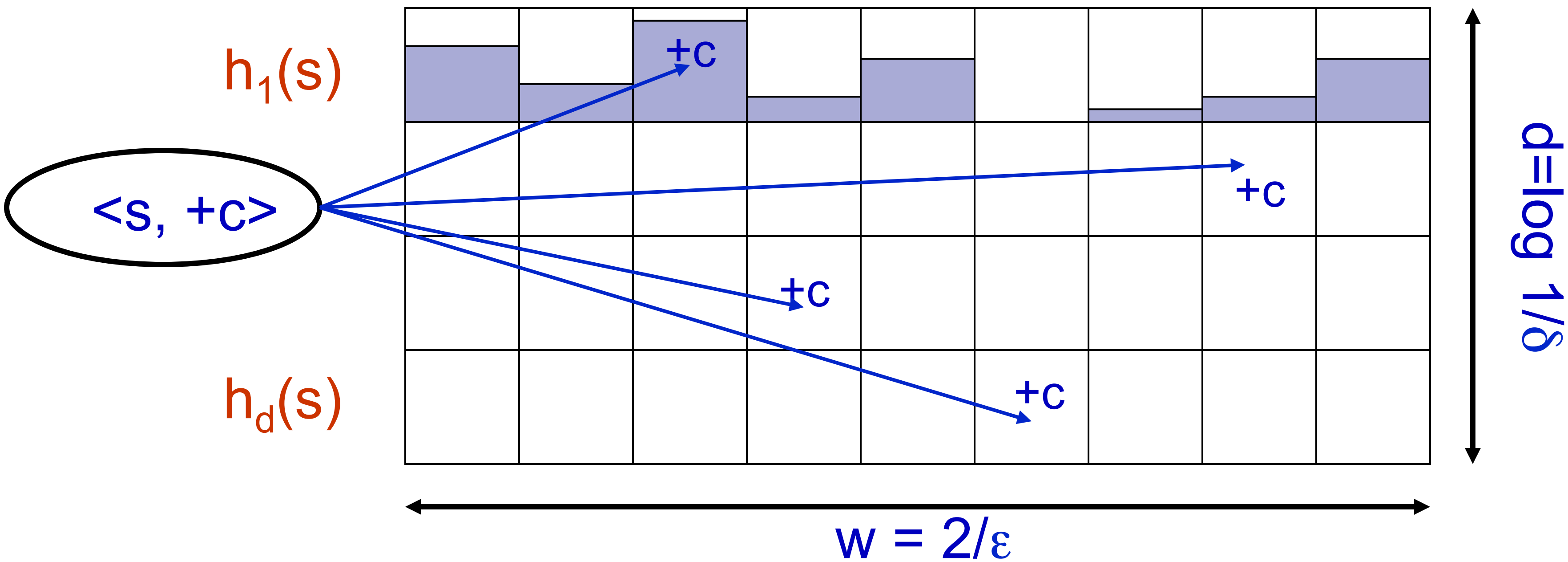
- An implementation
  - Allocate M bits, bit[0]…,bit[1-M]
  - Pick K hash functions hash(1,2),hash(2,s),….
    - E.g: hash(i,s) = hash(s+ randomString[i])
  - To add string s:
    - For i=1 to k, set bit[hash(i,s)] = 1
  - To check contains(s):
    - For i=1 to k, test bit[hash(i,s)]
    - Return "true" if they're all set; otherwise, return "false"
  - We'll discuss how to set M and K soon, but for now:
    - Let M = 1.5*maxSize *// less than two bits per item!*
    - Let K = 2*log(1/p) *// about right with this M*

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?

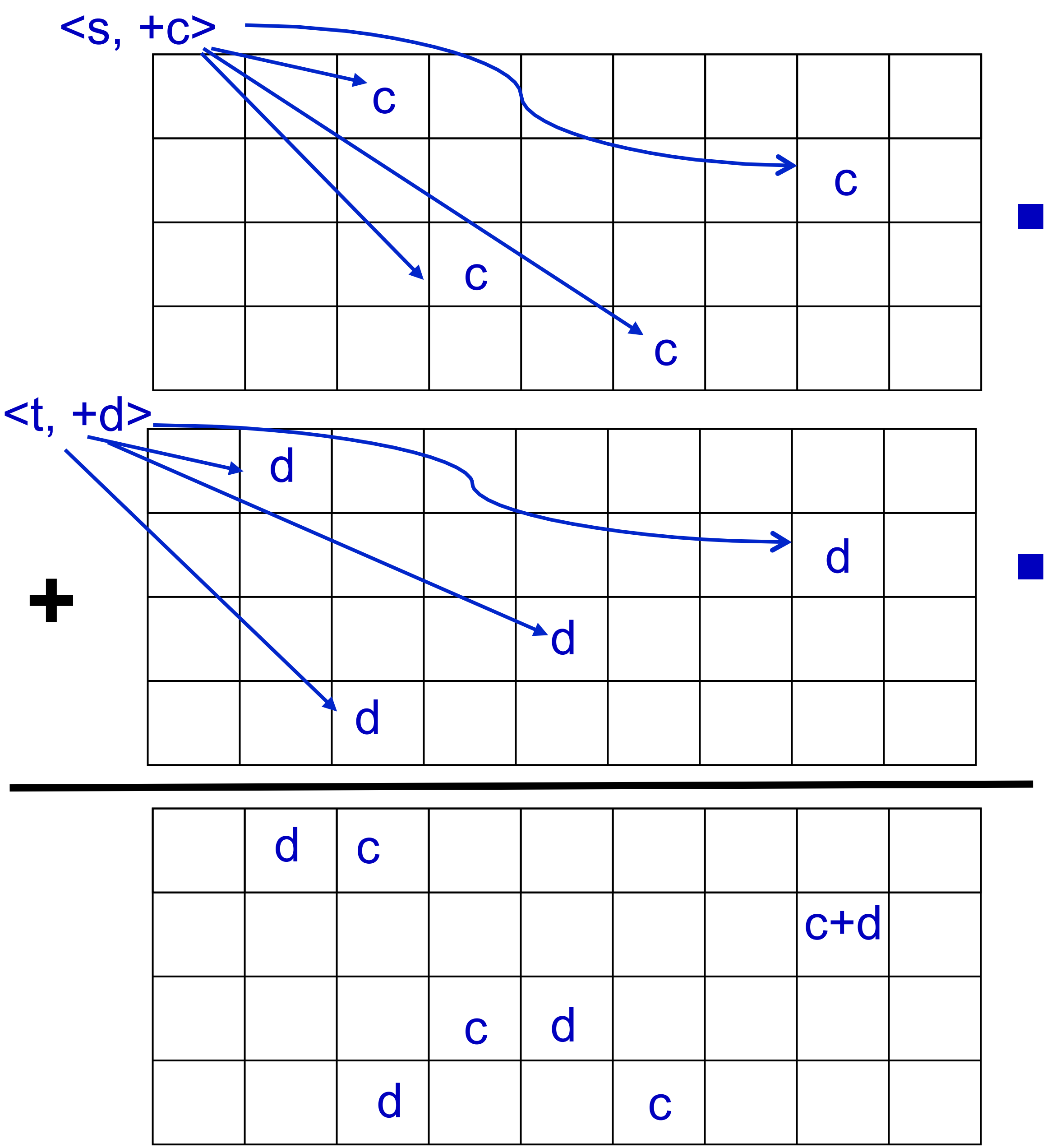
CM Sketch Structure



- 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

YAHOO! RESEARCH



- You can find the *sum* of two sketches by doing element-wise summation
- Also, you can compute a weighted sum of MC sketches
- Same result as adding <s,+c> and then <t,+d> to an empty sketch

CM Sketch Guarantees

- *[Cormode, Muthukrishnan'04]* CM sketch guarantees approximation error on point queries less than $\varepsilon||A||_1$ in space $O(1/\varepsilon \log 1/\delta)$
  - Probability of more error is less than $1-\delta$

- This is sometimes enough:
  - Estimating a multinomial: if A[s] = Pr(s|…) then $||A||_1 = 1$
  - Multiclass classification: if $A_x[s]$ = Pr(x in class s) then $||A_x||_1$ is probably small, since most x's will be in only a few classes

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]*

Lemma 1 (Cormode and Muthukrishnan [6], Eqn 5.1) *Let* $\mathbf{y}$ *be an vector, and let* $\tilde{y}_i$ *be the estimate given by a count-min sketch of width* w *and depth* d *for* y_i*. Let the* k *largest components of* $\mathbf{y}$ *be* $y_{\sigma_1}, \ldots, y_{\sigma_k}$*, and let* $t_k = \sum_{k'>k} y_{\sigma_1}$ *be the weight of the "tail" of* $\mathbf{y}$*. If* $w \geq \frac{1}{3k}$*,* $w > \frac{e}{\eta}$ *and* $d \geq \ln\frac{3}{2}\ln\frac{1}{\delta}$*, then* $\tilde{y}_i \leq y_i + \eta t_k$ *with probability at least 1-*δ*.*

Theorem 3 (Cormode and Muthukrishnan [6], Theorem 5.1) *Let* $\mathbf{y}$ *represent a Zipf-like distribution with parameter* z*. Then with probability at least 1-*δ*,* $\mathbf{y}$ *can be approximated to within error* η *by a count-min sketch of width* $O(\eta^{-\min(1,1/z)})$ *and depth* $O(\ln\frac{1}{\delta})$*.*

YAHOO! RESEARCH

An Application of a Count-Min Sketch

- Problem: find the semantic orientation of a work (positive or negative) using a large corpus.
- Idea:
  - positive words co-occur more frequently than expected near positive words; likewise for negative words
  - so pick a few pos/neg seeds and compute

x appears near y

$$\mathrm{pmi}(x;y) \equiv \log \frac{p(x,y)}{p(x)p(y)}$$

$$\mathrm{SO(w)} = \sum_{\mathrm{p} \in \mathrm{Pos}} PMI(\mathrm{p}, \mathrm{w}) - \sum_{\mathrm{n} \in \mathrm{Neg}} PMI(\mathrm{n}, \mathrm{w})$$

An Application of a Count-Min Sketch

x appears near y → $p(x,y)$

$$\text{pmi}(x;y) \equiv \log \frac{p(x,y)}{p(x)p(y)}$$

$$\text{SO(w)} = \sum_{\text{p} \in \text{Pos}} PMI(\text{p}, \text{w}) - \sum_{\text{n} \in \text{Neg}} PMI(\text{n}, \text{w})$$

Example: Turney, 2002 used two seeds, "excellent" and "poor"

$$SO(phrase) = log_2(\frac{hits(phrase\ NEAR\ 'excellent')hits('excellent')}{hits(phrase\ NEAR\ 'poor')hits('excellent')})$$

In general, SO(w) can be written in terms of logs of products of counters for w, with and without seeds

An Application of a Count-Min Sketch

- Use 2B counters, 5 hash functions, “near” means a 7-word window, GigaWord (10 Gb) and GigaWord + Web news 50 Gb)

Data	Exact	CM-CU	CMM-CU	LCU-WS
GW	*74.2*	*74.0*	65.3	*72.9*
GWB50	**81.2**	**80.9**	74.9	78.3

Table 2: Evaluating Semantic Orientation on accuracy metric using several sketches of 2 billion counters against exact. Bold and italic numbers denote no statistically significant difference.

An Application of a Count-Min Sketch

CM-CU: CM with "conservative update" - for <*j*, +*c*> increment counters just enough to make the new estimate for *j* grow by *c*

Data	Exact	CM-CU	CMM-CU	LCU-WS
GW	*74.2*	*74.0*	65.3	*72.9*
GWB50	**81.2**	**80.9**	74.9	78.3

Table 2: Evaluating Semantic Orientation on accuracy metric using several sketches of 2 billion counters against exact. Bold and italic numbers denote no statistically significant difference.

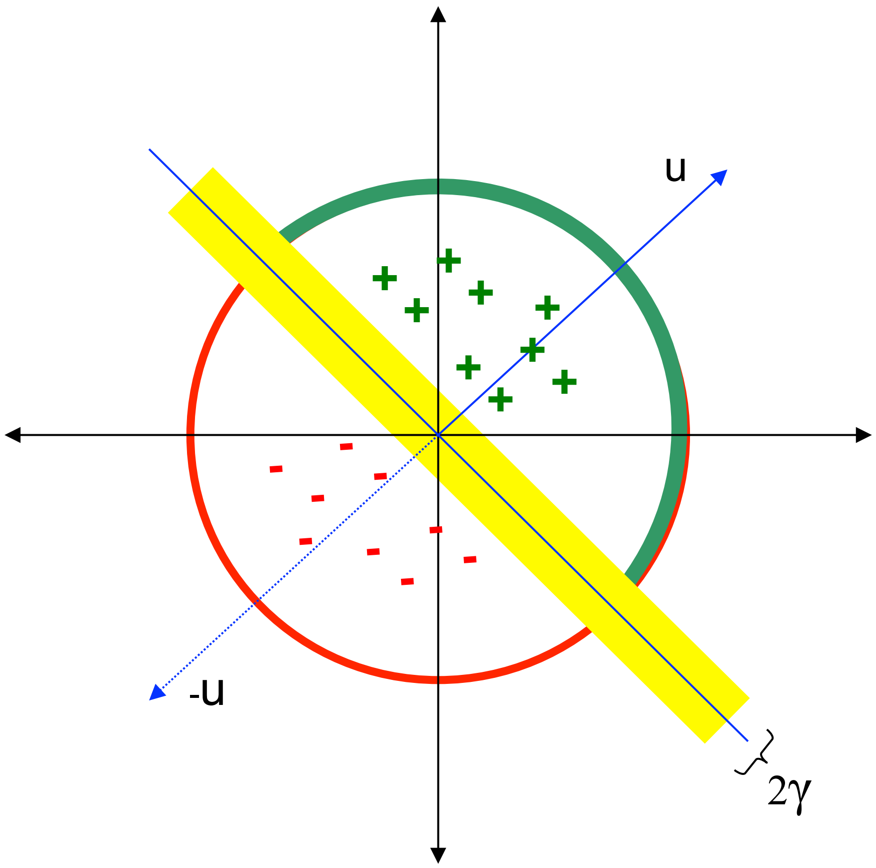
LOCALITY SENSITIVE HASHING (LSH)

LSH: key ideas

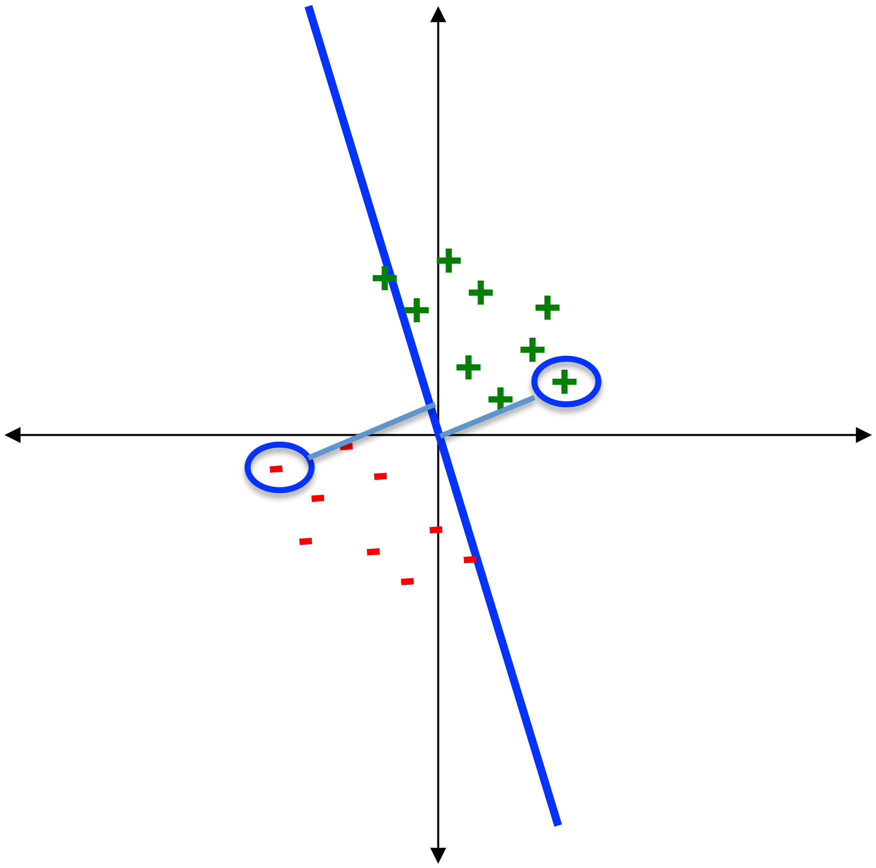
- Goal:
  - map feature vector **x** to bit vector **bx**
  - ensure that **bx** preserves "similarity"

Random Projections

Random projections

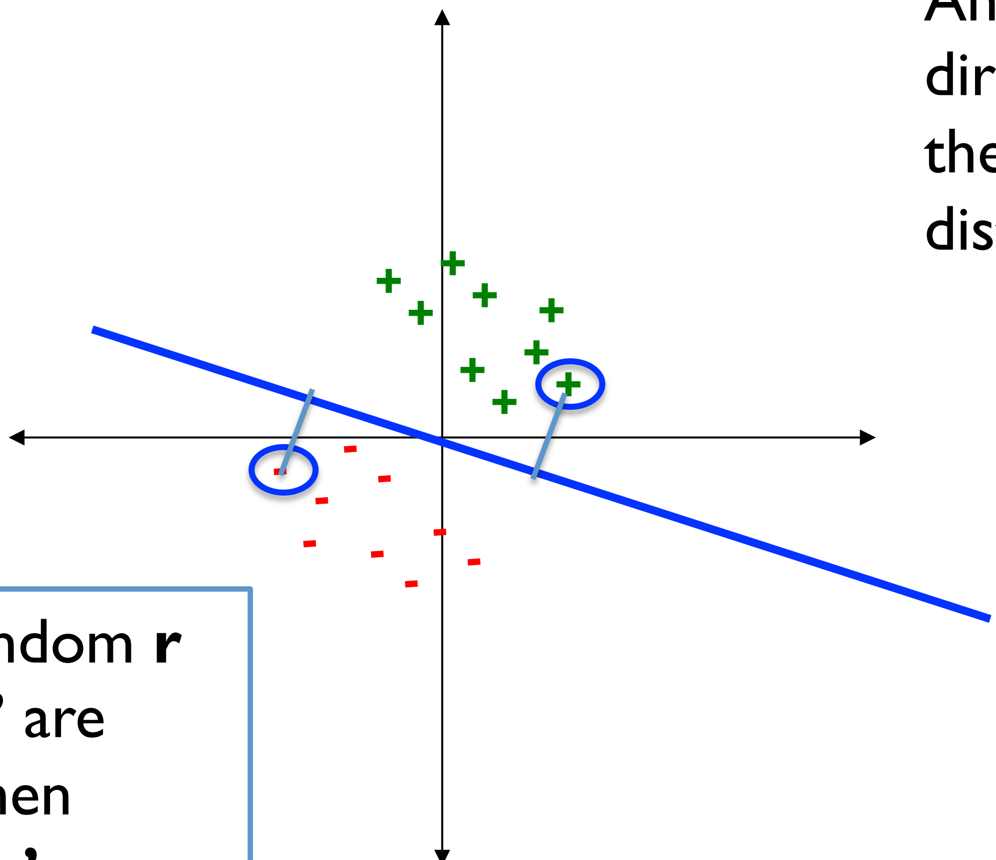


Random projections



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

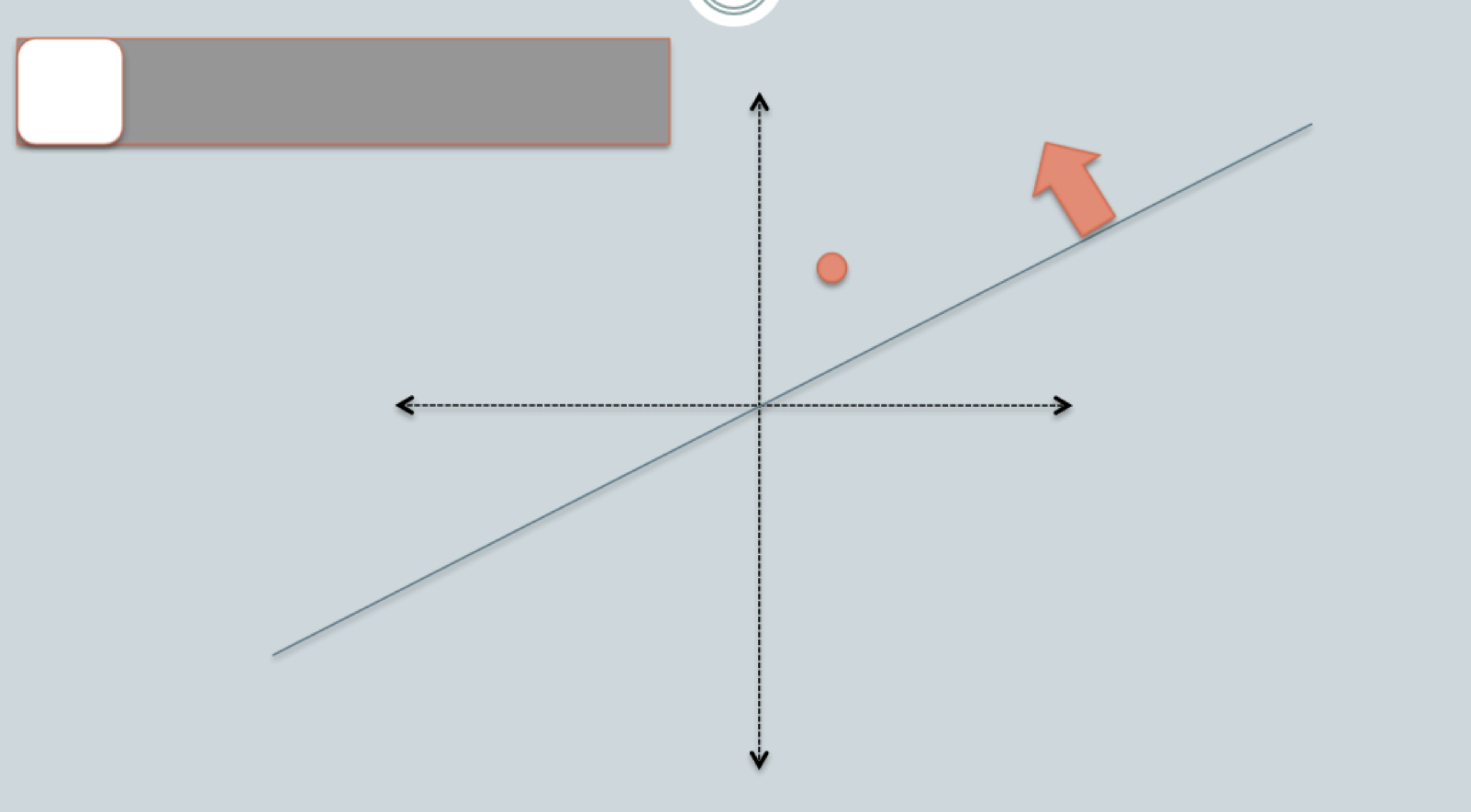
- Goal:
  - map feature vector **x** to bit vector **bx**
  - ensure that **bx** preserves "similarity"
- Basic idea: use *random projections* of **x**
  - Repeat many times:
    - Pick a random hyperplane **r** by picking random weights for each feature (say from a Gaussian)
    - Compute the inner 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

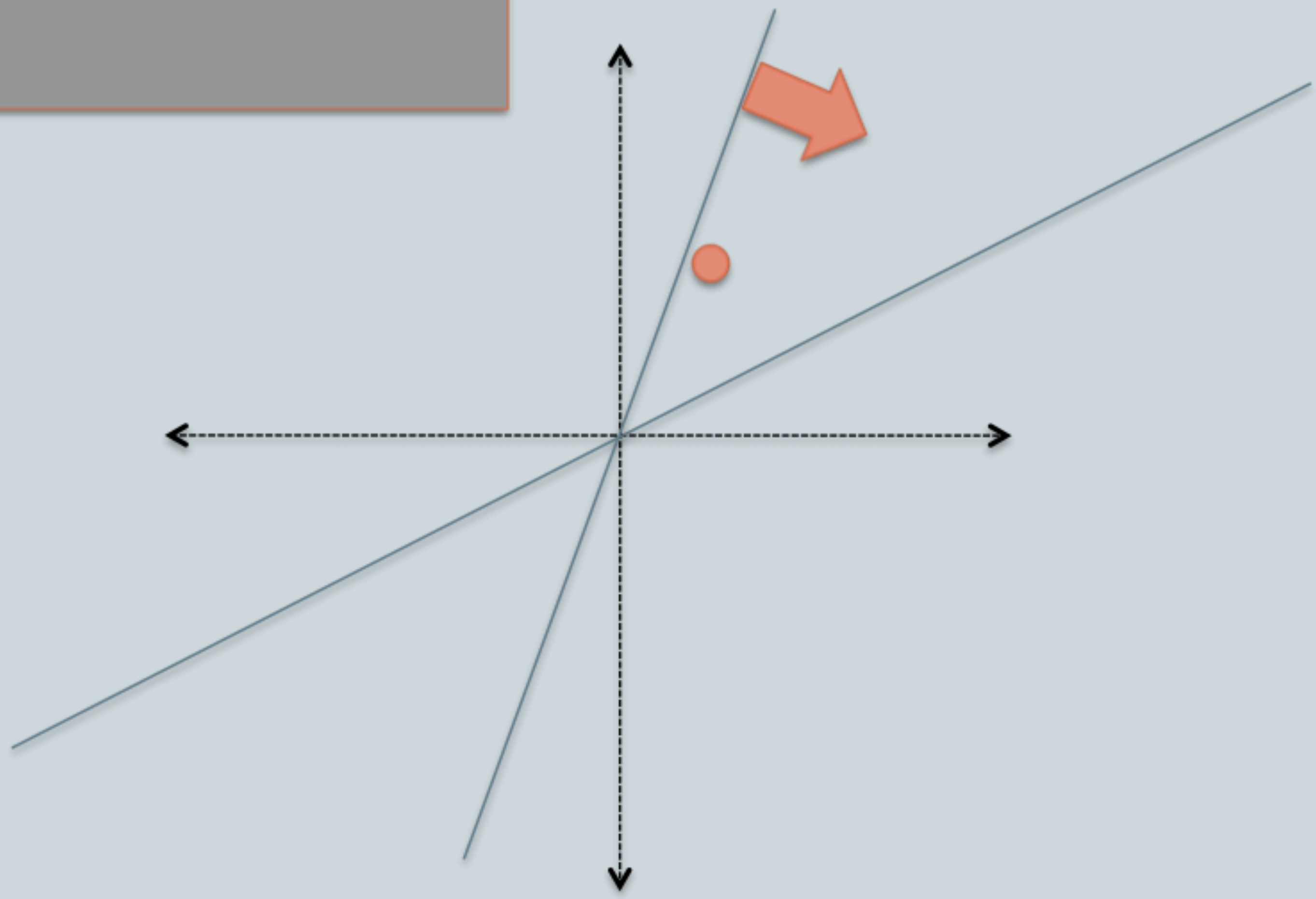
Online Generation of Locality Sensitive Hash Signatures

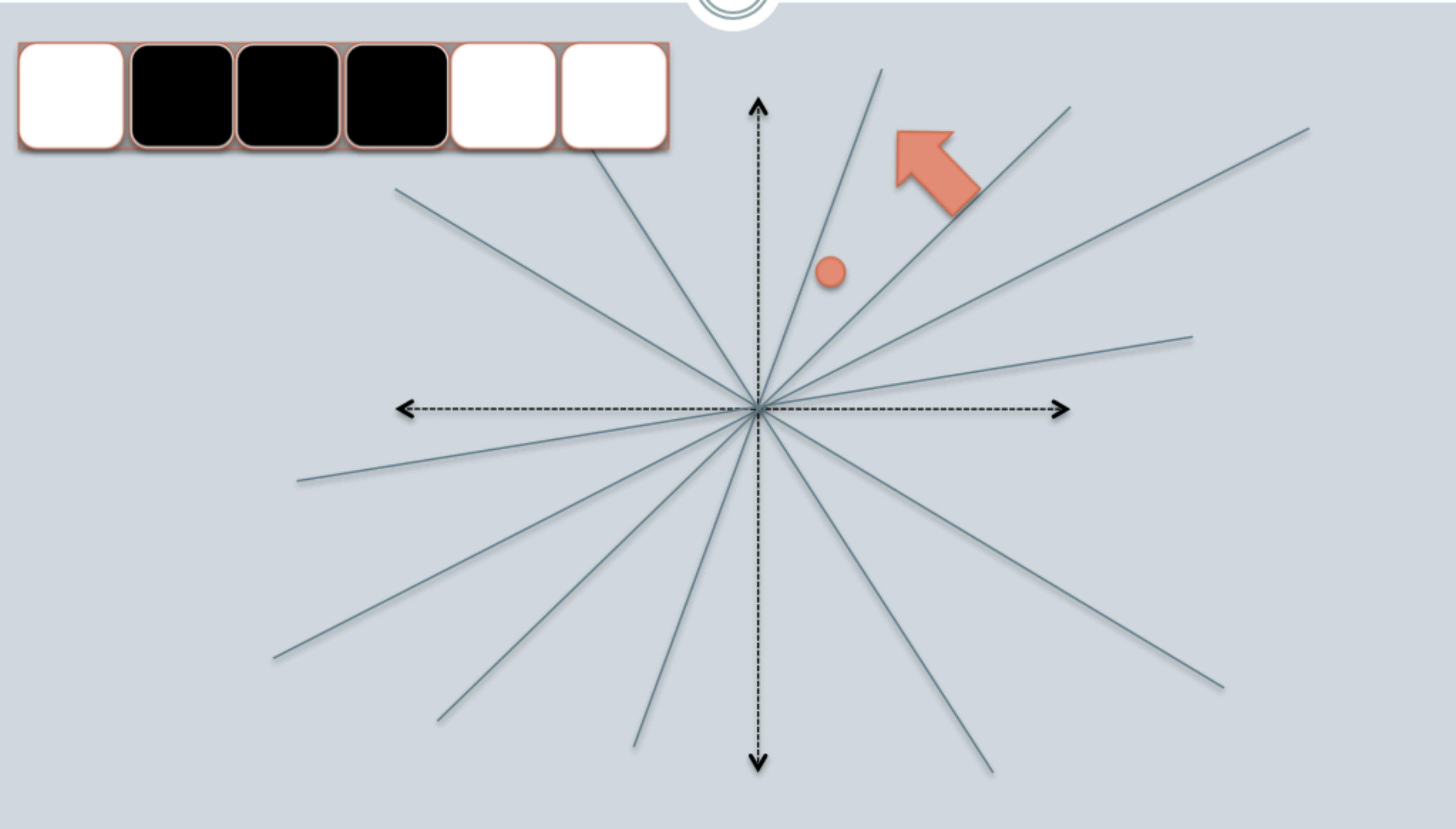
Benjamin Van Durme and Ashwin Lall

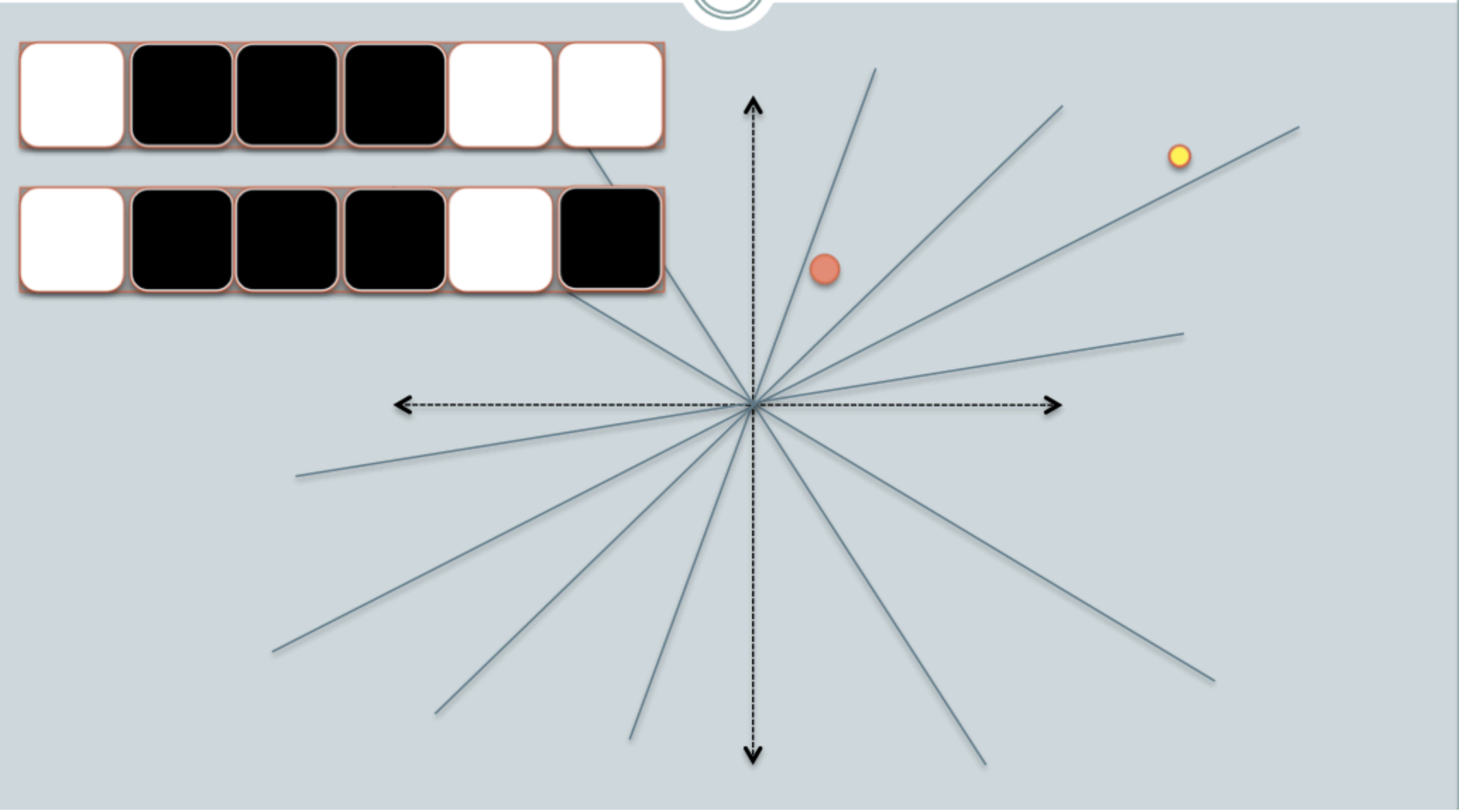
human language technology
center of excellence
JOHNS HOPKINS UNIVERSITY

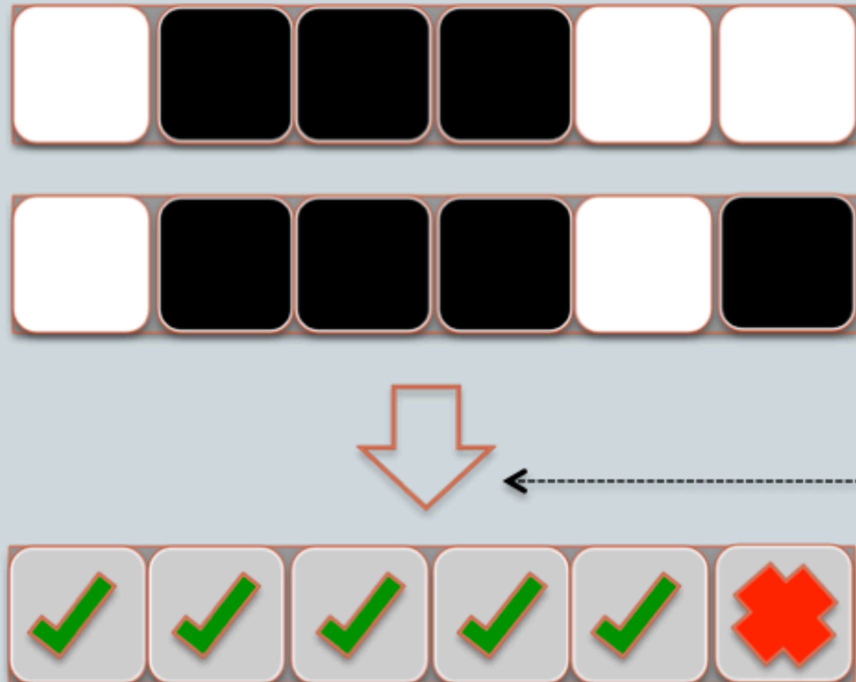
DENISON UNIVERSITY

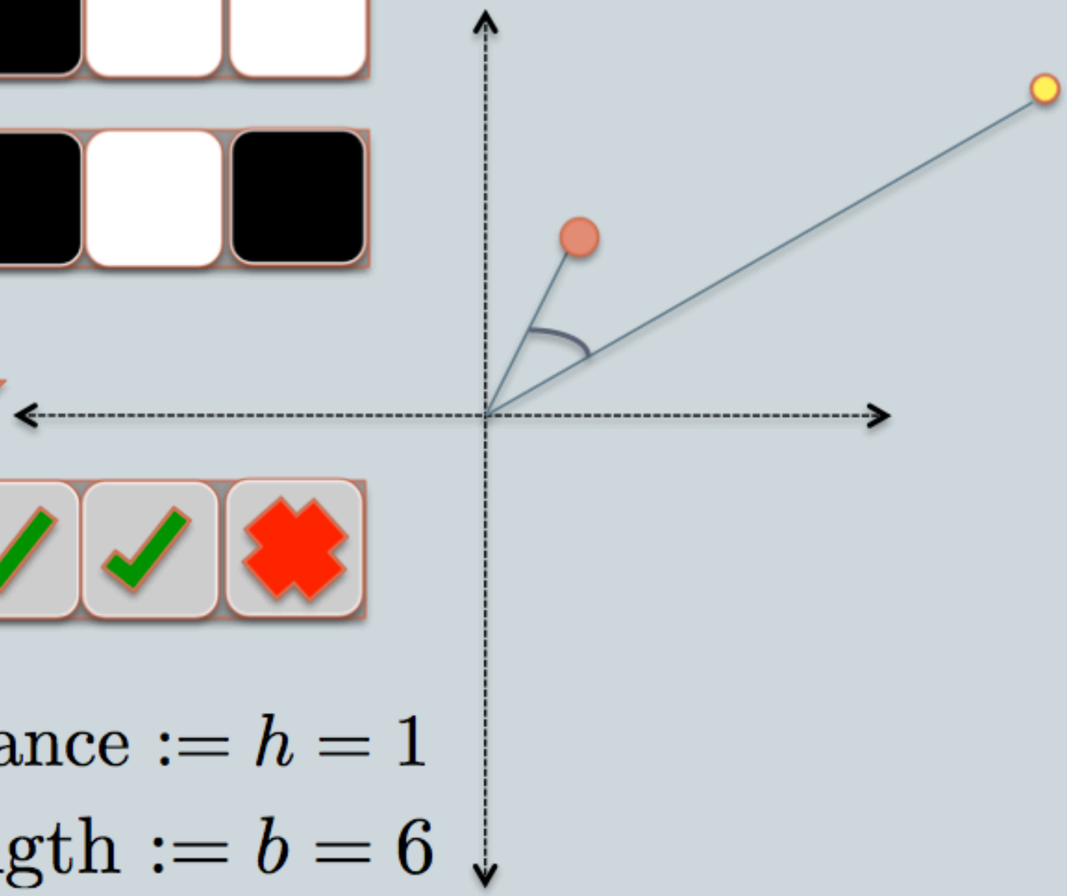








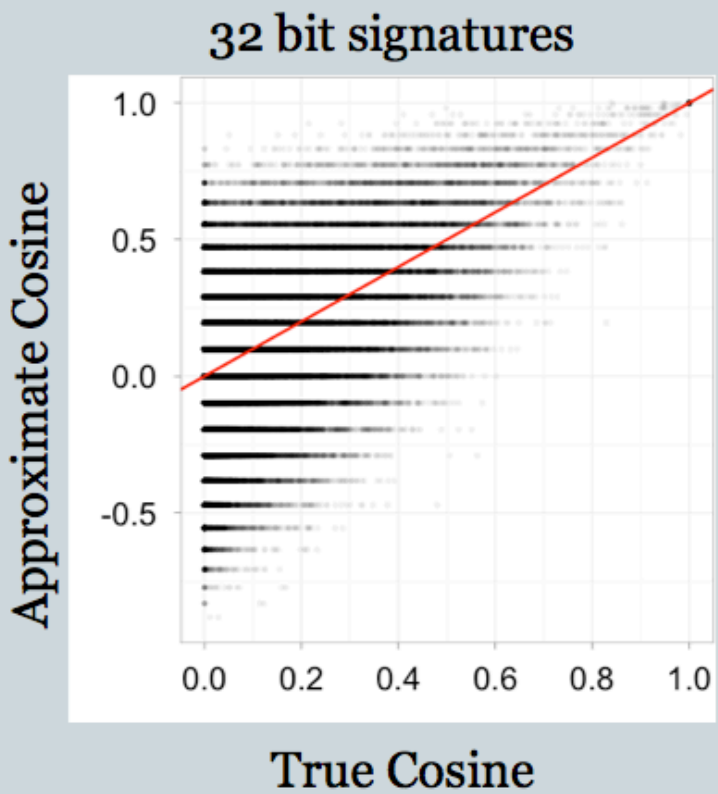


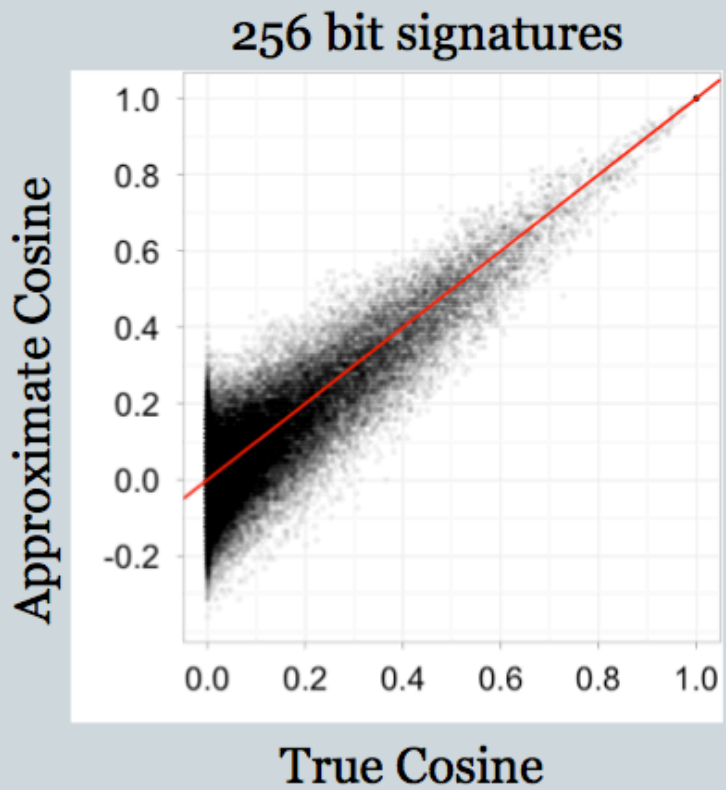


Hamming Distance $:= h = 1$

Signature Length $:= b = 6$

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





Cheap

Accurate

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

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

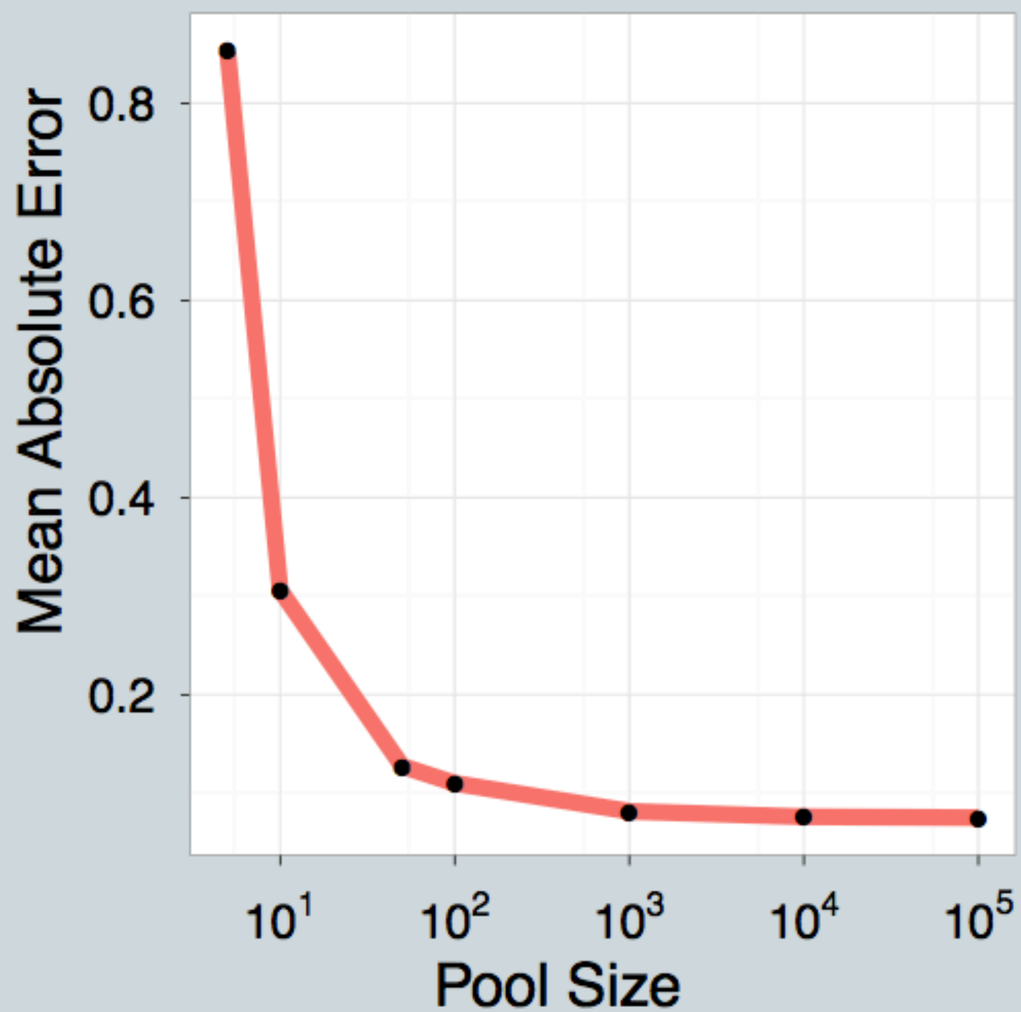
LSH: "pooling" (van Durme)

- 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:

Initialize floating point array $P[0, \ldots, m-1]$
Initialize H, a hashtable mapping words to floating point arrays of size d
for $i := 0 \ldots m-1$ **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 $j := 1 \ldots d$ **do**
            $H[w][j] := H[w][j] + P[h_j(s, f_i)]$

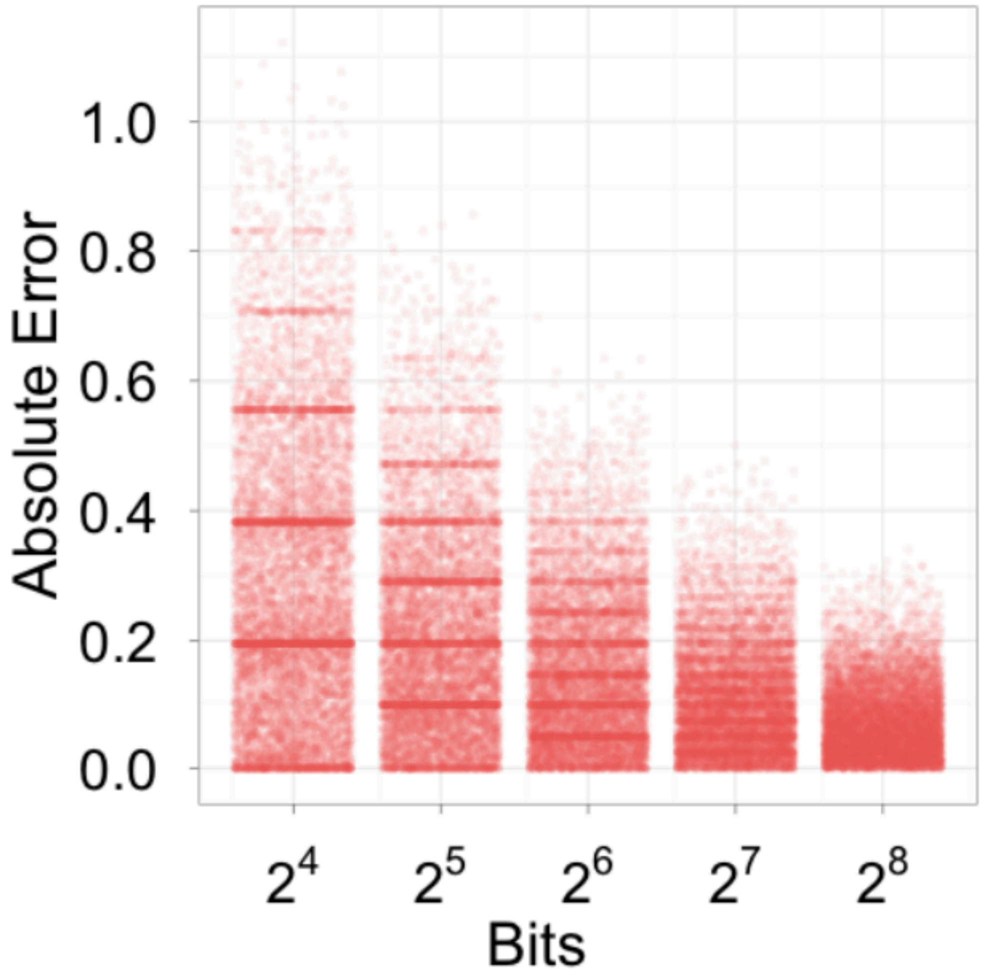
SIGNATURECOMPUTATION:

for each $w \in H$ **do**
    for $i := 1 \ldots d$ **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