Randomized Algorithms Part 3

William Cohen

Outline

- Randomized methods - so far
  - SGD with the hash trick
  - Bloom filters
  - count-min sketches
- Today:
  - Review and discussion
  - More on count-min
  - Morris counters
  - **locality sensitive hashing**

Locality Sensitive Hashing (LSH)

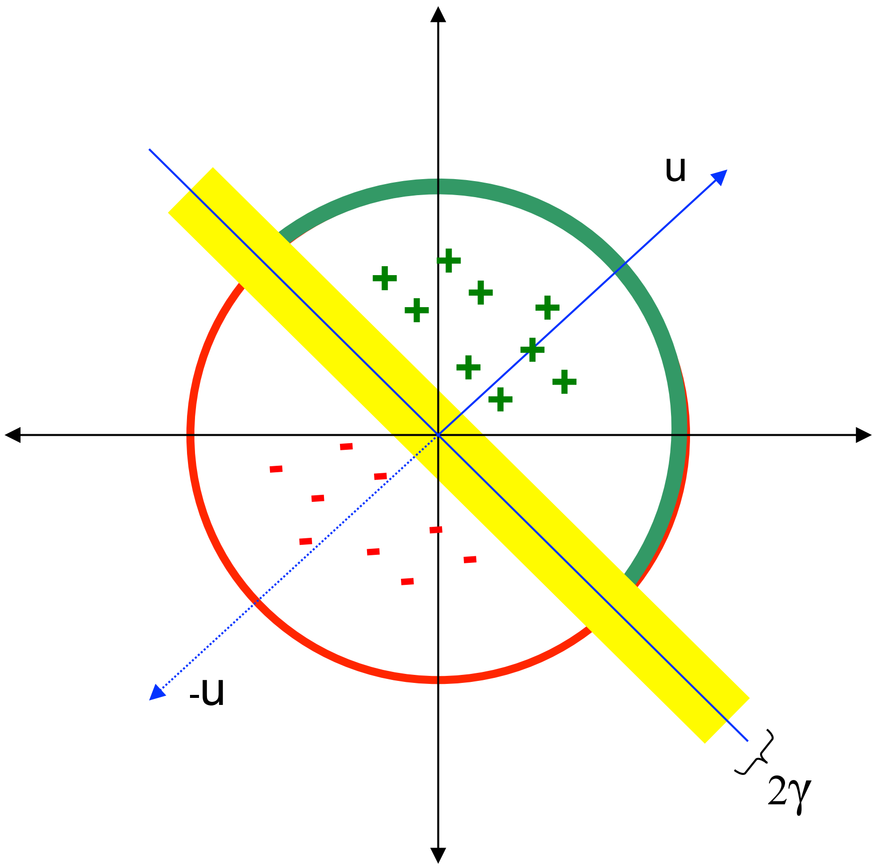
- *Bloom filters:*
  - **set of objects** mapped to a **bit vector**
  - **allows**: add to set, check containment
- *Countmin sketch:*
  - **sparse vector, x** mapped to **small dense matrix**
  - **allows**: recover approximate value of $\boldsymbol{x_i}$ especially useful for largest values
- **Locality sensitive hash:**
  - **feature vector, x** mapped to bit vector, **bx**
  - **allows**: compute approximate similarity of **bx** and **by**

LSH: key ideas

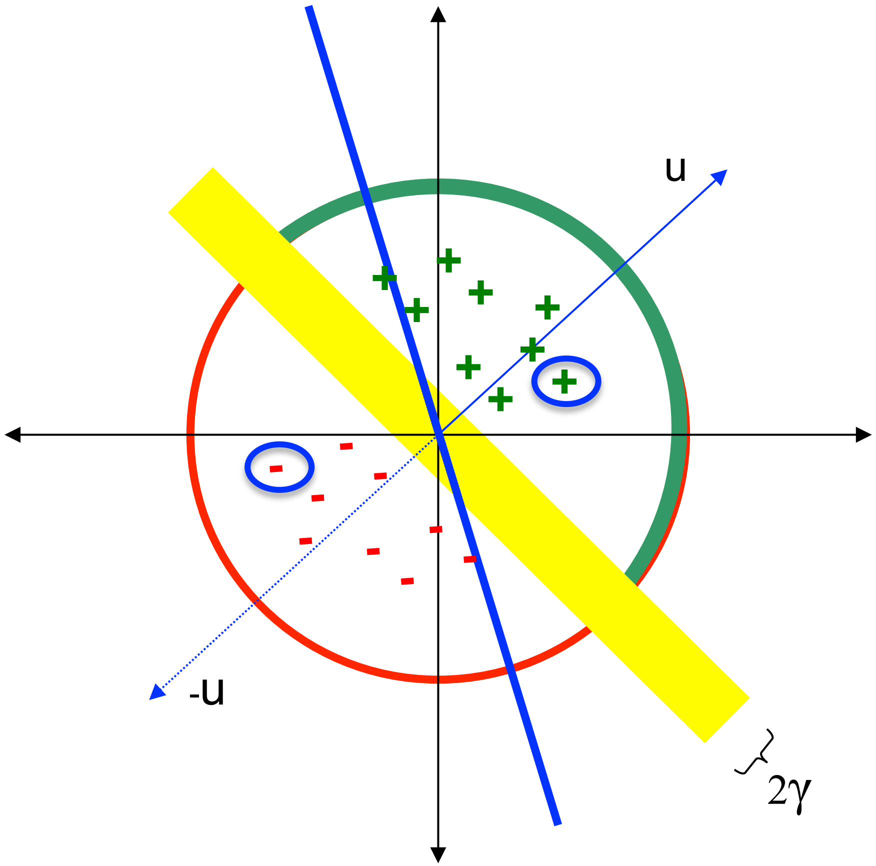
- Goal:
  - map feature vector **x** to bit vector **bx**
  - ensure that **bx** preserves “similarity”

Random Projections

Random projections

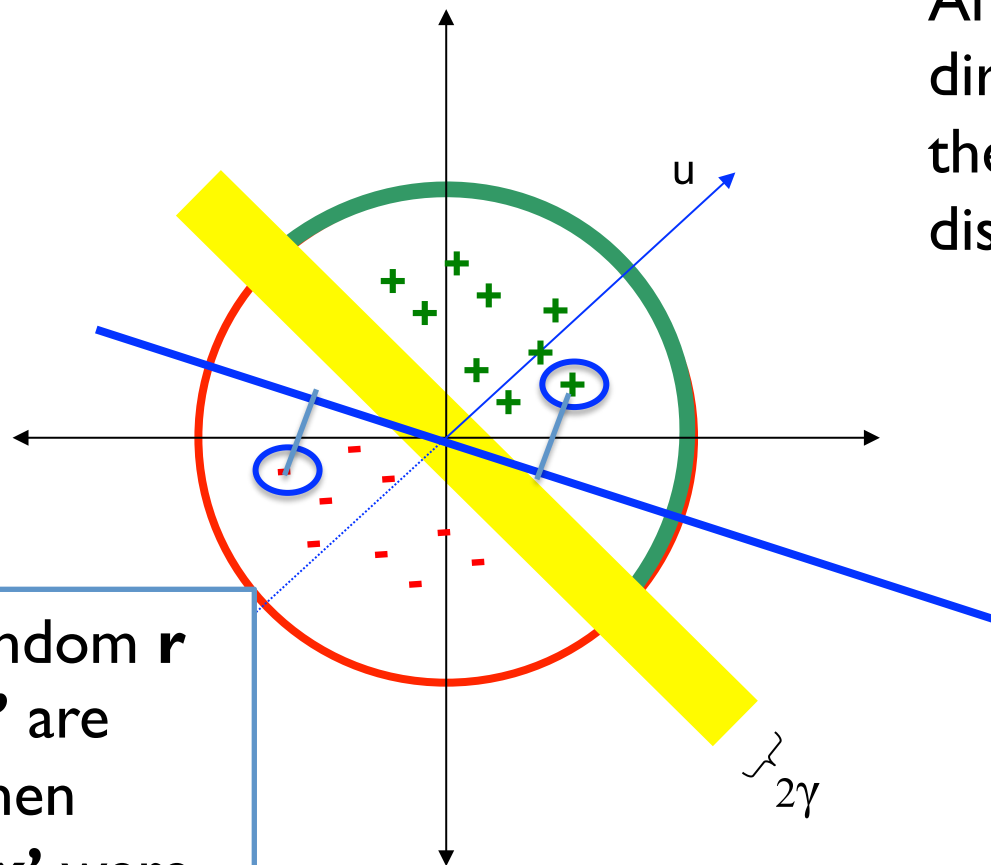


Random projections



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

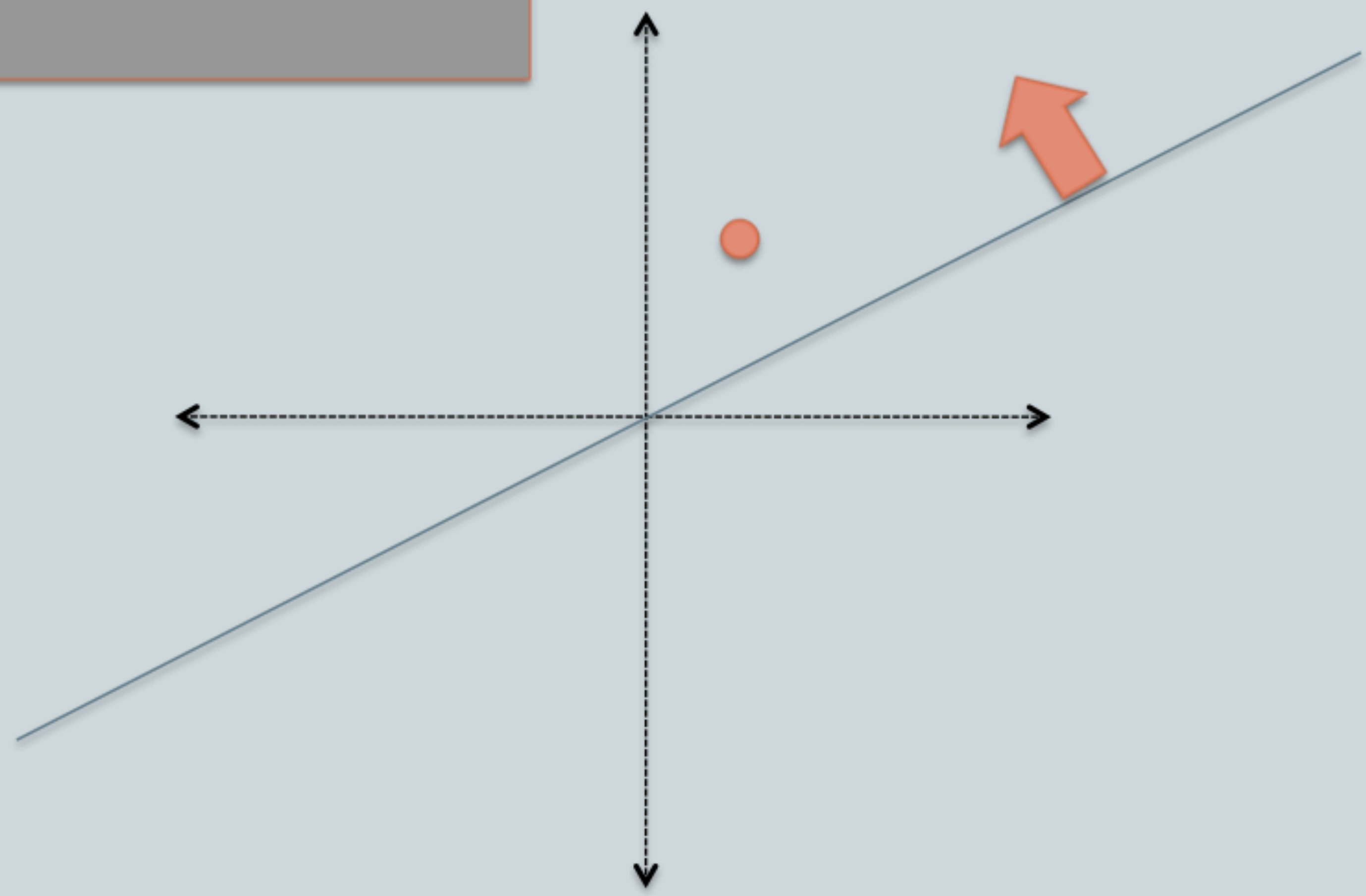
- Goal:
  - map feature vector **x** to bit vector **bx**
  - ensure that **bx** preserves "similarity"
- Basic idea: use random projections of **x**
  - Repeat many times:
    - Pick a random hyperplane **r** by picking random weights for each feature (say from a Gaussian)
    - Compute the inner product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

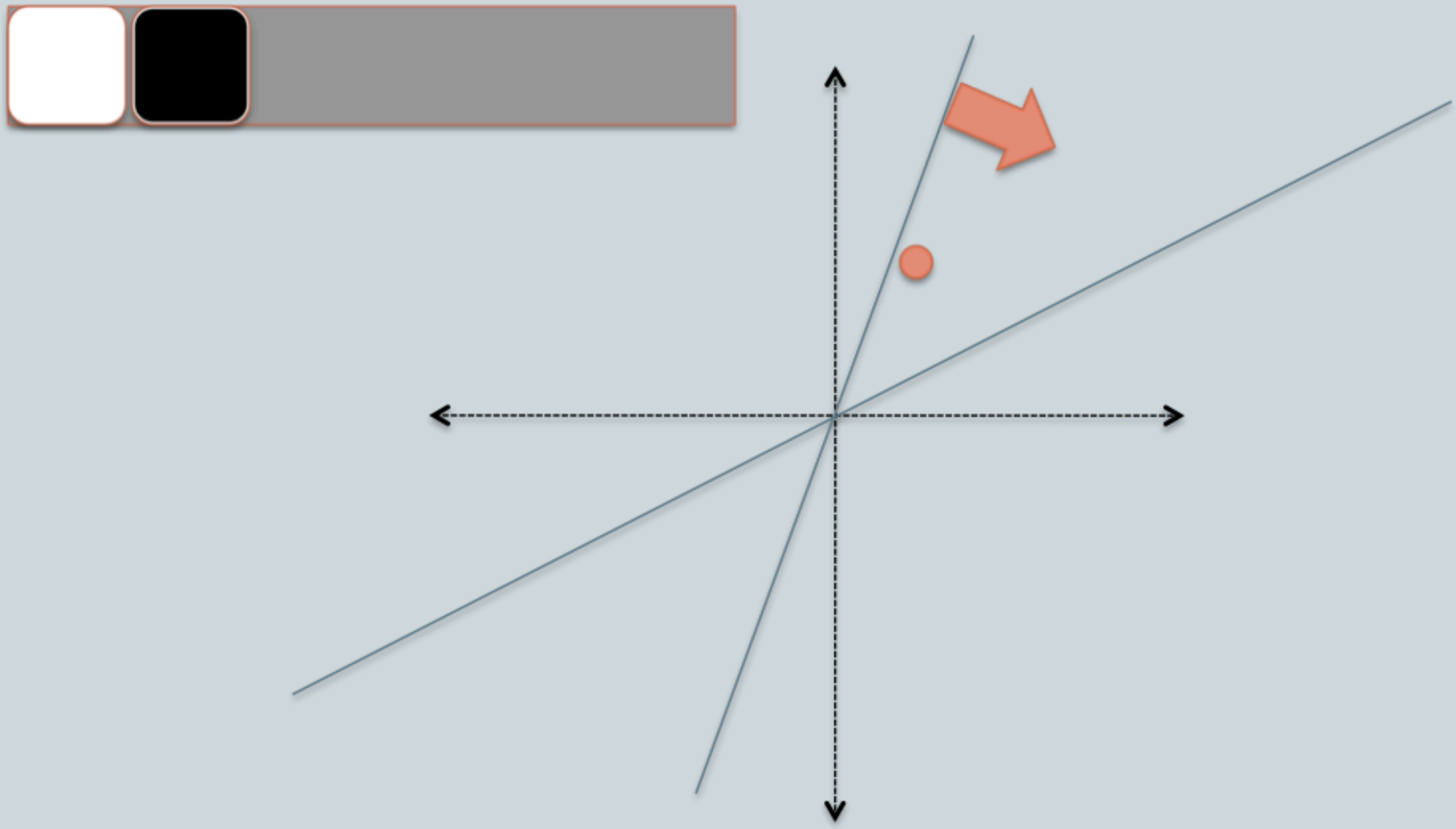
Online Generation of Locality Sensitive Hash Signatures

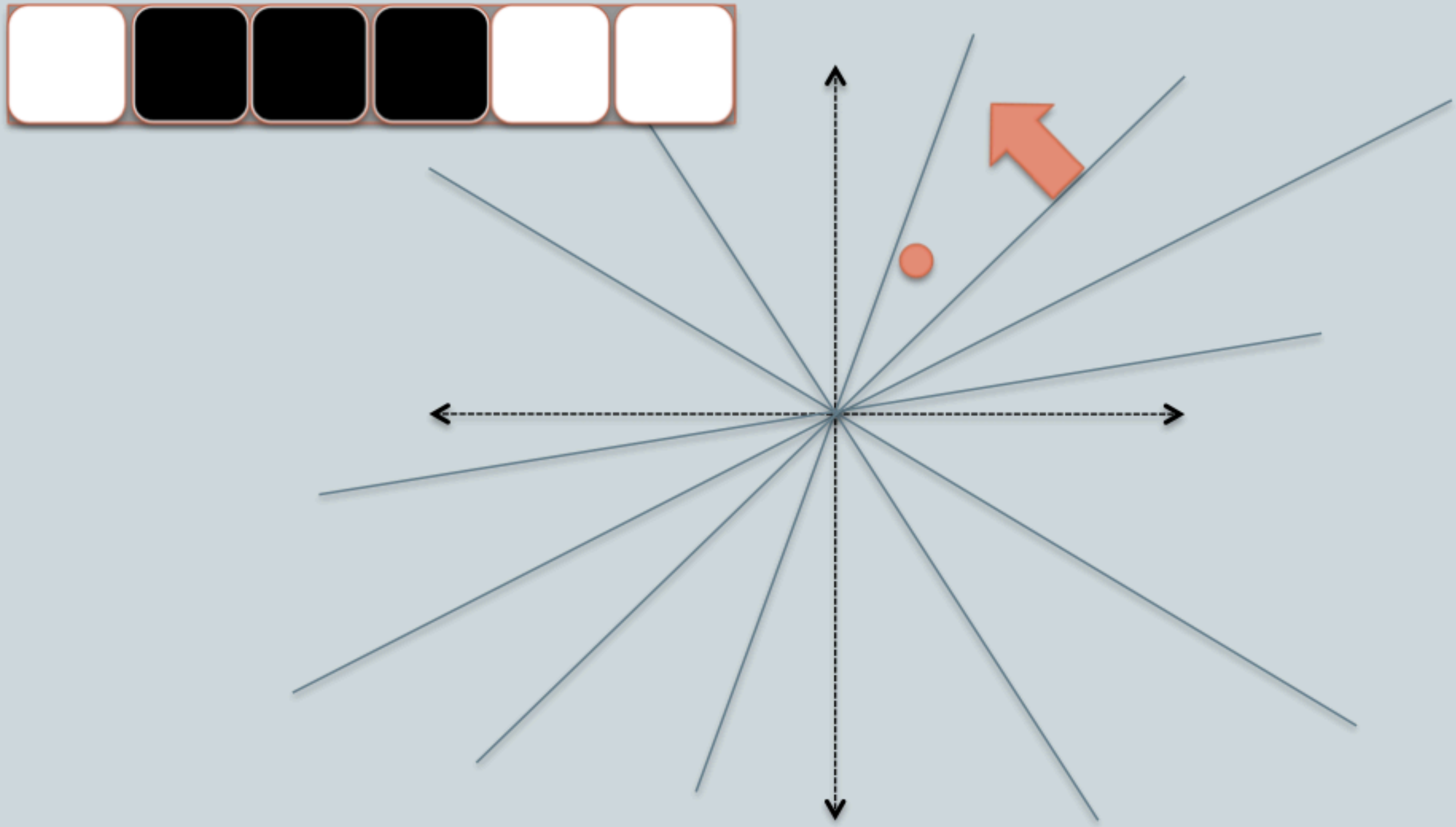
Benjamin Van Durme and Ashwin Lall

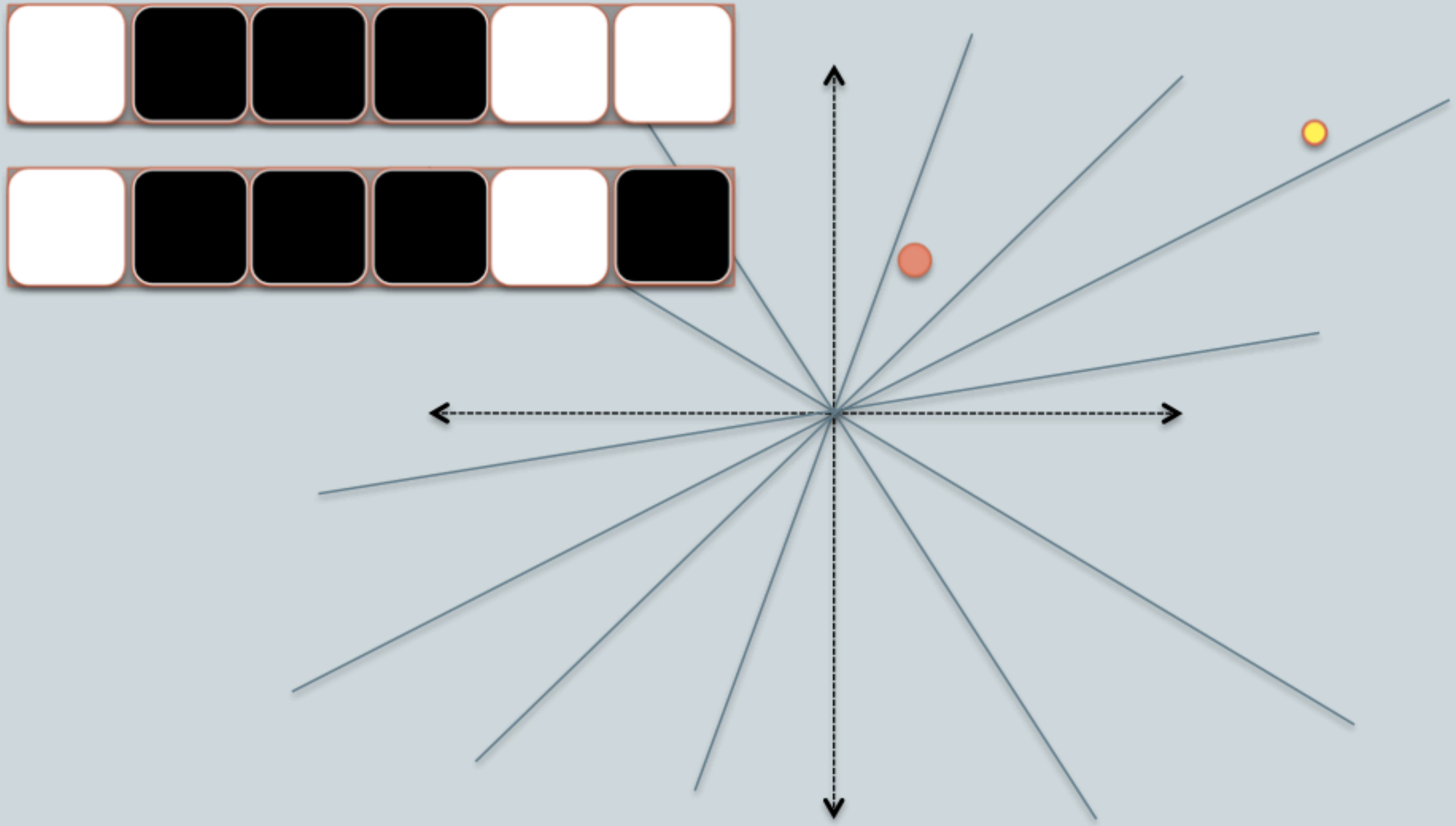
human language technology
center of excellence
JOHNS HOPKINS UNIVERSITY

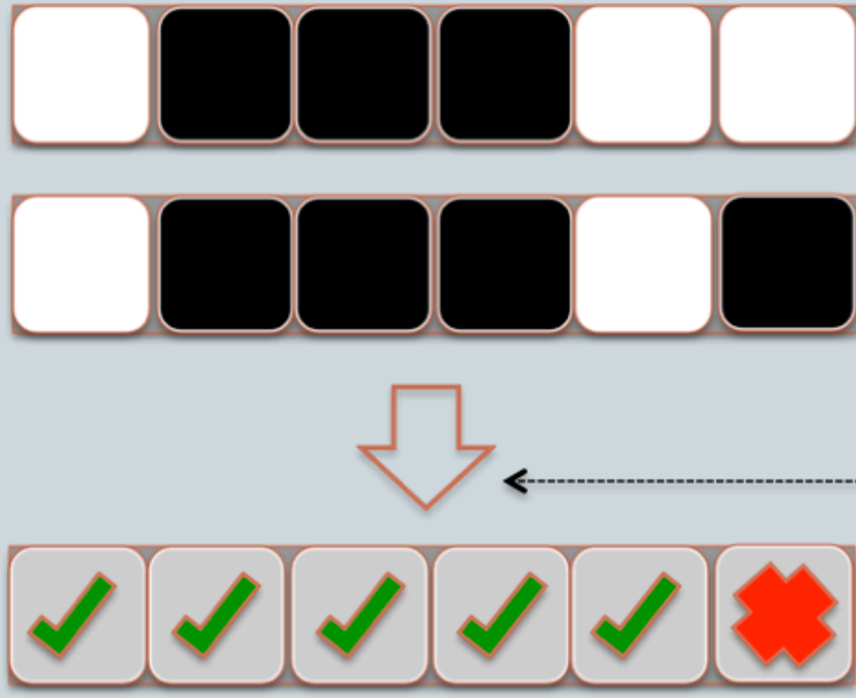
DENISON UNIVERSITY

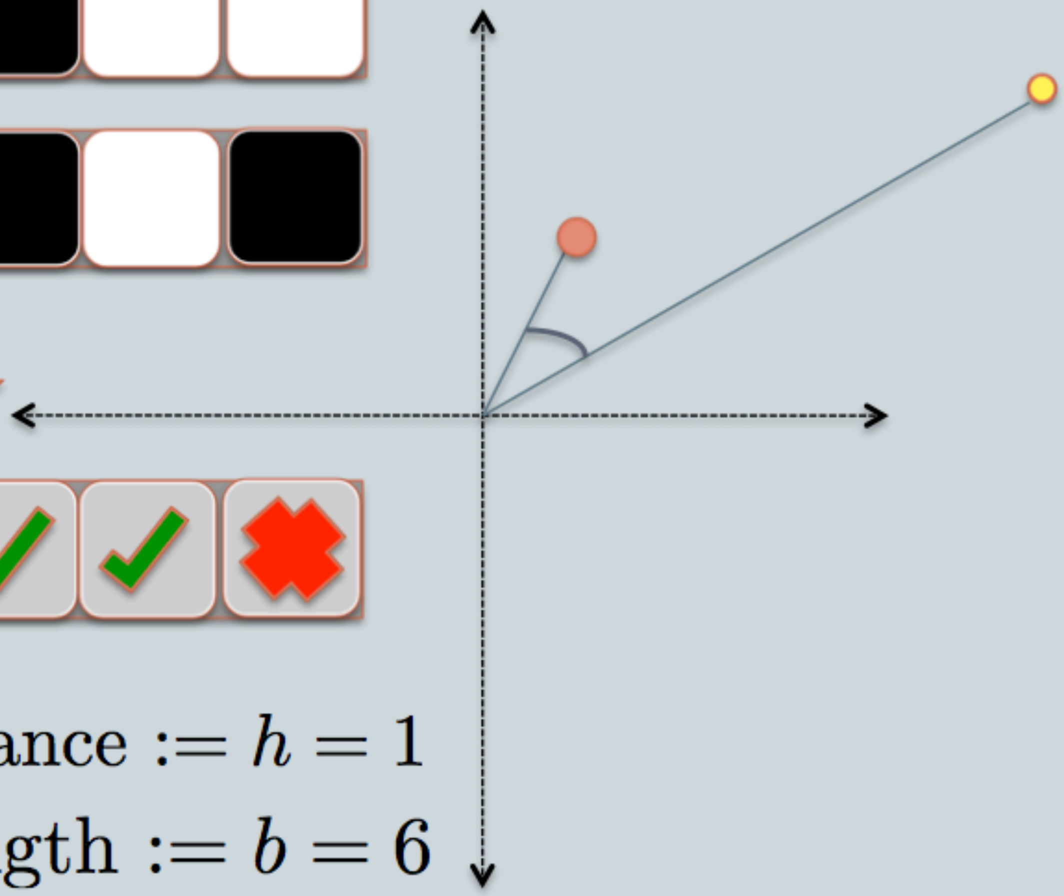








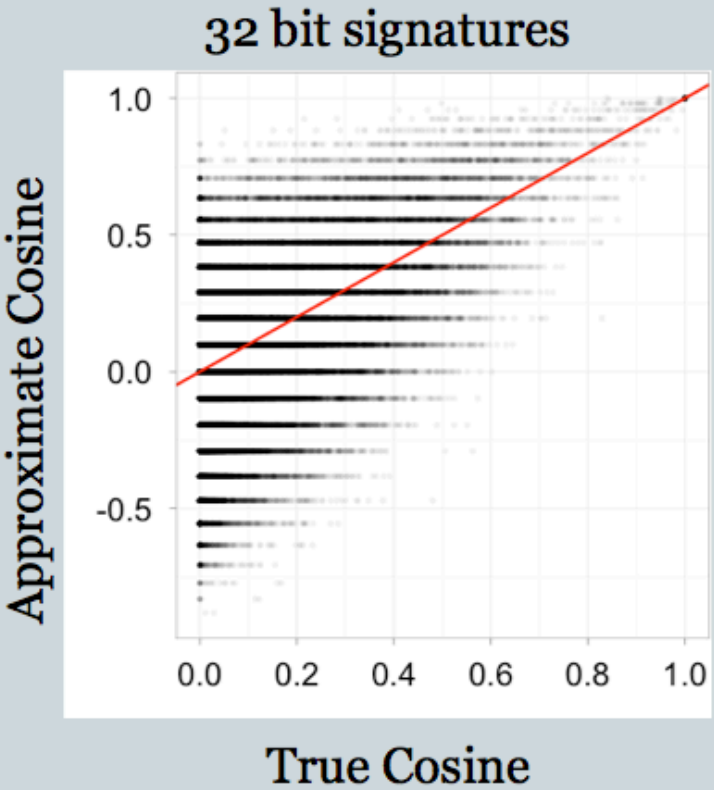


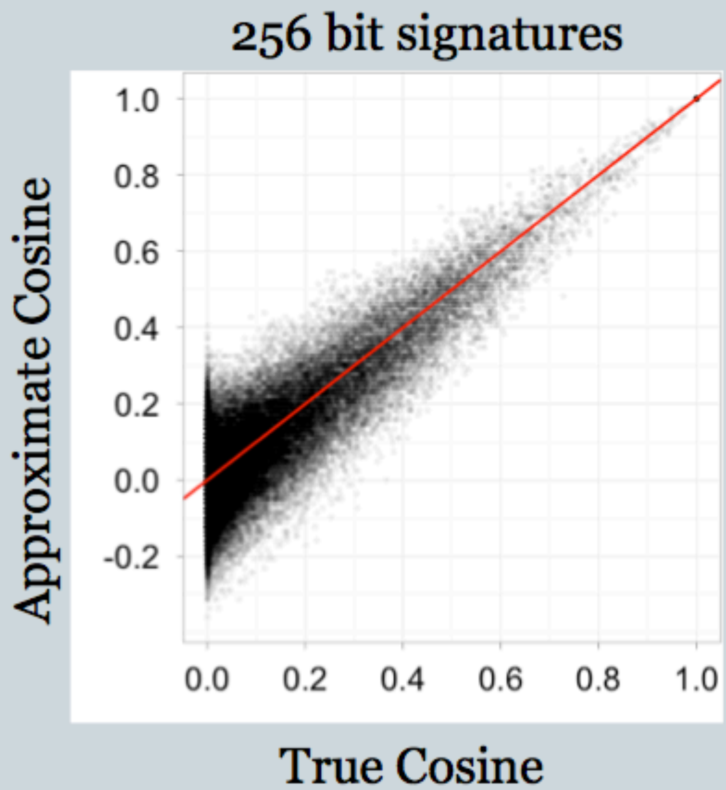


$$\text{Hamming Distance} := h = 1$$

$$\text{Signature Length} := b = 6$$

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





Cheap

Accurate

LSH applications

- Compact storage of data
  - and we can still compute similarities
- LSH also gives very fast …:
  - approx nearest neighbor method
    - just look at other items with **bx'=bx**
    - also very fast nearest-neighbor methods for Hamming distance
  - approximate clustering/blocking
    - 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

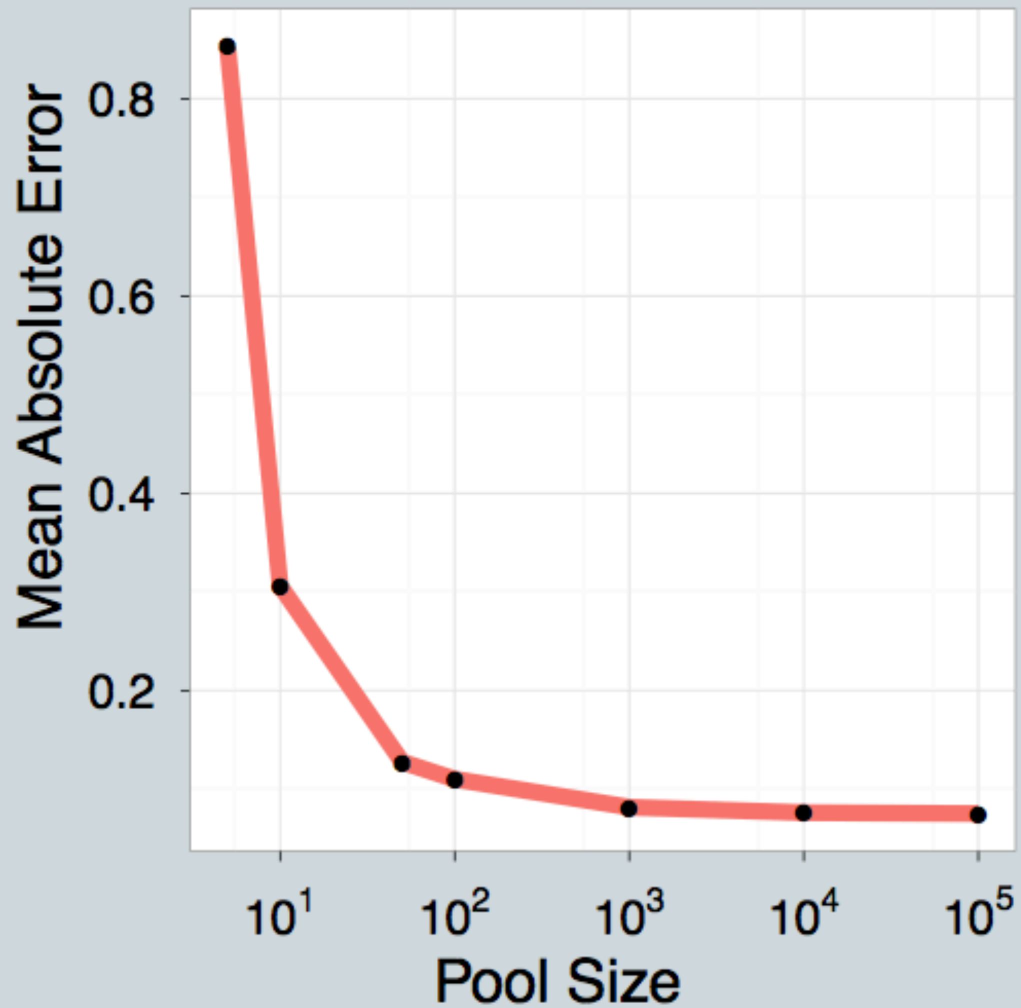
- But: storing the *k classifiers* is expensive in high dimensions
  - For each of 256 bits, a dense vector of weights for every feature in the vocabulary
- Storing seeds and random number generators:
  - Possible but somewhat fragile

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

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

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

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

if $\vec{v} = \Sigma_j \vec{v}_j$

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

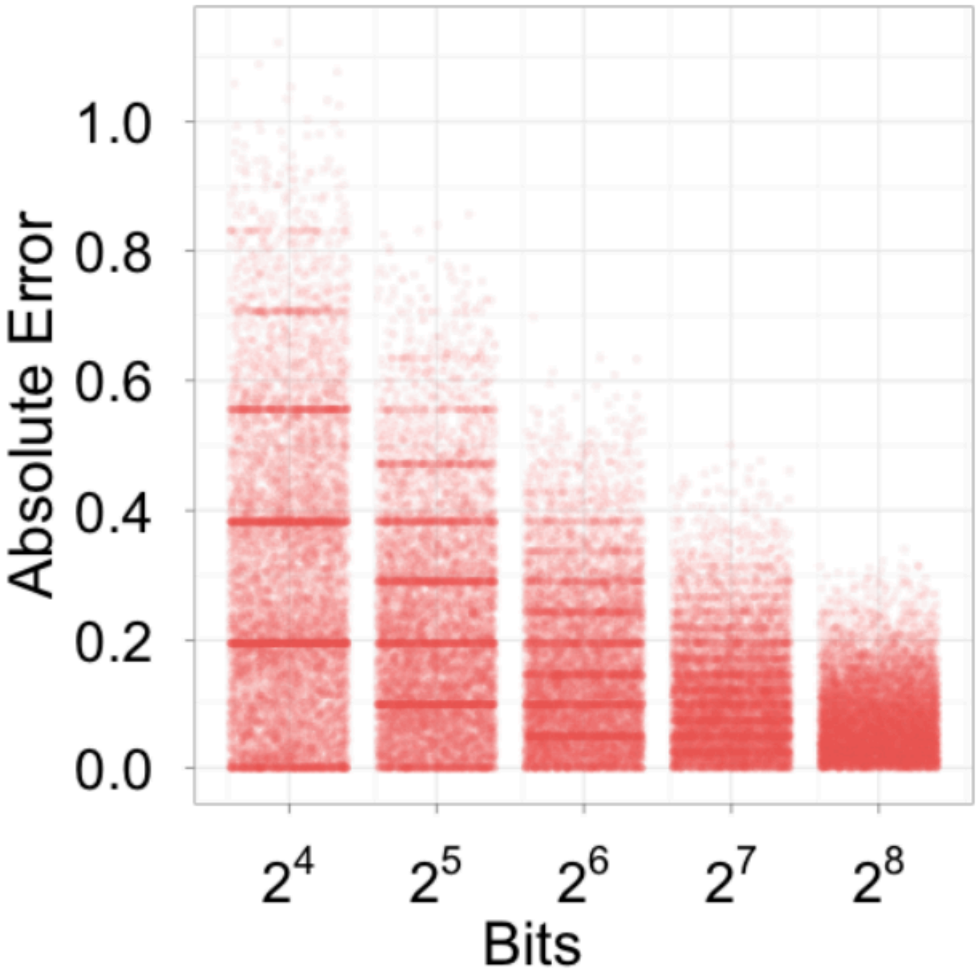
      $S[w][i] := 0$

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

similar to problem we looked at Tuesday using sketches

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