10-605

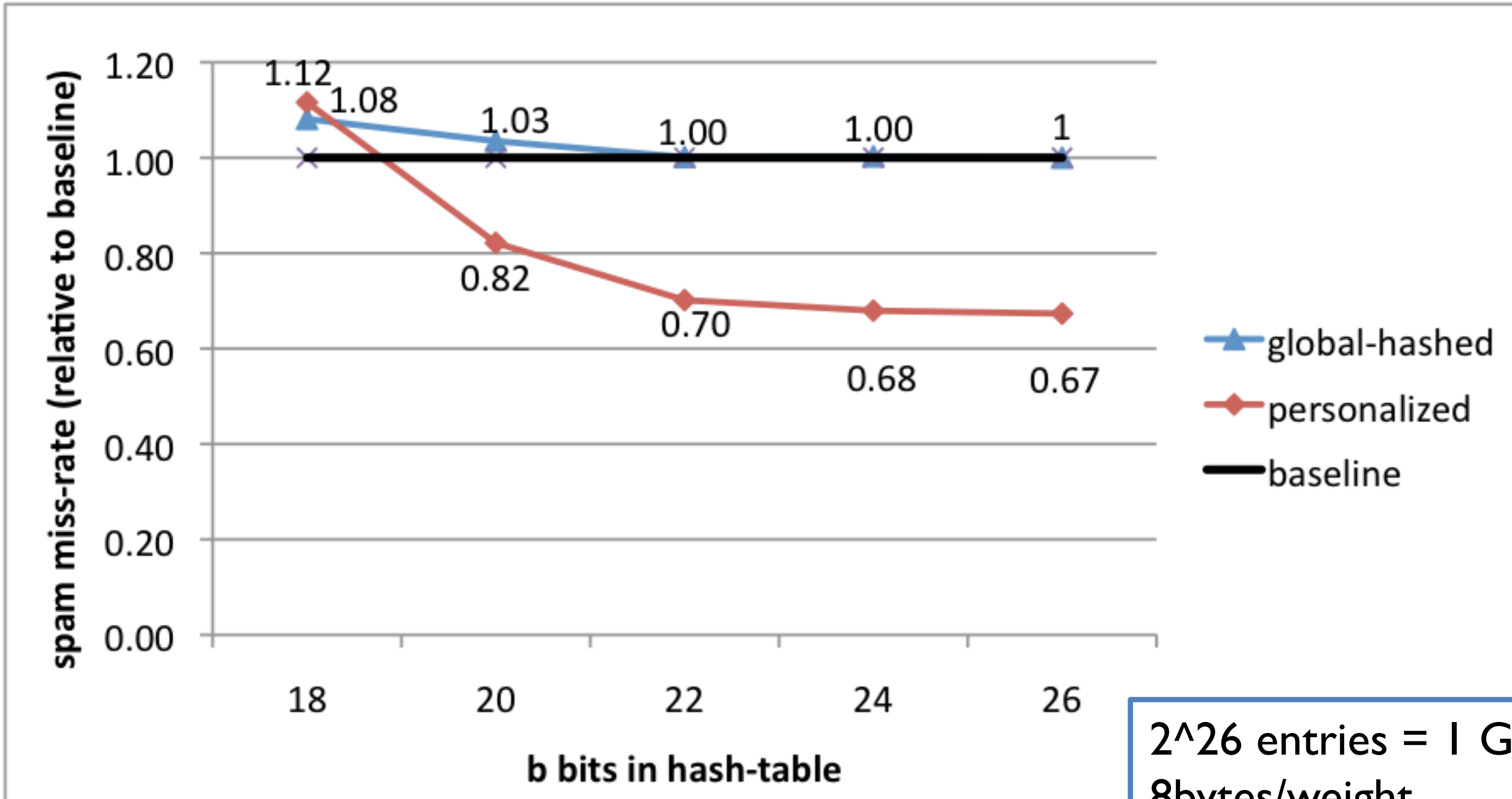
The course so far

- Map-reduce and parallel counting
- Streaming learning algorithms
  - some tricks like lazy L2 and hashing
- Parallelizing streaming learning algorithms
  - param mixing – for clusters
  - vectorization and minibatches – for GPUs
- Now: Some more tricks like hashing
  - randomized data structures aka *sketches*
  - these can make a huge difference in memory in many cases

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



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%.

MOTIVATING BLOOM FILTERS: VARIANT OF THE HASH TRICK

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

- **Bloom filters** are another famous trick that exploits these insights….

BLOOM FILTERS

Bloom filters

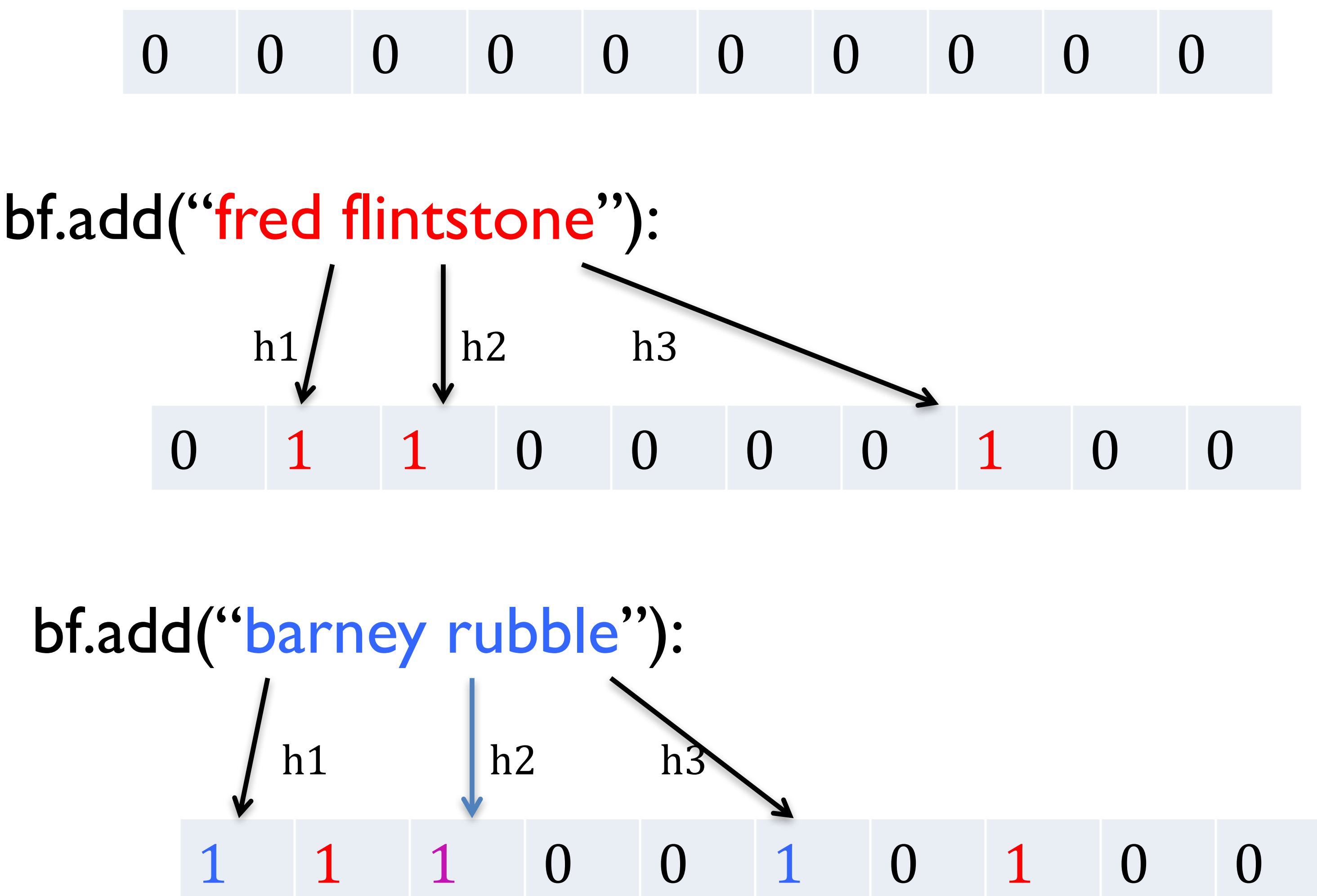
- Interface to a Bloom filter
  - BloomFilter(int maxSize, double p);
  - void bf.add(String s); // insert s
  - bool bd.contains(String s);
    - // If s was added return true;
    - // else with probability at least *1-p* return false;
    - // else with probability at most *p* return true;

  - 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

1	1	1	0	0	1	0	1	0	0

bf.contains (“fred flintstone”):

h1 h2

1	1	1	0	0	1	0	1	0	0

bf.contains(“barney rubble”):

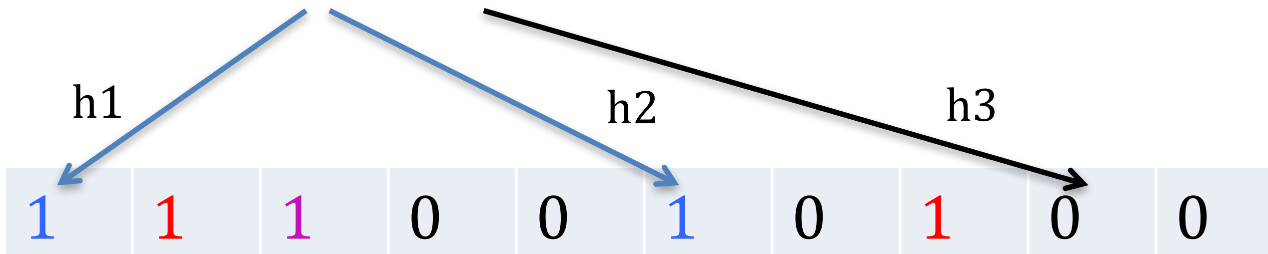
h1 h2 h3

1	1	1	0	0	1	0	1	0	0

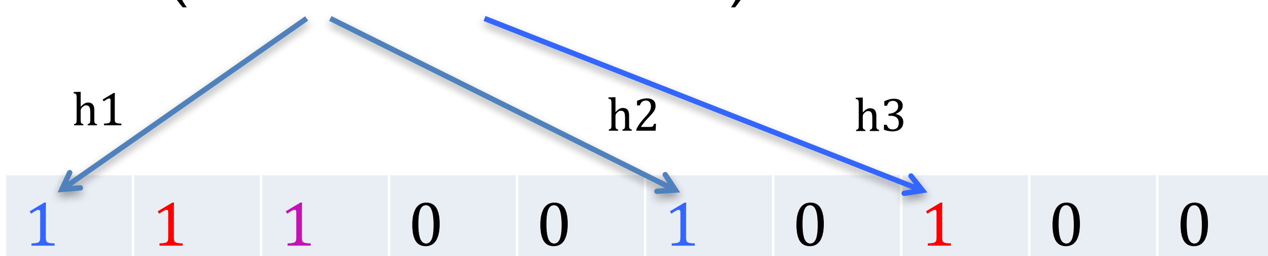
Bloom filters

1	1	1	0	0	1	0	1	0	0

bf.contains("wilma flintstone"):



bf.contains("wilma flintstone"):



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):

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

  - 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

What do you do with a Bloom Filter?

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)

```
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
```

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/#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

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

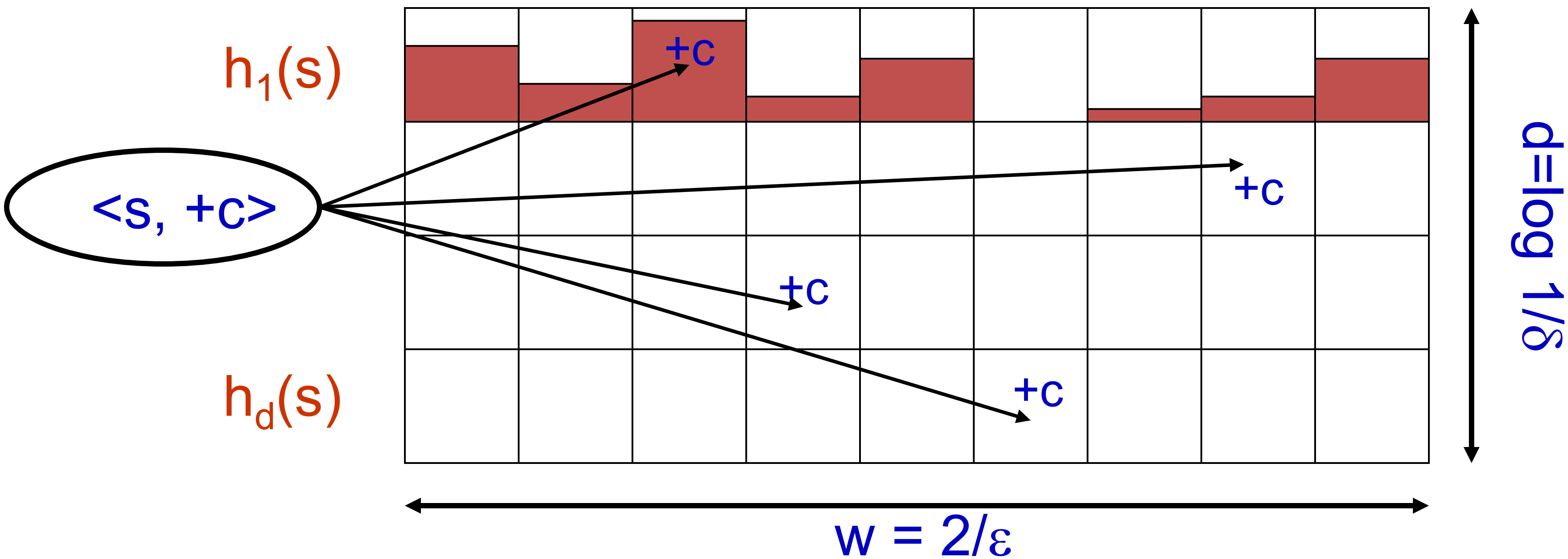
- An implementation
  - Allocate M bits, bit[0]…,bit[1-M]
  - Pick K hash functions hash(1,2),hash(2,s),….
    - E.g: hash(i,s) = hash(s+ randomString[i])
  - To add string s:
    - For i=1 to k, set bit[hash(i,s)] = 1
  - To check contains(s):
    - For i=1 to k, test bit[hash(i,s)]
    - Return “true” 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.

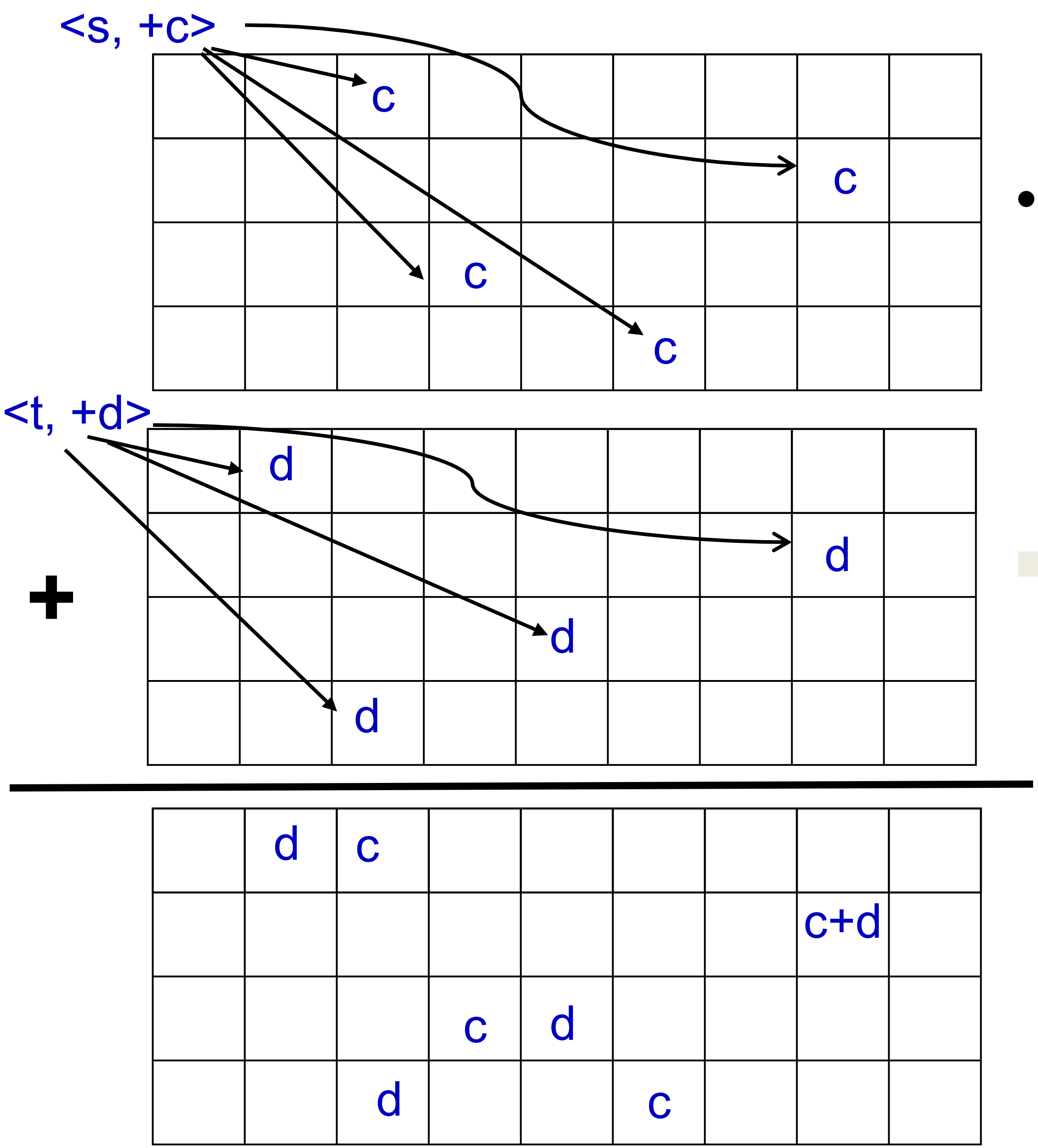
Question: what does this look like if d=1?

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: $d=\log 1/\delta$, $w=2/\varepsilon$ ➔ error is usually less than $e||A||_1$

i.e. with prob > 1-δ



- 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 $A[s] = Pr(s|\ldots)$ then $||A||_1 = 1$
  - Multiclass classification: if $A_x[s] = Pr(x \text{ in class } s)$ 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* $O(\ln \frac{1}{\delta})$.

What do you do with a count-min sketch?

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

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

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

An Application of a Count-Min Sketch

x appears near y

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

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

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.

Deep Learning and Sketches

ICLR 2017

Short and Deep: Sketching and Neural Networks

Amit Daniely, Nevena Lazic, Yoram Singer, Kunal Talwar*

ICLR 2017

Short and Deep: Sketching and Neur

weights and reLU compute the AND that decodes x_{24}

summation w/o nonlinearity

Nevena Lazic, Yoram S ger, Kunal Talwar*

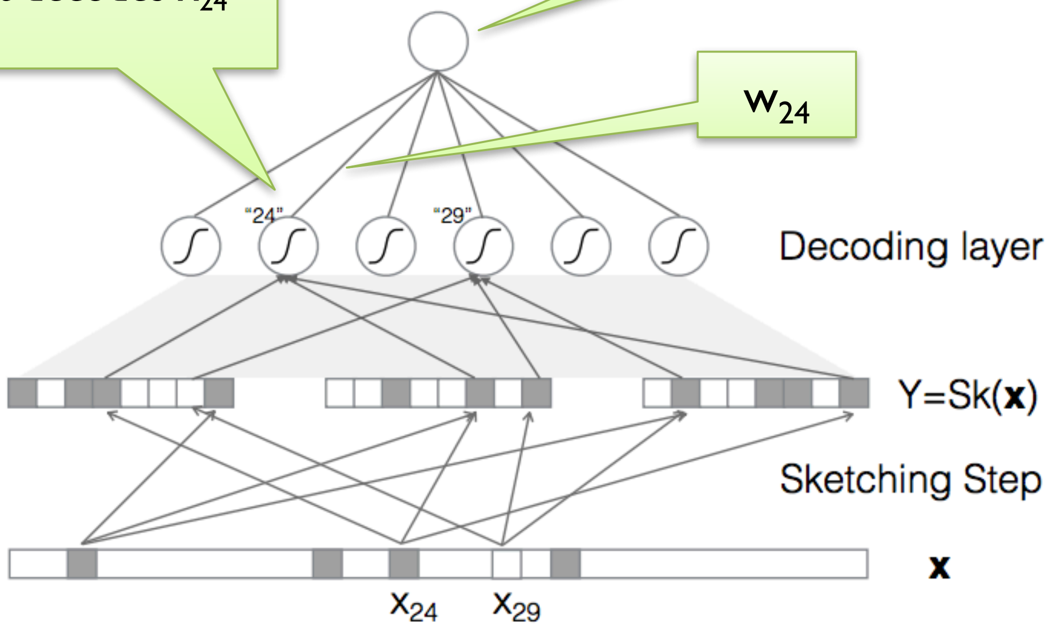


Figure 1: Neural-network sketching: sparse vector **x** maps to sketch using $t = 3$ hashes & $m = 8$; shaded squares designate 1's; sketching step is random; sketch then used as input to single-layer net: $\mathbf{w}^\top \mathbf{x}$; nodes labelled "24" & "29" correspond to decoding of x_{24} & x_{29} and shown with non-zero incoming edges.

Claim: for any **w** there is a one-layer neural network that can compute* **<w,x>** using as input a Bloom-filter sketch** of **x**, if **x** is a *k*-hot binary vector over *d* dimensions.

* with prob $\geq 1 - \delta$

** of size $O(ek \log ||\mathbf{w}||_0/\delta)$

ICLR 2017

Short and Deep: Sketching and Neural Networks

Nevena Lazic, Y… …war*

weights and reLU compute the AND that decodes x_{24}

now weights and reLU compute x_{24}* x_{29}

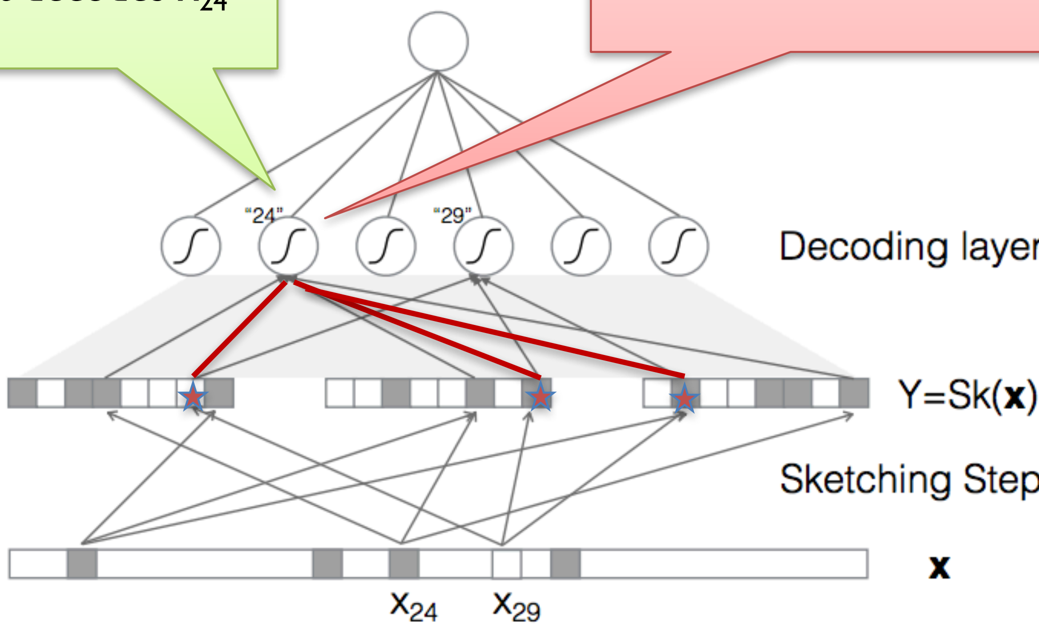
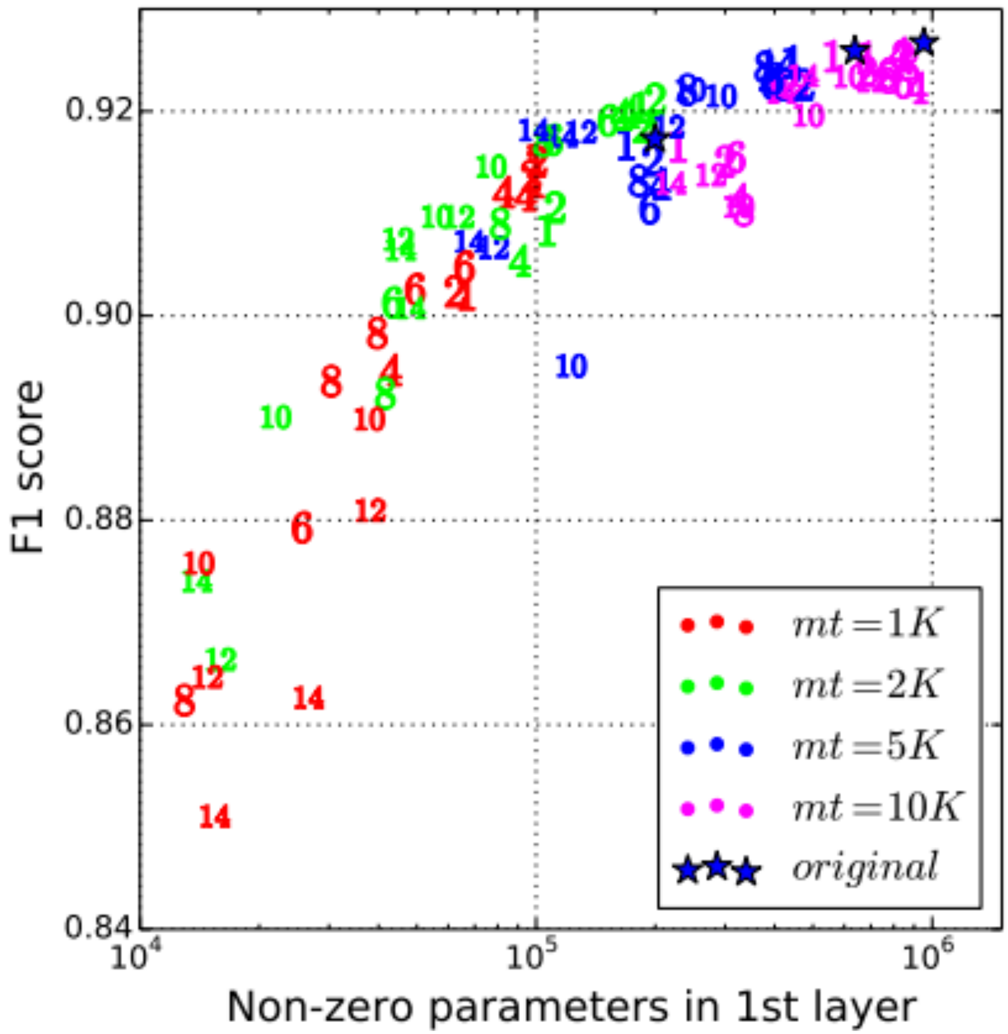


Figure 1: Neural-network sketching: sparse vector **x** maps to sketch using $t = 3$ hashes & $m = 8$; shaded squares designate 1's; sketching step is random; sketch then used as input to single-layer net: $\mathbf{w}^\top \mathbf{x}$; nodes labelled "24" & "29" correspond to decoding of x_{24} & x_{29} and shown with non-zero incoming edges.

Also:

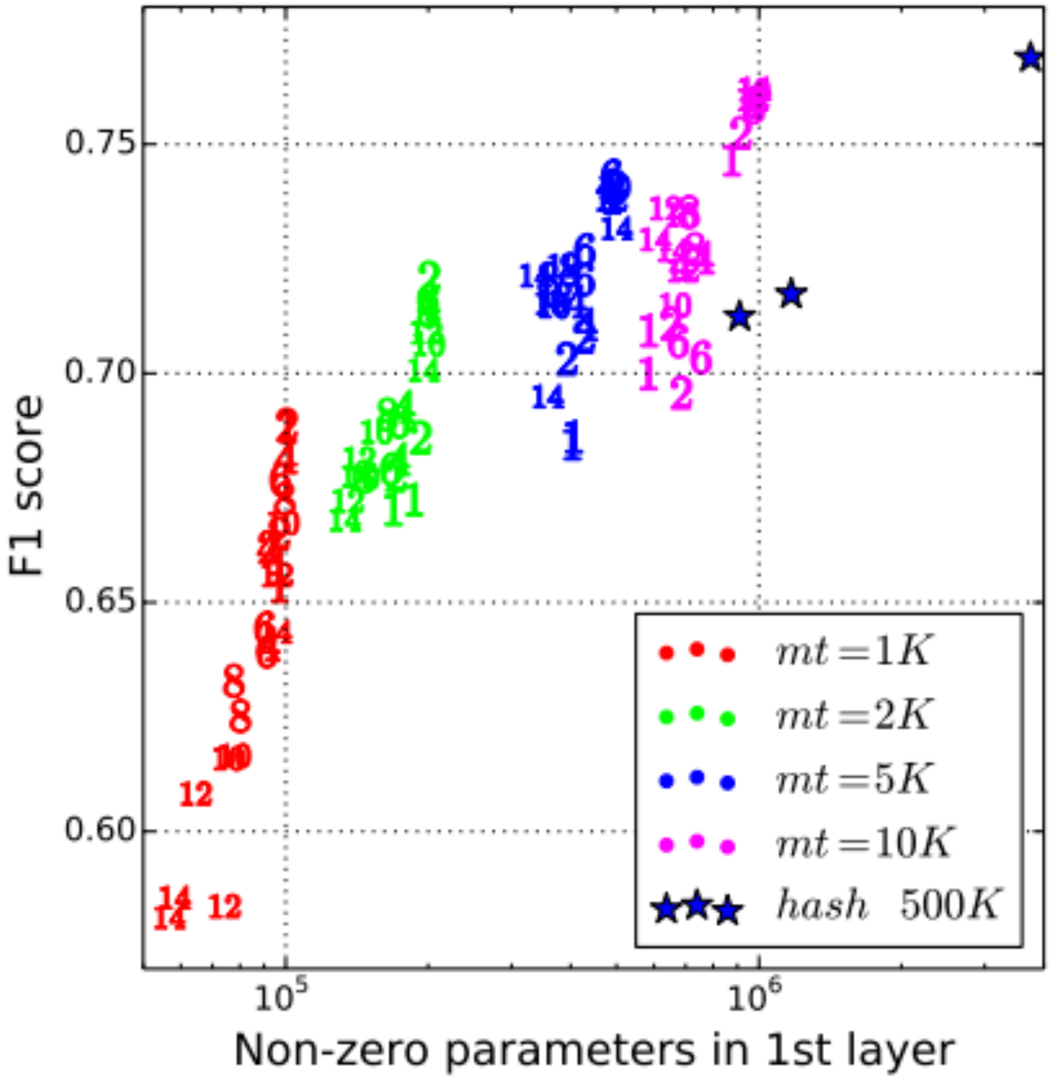
- weights in network are between 0 and 1
- only $O(ek \log ||\mathbf{w}||_0/\delta)$ of the weights are non-zero
- you can express more complex functions with a few more non-zero weights – e.g. polynomial kernels
- we can learn these networks using an L1 regularizer



RCV1, 4 categories, 113k dimensions, most examples are 120-sparse

mt is sketch size

3 values of L1-regularizer are used



entity-tagging task with very large feature vocabulary

mt is sketch size

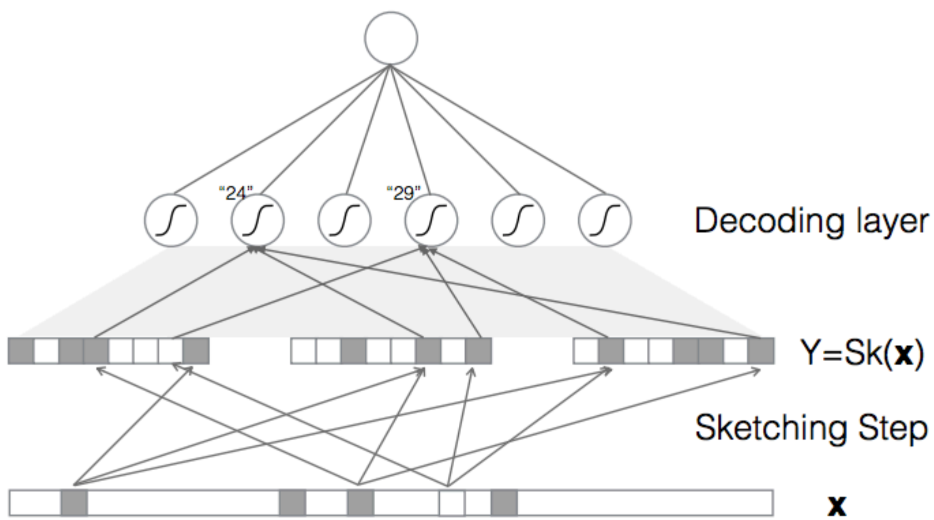
3 values of L1-regularizer are used

compared to feature hashing

Compact Embedding of Binary-coded Inputs and Outputs using Bloom Filters

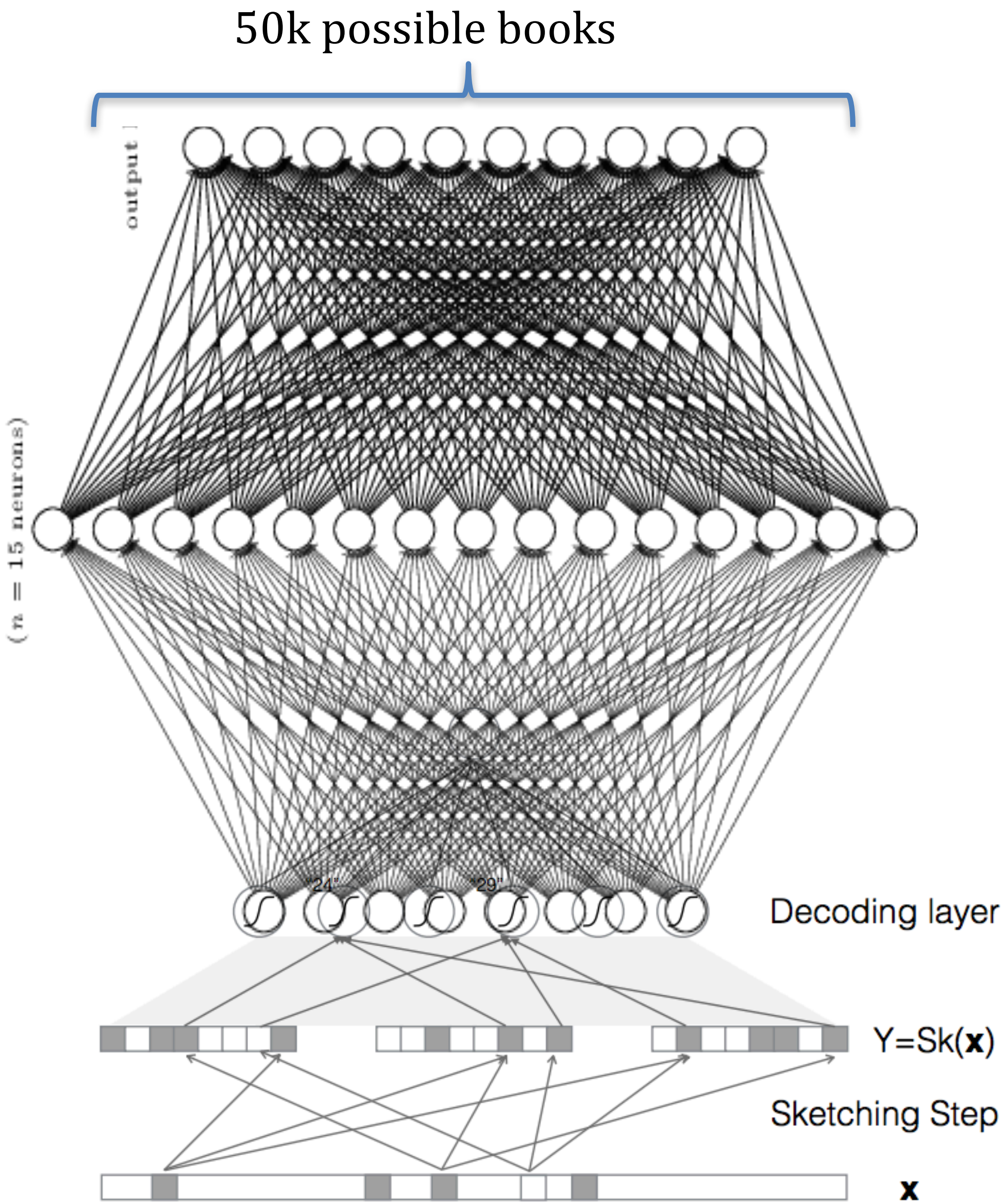
ICLR 2017

Joan Serrà & Alexandros Karatzoglou
Telefónica Research
Pl. Ernest Lluch i Martín, 5
Barcelona, 08019, Spain
firstname.lastname@telefonica.com



What if you are mapping many inputs to many possible outputs?

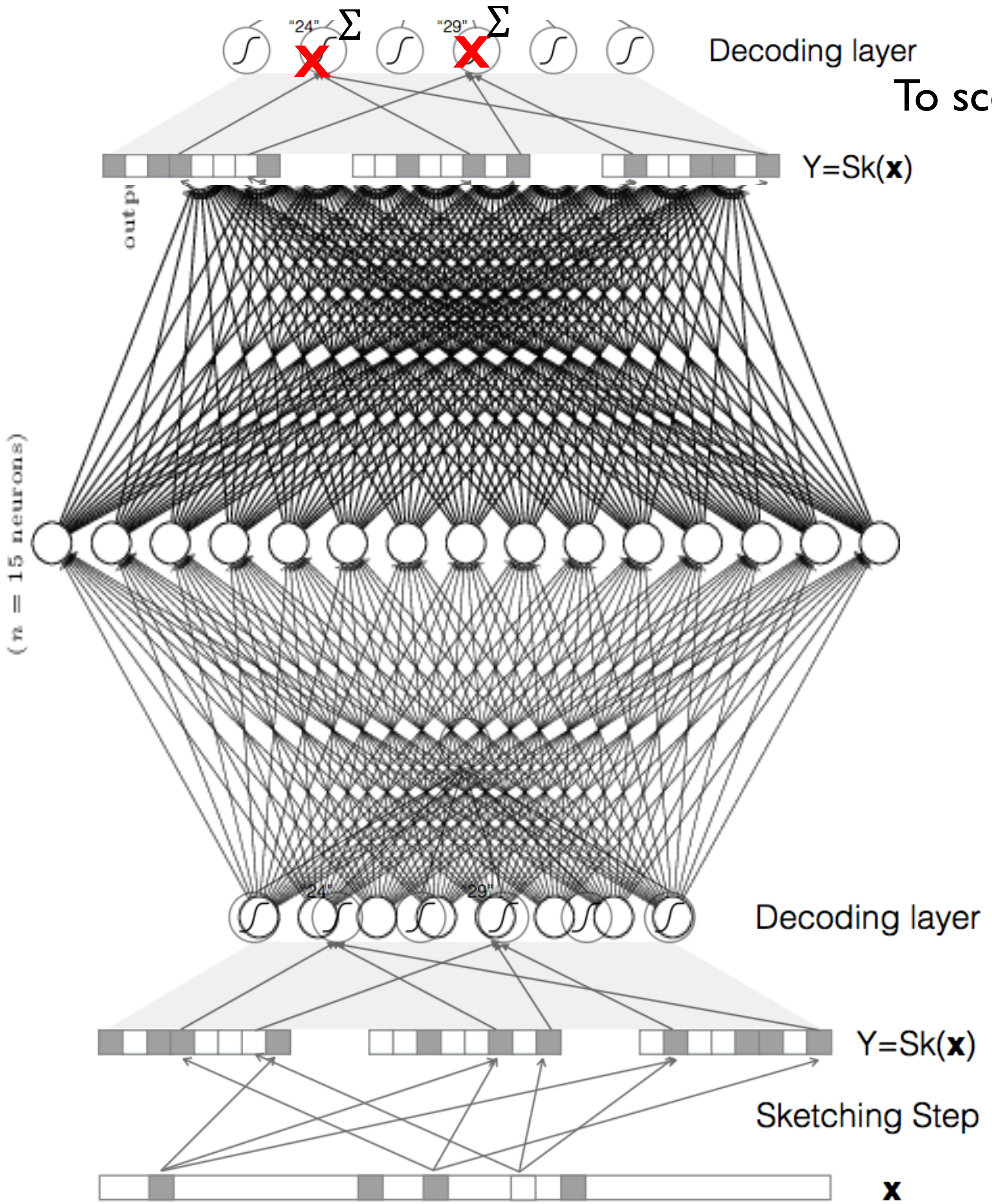
- Song recommendation: output is a song
- Language modeling: output is a word
- …



What if you are mapping many inputs to many possible outputs?

- Book recommendation: output is a book
- Language modeling: output is a word

- Solution: output a sketch!



To score y, addup the scores of the codes for y

Softmax predicts the BF encoding bits

What if you are mapping many inputs to many possible outputs?

- Book recommendation: output is a book
- Language modeling: output is a word

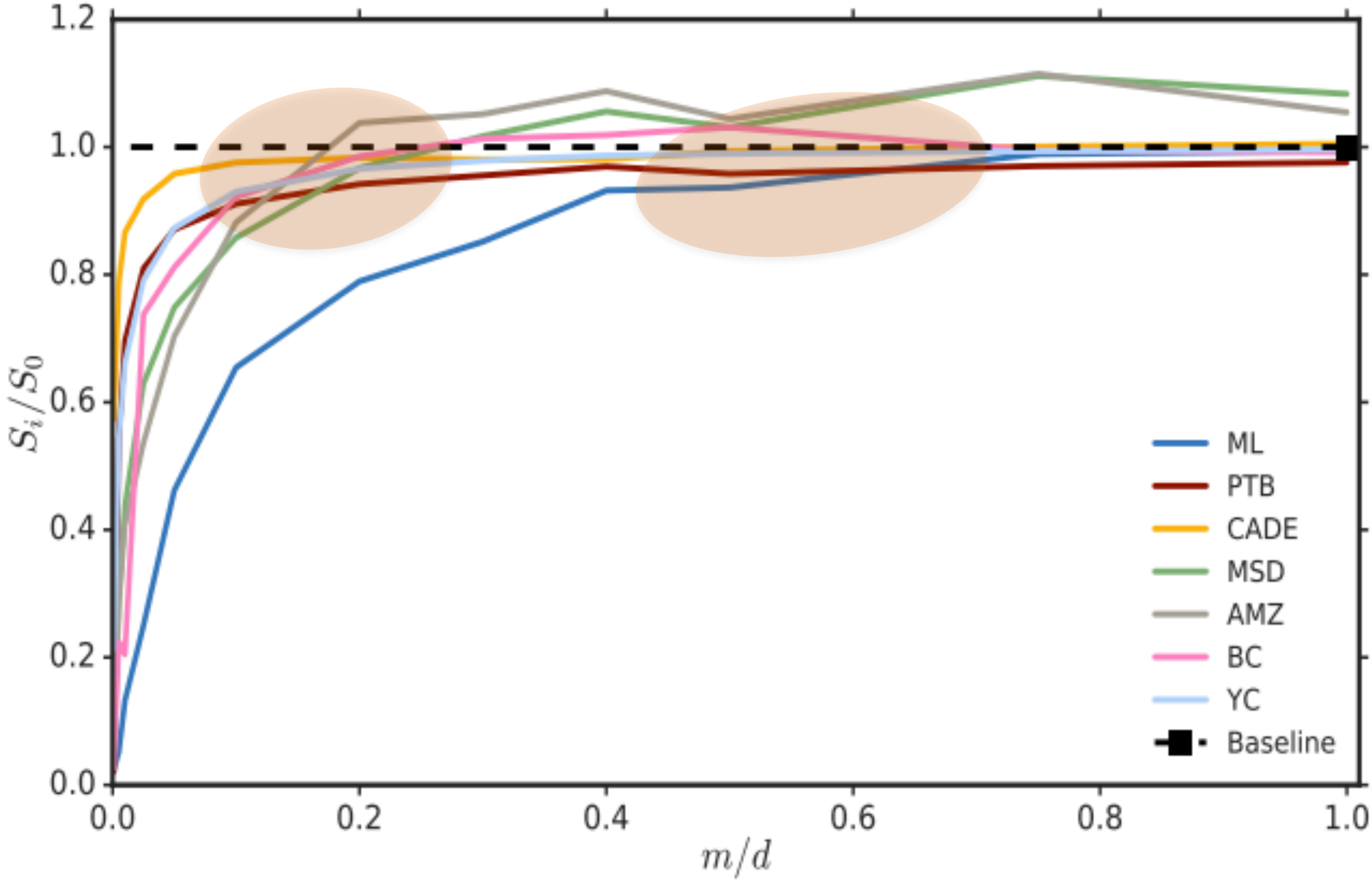
- Solution: output a sketch!

Table 1: Data set statistics after data cleaning and splitting. From left to right: data set name, number of instances n, test split size, instance dimensionality d, median number of non-zero components c, and median density c/d.

Data set	n	Split	d	c	c/d
ML	138,224	10,000	15,405	18	$1.2 \cdot 10^{-3}$
PTB	929,589	82,430	10,001	1	$1.0 \cdot 10^{-4}$
CADE	40,983	13,661	193,998	17	$8.8 \cdot 10^{-5}$
MSD	597,155	50,000	69,989	5	$7.1 \cdot 10^{-5}$
AMZ	916,484	50,000	22,561	1	$4.4 \cdot 10^{-5}$
BC	25,816	2,500	54,069	2	$3.7 \cdot 10^{-5}$
YC	1,865,997	50,000	35,732	1	$2.8 \cdot 10^{-5}$

Table 2: Experimental setup and baseline scores. From left to right: data set name, network architecture and optimizer, evaluation measure, random score S_R, and baseline score S_0.

Data set	Architecture + Optimizer	Meas.	S_R	S_0
ML	Feed-forward + Adam	MAP	0.003	0.160
PTB	LSTM + SGD	RR	0.001	0.342
CADE	Feed-forward + RMSprop	Acc	8.5	58.0
MSD	Feed-forward + Adam	MAP	<0.001	0.066
AMZ	Feed-forward + Adam	MAP	<0.001	0.049
BC	Feed-forward + Adam	MAP	<0.001	0.010
YC	GRU + Adagrad	RR	<0.001	0.368



1.2
1.0
0.8
0.6
0.4
0.2
0.0
S_i/S_0
0.0
0.2
0.4
0.6
0.8
1.0
m/d
ML
PTB
CADE
MSD
AMZ
BC
YC
Baseline