

# LSML #8

С чего начали, тем и закончим: хэширование

# Хэширование полезно

- Уже знаем:
  - Хэширование признаков (1 лекция)
- Узнаем сегодня для **больших** множеств:
  - Определение принадлежности множеству (Bloom Filter)
  - Оценка частот элементов множества в потоке (Count-min sketch)
  - Оценка похожести двух множеств (MinHash)
  - Отбор самых похожих множеств на заданное (LSH)

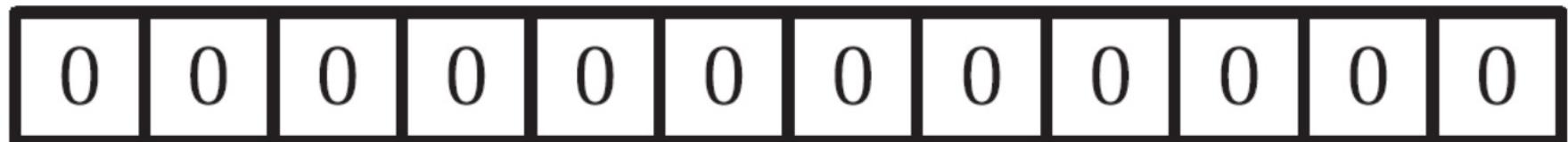
**Будем решать приближенно!**

# Bloom Filter (1970)

- **Определение принадлежности множеству (в память не лезет)**
- Bloom Filter – это массив из  $t$  бит, который хранит информацию о множестве  $S = \{x_1, x_2, \dots, x_n\}$ . Начинаем с массива нулей.
- Для работы фильтра нам нужно  $k$  независимых хэш-функций  $h_1, \dots, h_k$  с  $t$  корзинками (равномерно раскидывают по корзинкам).
- При вставке каждого элемента  $x \in S$  выставляем биты  $h_i(x)$  в 1 для всех  $i \in [1, k]$ .

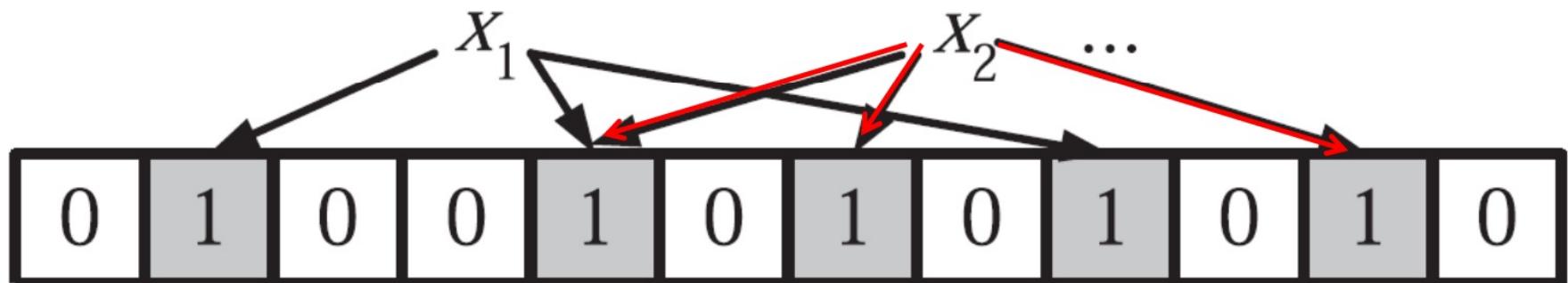
# Вставка и проверка принадлежности

$m = 12$



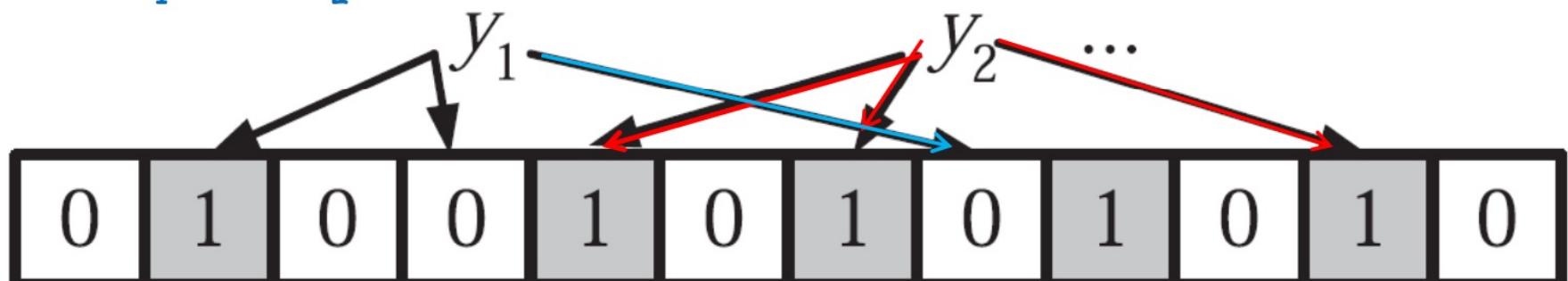
Insert  $X_1$  and  $X_2$

$k = 3$



Check  $Y_1$  and  $Y_2$

$y_1$  **точно нет**  
 $y_2$  **возможно** есть



# То есть может быть только false positive

- Вероятность того, что  $h_i(x)$  не выставит бит  $j$ :  $1 - \frac{1}{m}$
- Вероятность того, что ни одна из  $k$  хэш-функций не выставит:  $\left(1 - \frac{1}{m}\right)^k$
- Мы вставили  $n$  элементов, значит вероятность не выставить:  $\left(1 - \frac{1}{m}\right)^{kn}$
- Вероятность того, что бит  $j$  в фильтре выставлен:  $1 - \left(1 - \frac{1}{m}\right)^{kn}$
- Вероятность того, что  $k$  бит будут уже выставлены:

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

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

# Оптимальные параметры для заданной ошибки

- Минимизируем  $(1 - e^{-kn/m})^k$  по  $k$ :  $k = \frac{m}{n} \ln 2$

substituting the optimal value of  $k$  in the probability expression above:

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

which can be simplified to:

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

This results in:

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

$$\frac{m}{n} = -\frac{\log_2 p}{\ln 2} \approx -1.44 \log_2 p$$

with the corresponding number of hash functions  $k$  (ignoring integrality):

$$k = -\frac{\ln p}{\ln 2} = -\log_2 p.$$

Хотим:

$$n = 10^6 \quad p = 0.01$$

Рассчитываем:

$$m = 10^7 \quad k = 7$$

# А профит вообще есть?

- Предыдущий пример:

$$n = 10^6 \quad p = 0.01 \quad m = 10^7 \quad k = 7$$

- Пусть множество из строк длины 1024 (например URL страниц)
- Для хранения множества нужно  $1024 \times 10^6$  байт, то есть **1 ГБ**
- Для хранения фильтра:  $10^7$  бит, то есть **1.2 МБ**

# Примеры применения

- Medium uses Bloom filters to avoid recommending articles a user has previously read.
- Bitcoin uses Bloom filters to speed up wallet synchronization.
- Google Bigtable, Apache HBase and Apache Cassandra, and PostgreSQL use Bloom filters to reduce the disk lookups for non-existent rows or columns.

# Пример из ML практики

- Нужно сделать **join**:
  - огромной таблицы с логами юзеров
  - и набором интересующих юзеров (*не влезает в память*)
- Простой join на MapReduce будет шафлить (куча пересылок по сети) огромное количество логов зря → медленно
- Сделаем Map шаг (без пересылок по сети) с фильтрацией огромной таблицы при помощи Bloom Filter → суммарно в 4 раза быстрее

# Операции над множествами

- Bloom Filter заменяет нам само множество с небольшими потерями
- Для объединения множеств нужно применить побитовый OR
- Для пересечения множеств – побитовый AND
- Вычитать сложнее – нужен Counting Bloom Filter...
- Создание Bloom Filter легко распараллелить!

# Count-min sketch (2003)

- **Оценка частот элементов множества в потоке (поток и множество большие)**
- Заведем двумерный массив из  $d$  строк и  $w$  столбцов.
- Положим  $w = \lceil e/\varepsilon \rceil$  и  $d = \lceil \ln 1/\delta \rceil$ , где ошибка в запросе частоты в пределах аддитивной добавки  $\varepsilon$  с вероятностью  $1 - \delta$ .

# Модель потока

- В момент времени  $t$  счетчики для всех  $n$  элементов:

$$\vec{a}(t) = (a_1(t), \dots, a_i(t), \dots a_n(t))$$

- Начинаем с нулей:

$$a_i(0) = 0 \quad \forall i$$

- На шаге  $t$  увеличиваем счетчик элемента  $i_t$  на  $c_t$ :

$$a_{i'}(t) = a_{i'}(t-1) \quad \forall i' \neq i_t$$

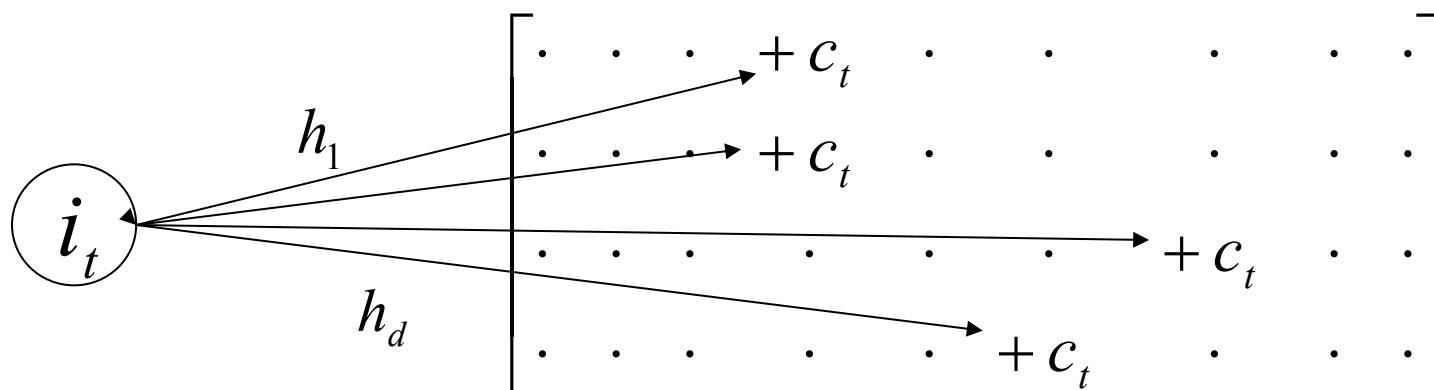
$$a_{i_t}(t) = a_{i_t}(t-1) + c_t$$

# Обновление массива

- Пусть  $h_i(x), i \in [1, d]$  – попарно независимые хэш-функции с  $w$  корзинками

When  $(i_t, c_t)$  arrives, set  $\forall 1 \leq j \leq d$

$$count[j, h_j(i_t)] \leftarrow count[j, h_j(i_t)] + c_t$$



Будем хранить счетчики, а не частоту

# Какие запросы можно делать

- point query       $Q(i)$             approx.  $a_i$

- range queries     $Q(l, r)$         approx.  $\sum_{i=l}^r a_i$

- inner product queries  $Q(\vec{a}, \vec{b})$         approx.  $\vec{a} \cdot \vec{b} = \sum_{i=1}^n a_i b_i$

# Point query ( $a_{i_t}(t) > 0$ )

$$Q(i) \rightarrow \hat{a}_i = \min_j count[j, h_j(i)]$$

- Теорема:  $a_i \leq \hat{a}_i$   $P[\hat{a}_i > a_i + \varepsilon \|\vec{a}\|_1] \leq \delta$

$$P\left[\frac{\hat{a}_i}{\|\vec{a}\|_1} > \frac{a_i}{\|\vec{a}\|_1} + \varepsilon\right] \leq \delta$$

То есть частота оценивается хорошо

# Point query ( $a_{i_t}(t) > 0$ )

$$Q(i) \rightarrow \hat{a}_i = \min_j count[j, h_j(i)]$$

- Теорема:   $a_i \leq \hat{a}_i$   $P[\hat{a}_i > a_i + \varepsilon \|\vec{a}\|_1] \leq \delta$

$$I_{i,j,k} = \begin{cases} 1 & \text{if } (i \neq k) \wedge (h_j(i) = h_j(k)) \\ 0 & \text{otherwise} \end{cases}$$

$$w = [e/\varepsilon] \quad E(I_{i,j,k}) = \Pr[h_j(i) = h_j(k)] \leq \frac{1}{w} = \frac{\varepsilon}{e}$$

$$X_{i,j} = \sum_{k=1}^n I_{i,j,k} a_k \implies count[j, h_j(i)] = a_i + X_{i,j} \implies \min count[j, h_j(i)] \geq a_i$$

# Point query ( $a_{i_t}(t) > 0$ )

$$Q(i) \rightarrow \hat{a}_i = \min_j count[j, h_j(i)]$$

- Теорема:  $\star a_i \leq \hat{a}_i \quad \star P[\hat{a}_i > a_i + \varepsilon \|\vec{a}\|_1] \leq \delta$

$$E(X_{i,j}) = E\left(\sum_{k=1}^n I_{i,j,k} a_k\right) = \sum_{k=1}^n a_k E(I_{i,j,k}) \leq \frac{\varepsilon}{e} \|\vec{a}\|_1$$

$$\Pr[\hat{a}_i > a_i + \varepsilon \|\vec{a}\|_1] = \Pr[\forall j. count[j, h_j(i)] > a_i + \varepsilon \|\vec{a}\|_1]$$

для всех  $j$  одновременно

$$= \Pr[\forall j. a_i + X_{i,j} > a_i + \varepsilon \|\vec{a}\|_1]$$

$$\leq \Pr[\forall j. X_{i,j} > eE(X_{i,j})] < e^{-d} \leq \delta$$

$$d = \lceil \ln 1/\delta \rceil$$

Markov inequality

$$\Pr[X \geq t] \leq \frac{E(X)}{t} \quad \forall t > 0$$



# Оценки сложности

$$w = \lceil e/\varepsilon \rceil$$

$$d = \lceil \ln 1/\delta \rceil$$

Time to produce the estimate

$$O(\ln \frac{1}{\delta})$$

Space used

$$O(\frac{1}{\varepsilon} \ln \frac{1}{\delta})$$

Time for updates

$$O(\ln \frac{1}{\delta})$$

# Inner Product Query

Есть счетчики для двух потоков:  $a$  и  $b$

Set  $(\vec{a} \cdot \vec{b})_j = \sum_{k=1}^w count_{\vec{a}}[j, k] * count_{\vec{b}}[j, k]$

$$Q(\vec{a}, \vec{b}) \rightarrow (\vec{a} \cdot \vec{b}) = \min_j (\vec{a} \cdot \vec{b})_j$$

Например, для оценки размера **join** двух таблиц:  
`x join y on x.a=y.b`

# Inner Product Query

- Теорема:  $(\vec{a} \cdot \vec{b}) \leq (\hat{\vec{a}} \cdot \hat{\vec{b}})$   $\Pr[(\hat{\vec{a}} \cdot \hat{\vec{b}}) > \vec{a} \cdot \vec{b} + \varepsilon \|\vec{a}\|_1 \|\vec{b}\|_1] \leq \delta$

$$(\hat{\vec{a}} \cdot \hat{\vec{b}})_j = \sum_{k=1}^w count_{\vec{a}}[j, k] * count_{\vec{b}}[j, k]$$

$$(\hat{\vec{a}} \cdot \hat{\vec{b}}) = \min_j (\hat{\vec{a}} \cdot \hat{\vec{b}})_j$$

$$(\hat{\vec{a}} \cdot \hat{\vec{b}})_j = \sum_{i=1}^n a_i b_i + \sum_{p \neq q, h_j(p) = h_j(q)} a_p b_q \rightarrow (\hat{\vec{a}} \cdot \hat{\vec{b}}) \leq (\hat{\vec{a}} \cdot \hat{\vec{b}})$$

$$E(\hat{\vec{a}} \cdot \hat{\vec{b}} - \vec{a} \cdot \vec{b}) = \sum_{p \neq q} \Pr[h_j(p) = h_j(q)] a_p b_q \leq \sum_{p \neq q} \frac{\varepsilon a_p b_q}{e} \leq \frac{\varepsilon \|\vec{a}\|_1 \|\vec{b}\|_1}{e}$$

Так же, как в прошлый раз

Markov inequality  
 $\Pr[X \geq t] \leq \frac{E(X)}{t} \quad \forall t > 0$



$$\Pr[\hat{\vec{a}} \cdot \hat{\vec{b}} - \vec{a} \cdot \vec{b} > \varepsilon \|\vec{a}\|_1 \|\vec{b}\|_1] \leq \delta$$

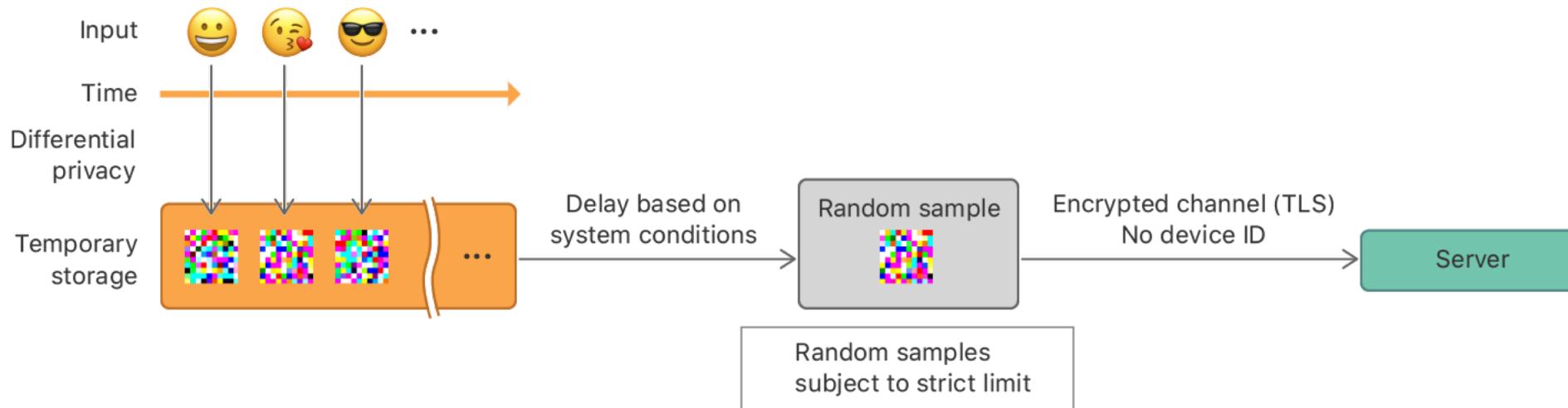


# Примеры применений

- Статистики для больших корпусов в NLP
  - <http://www.aclweb.org/anthology/D12-1100>
- Приближенные статистики в Apache Spark
  - <https://databricks.com/blog/2016/05/19/approximate-algorithms-in-apache-spark-hyperloglog-and-quantiles.html>
  - <https://mapr.com/blog/some-important-streaming-algorithms-you-should-know-about/>

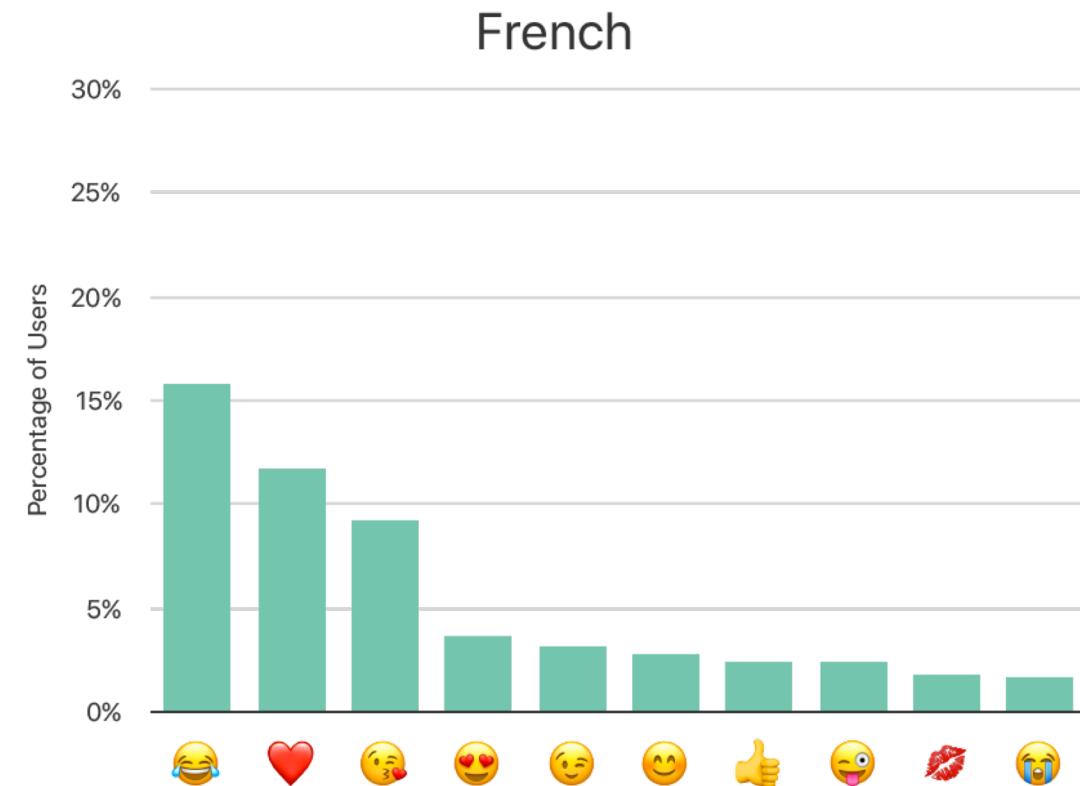
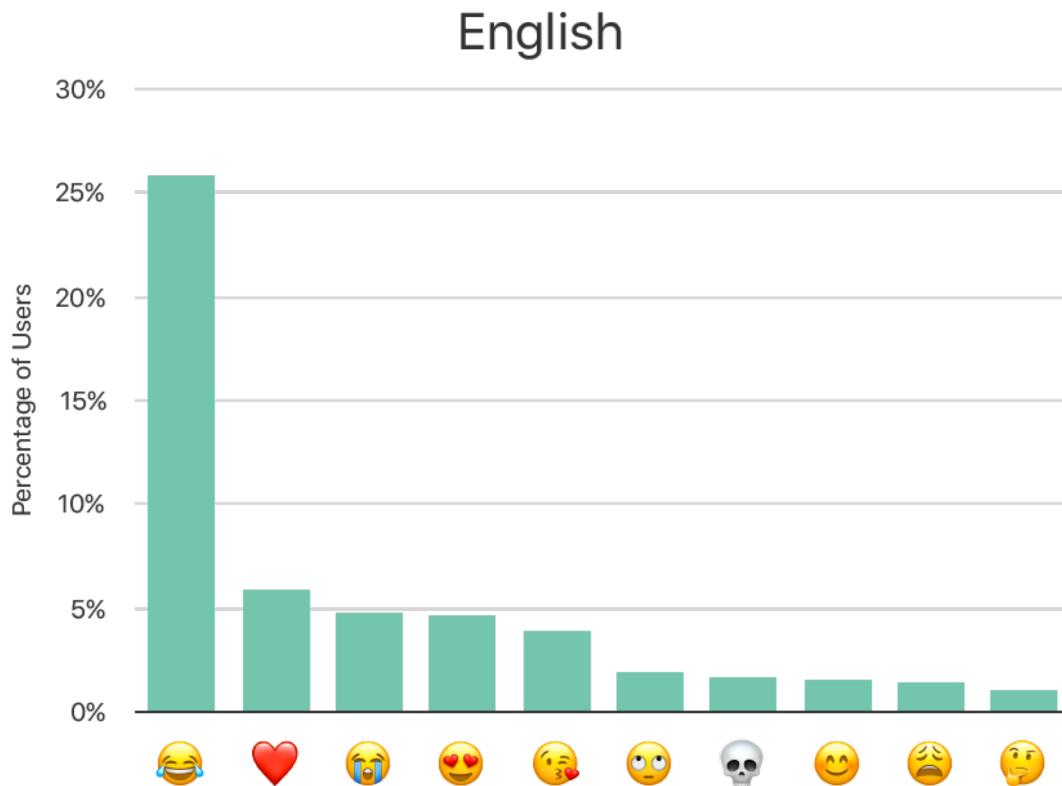
# Пример от Apple

- Learning with Privacy at Scale (2017)
- Нельзя передавать в открытую все, что печатает юзер на клавиатуре
- Отшлем Count-Mean Sketch и проверим только интересные нам слова



# Пример от Apple

- Можно восстановить распределение



# MinHash (1997)

- **Оценка похожести двух множеств (больших)**
- Мера Жаккара для двух множеств:  $J(S_1, S_2) = \frac{|S_1 \cap S_2|}{|S_1 \cup S_2|}$
- Представим множества  $S_1$  и  $S_2$  как столбцы в пространстве универсальных элементов  $E_i$ :

	$S_1$	$S_2$	
$E_1$	1	1	
$E_2$	0	1	$J(S_1, S_2) = \frac{2}{3}$
$E_3$	1	1	
$E_4$	0	0	

# MinHash

Всего **4 группы** элементов:

	$s_1$	$s_2$
$A$	1	1
$B$	1	0
$ A  = A$		
$C$	0	1
$D$	0	0

$$J(s_1, s_2) = \frac{A}{A + B + C}$$

# MinHash

Всего **4 группы** элементов:

	$S_1$	$S_2$
$A$	1	1
$B$	1	0
$C$	0	1
$D$	0	0

$$J(S_1, S_2) = \frac{A}{A + B + C}$$

$$|A| = A$$

- Случайно перемешаем строчки
- Хэш  $h(S_i)$  = индекс первой строчки с 1

$$P(h(S_1) = h(S_2)) = J(S_1, S_2)$$

- После случайной перестановки строк идем до **первой** строчки не из группы  $D$
- При этом хэши совпадут только если это строка из группы  $A$

# MinHash сигнатура

- По одному хэшу ничего не понятно (совпал или нет)
- Выберем  $k$  случайных перестановок
- Сигнатура  $\text{sig}(S) = (h_1(S), \dots, h_k(S))$
- Пусть  $\text{sim}(\text{sig}(S_i), \text{sig}(S_j))$  – процент совпадений в сигнатаурах
- Мат. ожидание этой похожести –  $J(S_i, S_j)$
- В силу ЗБЧ с ростом  $k$  ошибка убывает как  $= O\left(\frac{1}{\sqrt{k}}\right)$
- Для ошибки 0.05 хватит  $\approx 400$  хэш-функций

# MinHash сигнатура: пример

	C <sub>1</sub>	C <sub>2</sub>	C <sub>3</sub>
R <sub>1</sub>	1	0	1
R <sub>2</sub>	0	1	1
R <sub>3</sub>	1	0	0
R <sub>4</sub>	1	0	1
R <sub>5</sub>	0	1	0

## Signatures

	S <sub>1</sub>	S <sub>2</sub>	S <sub>3</sub>
Perm 1 = (12345)	1	2	1
Perm 2 = (54321)	4	5	4
Perm 3 = (34512)	3	5	4

## Similarities

	1-2	1-3	2-3
Col-Col	0.00	0.50	0.25
Sig-Sig	0.00	0.67	0.00

Заменили сравнение больших множеств на  
сравнение маленьких сигнатур!

# MinHash сигнатура: имплементация

- Генерировать перестановку элементов просто!
- Давайте возьмем хорошую хэш-функцию  $h(x)$  (murmur32 например)
- Сортировка по индексу корзинки  $h(x)$  даст случайную перестановку!
- Тогда MinHash считается легко: хэшируем все элементы и берем минимальное значение хэша

# MinHash сигнатура: имплементация

	$C_1$	$C_2$
$R_1$	1	0
$R_2$	0	1
$R_3$	1	1
$R_4$	1	0
$R_5$	0	1

Хэш-функции:

$$h(x) = x \bmod 5$$

$$g(x) = 2x+1 \bmod 5$$

	$C_1$ slots	$C_2$ slots	
$h(1) = 1$	1	-	Один проход
$g(1) = 3$	3	-	
$h(2) = 2$	1	2	
$g(2) = 0$	3	0	
$h(3) = 3$	1	2	
$g(3) = 2$	2	0	
$h(4) = 4$	1	2	
$g(4) = 4$	2	0	
$h(5) = 0$	1	0	Сигнтуры
$g(5) = 1$	2	0	

# MinHash сигнатура: использование

- Придумали для поисковика AltaVista для поиска дубликатов веб-страниц и удаления из выдачи (страница как мешок слов)
- Применяют в биологии для сравнения геномов

# Locality-Sensitive Hashing (LSH)

- Отбор самых похожих множеств (много больших) на заданное
- Если множеств много, то медленно сравнивать даже MinHash сигнатуры
- Ускорим сравнение MinHash сигнатур!
- Идея: придумаем хэш, который в одну корзинку будет складывать похожие

# LSH для MinHash сигнатур



Figure 3.6: Dividing a signature matrix into four bands of three rows per band

Заменим сравнение в полосе на совпадение хэшей **кусочков сигнатур**  
(пусть корзинок много, коллизии маловероятны)!

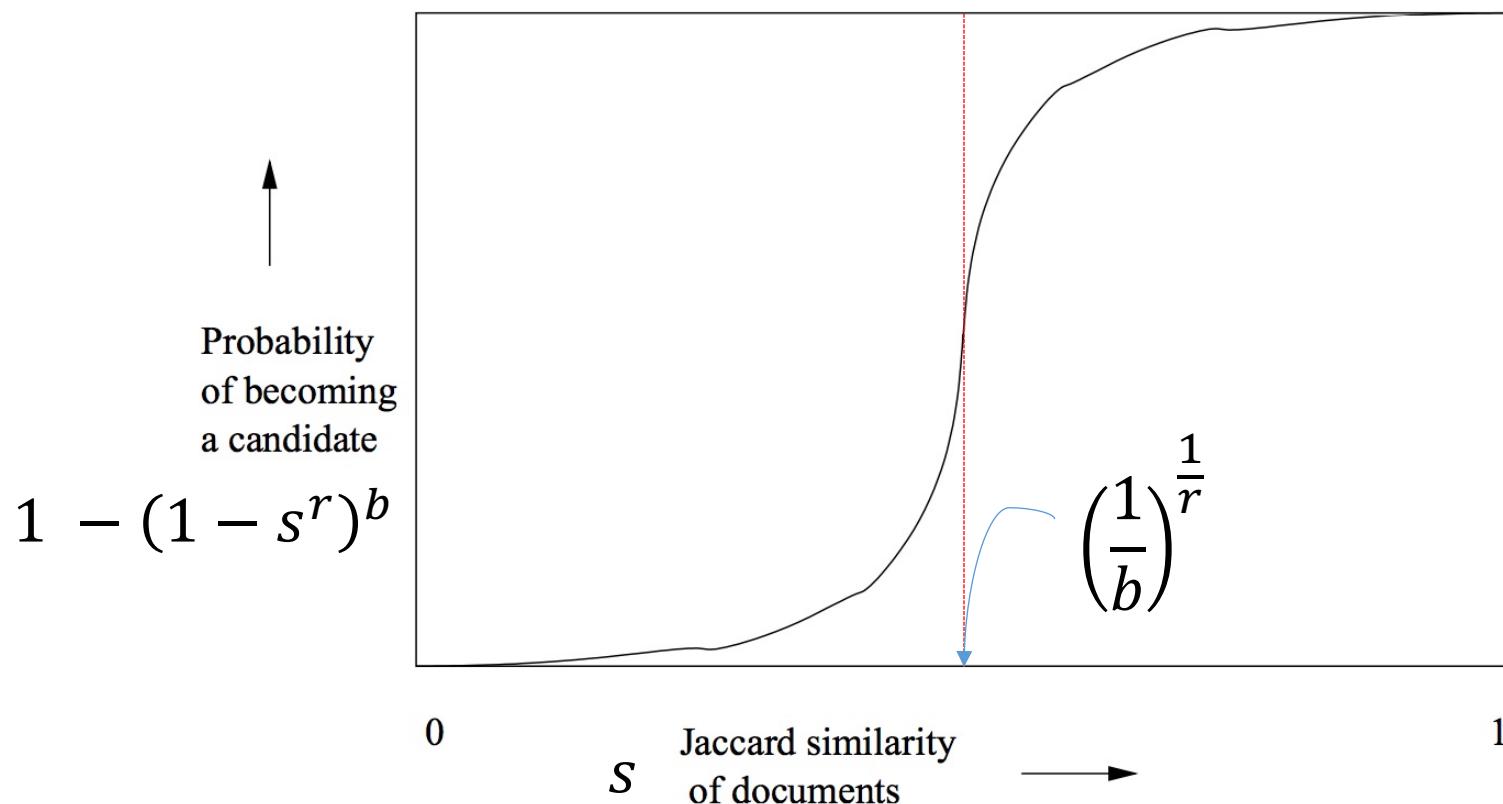
# LSH для MinHash сигнатур

- Пусть два множества имеют  $J(S_1, S_2) = s$
- Оценим вероятность того, что хэши **хотя бы в одной** полосе совпадут:
  1. The probability that the signatures agree in all rows of one particular band is  $s^r$ .
  2. The probability that the signatures disagree in at least one row of a particular band is  $1 - s^r$ .
  3. The probability that the signatures disagree in at least one row of each of the bands is  $(1 - s^r)^b$ .
  4. The probability that the signatures agree in all the rows of at least one band, and therefore become a candidate pair, is  $1 - (1 - s^r)^b$ .

Кандидатов будем проверять уже более тщательно!

# LSH для MinHash сигнатур

- Вероятность стать кандидатом на *медленную* проверку похожести:



$$b = 20 \quad r = 5$$

$s$	$1 - (1 - s^r)^b$
.2	.006
.3	.047
.4	.186
.5	.470
.6	.802
.7	.975
.8	.9996

# LSH

- Существуют реализации для других мер:
  - Евклидова
  - Косинусная
  - ...
- Позволяет быстро сузить круг похожих множеств для медленной проверки

# LSH: примеры применения

- Google News (для поиска похожих сюжетов)
- Поиск похожих картинок (по нейросетевым сигнатурам)
- Audio similarity identification

# Хэширование полезно

- Уже знаем:
  - Хэширование признаков (Vowpal Wabbit)
  - Определение принадлежности множеству (Bloom Filter)
  - Оценка частот элементов множества в потоке (Count-min sketch)
  - Оценка похожести двух множеств (MinHash)
  - Отбор самых похожих множеств на заданное (LSH)

На этом все

Спасибо за внимание!