Big Data Algorithms

Selected topics

JACEK CICHOŃ

WPPT • 2019

Fundusze Europejskie
Wiedza Edukacja Rozwój

Politechnika Wrocławska

Unia Europejska
Europejski Fundusz Społeczny

Contents

Initial remarks

In 2004, the first book was published containing a relatively complete overview of the basic techniques used in the IT department called Big Data. It was written by Anand Rajaraman, Jure Leskovec and Jeffrey D. Ullman and was entitled *Mining Massive Datasets* ([4]). This book was the basic material for two Big Data courses for IT students at the Faculty of Fundamental Problems of Technology at the Wrocław University of Science and Technology.

During the lectures it turned out that the material contained there can not be implemented in the one - semester course. Certain issues, mainly related to the problem of on-line algorithms were omitted in these lectures - they are offered to students on other courses.

It also turned out that some of the algorithms discussed in this book require clarification. All the necessary scientific articles could easily be found online. The considerations presented in this document contain clarifications, simplifications and additions to a number of reasoning from the basic book, scientific articles. So they are an auxiliary material for the Big Data course.

Chapter 1

Introduction

Torture the data, and it will
confess to anything.

Ronald Coase, economist,
Nobel Prize Laureate

1.1 The Bonferroni Principle

There is a concept in statistics that goes like this: even in completely random datasets, you can expect particular events of interest to occur, and to occur in increasing numbers as the amount of data grows. These occurrences are nothing more than collections of random features that appear to be instances of interest, but are not. This bears repeating: even amounts of random data lead to what seem to be events of interest, and the number of these seemingly interesting events grows as does the size of the dataset.

Ullman's example

Let us assume that we want to check how many pairs of people from a large population meet several times in the same hotel for a selected period of time. This information may be useful for intelligence services to detect attempts to conduct illegal activities. Let's set several parameters:

- N = size of observed population ($N = 10^9$)

- p = probability that a a randomly chosen object spends time in a hotel in a given day ($p = 10^{-2}$)
- M = average number of rooms in a hotel ($M = 10^2$)
- H = number of hotels
- L = number of days of observation ($L = 10^3$)

In a given day approximately $N \cdot p$ spend time in a hotel. The average number of rooms in hotels is $H \cdot M$. Zatem $N \cdot p \approx H \cdot M$. Hence $H \approx \frac{Np}{M}$. We shall assume that the equality holds.

Let us denote by c the expected number of days the number of days that the selected person spends in the hotel during the observation period and appoximate it to a natural number. We have $c = L \cdot p$ (in our example we have $c = 10$)

Timetable: a function from a subset o cardinality c of the set $\{1, \ldots, L\}$ into the set $\{1, \ldots, H\}$. Then the number of timetables is $\binom{L}{c} H^c$ and the number of timetable pairsis $(\binom{L}{c} H^c)^2$.

The number of timetable pairs with two or more common localization is $\binom{L}{2} H^2 \left(\binom{L-2}{c-2} H^{c-2}\right)^2$. Hence the number of 2-collisions:

$$\Pr[2 - \text{collision } \{a, b\}] = \frac{\binom{L}{2} H^2 \left(\binom{L-2}{c-2} H^{c-2}\right)^2}{(\binom{L}{c} H^c)^2} =$$

$$\frac{\binom{L}{2} \left(\binom{L-2}{c-2}\right)^2}{(\binom{L}{c})^2} \cdot \frac{1}{H^2} = \frac{\binom{L}{2} \left(\binom{L-2}{c-2}\right)^2}{(\frac{L(L-1)}{c(c-1)} \binom{L-2}{c-2})^2} \cdot \frac{1}{H^2} =$$

$$\frac{1}{2} \frac{(c(c-1))^2}{L(L-1)} \frac{1}{H^2} \approx \frac{1}{2} \left(\frac{c^2}{LH}\right)^2$$

After substituting the numerical data we get

$$\Pr[2 - \text{collision}\{a, b\}] = \frac{1}{2} (\frac{10}{10^2 10^5})^2 = \frac{1}{2} 10^{-12} .$$

It is a very small number. It is its interpretation: this is the probability of the event that two given persons will meet two times in the same hotel.

Let us calculate the expected number of 2-colliding pairs L_2:

$$E[L_2] = \binom{N}{2} \Pr[2 - \text{collisions}\{a, b\}] \approx \frac{1}{4} N^2 \left(\frac{c^2}{LH} \right)^2$$

In our example we have $E[L_2] \approx \frac{1}{4} 10^{18} 10^{-12} = \frac{1}{4} 10^6 = 250000$. This is the number of random meetings. It comes to them as a result of a purely random mechanism. These calculations show that the task of catching suspicious meetings is practically impossible to realize. The number of purely random meetings is so large that a more accurate analysis of the reasons for these meetings was very expensive.

These calculations reveal a typical phenomenon for Big Data: if we have very large sets of data, there are many random dependencies, which should be interpreted as pure cases and do not testify to the existence of hidden dependencies or rules.

1.2 TF-IDF

TF-IDF stands for term frequency-inverse document frequency, and the tf-idf weight is a weight often used in information retrieval and text mining. This weight is a statistical measure used to evaluate how important a word is to a document in a collection or corpus. The importance increases proportionally to the number of times a word appears in the document but is offset by the frequency of the word in the corpus. Variations of the tf-idf weighting scheme are often used by search engines as a central tool in scoring and ranking a document's relevance given a user query.

Definition 1. *Let $D_1, \ldots, D_N$ be a collection of documents. Let $\{w_1, \ldots, w_m\}$ be the set of words in these documents. Let f_{ij} be the number of ocurrences of word w_i in the document D_j. The number*

$$TF_{ij} = \frac{f_{ij}}{\max_{k=1,\ldots,n}(f_{kj})}$$

term frequency of the word w_i in the document D_j.

The code in Scala for the pre-treatment of texts is shown below.

```
def fileToStr(path:String): String = {
    val bufor  = io.Source.fromFile(path,"UTF-8")
    val tekst  = bufor.mkString
      .toLowerCase
      .replaceAll("[,.!:?*;()]","")
      .replaceAll("\\n\\t", " ")
      .replaceAll("\\s+"," ")
    bufor.close
    tekst
  }
```

A typical example for Big Data program, the equivalent of the "Hello World" program in typical programming lectures, is a program that loads a collection of documents, breaks them into words and calculates for them indexes TF-IDF. A very usefull resource of data for such analysis are lists of so called stop-words (i.e. words which are filtered out before or after processing of natural language text)can be found at page
https://sites.google.com/site/kevinbouge/stopwords-lists

Chapter 2

Similarity of objects

Finding similar objects in large data-sets is an important database operation. The operation is used in applications like plagiarism detection, finding mirror or similar pages. Is often a first step in cluster analysis, which goal is to group observed events into subgrops having some common features. We start with the definition of metric space.

Definition 2. *A pair (X, d) is a metric space if x is an nonempty set and $d : X \times X \to [0, \infty)$ is a function such that*

1. $(\forall x, x \in X)(d(x, y) = 0 \leftrightarrow x = y)$
2. $(\forall x, y \in X)(d(x, y) = d(y, x))$ *(symmetry)*
3. $(\forall x, y, z \in X)(d(x, z) \leq d(x, y) + d(y, z))$ *(triangle inequality)*

The function d is called a metric of the space X.

A basic example of a metric space is the euclidean space $(\mathbf{R}^n, d_e)$ with the euclidean metric d_e defined by formula

$$d_e(\vec{x}, \vec{y}) = \sqrt{\sum_{i=1}^{n} (x_i - y_i)^2}\ ,$$

where $\vec{x} = (x_1, \ldots, x_n)$ and $\vec{y} = (y_1, \ldots, y_n)$. The proof of the first two properties of metric for this function is trivial. The proof of the triangle inequality maybe based on the classical Cauchy inequality.

A generalization of euclidean metric space are so-called l_p spaces for $p \in [0, \infty)$. They consists of pair $(\mathbf{R}^n, d_p)$ where d_p is defined by the formula

$$d_p(\vec{x}, \vec{y}) = \left(\sum_{i=1}^{n} |x_i - y_i|^p \right)^{\frac{1}{p}} .$$

The space $(\mathbf{R}^n, d_p)$ is denoted by l_p^n. This time, to show that d_p satisfies the triangle inequality we need to use a stronger result than Cauchy inequality - we need the Holder inequality.

Notice that d_2 is the euclidean distance, so l_p spaces are generalizations of euclidean spaces. Quite interesting and often used metric is the limit case of d_p as $p \to \infty$. Namely, this operation gives us a metric defined by formula

$$d_\infty^n = \max\{|x_i - y_i| : i = 1, \ldots, n\} .$$

Very often in Big Data applications it is necessary to use euclidean spaces with very high dimension. We shall discuss the problem of reasonable reduction of dimension in a special chapter of this notes.

Notice that the euclidean metric is unbounded, i.e. $\sup\{d_e(x, y) : x, y \in \mathbf{R}^n\} = \infty$. IIn some applications, it causes great trouble. However, there are a several techniques of transformation of metrics. We start with a relatively simple, but vary powerful method.

Theorem 1. *Let $f : [0, \infty) \to [0, \infty)$ be a monotonic and concave function such that $f(0) = 0$. Let (X, d) be an arbitrary metric space. Then the function $\rho(x, y) = f(d(x, y))$ is also a metric on X.*

Proof. Let us fix a monotonic and concave function $f : [0, \infty) \to [0, \infty)$. It is left to the reader as an exercise to show that for any $a, b \geq 0$ we have $f(a + b) \leq f(a) + f(b)$ (hint: we may assume that że $a + b > 0$; notice next that $a = (a + b)\frac{a}{a+b}$ and $b = (a + b)\frac{b}{a+b}$ and try to use Jensen inequality for concave functions). Notice that we need only to show the triangle inequality for that function ρ. So, let us fix $a, b, c \in X$. The we have

$$\rho(a, c) = f(d(a, c)) \leq f(d(a, b) + d(b, c)) \leq$$
$$f(d(a, b)) + f(d(b, c)) = \rho(a, b) + \rho(b, c) ,$$

so the theorem is proved. □

Corollary 1. *Suppose that $\epsilon \in (0,1)$ and that d is a metric on X.Then the function $\rho(x,y) = d(x,y)^\epsilon$ is also a metric on X.*

Proof. Let $0 < \epsilon < 1$ and $f(x) = x^\epsilon$. Then $f'(x) = \epsilon x^{\epsilon-1}$ and $f''(x) = \epsilon(\epsilon-1)x^{\epsilon-2}$, so $f'(x) > 0$ and $f''(x) < 0$ for $x > 0$, so we may apply the previous theorem. □

The next corollary is often used for transformation of a given standard metric (such as euclidean metric) into a bounded metric.

Corollary 2. *Suppose that d is a metric on set X. Then the function*

$$\rho(x,y) = \frac{d(x,y)}{1+d(x,y)}$$

is also a metric on X.

Proof. Let $f(x) = \frac{x}{1+x}$. Then $f'(x) = \frac{1}{(x+1)^2}$ and $f''(x) = -\frac{2}{(x+1)^3}$, so $f'(x) > 0$ and $f''(x) < 0$ for $x > 0$, so we may apply the previous theorem. □

Notice that the metric defined in last corollary is bounded by one. Moreover, from strictly mathematical point of view, this metric has a one interesting property: namely, suppose that (X,d) is a metric space and that $\rho(x,y) = d(x,y)/(1+d(x,y)$. Consider any sequence $(x_n)_{n\in\mathbf{N}}$ of elements of X. Then the following two sentences are equivalent:

1. sequence $(x_n)_{n\in\mathbf{N}}$ is convergent in metric d
2. sequence $(x_n)_{n\in\mathbf{N}}$ is convergent in metric ρ

From this easy observation we conclude that both metrics d and ρ generates the same topology, i.e. that the notions of open and closed sets for ths metrics coincides.

2.1 Jaccard similarity

Jaccard similarity is a simple method, developed to compare regional floras in alpiane zone in 1912 by P. Jaccard, is nowadays a popular method for measuring similarities between finite sets. Let us start with the definition.

Definition 3. *Jaccard similarity of two sets A and B is the number*

$$J(A,B) = \frac{|A \cap B|}{|A \cup B|}$$

If $A = B = \emptyset$ then we put $J(A,B) = 1$. Jaccard distance between sets A i B is the number

$$d_J(A,B) = 1 - J(A,B)\ .$$

Let us observe that $0 \leq J(A,B) \leq 1$ and that

$$d_J(A,B) = 1 - \frac{|A \cap B|}{|A \cup B} = \frac{|A \cup B| - |A \cap B|}{|A \cup B|} = \frac{|A \triangle B|}{|A \cup B|}$$

where $A \triangle B = (A \cap B^c) \cup (A^c \cap B)$ is the symmetric difference of sets A and B. Jaccard distance is a metric on family of finite subsets of a fixed universe.

Theorem 2. *Let Ω be a fixed nonempty set. Then the Jaccard distance d_J is a metric on the set $P_{fin}(\Omega) = \{X \subset \Omega : |X| < \infty\}$.*

We shall begin with a proof of additional result about modification of metrics.

Lemma 1 (Sheinhaus)**.** *Suppose that (X,d) is a metric space and let $a \in X$. Let*

$$\rho(x,y) = \frac{2d(x,y)}{d(x,a) + d(y,a) + d(x,y)}\ .$$

Then ρ is a metric on X.

Proof. Let us observe that if $0 < p \leq q$ and $r \geq 0$ to $\frac{p}{q} \leq \frac{p+r}{q+r}$.

Let $p = d(x,y)$, $q = d(x,y) + d(x,a) + d(y,a)$ and $r = d(x,z) + d(y,z) - d(x,y)$. To finish the proof it is sufficient to substitute this numbers into formula for $\rho(x,z)$ and use the above inequality to prove the trangle inequality for ρ. □

From Steinhauss Lemma we easily deduce that the function $\rho(A,B) = 1 - J(A,B)$ is a metric on the family of all non-empty subsets of a given finite set (put $a = \emptyset$ in this Lemma and as the original metric d take the Hamming distance $d(A,B) = |A \triangle B|$). It is called the Jaccard distance. This is a very important observation, because all notions of similarity which a constructed by transformation $s(x,y) = 1 - d(x,y)$ from metrics have useful properties.

The Jaccard similarity between two sets can be generalized into multi-sets. This is its natural generalization:

Definition 4. *Jaccarda similarity between two multi-sets A i B is the number*

$$J(A,B) = \frac{\sum_i \min\{a_i, b_i\}}{\sum_i \max\{a_i, b_i\}}$$

2.2 Shingles

One of the classical way of transformation of any text into objects which are easy to compare is based on the notion of shingles.

Definition 5. *k - shinge of a sequence $a_1 \dots a_n$ is an arbitrary block of the form $a_l \dots a_{l+k-1}$, where $1 \le l \le n-k+1$.*

Notice that there are approximately n shingles of length k, so the list of all shingles occupies $m \cdot k$ of bytes of memory. So, this lists is not a compact representation of the original sequence. But, in fack we are interested in the set of all k - shingles of a given string, i.e. our goal is to consider the set

$$S_k(A) = \{[A[i..i+k-1] : 1 \le i \le \mathit{length}(A) - k + 1\}$$

Notice that set do no contain repeated elements, so the size of the set $S_k(A)$ can be much smaller than the original string A. The following very compact code written in the language Scala transform any string into the set of its shingles:

```
def toShingles(S: String, k:Int) : Set[String] = {
  S.sliding(k).toSet
}
```

The classical way of dealing with large collections of texts is to transform them to sets of shingles and then to apply Jaccard similarity to cluster them into similarity classes.

2.3 Min-hash

MinHash is a technique for approximating the Jaccard Similarity between two different sets. The scheme was invented by Andrei Broder in 1997. The simplest version of the minhash scheme uses k different hash functions with values in some bounded set of integers, where k is some fixed integer parameter, and represents each set A by the k values of $hmin(A)$ for these k functions.

The estimate of Jaccard Similarity $J(A,B)$ using this version is calculated as follows: we calculate the number y of hash functions for which $hmin(A) = hmin(B)$, and use $\frac{y}{k}$ as the estimate. This estimate is the average of k different $0-1$ random variables, each of which is one when $hmin(A) = hmin(B)$ and zero otherwise, and each of which is an unbiased estimator of $J(A,B)$.

Permutation similarity

In order to understand the correctness of the MinHas scheme presented above, we will begin with an idealized version consisting in selecting random permutations.

Let us fix a number n and let $A \in \{0,1\}^n$. For any permutation π of the set $[n] = \{1,\dots,n\}$we put

$$h_\pi(A) = \min\{k : A(\pi(k)) = 1\} \ .$$

Theorem 3. *Let us consider the probability space on the set S_n of all permutations of the set $[n]$ equipped with the uniform probability (i.e.* $\Pr[\pi = \frac{1}{n!}$ *for each $\pi \in S_n$. Then*

$$\Pr_\pi[h_\pi(A) = h_\pi(B)] = J(A,B)$$

Here, $\Pr_\pi$ denotes the probability in the space of all permutations with uniform distribution.

Proof. Niech $x = |\{i : A[i] = B[i] = 1\}|$, $y = |\{i : A[i] \neq B[i]\}|$ i $z = |\{i : A[i] = B[i] = 0\}|$. Then $x+y+z = n$ oraz $J(A,B) = \frac{x}{x+y}$. It is sufficient now to calculate the probability of the set

$$K = \{\pi \in S_n : \min\{k : \pi(k) \in A \cup B\} \in A \cap B\}$$

This can be done as follows: suppose that we are observing the process of generation a random permutation choosing one element at each time. Let k be the first moment when we choose some element from the set $A \cup B$. Then independently on k the probability that we choose en element from $A \cap B$ is $\frac{x}{x+y}$. Summing over all possible choices of $k \in [1, \ldots, n-(x+y)]$ we get the desired result. $\square$

Quick min-hashing algorithm

Solutions described in previous section work correctly, but they are absolutely too slow - he generation of random permutation is a difficult process. We need to construct the full matrix, and we need to permute it k times. A faster way is proposed by the Min-Hash algorithm.

And here's the main fantastic trick. Let's assume that we have an ideal hash function $h : \Sigma^* \to \mathbf{R}$ for our disposal. Then the sequence of values $(h(a) : a \in A)$ from the order point of view we obtain a random ordering of the set $\{h(a) : a \in A\}$. So we may try to use this observation for generation of random permutations. However there are two problems. The first one is a quality of real hash functions. It occurs that this is not a serious problem - such hash functions like MurmurHash (a non-cryptographic hash function suitable for general hash-based lookup, created by Austin Appleby in 2008) behaves wery well in all practical situations. The second problem is connected with granuality - real hasa never produce infinite sequences, so we must deal will collisions. But here comes the Birthday Paradox with help. Namely, we know that if we randomly throw k balls into n urns, where $k < \sqrt{n}$ then the probability of a collision is very small. So this is the idea. And now more concrete its realization.

Let $N = |E|$. We fix k random hashing mappings $\{h_1, h_2, \ldots, h_k\}$ such that $h_j : E \to [N]$. We initialize k counters $\{c_1, c_2, \ldots, c_k\}$ by setting $c_i = \infty$. The pseudo-code is given in listing Algorithm 2.3.

This solution works well. Namely, the following , almost obvious after the above discussion, theorem holds:

Theorem 4. *Let $h : \Omega \to \{0, \ldots, L\}$ be a random hash function. For $A \subseteq \Omega$ such that, że $|A| \leq \sqrt{L}$ we put*

$$h^*(A) = \min\{h(a) : a \in A\}$$

Algorithm 1 Min Hash on the set S

```
for $i = 1$ to N do
    if $S(i) = 1$ then
        for j=1 to k do
            if $h_j(i) < c_j$ then
                $c_j \leftarrow h_j(i)$
            end if
        end for
    end if
end for
```

Then

$$Pr_h[h^*(A) = h^*(B)] \sim \frac{|A \cap B|}{|A \cup B|}(= s(A, B)) \ .$$

Chapter 3

Streaming

Perfect is the enemy of good.

Voltair

3.1 Bloom filters

A Bloom filter is a space-efficient probabilistic data structure, that is used to test whether an element is a member of a set. They were discovered by Burton Howard Bloom in 1970 ([1]). False positive matches are possible, but false negatives are not – in other words, a query returns either "possibly in set" or "definitely not in set". In the basic version of this filters elements can be added to the set, but not removed. A specific property of this filters is that the more elements that are added to the set, the larger the probability of false positives.

The main purpose of Bloom filters is to build a space-efficient data structure for set membership. Indeed, to maximize space efficiency, correctness is sacrifized: if a given key is not in the set, then a Bloom filter may give the wrong answer (this is called a false positive), but the probability of such a wrong answer can be made small.

A typical application of Bloom filters is web caching. An ISP may keep several levels of carefully located caches to speed up the loading of commonly viewed web pages, in particular for large data objects, such as images and videos. If a client requests a particular URL, then the service needs to determine quickly if the requested page is in one of its caches.

False positives, while undesirable, are acceptable: if it turns out that a page thought to be in a cache is not there, it will be loaded from its native URL, and the "penalty" is not much worse than not having the cache in the first place.

Informal description

We want to represent n-element sets $S = \{s_1, \ldots, s_n\}$ from a very large universe U, with $|U| = u \gg n$. We may think of U as the set of URLs, n as the cache size, and S as the URLs of those web pages that are currently in the cache. We want to support insertions and membership queries ("Given $x \in U$, is $x \in S$?") so that:

1. If the answer is No, then $x \notin S$.
2. If the answer is Yes, then x may or may not be in S, but the probability that $x \notin S$ (false positive) is low.

Both insertions and membership queries should be performed in constant time.

Simple, noneffective solution

Suppose that we have a bit vector $B = [b_0, \ldots, b_{m-1}]$ of m bits and one hash function $h : \Sigma^* \to \{0, \ldots m-1\}]$. Suppose that initially all entries of B are set to 0. Let $D \subseteq \Sigma^*$ and let $n = |D|$. For each $d \in D$ we put $B[h(d)] = 1$.

After processing all elements from D the array B is fulfilled with 0 and 1. We want to use the following procedure

```
check(x) : Bool = { return B[h(x)] == 1;}
```

for checking whether an element $x \in \Sigma^*$ belongs to D. Clearly, if $x \in D$ then `check(x)==True`. But it may happen, that $x \notin D$ and `check(x)==True`. We call this phenomenon a **false-positive** event. We want to estimate the probability of the false-positive event.

First of all let us estimate the number of bits set to 1 in the table B. Let us fix $i \in \{0, \ldots, m-1\}$. Then

$$\Pr[B[i] = 0] = \left(\frac{m-1}{m}\right)^n = \left(1 - \frac{1}{m}\right)^n ,$$

so the expected number of entries in B with value 0 is $m\left(1-\frac{1}{m}\right)^n$. So, the expected number of entries in B with value 1 is $m - m\left(1-\frac{1}{m}\right)^n$. Thus the probability of the event "*for a randomly chosen element from* $\{0,\ldots,m-1\}$ *we have* $B[i]=1$" is $1-\left(1-\frac{1}{m}\right)^n$. Thus

$$\Pr[\text{false-positive}] \approx 1-\left(1-\frac{1}{m}\right)^n = 1-\left(\left(1-\frac{1}{m}\right)^m\right)^{\frac{n}{m}} \approx 1-e^{-\frac{n}{m}} .$$

Our goal is to check when for a given $\epsilon > 0$ we have $\Pr[\text{false-positive}] < \epsilon$. Notice that

$$(1-e^{-\frac{n}{m}} < \epsilon) \equiv m > \frac{n}{\ln\frac{1}{1-\epsilon}} \approx \frac{n}{\epsilon} .$$

Therefore, say, if we would like to reduce the false-positive events to 2% we should use a table B of length at least $m = \frac{n}{0.02} = 50 \cdot n$ of bits. We shall see that if we use Bloom filters, them the number m can be significantly reduced.

Formal description

A Bloom filter (see [5]) is a bit vector $B = [b_0,\ldots,b_{m-1}]$ of m bits, with k independent hash functions $\mathcal{H} = (h_1,\ldots,h_k)$ that map each key in U to the set $R_m = \{0,1,\ldots,m-1\}$. We assume that each hash function h_i maps a uniformly at random chosen key $x \in U$ to each element of R_m with equal probability. Since the hash functions are independent, it follows that the vector $(h_1(x),\ldots,h_k(x))$ is equally likely to be any of the m^k k-tuples of elements from the set R_m. Initially all m bits of B are set to 0.

- Insert x into S: compute $h_1(x),\ldots,h_k(x)$ and set

  $$B[h1(x)] = B[h2(x)] = \ldots = B[hk(x)] = 1$$

- Query if $x \in S$. Compute $h_1(x),\ldots,h_k(x)$. If

  $$B[h_1(x)] = B[h_2(x)] = \ldots = B[h_k(x)] = 1$$

  then answer Yes, else answer No.

The running times of both operations depend only on the number k of hash functions (we will later see how to choose a suitable value for k in order to minimize the probability of false positives). The space requirement of the data structure is m bits, and we will later see that a reasonable value for m is, say, $8 \cdot n$. Note that any non-randomized data structure that represents n-element subsets of U must use $\Omega(n \cdot \log u)$ bits.

Formal analysis

We start with computations of the probability of a false positive. The probability that one hash fails to set a given a bit is $1 - \frac{1}{m}$. Hence, after all n elements of S have been inserted into the Bloom filter, the probability that a specific bit is still 0 is

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

Notice that we uses the assumption that the hash functions are independent and perfectly random (hence our arguments are only estimations).

The probability of a false positive is the probability that a specific set of k bits are 1, which is

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

where $p = e^{-\frac{kn}{m}}$.

This shows that there are three performance metrics for Bloom filters that can be traded off: computation time (corresponds to the number k of hash functions), size (corresponds to the number m of bits), and probability of error, which corresponds to the false positive rate

$$f(n, m, k) = (1-p)^k = \left(1 - e^{-\frac{kn}{m}}\right)^{k} .$$

Suppose now that we are given the ratio $\frac{m}{n}$. Our goal is to optimize the number k of hash functions to minimize the false positive rate f. Note that more hash functions increase the precision but also the number of 1's in the filter, thus making false positives both less and more likely at the same time. We can find the minimum by taking the derivative of f. To simplify the math, we minimize the logarithm of f with respect to k (we can do it because log is a monotonic function). Let

$$g(k) = \ln(f(k)) = k \cdot \ln(1-p) = k \cdot \ln\left(1 - e^{-\frac{kn}{m}}\right) .$$

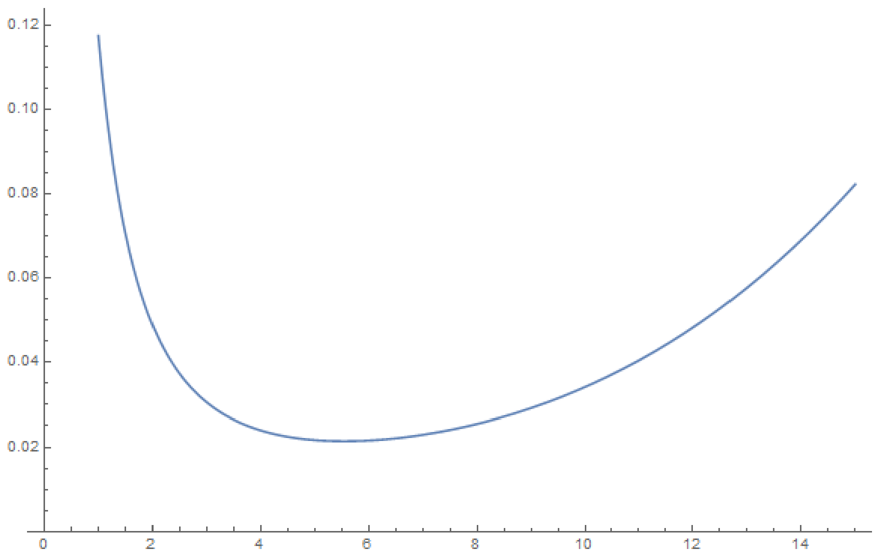

Figure 3.1: Plot of $f(n, 8\cdot n, k)$ for $k \in [1, 15]$

Now we have

$$\frac{\partial g}{\partial k} = \ln\left(1 - e^{-\frac{kn}{m}}\right) + \frac{kne^{-\frac{kn}{m}}}{m\left(1 - e^{-\frac{kn}{m}}\right)} \ .$$

This formula does not look nice. However, it is quite easy to check that for fixed parameters m i n it has exactly one global minimum. What's more, you can easily check that at point $k = \ln(2)\frac{m}{n}$ the derivative is equal to zero. Namely, $k = \ln(2)\frac{m}{n}$ for we have

$$e^{-\frac{kn}{m}} = e^{-k\frac{n}{m}} = e^{-\ln 2} = \frac{1}{2} \ ,$$

so

$$\frac{\partial g}{\partial k}((\ln 2)\frac{m}{n}) = \ln\frac{1}{2} + \ln 2 = 0$$

Therefore at this point, we have a a global minimum.Notice that

$$f(n, m, \ln(2)\frac{m}{n}) = \left(\frac{1}{2}\right)^{\ln(2)\frac{m}{n}} = \left(\left(\frac{1}{2}\right)^{\ln(2)}\right)^{\frac{m}{n}}$$

Therefore for the optimal value of k, the false positive rate is

$$f \approx (0.618503)^{\frac{m}{n}}$$

m/n	false-positive	k
3	23.66 %	2
4	14.63 %	3
5	9.051 %	3
6	5.598 %	4
7	3.463 %	5
8	2.142 %	6
9	1.325 %	6
10	0.819 %	7

Table 3.1: False - positive of Bloom filters for different proportion of parameters m/n and optimal number of hash functions k

Observe that as m grows in proportion to n, the false positive rate decreases. Already $m = 8 \cdot n$ reduces the chance of error to roughly 2%, and $m = 10 \cdot n$ to less than 1%. In Table 3.1

Implementation details

Bloom filters are easy to implement in any reasonable programming language. However, some details should be explained.

The first detail is connected with the choice of hash functions. MurMurHash is the proven choice. It is an example of non-cryptographic hash function and in most languages there are 64 and 128 bits variants of this functions. So we may use k copies of MurMuhHash functions intialized with k independent seeds (initial values).

It is worth to remember that calculations of hash functions is costly. And in some situations this may be a problem. So, let us analyze a specific case more detail. Suppose that we have an upper bound on number of items n. Let $n = 10^6$. Let $m = 8 \cdot 10^6$. So we will need approximately 1 Mb of memory for storing the Bloom filter. Moreover $\log_2(m) \approx 23 \ll 32$. So we can use one 128 bits MurMurHash for generation of four 32 bits hash functions - simply split the sequence of 128 bits into four sequences of 32 bits. So we see that instead of 6 copies of MurMurHash, we need only 2 copies. Hence we obtained 3-times speed up.

3.2 Reservoir Sampling

In this chapter we will discuss the basic method of generating random samples from data streams of any length. We will start our discussion by discussing the method of selecting a random element of the observed data stream in accordance with the uniform distribution. The basic algorithm is surprisingly simple:

Algorithm 2 Uniform sampling

```
procedure INT
    n=0
    sample = nil
    index = nil
end procedure
procedure onGet($x$)
    n++
    if (random() < $\frac{1}{n}$) then
        sample = x
        index = n
    end if
end procedure
```

Let L_n be the random variable pointing on the index selected after nth call of the procedure onGet. Clearly, $1 \leq L_n \leq n$.

Theorem 5. $(\forall n)(\forall 1 \leq i \leq n)(\Pr[L_n = i] = \frac{1}{n})$

Proof. We have $L_1 = 1$. Suppose that this theorem is true for the number n. Let $m = n+1$ The the value of index is changed with probability $1/(m)$, so $\Pr[L_m = m]\frac{1}{m}$. Let us fix a number $i \in \{1, \ldots, n\}$. Then $L_m = i$ if $L_n = 1$ and the index is not changed. Therefore

$$\Pr[L_m = i] = \Pr[L_n = i] \cdot (1 - \frac{1}{m}) = \frac{1}{n}\left(1 - \frac{1}{n+1}\right) = \frac{1}{n+1}\ .$$

□

3.3 The Heavy Hitters Problem

In the heavy hitters problem, the input is an array A of length n, and also some parameter k. We should think of n as very large (in the hundreds of millions, or billions), and k as modest (10, 100, or 1000). The goal is to compute the values that occur in the array at least $\frac{n}{k}$ times.

Note that there can be at most k such values; and there might be none. The problem of computing the majority element corresponds to the heavy hitters problem with $k \approx 2 - \delta$ for a small value $\delta > 0$, and with the additional promise that a majority element exists. The heavy hitters problem has lots of applications. We'll be more specific later when we discuss a concrete solution, but here are some high-level examples

1. Computing popular products. For example, A could be all of the page views of products on https://allegro.pl/ yesterday. The heavy hitters are then the most frequently viewed products.

2. Computing frequent search queries. For example, A could be all of the searches on Google yesterday. The heavy hitters are then searches made most often.

3. Identifying heavy TCP flows. Here, A is a list of data packets passing through a network switch, each annotated with a source-destination pair of IP addresses. The heavy hitters are then the flows that are sending the most traffic. This is useful for, among other things, identifying denial-of-service attacks.

Heavy Hitters in Data Streams

Consider the goal of finding all frequent elements in a data stream. In particular consider finding a majority element i.e., an element x such that $f_x > \frac{n}{2}$, where f_x is a number of times the element x occurs in a data stream, i.e.

$$f_x = |\{i \in [1 \ldots n] : A[i] = x\} \ .$$

The code of algorithm analyzed in this section in in listing Algorithm 3.

Theorem 6. *If $f_x > \frac{n}{2}$ then the output of the Algorithm 3 the output will be x.*

Algorithm 3 Majority Algorithm

```
procedure INT
    c=0
    best = nil
end procedure
procedure ONGET($x$)
    if c==0 then
        best = x
    end if
    if x==best then
        c++
    else
        c–
    end if
      return best
end procedure
```

Proof. Observe that while $c > 0$, the value of j does not change. We can therefore divide the input into segments based on when $c = 0$, and let $j_1; j_2; :::$ be the values of j during those segments. It is also clear that during the $i - th$ segment, the value of j_i appears precisely $\frac{1}{2}$ the time among the inputs. Therefore if $f_j > \frac{n}{2}$, the last segment must end with $c > 0$ and the value must be j. □

What happens if there is no x such that $f_x > \frac{n}{2}$? The value of x might be any input value. It might even occur only once in the input. One could verify this using a second pass over the input.

Misra-Gries Summary

Misra and Gries generalized that majority algorithm above to a more general method that we can think of in this form. They published their algorithm in paper ”Finding repeated elements” in 1982 (see [3]). The code of algorithm analyzed in this section in in listing Algorithm 4. In this code we use associative arrays.

For $x \in A.keys$ we put $\hat{f}_x = A[x]$ and $\hat{f}_x = 0$ otherwise. Let n be an upper bound on the length of observed stream. Note that we need

Algorithm 4 Misra-Gries Algorithm

```
procedure INT
    A=[]
end procedure
procedure onGet($x$)
    if $x \in A$ then
        $A[x]++$
    else
        if A.length < k then
            A[x] = 1
        else
            for x ∈ A.keys do
                A[x]- -
            end for
            for x ∈ A.keys do
                if A[x]==0 then
                    remove x from A
                end if
            end for
        end if
    end if
end procedure
```

$\log_2 n + 1$ bits for binary representation of any number from the range 0..n. For example, if $n = 10^{10}$ then we need 34 bits, i.e 5 bytes is sufficient. Let m denotes an upper bound (in bits) on the observed values.

Theorem 7. *The Misra-Gries algorithm with parameter k uses one pass and $\mathrm{O}\left(k(\log n + m)\right)$ bits of space and provides for any token x an estimate $\hat{f}_x$, satisfying*

$$f_x - \frac{n}{k} \leq \hat{f}_x \leq f_x \ .$$

Proof. Let us pretend that A consists of n key/value pairs, with $A[j] = 0$ whenever j is not actually stored in A by the algorithm. Notice that the counter $A[j]$ is incremented only when we process an occurrence of j in the stream. Thus, $\hat{f}_j \leq f_j$. On the other hand, whenever $A[j]$ is decremented (observe that if $A[j]$ is incremented from 0 to and A store mode than k keys then of then immediately decremented back to 0), we also decrement $k-1$ other counters, corresponding to distinct tokens in the stream. Thus, each decrement of $A[j]$ is "witnessed" by a collection of k distinct tokens (one of which is a j itself) from the stream. Since the stream consists of m tokens, there can be at most m/k such decrements. Therefore, $\hat{f}_j \geq f_j - \frac{m}{k}$. $\square$

Count-Min Sketch

The Count–Min Sketch algorithm was invented in 2003 by Graham Cormode and S. Muthu Muthukrishnan and published in 2005 in paper "An Improved Data Stream Summary: The Count-min Sketch and Its Applications" (see [2]).

The algorithm is parametrized by two natural numbers b and k. We fix a sequence $(h(0,*),\ldots,h(b-1,*))$ of independent hash functions with values in the set $\{0,\ldots,L-1\}$. We reserve a matrix $A[0\ldots b-1][0\ldots L-1]$. We fill this matrix with zeros at the initialization phase.

Here is the surprisingly simple pseudo-code of the algorithm. It consists with two procedures: the first used after receiving new element x from a stream and the second for final counting prequency of elements.

```
procedure add(x){
  for i=0 to b-1 do A[i,h(i,x)]++
}

function count(x){
  return min {A[i,h(i,x)] : i = 0 ...b-1}
}
```

Let x be an element occurring in the considered stream. We denote, as before, by f_x the number of times the element x occurs in stream. Observe that each time when x is observed the procedure add increases all counters $A[i,h(i,x)]$ where $i=0,\ldots,L-1$. Therefore after observing this stream we have $\mathrm{count}(x) \geq f_x$.

We will prove the following theorem which confirms that with a high and controlled probability we have $f_x \leq \mathrm{count}(x) \leq f_x(1+\frac{e}{L})$.

Theorem 8. *If $b = \ln(\frac{1}{\delta})$ and $L = \frac{e}{\epsilon}$ then*

$$\Pr[count(x) \geq f_x + \epsilon n] \leq \delta\ .$$

Proof. Let n denotes the length of the stream. Let us fix index $i \in \{0, \dots, b-1\}$ and an element x from the stream. Let $b = h(i, x)$ Let Z denotes the content of the cell $A[i, b]$ Then

$$Z = f_x + \sum_{y \in C} f_y$$

where $C = \{y \in S : y \neq x \wedge h(i, y) = b\}$. This sum may be written in the following way:

$$Z = f_x + \sum_{y \in S; y \neq x} f_y \|h(i, y) = b\| \ .$$

Therefore

$$\mathrm{E}[Z] = f_x + \sum_{y \in S; y \neq x} \mathrm{E}[f_y \|h(i, y) = b\|] = f_x + \sum_{y \in S; y \neq x} f_y \frac{1}{L} \leq f_x + \frac{n}{L} \ ,$$

so

$$\mathrm{E}[Z - f_x] \leq \frac{n}{L}$$

Let us recall the Markov inequality, which holds for any random variable X with positive values and a number $a > 0$:

$$\Pr[X \geq a] \leq \frac{\mathrm{E}[X]}{a}$$

We apply this inequality for random variable $Z - f_x$ and for $a = \frac{ne}{L}$ and get

$$\Pr[Z - f_x \geq \frac{ne}{L}] \leq \frac{n}{L} \frac{L}{ne} = \frac{1}{e} \ .$$

If we repeat this inequality for each row i, then we get

$$\Pr[\mathrm{count}(x) - f_x \frac{ne}{L}] \leq \left(\frac{1}{e}\right)^b \ .$$

Putting $L = \frac{e}{\epsilon}$ and $b = \ln \frac{1}{\delta}$ we finally get

$$\Pr[\mathrm{count}(x) \geq f_x + \epsilon n] \leq \delta \ ,$$

so the theorem is proved. □

Putting $\epsilon = 10^{-3}$ and $\delta = 10^{-6}$ we get $L = 2719$ and $b = 14$, hence our table A has 38066 entries. We should store in entries of the table A 8 bytes integers, so a memory requirements are of order 300 kB, hence are relatively small. And what is interesting is that this memory is independent on the size of observed data stream.

Proper use of Count-Min-Sketch

Let us fix two parameters:

1. number ϵ determining the precision of heavy hitter detector (typical use $\epsilon = 10^{-3}$; $\epsilon = 10^{-4}$)
2. number δ bounding the probability of error (typical setting: $\delta = 10^{-4}$)

Calculate

1. $b = \lceil \ln(\frac{1}{\delta} \rceil$
2. $L = \lceil \frac{e}{\epsilon} \rceil$

Using parameters b and L use Min-Count-Algorithm with b hash functions with values in the set $\{0, \ldots, L-1\}$ and two dimensional array A of size $[0, \ldots, b-1] \times [0, \ldots, L-1]$.

Chapter 4

Model Map-Reduce

MapReduce is a programming model and an associated implementation for processing and generating large data sets. Users specify a map function that processes a key/value pair to generate a set of intermediate key/value pairs, and a reduce function that merges all intermediate values associated with the same intermediate key. More precisely, Map-Reduce consists of two basic functions:

1. MAPPER - read an input (usually line after line), transform the input into a pair (key,value) and sends this pair to a server h(key), where h is some hash function with values in the set $\{0, ldots, n\}$ where n is a number of servers

2. REDUCER - a function which works on each servers and reduce the list of all obtained pairs of the form (key,val) to one pair (key, val); before the reduction this function groups all items into sub-lists with the same key and the reduction is done within this sublist.

From pure formal point of view Map-Reduce is the composition of two functions: map $\circ$ reduce. Good introduction to implementation details of his methodology may be found in [4].

Example 1. *Suppose that in the memory of MAPPERS we store a large vector* $X = [x_i, \ldots, x_n]$ *Our goal is to multiply a large matrix* A *of size* $n \times n$ *by this vector. We also assume that MAPPER will obtain entries of matrix* A *in the form* (i, j, a_{ij})*. The following code will do the job:*

1. MAPPER $(i, j, a) \rightarrow (i, a \cdot x_i)$

2. REDUCER $(i, L) \to (i, sum(L))$

Let us explain the job done by REDUCER. Namely, assume that $n = 3$ *and suppose that it got after the end of MAPPERS job the following list*

$$[(1, a_{1,3}x_3), (3, a_{3,2}x_2), (1, a_{1,2}x_2), (1, a_{1,1}x_1), (3, a_{3,1}x_1), (3, a_{3,3}x_3)]$$

It starts from grouping elements by key - the key if the element of each pair. So it get

$$\begin{aligned}&[\\ &\quad [(1, a_{1,3}x_3), (1, a_{1,2}x_2), (1, a_{1,1}x_1)],\\ &\quad [(3, a_{3,2}x_2), (3, a_{3,1}x_1), (3, a_{3,3}x_3)]\\ &]\end{aligned}$$

This list has two lists. For each of this sub-lists the function sum is applied, so it transform this list to list

$$[(1, a_{1,3} \cdot x_3 + a_{1,2} \cdot x_2 + a_{1,1} \cdot x_1), (3, a_{3,2} \cdot x_2 + a_{3,1} \cdot x_1 + a_{3,1} \cdot x_1)]$$

So we see that it calculated entries y_1 *and* y_3 *of the vector* $\vec{y} = A \cdot \vec{x}$*. The remaining entry* 2 *was calculated by other mapper. The final result of the work of reducer is a file containing two lines:*

```
1, y1
3, y3
```

Usually it is a plain text file. So we finally obtain one file from each reducer.

4.1 Basic set - theoretic operations

We shall see that basic set theoretic operations can be performed on large sets in the model map reduce. Let us assume that we analyze two set A and B. Input will be organized as follows: elements x from A will be marked as `('a',x)` and elements y from B as `('b',y)`

Sum

The following functions calculate sum $A \cup B$:

1. MAPPER $(a, x) \to (x, 1)$; MAPPER $(b, x) \to (x, 2)$;
2. REDUCER (a, L) : return a

Intersection

The following functions calculate intersection $A \cap B$:

1. MAPPER $(a, x) \to (x, 1)$; MAPPER $(b, x) \to (x, 2)$;
2. REDUCER (a, L) : if $(|L| = 2)$ then a else nop

Difference

The following functions calculate difference $A \setminus B$:

1. MAPPER $(a, x) \to (x, 1)$; MAPPER $(b, x) \to (x, 2)$;
2. REDUCER (a, L) : if $(L = [1])$ then a else nop

4.2 Data base operations

The join $R(a, b) \bowtie_b S(b, c)$ can be implemented as follows:

1. MAPPER $((a, b) \in R) \to (b, (1, a))$, MAPPER $((b, c) \in S) \to (b, (2, c))$,
2. REDUCER (b, L): sort L according first coordinate, transform $L$to the form

   $$[(1, a_1), (1, a_2), \ldots, (1, a_n)] \quad || \quad [(2, c_1), (2, c_2), \ldots, (2, c_m)]$$

   and generates $n \cdot m$ pairs

   $$\{(a_i, c_j) : 1 \leq i \leq n \wedge 1 \leq j \leq m\}$$

Another data-base operations operations such as projections or mappings can be implemented in a truvioal way.

4.3 Combiners

A Combiner, also known as a semi-reducer, is an optional class that operates by accepting the inputs from the Map class and thereafter passing the output key-value pairs to the Reducer class.

The main function of a Combiner is to summarize the map output records with the same key. The output (key-value collection) of the combiner will be sent over the network to the actual Reducer task as input.

It can be implemented if the operation θ applied to lists is associative and commutative. For example, the average value is does not have his properties (it is not associative). But can be implement be the operation

$$(n, s) \oplus (m, t) = \left(n + m, \frac{ns + mt}{n + m}\right)$$

4.4 Matrix multiplication

Suppose that we want to multiply two larde matrices $A = (a_{ij})$ and $B = (b_{jk})$.

Method I

1. MAPPER $((A, i, j, a_{ij})) \to (j, (1, i, a_{ij}))$, MAPPER $((B, j, k, b_{jk})) \to (j, (2, k, b_{jk}))$
2. REDUCER $((j, L))$ transform it to list $(i, j, k, a_{ij}b_{jk}) : i, k = 1, \ldots, n$

This method requires the re-use of MapReduce.

Method II

1. $MAP(A, i, j, a_{ij}) \to ((i, k), (1, j, a_{ij}), k = 1 \ldots, n$ and MAPPER $((B, j, k, b_{jk})) \to ((i, k), (2, j, b_{jk}), i = 1 \ldots, n$
2. REDUCER $(((i, k), L)) \to (i, j, \sum L)$

4.5 Grouping similar objects

Our input consist of pairs (i, A_i), where A_i is an object; we fix some number g; we simulate distribution into $\{0, \ldots, g-1\}^2$ groups; we kate an additional hash function $h : \mathbf{Z} \to \{0, \ldots, g - 1\}$ and

1. $MAPi, A_i \to \{(\{h(i), v\}, (i, A_i)) : v \in \{0, \ldots, g - 1\}, v \neq h(v)\}$
2. implementation of REDUCER () depends on contex.

Bibliography

[1] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422–426, July 1970.

[2] Graham Cormode and S. Muthukrishnan. An improved data stream summary: The count-min sketch and its applications. *J. Algorithms*, 55(1):58–75, April 2005.

[3] J. Misra and David Gries. Finding repeated elements. *Science of Computer Programming*, 2(2):143 – 152, 1982.

[4] Anand Rajaraman, Jure Leskovec, and Jeffrey D. Ullman. *Mining Massive Datasets*. 2014.

[5] S. Tarkoma, C. E. Rothenberg, and E. Lagerspetz. Theory and practice of bloom filters for distributed systems. *IEEE Communications Surveys Tutorials*, 14(1):131–155, 2012.

Index