Three Big Data Tools for a Data Scientist's Toolbox

Toon Calders[1,2](✉)

[1] Université Libre de Bruxelles, Brussels, Belgium
[2] Universiteit Antwerpen, Antwerp, Belgium
toon.calders@uantwerpen.be

Abstract. Sometimes data is generated unboundedly and at such a fast pace that it is no longer possible to store the complete data in a database. The development of techniques for handling and processing such streams of data is very challenging as the streaming context imposes severe constraints on the computation: we are often not able to store the whole data stream and making multiple passes over the data is no longer possible. As the stream is never finished we need to be able to continuously provide, upon request, up-to-date answers to analysis queries. Even problems that are highly trivial in an off-line context, such as: "How many different items are there in my database?" become very hard in a streaming context. Nevertheless, in the past decades several clever algorithms were developed to deal with streaming data. This paper covers several of these indispensable tools that should be present in every big data scientists' toolbox, including approximate frequency counting of frequent items, cardinality estimation of very large sets, and fast nearest neighbor search in huge data collections.

1 Introduction

Many data sources produce data as a never-ending stream of records. Examples include sensor networks, logs of user activities on the web, or credit card transactions. Processing these data becomes a challenge, because often there is no storage space or time to store the data for an in-depth off-line analysis. Imagine for instance a credit card fraud detection system that requires that transactions are collected over time and stored on disk for analysis later on. In such a scenario the delay between a credit card fraud and the actual detection of this fraud would be unacceptable. In this case not only the application of fraud prediction methods needs to be online and on the fly, but also the collection of several statistics and modeling parameters needs to be immediate to be able to keep the model up-to-date. Indeed, an important factor in fraud detection is learning what is the normal behavior of a person. This behavior may be changing over time, necessitating flexible and dynamic modelling of what constitutes normal behavior.

We call this type of dynamic processing of data, *stream processing* [4]. We distinguish three different types of stream processing. In the literature these terms are often lumped together while in fact their requirements are quite different.

© Springer International Publishing AG, part of Springer Nature 2018
E. Zimányi (Ed.): eBISS 2017, LNBIP 324, pp. 112–133, 2018.
https://doi.org/10.1007/978-3-319-96655-7_5

1. Online Stream Processing: the distribution of the stream is changing over time and we need to have, at any point in time, an up-to-date model of the current situation. An examples of this challenging processing type is monitoring web traffic for intrusion detection, where the intrusion patterns may change over time. More recent data is more important, and data loses its importance over time. For algorithms under this computational model it is very important that they scale very well with data size as in theory the streams could go on forever. Memory bounds that are logarithmic in the number of instances seen over the stream sofar are considered reasonable. Furthermore, it is important that the algorithms require processing time which is independent from the number of instances already seen as otherwise the streaming algorithms would become increasingly slower. A popular technique to deal with online stream processing is using the window-based technique which considers a conceptual window of the most recent instances in the stream only. Continuously new instances enter the window while old, outdated instances leave the window. A window-based algorithm then continuously and incrementally maintains a summary of the contents of the window that allows to quickly answer analytical queries over the data.
2. Batch Processing: new data are processed in batches. This is for instance the case when new documents arrive that need to be indexed in an information retrieval context, or predictive models need to be updated. Often it is sufficient if the new data are processed continuously, but not necessarily immediately. This setting is far less challenging than the online stream processing model and is hence preferable if the application allows. Algorithms in this category are often incremental in the sense that they are able to incrementally update an existing model with a new batch of data.
3. One-pass algorithms: sometimes datasets to be processed are extremely large and disk-based. Given the relative efficiency of sequential data processing for secondary memory as compared to random access, algorithms that can process the data in one scan are preferable. Such algorithms are often termed streaming as well, since data is streamed from disk into the algorithm for processing. The requirements, however, are different from those of online or batch stream processing as there is not necessarily a temporal aspect in the data; there is no notion of more important recent tuples nor online results that need to be maintained.

It is important to note that distributed computing facilities such as offered by Hadoop [2], Spark [3], Flink [1], can only be part of the answer to the need expressed by these three categories of stream processing. First of all, distributed computing does not address the online aspect of the stream mining algorithms, although it may actually help to increase the throughput. For most batch processing algorithms it is conceivable that multiple batches could be treated in parallel, yet this would introduce an additional delay: handling n batches in parallel implies that batch 1 is still being processed while batch n is fully received, realistically putting a limitation on the scaling factors achievable. And last but not least, distributing computations over 1000 data processors can make processing

at most 1000 times faster, and usually because of communication overhead the speedup is far less. In contrast, here in this paper we will exhibit several methods that achieve exponential performance gains with respect to memory consumption, albeit at the cost of having approximate results only.

Most streaming algorithms do not provide exact results as exact results often imply unrealistic lower complexity bounds. For many applications approximations are acceptable, although guarantees on the quality are required. Approximate results without guarantee should not be trusted any more than a gambler's "educated guess" or a manager's "gut feeling". Guarantees can come in many different forms; a method that finds items exceeding a minimal popularity threshold may guarantee that no popular items are missed, although maybe some items not meeting the threshold may be returned, or a method counting frequencies of events may have a guarantee on the maximal relative or absolute error on the reported frequency. A popular generalization of these guarantees are the so-called ϵ, δ-guarantees. An approximation algorithm A for a quantity q provides an ϵ, δ-guarantee if in at most $1-\delta$ of the cases, the quantity $A(D)$ computed by the algorithm for a dataset D differs at most ϵ from the true quantity $q(D)$; i.e., $P[|A(D) - q(D)| > \epsilon] < 1 - \delta$. Notice incidentally that this guarantee requires some notion of probability over all possible datasets and hence always has to come with an assumption regarding the distribution over possible datasets, such as a uniform prior over all possible datasets.

In this paper we will see three different building blocks that were, arguable subjectively, selected on the basis that at some point in the author's scientific career they proved to be an indispensable algorithmic tool to solve a scientific problem. The content of the paper can as such be seen as a tools offered to the reader to acquire and add into his or her data scientist's toolbox. The building blocks that will be provided are the following:

1. What is hot? Tracking heavy hitters: count which items exceed a given frequency threshold in a stream. We'll see *Karp's algorithm* [9] and *Lossy Counting* [11] as prototypical examples and show an application in blocking excessive network usage.
2. Extreme Counting: estimate the cardinality of a set. *Flajolet-Martin sketches* [7] and the related *HyperLogLog sketch* [6] are discussed. These sketches offer a very compact representation of sets that allow cardinality estimation of the sets. There are many applications in telecommunication, yet we will show an example use of the HyperLogLog sketch for estimating the neighborhood function of a social network graph.
3. Anyone like me? Similarity search: last but not least, we consider the case of similarity search in huge collections. Especially for high-dimensional data, indexing is extremely challenging. We show how *Locality Sensitive Hashing* [8] can help reduce complexity of similarity search tremendously. We show an application for plagiarism detection in which the detection of near duplicates of a given document decreases in complexity from hours to execute to subsecond response times.

We do not claim that our list of techniques is exhaustive in any sense. Many other very important building blocks exist. However, we are convinced that the set provided in this paper is a nice addition to any data scientist's professional toolbox. The individual blocks should not be seen as the endpoint, but rather as a set of blocks that can be freely adapted and combined, depending on need. For additional resources we refer the reader to the excellent books by *Aggrawal* on stream processing [4] and by *Leskovec et al.* on mining massive datasets [10].

2 Efficient Methods for Finding Heavy Hitters

The first building block we consider is efficiently finding heavy hitters. We assume that the stream consists of items from a fixed but potentially infinite universe. For example, keyword sequences entered in a search engine, IP addressed that are active in a network, webpages requested, etc. Items arrive continuously and may be repeating. A *heavy hitter* is an item whose frequency in the stream observed sofar exceeds a given relative frequency threshold. That is, suppose that the stream we observed sofar consists of the sequence of items

$$\mathcal{S} \;=\; \langle i_1, \ldots, i_N \rangle.$$

The relative frequency of an item a is defined as:

$$\mathit{freq}(a, \mathcal{S}) \;:=\; \frac{|\{j \mid i_j = a\}|}{N}.$$

The heavy hitters problem can now be stated as follows:

Heavy Hitters Problem: Given a threshold θ and a stream $\mathcal{S}$, give the set of items

$$\mathit{HH}(\mathcal{S}, \theta) \;:=\; \{a \mid \mathit{freq}(a, \mathcal{S}) \geq \theta\}.$$

Before we go into the approximation algorithms, let's first see how much memory would be required by an exact solution. First of all it is important to realize that in an exact solution we need to maintain counts for all items seen sofar, because the continuation of the stream in future is unknown and even an error on the count of the frequency of 1 will result in a wrong result. As such, we need to be able to distinguish any two situations in which the count of even a single item differs. Indeed, suppose $\theta = 0.5$, we have seen $N/2 + 1$ items in the stream, and the count of item a is 1. Then, if the next $N/2 - 1$ items are all a's, a should be in the output. On the other hand, if in the first $N/2 + 1$ items there are no occurrences of a, a should not be in the answer, even if all $N/2 - 1$ items are a's. Therefore, the internal state of the algorithm has to be different for these two cases, and we need to keep counters for each item that appeared in the stream. In worst case, memory consumption increases linearly with the size of the stream. If the number of different items is huge, this memory requirement is prohibitive.

To solve this problem, we will rely on approximation algorithms with much better memory usage. We will see two prototypical algorithms for *approximating* the set of heavy hitters. Both algorithms have the property that they produce a superset of the set of heavy hitters. Hence, they do not produce any false negatives, but may produce some false positives. The first algorithm by Karp uses maximally $\frac{1}{\theta}$ counters, and may produce up to $\frac{1}{\theta}$ false positives, while the second algorithm, called *Lossy Counting*, is parameterized by ϵ and has the guarantee that it produces no items b with $freq(b) < \theta - \epsilon$. Hence, the only false positives are in the range $[\theta - \epsilon, \theta[$ which likely represents still acceptable results given that the threshold θ is fuzzy anyway in most cases. The algorithm realizes this guarantee using only $\mathcal{O}\left(\frac{1}{\epsilon}\log(N\epsilon)\right)$ space in worst case.

2.1 Karp's Algorithm

Karp's algorithm [9] is based on the following simple observation: suppose we have a bag with N colored balls. There may be multiple balls of the same color. Now repeat the following procedure: as long as it is possible, remove from the bag sets of exactly k balls of all different color. This procedure is illustrated in Fig. 1. When our procedure ends, it is clear that there will be balls of at most $k-1$ colors left. Furthermore, each color that appeared more than N/k times in the original bag will still be present. That is easy to verify: suppose there are $\lfloor N/k+1 \rfloor$ red balls. In order to remove all red balls, there need to be $\lfloor N/k+1 \rfloor$ sets of size k of balls of different color that were removed. But this is impossible as $k\lfloor N/k+1 \rfloor > N$. Hence, if we want to find all colors that have a relative frequency of at least θ, then we can run the algorithm with $k = \lceil 1/\theta \rceil$. In this way we are guaranteed that in the final bag we will have all θ-frequent colors left. If we want to get rid of the false positives, we can run through our original bag for a second time, counting only the at most $k-1$ different colors that were left in the bag after our procedure.

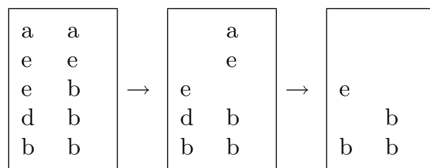


Fig. 1. Iteratively removing 3 different items from a bag; all element that had a relative frequency exceeding 1/3 will be left in the final result. In this case b and e are left. b indeed has a frequency exceeding 1/3, while e is a false positive. This procedure cannot have false negatives

The nice part of this observation is that it easily can be generalized to streaming data. Indeed, suppose we have a stream of items arriving. Each item can be considered a "color" and we need to retrieve all items that have a relative frequency exceeding θ. This can be realized by following the remove-k-different-colors procedure with $k = \lceil 1/\theta \rceil$. Because of the streaming aspect we do not have

New item	a	a	e	e	e	b	d	b	b	b
Updated counters	a:1	a:2	a:2 e:1	a:2 e:2	a:2 e:3	a:1 e:2	e:1	b:1 e:1	b:2 e:1	b:3 e:1

Fig. 2. Streaming version of the procedure in Fig. 1

any controle, however, over the order in which we need to treat the items/balls. Therefore we remember what we have seen sofar, until we reach k different items/colors. As soon as that happens, we throw out k different items. In order to remember what remains, it is easy to see that we need at most $k-1$ variables holding the items/colors we still have, plus $k-1$ counters holding how many times we still have each of them. Whenever a new ball/item arrives, we check if that color/item is already among the variables. If that is the case, we increase the associated counter. Otherwise, either we start a new counter if not all counters are in use yet, or we decrease all $k-1$ counters by 1 in order to reflect that we remove a k-tuple (one ball of each of the $k-1$ colors we already have plus the new color that just arrived). This leads to Karp's algorithm which is given in Algorithm 1 and illustrated in Fig. 2.

Notice that the update time of Algorithm 1 is $\mathcal{O}(k)$ in worst case, but in their paper, Karp et al. describe a data structure which allows processing items in constant time amortized.

Algorithm 1. Karp's algorithm

Input: Threshold ω, sequence $\mathcal{S}$ of items $i_1, \ldots, i_N$ arriving as a stream
Output: Superset of $HH(\mathcal{S}, \theta)$.

```
1:  k ← ⌈1/θ⌉
2:  L ← empty map
3:  for each item i arriving over S do
4:      if exists key i in L then
5:          L[i] ← L[i] + 1
6:      else
7:          if |L| = k − 1 then
8:              for key k of L do
9:                  L[k] ← L[k] − 1
10:                 if L[k] = 0 then
11:                     Remove the element with key k from L
12:                 end if
13:             end for
14:         else
15:             L[i] ← 1
16:         end if
17:     end if
18: end for
19: return {k | k key in L}
```

2.2 Lossy Counting

One of the disadvantages of Karp's algorithm is that it only allows for identifying a set of candidate heavy hitters, but does not provide any information regarding their frequencies. The *Lossy counting* algorithm [11] covered in this subsection, however, does allow for maintaining frequency information. Lossy counting is parameterized by ϵ. ϵ will be the bound on the maximal absolute error on the relative frequency that we guarantee. Lossy counting is based on the observation that we do not have to count every single occurrence of an item. As long as we can guarantee that the relative frequency of an item in the part of the stream in which it was not counted, does not exceed ϵ, the absolute error on the relative frequency will be at most ϵ. Indeed: suppose $\mathcal{S}$ can be divided into two disjoint sub-streams $\mathcal{S}_1$ and $\mathcal{S}_2$, and we do have the exact number of occurrences cnt_1 of item a in $\mathcal{S}_1$, and an upper bound of ϵ on the exact relative frequency $f_2 = \frac{cnt_2}{|\mathcal{S}_2|}$ of a in $\mathcal{S}_2$. Then the true relative frequency of a in $\mathcal{S}$ equals:

$$freq(a, \mathcal{S}) = \frac{cnt_1 + cnt_2}{|\mathcal{S}|} < \frac{cnt_1 + \epsilon\mathcal{S}_2}{|\mathcal{S}|} \leq \frac{cnt_1}{|\mathcal{S}|} + \epsilon.$$

This observation means that we can postpone counting any item that has a relative frequency below ϵ if we are fine with an absolute error of at most ϵ. This is exactly what Lossy Counting does: basically it counts everything, but from the moment on that it is noticed that an item's relative frequency in the window we are counting it, drops below ϵ we immediately stop counting it. If the item reappears, we start counting it again. In this way, at any point in time we can guarantee that any item that isn't counted has a relative frequency below ϵ. This principle is illustrated in Fig. 3.

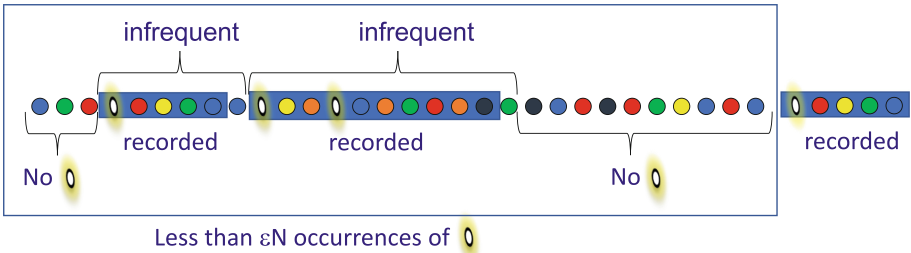


Fig. 3. Illustration of the Lossy Counting algorithm. The blue rectangles indicate periods in which the item was counted. If one takes the whole stream except for the last rectangle which is still open, then the item is ϵ-infrequent in that area (Color figure online)

The pseudo code of Lossy Counting is given in Algorithm 2. Notice that lines 10–14 constitute a potential bottleneck as we need to check after each item received from the stream, if the item is still frequent. We can, however, avoid

Algorithm 2. Lossy Counting algorithm

Input: Threshold ω, threshold ϵ, sequence $\mathcal{S}$ of items $i_1, \ldots, i_N$ arriving as a stream
Output: (i, f)-pairs such that $freq(i, \mathcal{S}) \in [f, f + \epsilon]$ and $f \geq \theta - epsilon$. The output contains a pair for each element of $HH(\mathcal{S}, \theta)$

1: $Cnt \leftarrow$ empty map
2: $Start \leftarrow$ empty map
3: **for** each item i_j arriving over $\mathcal{S}$ **do**
4:     **if** exists key i in Cnt **then**
5:         $Cnt[i_j] \leftarrow Cnt[i_j] + 1$
6:     **else**
7:         $Cnt[i_j] \leftarrow Cnt[i_j] + 1$
8:         $Start[i_j] \leftarrow j$
9:     **end if**
10:     **for** all keys k of Cnt **do**
11:         **if** $\frac{Cnt[k]}{j - Start[k] + 1} < \epsilon$ **then**
12:             Remove the elements with key k from Cnt and $Start$
13:         **end if**
14:     **end for**
15: **end for**
16: **return** $\left\{(i, f) \mid i \text{ key of } Cnt, f := \frac{Cnt[k]}{j - Start[k] + 1} > \theta - \epsilon\right\}$

this costly check by associating with every item an "expiration date"; that is: whenever we update the count of an item, we also compute after how many steps the item will no longer be ϵ-frequent unless it occurs again. This is easily achieved by finding the smallest number t such that:

$$\frac{Cnt[k]}{t - Start[i] + 1} < \epsilon.$$

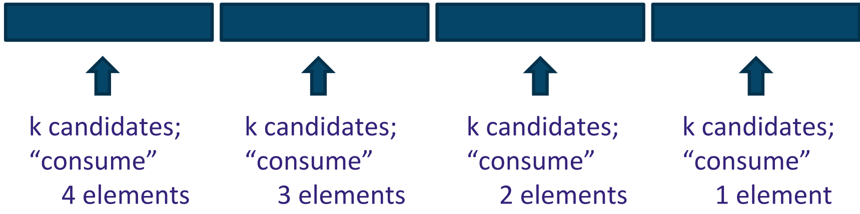
The smallest t that satisfies this inequality is:

$$\left\lfloor \frac{Cnt[k]}{\epsilon} + start[i] \right\rfloor.$$

We can order the items for which a counter exists in a priority queue according to this number and update the number and position of the item in this queue every time the item occurs. Steps 10–14 then simply become evading all items having the current time as expiration date.

For didactic purposes, the Lossy Counting variant we explained in this subsection differs slightly from the one given by *Manku and Motwani* in [11]. The computational properties, intuitions and main ideas, however, were preserved.

Let us analyze the worst case memory consumption of the Lossy Counting algorithm. The analysis is illustrated in Fig. 4. The memory consumption is proportional to the number of items for which we are maintaining a counter. This number can be bounded by the observation that every item for which we maintain a counter, must be frequent in a suffix of the stream. To analyze how



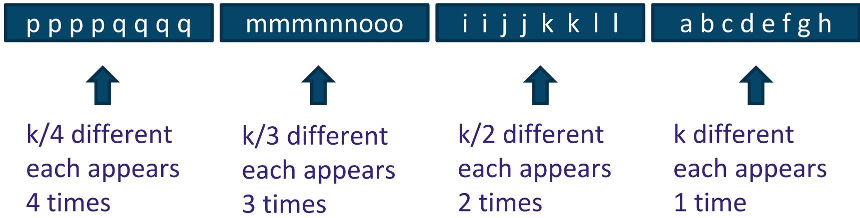


Fig. 4. Worst possible case w.r.t. memory consumption for the Lossy Counting algorithm

this affects the number of items being counted, we conceptually divide our stream in blocks of size $k = 1/\epsilon$. For an item to be supported, it needs to appear either:

- at least once in the last block;
- at least twice in the last two blocks;
- ...
- at least i times in the last i blocks;
- ...
- at least $N/k = N\epsilon$ times in the last N/k blocks; i.e., in the whole stream.

Let n_i denote the number of items that fall in the ith category above. The above observations translate into the following constraints:

- $n_1 \leq k$;
- $n_1 + 2n_2 \leq 2k$;
- ...
- $n_1 + 2n_2 + 3n_3 + \ldots + in_i \leq ik$;
- ...
- $n_1 + 2n_2 + 3n_3 + \ldots + N/kn_{N/k} \leq N$.

This number is maximized if $n_1 = k$, $n_2 = k/2$, ..., $n_i = k/i$, ... In that situation we obtain the following number of items for which we are maintaining a counter ($H(i)$ is the ith Harmonic number):

$$\sum_{i=1}^{N/k} k/i = kH(N/k) = \mathcal{O}(k \log(N/k)) = \mathcal{O}(1/\epsilon \log(\epsilon N)).$$

The memory requirements are hence logarithmic in the size of the stream *in worst case*. This worst-case, however, is a pathological case; in experiments with real-life data it was observed that the memory requirements are far less.

2.3 Applications of Heavy Hitters

Approximation algorithms for heavy hitters have many useful applications. Imagine for instance a network provider wanting to monitor its network for unreasonable bandwidth usage, and block or slow down the connection of any user using more than 0.1% of the bandwidth on any of its routers. To achieve this policy, the provider could install a lossy counter on each of its routers with θ set to 0.11% and ϵ to 0.01%. The lossy counter counts how many times IP-addresses participate in the traffic; for every packet the sender and receiver IP address is monitored. The lossy counters would catch all items with a frequency higher than 0.11% as well as some items with a frequency in the interval $[0.1\%, 0.11\%]$. Some of the users using between 0.1% and 0.11% of the bandwidth may remain unnoticed, but that can be acceptable. Installing such a lossy counter would require $10000 \log(N/10000))$ items to be stored in the absolute worst case. If the counters are reset every 1 billion packets, this would add up to at most 12K counters. That is quite acceptable for finding heavy hitters in up to a billion items.

3 Approximation Algorithms for Cardinality Counting over Streams

Another interesting building block in our toolbox is efficient cardinality counting. The setting is similar as in previous section; items are arriving one by one over a stream. This time, however, we are not interested in tracking the frequent items, but instead we want to know how many *different* items there are. Any exact solution must remember every item we have already seen. For large data collections this linear memory requirement may be unacceptable. Therefore, in this section we describe a sketching technique that maintains a succinct sketch of the stream that allows for accurately estimating the number of different items.

Cardinality Estimation Problem: Given a stream $\mathcal{S} = \langle i_1, \ldots, i_N \rangle$, give the cardinality of the set $\{i_1, \ldots, i_N\}$. That is, count the number of unique items in $\mathcal{S}$.

Reliable and efficient cardinality estimation has many applications such as counting the number of unique visitors to a website, or estimating the cardinality of projecting a relation that does not fit into memory onto a subset of its attributes without sorting.

3.1 Flajolet-Martin Sketches

The *Flajolet-Martin* sketch [7] is based on the observation that if we have a set of random numbers, the probability of observing a very high number increases

with increasing size of the set. This observation is exploited as follows: suppose we have a randomly selected hash function h that hashes every element that can arrive over our stream to a random natural number. h is a function, so if an item a occurs repeatedly in the stream, it gets assigned the same natural number. Whenever an item a arrives over the stream, we apply the hash function to it, and record the size of the longest suffix of $h(a)$ consisting only of 0's. Let $\rho(a)$ denote this number. For example, if $h(a) = 0110010_b$, then $\rho(a) = 1$, as $h(a)$ ends with only 1 zero; for $h(b) = 01101011_b$, $\rho(b) = 0$, and for $h(c) = 1000000_b$, $\rho(c) = 6$. The more different elements we observe in the stream, the more likely it is that we have seen an element x with a high $\rho(x)$, and vice versa, the higher the highest $\rho(x)$ we observed, the more likely it is that we have seen many different elements. The Flajolet-Martin sketch is based on this principle, and records the highest number $\rho(x)$ we have observed over the stream. For this we only need to remember one number: the highest $\rho(x)$ observed sofar, and update this number whenever an element y arrives over the stream with an even higher $\rho(y)$. Let's use R to denote this highest observed $\rho(x)$.

If we have one element x, the probability that $\rho(x) = t$ for a given threshold t equals $1/2^{t+1}$. Indeed, half of the numbers ends with a 1, 1/4th with 10, 1/8th with 100 and so on. The probability that $\rho(x) < t$ equals $1/2 + 1/4 + \ldots + 1/2^t = 1 - 1/2^t$.

So, suppose we have a set S with N different items, what is the probability that R exceeds a threshold t? This equals

$$
\begin{aligned}
P[\max_{x \in S} \rho(x) \geq t] &= 1 - \prod_{x \in S} P[\rho(x) < t] && (1)\\
&= 1 - (1 - 1/2^t)^N && (2)\\
&= 1 - \left((1 - 1/2^t)^{2^t}\right)^{N/2^t} && (3)\\
&\approx 1 - e^{-N/2^t} && (4)
\end{aligned}
$$

Hence, we can conclude that if $N \gg 2^t$, the probability that $R \geq t$ is close to 0, and if $N \ll 2^t$, the probability that $R \geq t$ is close to 1. We can thus use 2^R as an estimate for the cardinality N. In the original Flajolet-Martin algorithm not the maximum number $\rho(x)$ observed is used, but instead the smallest number r such that no element a was observed with $\rho(a) = r$. Then the estimator $2^r/\phi$ where ϕ is a correction factor approximately equal to 0.77351 has to be used.

The variance of this estimation, however, can be high. Therefore we can use multiple independent hash functions to create multiple independent estimators and combine them. Averaging them, however, is very susceptible to outliers, while taking the median has the disadvantage of producing estimates which are always a power of 2. Therefore, a common solution is to group estimates, take the average for each group, and take the median of all averages. In this way we get an estimate which is less susceptible to outliers because of the median, and is not necessarily a power of 2 because of the averages.

3.2 HyperLogLog Sketch

A HyperLogLog (HLL) sketch [6] is another probabilistic data structure for approximately counting the number of distinct items in a stream. The HLL sketch approximates the cardinality with no more than $\mathcal{O}(\log(\log(N)))$ bits. The HLL sketch is an array with $\beta = 2^k$ cells $(c_1, \ldots, c_\beta)$, where k is a constant that controls the accuracy of the approximation. Initially all cells are 0. Every time an item x in the stream arrives, the HLL sketch is updated as follows: the item x is hashed deterministically to a positive number $h(x)$. The first k bits of this number determine the 0-based index of the cell in the HLL sketch that will be updated. We denote this number $\iota(x)$. For the remaining bits in $h(x)$, the position of the least significant bit that is 1 is computed. Notice that this is the $\rho(x) + 1$. If $\rho(x) + 1$ is larger than $c_{\iota(x)}$, $c_{\iota(x)}$ will be overwritten with $\rho(x) + 1$.

For example, suppose that we use a HLL sketch with $\beta = 2^2 = 4$ cells. Initially the sketch is empty:

0	0	0	0

Suppose now item a arrives with $h(a) = 1110100110010110_b$. The first 2 bits are used to determine $\iota(a) = 11_\beta = 3$. The rightmost 1 in the binary representation of $h(a)$ is in position 2, and hence c_3 becomes 2. Suppose that next items arrive in the stream with $(c_{\iota(x)}, \rho(x))$ equal to: $(c_1, 3)$, $(c_0, 7)$, $(c_2, 2)$, and $(c_1, 2)$, then the content of the sketch becomes:

7	3	2	2

Duplicate items will not change the summary. For a random element x, $P(\rho(x) + 1 \geq \ell) = 2^{-\ell}$. Hence, if d different items have been hashed into cell c_ι, then $P(c_\iota \geq \ell) = 1 - (1 - 2^{-\ell})^d$. This probability depends on d, and all c_i's are independent. Based on a clever exploitation of these observations, *Flajolet et al.* [6] showed how to approximate the cardinality from the HLL sketch.

Last but not least, two HLL sketches can easily be combined into a single sketch by taking for each index the maximum of the values in that index of both sketches.

3.3 Applications: Estimating the Neighborhood Function

One application of the HLL sketch is approximating the so-called neighborhood function [5]. The algorithm we will see computes the neighborhood vector for all nodes in a graph at once. The neighborhood vector of a node is a vector $(n_1, n_2, \ldots)$ holding respectively the number of nodes at distance 1, distance 2, etc. The more densely connected a node is, the larger the numbers at the start of the vector will be. The neighborhood function is then the componentwise average of the neighborhood vector of the individual nodes; it gives the average number of neighbors at distance 1, 2, 3, etc. The neighborhood function is useful for instance to compute the effective diameter of a graph; that is: the average number of steps needed from a random node to reach a predefined fraction of the other nodes in the graph. For instance, the effective diameter for a ratio of

Algorithm 3. Neighborhood function

Input: Graph $G(V, E)$.
Output: Neighborhood function $(N_0, N_1, \ldots)$.

1: **for** $v \in V$ **do**
2: $nv_0(v) \leftarrow \{\}$
3: **end for**
4: $N_0 \leftarrow 1$
5: $i \leftarrow 0$
6: **while** $N_i \neq 0$ **do**
7: $i \leftarrow i + 1$
8: **for** $v \in V$ **do**
9: $nv_i(v) \leftarrow nv_{i-1}(v)$
10: **for** $\{v, w\} \in E$ **do**
11: $nv_i(v) \leftarrow nv_i(v) \cup nv_{i-1}(w)$
12: $nv_i(w) \leftarrow nv_i(w) \cup nv_{i-1}(v)$
13: **end for**
14: **end for**
15: $N_i \leftarrow avg_{v \in V}(|nv_i(v)|) - N_{i-1}$
16: **end while**
17: **return** $(N_0, N_1, \ldots, N_{i-1})$

90% is the average number of steps needed from a random node to reach 90% of the other nodes. Using the neighborhood function, we can easily see from which point in the vector 90% of the other nodes are covered. For instance, if the neighborhood function is $(12, 1034, 12349, 234598, 987, 3)$, then the number of steps needed is 4, as more than 90% of the nodes are at distance 4 or less. The diameter of the graph we can get by observing the rightmost entry in the neighborhood function that is nonzero. We can see if the graph is connected by adding up all numbers and comparing it to the total number of nodes. The neighborhood function of a graph is hence a central property from which many other characteristics can be derived.

A straightforward algorithm for computing the neighborhood function is given in Algorithm 3. It is based on the observation that the nodes at distance i or less of node v can be gotten by taking the union of all nodes at distance $i-1$ or less of its neighbors. Iteratively applying this principle gives subsequently the neighbors at distance 1, 2, 3, etc. of all nodes in the graph. This we continue as long as new nodes are being added for at least one vector.

The space complexity of this algorithm is $\mathcal{O}(|V|^2)$, as for every node we need to keep all other reachable nodes. This complexity, however, can easily be reduced using a HyperLogLog sketch to approximate the neighbors for all nodes. Instead of storing $nv_i(v)$ for each node, we store $HLL(nv_i(v))$. All operations we need in the algorithm are supported for the HLL; that is: taking unions (componentwise maximum of the two HLL sketches), and estimating the cardinality. In this way we get a much more efficient algorithm using only $\mathcal{O}(|V| b \log\log(|V|))$ space,

where b is the number of buckets in the HyperLogLog sketches we keep. b depends only on the accuracy of the approximation, and not on the size of the graph.

4 Anyone Like Me? Similarity Search

In a big data context, high-dimensional similarity search is a very common problem. One of the most successful classification techniques is nearest neighbor, which requires quickly finding all closest points to a given query point. Although the setting is strictly speaking no longer a streaming setting, the *Locality Sensitive Hashing* technique [8] which we will cover in this section can usefully be applied whenever items arrive at a fast pace, and quickly need to be matched to a large database of instances to find similar items. Examples include face recognition where cameras are continuously producing a sequence of faces to be recognized in a large database, or image search where one quickly needs to produce images which are alike a given image. One example we will use to illustrate the locality sensitive hashing is that of plagiarism detection, where we assume that we have a large collection of documents and whenever a new document arrives we need to be able to quickly generate all near neighbors; that is: candidate original sources of a plagiarized document.

4.1 Similarity Measure: Jaccard

We will first introduce the locality sensitive hashing technique with the so-called *Jaccard similarity measure*. The Jaccard similarity measures distances between sets. These could be sets of words occurring in a document, sets of properties or visual clues of pictures, etc. Later we will see how the Locality Sensitive Hashing technique can be extended to other similarity measure, such as the Cosine Similarity measure for instance. Given two sets A and B, their similarity is defined as:

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

Suppose now that we order all elements of the universe from which the sets A and B are drawn. Let $r(a)$ denote the rank of element a in this order, $\min_r(A)$ is then defined as $\min\{r(a) \mid a \in A\}$. We now have the following property which will be key for the locality sensitive hashing technique we will develop in the next subsection.

Minranking Property: Let r be a random ranking function assigning a unique rank to all elements from a domain U. Let $A, B \subseteq U$. Now the following property holds:

$$P[\min_r(A) = \min_r(B)] \ = \ J(A,B).$$

The probability is assuming a uniform distribution over all ranking functions r. Indeed, every element in $A \cup B$ has the same probability of being the unique element in $A \cup B$ that has the minimal rank in $A \cup B$. Only if this element is

in the intersection of A and B, $\min_r(A) = \min_r(B)$. The probability that the minimum over all elements in $A \cup B$ is reached in an element of $A \cap B$ equals $J(A, B)$.

Minrank Sketch of a Set: If we have multiple ranking functions $r_1, \ldots, r_k$, we can use these functions in order to get an estimate for $J(A, B)$ as follows: compute $\min_{r_i}(A)$ and $\min_{r_i}(B)$. Count for how many $i = 1, \ldots, k$, $\min_{r_i}(A) = \min_{r_i}(B)$. This gives us an estimate of $P[\min_r(A) = \min_r(B)] = J(A, B)$. The higher k, the more accurate our approximation will become.

There is one problem with the minrank sketch: a ranking function is very expensive to represent and store. Indeed: for a universe with n elements, there exist $n!$ rankings. Representing them requires on average $\log(n!)$ space. Therefore, instead of using a ranking function, we can use a hash function assigning numbers to the items in the range $[0, L]$ where L is significantly smaller than n. Such hash functions are usually easy to represent, and a popular choice for a hash function is $((ax + b) \ mod \ p) \ mod \ L$, where p is a prime number larger than $|U|$, and a and b are randomly drawn integers from $[1, p-1]$ and $[0, p-1]$ respectively. One problem with hash functions is that $P[\min_h(A) = \min_h(B)]$ is no longer equal to $J(A, B)$, but is slightly higher as there may be hash collisions. The probability of such a collision is, however, extremely low: let $a, b \in U$ be two different items. $P[h(a) = h(b)] = 1/L$. If L is sufficient large, this quantity can be neglected, the more since it will only cause problems if the collision happens between the smallest element in A and the smallest element in B. Unless the sets A and B are extremely large, in the order of L, we can replace ranking function by hash function in the above definitions. In this way we obtain the minhash sketch of a set A as $(\min_{h_1}(A), \ldots, \min_{h_k}(A))$. When comparing two sketches $(a_1, \ldots, a_k)$ and $(b_1, \ldots, b_k)$ we hence have the approximation $\frac{|\{i=1\ldots k \ | \ a_i=b_i\}|}{k}$.

4.2 Locality-Sensitive Hash Functions

The name *Locality Sensitive Hashing* comes from the idea that in order to index high dimensional data, we need a way to hash instances into buckets in a way that is sensitive to *locality*. Locality here means that things that are similar should end up close to each other. In other words, we look for hash functions such that the probability that two instances are hashed into the same bucket, monotonically increases if their similarity increases. If we have such a family of independent hash functions at our disposition, there are principled ways to combine them into more powerful and useful hash functions. Our starting point is a family of independent hash functions $\mathcal{F}$ which is (s_1, p_1, s_2, p_2)-*sensitive* for a similarity measure *sim*. That means that for any $h \in \mathcal{F}$, $P[h(x) = h(y)]$ is non-decreasing with respect to $sim(x, y)$, $P[h(x) = h(y) | sim(x, y) < s_1] \leq p_1$, and $P[h(x) = h(y) | sim(x, y) \geq s_2] \geq p_2$. For the Jaccard-index we do have such a family of hash functions, namely the functions $\min_h(.)$ for random hash functions h. This family of functions is (s_1, s_1, s_2, s_2)-sensitive.

Suppose we have a set D of objects from universe U and we need to index them such that we can quickly answer the following (at this point still informal)

near-duplicate query: given a query point q from universe U, give all objects d in D such that $sim(q,d)$ is high. If we have a family of (s_1,p_1,s_2,p_2)-sensitive hash functions, we could index the objects in D as follows: pick a hash function h from $\mathcal{F}$ and divide D into buckets according to the hash value given by h; that is: for each hash value v in $h(D)$, there is a bucket $D[v] := \{d \in D \mid h(d) = v\}$. If we now need a near duplicate of a query point $q \in U$, we will only check the elements d in the bucket $D[h(q)]$. Since h is from a (s_1,p_1,s_2,p_2)-sensitive family, we are guaranteed that if an object $d \in D$ has $sim(d,q) \geq s_2$, then $P[d \in D[h(q)]] \geq p_2$. On the other hand if the similarity $sim(q,d)$ is lower than s_1, the chance of finding d in the same bucket as q decreases to p_1. If the documents with similarity exceeding s_2 represent the ones we need to retrieve, the ones in the interval $[s_1,s_2]$ are acceptable but not necessary in the result, and the ones with similarity lower than s_1 represent documents that shouldn't be in the answer, then p_1 can be interpreted as the *False Positive Ratio*; that is, the probability that a negative example is a FP, and p_2 the probability that a positive example is correctly classified and hence a TP; i.e., the *True Positive Ratio*.

Often, however, the sensitivities (s_1,p_1,s_2,p_2) are insufficient for applications. For instance, if we use minhashing for finding plagiarized texts where documents are represented as the set of words they contain, the sensitivity we get is (s_1,s_1,s_2,s_2) for any pair of numbers $s_1 < s_2$. So, if we consider a text plagiarized if the similarity is above 90% and not plagiarized if the similarity is less than 80%, then the indexing system proposed above has a guaranteed true positive rate of only 90% and a false positive rate of up to 80%. This is clearly not acceptable. Fortunately, there exist techniques for "boosting" a family of sensitive hash functions in order to achieve much better computational properties. This boosting technique can be applied on any family of hash-functions that are (s_1,p_1,s_2,p_2)-sensitive for a given similarity function, as long as the hash functions are independent, as we will see next.

4.3 Combining Locality-Sensitive Hash Functions

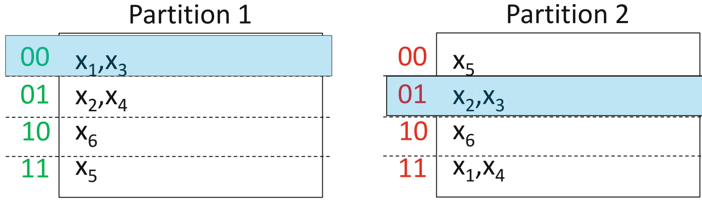
Instead of creating an index based on one hash function, we can combine up to k hash functions $h_1,\ldots,h_k$ as follows: assign each document x in our collection D to a bucket $D[h_1(x),h_2(x),\ldots,h_k(x)]$ where $D[v_1,\ldots,v_k] := \{d \in D \mid h_1(d) = v_1,\ldots,h_k(d) = v_k\}$. If a query q comes, we check the similarity with the documents in bucket $D[h_1(q),\ldots,h_k(q)]$ only. We now get:

$$P[q \in D[h_1(d),\ldots,h_k(d)]] = P[\forall i = 1\ldots k : h_i(q) = h_i(d)] \tag{5}$$

$$= \prod_{i=1}^{k} P[h_i(q) = h_i(d)] \tag{6}$$

Hence, the combination of hash functions is (s_1,p_1^k,s_2,p_2^k)-sensitive. In this way we can reduce the number of false positives tremendously; for $p_1 = 80\%$ as in our example above, for $k = 10$, the false positive rate decreases from 80% to $(80\%)^{10}$, which is less than 11%!

- D = { 00110011, 01010101, 00011100, 01110010, 11001100, 10101010 }



- Query q = 00011100
  - Only compute distance to x_1,x_2,x_3

Fig. 5. LSH-Index based on a (2, 2)-scheme. For illustrative purposes a simple hamming distance between 0–1 vectors is chosen, defined as the fraction of entries on which the vectors correspond. The first index is based on the first two entries in the vector, and the second index on the next two entries. A query point is compared to all vectors in the 2 buckets in which the query point is hashed (one for the first index, one for the second)

The true positive rate, however, decreases as well: from 90% to around 35%. To counter this problem, however, we can create multiple indices for different sets of hash functions: $H^1 = (h_1^1, \ldots, h_k^1)$, ..., $H^\ell = (h_1^\ell, \ldots, h_k^\ell)$. For each $j = 1 \ldots \ell$ we create an independent index for the documents. Each document $d \in D$ gets assigned to ℓ buckets: $D^1[H^1(d)]$, ..., $D^\ell[H^\ell(d)]$, where $H^i(d)$ is shorthand for the composite tuple $(h_1^i(d), \ldots, h_k^i(d))$. If a query q comes, we will compare q to all documents in $D^1[H^1(q)] \cup \ldots \cup D^\ell[H^\ell(q)]$. This way of indexing data is illustrated in Fig. 5.

Suppose that $P[h(x) = h(y)] = p$ for al given pair of documents x, y and a random hash function from a given family of hash functions. Then the probability that x and y share at least one bucket in the ℓ indices under our (k, ℓ)-scheme equals:

$$P[x \text{ and } y \text{ share at least one bucket}] = 1 - P[x \text{ and } y \text{ share no bucket}] \quad (7)$$

$$= 1 - \prod_{j=1}^{\ell} P[H^j(x) \neq H^j(y)] \quad (8)$$

$$= 1 - \prod_{j=1}^{\ell} (1 - P[H^j(x) = H^j(y)]) \quad (9)$$

$$= 1 - \prod_{j=1}^{\ell} (1 - p^k) \quad (10)$$

$$= 1 - (1 - p^k)^\ell \quad (11)$$

Hence our (k, l)-scheme is $(s_1, 1-(1-p_1^k)^\ell, s_2, 1-(1-p_2^k)^\ell)$-sensitive. As long as our family of hash functions is large enough to allow for $k\ell$ hash functions, we can achieve any desired precision (s_1, P_1, s_2, P_2) for our indexing scheme by solving the following system of equations for l and k:

$$\begin{cases} P_1 = 1-(1-p_1^k)^\ell \\ P_2 = 1-(1-p_2^k)^\ell \end{cases}$$

Figure 6 plots some examples in which the similarity of two documents is plotted against the probability that they share at least one bucket, for the Jaccard similarity measure using the minhash family. Recall that $P[h(x) = h(y)] \approx J(x, y)$, which makes the relation between the similarity of two documents x and y and their probability of sharing a bucked straightforward: $P[x \text{ shares bucket with } y] = 1-(1-J(x, y)^k)^\ell$.

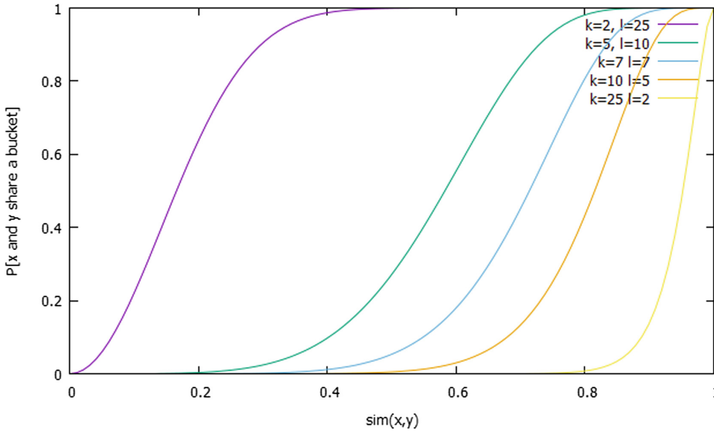


Fig. 6. Relation between the similarity and probability of being in the same bucket under different (k, ℓ)-hashing schemes (LSH for Jaccard using minhash)

4.4 LSH for Cosine Similarity

Locality-sensitive hashing works not only for the Jaccard-index; any similarity measure for which we can find an appropriate family of hash functions for, can benefit from this framework. We will illustrate this principle with one more example: the *cosine similarity measure*. The universe from which our documents and queries come are N-dimensional vectors of non-negative numbers, for instance TF.IDF-vectors for text documents. Given two vectors, $\mathbf{x} = (x_1, \ldots, x_N)$ and $\mathbf{y} = (y_1, \ldots, y_N)$, the *cosine similarity* between them is defined as $\frac{\mathbf{x} \cdot \mathbf{y}}{|\mathbf{x}||\mathbf{y}|}$, where $\cdot$ is the scalar product and $|.|$ the $l2$-norm. The cosine similarity measure thanks its name to the fact that it equals the cosine of the angle formed by the two vectors.

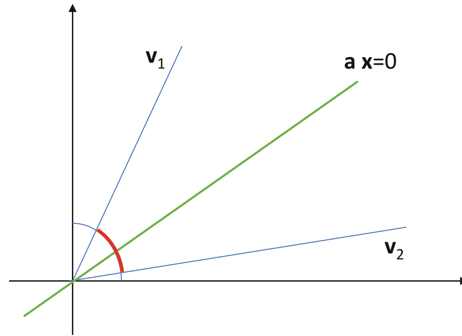


Fig. 7. Random hyperplane separating 2 2D vectors (Color figure online)

It is this property that will be exploited by the hash functions we will design: every hash function we consider is associated with a random hyperplane through the origin. All points on one side of the hyperplane get assigned 0, and all points on the other side get assigned 1. That is, if the equation of the hyperplane is $\mathbf{a} \cdot \mathbf{x} = 0$, then the hash function $h_{\mathbf{a}}$ we consider is defined as $h_{\mathbf{a}}(\mathbf{x}) = \mathit{sign}(\mathbf{a} \cdot \mathbf{x})$. It can be shown that if we chose the hyperplane by drawing each of the components of $\mathbf{a}$ from an independent standard normal distribution, then the probability that we separate two vectors by the random hyperplane $\mathbf{a} \cdot \mathbf{x} = 0$ is proportional to the angle between the two vectors. This situation is depicted for 2 dimensions in Fig. 7. The green line represents the separating hyperplane between two vectors. The plane, in 2D a line, separates the vectors if its slope is in $[\alpha_1, \alpha_2]$ where α_i is the angle between the horizontal axis and the vector $\mathbf{v}_i$. The probability that this happens, if all slopes are equally likely, is $\frac{\alpha}{\pi/2}$, where α is the angle between $\mathbf{v}_1$ and $\mathbf{v}_2$; i.e., $\alpha = |\alpha_2 - \alpha_1|$. Hence, we get:

$$P[h_{\mathbf{a}}(\mathbf{x}) = h_{\mathbf{a}}(\mathbf{y})] = 1 - \frac{\alpha}{\pi/2} \tag{12}$$

$$= 1 - \frac{\arccos(\mathit{sim}(\mathbf{x}, \mathbf{y}))}{\pi/2} \tag{13}$$

As $\arccos(x)$ is monotonically decreasing for $x \in [0, 1]$, the probability that two elements share a bucket is monotonically increasing with the cosine similarity between the two elements, which is exactly the LSH property we need to use the technique of last subsection. We can again combine the hash functions into ℓ groups of k independent hashes. In this way we get an index where two elements share at least one bucket with a probability of:

$$1 - \left(1 - \left(1 - \frac{\arccos(\mathit{sim}(\mathbf{x}, \mathbf{y}))}{\pi/2}\right)^k\right)^\ell$$

This probability in function of the cosine similarity between two documents is depicted in Fig. 8.

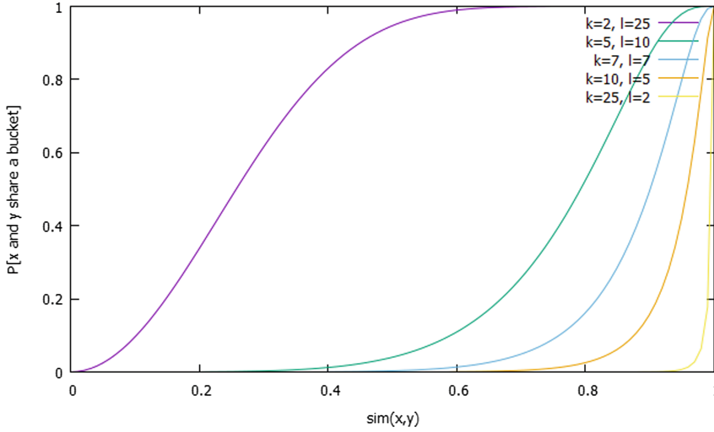


Fig. 8. Relation between the similarity and probability of being in the same bucket under different (k, ℓ)-hashing schemes (LSH for cosine similarity)

4.5 Application: Plagiarism Detection

One potential application of LSH is plagiarism detection. We will illustrate this application with a collection of 23M Wikipedia documents. Each document consists of one chapter of a Wikipedia page. The pages are preprocessed as follows: first the pages are decomposed into their 4-shingles; that is: each page is represented by the set of all 4 consecutive words in the text. For instance, if the document is "Royal Antwerp Football Club is the number 1 team in Belgium", then the representation becomes: {"Royal Antwerp Football Club", "Antwerp Football Club is", "Football Club is the", ..., "1 team in Belgium"}. Subsequently, to reduce space, all shingles are hashed into a unique number. After that, two documents are compared using the Jaccard similarity. Via minhashing we create 49 independent (s_1, s_1, s_2, s_2)-sensitive hash functions. These are combined into an LSH-index using a $(7, 7)$-scheme.

In order to get an idea of the overall distribution of the similarities between two random documents in the collection, we sampled a subset of 1000 documents. For these 1000 documents, the similarity between all pairs is measured. These numbers, extrapolated to the whole collection, are plotted as a histogram in Fig. 9. As can be seen in this histogram, the vast majority of pairs of documents has low similarity (notice incidentally that the scale on the vertical axis is logarithmic). Only about 100 document pairs have a similarity higher than 90%, and there is a gap between 70% and 90%. This indicates that, as to be expected, there is some duplicate content in Wikipedia, which scores a similarity higher than 90%, while the normal pairs score at most around 70% with a vast majority of pairs of documents having similarities around 10%, 20%, 30%. This is very good news for the application of LSH. Indeed, it indicates that any indexing scheme which is $(30\%, p_1, 90\%, p_2)$-sensitive with low p_1 and high p_2 will perform very well. The large gap between $s_1 = 30\%$ and $s_2 = 90\%$ means

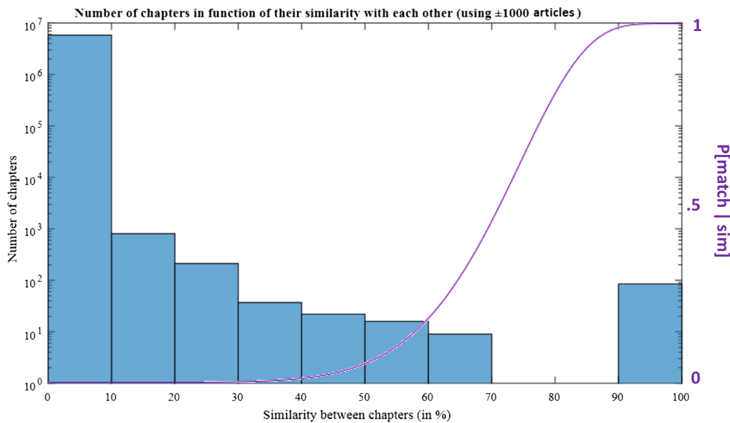


Fig. 9. Histogram representing the extrapolated numbers of pairs of documents with respect to similarity (binned per 10%; logscale). Overlayed is the probability that two documents share a bucket in a $(7, 7)$ LSH index (normal scale)

that we will not need a lot of hash functions. In Fig. 9, the histogram has been overlayed with the probability of sharing a bucket for a $(7, 7)$-scheme. As can be seen, this indexing scheme should perform very well; for most of the pairs of documents the similarity is low, and at the same time the probability that those pairs end up in the same bucket is extremely low. Hence, the number of false positives will be very low relative to the total number of pairs. On the other hand, for the highly similar documents, the similarity is high and the probability of those pairs ending up in the same bucket is nearly 1. So, not too many false negatives to be expected either.

Because of the low number of candidates that will have to be tested, the time of finding duplicates in an experiment with this setup went down from over 6 hours to compare a query document to all documents in the collection, to less than a second, all on commodity hardware. The exact run times depend on the exact characteristics of the setup and the similarity distribution among the documents, but in this particular case a speedup of over 20000 times could be observed using LSH, with virtually no false negatives.

5 Conclusions

In this overview we reviewed three techniques which can come in handy when working with large amounts of data. First of all, we looked into fast and efficient algorithms for recognizing *heavy hitters*; that is: highly frequent items, in a stream. Then we went into even more efficient sketches for streaming data which allow for cardinality estimation of a stream. Last but not least, we reviewed the *Locality Sensitive Hashing* technique for similarity search in large data collections. These techniques, and combinations thereof are frequently handy when

working with large data collections, and are a nice addition to a data scientists toolbox. A number of applications we gave were: finding users in an IP network using an excessively large fraction of the bandwidth, computing the neighborhood function of a graph, and plagiarism detection.

References

1. Apache flink. https://flink.apache.org/
2. Apache hadoop. http://hadoop.apache.org
3. Apache spark. https://spark.apache.org/
4. Aggarwal, C.C.: Data Streams. ADBS, vol. 31. Springer, Boston (2007). https://doi.org/10.1007/978-0-387-47534-9
5. Boldi, P., Rosa, M., Vigna, S.: HyperANF: approximating the neighbourhood function of very large graphs on a budget. In: Proceedings of the 20th International Conference on World Wide Web, pp. 625–634. ACM (2011)
6. Flajolet, P., Fusy, É., Gandouet, O., Meunier, F.: HyperLogLog: the analysis of a near-optimal cardinality estimation algorithm. In: AofA: Analysis of Algorithms. Discrete Mathematics and Theoretical Computer Science, pp. 137–156 (2007)
7. Flajolet, P., Martin, G.N.: Probabilistic counting algorithms for data base applications. J. Comput. Syst. Sci. **31**(2), 182–209 (1985)
8. Indyk, P., Motwani, R.: Approximate nearest neighbors: towards removing the curse of dimensionality. In: Proceedings of the Thirtieth Annual ACM Symposium on Theory of Computing, pp. 604–613. ACM (1998)
9. Karp, R.M., Shenker, S., Papadimitriou, C.H.: A simple algorithm for finding frequent elements in streams and bags. ACM Trans. Database Syst. (TODS) **28**(1), 51–55 (2003)
10. Leskovec, J., Rajaraman, A., Ullman, J.D.: Mining of Massive Datasets. Cambridge University Press, Cambridge (2014)
11. Manku, G.S., Motwani, R.: Approximate frequency counts over data streams. In: Proceedings of the 28th International Conference on Very Large Data Bases, pp. 346–357. VLDB Endowment (2002)