What is Data Sketching, and Why Should I Care?

Graham Cormode

September 15, 2017

Do you ever feel overwhelmed by a constant stream of information? It can seem like there is a barrage of new email and text messages arriving, phone calls, articles to read, and knocks on the door. Putting these pieces together to keep track of what's important can be a real challenge.

The same information overload is of concern in many computational settings. For example, telecommunications companies want to keep track of the activity on their network, to identify the overall network health, and spot anomalies or changes in behavior. Yet, the scale of events occurring is huge: many millions of network events per hour, per network element. And while new technologies allow the scale and granularity of events being monitored to increase by orders of magnitude, the capacity of computing elements to make sense of these (processors, memory and disks) is barely increasing. Even on a small scale, the amount of information may be too large to store in an impoverished setting (say, an embedded device), or be too big to conveniently keep in fast storage.

In response to this challenge, the model of streaming data processing has grown in popularity. In this setting, the aim is no longer to capture, store and index every minute event, but rather to quickly process each observation to create some summary of the current state. Following its processing, an event is dropped, and is no longer accessible. The summary that is retained is often referred to as sketch of the data. Coping with the vast scale of information means making a number of compromises: the description of the world is approximate, rather than exact; the nature of queries to be answered must be decided in advance, rather than after the fact; and some questions are now insoluble. However, the ability to process vast quantities of data at blinding speeds with modest resources can more than make up for these limitations. As a consequence, streaming methods have been adopted in a number of domains, starting with telecommunications, but spreading to search engines, social networks, finance, and time-series analysis. These ideas are also finding application in areas where traditional approaches are applicable, but the rough and ready sketching approach is more cost-effective. Successful applications of sketching involve a mixture of algorithmic tricks, systems know-how, and mathematical insight, and have led to new research contributions in each of these areas.

In this article, we introduce the ideas behind, and applications, of sketching, with a focus on the algorithmic innovations. That is, we describe some algorithmic developments in the abstract, and then indicate the subsequent steps needed to put them into practice, with examples. We will see four novel algorithmic ideas, and discuss some emerging areas.

1 Simply Sampling

When faced with a large amount of information to process, there may be a strong temptation to just ignore it entirely. A slightly more principled approach is to just ignore *most of it*. That is,

take a small number of examples from the full data set, perform the computation on this subset, and then try to extrapolate to the full data. This puts us in the realm of sampling: to give a good estimation, we should ensure that the examples picked are randomly chosen. There are a huge number of variations of sampling that have been described, but here we'll discuss the most basic, uniform random sampling. Suppose we have a large collection of customer records. Randomly picking out a small number of records gives our sample. Then, various questions can be answered accurately by only looking at the sample: for example, we can estimate what fraction of customers live in a certain city, or have bought a certain product.

The Method. To flesh this out, we need to fill in a few gaps. Firstly, how big should the sample be to give good answers? Using standard statistical results, for questions like those in the customer records example, the standard error of a sample of size s is proportional to $\frac{1}{\sqrt{s}}$. Roughly speaking, this means that if we estimate a proportion from the sample, we should expect the error to look like $\pm\frac{1}{\sqrt{s}}$. So, if we take a subset of 1000 voters, and look at their voting intention, we have an opinion poll whose error is approximately 3%. We have high confidence (but not certainty) that the true answer is within 3% of the result on the sample, assuming that the sample was drawn randomly, and the participants responded honestly. Increasing the size of the sample causes the error to decrease in a predictable, albeit expensive, way: if we want to reduce the margin of error of an opinion poll to 0.3%, we need to contact 100,000 voters.

Secondly, how should we draw the sample? Simply taking the first s records is not guaranteed to be random: there may be clustering through the data. We need to ensure that every record has an equal chance of being included in the sample. This can be achieved by using standard random number generators to pick which records to include in the sample. A common trick is to attach a random number to each record, then sort the data based on this random tag, and take the first s records in the sorted order. This works fine, so long as sorting the full data is not too costly.

Last, how do we maintain the sample as new items are arriving? A simple approach is to pick every record with probability p, for some chosen value of p. When a new record comes, we pick a random fraction between 0 and 1, and if it is smaller than p, we put the record in the sample. The problem with this approach is that we do not know what p should be in advance. For the analysis above, we want a fixed sample size s, and using a fixed sampling rate p means we have too few elements initially, but then too many as more records arrive. Presented this way, the question has the appearance of an algorithmic puzzle, and indeed this was a common question in technical interviews for many years. One can come up with clever solutions that incrementally adjust p as new records arrive. A simple and elegant way to maintain a sample is to adapt the idea of random tags. Attach to each record a random tag, and define the sample to be the s records with the smallest tag values. As new records arrive, we can use the tag values decide whether to add the new record to the sample (and to remove an old item to keep the sample size fixed at s).

Discussion and Applications. Sampling methods are so ubiquitous that many examples can be given. One simple case is within database systems: it is common for the database management system to keep a sample of large relations for the purpose of query planning: when determining how to execute a query, evaluating different strategies gives an estimate of how much data reduction there may be at each step, with some uncertainty of course. Another example arrives in data integration and linkage, when a subproblem is to test whether two columns from separate tables can relate to the same set of entities. Comparing the columns in full can be time consuming, especially when we want to test all pairs of columns for compatibility. Comparing a small sample is often enough to determine whther the columns have any chance of relating to the same entities.

Entire books have been written on the theory and practice of sampling, particularly around sampling schemes which try to preferentially sample the more important elements, to reduce the

error in estimating from the sample. A good recent survey with a computational perspective is given by Jermaine [11].

Given the simplicity and generality of sampling, why would we need any other method to summarize data? It turns out that there are some questions that sampling is not well-suited for. Any question that requires detailed knowledge of individual records in the data cannot be answered by sampling. For example, if we wanted to know whether one specific individual is amongst our customers, then a sample will leave us uncertain. If that customer is not in the sample, we do not know whether it is because they are not in the data, or because they did not happen to be sampled. Questions like this ultimately need all the presence information to be recorded, and are answered by highly compact encodings such as the Bloom Filter (described next).

A more complex example is when the question involves determining the cardinality of quantities: we might ask, in a data set that has many different values, how many distinct values are there of a certain type? For example, we might want to know how many distinct surnames there are in our customer data set. Using a sample does not tell us much about this information. Let's say that in a sample of size a thousand out of a million records, we see that 900 of the surnames in the sample occur just once among the sampled names. What can we conclude about the popularity of these names in the rest of the dataset? It might be that almost every other name in the full data is also unique. Or it might be that each of the unique names in our sample reoccurs tens or hundreds of times in the remainder of the data. From the sampled information, we have no way to distinguish these two cases, which leads to huge confidence intervals on these kind of statistics. Tracking information about cardinalities, and omitting duplicates, is addressed by techniques like Hyperloglog, discussed below.

Last, there are quantities that samples can estimate, but for which better special purpose sketches exist. Recall that the standard error of a sample of size s is $1/\sqrt{s}$. For problems like estimating the frequency of a particular attribute (such as city of residence), we can build a sketch of size s so that the error it guarantees is proportional to $1/s$. This is considerably stronger than the sampling guarantee, and only improves as we increase more space s to the sketching. The Count-Min sketch described below has this property. A limitation is that we need to specify the attribute of interest in advance of setting up the sketch, while a sample allows us to evaluate a query for any attribute that is recorded of the sampled items.

Because of its flexibility, sampling is a powerful and natural way to build a sketch of a large data set. There are many different approaches to sampling which aim to get the most out of the sample, or to target different types of query that the sample may be used to answer [11]. Next, we give more information about less flexible methods which address some of the limitations identified above of sampling.

2 Summarizing Sets with Bloom Filters

The Bloom filter is a compact data structure that summarizes a set of items. Any computer science data structures class is littered with examples of "dictionary" data structures, such as arrays, linked lists, hash tables and many esoteric variants of balanced tree structures. These common feature of these structures is that they can all answer "membership questions" of the form "is a certain item stored in the structure or not?". The Bloom filter can also respond to such membership questions. However, the answers given by the structure are either "the item has definitely not been stored" or "the item has *probably* been stored". This introduction of uncertainty over the state of an item (we might think of it as introducing potential false positives) allows the filter to to use an amount of space that is much smaller than its exact relatives. The filter also does not allow

us to list the items that have been placed into it – instead, we can only pose membership questions for specific items.

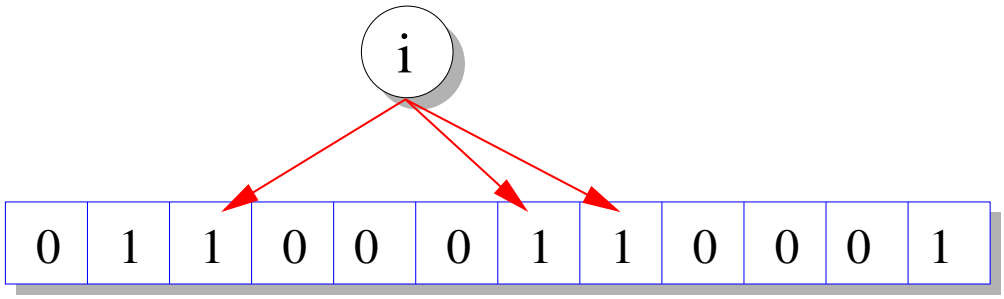
The Method. To understand the filter, it is helpful to think of a simple exact solution to the membership problem. Suppose we want to keep track of which of a million possible items we have seen, and each one is helpfully labeled with its id number (an integer between one and a million). Then we can keep an array of a million bits, initialized to all zeros. Every time we see an item i, we just set the ith bit in the array to 1. A look-up query for item j is correspondingly straightforward: just see whether bit j is a 1 or a 0. The structure is very compact: 125KB will suffice, if we pack the bits into memory.

However, real data is rarely this nicely structured. In general, we might have a much larger set of possible inputs – think again of the names of customers, where there are a huge number of possible name strings. Nevertheless, we can adapt our bit array approach if we borrow from a different dictionary structure. We can imagine that the bit array is a hash table: we will use a hash function h to map from the space of inputs onto the range of indices for our table. That is, given input i, we now set bit $h(i)$ to 1. Of course, now we have to worry about hash collisions. That is, multiple entries might map onto the same bit. In a traditional hash table, we can handle this, as we can keep information about the entries in the table. However, if we stick to our guns and only keep the bits in the bit array we will get false positives: if we look up item i, it may be that entry $h(i)$ is set to 1, but i has not been seen; instead there is some item j that was seen, where $h(i) = h(j)$.

Can we fix this while sticking to a bit array? Not entirely, but we can make it less likely. Rather than just hashing each item i once, with a single hash function, we can use a collection of k hash functions $h_1, h_2, \ldots h_k$, and map i with each them in turn. We set all the bits corresponding to $h_1(i)$, $h_2(i) \ldots h_k(i)$ to one. Now to test membership of j, we check that all the entries it is hashed to, and say 'no' if any of them are zero.

There's clearly a tradeoff here: initially, adding extra hash functions reduces the chances of a false positive as more things need to "go wrong" for an incorrect answer to be given. However, as more and more hash functions are added, the bit array gets fuller and fuller of 1 values, and so we are more likely to see collisions. This tradeoff can be analyzed mathematically, and the sweet spot found that minimizes the chance of a false positive. The analysis works by assuming that the hash functions look completely random (which is a reasonable assumption in practice), and looks at the chance that an arbitrary element not in the set is reported as present.

If there are n distinct items being stored in a Bloom filter of size m, and k hash functions used, then the chance of an membership query which should receive a negative answer yielding a false positive is approximately $1 - \exp(k \ln(1 - \exp(-kn/m)))$ (see [4] for the derivation of this expression). While extensive study of this expression may not be rewarding in the short term, some simple analysis shows that this rate is minimized by picking $k = (m/n) \ln 2$. This corresponds to the case when about half the bits in the filter are 1 and half are 0. For this to work, we should choose the number of bits in the filter to be some multiple of the number of items that we expect to store in it. A common setting is to set $m = 10n$ and $k = 7$, which means a false positive rate of below 1%. Note that there is no magic here to compress data beyond information theoretic limits: under these parameters, the Bloom filter uses about 10 bits per item, and must use space proportional to the number of different items stored. This is a modest saving when representing integer values, but is a considerable benefit when the items stored have large descriptions – say, arbitrary strings such as URLs. Storing these in a traditional structure such as a hash table or balanced search tree would consume tens or hundreds of bytes per item. A simple example is shown in Figure 1.



An item i is mapped by $k = 3$ hash functions to a filter of size $m = 12$, and these entries are set to 1.

Figure 1: Bloom Filter with $k = 3$, $m = 12$

Discussion and Applications. The possibility of false positives needs to be handled carefully. Bloom filters are at their most attractive when the consequence of a false positive is not the introduction of an error in a computation, but rather when it causes some additional work that does not adversely impact the overall performance of the system. A good example comes in the context of browsing the web. It is now common for web browsers to warn users if they are attempting to visit a site which is known to host malware. This is done by checking the URL against a database of "bad" URLs. The database is large enough, and URLs are long enough, that keeping the full database as part of the browser would be unwieldy, especially on mobile devices. Instead, we can include a Bloom filter encoding of the database with the browser, and check each URL visited against this. The consequence of a false positive is that the browser may believe that an innocent site is on the bad list. To handle this, the browser can contact the database authority, and check whether the full URL is on the list. Hence, false positives are removed, at the cost of a remote database look-up. Now notice the effect of the Bloom filter: it gives the all-clear to most URLs, and incurs a slight delay for a small fraction (or when a bad URL is visited). It is preferable to the solution of a copy of the database with the browser, and to doing a remote look-up for every URL visited. Browsers such as Chrome and Firefox adopt this concept. Current versions of Chrome use a variation of the Bloom filter based on more directly encoding a list of hashed URLs, since the local copy does not have to be updated dynamically, and so more space can be saved this way.

The Bloom filter is approaching its half-century. It was introduced in 1970 as a compact way of storing a dictionary, when space was really at a premium [3]. As computer memory increased in size, it seemed that the filter was no longer needed. However, along with the rapid growth of the Web, a host of applications for the filter have been found since around the turn of the century [4]. Many of the applications have the flavor of the above example: the filter gives a fast answer to look-up queries, and positive answers may be double-checked in an authoritative reference. Bloom filters have been widely used to avoid storing unpopular items in caches. This enforces the rule that an item is only added to the cache if it has been seen once before. The Bloom filter is used to compactly represent the set of items that have been seen. The consequence of a false positive is that a small fraction of rare items might also be stored in the cache, contradicting the letter of the rule. Many large distributed databases (Google's BigTable, Apache's Cassandra and HBase) use Bloom filters as indexes on distributed chunks of data. They use the filter to keep track of which rows or columns of the database are stored on disk, and avoid a (costly) disk access for non-existent attributes.

3 Counting with Count-Min sketch

Perhaps the canonical data summarization problem is the most trivial: to count the number of items of a certain type that have been observed, we do not need to retain each item. Instead, a simple counter suffices, incremented with each observation. The counter has to be of sufficient bit-depth in order to cope with the magnitude of events observed. When the number of events gets truly huge, ideas such as Morris' approximate counter can be used to provide an approximate counter in fewer bits [12] (another example of a sketch).

When there are different types of item, and we want to count each type, then the natural approach is to allocate a counter for each item. However, when the number of item types grows huge, then we encounter difficulties. It may not be practical to allocate a counter for each item type. Even if it is, when the number of counters exceeds the capacity of fast memory, then the time cost of incrementing the relevant counter may become too high. For example, a social network like Twitter may wish to track how often a tweet is viewed when it is displayed via an external website. There are billions of webpages, each of which could in principle link to one or more tweets, so allocating counters for each is infeasible and unnecessary. Instead, it is natural to look for a more compact way to encode counts of items, possibly with some tolerable loss of fidelity.

The Count-Min sketch is a data structure that allows this tradeoff to be made. It encodes a potentially massive number of item types in a small array. The guarantee is that large counts will be preserved fairly accurately, while small counts may incur greater (relative) error. This means that it is good for applications where we are interested in the head of a distribution, and less so in its tail.

The Method. At first glance, the sketch looks quite like a Bloom filter, as it is involves the use of an array and a set of hash functions. However, there are significant difference in the details, so we will present it afresh. The sketch is formed of an array of counters, and a set of hash functions which map items into the array. More precisely, we treat the array as a sequence of rows, and each item is mapped by the first hash function into the first row, by the second hash function into the second row, and so on (note that this is in contrast to the Bloom filter, which allows the hash functions to map onto overlapping ranges). To process an item, we map it to each row in turn via the corresponding hash function, and increment the counters to which it is mapped.

Given an item, the sketch allows its count to be estimated. This follows a similar outline to processing an update: we inspect the counter in the first row where the item was mapped by the first hash function, and the counter in the second row where it was mapped by the second hash, and so on. In each row, we find a counter that has been incremented by every occurrence of the item. However, the counter was also potentially incremented by occurrences of other items that were mapped to the same location, since we expect that there will be collisions. Given the collection of counters containing the desired count, plus noise, our best guess at the true count of the desired item is to take the smallest of these counters as our estimate.

Figure 2 shows the update process: an item i is mapped to one entry in each row j by the hash function h_j, and the update of c is added to each entry. It can also be seen as modeling the query process: a query for the same item i will result in the same set of locations being probed, and the smallest value returned as the answer.

Discussion and Applications. As with the Bloom filter, the sketch achieves a compact representation of the input, with a tradeoff in accuracy. Both give us some probability of an unsatisfactory answer. With a Bloom filter, the answers are binary, so we have some chance of a false positive response; with a Count-Min sketch, the answers are frequencies, so we have some chance of an inflated answer.

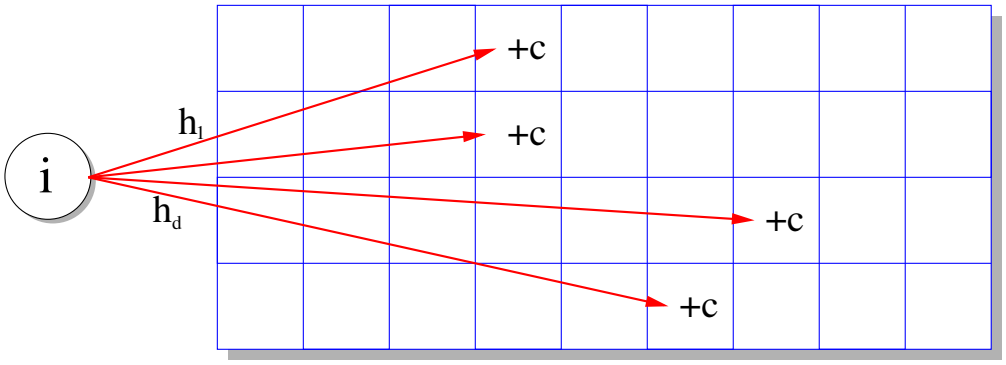


Figure 2: Count-Min sketch data structure with $t = 9$ and $d = 4$

What may be surprising at first is that the obtained estimate is very good. Mathematically, it can be shown that there is a good chance that the returned estimate is close to the correct value. The quality of the estimate depends on the number of rows in the sketch (each additional row halves the probability of giving a bad estimate), and on the number of columns (doubling the number of columns halves the scale of the noise in the estimate). These guarantees follow from the random selection of hash functions, and do not rely on any structure or pattern in the data distribution which is being summarized. That is, for a sketch of size s, the error is proportional to $\frac{1}{s}$. This is an improvement over the case for sampling where, as was noted above, the corresponding behavior is proportional to $\frac{1}{\sqrt{s}}$.

Just as Bloom filters suit best the cases where false positives can be tolerated and mitigated, Count-min sketches are best suited to when we can handle a slight inflation of frequency. This means in particular that they do not apply to cases where we might use a Bloom filter: if it matters a lot whether an item has been seen or not, then the uncertainty that the Count-Min sketch introduces will obscure this level of precision. However, the sketches are very good for tracking which items exceed a given popularity threshold. In particular, while the size of a Bloom filter must remain proportional to the size of the input it is representing, a Count-Min sketch can be much more compressive: its size can be considered as independent of the input size, and depends instead on the desired accuracy guarantee only (i.e. to achieve a target accuracy of ϵ, we fix a sketch size of s proportional to $1/\epsilon$ that does not vary over the course of processing data).

Discussion and Applications. The Twitter scenario above is a good example. Tracking the number of views that a tweet receives across each occurrence in different websites creates a large enough volume of data to be difficult to manage. Moreover, the existence of some uncertainty in this application seems acceptable: the consequences of inflating the popularity of one website for one tweet are minimal. Using a sketch for each tweet consumes only moderately more space than the tweet itself and associated metadata, and allows tracking which venues attract the most attention for the tweet. Hence, a kilobyte or so space is sufficient to track the percentage of views from different locations, with an error of less than one percentage point, say.

Since their introduction over a decade ago [7], Count-min sketches have found applications in systems which track frequency statistics, such as tracking popularity of content within different groups: say, online videos among different sets of users, or which destinations are popular for nodes within a communications network. Sketches are used in telecommunications networks, where the volumes of data passing along links are immense, and are never stored. Summarizing the network traffic distribution allows hotspots to be detected, informing network planning decisions, and allowing configuration errors and floods to be detected and debugged [6]. Since the

sketch compactly encodes a frequency distribution, it can also be used to detect when there is a shift in popularities, as a simple example of anomaly detection.

4 Counting Distinct Items with Hyperloglog

Another basic problem is to keep track of how many different items have been seen, out of a large set of possibilities. For example, a web publisher might want to track how many different people have been exposed to a particular advertisement. Here, we do not want to count the same viewer more than once. When the number of possible items is not too large, keeping a list, or a binary array is a natural solution. As the number of possible items becomes very large, the space needed by these methods grows proportional to the number of items tracked. Switching to an approximate method such as a Bloom Filter means the space remains proportional to the number of distinct items, although the constants are improved.

Could we hope to do better? Well, if we just counted the total number of items, without removing duplicates, then a simple counter suffices, using a number of bits that is proportional to the logarithm of the number of items encountered. If only we had some way to know which items were new, and count only them, then we could achieve this cost. The Hyperloglog algorithm promises something even stronger: that the cost need depend only on the logarithm of the logarithm of the quantity computed! Of course, there are some scaling constants that mean the space needed is not quite so tiny as this might suggest, but the net result is that quantities can be estimated with high precision (say, up to 1 or 2% error) with a couple of kilobytes of space.

The Method. The essence of the method is to use hash functions applied to item identifiers to determine how to update counters, so that duplicate items are treated identically. A Bloom Filter has a similar property: if we attempt to insert an item that is already represented within a Bloom Filter, it means that we set a number of bits to 1 that are already recording 1 values. One approach we could use is to keep a Bloom Filter, and look at the final density of 1s and 0s to estimate the number of distinct items represented (taking into account collisions under hash functions). This still requires space proportional to the number of items, and is the basis of early approaches to this problem [15]. To break this linearity, we can think of a different approach to building a binary counter. Instead of adding 1 to the counter for each item, we could instead add 1 with probability $\frac{1}{2}$, 2 with probability $\frac{1}{4}$, 4 with probability $\frac{1}{8}$ and so on. This use of randomness decreases the reliability of the counter, but we can check that the expected count corresponds to the true number of items encountered. This makes more sense when we use hash functions: we apply a hash function g to each item i, with the same distribution: g maps items to j with probability 2^{-j}. We can then keep a set of bits indicating which j values have been seen so far. This is the essence of the early Flajolet-Martin approach to tracking the number of distinct items [8]. Here, we need a logarithmic number of bits, as there are only this many distinct j values that we expect to see.

The Hyperloglog method (HLL) reduces the number of bits further by only retaining the highest j value that has been seen when applying the hash function. We might expect this to be correlated to the cardinality, although with high variation – for example, there might be only a single item seen which happens to hash to a large value. To reduce this variation, the items are partitioned into groups using a second hash function (so the same item is always placed in the same group), and information about the largest hash in each group is retained. Each group yields an estimate of the local cardinality, which are all combined to obtain an estimate of the total cardinality. A first effort would be to take the mean of the estimates, but this still allows one large estimate to skew the result; instead, the harmonic mean is used to reduce this effect. It can be shown that by hashing to s separate groups, the standard error is proportional to $\frac{1}{\sqrt{s}}$. A small example is shown

We show a small example HLL sketch with $s = 3$ groups. Consider 5 distinct items a, b, c, d, e, with the following hash values:

x	a	b	c	d	e
$h(x)$	1	2	3	1	3
$g(x)$	0001	0011	1010	1101	0101

From this, we obtain the following array:

3	2	1

The estimate is obtained by taking 2 to the power each of the array entries, and computing the sum of the reciprocals of these values, obtaining $\frac{1}{8} + \frac{1}{4} + \frac{1}{2} = \frac{7}{8}$ in this case. The final estimate is made by multiplying $\alpha_s s^2$ by the reciprocal of this sum. Here, α_s is a scaling constant that depends on s. $\alpha_3 = 0.5305$, so we obtain 5.46 as our estimate – close to the true value of 5.

Figure 3: Example of Hyperloglog in action

in Figure 3. The analysis of the algorithm is rather technical, but the proof of the pudding is in the deployment: the algorithm has been widely adopted and applied in practice.

Discussion and Applications. One example of its use is in tracking the viewership of online advertising. Across many websites and different adverts, there may be trillions of view events every day. Advertisers are interested in the number of "uniques": how many different people (or rather, browsing devices) have been exposed to the content. Collecting and marshalling this data is not infeasible, but rather unwieldy, especially if it is desired to do more advanced queries (count how many uniques saw both of two particular adverts). Use of hyperloglog sketches allow this kind of query to be answered directly by combining the two sketches, rather than trawling through the full data. Sketches have been put into use for this purpose, where the small amount of uncertainty from the use of randomness is comparable to other sources of error, such as dropped data or measurement failure.

Approximate distinct counting is also widely used "behind the scenes" in web-scale systems. For example, in Google's Sawzall system, provides a variety of sketches, including count distinct, as primitives for log data analysis [13]. Google engineers have described some of the implementation modifications made to ensure high accuracy of the hyperloglog across the whole range of possible cardinalities [10]. A last interesting application of distinct counting is in the context of social network analysis. In 2016, Facebook set out to test the "six degrees of separation" claim within their social network. The Facebook friendship graph is sufficiently large (over a billion nodes and hundreds of billions of edges) that maintaining detailed information about the distribution of long range connections for each user would be infeasible. Essentially, the problem is to count, for each user, how many friends that have at distance 1, 2, 3 etc. This would be a simple graph exploration problem, but for the fact that some friends at distance 2 are reachable by multiple paths (via different mutual friends). Hence, distinct counting is used to generate accurate statistics on reachability without double counting, and to provide accurate dirstance distributions (the estimated number of degrees of separation in the facebook graph is 3.57) [2].

5 Advanced Sketching

Roughly speaking, the four examples of sketching described in this article cover most of the current practical applications of this model of data summarization. Yet, unsurprisingly, there is a large body of research into new applications and variations of these ideas. Just around the corner are a host of new techniques for data summarization which are currently on the cusp of practicality. This section mentions a few of the directions that seem most promising.

5.1 Sketching for Dimensionality Reduction

When dealing with large high-dimensional numerical data, it is common to seek to reduce the dimensionality while preserving fidelity of the data. Let's assume that the hard work of data wrangling and modeling is done, and the data can be modeled as a massive matrix, where each row is one example point, and each column encodes an attribute of the data. A common technique is to apply Principal Components Analysis (PCA) to extract a small number of 'directions' from the data. By projecting each row of data along each of these directions, we obtain a different representation of the data which captures most of the variation of the data set.

A limitation of PCA is that finding the direction represents quite a substantial amount of work. It requires finding eigenvectors of the covariance matrix, which rapidly becomes unsustainable for large matrices. The competing approach of random projections argues that rather than finding "the best" directions, it suffices to use a (slightly larger number) of random vectors. Picking a moderate number of random directions captures a comparable amount of variation, while requiring much less computation.

The random projection of each row of the data matrix can be seen as an example of a sketch of the data. But more directly, there are close connections between random projection and the sketches above. The Count-Min sketch can be viewed as a random projection of sorts; moreover, the best constructions of random projections for dimensionality reduction look a lot like Count-Min sketches with some twists (such as randomly multiplying each column of the matrix by a $+1$ or -1 factor). This is the basis of methods for speeding up high-dimensional machine learning, such as the Hash kernels approach [14].

5.2 Randomized Numerical Linear Algebra

A grand objective for sketching is to allow arbitrary complex mathematical operations over large volumes of data to be answered approximately and quickly via sketches. While this objective appears quite a long way off, and perhaps infeasible due to some impossibility results, there are a number of core mathematical operations that can be solved using sketching ideas, that lead to the notion of "randomized numerical linear algebra". A simple example is matrix multiplication: given two large matrices A and B, we want to find their product AB. An approach using sketching is to build a dimensionality reducing sketch of each row of A and each column of B. Combining each pair of these provides an estimate for each entry of the product. Similar to other examples, small answers are not well-preserved, but large entries are accurately found. Other problems that have been tackled in this space include regression. Here the input is a high dimensional data set modeled as a matrix A and column vector b: each row of A is a data point, with the corresponding entry of b the value associated with the row. The goal is to find regression coefficients x which minimize $\|Ax - b\|_2$. An exact solution to this problem is possible, but costly in terms of time as a function of the number of rows. Instead, we can apply sketching to matrix A, and solve

the problem in the lower dimensional sketch space [5]. The monograph by Woodruff provides a comprehensive mathematical survey of the state of the art in this area [16].

5.3 Rich Data: Graphs and Geometry

The applications of sketching so far can be thought of as summarizing data that might be thought of as a high dimensional vector, or matrix. These mathematical abstractions capture a large number of situations, but there are increasingly cases where we want a richer model of data: say, to model links in a social network (best thought of as a graph), or to measure movement patterns of mobile users (best thought of as points in the plane or 3D). Sketching ideas have been applied here also.

For graphs, there are techniques to summarize the adjacency information of each node, so that connectivity and spanning tree information can be extracted [1]. These methods provide a surprising mathematical insight that much edge data can be compressed while still preserving fundamental information about the graph structure. However, these techniques have not found significant use in practice yet, perhaps due to high overheads in the encoding size.

For geometric data, there has been much interest in solving problems such as clustering [9]. The key idea for clustering is that a clustering of part of the input can capture a lot of the overall structural information, and by merging clusters together ("clustering clusters") we still retain a good picture of the overall point density distribution.

6 Why should I care?

The aim of this article has been to introduce a selection of recent techniques that provide approximate answers to some general questions that often occur in data analysis and manipulation. In all cases, there are simple alternative approaches which provide exact answers, at the expense of keeping complete information. However, the examples shown have illustrated that there are many cases where the approximate approach can be faster and more space-efficient. The use of these methods is growing – it is sometimes claimed that Bloom Filters are one of the core technologies that "big data experts" must know. At the very least, it is important to be aware of sketching techniques to test claims that solving a problem a certain way is the only option – often, there are fast approximate sketch-based techniques that provide a different tradeoff.

Acknowledgements

I thank S. Muthukrishnan and Peter Bailis for advice and encouragement in preparing this article.

The work of GC is supported in part by European Research Council grant ERC-2014-CoG 647557, The Alan Turing Institute under the EPSRC grant EP/N510129/1, Jaguar Land Rover under the EPSRC grant EP/N012380/1, and a Royal Society Wolfson Research Merit Award.

References

[1] Kook Jin Ahn, Sudipto Guha, and Andrew McGregor. Analyzing graph structure via linear measurements. In *ACM-SIAM Symposium on Discrete Algorithms*, 2012.

[2] Smriti Bhagat, Moira Burke, Carlos Diuk, Ismail Onur Filiz, and Sergey Edunov. Three and a half degrees of separation. `https://research.fb.com/three-and-a-half-degrees-of-separation/`, February 2016.

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

[4] Michael Broder, Andrei; Mitzenmacher. Network applications of Bloom filters: A survey. *Internet Mathematics*, 1(4):485–509, 2005.

[5] Kenneth L. Clarkson and David P. Woodruff. Low rank approximation and regression in input sparsity time. In *ACM Symposium on Theory of Computing*, pages 81–90, 2013.

[6] G. Cormode, F. Korn, S. Muthukrishnan, T. Johnson, O. Spatscheck, and D. Srivastava. Holistic UDAFs at streaming speeds. In *ACM SIGMOD International Conference on Management of Data*, pages 35–46, 2004.

[7] G. Cormode and S. Muthukrishnan. An improved data stream summary: The Count-Min sketch and its applications. *Journal of Algorithms*, 55(1):58–75, 2005.

[8] P. Flajolet and G. N. Martin. Probabilistic counting. In *IEEE Conference on Foundations of Computer Science*, pages 76–82, 1983. Journal version in *Journal of Computer and System Sciences*, 31:182–209, 1985.

[9] S. Guha, N. Mishra, R. Motwani, and L. O'Callaghan. Clustering data streams. In *IEEE Conference on Foundations of Computer Science*, 2000.

[10] Stefan Heule, Marc Nunkesser, and Alex Hall. Hyperloglog in practice: Algorithmic engineering of a state of the art cardinality estimation algorithm. In *International Conference on Extending Database Technology*, 2013.

[11] Chris Jermaine. Sampling techniques for massive data. In Graham Cormode, Minos Garofalakis, Peter Haas, and Chris Jermaine, editors, *Synposes for Massive Data: Samples, Histograms, Wavelets and Sketches*, Foundations and Trends in Databases. *NOW* publishers, 2012.

[12] Robert Morris. Counting large numbers of events in small registers. *Communications of the ACM*, 21(10):840–842, 1977.

[13] Rob Pike, Sean Dorward, Robert Griesemer, and Sean Quinlan. Interpreting the data: Parallel analysis with sawzall. *Dynamic Grids and Worldwide Computing*, 13(4):277–298, 2005.

[14] Kilian Q. Weinberger, Anirban Dasgupta, John Langford, Alexander J. Smola, and Josh Attenberg. Feature hashing for large scale multitask learning. In *International Conference on Machine Learning (ICML)*, 2009.

[15] K. Y. Whang, B. T. Vander-Zanden, and H. M. Taylor. A linear-time probabilistic counting algorithm for database applications. *ACM Transactions on Database Systems*, 15(2):208, 1990.

[16] David Woodruff. Sketching as a tool for numerical linear algebra. *Foundations and Trends in Theoretical Computer Science*, 10(1-2):1–157, 2014.