School of IT
Technical Report

The University of Sydney

AN INCREMENTAL DATA-STREAM SKETCH USING SPARSE RANDOM PROJECTIONS TECHNICAL REPORT 609

ADITYA KRISHNA MENON
ANH PHAM
SANJAY CHAWLA
ANASTASIOS VIGLAS

SCHOOL OF IT, THE UNIVERSITY OF SYDNEY

JANUARY, 2007

An incremental data-stream sketch using sparse random projections

Aditya Krishna Menon* Anh Pham† Sanjay Chawla‡ Anastasios Viglas§

Abstract

We propose the use of random projections with a sparse matrix to maintain a sketch of a collection of high-dimensional data-streams that are updated asynchronously. This sketch allows us to estimate L_2 (Euclidean) distances and dot-products with high accuracy. We verify the validity of this sketch by applying it to an online clustering problem, where we compare our results to the offline algorithm and an existing L_2 sketch, and observe comparable results in terms of accuracy, and a reduced runtime cost.

1 Introduction.

1.1 Data-streams. A data stream, as the metaphor suggests, is the continuous flow of data generated at a source (or multiple sources) and transmitted to various destinations. Some examples of data streams are the flow of data in the Internet, environmental data collection(through satellite and land-based stations), financial markets, telephony, etc. Since the number of applications that generate data streams are growing, there is a need for computational methods that can efficiently, and accurately, carry out sophisticated analysis on such sources of data.

Data-stream mining (DSM) and data-stream management systems (DSMS) are active areas of research which generalize and adapt existing techniques to handle streaming data. For example, in traditional database management systems (DBMS) the data is assumed to be fixed and persistent, and the set of queries on the database is assumed to be changing. But in contrast, the DSMS view is that the set of queries is fixed, and data is continuously flowing through the queries [4, 6]. In DSM, the focus has been on efficient and incremental learning, i.e. a model is learnt on a batch of data, and as new data arrives, the model has to be updated to accommodate the new data without simply relearning from scratch (as this would be too expensive). Further, the effect of "old" data has to be discounted from the model in order for it to be an accurate reflection reality [2, 12].

*The University Of Sydney.
†The University Of Sydney.
‡The University Of Sydney.
§The University Of Sydney.

However, most data-stream frameworks for data-mining assume that data arrives incrementally and sequentially (in-order), and develop solutions for this case. But in many situations, the data not only arrives incrementally, but also *asynchronously* - that is, at any point in time, we may be asked to update any arbitrary component or attribute of the existing data. The stream here cannot be mined with a one-pass algorithm, since the update of components is not (necessarily) sequential. Therefore, the challenge is to carry out our data mining task without having to instantiate the data vector - which may not be possible or computationally feasible. We explain this situation with the help of an example.

1.2 Prototype application. Consider a group of stock traders who trade on several equities listed in several stock markets. At a given moment in time, we associate a stock vector $s(t) = [v_1, v_2, \ldots, v_d]$ for each trader s. Here, v_i is the size of the holding for stock i, and d is number of all the stocks we consider (size of the universe of stocks). Now, the updates for each trader s arrive as tuples (j_s, u_s), where j_s is the index of the stock vector, and u_s is the value of the update. For example, if the stock vector represents the monetary value of stocks, then u_s can be positive or negative dollar value depending upon whether the particular stock j_s was bought or sold. The actual update is performed as

$$v_i(t+1) = \begin{cases} v_i(t) + u_s & \text{if } i_s = j_s \\ v_i(t) & \text{otherwise} \end{cases}$$

Note that the fact that u_s is negative makes this a turnstile-model of data streams [20].

Now, at a given point in time we would like to cluster the stock traders in real time, which will give us information about traders who are behaving similarly, or those who have moved from one cluster to another. The challenge is to do this efficiently, given that the stream is constantly being updated and that there is limited space. Theoretically, the stock vector can be very large, as large as the size of the union of all stocks that are traded in the world. So, if we want to use the standard k-means algorithm for clustering, then the complexity of each iteration is $O(nkd)$ where n, k and d are the number of stock traders, number of clusters and

number of stocks respectively. As we will show below, using special data stream sketches, we can reduce the complexity to $O(nkD)$ where D is typically $O(\log n \frac{1}{\epsilon^2})$, and ϵ is the accuracy that we want to achieve.

1.3 Problem definition. Consider a collection of objects $\{\mathbf{u_1}, \mathbf{u_2}, \ldots, \mathbf{u_n}\}$, each with dimension d. A natural representation of this data is as an $n \times d$ matrix

$$A = \begin{bmatrix} \mathbf{u_1} \\ \mathbf{u_2} \\ \vdots \\ \mathbf{u_n} \end{bmatrix}$$

Suppose further that this matrix is updated asynchronously at various points in time, by updates of the form

$$(i, j, c)$$

which instructs us to update row i, column j with the value c (which may be negative). We can therefore consider A as holding n streams that may be updated at any time i.e. asynchronously, and denote the value of A at time t by $A(t)$. Therefore, $A(t)$ represents the state of each of our n streams at time t.

Now, in some problems, the value of d is very large (e.g. study of internet routers, stocks, etc.), in fact so large as to make instantiation of the matrix infeasible. However, we would still like to be able to extract some useful information about the data; for example, we might want to estimate the L_2 norm (Euclidean distance) of the individual rows/streams of the data, or perhaps the dot-product between streams.

To make such estimates, a *sketch* of the streams is usually kept. A sketch is some approximation of a stream (or a series of streams) that has two desirable properties:

1. It occupies very little space
2. It allows us to give quick and accurate answers to queries on some quantity associated with the stream

A Euclidean distance-preserving sketch $\mathbf{c}$ for a stream $\mathbf{S}$, then, is some vector $\mathbf{c}(\mathbf{S})$ that occupies very little space and allows us to estimate $||\mathbf{S}||^2$ quickly and accurately.

1.4 Our contribution. There have been several approaches already proposed to estimate the L_p norm for a range of values of p, and in particular, for the case $p = 2$ which gives us an estimate for the L_2 norm. One such approach is that of Indyk [14], which requires the generation of variables from a Gaussian distribution to estimate the L_2 norm of a stream. We explicitly show that this sketch can be viewed in the context of random projections [21], and that by doing so we can access the theory of random projections to improve the efficiency of the sketch.

Specifically, we propose the use of random projections with a sparse matrix, defined by Achlioptas [1], whose entries can be generated from a uniform distribution to maintain a sketch of a collection of high-dimensional streams that are updated asynchronously. This sketch allows us to estimate L_2 (Euclidean) distances and dot-products with high accuracy. The advantage of this approach is simplicity, and efficiency - since our projections are with a sparse matrix, and also since we do not have to generate Gaussian variables, our approach is faster.

Further, we show how the results on L_2 preservation allow us to estimate dot-products as well, and derive bounds on the maximal error of our estimate. We derive a similar result for Indyk's L_2 sketch, which, to our knowledge, has not been made before.

We verify the validity of our projection-based sketch by applying it to an online clustering problem, where we have to cluster high-dimensional data streams that are updated incrementally at some given points in time. We compare our results to the offline algorithm Indyk's L_2 sketch, and observe comparable results in terms of accuracy, and an improved runtime.

1.5 Related work.

1.5.1 L_2-approximating sketches. In [3], it was shown how to estimate frequency-moments F_k of a stream, where $F_k(\mathbf{a}) := \sum a_i^k$. In particular, for the case F_2, an efficient one-pass algorithm was proposed, using $O\left(\frac{1}{\epsilon^2}\right)$ space and giving a $1 \pm \epsilon$ approximation. However, this does not allow for asynchronous updates - if we think of the pass as a series of sequential updates to the components of the stream, then once we have updated the value of a particular component, we cannot go back and change it later.

An important sketch for preserving L_2 distance is the work of Indyk [14], who proposed the use of a p-stable distribution in order to estimate the L_p norm for arbitrary $p \in (0, 2]$,

$$||\mathbf{x}||_p := \left(\sum_i |x_i|^p \right)^{1/p}$$

For $p = 2$, this requires the use of Gaussian random variables to maintain a sketch. The method used is essentially a random projection, though there are some minor differences regarding the way estimates are

made; we look at the connection in more depth in §4. Surprisingly, while it has been noted that this sketch is inspired by random projection style embeddings, to our knowledge there has not been any work done using a sparse random matrix for the projection.

Since our work explicitly uses random projections as defined in the literature, we are able to access the theory of these projections. So, for instance, our bounds on the reduced dimension follow immediately from the Johnson-Lindenstrauss lemma [15], and our analysis does not have to deal with the accuracy of projection-based sketches. More importantly, we are able to justify the use of a sparse projection matrix with no loss in accuracy, and improved efficiency.

Of course, the L_p sketch offers us the ability to estimate L_p distances for general $p \in (0, 2]$, which makes it fairly versatile. An example of an interesting application of this sketch that our projection-based sketch cannot be used for is the estimation of the Hamming norm of streams [7].

1.5.2 Stream clustering. An important paper in the field of clustering of data streams is the work of [13], which provides an algorithm for doing clustering of a stream of data using only $O(nk)$ time, where k is the number of clusters. This however does not focus on a stream with asynchronous updates, and instead provides a one-pass algorithm for a stream where we get new points that we have to cluster, but not updates to existing points.

In [10], random projections were used to cluster data using the EM algorithm. It was found that the results from the use of a single projection were unsatisfactory, and it was recommended that an *ensemble* of projections be used to try and overcome stability issues related to projections. Our concern is not so much with the accuracy of clustering, but rather with the accuracy of our sketch as compared the "true" value of the underlying stream. Whether the projections produce a bad clustering or not is dependent on the nature of the clustering algorithm e.g. if it is extremely sensitive to perturbations in the data/distances between points. An ensemble-based approach might be a natural extension to the sketch we propose here.

2 Background.

Random projections are a powerful method of dimensionality reduction that have been applied in numerous practical problems [5, 17, 11], and have also served as a useful tool in algorithmic design [21]. The basis of projections is the remarkable Johnson-Lindenstrauss lemma [15], which say that for any data set of n points in d dimensions, and some error parameter ϵ, it is possible to find a mapping $f : \mathbb{R}^{n\times d} \to \mathbb{R}^{n\times k}$ such that

- $k = O\left(\frac{\log n}{\epsilon^2}\right)$
- All pairwise distances between points are preserved within a factor of ϵ

The original statement of the lemma was non-constructive, and did not mention how one would find such an f in general. There has been subsequent work on finding such an f in $O\left(dn^2(\log n + \frac{1}{\epsilon})^{O(1)}\right)$ time [9], but in practise, most work on projections involves finding an f that gives us approximate Johnson-Lindenstrauss guarantees, that is, where pairwise distances are preserved with high probability.

We can think of projections as a multiplication by a suitably chosen random matrix (possibly with some appropriate scaling):

$$A \mapsto E = A.R$$

where R is a matrix whose entries follow some random distribution.

The "classic" form of random projections used a matrix whose entries were i.i.d. $\mathcal{N}(0, 1)$, which yields fairly tight tail bounds [8]. Achlioptas [1] however showed that a much simpler distribution would suffice, and derived strong bounds for a matrix where each entry has distribution

$$r_{ij} = \begin{cases} +\sqrt{3} & p = 1/6 \\ 0 & p = 2/3 \\ -\sqrt{3} & p = 1/6 \end{cases}$$

We henceforth call this matrix the Achlioptas matrix. This matrix has an expected sparsity of $\frac{2}{3}$rds, making it quite desirable to use in practise. In fact, this result was recently generalized in [16], where the sparsity of the projection matrix was increased to being potentially $1 - \frac{1}{\sqrt{d}}$.

3 Random projections and streams.

3.1 Projection-based sketch. Suppose we have a data matrix $A(t) \in \mathbb{R}^{n\times d}$. Now, suppose that updates happen to this matrix in the following way. At time t, we get a tuple $d(t) = (i_t, j_t, c_t)$. When this happens,

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

That is, the entry in cell (i_t, j_t) gets incremented by the quantity c_t. We can express this as

$$A(t) = A(t - 1) + C(t)$$

where $C(t) = \{C_{mn}\}$, such that $C_{mn} = c_t \delta_{i_t m} \delta_{j_t n}$ (where δ denotes the Kronecker delta function).

If d is large, it might be infeasible instantiate the matrix A in memory. But we still want to keep the information associated with it, and we can try to do this with the help of a projection. Suppose we keep the projected matrix $E(t) \in \mathbb{R}^{n \times k}$, where k is a more manageable number. If we generate $E(t)$ via a random projection, then we have

$$E(t) = \frac{1}{\sqrt{k}} A(t) R$$

where R is, say, the Achlioptas matrix.

But of course, $R \in \mathbb{R}^{d \times k}$, which is also comparable with A in terms of size. So, we can't really store a single random matrix R and use it for each of our projections. What can we do? An immediate idea is to try and do the generation on the fly. We show that such an approach is not likely to work, and show instead a different method to get around the space restriction of R.

3.2 On-the-fly random matrix generation. One option is to generate the matrix at every time step; of course, we don't generate the whole matrix due to space constraints, but only go one component at a time for the matrix multiplication. Still, for analysis' sake, let us assume we can remember the whole matrix for a time-step at least. Therefore, we have

$$E(t) = A(t)R(t)$$

Suppose that initially, $A(0) \equiv \mathbf{0}$ - we have no information about any component at all. It is clear then that, based on the nature of the updates, we must have

$$A(t) = \sum_{n=1}^{t} C(n)$$

The projection of this matrix at time t will therefore be

$$E(t) = \sum_{n=1}^{t} C(n)R(t)$$

What happens the next time we get a tuple? Ideally, we would like to use the existing $E(t)$ matrix, and update it rather than doing a full recomputation. We can see that the expression for the new projected matrix is:

$$\begin{aligned} E(t+1) &= \sum_{n=1}^{t+1} C(n)R(t+1) \\ &= C(t+1)R(t+1) + \sum_{n=1}^{t} C(n)R(t+1) \\ &= C(t+1)R(t+1) + E(t) \\ &+ \sum_{n=1}^{t} C(n)(R(t+1) - R(t)) \\ &= E(t) + C(t+1)R(t+1) \\ &+ \sum_{n=1}^{t} C(n)S(t+1) \ , \ S(t) := R(t) - R(t-1) \end{aligned}$$

3.3 Naive approximation. Certainly one simple approximation for $E(t+1)$, based on the previous section, is

$$E(t+1) \approx E(t) + C(t+1)R(t+1)$$

This is easy to compute, but how bad is this as an approximation? The error at time $t+1$ will be

$$\delta(t+1) = \sum_{n=1}^{t} C(n)S(t+1)$$

To get a sense of the behaviour of this variable, suppose $X = r_{ij} - s_{ij}$, where r, s are from the Achlioptas distribution. Clearly, the quantity $S(t)$ must follow this distribution. By independence and linearity, it must follow that

$$\begin{aligned} E(X) &= 0 \\ Var(X) &= 2 \end{aligned}$$

Therefore, we get

$$\begin{aligned} E(\delta(t)_{ij}) &= 0 \\ Var(\delta(t)_{ij}) &= 2\left(\sum C(n)\right)^2 \end{aligned}$$

So, while the expected error is zero, there is potentially quite a large variance, which only gets worse as t gets larger!

3.4 Repeatable generation. We have seen that we really do need to generate the same random matrix at every time-step, because otherwise the errors obtained get far too large. Before we suggest how to solve this problem, we observe that the matrix multiplication involved in the update step simplifies to the manipulation of a single vector alone.

3.4.1 Observation on multiplication by C. The fact that there are many zeros in the C matrix naturally suggests that we are saved a lot of work, and this is precisely the case. The situation we have is merely the following:

$$C(t).R(t) = \begin{bmatrix} \mathbf{0} \\ \mathbf{0} \\ \vdots \\ \mathbf{c_t} \\ \mathbf{0} \\ \vdots \end{bmatrix} [\mathbf{r_1}(t) \quad \mathbf{r_2}(t) \quad \dots \quad \mathbf{r_k}(t)]$$

$$= c_t \begin{bmatrix} 0 & 0 & \dots & 0 \\ 0 & 0 & \dots & 0 \\ \vdots & \vdots & \dots & \vdots \\ r_{j_t 1}(t) & r_{j_t 2}(t) & \dots & r_{j_t k}(t) \\ \vdots & \vdots & \dots & \vdots \\ 0 & 0 & \dots & 0 \end{bmatrix}$$

where

$$\mathbf{c_t} = [0 \quad 0 \quad \dots \quad c_t \quad 0 \quad \dots 0]$$

So, from the point of view of one update, we only need to compute k random numbers $r_{j_t i}(t)$. That is, every update only requires the generation of a single random row. From the point of view of this single update, it does not matter what the other elements of $R(t)$ are, since they are all cancelled by the zeros!

3.4.2 Pseudo-random generator. We can see that for a single update, we just need to generate $\Theta(k)$ random numbers - this is very much possible. The problem is that we want to generate the *same* random numbers each time we refer to the same row.

To get repeatable generation of the random rows, we use a pseudo-random number generator for generating the row values. Suppose that row i is generated using a seed $s(i)$ - then certainly if we fix $s(i)$ to be constant at all times, we can be guaranteed to get the same random row each time. However, since there are d rows, if we wanted to store d seeds we would $d \log M$ bits of space, where M is the maximal seed value. In many cases, this might be comparable to simply storing the original data matrix.

But we can get around this in a very simple way, because the seed can be set to be the original column index, or the random row index, j_t [14]. With this seed, the pseudo-random number generator generates for us k values corresponding to the random row j_t. Since at times t_1 and t_2 where we refer to the same row, $j_{t_1} \equiv j_{t_2}$ trivially, we will generate the same random row.

With this method of generating R, the result of the earlier section reduces to the exact equation

$$E(t+1) = E(t) + C(t+1)R$$

This means that the update to the projected matrix is very simple - we just have to project the matrix $C(t+1)$ down (which is easy), and add this to our existing sketch.

Here, R only exists conceptually - to generate it one row at a time, we just use the above pseudo-random number generator with seed j_t. The above section on matrix multiplication shows that we really just need to generate a random row at each iteration, and further that this can be done on-the-fly, requiring $\Theta(1)$ space.

3.5 Complexity and comments. Recall that our sketch-update is

$$E(t+1) = E(t) + C(t+1)R$$

From the result of §3.4.1, we see that the matrix multiplication takes $\Theta(k)$ time. Hence, using random projections, we can keep a matrix $E(t)$ with space $\Theta(nk)$ that serves as an approximation to the original matrix $A(t)$, where updates take $\Theta(k)$ time (assuming that the matrix-row addition can be done in $\Theta(k)$ time as well - note that we don't have to update every row of $E(t)$, just row i_t).

Further, at any time t, the matrix $E(t)$ corresponds exactly with the matrix $A(t).R$, for some suitably chosen random-projection matrix R (that is, $E(t)$, which is incrementally updated, is the same as if we had done the expensive projection of the matrix $A(t)$ offline).

By projecting using a suitable k for the reduced dimension, we get the usual pairwise-distance preservation properties. The Johnson-Lindenstrauss lemma asserts that we only need space $nk = O\left(\frac{n \log n}{\epsilon^2}\right)$ if we want to achieve at most ϵ distortion in our L_2 estimation.

3.6 The preservation of dot-product.

3.6.1 Dot-product behaviour under projections. While distances have been the primary property of interest in the theory of random projections, the dot-product has also received some attention, since it is a useful quantity that arises quite commonly in practise. It was shown in [16] that

$$E(\mathbf{v_1}.\mathbf{v_2}) = \mathbf{u_1}.\mathbf{u_2}$$

$$Var(\mathbf{v_1}.\mathbf{v_2}) = \frac{||\mathbf{u_1}||^2||\mathbf{u_2}||^2 + (\mathbf{u_1}.\mathbf{u_2})^2}{k}$$

where $\mathbf{v_i}$ is the projection of $\mathbf{u_i}$, and k is the reduced dimension.

This says that while on average projections are expected to preserve the dot-product, there is a large variance in the approximation, which means that the dot-product estimate is not a tight one. The reason for this is the fact that as the dot-product in the original space tends to zero, the dot-product in the new space does not correspondingly tend to zero. To see this, notice that

$$Var\left(\frac{\mathbf{v_1}.\mathbf{v_2}}{\mathbf{u_1}.\mathbf{u_2}}\right) = \frac{\sec^2\theta + 1}{k}$$

using the fact that $\mathbf{u_1}.\mathbf{u_2} = ||\mathbf{u_1}||||\mathbf{u_2}||\cos\theta$. This says that as $\theta \to \frac{\pi}{2}$, that is, as $\mathbf{u_1}.\mathbf{u_2} \to 0$, the variance of the ratio of the two dot-products becomes unbounded; this means that the projected dot-product does not tend to zero along with the original dot-product.

As a result, the dot-product is not "well-behaved" under random projections, because we cannot uniformly guarantee its preservation. We can verify this fact by plotting the mean error in the projected dot-product versus the original value of the dot-product. We ran a simple simulation in MATLAB for a 1000×1000 matrix, and we obtained the result shown in Figure 1. We can see that when the original dot-product tends to zero, the error in our projection-based estimate starts to increase without bound.

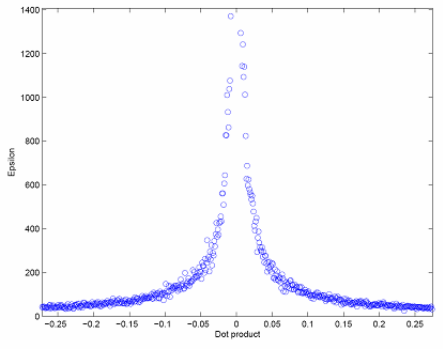

Figure 1: The error parameter ϵ in the dot-product estimate provided by a random projection can be seen to increase without bound as the dot-product in the original space tends to zero

3.6.2 Bound on dot-product error The previous section indicates that the dot-product is not "well-behaved", and so we cannot expect to have as tight guarantees on its preservation as we do for distances. However, it is still of interest as to what sort of guarantees we can place on the error incurred.

In the following section, we take $A = \{\mathbf{u_i}\}_{i=1\ldots n}$ to represent an arbitrary collection of d-dimensional vectors, and $\{\mathbf{v_i}\}_{i=1\ldots n}$ to be the corresponding random projections into $k = O(\log n/\epsilon^2)$ dimensions, where ϵ is the error parameter for distances. We are able to derive that, with high probability,

$$|\mathbf{v_i}.\mathbf{v_j} - \mathbf{u_i}.\mathbf{u_j}| \leq \frac{\epsilon}{2}(||\mathbf{u_i}||^2 + ||\mathbf{u_j}||^2) \tag{3.1}$$

To see this, we use the fact that

$$\mathbf{x}.\mathbf{y} = \frac{||\mathbf{x}+\mathbf{y}||^2 - ||\mathbf{x}-\mathbf{y}||^2}{4}$$

Now, by the theory of random projections, it is certainly true that with high probability:

$$(1-\epsilon)||\mathbf{u_i} - \mathbf{u_j}||^2 \leq ||\mathbf{v_i} - \mathbf{v_j}||^2 \leq (1+\epsilon)||\mathbf{u_i} - \mathbf{u_j}||^2$$

We can show a related result as well. Suppose that we have our data set A. Form the data-set B with $2n$ points, consisting of the points in A and their negatives: $B = \{\mathbf{u_i}, -\mathbf{u_i}\}$. Our projection guarantees tell us that, with high probability, we must have

$$(1-\epsilon)||\mathbf{u_i} + \mathbf{u_j}||^2 \leq ||\mathbf{v_i} + \mathbf{v_j}||^2 \leq (1+\epsilon)||\mathbf{u_i} + \mathbf{u_j}||^2$$

since $\log(2n) = O(\log n)$, meaning we are still reducing to an $O(\log n/\epsilon^2)$ subspace. Note that the independence of the random rows means that we don't really have to project the set B down to k dimensions - we can just project A, and provided k is large enough, the guarantees must still hold anyway.

This implies that we can get the bound

$$\begin{aligned}
&(1-\epsilon)||\mathbf{u_i} + \mathbf{u_j}||^2 - (1+\epsilon)||\mathbf{u_i} - \mathbf{u_j}||^2 \leq \\
&||\mathbf{v_i} + \mathbf{v_j}||^2 - ||\mathbf{v_i} - \mathbf{v_j}||^2 \leq \\
&(1+\epsilon)||\mathbf{u_i} + \mathbf{u_j}||^2 - (1-\epsilon)||\mathbf{u_i} - \mathbf{u_j}||^2 \\
\implies &(1-\epsilon)\frac{||\mathbf{u_i} + \mathbf{u_j}||^2}{4} - (1+\epsilon)\frac{||\mathbf{u_i} - \mathbf{u_j}||^2}{4} \leq \\
&\frac{||\mathbf{v_i} + \mathbf{v_j}||^2}{4} - \frac{||\mathbf{v_i} - \mathbf{v_j}||^2}{4} \leq \\
&(1+\epsilon)\frac{||\mathbf{u_i} + \mathbf{u_j}||^2}{4} - (1-\epsilon)\frac{||\mathbf{u_i} - \mathbf{u_j}||^2}{4} \\
\implies &\mathbf{u_i}.\mathbf{u_j} - \frac{\epsilon}{4}(||\mathbf{u_i} - \mathbf{u_j}||^2 + ||\mathbf{u_i} + \mathbf{u_j}||^2) \leq \\
&\mathbf{v_i}.\mathbf{v_j} \leq \\
&\mathbf{u_i}.\mathbf{u_j} + \frac{\epsilon}{4}(||\mathbf{u_i} - \mathbf{u_j}||^2 + ||\mathbf{u_i} + \mathbf{u_j}||^2)
\end{aligned}$$

From here, we can say

$$\begin{aligned}
&|\mathbf{v_i}.\mathbf{v_j} - \mathbf{u_i}.\mathbf{u_j}| \leq \frac{\epsilon}{4}(||\mathbf{u_i} - \mathbf{u_j}||^2 + ||\mathbf{u_i} + \mathbf{u_j}||^2) \\
\implies &|\mathbf{v_i}.\mathbf{v_j} - \mathbf{u_i}.\mathbf{u_j}| \leq \frac{\epsilon}{2}(||\mathbf{u_i}||^2 + ||\mathbf{u_j}||^2)
\end{aligned}$$

So, we see that with high probability,

$$|\mathbf{v_i}.\mathbf{v_j} - \mathbf{u_i}.\mathbf{u_j}| \leq \frac{\epsilon}{2}(||\mathbf{u_i}||^2 + ||\mathbf{u_j}||^2)$$

4 Comparison with L_p sketch.

4.1 The L_p sketch. In [14], it was proposed to use a p-stable distribution in order to estimate the L_p norm,

$$||\mathbf{x}||_p := \left(\sum_i |x_i|^p\right)^{1/p}$$

In particular, [14] deals with the problem of a single vector $\mathbf{a}$ that gets updates of the form (i_t, a_t) at time t, indicating that index i_t should be updated by the value a_t. We can therefore represent the update vector as

$$C(t) = [0 \quad 0 \quad \dots \quad a_t \quad 0 \quad \dots \quad 0]$$

and thus the state of the true/underlying vector A at time t is just

$$\mathbf{a}(t) = \sum_{n=1}^{t} C(n)$$

A sketch of this vector, $\mathbf{s}(t)$, is maintained using a matrix

$$X = [\mathbf{x_1} \quad \mathbf{x_2} \quad \dots \quad \mathbf{x_k}]$$

where $\mathbf{x_i}$ is a $d \times 1$ column vector, the entries x_{ij} come from a p-stable distribution Z:

$$\sum a_i Z_i \sim \left(\sum |a_i|^p\right)^{1/p} Z$$

and the value k is shown to be $O(\log \frac{1}{\delta}/\epsilon^2)$ for giving $1 \pm \epsilon$ accurate estimates with probability at least $1 - \delta$.

At every update, the sketch is updated with

$$\mathbf{s}(t+1) = \mathbf{s}(t) + a_{t+1}\mathbf{r}(t+1)$$

where

$$\mathbf{r}(t+1) = [X_{i_{t+1}1} \quad X_{i_{t+1}2} \quad \dots \quad X_{i_{t+1}k}]$$

The L_p approximation is given by

$$\text{est}_{L_p}(||\mathbf{a}||^p) = \frac{\text{median}(s_1^2, s_2^2, \dots, s_k^2)}{\text{median}(|Z|^p)}$$

where Z is a variable from a p-stable distribution, and this is guaranteed to be a $(1 \pm \epsilon)$ approximation with probability at least $1 - \delta$. The key fact here is that we can get strong guarantees on the closeness of the median of the components of the sketch to the norm of the original stream/vector.

So, for $p = 2$, this means we can estimate the L_2 norm using a 2-stable distribution, such as a Gaussian distribution. As a result, we are essentially doing a projection with a matrix whose entries are i.i.d. Gaussian variables, which is a classical random projection matrix. However, one minor difference exists, namely, the lack of the $\frac{1}{\sqrt{k}}$ scaling we normally associate with projections. This is because the estimate of the L_2 norm is given by

$$\text{est}_{L_2}(||\mathbf{a}||^2) = \frac{\text{median}(s_1^2, s_2^2, \dots, s_k^2)}{\text{median}(Z^2)}$$

where $Z \sim \mathcal{N}(0, 1)$. This differs from the projection estimate, which is the explicit computation of the norm:

$$\text{est}_{\text{proj}}(||\mathbf{a}||^2) = ||\mathbf{s}||^2$$

but gives the same guarantee, namely that with probability at least $1 - \delta$, we have

$$(1 - \epsilon)\text{est}(||\mathbf{a}|^2) \leq ||\mathbf{a}||^2 \leq (1 + \epsilon)\text{est}(||\mathbf{a}|^2) \tag{4.2}$$

4.2 Advantage of sparse projections. Our insight is that the computation of 2-stable values is *not* essential to make accurate estimates of the L_2 norm of a stream, and that we can use the simpler Achlioptas matrix, as outlined in §2, to do our projection. This has negligible impact on the accuracy of the sketch, and leads to greatly improved performance. One potential problem with using 2-stable distributions in this streaming context is that we are required to regenerate the random rows every time there is an update - were it possible to store the random matrix once at the start, there would likely not be much difference between the two approaches. However, since every regeneration of a random row requires the generation of $\Theta(k)$ Gaussian variables, in streaming problems the effect of this cost can become noticeably large after long periods of time. Of course, depending on implementation, the precise difference between generating Gaussians and uniform random variables may differ, but with a high-volume of updates, even a small difference can prove to be very costly.

Further, the update itself is simplified by the fact that the Achlioptas matrix is, on average, $\frac{2}{3}$rds sparse - this means that we essentially don't have to do $\frac{2}{3}$rds of the multiplication of the update value with our random row (whereas for the Gaussian case, we would have to multiply this with every element in our random row of size k).

4.3 Deriving dot-product estimate. We can use the L_p sketch to get a dot-product estimate, as with

projections. Note that we can write the dot-product using norms:

$$\mathbf{x}.\mathbf{y} = \frac{||\mathbf{x}+\mathbf{y}||^2 - ||\mathbf{x}-\mathbf{y}||^2}{4}$$

This motivates an initial estimate for the dot-product,

$$\widehat{\text{est}}(\mathbf{u_i}.\mathbf{u_j}) := \frac{\text{est}(||\mathbf{u_i}+\mathbf{u_j}||^2) - \text{est}(||\mathbf{u_i}-\mathbf{u_j}||^2)}{4}$$

Recall that by Equation 4.2, we have

$$(1-\epsilon)\text{est}(||\mathbf{u_i}|^2) \leq ||\mathbf{u_i}||^2 \leq (1+\epsilon)\text{est}(||\mathbf{u_i}|^2)$$

Using this result and the linearity of the sketch, we can bound the error on our estimate:

$$\begin{aligned}\widehat{\text{est}}(\mathbf{u_i}.\mathbf{u_j}) &\leq \frac{1}{4}\left(\frac{||\mathbf{u_i}+\mathbf{u_j}||^2}{(1-\epsilon)} - \frac{||\mathbf{u_i}-\mathbf{u_j}||^2}{(1+\epsilon)}\right)\\ &= \frac{1}{4}\left(\frac{2\epsilon(||\mathbf{u_i}||^2+||\mathbf{u_j}||^2)}{(1-\epsilon^2)} + \frac{4\mathbf{u_i}.\mathbf{u_j}}{(1-\epsilon^2)}\right)\\ &= \frac{\epsilon}{(1-\epsilon^2)}\frac{||\mathbf{u_i}||^2+||\mathbf{u_j}||^2}{2} + \frac{\mathbf{u_i}.\mathbf{u_j}}{(1-\epsilon^2)}\end{aligned}$$

Similarly, we can derive the lower bound

$$\widehat{\text{est}}(\mathbf{u_i}.\mathbf{u_j}) \geq -\frac{\epsilon}{(1-\epsilon^2)}\frac{||\mathbf{u_i}||^2+||\mathbf{u_j}||^2}{2} + \frac{\mathbf{u_i}.\mathbf{u_j}}{(1-\epsilon^2)}$$

Now define the estimate of the dot-product to be

$$\text{est}(\mathbf{u_i}.\mathbf{u_j}) := (1-\epsilon^2)\widehat{\text{est}}(\mathbf{u_i}.\mathbf{u_j})$$

This gives us the following high-probability bound on the error of our estimate:

$$|\text{est}(\mathbf{u_i}.\mathbf{u_j}) - \mathbf{u_i}.\mathbf{u_j}| \leq \frac{\epsilon}{2}(||\mathbf{u_i}||^2+||\mathbf{u_j}||^2)$$

which is the same error bound as with projections (Equation 3.1), since for projections we have

$$\text{est}(\mathbf{u_i}.\mathbf{u_j}) := \mathbf{v_i}.\mathbf{v_j}$$

Therefore, for both sketches, we have the same high-probability upper bound on the error incurred using a simple dot-product estimate.

5 Experiments.

5.1 Time for generation of Gaussian variables. It is intuitive that the generation of a Gaussian random variable would, on average, take longer than the generation of a uniform random variable. Of course, the precise difference depends a lot on implementation; for instance, if we use the Box-Muller transform to generate a Gaussian, we would expect worse results than if we use the Ziggurat method [18].

MATLAB implements an optimized version of the Ziggurat method to generate Gaussian variables [19]. Our results on MATLAB indicated that, surprisingly, the generation of Gaussian variables via the built-in `randn` function was faster than the generation of uniform random variables via the built-in `rand`. These tests were performed on a Pentium-D 3.2 GHz machine with 3.5 GB of RAM.

We also tried the GNU Scientific Library[1] for C, which provides a Mersenne-twister implementation for uniform random variables (`gsl_rng_mt19937`), and a Ziggurat-based method for generating normal random variables (`gsl_ran_gaussian_ziggurat`). Our results here indicated that uniform random generation was on average faster than the generation of a Gaussian.

5.2 Clustering test. We looked at the quality of the solution generated by random projections by applying the above idea to the clustering of data streams. The scenario is as follows. We have a set of data points, each of which can be thought of as a stream. The points are updated asynchronously - that is, at every update, an arbitrary coordinate is varied. At given moments in time (say every 100 time-steps), we would like to cluster the data as it stands at that instant - we can use this information to find groups points that, at this instant, are "similar" to one another. The problem is that if the streams are high-dimensional, clustering them can be quite expensive.

So, we looked at trying clustering using both our projection-based sketch with a sparse matrix, and the L_2 sketch of Indyk to keep a sketch of the high-dimensional streams, and then perform clustering on these lower-dimensional points. We ran experiments for a randomly generated data-set (described below) and used the k-means and kernel k-means clustering algorithm to cluster the data. The latter uses dot-products to measure distance, and the former Euclidean (L_2) distances.

5.2.1 Data-set. We generated an artificial clustered data-set as follows. We chose m random cluster centres μ_i, and some variances σ_i^2 for each of them. We then created a mixture of Gaussians based on this:

$$X = \sum_{i=1}^{m} \alpha_i \mathcal{N}(\mu_i, \sigma_i^2)$$

where $\sum_{i=1}^{m} \alpha_i = i$.

This forms fairly distinct clusters. We made n points, which were randomly assigned to one of these clusters.

[1] `http://www.gnu.org/software/gsl/`

# values	Uniform	Gaussian
10^5	0.006074 ± 0.000050	0.004282 ± 0.000056
10^6	0.062306 ± 0.000252	0.044472 ± 0.000120
10^7	0.625017 ± 0.006305	0.441509 ± 0.000677
10^8	6.22510 ± 0.017631	4.415166 ± 0.022836

Table 1: Average time taken and standard deviation, in seconds, to generate uniform and Gaussian random variables over 10,000 runs, with MATLAB

# variables	Uniform	Gaussian
10^5	0.0043 ± 0.0049	0.0016 ± 0.0050
10^6	0.0441 ± 0.0048	0.1572 ± 0.0045
10^7	0.4726 ± 0.0267	1.6406 ± 0.0945
10^8	4.4054 ± 0.1700	17.8785 ± 0.5173

Table 2: Average time taken and standard deviation, in seconds, to generate uniform and Gaussian random variables over 10,000 runs, with C and GSL

We then generated a new mixture, and for each point in the original clustering, chose a random point in the new clustering as its "partner". The updates then simply made the original points become the new points. For example, say we the first point in the original mixture was $(1, 2, 3)$, and its partner in the new mixture was judged to be $(0, 5, 4)$. Then, the set of generated updates would be $\{(1, 1, -1), (1, 2, 3), (1, 3, 1)\}$

We chose the variances σ^2 randomly, but so as to create fairly well-defined clusters. We found that an average σ^2 of about 9 was able to create such clusters.

Also, the desired number of clusters C in our clustering step was chosen to be m, the true number of clusters in the data-set, so that the offline algorithm would be expected to give excellent results, which the streaming algorithms could be compared to.

5.2.2 Our implementation. We ran our simulations using MATLAB, on a Pentium-D 3.2 GHz machine with 3.5 GB of RAM. We used Achlioptas' matrix, defined in §2 to do our projections. The reduced dimension k was varied, and we observed the effect on the solution quality. We did not follow Achlioptas' bound (Theorem 2 in [1]) because of literature suggesting that these bounds, while theoretically important, are somewhat weak in practise [5, 17, 16].

5.2.3 Measuring cluster quality. To assess the cluster quality, we used two metrics - a similarity metric, and an intra-cluster centroid distance metric.

The first measure, which we henceforth refer to as the *similarity* of two clusterings, finds how many pairs of points lie in the same/different cluster for two given clusterings. This is useful for k-means clustering, because it is possible that we get clusterings that "look" different, but are actually the same. We computed the similarity of the online (sketch-based) clustering with respect to the offline clustering.

For instance, suppose we have two points. Are the cluster allocations (1, 2) and (2, 1) the same? Even though they are different, clearly they are just re-labellings of each other, and so for most intents and purposes, they are the same. This would not be caught if we compared how many points were in the *same* cluster, though.

For example, suppose we have the following cluster indicator matrices:

$$A_1 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \end{bmatrix}$$

$$A_2 = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix}$$

Now look at pairs of points:

- (1, 2), (1, 3), (2, 4), (3, 4) are in different clusters in both cases
- (1, 4) are in the same clusters in both cases (disregard the fact that they are in cluster 1 in the first case, but cluster 2 in the second case)
- (2, 3) are in different clusters in the first case, but move to the same cluster in the second case. So, this is counted as a 'bad' pair

We say the second clustering is $\frac{14}{16} \times 100 = 87.5\%$ similar to the first - only two pairs (namely, (3, 4) and (4, 3)) are 'bad' out of a possible 16. Note that we have side-stepped the problem that we may classify clusters in a different way.

The other measure is that of the distance of points to the centroids of the clusters they are in. Suppose we have m clusters $c_1, c_2, \ldots, c_m$ of our points $x_1, x_2, \ldots, x_n$. These clusters are defined by centroids, $r_1, r_2, \ldots, r_m$. The objective of k-means clustering is to minimize

$$\sum_{i=1}^{m} \sum_{x_j \in c_i} ||\mathbf{x_j} - \mathbf{r_i}||^2$$

A natural measure of cluster quality is therefore the value of this objective function - a clustering that has a smaller objective function value than another clustering would be called a "better" clustering. We computed the ratio of the centroid sum of the online clustering to that of the offline clustering.

5.2.4 Results. For the Gaussian mixture data, our results for k-means and kernel k-means are presented in Tables 3 to 6.

We can see that as the reduced dimension k increases, we usually get better results in terms of clustering quality. This is as expected, since the higher the dimension we project to, the less error we incur, and hence the "truer" the representation of the original data. We also see that in general, the results for kernel k-means are not as accurate as those for k-means. This is also as expected, as the error incurred in the dot-product does not have as tight a bound as there is for distance.

We also note that projections, on average, manage to out-perform the L_2 sketch in terms of quality of clustering, using either measure.

We measured the time taken for the update of the sketch, which essentially involves the multiplication of a random row by the update value. This does not include the generation of the row itself, which simply involves the generation of $\Theta(k)$ random variables (Gaussians in the case of the L_2 sketch, and uniform random variables for the projection sketch). Our results are presented in Table 7. We varied the number of reduced dimensions k, and tried a large volume of updates of each of the sketches. Our experiments indicated that the update time for the projection sketch was faster than that of the L_2 sketch, which is to be expected, since our sketch involves a $\frac{2}{3}$rds sparse row, allowing for a reduction in the number of multiplications that are required. Also, there is a linear growth in the difference as k increases, which is again as expected (as each update requires $\Theta(k)$ time).

6 Conclusion.

We propose the use of random projections with a sparse matrix in data-streaming problems, as an extension of the L_2 sketch outlined by [14]. We test the quality of the sketch provided by sparse-projections by applying the sketch to an online clustering problem, and our results indicate comparable accuracy to the earlier L_2 sketch, and an improved runtime performance.

A natural extension of our work would be to use Li's matrix [16], which might give more efficient computation of stream updates, although there might be loss in the accuracy of the sketch.

References

[1] Dimitris Achlioptas. Database-friendly random projections. In *PODS '01: Proceedings of the Twentieth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, pages 274–281, New York, NY, USA, 2001. ACM Press.

[2] Charu Aggarwal, Jiawei Han, Jianyong Wang, and Philip Yu. On high dimensional projected clustering of data streams. *Data Mining and Knowledge Discovery*, 10(3):251–273, May 2005.

[3] Noga Alon, Yossi Matias, and Mario Szegedy. The space complexity of approximating the frequency moments. In *STOC '96: Proceedings of the twenty-eighth annual ACM symposium on Theory of computing*, pages 20–29, New York, NY, USA, 1996. ACM Press.

[4] Arvind Arasu, Brian Babcock, Shivnath Babu, Mayur Datar, Keith Ito, Itaru Nishizawa, Justin Rosenstein, and Jennifer Widom. Stream: The stanford stream data manager. *IEEE Data Engineering Bulletin*, 26:19–26, 2003.

[5] Ella Bingham and Heikki Mannila. Random projection in dimensionality reduction: applications to image and text data. In *KDD '01: Proceedings of the Seventh ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 245–250, New York, NY, USA, 2001. ACM Press.

[6] Sirish Chandrasekaran, Owen Cooper, Amol Deshpande, Michael J. Franklin, Joseph M. Hellerstein, Wei Hong, Sailesh Krishnamurthy, Samuel R. Madden, Fred Reiss, and Mehul A. Shah. Telegraphcq: continuous dataflow processing. In *SIGMOD '03: Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, pages 668–668, New York, NY, USA, 2003. ACM Press.

[7] Graham Cormode, Mayur Datar, Piotr Indyk, and S. Muthukrishnan. Comparing data streams using hamming norms (how to zero in). *IEEE Transactions on Knowledge and Data Engineering*, 15(3):529–540, 2003.

[8] Sanjoy Dasgupta and Anupam Gupta. An elementary proof of the Johnson-Lindenstrauss lemma. Technical

Clusters	k	Similarity		Centroid	
		Projections	L_2	Projections	L_2
2	5	76.1949 ± 20.7221	67.8897 ± 19.2811	0.9513 ± 0.1502	0.8756 ± 0.1959
2	100	92.8257 ± 17.0333	88.0959 ± 19.1235	1.0055 ± 0.0229	0.9953 ± 0.0299
2	200	91.3545 ± 18.9948	89.2579 ± 19.7003	1.0187 ± 0.0808	1.0143 ± 0.0816
5	5	71.0404 ± 9.2157	68.4949 ± 6.8828	0.8900 ± 0.3006	0.8191 ± 0.2000
5	100	84.8283 ± 9.4887	80.3636 ± 7.9861	1.0011 ± 0.4127	0.9999 ± 0.5335
5	200	87.7879 ± 9.5612	83.2020 ± 9.0100	1.0013 ± 0.2346	1.0025 ± 0.8929

Table 3: Average similarity and centroid radio with standard deviation for k-means when $n = 100, d = 1000$

Clusters	k	Similarity		Centroid	
		Projections	L_2	Projections	L_2
2	5	84.8782 ± 16.9261	77.2299 ± 18.0145	0.9544 ± 0.0359	0.8925 ± 0.0655
2	100	99.1558 ± 3.1589	93.2765 ± 13.7547	0.9996 ± 0.0012	0.9919 ± 0.0122
2	200	99.7285 ± 1.2769	95.6571 ± 11.7832	0.9999 ± 0.0004	0.9967 ± 0.0069
5	5	71.1870 ± 10.9173	75.5780 ± 10.1014	0.8138 ± 0.2546	0.8480 ± 0.5164
5	100	93.0631 ± 6.9965	89.2505 ± 10.0273	1.0016 ± 0.3315	0.9871 ± 0.2433
5	200	94.1880 ± 5.6249	92.6793 ± 9.6829	0.9998 ± 0.2616	0.9962 ± 0.2126

Table 4: Average similarity and centroid radio with standard deviation for k-means when $n = 1000, d = 1000$

Report TR-99-006, International Computer Science Institute, Berkeley, CA, USA, 1999.

[9] Lars Engebretsen, Piotr Indyk, and Ryan O'Donnell. Derandomized dimensionality reduction with applications. In *SODA '02: Proceedings of the thirteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 705–712, Philadelphia, PA, USA, 2002. Society for Industrial and Applied Mathematics.

[10] Xiaoli Fern and Carla Brodley. Random projection for high dimensional data clustering: A cluster ensemble approach. In *The Twentieth International Conference on Machine Learning (ICML-2003)*, August 2003.

[11] Dmitriy Fradkin and David Madigan. Experiments with random projections for machine learning. In *KDD '03: Proceedings of the Ninth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 517–522, New York, NY, USA, 2003. ACM Press.

[12] Mohamed Medhat Gaber, Arkady Zaslavsky, and Shonali Krishnaswamy. Mining data streams: a review. *SIGMOD Rec.*, 34(2):18–26, 2005.

[13] S. Guha, N. Mishra, R. Motwani, and L. O'Callaghan. Clustering data streams. In *FOCS '00: Proceedings of the 41st Annual Symposium on Foundations of Computer Science*, page 359, Washington, DC, USA, 2000. IEEE Computer Society.

[14] Piotr Indyk. Stable distributions, pseudorandom generators, embeddings, and data stream computation. *J. ACM*, 53(3):307–323, 2006.

[15] W.B. Johnson and J. Lindenstrauss. Extensions of Lipschitz mappings into a Hilbert space. In *Conference in Modern Analysis and Probability*, pages 189–206, Providence, RI, USA, 1984. American Mathematical Society.

[16] Ping Li, Trevor J. Hastie, and Kenneth W. Church. Very sparse random projections. In *KDD '06: Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 287–296, New York, NY, USA, 2006. ACM Press.

[17] Jessica Lin and Dimitrios Gunopulos. Dimensionality reduction by random projection and latent semantic indexing. In *Proceedings of the Text Mining Workshop, at the 3rd SIAM International Conference on Data Mining*, 2003.

[18] George Marsaglia and Wai Wan Tsang. The ziggurat method for generating random variables. *Journal of Statistical Software*, 5(8):1–7, 2000.

[19] Cleve Moler. Normal behavior. `http://www.mathworks.com/company/newsletters/news_notes/clevescorner/spring01_cleve.html`, 2001. Accessed September 28, 2006.

[20] S. Muthukrishnan. *Data Streams: Algorithms And Applications*. Now Publishers, 2005.

[21] Santosh S. Vempala. *The Random Projection Method*, volume 65 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*. American Mathematical Society, Providence, RI, USA, 2004.

		Similarity		Centroid	
Clusters	k	Projections	L_2	Projections	L_2
2	5	78.7045 ± 19.8381	70.7498 ± 15.1127	0.9393 ± 0.0621	0.8260 ± 0.1318
2	100	92.3418 ± 17.9205	68.9263 ± 16.6598	1.0114 ± 0.0647	0.8403 ± 0.1530
2	200	92.5962 ± 17.6414	70.2172 ± 16.1942	1.0410 ± 0.2690	0.8700 ± 0.3594
5	5	68.0808 ± 7.3734	66.7273 ± 2.7022	0.9033 ± 0.2361	0.7395 ± 0.1977
5	100	74.3030 ± 10.7344	66.3636 ± 3.6998	1.0026 ± 0.1360	0.7699 ± 0.1846
5	200	76.3030 ± 9.5702	66.6667 ± 3.9984	1.0030 ± 0.3598	0.7434 ± 0.2427

Table 5: Average similarity and centroid radio with standard deviation for kernel k-means when $n = 100, d = 1000$

		Similarity		Centroid	
Clusters	k	Projections	L_2	Projections	L_2
2	5	74.3242 ± 18.8077	75.3663 ± 17.9369	0.8772 ± 0.0794	0.8797 ± 0.0766
2	100	98.8904 ± 4.3437	75.8661 ± 17.2666	0.9995 ± 0.0015	0.8790 ± 0.0777
2	200	99.6458 ± 1.6092	76.6790 ± 17.3473	0.9999 ± 0.0005	0.8871 ± 0.0724
5	5	60.7976 ± 11.1999	49.4422 ± 12.1253	0.9416 ± 0.0751	0.7160 ± 0.2598
5	100	72.7155 ± 12.5398	55.8168 ± 11.7787	0.9959 ± 0.1151	0.7087 ± 0.2492
5	200	74.4466 ± 12.6621	51.9475 ± 12.1305	0.9978 ± 0.1351	0.7135 ± 0.2497

Table 6: Average similarity and centroid ratio with standard deviation for kernel k-means when $n = 1000, d = 1000, k = 200$

Updates	k	Projections	L_2
10^4	5	0.0588 ± 0.0007	0.0594 ± 0.0006
10^4	100	0.0772 ± 0.0006	0.0870 ± 0.0007
10^4	200	0.0981 ± 0.0007	0.1096 ± 0.0005
10^4	300	0.1195 ± 0.0009	0.1265 ± 0.0007
10^4	400	0.1312 ± 0.0019	0.1438 ± 0.0013
10^4	500	0.1580 ± 0.0007	0.1608 ± 0.0012
10^5	5	0.6061 ± 0.0037	0.5977 ± 0.0036
10^5	100	0.8055 ± 0.0048	0.8623 ± 0.0046
10^5	200	1.0476 ± 0.0079	1.0928 ± 0.0058
10^5	300	1.2738 ± 0.0047	1.2683 ± 0.0057
10^5	400	1.4301 ± 0.0051	1.4159 ± 0.0074
10^5	500	1.4601 ± 0.0040	1.5827 ± 0.0035
10^6	5	5.8064 ± 0.0066	5.8430 ± 0.0146
10^6	100	7.7333 ± 0.0119	8.6419 ± 0.0311
10^6	200	10.6300 ± 0.0353	10.6701 ± 0.0513
10^6	300	11.7557 ± 0.0575	12.5707 ± 0.0777
10^6	400	13.0989 ± 0.0941	14.1790 ± 0.0220
10^6	500	15.6372 ± 0.0197	15.9051 ± 0.0276

Table 7: Average time and standard deviation, in seconds, for sketch updates

School of Information Technologies, J12
The University of Sydney
NSW 2006 AUSTRALIA
T +61 2 9351 3423
F +61 2 9351 3838
www.it.usyd.edu.au

ISBN 1 86487 887 8