Fast Algorithms for Approximating the Singular Value Decomposition

ADITYA KRISHNA MENON and CHARLES ELKAN, University of California, San Diego

A low-rank approximation to a matrix A is a matrix with significantly smaller rank than A, and which is close to A according to some norm. Many practical applications involving the use of large matrices focus on low-rank approximations. By reducing the rank or dimensionality of the data, we reduce the complexity of analyzing the data. The singular value decomposition is the most popular low-rank matrix approximation. However, due to its expensive computational requirements, it has often been considered intractable for practical applications involving massive data. Recent developments have tried to address this problem, with several methods proposed to approximate the decomposition with better asymptotic runtime. We present an empirical study of these techniques on a variety of dense and sparse datasets. We find that a sampling approach of Drineas, Kannan and Mahoney is often, but not always, the best performing method. This method gives solutions with high accuracy much faster than classical SVD algorithms, on large sparse datasets in particular. Other modern methods, such as a recent algorithm by Rokhlin and Tygert, also offer savings compared to classical SVD algorithms. The older sampling methods of Achlioptas and McSherry are shown to sometimes take longer than classical SVD.

Categories and Subject Descriptors: G.1.3 [**Numerical Analysis**]: Numerical Linear Algebra—*Singular value decomposition*; G.4 [**Mathematical Software**]: Efficiency

General Terms: Experimentation, Algorithms, Performance

Additional Key Words and Phrases: Singular value decomposition, low rank approximation, experimental evaluation

ACM Reference Format:
Menon, A. K. and Elkan, C. 2011. Fast algorithms for approximating the singular value decomposition. ACM Trans. Knowl. Discov. Data 5, 2, Article 13 (February 2011), 36 pages.
DOI = 10.1145/1921632.1921639 http://doi.acm.org/10.1145/1921632.1921639

1. INTRODUCTION

Many practical applications involve the manipulation of large matrices from which certain information needs to be extracted. For example, in the problem of semantic analysis, where one wishes to determine the semantic concepts underlying a set of documents, one can represent the data in terms of a word-document matrix. Here, the (i, j)th entry in the matrix represents whether word i occurs in document j. We then wish to perform some analysis on this matrix in order to obtain the desired information, which in this case is a set of concepts.

There are at least two problems that arise when trying to analyze these large matrices. First, large matrices cause problems for the time and space complexity of most algorithms: for a start, it might not even be possible to store them in memory. Second, while real-world data has high *apparent* dimensionality (i.e., a large number of rows

Authors' address: A. K. Menon and C. Elkan, Department of Computer Science and Engineering, University of California, San Diego, La Jolla, CA 92037-0404; email: {akmenon, elkan}@cs.ucsd.edu.

DOI 10.1145/1921632.1921639 http://doi.acm.org/10.1145/1921632.1921639

or columns), it often has much lower *intrinsic* dimensionality. This means that many dimensions are actually redundant, and obscure the genuine dimensionality of the data. To avoid both these problems, applications typically focus on *low-rank approximations* of matrices. A low rank approximation to a matrix A is a matrix $\hat{A}$ for which $\text{rank}(\hat{A}) \ll \text{rank}(A)$, and where $||A - \hat{A}||_M$ is bounded for some matrix norm $||.||_M$. As neatly summarized in Gorrell [2006], low-rank approximations can be thought of as forcing the data to describe itself succinctly, distilling only its most important components into a more compact representation.

One of the most popular low-rank approximations is the *singular value decomposition* (SVD), which breaks up a matrix into three components,

$$A = USV^T, \tag{1}$$

where S is a (possibly rectangular) diagonal matrix containing the so-called *singular values*, and U, V are orthogonal. The most appealing property of this particular approximation is that it is optimal with respect to rank in the following sense: among all rank k matrices, the matrix A_k formed by considering only the top k singular values in S has the smallest distance to A measured in any unitarily invariant norm[1] $|| \cdot ||_M$:

$$||A - A_k||_M = \min_{\text{rank}(B)=k} ||A - B||_M.$$

We give more details in Theorem 2.1.

Due to this optimality property, the SVD is used to solve the problem of semantic analysis introduced earlier. In *latent semantic indexing* (LSI), we use a low-rank approximation to the term-document matrix to find the semantic concepts that underly a collection of documents [Furnas et al. 1988; Papadimitriou et al. 1998a]. We can think of the columns in the low-rank approximation as representing concepts or topics, rather than just words. This is because the new columns are linear combinations of the original columns, which correspond to words that are related by virtue of their appearance in similar documents.

Historically, the problem with the singular value decomposition has been the time required to compute it, which is $O(mn \min\{m, n\})$ for a dense $m \times n$ matrix [Golub and Van Loan 1996, p. 263]. This is superlinear in the size of the input data, making it infeasible to compute on very large datasets. Ironically, this the very place where one would like to use a low-rank approximation. Consequently, there has been a body of research on how one can find an *approximation* to the SVD of a matrix with a more favorable runtime. Specifically, we would like to compute some low-rank matrix A^* so that $||A - A^*||_F \approx ||A - A_k||_F$, and A^* is quicker to compute than A_k. Results in the literature have given disparate guarantees on the approximation factor and asymptotic runtime of computing A^*. However, to the best of our knowledge, there has been no systematic empirical comparison of these methods on practical datasets. This article is the first to study in a common framework, both analytically and experimentally, all the major algorithms that approximate the SVD. In the following sections, we describe previous work in the literature in more detail, outline our experimental setup, and provide experimental results that compare the various methods.

We wish to note that the focus of this article is on approximations to the SVD. Therefore, we do not share the same goals as earlier experimental comparisons of a range of a techniques for low-rank approximation and dimensionality reduction such as Fradkin and Madigan [2003] and Sun et al. [2007].

[1]A norm $|| \cdot ||_M$ is unitarily invariant if $||UAV||_M = ||A||_M$ for all A and all unitary U, V. If U, V are defined over the reals, then are simply orthogonal matrices.

Table I. Some Popular Low Rank Approximation Techniques and Their Decomposition

Approximation technique	**Decomposition**	**Reference**
SVD	A = USV^T $U \in \mathbb{R}^{m\times k}$, $S \in \mathbb{R}^{k\times k}$, $V \in \mathbb{R}^{n\times k}$	[Golub and Van Loan 1996]
CUR	$A = CUR$ $C \in \mathbb{R}^{m\times k}$,$U \in \mathbb{R}^{k\times k}$, $R \in \mathbb{R}^{n\times k}$ C contains exactly k columns of A	[Sun et al. 2008]
Interpolative decomposition	$A = PB$ $P \in \mathbb{R}^{m\times k} B \in \mathbb{R}^{k\times n}$ P contains exactly k columns of A	[Liberty et al. 2007]

2. BACKGROUND AND RELATED WORK

2.1 Low-Rank Approximations

A *low-rank approximation* to a matrix $A \in \mathbb{R}^{m\times n}$ is some matrix $\hat{A} \in \mathbb{R}^{m\times n}$ which satisfies two properties: (i) rank($\hat{A}$) $\ll$ rank(A), and (ii) $||A - \hat{A}||_M$ is small for some matrix norm $||\cdot||_M$. Most low-rank approximations work by decomposing A into the product of two or more matrices; for example, as we saw in the introduction, the SVD decomposes A as the product USV^T. There are at least three reasons why such low-rank approximations are useful.

— *Noise removal*. As mentioned in the introduction, Gorrell [2006] describes the process of finding a low-rank approximation as forcing the original matrix to provide a concise description of itself. Intuitively, this can be seen as a form of noise removal. It has been observed that in many real-world applications, data can possess much smaller intrinsic than apparent dimensionality. In other words, it lies close to a rank k subspace, where $k \ll \min\{m, n\}$.

— *Space savings*. While the approximation $\hat{A}$ has the same dimensions as A, and thus ostensibly has the same storage requirements, by writing it as a decomposition one can dramatically reduce the storage requirement. For example, if we consider a rank k approximation via the SVD, the space requirement is $O((m+n)k + k^2)$ as opposed to $O(mn)$.

— *Data description*. The individual components of a matrix decomposition can provide valuable information that helps one analyze the structure of the input. For example, in the application of collaborative filtering to the movie recommendation problem, the input is a matrix A where A_{ij} represents the (possibly unknown) rating that user i has given movie j. If one writes[2] $A \approx UV$ where $U \in \mathbb{R}^{m\times k}$, $V \in \mathbb{R}^{k\times n}$, then we can interpret the rows of U and the columns of V as containing weights of some latent attributes of movies (for example, whether a movie has comedy elements, whether it is animated, and so on). That is, each row of U can be thought of as a user's preferences for each of these latent attributes, and similarly the columns of V represent how much each attribute features in a particular movie.

Table I gives examples of some popular decomposition-based low rank approximation techniques.

Of the various low-rank approximations, the SVD is by far the most well studied and arguably the most popular. Our focus in this article will be solely on approximations of the SVD; our goal is *not* to evaluate how the SVD performs compared to other low

[2]In collaborative filtering, A contains missing entries. Hence the error of the approximation is only measured with respect to the known entries.

rank techniques. It should be noted that there are situations where the SVD is not necessarily the ideal low rank approximation. A simple example is when A is sparse: the matrices U and V produced by the SVD do not necessarily share the sparsity pattern of A, and thus require *more* storage capacity than the original matrix. Some alternative approximations where the decomposition consists of sparse matrices have been proposed to solve this problem [Zhang et al. 2001].

A related procedure to low-rank approximation is *dimensionality reduction*. Here, we are interested in a matrix $\tilde{A} \in \mathbb{R}^{k \times n}$, where $k \ll m$ and $\tilde{A}$ preserves some structure in the matrix A. Since the dimensions of A and $\tilde{A}$ are not the same, the measure of similarity can no longer be a simple matrix norm. An example of dimensionality reduction is *random projection* method [Vempala 2004], which guarantees that with high probability, the pairwise distances between columns of $\tilde{A}$ are close to the distances of the corresponding columns in $\tilde{A}$:

$$(\forall i, j)\,(1-\epsilon)||A^{(i)} - A^{(j)}||_2^2 \leq ||\tilde{A}^{(i)} - \tilde{A}^{(j)}||_2^2 \leq (1+\epsilon)||A^{(i)} - A^{(j)}||_2^2 \text{ with high probability.}$$

2.2 The SVD of a Matrix

The singular value decomposition is a classical mathematical technique for factorizing a matrix. As expressed in Equation (1), the SVD for a matrix $A \in \mathbb{R}^{m \times n}$ is the decomposition $A = USV^T$, where $U \in \mathbb{R}^{m \times m}$, $V \in \mathbb{R}^{n \times n}$ and $S \in \mathbb{R}^{m \times n}$. The matrices U, V are orthogonal, with their columns being the eigenvectors of XX^T and X^TX respectively. S is a (possibly rectangular) diagonal matrix diag$(\sigma_1, \ldots, \sigma_p)$ where $p = \min\{m, n\}$. The σ_i's are sorted in decreasing order and are known as the *singular values* of A. The σ_i's satisfy $\sigma_i \geq 0$ for all i, and the squares of the nonzero σ_i's are the eigenvalues of XX^T (or equivalently X^TX).

One of the most common operations one performs using the decomposition USV^T is to form the truncated SVD

$$A_k = U_k S_k V_k^T,$$

where we take the first k columns of U, V and consider the $k \times k$ submatrix S_k. Due to the orthogonality of U, V, one can equivalently write A_k as $U_k U_k^T A$ or $A V_k V_k^T$. This matrix has rank at most k, and a classic theorem of Eckart-Young-Mirsky says that it is a good approximation to the matrix A in the following specific sense.

THEOREM 2.1 [ECKART AND YOUNG 1936; MIRSKY 1960]. *For a given matrix $A \in \mathbb{R}^{m \times n}$, let A_k be its truncated SVD. Let $|| \cdot ||_M$ be a unitarily invariant matrix norm. Then, for all $k \leq rank(A)$,*

$$||A - A_k||_M = \min_{rank(B)=k} ||A - B||_M.$$

To see the intuition behind the theorem, another way to express the SVD is to write each element of the matrix A as

$$A_{ij} = \sum_l U_{il} V_{jl} S_{ll} = \sum_l U_{il} V_{jl} \sigma_l^2.$$

Now suppose that we keep just the top k singular values of A, and effectively treat the rest as being 0: this creates a rank k approximation to A. Since $\sigma_1 \geq \sigma_2 \ldots \geq \sigma_p$, intuitively this approximation should be quite good, because we are keeping the terms that contribute the most to A_{ij}. What the theorem says is that this approximation is not only good, but is optimal.

Two commonly studied unitarily invariant norms are the Frobenius and 2-norm. The Frobenius norm of a matrix, $||.||_F$, is an analogue of the ℓ_2 norm for vectors:

$$||A||_F = \sqrt{\sum_i \sum_j A_{ij}^2} = \sqrt{\sum_i \sigma_i^2}.$$

The 2-norm (or *spectral norm*) of a matrix, $||.||_2$, is an induced norm that equals the largest singular value of a matrix:

$$||A||_2 = \sigma_1 = \max_i \sigma_i.$$

An important connection between the 2-norm and the singular values of a matrix A is the following result:

$$||A - A_k||_2 = \sigma_{k+1}.$$

It is well known that $||A||_2 \leq ||A||_F \leq \sqrt{n} \cdot ||A||_2$ [Golub and Van Loan 1996, p. 56].

Before discussing how one algorithmically computes the SVD, we briefly point out some of its applications in computer science.

2.3 Applications of the SVD

The popularity of the SVD as a data analysis tool comes from the optimality property of Theorem 2.1. It tells us that of all rank k matrices, the truncated SVD is the one that is closest to the original data matrix in terms of a matrix norm, such as the Frobenius norm. This means that it is the low-rank approximation that preserves as much structure of the original data matrix as possible. There are several applications of the SVD in computer science, and especially in the fields of machine learning and data mining. We list a few interesting ones here.

— *Data visualization*. A fundamental step in data analysis is understanding the structure of the data. For example, to check if clustering is meaningful, we would like to first visually identify if there are natural clusters in the data; to see if classification is meaningful, we would like to see if the various classes are well-separated; and so on. It is challenging to visualize high-dimensional data as-is, and so a common technique is to *project* the data into 2D or 3D, where it can be easily visualized. Principal component analysis (PCA) [Jolliffe 1986] is a popular method that allows one to project data onto the directions that have the maximum variance, which are intuitively are the most "interesting." It thus serves as a powerful way to visualize high-dimensional data. PCA can be shown to be nothing but the SVD of a transformed version of the data matrix where the columns are centered about the origin.

— *Latent semantic indexing*. As mentioned in the Introduction, a classic problem in information retrieval is discovering a set of topics that underly a corpus of documents. Latent semantic indexing (LSI) [Furnas et al. 1988] is one approach to this problem based on the SVD. The idea is to treat the corpus as a document-by-term matrix, where the (i, j)th entry is the number of times word j occurs in document i. Computing the truncated SVD of this matrix will find a low rank representation of it that captures most of the relevant structure. The resulting factors in U and V can be thought to be the latent features or topics for the given corpus.

— *Collaborative filtering*. In collaborative filtering, the input is a matrix of users-by-items, and each entry is the rating a user gives to a particular item. The biggest characteristic of such data is that it contains missing entries, corresponding to (user, item) pairs for which we do not know the corresponding rating. Our goal is to predict the values of these missing enries, so that we can recommend new items to a user. A popular approach is to learn a decomposition akin to the SVD of the data matrix, but restricted to only the *observed* entries [Paterek 2007]. (Additionally, to prevent overfitting, one adds Frobenius norm regularization of the U and V matrices also.) Such a decomposition can be thought of as learning a latent feature representation for both users and items, which are then used to predict the values of missing ratings.

2.4 Computing the SVD

In general, the problem of computing the SVD of a matrix reduces to that of computing the eigendecomposition of a symmetric matrix. Eigenvalue computation is deeply connected to the SVD because of the following fact (see, e.g., Berry et al. [2005], Golub and Van Loan [1996, p. 448]).

Fact 2.2. Let $A \in \mathbb{R}^{m\times n}$ have the decomposition USV^T, where $S = \operatorname{diag}(\sigma_i)$. Let u_i, v_i denote the columns of U, V respectively. Then,

— $C = A^T A$ has eigenvalues σ_i^2 with corresponding eigenvectors v_i.
— $B = \begin{bmatrix} 0 & A \\ A^T & 0 \end{bmatrix}$ has eigenvalues $\pm\sigma_i$ with corresponding eigenvectors $\frac{1}{\sqrt{2}}\begin{bmatrix} u_i \\ \pm v_i \end{bmatrix}$.

Therefore, common methods for computing the SVD of a matrix are standard eigensolvers such as QR iteration and Arnoldi/Lanczos iteration, or are slight modifications of the same, such as the modified Golub-Reinsch method [Chan 1982]. We note that any algorithm for SVD computation can only produce an approximation to the true SVD of a matrix. The reason for this is an impossibility result which says that no algorithm can find the exact eigenvalues of a general matrix. The proof of this claim is that computing the eigenvalues of a matrix is equivalent to finding the roots of its characteristic polynomial, and further, that every polynomial is the characteristic polynomial of some matrix (known as the *companion matrix* [Horn and Johnson 1991, p. 147]). However, the Abel-Ruffini theorem [Artin 1942] says that there is no formula for finding the roots of polynomials of degree ≥ 5. This implies that no general algorithm exists for finding eigenvalues, which shows the claim.

For most classical SVD algorithms, we have the following error guarantees due to finite precision arithmetic (see Anderson et al. [1992, p. 113], Golub and Van Loan [1996, p. 261]). Throughout, we let ϵ denote the machine-precision constant, that is, the maximum relative error in storing a real number on a computer.

(1) The reported singular values $\widetilde{\sigma_i}$ are close to the true ones σ_i:

$$|\sigma_i - \widetilde{\sigma_i}| \leq \epsilon \cdot \sigma_1.$$

Note that the error is bound is independent of i.

(2) The reported left-singular vectors $\widetilde{u_i}$ are close in angle $\theta(\widetilde{u_i}, u_i)$ to the true left-singular vectors u_i:

$$\theta(\widetilde{u_i}, u_i) \leq \frac{\epsilon ||A||_2}{\min_{j\neq i} |\sigma_i - \sigma_j|}.$$

The presence of the term $\min_{j\neq i} |\sigma_i - \sigma_j|$ tells us that if the ith singular value is very close to some other singular value, it is difficult to get a good approximation to u_i.

(3) The reported matrices $\tilde{U}, \tilde{\Sigma}, \tilde{V}$ are precisely the SVD of a matrix $A + \Delta A$, where

$$||\Delta A||_2 \leq \epsilon ||A||_2.$$

On dense datasets, the classical methods have complexity of $O(mn \min\{m, n\})$ to compute the *thin (or economy) SVD*, defined as:

$$A = U_n S_n V_n^T,$$

that is, the truncated SVD with $k = n$. This makes them infeasible if $\min\{m, n\}$ is large. This is the motivation for approximations to the SVD, which we look at in the next section. We wish to comment first that we will usually be interested in approximations of rank k where $k \ll \min\{m, n\}$. This means we compute only the top few singular values and vectors of the matrix A. One would hope that in this case, the runtime is only $O(mnk)$, but we are unaware of any references formally proving such a result. Appendix B explores this issue empirically.

On sparse datasets, ideally one would like the runtime to depend on the number of nonzero entries rather than the total size of the matrix. This is true in classical algorithms such as power iteration [Golub and Van Loan 1996, p. 330], where the main bottleneck in each iteration is the computation of the product Az for some iteration-dependent vector z. Clearly, if A is sparse then this computation can be done in $O(\text{nnz}(A))$ rather than $O(mn)$ time, where $\text{nnz}(A)$ denotes the number of nonzeros in A. For more modern algorithms such as Arnoldi iteration, however, we are not aware of formal bounds on the runtime depending on the number of nonzeros in A.

2.5 Computing Approximations to the SVD

The truncated SVD of a matrix is mathematically guaranteed to be the optimal low rank approximation in the sense described in Theorem 2.1. However, a natural question is whether we can find a suboptimal approximation much quicker: provided the additional error is not too great, one can typically use this instead of the truncated SVD. For a norm $||\cdot||_M$, the quantity of interest is

$$\delta_M(A, \hat{A}) := ||A - \hat{A}||_M - ||A - A_k||_M,$$

where A_k is the truncated SVD of A and $\hat{A}$ is an approximation to A_k. If $\delta_M(A, \hat{A})$ is small, that means that $\hat{A}$ is close to being an optimal low rank approximation to A. Several results in the literature confirm that we can indeed find a matrix with small $\delta_M(A, \hat{A})$ asymptotically faster than finding A_k, and are the focus of the next two sections.

At the outset, we make some observations about most methods for approximating the SVD.

(1) They do not completely eschew a standard SVD computation,[3] but rather look to compute the SVD of a simpler matrix, that is, a matrix for which computing the SVD is more computationally tractable. This is accomplished in one of two ways. One way is to sparsify the input in hopes of computing the SVD faster; recall that in the previous section we said that classical algorithms work faster on sparse matrices (although modern algorithms do not have similar guarantees). The other way is to reduce the dimensionality of the matrix, which can cause a significant speedup. For reducing the dimensionality of the matrix, there are a further two approaches, namely sampling columns from A or performing an embedding into a lower dimension space. To summarize, the three general methods in the literature for approximating the SVD are outlined in Table II.

(2) The choice of the matrix norm $||\cdot||_M$ is usually $||\cdot||_F$ or $||\cdot||_2$, but sometimes it is the square version of these norms, that is, $||\cdot||_F^2$ or $||\cdot||_2^2$. As observed in Achlioptas and

[3]An exception to this is the algorithm of [Har-Peled 2006], which uses geometric tools.

Table II. The Three General Ways to Approximate SVD Computation

Method	**Sparsification**	**Column sampling**	**Embedding**
Algorithm	(1) Retain element A_{ij} with probability p_{ij}, otherwise replace it with 0, giving an approximation $\hat{A}$. (2) Call SVD on $\hat{A}$, giving $\hat{A} = USV^T$. (3) Return U, S, $\Pi_{C[V]}$.	(1) Pick c columns $A^{i_1}, \ldots, A^{(i_c)}$ with probabilities $p_1, \ldots, p_n$. (2) Form the matrix $C = \left[s_1 A^{(i_1)} \ldots s_c A^{(i_c)}\right]$ where $s_1, \ldots, s_c$ are scale factors. (3) Call SVD on $C^T C$, giving $C^T C = USV^T$. (4) Return $C \times \left[t_1 U^{(1)} \ldots t_c U^{(c)}\right]$ where $t_1, \ldots, t_c$ are scale factors.	(1) Compute $C = RA$ for some random matrix R. (2) Find $\Pi_{C[A]}$, the projection of A onto the rowspace of C. (3) Call SVD on $\Pi_{C[A]}$, giving $\Pi_{C[A]} = USV^T$. (4) Return U, S, V.
Papers	[Achlioptas and McSherry 2001], [Achlioptas and McSherry 2007]	[Frieze et al. 1998], [Drineas et al. 2006b]	[Sarlos 2006], [Rokhlin et al. 2009], [Nguyen et al. 2009]

McSherry [2007], theoretically comparing methods with disparate guarantees is not straightforward. We will use the notation δ_M^2 to refer to the error for the square norm $||\cdot||_M^2$. Further, some methods have guarantees for only one of the norms. We find in practice that these methods perform well on both norms, however.

(3) The guarantee on the approximation is said to be *additive* if $\delta_M(A, \hat{A}) \leq \epsilon ||A||_M$, and *multiplicative* if $\delta_M(A, \hat{A}) \leq \epsilon ||A - A_k||_M$. A multiplicative guarantee is much stronger than an additive error, because if the matrix A can be decomposed as a low-rank matrix plus noise, then one can expect $||A - A_k||_M$ to be much smaller than $\epsilon ||A||_M$.

With this overview in place, we now summarize the relevant literature on these SVD approximations.

2.6 Literature on SVD Approximations

The genesis of most research in computing faster low-rank approximations is the work of Frieze et al. [1998], which provided a method for computing a low-rank matrix C that gives additive error compared to the optimal rank-k approximation:

$$\delta_F^2(A, C) \leq \epsilon ||A||_F^2. \tag{2}$$

Here, ϵ is a user-controlled parameter that determines the extra error term. The algorithm works by random sampling of the columns of A to produce a rank-k submatrix of A, from which one can construct C. However, the algorithm has a strong dependence on k and ϵ, as it requires finding the SVD of a square matrix with $O(k^4/\epsilon^3)$ columns. This does not diminish the importance of the result, though, as it was the first to show that randomization and sampling could indeed solve this problem.

After Frieze et al. [1998], the next major contribution was the seminal work of Achlioptas and McSherry [2001], which presents a sampling approach that chooses whether each particular entry in the matrix should be kept or not. The idea is as follows: given a matrix A, we go through each element and retain a scaled version of it with some probability p, and otherwise discard it (i.e., make the corresponding entry 0). If we denote the sampled matrix by $\widehat{A}$, then

$$\widehat{A}_{ij} = \begin{cases} A_{ij}/p & \text{with probability } p \\ 0 & \text{otherwise.} \end{cases}$$

This creates a sparse matrix on which we run the standard SVD algorithm, which will in principle run faster. It is shown that the resulting approximation $\widehat{A}_k$ is a good approximation to A_k, with:

$$\delta_F(A, \hat{A}) \leq 3b(nk/p)^{1/4}||A||_F$$
$$\delta_2(A, \hat{A}) \leq 8b\sqrt{n/p}||A||_F,$$

where $b = \max|A_{ij}|$. Observe that the Frobenius norm $||A||_F$ is in the error term even when we look at optimality with respect to 2-norm.

This idea is refined in Achlioptas and McSherry [2007], the journal version of Achlioptas and McSherry [2001], which shows how one can tune this approach to allow for nonuniform sampling based on the magnitude of an entry: here, we have different probabilities p_{ij} for each element rather than a universal probability p, so that

$$\widehat{A}_{ij} = \begin{cases} A_{ij}/p_{ij} & \text{with probability } p_{ij} \\ 0 & \text{otherwise,} \end{cases}$$

where, for some $p > 0$,

$$p_{ij} = \sqrt{p}\frac{|A_{ij}|}{b} \max\left\{\sqrt{p}\frac{|A_{ij}|}{b}, \frac{64(\log n)^2}{\sqrt{n}}\right\},$$

where again $b = \max|A_{ij}|$. Intuitively, this ought to perform better than uniform sampling because one expects entries with large magnitude to be more important for the low-rank structure of the matrix. The expected number of nonzero $\hat{A}_{ij}$'s can be shown to be $O(s + m(8\log n)^4)$. The error guarantee in this paper is also additive, and for $p = \frac{16nb^2}{\epsilon^2||A||_F^2}$ is of the form

$$\begin{aligned} \delta_F(A, \hat{A}) &\leq 3\sqrt{\epsilon}k^{1/4}||A||_F \\ \delta_2(A, \hat{A}) &\leq \epsilon||A||_F. \end{aligned} \tag{3}$$

The ideas of Frieze et al. [1998] were extended in Drineas et al. [2006b], which uses the intuition that a well-chosen sample of the columns of A can provide a good approximation to the subspace spanned by all columns of A. The paper shows that if we choose $c = O(k/\epsilon^2)$ such columns, we can achieve the same approximation result as Equation (2) for the $||\cdot||_F^2$ norm, and also that if we choose $c = O(1/\epsilon^2)$ columns,

$$\delta_2^2(A, \hat{A}) \leq \epsilon||A||_F^2.$$

As noted in Achlioptas and McSherry [2007], this approximation guarantee is superficially similar to that of Equation (3), but the subtle difference is that it makes a guarantee about the square-norm, $||\cdot||_2^2$, rather than just the $||\cdot||_2$ norm.

The next major step was the slew of results providing *multiplicative* error guarantees [Deshpande and Vempala 2006; Drineas et al. 2006c; Har-Peled 2006; Sarlos 2006], which are the standard type of bound in most literature on approximation algorithms. Sarlos [2006] approaches the problem via embedding rather than sampling. The basic idea is as follows: we consider the subspace spanned by a random projection

[Vempala 2004] $V = RA$ of the input matrix A, where $R_{ij} \in \text{Uni}(\{-1, +1\})$. It is shown that by projecting A onto the rowspace of V, and then finding the best rank-k approximation to this new space (i.e., the truncated SVD), we get a good approximation to the best rank-k approximation of A itself. Note that this is a double projection: first we project A onto a random subspace, and then we project A onto the row-span of the resulting space. If we denote the projection of A onto the rowspace of B by $\Pi_B(A)$, then the paper guarantees

$$\delta_F(A, \left(\Pi_{RA}(A)\right)_k) \leq \epsilon ||A - A_k||_F,$$

which is of course a multiplicative error bound.

Liberty et al. [2007] gave an extension to the Sarlos [2006] algorithm using the interpolative decomposition. This technique tries to write A as the product BP, where $P \in \mathbb{R}^{k \times n}$ is a downsampled version of the identity matrix, that is, some k columns of P constitute I_k. It gives a bound

$$\delta_2(A, \hat{A}) = (\sqrt{4k(n-k)+1} - 1)||A - A_k||_2.$$

While the results of the paper are given for complex matrices, it is claimed that they extend naturally to the case of real matrices.

Recently, Rokhlin et al. [2009] proposed a method for fast computation of PCA via the SVD. Their algorithm can be seen as an extension of Sarlos [2006] in the following way: instead of computing $V = RA$, $R \in \mathbb{R}^{k/\epsilon \times m}$, they find $V = R(AA^T)^i A$ for $R \in \mathbb{R}^{l \times m}$, where i, l are user-supplied parameters. (In the analysis, it is required that $l \geq k$.) The other difference is the choice of the random matrix R: for Sarlos [2006], it is a matrix with entries that are uniformly ± 1, whereas in Rokhlin et al. [2009] R has standard Gaussian entries. Their algorithm is thus almost exactly that of [Sarlos 2006] when $i = 0$ and $l = k/\epsilon$. The technical details of the algorithm are also similar to that of Liberty et al. [2007]. The approximation $\hat{A}$ is shown to be close only in terms of spectral norm:

$$\delta(A, \hat{A}) = (Cm^{1/(4i+2)} - 1)||A - A_k||_2,$$

where C is some function that increases slowly with k. There is another algorithm presented in the paper based on the block Lanczos method. Here, instead of using $V = R(AA^T)^i A$, we use

$$V = \begin{bmatrix} RA \\ RA(A^TA) \\ \vdots \\ RA(A^TA)^i \end{bmatrix}.$$

As the algorithm is shown to be a generalization [Liberty et al. 2007], we do not include that algorithm in our comparison.

We also look at a recent result by Nguyen et al. [2009] that builds on similar ideas: as with the previous results providing multiplicative guarantees, the idea is to simply project onto the rowspace of some projection C, and then find the best rank k approximation to this projection. It generalizes the result of Sarlos [2006] because (1) the projection matrix is allowed to be *any* orthonormal matrix, for example the Walsh-Hadamard matrix, and (2) the projection step is preceded by a downsampling step, where only some d rows of the projection matrix are actually used. The algorithm therefore combines the rowspace projection idea of Sarlos [2006] and the downsampling idea of Liberty et al. [2007]. The guarantees of the technique are

$$\delta_F(A, \hat{A}) = \epsilon||A - A_k||_F$$

$$\delta_2(A, \hat{A}) = (1 + \sqrt{2m/d})\epsilon||A - A_k||_2.$$

2.7 Summary of Approximation Methods

We summarize the theoretical guarantees for runtime and accuracy of these papers in Table III. We should note that a concern in much of the literature is the notion of *pass-efficiency*. This refers to the number of times that we need to make a full scan of the data in order to produce our approximation (e.g., we need one scan to compute $||A||_F^2$). This is clearly of importance when we look at data that is only available to us in a streaming fashion; an algorithm requiring even two passes might be infeasible for large data, as it necessitates storing the entire data in memory. We do not focus on this property of the various algorithms, and instead assume that the data is available to us in its entirety.

2.8 Previous Experimental Comparisons

To the best of our knowledge, the methods described above have not been systematically compared in terms of the time taken and the accuracy of the generated approximation. The sampling methods of Achlioptas and McSherry [2007] were compared with standard SVD and a decomposition method in terms of accuracy, but not runtime. The results showed that non-uniform sampling was able to achieve good accuracy, but that uniform sampling's performance degraded for large values of k. The column-sampling method of Drineas et al. [2006b] was compared with standard SVD in Drineas et al. [2001], which found the method to perform quite well on a series of image datasets. Our results below agree with the paper's observation that the column-sampling method performs better than standard SVD on dense data, but we find that on sparse data it performs slightly worse. "Performance" here is used to mean "relative speed," and we shall use this term henceforth in the article.

3. EXPERIMENTAL SETUP

In this section, we describe our goals in empirically analyzing the various methods, and give details of the setup on which we ran the experiments. We intend these descriptions to ease reproducibility of our results. In light of this goal, all source code for our experiments are provided online at `http://www-cse.ucsd.edu/\~akmenon/code`.

3.1 Experimental Aims

The methods discussed above all present theoretical bounds on their runtime and approximation error, but there are important reasons for being interested in empirical results. Firstly, they may reveal hidden constants in the big-O notation that can have a considerable impact on practical performance. Secondly, there is often a discrepancy between what one can say theoretically and what one observes in practice. With most of these algorithms, there is no guarantee of *tightness* of the error bounds; they prescribe the limits of what we can say theoretically, but one can potentially get much better accuracy than predicted when actually running the algorithm. One might also be able to apply the algorithms in scenarios not covered by the assumptions made to prove their accuracy. This fact was noted in the experimental results of Achlioptas and McSherry [2007], which showed that their algorithm performed well even in cases where they could not make any theoretical predictions about accuracy.

Table III. Comparison of the Theoretical Guarantees of Time Complexity and Accuracy of the Various Methods on an $m \times n$ Matrix A, with k Denoting the Desired Rank of Approximation

Algorithm	**Parameter**	**# passes**	**Asymptotic time complexity**	**Error guarantee**		
				$\|\| \cdot \|\|_F^{(2)}$	$\|\| \cdot \|\|_2^{(2)}$	**Type**
Uniform sampling [Achlioptas and McSherry 2001]	sampling factor p	1	$\mathsf{SSVD}(mn(1-p))$	$4z(z+\sqrt{\|\|A\|\|_F})$ $z = \left(\frac{nkb^2}{p}\right)^{1/4}$	$8b\sqrt{n/p}$	Add
Nonuniform sampling Achlioptas and McSherry 2007]	sampling factor p	1	$\mathsf{SSVD}(m(logn)^4)$	$4z(z+\sqrt{\|\|A\|\|_F})$ $A = \left(\frac{nkb^2}{p}\right)^{1/4}$	$8b\sqrt{n/p}$	Add
Drineas [Drineas et al. 2006b]	error ϵ	2	$D + mk^2/\epsilon^4 + k^3/\epsilon^6$	$\epsilon\|\|A\|\|_{F^2}$	$\epsilon\|\|A\|\|_{F^2}$	Add
Sarlos [Sarlos 2006]	error ϵ	2	$Dk/\epsilon + (m+n)k^2/\epsilon^2$	$\epsilon\|\|A - A_k\|\|_F$	N/A	Mult
[Liberty et al. 2007]	l	2	$mnlogl + klnlogn +$ $(m+n)k^2$	N/A	$\delta\|\|A - A_k\|\|_2$, $\delta = \sqrt{4k(n-k)+1} - 1$	Mult
Random PCA [Rokhlin et al. 2009]	i, l	2	$(il+k)D + i^2l^2(m+n)$	N/A	$\delta\|\|A - A_k\|\|_2$, $\delta = Cm^{1/(4i+2)} - 1$, C a constant	Mult
Nguyen [Nguyen et al. 2009]	error ϵ, probability β	2	$mnlogd + (m+n)d^2$, $d = O(klogm)$	$\epsilon\|\|A - A_k\|\|_F$	$(1+\sqrt{2m/d})\epsilon\|\|A - A_k\|\|_2$	Mult

D denotes the number of nonzeros in A, and $b = \max |A_{ij}|$. $||\cdot||_F$, $||\cdot||_2$ denote the Frobenius and 2-norm of a matrix respectively. SSVD(z) denotes the time for sparse SVD computation of a matrix with z nonzeros. “Mult” means multiplicative, “Add” means additive. The runtimes provided assume that the input matrix is sparse, but otherwise unstructured.

Our first aim is thus to look at the performance of the different methods in terms of accuracy and runtime. More specifically, we wish to ask: for a given k, how long does it take to compute an approximation to A_k, and how close is the approximation in the $||\cdot||_F$ or $||\cdot||_2$ sense? These are the two most natural metrics of performance for this class of algorithms, though there are others that might be of interest (e.g. number of passes). The intent of this is to reveal the relative performance of these methods on realistic data.

We are also interested in the performance of the sampling methods of when presented with sparse data. As noted in Section 2.6, Achlioptas and McSherry [2001] randomly sparsify input data to act as a sampling of the entries. This is intuitive for moderate to dense data, but it is a concern when the data is already, as we risk losing the only important information in the data. The method of Achlioptas and McSherry [2007], based on nonuniform sampling, ostensibly addresses this issue by weighting probabilities by the magnitude of entries rather than by a universal constant. We wish to empirically verify the efficacy of such an approach on sparse data, in order to get a sense of whether it is a generally applicable method.

When measuring runtime, we split the time taken for each method into two: the time for the SVD call (recall that in Section 2.5, we said that all methods we compare call a standard SVD function as a subroutine exactly once), and time for pre- and post-processing. For some methods, the pre- and postprocessing is simple: in the sampling techniques of Achlioptas and McSherry [2001], for example, we just need to randomly sparsify the input matrix. Another point worth noting is that the time to form the approximation to A_k is treated as postprocessing, *not* as part of the SVD time; this is because an SVD call simply returns the matrices U, S, V, which in the case of standard SVD need to be multiplied to form A_k. For methods such as Drineas et al. [2006b], however, going from these matrices to the approximation $\hat{A}_k$ requires more computational effort. Observe that even standard SVD will have a nonzero post-processing time because of this convention.

3.2 Methods Compared

As mentioned in Section 2.6, there are several methods in the literature that look to speed up low-rank approximations. We would be unable to provide a thorough comparison of all such methods, and so our scope is necessarily limited. We have looked to choose the most interesting methods from a practical viewpoint, which corresponds to the ones with the best known guarantees and which have been the basis of most novel research in the field. The methods we compare in this article are:

(1) The standard SVD algorithm as implemented by MATLAB and PROPACK; see the next section for details.
(2) The uniform and non-uniform sampling approaches of Achlioptas and McSherry [2001] and Achlioptas and McSherry [2007].
(3) The random projection method of Sarlos [2006].
(4) The column-sampling approach of Drineas et al. [2006b].
(5) The fast PCA approximation of Rokhlin et al. [2009].
(6) The Walsh-Hadamard transform based algorithm of Nguyen et al. [2009].

3.3 Implementation Notes

To the best of our knowledge, no source code has been published for any of the low-rank approximations. Therefore, the implementations on which the following results are reported are based entirely on code that we have written. We make some brief comments on particular implementation choices that arose.

The algorithm in Achlioptas and McSherry [2007] for nonuniform sampling prescribes the use of a priority queue in order to work in a single pass. However, given that Sarlos [2006], for example, operate in two passes, it is unfair to constrain Achlioptas and McSherry [2007] to work in a smaller number of passes. As a result, we made a faster implementation of this nonuniform sampling algorithm that operates in two passes. Similarly, the column-sampling approach of Drineas et al. [2006b] suggests the use of a one-pass algorithm SELECT [Drineas et al. 2006a] to compute the probabilities used for sampling the columns. We instead compute the probabilities directly by finding $||A^{(i)}||_F^2$ and $||A||_F^2$.

When the probability of keeping an element in the sampling approaches [Achlioptas and McSherry 2001, 2007] was above 1, we truncated this down to 1.

After sampling, the question is whether or not to store the matrices in sparse format. Where possible, we tried storing matrices in both MATLAB's sparse and dense format, and the reported results are for the better of these two. Appendix C discusses the choice of storage format in more detail. Where appropriate, we optimized our code for the sparse matrix format. For example, for the sampling methods, extracting the nonzero elements in the matrix as a dense vector and sampling on this is several times faster than sampling on the original matrix.

We also note that the straightforward implementation of Sarlos [2006] will result in inflated runtimes. Such an implementation would involve computing the approximation exactly as stated in the theorem, viz. first finding $\Pi_{RA}(A)$ using the orthogonal projection formula,[4] and then finding the best rank-k approximation to this quantity. While this gives the right answer, it offers no time saving because we are computing the SVD of an $m \times n$ matrix. The correct approach is presented in Deshpande and Vempala [2006]: here, we use the rowspace of RA as a basis in which we can rewrite the vectors of A. This can be achieved computationally by simply multiplying A by the matrix P, defined to be the top r left-singular vectors of RA, where r is the dimensionality of the space spanned by RA. This correctly reduces the problem to finding the SVD of an $m \times k/\epsilon$ matrix. We refer the reader to Deshpande and Vempala [2006] for details.

We chose to implement Nguyen et al. [2009] using the trimmed Walsh-Hadamard transform. This transform uses the Walsh-Hadamard (or natural-ordered Hadamard) matrix H_{2^n}, defined recursively by

$$H_{2^n} = \begin{bmatrix} H_{2^{n-1}} & H_{2^{n-1}} \\ H_{2^{n-1}} & -H_{2^{n-1}} \end{bmatrix}$$
$$H_1 = [1].$$

Observe that the matrix is only defined for subscripts that are powers of 2. We will call the trimmed Walsh-Hadamard matrix PH_{2^n}, where $P \in \mathbb{R}^{k \times 2^n}$ is a downsampled version of I_{2^n} i.e. k columns of P constitute the identity matrix I_k.

For a given vector $x \in \mathbb{R}^n$, the trimmed Walsh-Hadamard transform is defined to be $PH_N\tilde{x}$, where $\tilde{x} = \begin{bmatrix} x \\ \mathbf{0} \end{bmatrix} \in \mathbb{R}^N$ and $N = 2^{\lceil \log n \rceil}$. That is, we pad the input vector with zeros so that its length is a power of 2. This trick is suggested in, for example, Drineas

[4]If P is a matrix whose columns are an orthonormal basis for a space, then $PP^T x$ will project the vector x onto this space.

et al. [2007]. To compute the trimmed transform, following Ailon and Liberty [2008], observe that by writing all the terms in block form,

$$\begin{aligned} PH_N\tilde{x} &= \begin{bmatrix} P_1 & P_2 \end{bmatrix} \begin{bmatrix} H_{N/2} & H_{N/2} \\ H_{N/2} & -H_{N/2} \end{bmatrix} \begin{bmatrix} \tilde{x}_1 \\ \tilde{x}_2 \end{bmatrix} \\ &= P_1 H_{N/2}(\tilde{x}_1 + \tilde{x}_2) + P_2 H_{N/2}(\tilde{x}_1 - \tilde{x}_2). \end{aligned}$$

It can thus be shown that the computation of $PH_N\tilde{x}$ can be done in $O(N\log(k'))$ time, where k' is the number of nonzero rows in P.

In MATLAB, we found the simple recursive version that follows directly from the above formula to be too slow. We experimented with different base cases for the recursion to try to achieve a tradeoff between the overhead per recursive call and the overhead of multiplying by a general order Hadamard matrix for the base case, but were not successful in beating the time of just computing $(PH_N)\tilde{x}$ directly. We then modified code for an iterative version of the fast Walsh-Hadamard transform [Thomas 2007] and found that while it was faster in some cases, it was hampered by its access patterns of the vector x, making it slow in MATLAB. Thus, we ended up computing $(PH_N)\tilde{x}$ directly as this was the fastest solution in MATLAB. We provide details in Appendix A.

3.4 Datasets

We ran experiments on a number of dense and sparse datasets, intended to be characteristic of real-world applications of SVD such as eigenfaces [Turk and Pentland 1991] and latent semantic indexing [Furnas et al. 1988; Papadimitriou et al. 1998b]; the datasets are listed below.

— The ORL database of faces [AT&T Laboratories Cambridge 2002], a collection of 10 images each for 40 subjects, with each image being of size 112 × 92. This was converted into a 400 × 10304 matrix, where each row represents the pixel values for a single image. This data is completely dense, having no zero entries.
— Dataset 2 from [Liberty et al. 2007], which is a 4096 × 4096 matrix with singular values in the range $[10^{-15}, 10^{-25}]$. The matrix is formed by constructing explicitly U, S, V and taking their product. We form U, V via Gram-Schmidt orthonormalization of matrices whose entries are drawn from a standard normal distribution. For a given parameter l, S is defined to be a diagonal matrix for which $\sigma_i = 10^{-15(i-1)/(l-1)}$ for $1 \leq i \leq l$, and $\sigma_i = 10^{-15}$ for $i > l$. We see that l is a measure of the low rank structure of the matrix USV^T. We generated the dataset for $l = 16$. This dataset is also dense.
— The Reuters news dataset Lewis [2004], a 9603 × 22226 term-document matrix which is of interest for two reasons: firstly, it was used in experiments by Achlioptas and McSherry [2007], and secondly, it is extremely sparse (only about 0.23% of the entries are nonzero). While the original dataset is a combination of several document collections, it is treated as a single term-document matrix in the literature; specifically, the literature focusses on the "Mod-Apte" subset of the dataset. It should be noted that most literature reports the size of the dataset as 9603 × 22226; however, we have only achieved a dimensionality of 9603 × 21671 that is, there are around 500 words that we are missing. This is likely because we are being too strict in our definition of what a word is: we automatically reject all numerical values, for example. This difference is negligible, and we do not pursue the issue further.
— A large sparse matrix (`s3dkt3m2`) from the Harwell-Boeing collection [Duff et al. 1998]. This is a 90449 × 90449 matrix arising from a structural mechanics problem. Only about 0.04% of elements are nonzero.

— Synthetically generated dense data, in order to check the impact of sparsity/density on the approaches of Achlioptas and McSherry [2001]. This data is simply a matrix of i.i.d. uniformly random variables.

3.5 Environment

Our experiments are conducted in MATLAB, which is a high-level language with excellent library support for most mathematical applications. In particular, basic matrix operations have efficient implementations.

MATLAB, provides two standard functions for SVD computation, `svd` and `svds`. `svd`[5] is based on the LAPACK library [Anderson et al. 1992], which uses a QR-decomposition algorithm. `svds` on the other hand is based on the iterative Arnoldi method,[6] and while especially suited to sparse matrices, also works on dense matrices. We note that `svd` is a MATLAB, builtin command, while `svds` is simply a script. The `svd` routine computes the full SVD (U, S, V) of a dense matrix, while the sparse version `svds` directly computes the triple (U_k, S_k, V_k), where k is user-specified. This means that if our goal is only to find the truncated SVD, with `svd`, we must explicitly retain only the top k columns from U, S, V; this wastes a lot of time computing unnecessary columns when k is small. We slightly alleviated this problem by making `svd` always find the thin SVD (see Section 2.4) by default.

Aside from the built-in SVD functions, there are several external SVD packages for MATLAB, that outperform the builtin methods under certain conditions. One such popular package is PROPACK [Larsen 2005], which is an extended Lanczos bidiagonalization method. Like `svds`, PROPACK allows us to find the truncated SVD directly. PROPACK offers support for MEX files, which we did not use to ensure that all methods were based on MATLAB, primitives, making for a fairer comparison.

Each of these different functions for computing the SVD performs well in specific cases, and so choosing the right method for a particular matrix requires some care. In order to compare the various algorithms fairly, our strategy is to use all three methods in parallel and then report the runtime of SVD computation as the fastest of the three. The exception to this is when we know a priori that a certain method is infeasible for a particular computation, for example, using `svd` on a very large sparse matrix is not feasible. In the following section, we refer to the best of all three methods as being the "baseline SVD method."

All tests were run on a 2GHz AMD Opteron server with 8GB of RAM, running CentOS 4.6.

3.6 Parameters

There are a few user-modifiable parameters for the approximation methods we compared. Here, we describe how we set these parameters. The goal in setting all these parameters is ensuring that we have some sensible comparison of the methods. One way to do this is to choose the parameters so that the methods have similar error guarantees. As we discuss in the following, this is sometimes not feasible (for example, it might require that we project to a subspace with dimensionality larger than k, the desired rank of the approximation). In such cases, we have to move away from the theoretical guarantees and look at empirical performance. Our goal in such cases was to achieve comparable accuracy to other methods.

[5]http://www.mathworks.com/access/helpdesk/help/techdoc/ref/svd.html
[6]http://www.mathworks.com/access/helpdesk/help/techdoc/ref/eigs.html

We now describe the choice of parameters in detail.

(1) The uniform sampling method [Achlioptas and McSherry 2001] uses a sparsification parameter p that determines how many entries from the original matrix are retained on average. Observe from Table III that if we choose

$$p = s \cdot \frac{nb^2}{||A||_F^2}, \tag{4}$$

then the error guarantees we get are

$$|| \cdot ||_F : 4\left[\left(\frac{k}{s}\right)^{1/2} + \left(\frac{k}{s}\right)^{1/4}\right]||A||_F$$
$$|| \cdot ||_2 : \frac{8}{\sqrt{s}}||A||_F.$$

To make the bounds comparable with that of the other methods, one can choose $s = \frac{64}{\epsilon^2}$, so that $p = \frac{64nb^2}{\epsilon^2||A||_F^2}$. The lack of dependence on k means that we can expect the Frobenius error to increase as k increases. Section 4.6 looks at the effects of changing the sparsity parameter. Note that by virtue of the optimal p depending on n, we cannot keep p fixed and have commensurate error guarantees on the various datasets.

Unfortunately, we found that choosing p with the above formula often gave values larger than 1, which is not useful because it means we sample every element. There are several options here.

(a) The simplest approach is to eschew the above formula and set p to be some global constant for all datasets, for example $p = 0.1$ so that every dataset has 10% retention. This has the advantage of being very simple to implement. Of course, it is somewhat naïve, because there is no reason to believe that any fixed sampling factor will work well for all the datasets; indeed, the theoretical guarantee says that such an approach will cause the error to depend on $\sqrt{n}$.

(b) One can slightly tweak the formula for p to ensure it is always smaller than 1. It is clear that the factor in the formula for p that varies for different matrices is $\frac{nb^2}{||A||_F^2}$: this term is sometimes greater or smaller than 1. If we can ensure that it is smaller than $\frac{\epsilon^2}{64}$, then we can guarantee that p is smaller than 1. One way to do this is to apply a sigmoid transform σ with scaling: we found that we could generate reasonable values of p with

$$p = \frac{1}{2\epsilon^2} \cdot (2\sigma(nb^2/||A||_F^2) - 1). \tag{5}$$

It is possible to consider transformations depending on properties of the matrix A, such as $p \mapsto \frac{p}{\sqrt{mn}}$, but we found this gave p values that were far too small.

(c) We can empirically choose p based on what value gives good results, at the expense of any theoretical guarantee of accuracy. This assumes that the theoretical guarantees are extremely conservative, and so the hope is that even for small p we can achieve high accuracy. However, we found the choice of p from the previous method to be good, and since it has some relationship to the formal bound, we thought it preferable.

Table IV gives the retention probabilities on the various datasets.

(2) The nonuniform sampling method [Achlioptas and McSherry 2007] has the same error guarantee as uniform sampling, and also takes as input a parameter p to con-

Table IV. Uniform Sampling Probabilities for Each Dataset

Dataset	p_{unscaled}	p_{scaled}
Faces	2.66	0.25
Reuters	7681.60	0.50
Liberty et al.	4.64	0.25
Harwell-Boeing `s3dkt3m2`	584.65	0.45

p_{unscaled} is if we apply the original formula Equation (4), and p_{scaled} is when we apply the sigmoid transform and scaling Equation (5).

Table V. Scaling Factors α for Each of the Datasets, Nonuniform Sampling

Dataset	α	p_{scaled}
Faces	26600	0.0001
Reuters	22000	0.35
Liberty et al.	2200	0.002
Harwell-Boeing	0.008	75000

trol the sparsification. We can choose this p in one of two ways: we can try to match the number of elements retained by uniform sampling, or we can try to achieve the same error bound as uniform sampling. We chose the former, because the latter is difficult to achieve. The reason for this difficulty is that with non-uniform sampling, there are additional constraints placed on the sampling probabilities. Recall from Section 2.6 that the probability of retaining an element is proportional to

$$\max\left\{\sqrt{p}\frac{|A_{ij}|}{b}, \frac{64(\log n)^2}{\sqrt{n}}\right\}.$$

If n is large enough, the second term can dominate and push the sampling probability to be greater than 1, which is again not useful. To get around this, one needs to set p to be sufficiently small to offset this term. As with uniform sampling, setting $p = \frac{64nb^2}{\epsilon^2||A||_F^2}$ gives comparable bounds to the other methods. We set $p = \frac{64nb^2}{\alpha\epsilon^2||A||_F^2}$, where α is a constant chosen so that we retain roughly the same number of entries as uniform sampling. Clearly, this constant had to be tweaked depending on the dataset at hand, and the choices for the different datasets are listed in Table V.

(3) The other methods use an error parameter ϵ to determine how accurate their respective solutions are. We chose $\epsilon = 0.25$, a moderate choice for which the methods ought to run fairly quickly and produce a solution with reasonable accuracy.

(4) For the algorithm of [Nguyen et al. 2009], the most conservative choice for d, which appears in the proof of the main theorem of that paper, is

$$d = \frac{C}{\epsilon}\max\{k, \sqrt{k}\log(2m/\beta)\}\max\{\log k, \log(3/\beta)\},$$

where $C = \max\{5C_1, C_2\}$ for constants C_1, C_2. Plugging in the upper bounds $C_1 \leq 25$ and $C_2 \leq \frac{125}{16}\cdot 127$, this says

$$d = \frac{1}{\epsilon}\cdot\frac{125\cdot 127}{16}\max\{k, \sqrt{k}\log(2m/\beta)\}\max\{\log k, \log(3/\beta)\}.$$

If we let the probability of failure $\beta = 0.75$ and $\epsilon = 0.25$ as with the other methods, then we find this d to be quite large in some of our experiments. For example, on

Table VI. Leading Constants C used to Compute d for the Algorithm of [Nguyen et al. 2009]

Dataset	C
Faces	0.7
Reuters	0.2
Liberty et al.	Chosen so that $d = \frac{k}{\epsilon}$
Harwell-Boeing	Chosen so that $d = \frac{k}{\epsilon}$

Table VII. The Value of $k_{\max}$ for Each Dataset

Dataset	$k_{\max}$
Faces	50
Reuters	300
Liberty et al.	16
Harwell-Boeing	250

the Faces dataset, it is more than 5000 even for small k, which means it is bigger than the rank of the original matrix (viz. 400). Given that the above bounds on C_1, C_2 are quite conservative, we tried to scale down this constant factor. We found that even for a very small leading constant C, the resulting SVD approximation was close to the optimal one. As with the sampling methods, we can view this as saying that the practical performance of the algorithm is much better than what is guaranteed. Table VI summarizes the choice of constant C for the various datasets.

The other parameter for all methods, including standard SVD, is of course the rank of the approximation, k. In practice, the choice of this parameter depends on the dataset and application at hand. For each dataset, we chose 11 equally spaced points in the interval $[1, k_{\max}]$, where the $k_{\max}$ values for each dataset are listed in Table VII. We chose these $k_{\max}$ values based on standard values that appear in the literature. For example, in eigenfaces applications on the Faces dataset, $k_{\max} = 50$ is known to give good results for face recognition (see e.g., [Tjahyadi et al. 2004]).

4. EXPERIMENTAL RESULTS

In this section, we present our experimental results, divided according to the dataset being used.

4.1 Results on Faces Data

Figure 1 shows the accuracy results for the Faces dataset. The results indicate that most of the methods are successful, but the sampling approaches of Achlioptas and McSherry [2001] have worse performance with larger k for the Frobenius norm. As we need an approximation method to improve as the rank k increases, this implies that the uniform sampling method has limited use as a standalone method. For the spectral norm, the sampling methods stabilize at a fixed error rate, as predicted theoretically, but the error is relatively high compared to the baseline SVD method. The other four methods follow the baseline SVD curve fairly closely, and are more accurate as the value of k increases. We note that this is despite the fact that not all these methods provide guarantees for both the Frobenius and spectral norm.

Figure 2 shows the average runtime over the range of k values on the Faces dataset. Recall from Section 3.1 that we split the time into that for the SVD subroutine call (denoted "SVD time"), and the time used for pre- and post-processing in the method (denoted "Other time"). Recall also that pre- and post-processing includes the time to actually form the approximation matrix A_k, which is why even the baseline SVD

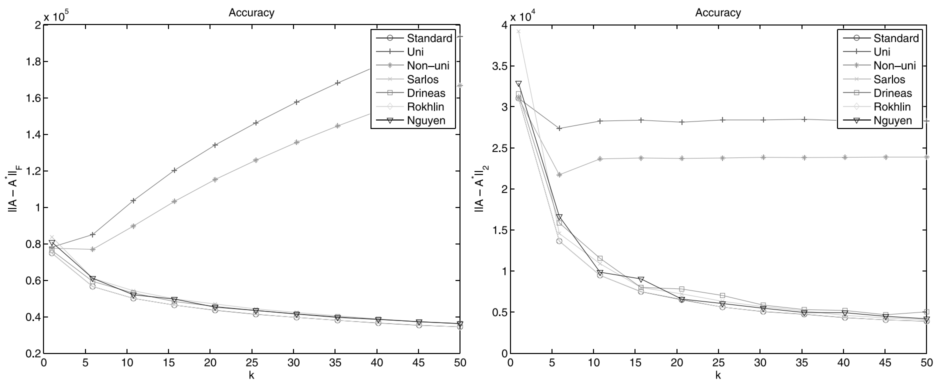


Fig. 1. Accuracy of all methods on the Faces dataset, Frobenius and spectral norms.

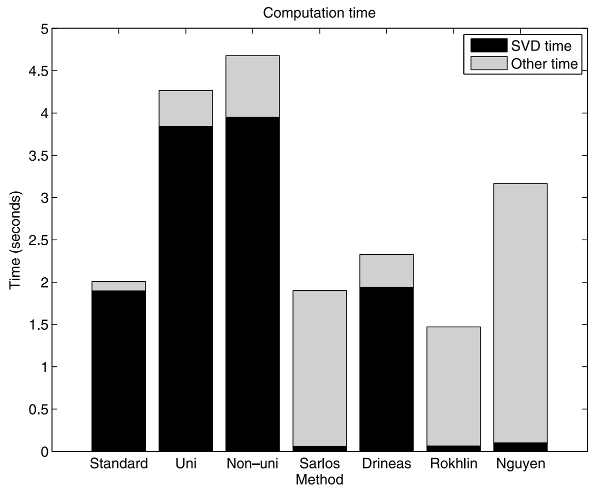


Fig. 2. Time for each method on Faces dataset (average of range of k). See text for definitions of "SVD" and "Other" time.

method has a nonzero value for this portion. Interestingly, only the methods of Sarlos and Rokhlin et al. manage to run faster than the baseline SVD method, and even for these methods, the speedup is marginal. The three embedding methods all spend the majority of their time doing pre- and post-processing, suggesting that while their respective SVD calls are dramatically faster than that on the original input matrix, these savings are outweighed by the pre- and post-processing required to return an approximation to the true SVD. The column sampling method of Drineas et al. [2006b], by contrast, spends most of its time on the SVD call and relatively little time on other processing. This is because the method takes the SVD of a $k/\epsilon^2 \times k/\epsilon^2$ matrix, which requires computation that is cubic in k.

The fact that the sampling methods of Achlioptas do not perform well is rather surprising. It is especially interesting that the time for the actual SVD computation is not improved after sampling of the original matrix: in fact, this time *increases*. We observed that the SVD computation on the original matrix and the sampled matrices is the best with the PROPACK method, while the Drineas and Sarlos methods were

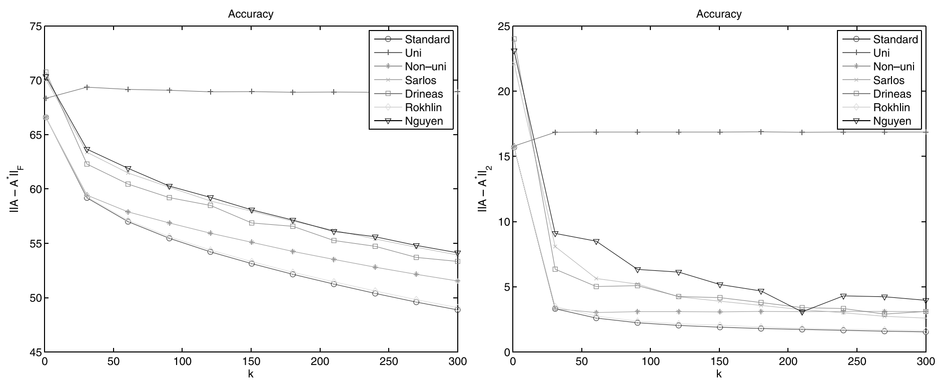


Fig. 3. Accuracy of all methods on the Reuters dataset, Frobenius, and spectral norms.

fastest with `svd`. This suggests that the PROPACK method does not necessarily perform better on sparser input. We found that if forced to use one of `svd` or `svds`, the sampling methods' SVD time was lower than that of standard SVD; however, these functions take around twice the time that PROPACK does. This issue is addressed more in the Appendix C.

4.2 Results on Reuters Data

We compared all the methods on the Reuters dataset, and the accuracy of all methods is presented in Figure 3. The accuracy results for the 2-norm are similar to those for the Frobenius norm. Overall, the results are mostly similar to the ones for the Faces dataset, which indicates that the methods are relatively agnostic about the sparsity of the input data as far as accuracy goes (we study the behaviour of the sampling methods of Achlioptas and McSherry [2007] in more detail in Section 4.6). There a couple of differences to the results on Faces, however. First, the accuracy uniform sampling does not degrade for larger k with the Frobenius norm. Second, we see that there is greater separation of the methods compared to the baseline error, with the method of Rokhlin et al. giving the highest accuracy.

The runtimes on the Reuters data are shown in Figure 4. Compared with the results on Faces, we see that the three embedding methods and the column sampling method of Drineas et al. [2006b] all manage to offer significant savings compared to standard SVD. The best performing method is Drineas et al. [2006b], which more than halves the total computation time required to get a good approximation. This is in contrast to the results on Faces, where it was hampered by an excessive cost in its SVD call. If we look at the breakdown of the runtimes, we note that in all cases we get significant savings in the actual SVD computation, with the Sarlos method providing nearly an order of magnitude saving. This is in contrast to results for the Faces data, where the sampling methods' SVD time actually increased. However, the pre- and postprocessing times curtail the overall savings compared to standard SVD. The reason for this is that the Reuters matrix is stored in a sparse matrix format, for which standard matrix operations are not as quick as on their dense counterparts.

4.3 Results on Dataset 2 from Liberty et al.

The accuracy results for the dataset from Liberty et al. [2007] are given in Figure 5. The sampling methods give the highest error rate, and as with the Faces dataset this rate increases with k for the Frobenius norm but stabilizes for the spectral norm. Note

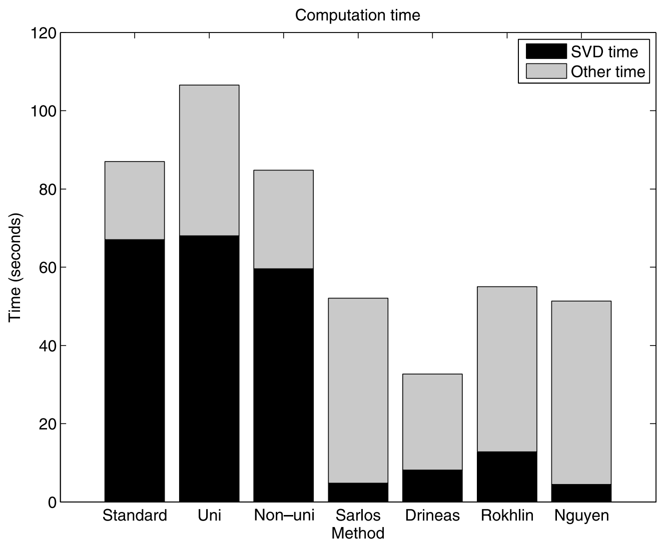


Fig. 4. Time for each method on Reuters dataset (average of range of k). See text for definitions of "SVD" and "Other" time.

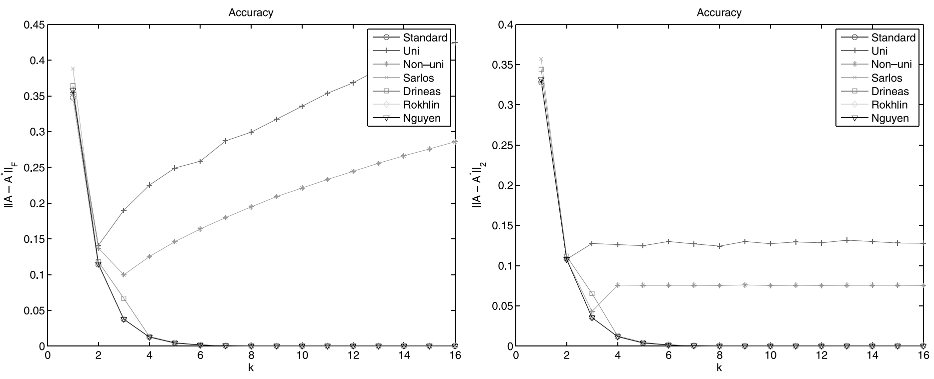


Fig. 5. Accuracy of all methods on the Liberty et al. dataset, Frobenius and spectral norms.

that the error rate for all other methods was identical to that of the baseline SVD algorithm for moderate k values, hence all their data points are collapsed onto one.

In terms of runtime, the results in Figure 6 show that *all* methods offer significant savings compared to the baseline SVD method. As before, the column sampling method and embedding methods manage to dramatically reduce the time required in their respective SVD calls, and add minimal overhead from pre- and postprocessing. As distinct from the results on the Faces data, the sampling methods also take much less time than the baseline method. This is because on the Faces dataset, their SVD calls took *longer* than the one on the original input matrix; here, by contrast, they take only a fraction of that time. As stated earlier, and discussed in the Appendix C, the precise runtime of SVD on a sparse matrix is unclear; in this particular case, one possible explanation is that the *eigengap* of the original matrix (referring to the difference between the kth and $(k + 1)$st eigenvalues) is small.

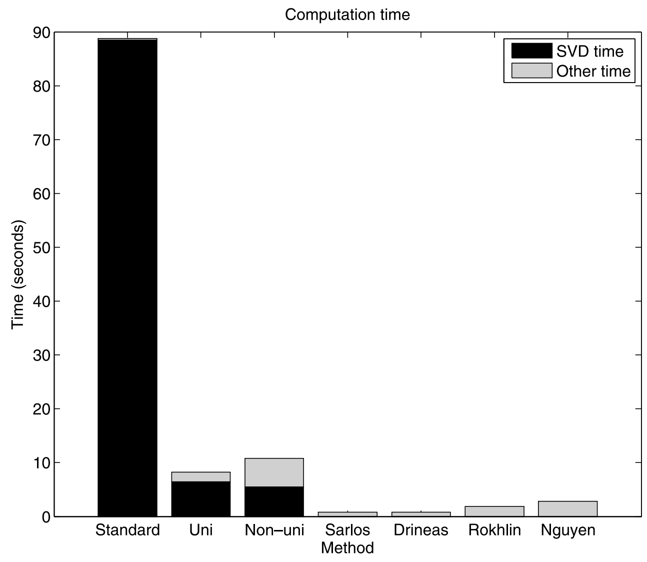


Fig. 6. Time for each method on the Liberty et al. dataset (average of range of k). See text for definitions of "SVD" and "Other" time.

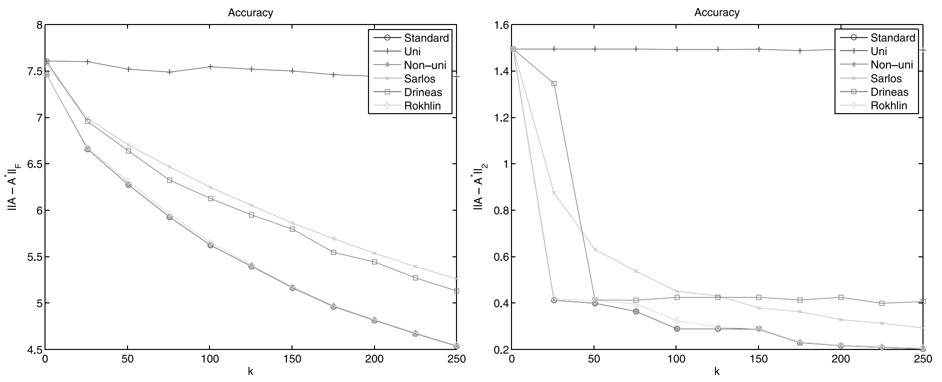


Fig. 7. Accuracy of all methods on the Harwell-Boeing `s3dkt3m2` dataset, Frobenius and spectral norms.

4.4 Results on Harwell-Boeing Data `s3dkt3m2`

The accuracy results on the Harwell-Boeing dataset `s3dkt3m2` are given in Figure 7, and the timing results in Figure 8. Both sets of results are similar to the Reuters dataset. In particular, note that the column sampling method of Drineas et al. [2006b] is the fastest, cutting down the computation time to a third of the baseline. One interesting difference to the Reuters results is that uniform sampling manages to roughly halve the "SVD time" of the baseline method, whereas we previously found that sparsification increased the time of this step.

The results on this dataset do not include the method of Nguyen et al. [2009], because we could not get the method to run with our MATLAB, implementation. This is because, as mentioned earlier, we found the method of Nguyen et al. [2009] runs fastest when we explicitly do a matrix multiplication with the Hadamard matrix. This is not feasible for large matrices such as `s3dkt3m2`.

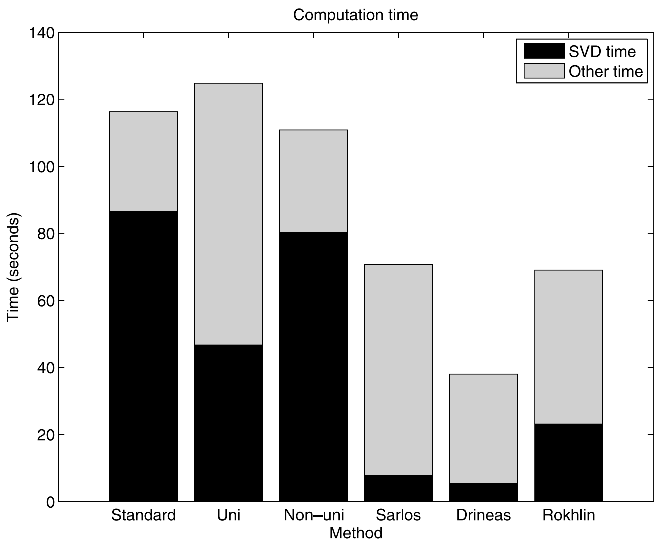
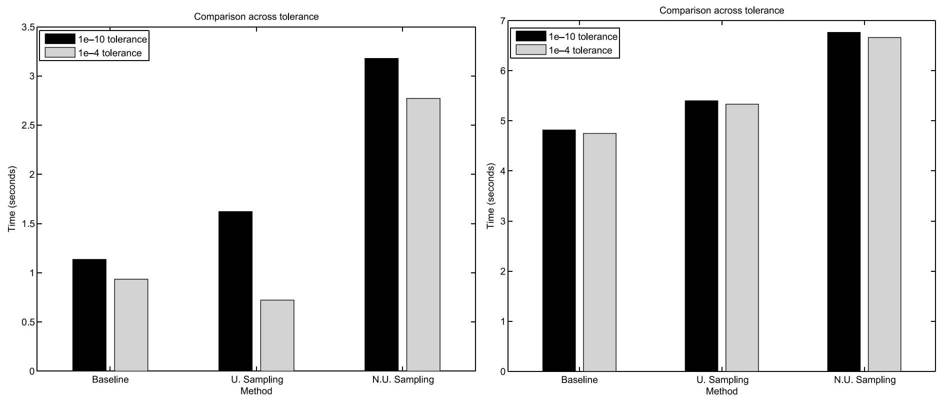


Fig. 8. Time for each method on the Harwell-Boeing `s3dkt3m2` dataset (average of range of k). See text for definitions of "SVD" and "Other" time.



(a) Faces data. (b) Reuters data.

Fig. 9. Time for reduced tolerance $\Delta = 10^{-4}$.

4.5 Modifying Tolerance Level

We established in Section 2.4 that standard SVD algorithms are iterative, and stop either when they reach a local optimum that is within some predefined tolerance parameter Δ, or when a maximum number of iterations I are executed. We did experiments to see whether reducing these values has a significant impact on the accuracy and timing results for the baseline SVD algorithm and the sampling methods of Achlioptas. Both the PROPACK and `svds` functions allow one to specify the parameters Δ and I, but the standard `svd` function does not; since the Sarlos and Drineas methods are fastest with `svd`, we were not able to tweak the parameters for these methods. We found that stopping criterion for all methods was always the tolerance parameter Δ, so we did not try to vary I. We ran the experiments of the previous section for $\Delta = 10^{-4}$, and compared them to the MATLAB, default of $\Delta = 10^{-10}$.

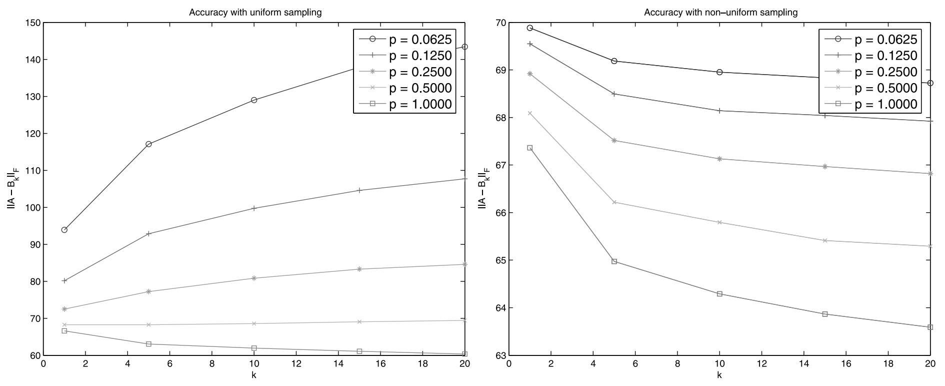


Fig. 10. Frobenius-norm accuracy of uniform vs. non-uniform sampling with varying sparsity parameter, Reuters dataset.

The runtime results for the Faces data are shown in Figure 9(a). We see that the methods all see a benefit in the runtime, especially the uniform sampling approach. The corresponding loss in accuracy is negligible, with a relative error of at most 1% in the case of the nonuniform sampling approach.

The results on the Reuters dataset are shown in Figure 9(b), and indicate that all methods converge in roughly the same time. This means that the SVD computations of these methods converge relatively quickly compared to the Faces data, and in particular, changing the convergence criterion does not improve the runtime of sampling methods on sparse data.

4.6 Effects of sparsification Parameter in Sampling Methods

While the results above indicate that the sampling methods in Achlioptas and McSherry [2007] are outperformed by more recent methods, they are still of interest. They show how a simple sparsification can theoretically ease the computation of low-rank approximations, and can also potentially act as a preprocessing step for other methods (we look at this more in Section 4.7). To study the behavior of the sampling methods under increased sparsity, we ran an experiment that tests accuracy on the Reuters dataset with different sampling rates. The results are shown in Figure 10. We can see that for uniform sampling, as predicted by the error bound, the error gets worse as we increase the sparsity of the input. Note that only for the case $p = 1$ (where we retain the original matrix) does the error decrease as a function of k; in all other cases, the error grows with larger k, matching the bound of $k^{1/4}$ growth (Section 2.6, Equation (3)).

For nonuniform sampling, the results indicate that high p gives us matching performance with the baseline case $p = 1$. Regardless of the sparsification parameter p, the error decreases as a function of k, with the gap between the contours lessening as we increase the sparsity. We can see that the nonuniform method is therefore able to adapt to the sparsity of the data, and is not as sensitive to the properties of the input.

To better understand how the sampling methods differ, we ran another experiment on a random 1000×1000 matrix. Here, we set $k = 1$ and sampled the data using both methods. We then measured the density of the sample, and the accuracy of the approximation. Figure 11 shows the results for the uniform sampling algorithm. The x-axis is plotted on an inverse log scale, so smaller x-values denote a larger parameter p. We see that the value $p = 1$ causes a density that is close to 100%, and very small values of p cause a density that approaches 0%, both of which are as expected. The

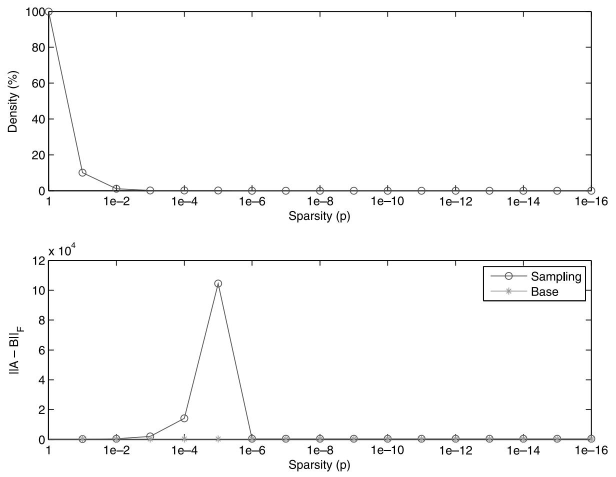


Fig. 11. Uniform sampling sparsity and error on a random dense 1000 × 1000 matrix.

density of the matrix dies down quickly with p, because we are forcibly reducing the density by a constant amount each time. The error in this approach has a very large spike for $p = 10^{-5}$, indicating that it does not deal with large sparsity very gracefully. Note that the error does not become unboundedly large for small p because in this case, the sampled matrix approaches the zero matrix: hence, the error approaches the constant quantity $||A||_F$.

Figure 12 shows the results of the same procedure for the nonuniform sampling algorithm. The behaviour for $0 < p < 1$ indicates an exponential decay in the density as we make the p parameter smaller. Comparing this to the curve for the uniform method, we see that non-uniform sampling is far more moderate in its sparsifying. If we then look at how the error behaves, we note that as the density tends to zero, our error stabilizes at some fixed amount. Note the different vertical axis for this subfloat compared to Figure 11. This behavior is in contrast to the spike we saw with the uniform sampling approach. These results indicate that adaptive sampling is a more robust approach which tunes itself to the data at hand.

4.7 Combining Methods

While the uniform sampling method did not perform well in the above experiments, a reasonable idea is to use it as a preprocessing step for another approximation method. However, doing so naïvely can lead to a large error in the approximation. More precisely, if we apply sparsification first and then run an approximation method on the resulting matrix, we can get a poor approximation. To see this, consider a matrix A with truncated SVD A_k. The uniform sampling procedure gives a matrix A^* that satisfies

$$||A - A^*||_F \leq ||A - A_k||_F + \epsilon||A||_F,$$

where $\epsilon = (nk/p)^{1/4}/\sqrt{||A_F||}$. If we take e.g. Sarlos' projection method, we get a matrix A^s that satisfies

$$||A - A^s||_F \leq (1 + \epsilon)||A - A_k||_F.$$

Now suppose we try to speed up Sarlos's method by first pre-processing A with Achlioptas's procedure. That means we apply the Sarlos's method on A^*, getting a

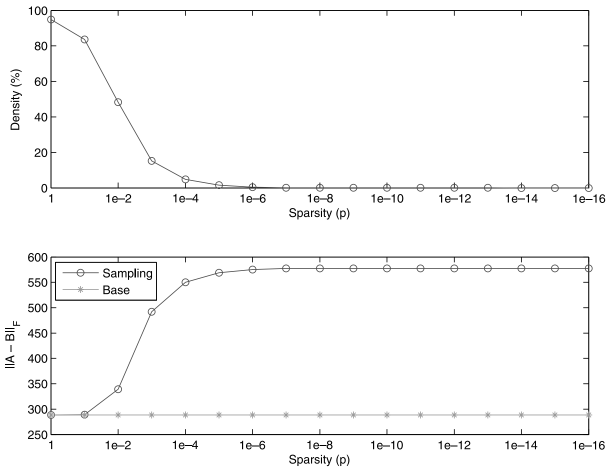


Fig. 12. Adaptive sampling sparsity and error on a random dense 1000 × 1000 matrix.

matrix A^{*s}. So, our guarantee is

$$||A^* - A^{*s}||_F \leq (1+\epsilon)||A^* - A^*_k||_F.$$

But the item of interest is $||A - A^{*s}||_F$. We can bound this using the triangle inequality:

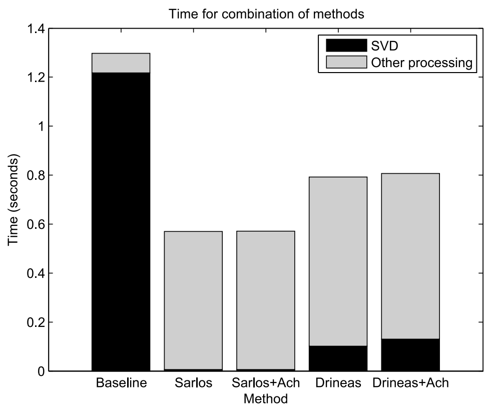
$$\begin{aligned} ||A - A^{*s}||_F &\leq ||A - A^*||_F + ||A^* - A^{*s}||_F && (6)\\ &\leq ||A - A_k||_F + \epsilon||A||_F + (1+\epsilon)||A^* - A^*_k||_F. && (7)\end{aligned}$$

The trouble with this bound is that we cannot guarantee that $||A^* - A^*_k||_F \approx ||A - A_k||_F$. While the matrices A and A^* are close in the sense that the norm of their difference is nearly optimal, that doesn't mean they are equally close to their rank k approximations.

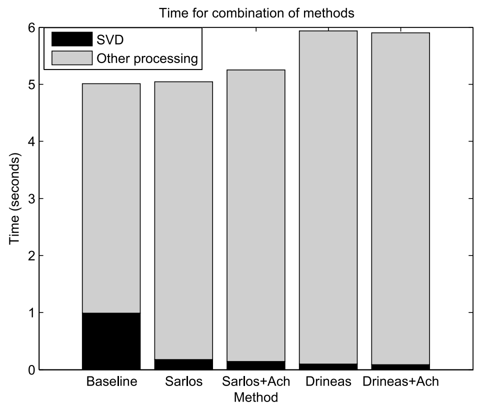
The other approach is to use the sampling procedure on the matrix we make an SVD call on: as noted in Section 2.6, most approximation methods end up computing the SVD of a simpler matrix than the original one. However, the problem with this, given the results in the previous sections, is that the SVD times of the Sarlos and Drineas methods are already very small. Further, we observed that the SVD computation time for the Achlioptas sampling methods actually *increased* compared to standard SVD. This suggests that we might not get savings from combining the sampling method with the Sarlos or Drineas methods. This intuition is corroborated in our results, shown in Figure 13. We see that the combination of the sampling and Sarlos/Drineas methods does not give any savings in runtime.

5. CONCLUSION AND FUTURE WORK

Low-rank approximations are of considerable interest in many practical applications. The SVD is the most popular low-rank approximation, but is quite expensive to compute. There has been a significant body of research devoted to speeding up the computation of a low-rank approximation that gives comparable error to SVD. We have empirically analyzed several of these methods on different datasets, and conclude that the column-sampling method of Drineas et al. [2006b] gives excellent accuracy with a low runtime. While the method makes relatively expensive SVD calls compared to other methods, it makes up for this with low pre- and post-processing overhead. The projection methods of Sarlos [2006] and Rokhlin et al. [2009] also perform well on all datasets. Their comparable performance is attributable to the fact that the latter



(a) Time taken for combination of methods, Faces data.



(b) Time taken for combination of methods, Reuters data.

Fig. 13. Time for combination of methods.

method is a generalization of Sarlos [2006]. A minimum desideratum of any low-rank approximation method is that its accuracy improves as k increases, and all methods meet this except the simple uniform sampling method on the Frobenius norm.

One important application of low-rank approximations is on text data, which is intrinsically sparse. Our results suggest that while such datasets add some overhead to the SVD approximation methods by virtue of their storage mechanisms, we are still able to achieve significant savings in computation time. For example, the method of Drineas et al. [2006b] more than halves the time required by the baseline SVD method on sparse data.

A natural idea is to combine the uniform sampling method of Achlioptas and McSherry [2001] with the other better performing methods. We showed that when combined with the methods of Sarlos [2006] and Drineas et al. [2006b], the resulting algorithm actually causes a slight *increase* in the computation time for the approximations.

An unanswered question in the results is why the column-sampling of Drineas performs as well as it does. The theoretical bound tells us to expect an additive error of about $\epsilon||A||_F$, but we observe the method to give results very close to optimal. It is possible that one can make a stronger statement about the method, especially given the work of Deshpande and Vempala [2006], who used a similar approach to obtain a $(1+\epsilon)$ multiplicative error similar to Sarlos [2006].

APPENDIXES

A. THE TRIMMED WALSH-HADAMARD TRANSFORM

Assume that $N = 2^n$ for some $n \in \mathbb{Z}$, and that[7] $x \in \mathbb{R}^N$. Recall that the Walsh-Hadamard transform is $H_N x$, where $H_N = \begin{bmatrix} H_{N/2} & H_{N/2} \\ H_{N/2} & -H_{N/2} \end{bmatrix}$ and $H_1 = [1]$. The trimmed Walsh-Hadamard transform is $PH_N x$, where $P \in \mathbb{R}^{k \times N}$ is a downsampling matrix, some subset of whose columns form the identity matrix I_k. The naïve $O(Nk)$ solution to compute the transform is to first find PH_N, which can be done in $O(Nk)$ time by just selecting appropriate rows of H_N, and then explicitly multiply this with x, which also takes $O(Nk)$ time. If $k = o(\lg n)$, this approach is superior to finding $H_N x$ and then selecting the k appropriate rows. There are several problems with this: (i) finding H_N explicitly might require too much memory, (ii) it is slightly involved to compute just the ith row of H_N, and (iii) we lose a lot of the structure of the Hadamard matrix this way. The goal of this section is to develop an iterative algorithm to compute $PH_N x$ faster.

We will build up from the iterative algorithm for the Walsh-Hadamard transform implemented in Thomas [2007]. This algorithm is summarized in Algorithm 1. In the following, we explain how it computes $H_N x$ in $O(N \log N)$ time.

Algorithm 1. Algorithm for the Fast Walsh-Hadamard transform, from [Thomas 2007].

```
for $i_1 = 1, \ldots, \lg N$ do
  for $i_2 = 1, \ldots, 2^{i_1 - 1}$ do
    for $i_3 = 1, \ldots, \frac{N}{2^{i_1}}$ do
      $i \leftarrow i_3 + (i_2 - 1)\frac{N}{2^{i_1 - 1}}$
      $j \leftarrow i + \frac{N}{2^{i_1}}$

      $A \leftarrow x_i$
      $B \leftarrow x_j$
      $x_i \leftarrow x_i + x_j$
      $x_j \leftarrow x_i - x_j$
    end for
  end for
end for
```

First, the $O(N \lg N)$ runtime is easy to establish: the runtime of the operations in the innermost loop are clearly $O(1)$, so the total time is

$$\sum_{i=1}^{\lg N} \sum_{i_2=1}^{2^{i_1-1}} \sum_{i_3=1}^{N/2^{i_1}} O(1) = \sum_{i_1=1}^{\lg N} O\left(\frac{N}{2^{i_1}} \cdot 2^{i_1-1}\right) = O(N \lg N).$$

[7]If this assumption does not hold, as discussed earlier we can just pad x with zeros.

To understand how the algorithm works, we will make the following claims.

CLAIM A.1. *Let $\otimes$ denote the Kronecker matrix product and $\oplus$ the direct sum operator. Let $A \in \mathbb{R}^n$ and I_n denote the $n \times n$ identity matrix. Then,*

$$I_n \otimes A = \underbrace{A \oplus \ldots \oplus A}_{n} = \begin{bmatrix} A & 0 & \ldots & 0 \\ 0 & A & \ldots & 0 \\ 0 & 0 & \ddots & 0 \\ 0 & 0 & 0 & A \end{bmatrix}.$$

PROOF. See Yarlagadda and Hershey [1997, p. 2]. □

CLAIM A.2. *The Hadamard matrix H_N can be written as*

$$H_N = \prod_{i=1}^{\lg N-1} I_{2^{i-1}} \otimes J_{N/2^i},$$

where

$$J_n = \begin{bmatrix} I_n & I_n \\ I_n & -I_n \end{bmatrix} \in \mathbb{R}^{2n\times 2n}.$$

PROOF. See, for example, Yarlagadda and Hershey [1997, p. 17]. □

CLAIM A.3. *For a fixed i_1, the innermost loop of the algorithm performs the operation*

$$x \mapsto (I_{2^{i_1-1}} \otimes J_{N/2^{i_1}})x$$

where J is as defined above.

PROOF. Let us fix i_2 as well. Then the innermost loop iterates over $i_3 = 1, \ldots, \frac{N}{2^{i_1}}$ and so $i = 1+(2i_2-2)\frac{N}{2^{i_1}}, \ldots, (2i_2-1)\frac{N}{2^{i_1}}$ while $j = 1+(2i_2-1)\frac{N}{2^{i_1}}, \ldots, 2i_2\frac{N}{2^{i_1}}$. We then compute $x_i + x_j$ and $x_i - x_j$. It is easy to see that this means i iterates over the left half columns of the i_2nd matrix $J_{N/2^{i_1}}$, while j iterates over the right half columns. The $x_i + x_j$ term arises because the top half of the matrix is $\begin{bmatrix} I_{N/2^{i_1}} & I_{N/2^{i_1}} \end{bmatrix}$, and the $x_i - x_j$ term arises since the bottom half is $\begin{bmatrix} I_{N/2^{i_1}} & -I_{N/2^{i_1}} \end{bmatrix}$. □

We see that the algorithm proceeds by doing a series of careful matrix multiplications explicitly. In particular, i_1 iterates over each of the individual matrices in the product defined in Claim A.2. i_2 iterates over the blocks within each matrix; that is, each of the J_n's. i_3 iterates over the elements within each block, and since these are blocks of identity matrices, the computation can be done with just addition and subtraction.

Now let us extend it to perform the trimmed transform. Suppose that $k = 1$ so that we select just a single entry, call it $p \in \{1, \ldots, n\}$. Then, we have effectively erased out all but one row of the matrix $I_{N/2} \otimes \tilde{H}_1$. But that implies that we can erase out the corresponding columns for the other matrices in the expansion for the Hadamard matrix. Doing the necessary calculations reveals the following.

CLAIM A.4. *Suppose that we wish to select just a single element p from $H_N x$. Then, we can modify the block matrix*

$$I_{2^{i-1}} \otimes J_{N/2^i} = \begin{bmatrix} J_{N/2^i} & 0 & \dots & 0 \\ 0 & J_{N/2^i} & \dots & 0 \\ 0 & 0 & \ddots & 0 \\ 0 & 0 & 0 & J_{N/2^i} \end{bmatrix}$$

by zeroing out all blocks except the $1 + \lfloor \frac{p}{N/2^{i-1}} \rfloor$th one. Further, within this block, we only need either the top or bottom half, depending on whether the i_1th bit in the binary expansion of p is 0 or 1 respectively.

This implies that the second loop, which moves between blocks, is no longer necessary: we can just fix $i_2 = 1 + \lfloor \frac{p}{N/2^{i-1}} \rfloor$. Further, we only need to do one of the addition or subtraction steps. The choice depends on the binary expansion of p: when the i_1st term is 0, then we do addition, else subtraction. Algorithm 2 summarizes the modified algorithm.

Algorithm 2. Algorithm for the trimmed Fast Walsh-Hadamard transform, based on Algorithm 1.

```
$P$ := indices that we want to select
$p^{\text{bin}} \leftarrow$ binary expansion of $p$ (with $\log N$ digits)

for $i_1 = 1, \dots, \lg N$ do
  for $p \in P$ do
    $i_2 \leftarrow 1 + \lfloor \frac{p}{N/2^{i_1 - 1}} \rfloor$
    for $i_3 = 1, \dots, \frac{p}{N/2^{i_1}}$ do
      $i \leftarrow i_3 + (i_2 - 1)\frac{N}{2^{i_1 - 1}}$
      $j \leftarrow i + \frac{N}{2^{i_1}}$

      $A \leftarrow x_i$
      $B \leftarrow x_j$

      if $p^{\text{bin}}_{i_1} == 0$ then
        $x_i \leftarrow x_i + x_j$
      end if
      if $p^{\text{bin}}_{i_1} == 1$ then
        $x_j \leftarrow x_i - x_j$
      end if
    end for
  end for
end for

return $\{x_p : p \in P\}$
```

Matlab implementation. There are some issues worth pointing out in our MATLAB, implementation of the above algorithm.

— Since MATLAB, stores matrices in column-major order, this is the preferred method of access of matrices. Unfortunately, the way the Hadamard transform operates is by modifying one coefficient of the transform at a time. Given n data points $X \in \mathbb{R}^{m \times n}$, that means that simultaneously computing the coefficient for all n points would require accessing a row of the input matrix. To get around this, we perform the one-off expensive operation of transposing the data and then access it column-wise.

— We found the bottleneck to be the selection of specific columns within the matrix. This requires MATLAB, to copy this data, which is a slow operation in general. This cost could possibly be offset if we did not have to rely on `for` loops and instead vectorized the code. One straightforward way to do this is to express the inner loops as matrix multiplications. The drawback here is that we have to generate the sparse matrices $I_{2^{i-1}} \otimes J_{N/2^i}$, which are also expensive to generate and store.

In the following, we give a simple comparison of our algorithm with two other approaches: explicitly finding $(PH_N)x$, and explicitly finding $P(H_N x)$. We see that our implementation is faster than first finding the Walsh-Hadamard transform of x and then selecting k elements. However, it is not as fast as directly computing $(PH_N)x$, most likely because operations expressible in terms of matrix multiplication are more efficient than their iterative counterparts in MATLAB.

```
>> m = 2^13;
>> A = 2 * rand(m) - 1;
>> k = 5;
>> P = eye(m); Omega = randperm(m); P = P(Omega(1:k), :); P = sparse(P);
>> tic; H = hadamard(m); H(Omega(1:k), :) * A; toc
Elapsed time is 4.645202 seconds.
>> tic; P * fwht1d(A); toc
Elapsed time is 34.332823 seconds.
>> tic; fwht1d(A, Omega(1:k)); toc
Elapsed time is 9.572422 seconds.
```

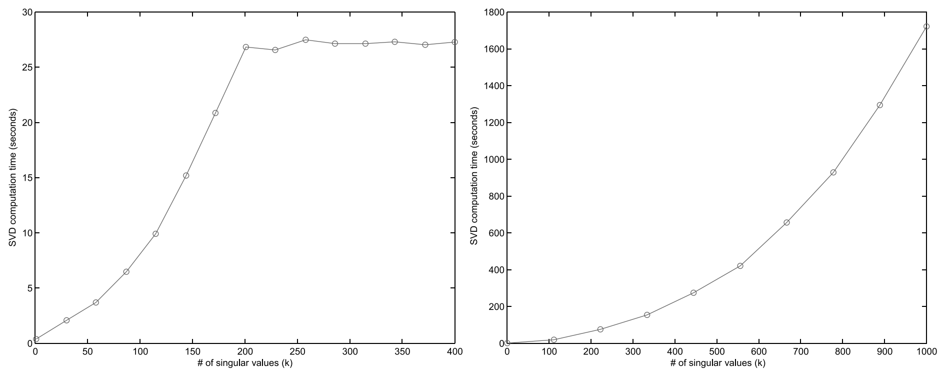
B. COMPUTATION TIME FOR THE TRUNCATED SVD

In Section 3.5, we mentioned that we only computed the truncated SVD in MATLAB, (which it calls the "economy SVD" option) for the given parameter k. The $O(mn \min\{m, n\})$ runtime for SVD we stated earlier is therefore for the case of $k = \min\{m, n\}$. For smaller k, we expect the runtime to be $O(mnk)$. We checked whether this is empirically true in MATLAB, using the PROPACK method; recall that this method is quite fast on both sparse and dense matrices, unlike `svd` and `svds`.

For the Faces dataset, we used a range of k values from 1 to 400. The results are shown in Figure 14. We see that after around $k = 200$ singular values, the runtime stabilizes and does not increase too much. One can verify that $k = 200$ captures 80% of the total latent structure in A, in the sense that $\sum_{i=1}^{200} \sigma_i^2 \approx 0.8 \sum_{i=1}^{400} \sigma_i^2$. Notice that while the runtime scales linearly with k for the dense Faces dataset, we have a quadratic dependence on k for the sparse Reuters dataset.

C. ARE SVD ALGORITHMS FASTER FOR SPARSE MATRICES?

We stated earlier that some algorithms for computing the SVD have runtime $O(mnc)$, where c is the average number of nonzeros per row. This suggests that as c gets smaller,



(a) Faces data. (b) Reuters data.

Fig. 14. PROPACK SVD computation time as k varies.

Table VIII. SVD Times in Seconds on A for Dense and Sparse Storage

Storage	PROPACK	`svd`	`svds`
A_{dense}	11.11	13.04	N/A
A_{sparse}	39.30	N/A	128.24

the algorithm has smaller computational complexity. However, we are not aware of references stating such a result for QR/Arnoldi iteration and Lanczos bidiagonalization, which are the algorithms used in `svd`, `svds` and PROPACK. Indeed, our experimental results showed that in some cases, the SVD time with the sampling methods actually *increased* compared to standard SVD. In this section, we explore this issue in more detail.

For a given matrix A, we can either store it in sparse or dense format in MATLAB. Let us denote these two representations of A by A_{sparse} and A_{dense} respectively. We would expect that it only makes sense to store a matrix in sparse format if a significant portion of its entries are zero. In terms of space requirements, it is not difficult to show[8] that if roughly more than $\frac{2}{3}$rds of a matrix's entries are nonzero, then A_{sparse} will take more space than A_{dense}. In terms of computation time, sparse matrix operations are usually slower than their dense counterparts, meaning that a careful tradeoff needs to be made in order to decide which representation to use. For our problem, this is pertinent when deciding how to store the sparsified version of A, which we denote $\hat{A}$, obtained after applying either uniform or non-uniform sampling.

With the above in mind, consider the following experiment on the Faces dataset; recall that this is a dense dataset. First we compute the SVD of this dataset using both a dense and sparse representation. Obviously since the matrix is completely dense, the sparse representation only adds overhead, but it will be a useful point of reference for the next part. We try all three basic SVD methods: MATLAB's `svd` and `svds`, and also PROPACK. The median times over 5 independent runs with $k = 100$ are shown in Table VIII. Unsurprisingly, we see that neither `svds` nor PROPACK is suited to dense matrices stored in sparse format.

Now let us do uniform sampling, storing the result in sparse and dense formats. The results are shown in Table IX. For the PROPACK method, it is surprising that the time on $\hat{A}$ stored in a dense format is significantly higher than the time on the original

[8]See `http://www.mathworks.com/access/helpdesk/help/techdoc/matlab_prog/brh72ex-25.html` for information on the sparse representation's memory requirements.

Table IX. SVD Times in Seconds on A Sampled with $p = 10^{-4}$, for Dense and Sparse Storage

Storage	PROPACK	`svd`	`svds`
$\hat{A}_{\text{dense}}$	32.57	10.03	N/A
$\hat{A}_{\text{sparse}}$	12.61	N/A	19.09

Table X. SVD Times in Seconds on A Sampled with $p = 10^{-5}$, for Dense and Sparse Storage

Storage	PROPACK	`svd`	`svds`
$\hat{A}_{\text{dense}}$	1.93	7.18	N/A
$\hat{A}_{\text{sparse}}$	0.59	N/A	6.84

matrix A, and is only marginally better than when we store A in a sparse format. We do not know what explains this unexpected behavior. Since the difference between the two runs is the sparsity of A, this is the most likely cause of the increase in runtime. However, it is difficult to determine how exactly the sparsity of A affects the runtime of the PROPACK method. One of its key operations involves finding Av and $A^T u$ for some vectors u, v, which are faster when A is sparse. But the major step in PROPACK, the one that distinguishes it from other approaches, is partial reorthogonalization, and this takes much linger when we do sampling. The reason for this is unclear.

`svd` and `svds`, on the other hand, respond well to the sparsified matrix A. This might be because they involve matrix-vector products as the core element of each iteration; we cannot verify this intuition though, because the source of these functions is not available to us. It is disheartening that PROPACK on the original matrix manages to be faster still.

When we reduce the sampling probability to 10^{-5}, the results are markedly different. Checking the profiler output in MATLAB, we see than when p is this small the number of iterations is reduced by more than a factor of 15. Since each iteration involves a relatively expensive partial reorthogonalization, this goes some way to explaining why the runtime is so much faster here. But we can only appeal to intuition to explain why the number of iterations reduces: namely, the spectral structure in the matrix is much simpler, and so we are able to discover the correct singular vectors faster. The results are given in Table X.

ACKNOWLEDGMENTS

The authors thank Michael Mahoney and Christos Boutsidis for insightful comments that helped improve the article.

REFERENCES

ACHLIOPTAS, D. AND MCSHERRY, F. 2001. Fast computation of low rank matrix approximations. In *Proceedings of the 33rd Annual ACM Symposium on Theory of Computing (STOC'01)*. ACM, New York, NY, 611–618.

ACHLIOPTAS, D. AND MCSHERRY, F. 2007. Fast computation of low rank matrix approximations. *J. ACM 54*, 2, 9.

AILON, N. AND LIBERTY, E. 2008. Fast dimension reduction using Rademacher series on dual BCH codes. In *Proceedings of the 19th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA'08)*. Society for Industrial and Applied Mathematics, Philadelphia, PA, 1–9.

ANDERSON, E., BAI, Z., BISCHOF, C., DEMMEL, J., DONGARRA, J., CROZ, J. D., GREENBAUM, A., HAMMARLING, S., MCKENNEY, A., OSTROUCHOV, S., AND SORENSEN, D. 1992. *LAPACK User's Guide*. SIAM, Philadelphia, PA.

ARTIN, E. 1942. *Galois Theory*. Notre Dame Mathematical Lectures, vol. 2. University of Notre Dame Press.

AT&T LABORATORIES CAMBRIDGE. 2002. The database of Faces. http://www.cl.cam.ac.uk/research/dtg/attarchive/facedatabase.html. (Accessed 5/08).

BERRY, M., MEZHER, D., BERNARD, P., AND SAMEH, A. 2005. *Handbook of Parallel Computing and Statistics*. Chapman & Hall/CRC, 117–164.

CHAN, T. F. 1982. An improved algorithm for computing the singular value decomposition. *ACM Trans. Math. Softw. 8*, 1, 72–83.

DESHPANDE, A. AND VEMPALA, S. 2006. Adaptive sampling and fast low-rank matrix approximation. In *APPROX-RANDOM*, J. Díaz, K. Jansen, J. D. P. Rolim, and U. Zwick Eds. Lecture Notes in Computer Science, vol. 4110, Springer, 292–303.

DRINEAS, P., DRINEA, E., AND HUGGINS, P. S. 2001. An experimental evaluation of a monte-carlo algorithm for singular value decomposition. In *Proceedings of the Panhellenic Conference on Informatics*. 279–296.

DRINEAS, P., KANNAN, R., AND MAHONEY, M. W. 2006a. Fast monte carlo algorithms for matrices I: Approximating matrix multiplication. *SIAM J. Comput. 36*, 1, 132–157.

DRINEAS, P., KANNAN, R., AND MAHONEY, M. W. 2006b. Fast Monte Carlo algorithms for matrices II: Computing a low-rank approximation to a matrix. *SIAM J. Comput. 36*, 1, 158–183.

DRINEAS, P., MAHONEY, M. W., AND MUTHUKRISHNAN, S. 2006c. Subspace sampling and relative-error matrix approximation: column-row-based methods. In *Proceedings of the 14th Annual European Symposium on Algorithm (ESA'06)*. Springer-Verlag, 304–314.

DRINEAS, P., MAHONEY, M. W., MUTHUKRISHNAN, S., AND SARLÓS, T. 2007. Faster least squares approximation. CoRR abs/0710.1435.

DUFF, I., GRIMES, R., AND LEWIS, J. 1998. Harwell-Boeing collection. http://math.nist.gov/MatrixMarket/collections/hb.html.

ECKART, C. AND YOUNG, G. 1936. The approximation of one matrix by another of lower rank. *Psychometrika 1*, 3, 211–218.

FRADKIN, D. AND MADIGAN, D. 2003. Experiments with random projections for machine learning. In *Proceedings of the 9th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD'03)*. ACM, New York, NY, 517–522.

FRIEZE, A., KANNAN, R., AND VEMPALA, S. 1998. Fast Monte-Carlo algorithms for finding low-rank approximations. In *Proceedings of the 39th Annual Symposium on Foundations of Computer Science (FOCS'98)*. IEEE Computer Society, Los Alamitos, CA, 370.

FURNAS, G. W., DEERWESTER, S., DUMAIS, S. T., LANDAUER, T. K., HARSHMAN, R. A., STREETER, L. A., AND LOCHBAUM, K. E. 1988. Information retrieval using a singular value decomposition model of latent semantic structure. In *Proceedings of the 11th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR'88)*. ACM, New York, NY, 465–480.

GOLUB, G. H. AND VAN LOAN, C. F. 1996. *Matrix Computations*, 3rd Ed. Johns Hopkins University Press, Baltimore, MD.

GORRELL, G. 2006. Generalized Hebbian algorithm for incremental singular value decomposition in natural language processing. In *Proceedings of the Conference of the European Chapter of the Association for Computer Linguistics (EACL)*.

HAR-PELED, S. 2006. Low rank matrix approximation in linear time. http://valis.cs.uiuc.edu/ sariel/papers/05/lrank.

HORN, R. AND JOHNSON, C. 1991. *Topics in Matrix Analysis*. Cambridge University Press.

JOLLIFFE, I. 1986. *Principal Component Analysis*. Springer, New York, NY.

LARSEN, R. M. 2005. PROPACK. http://sun.stanford.edu/ rmunk/PROPACK/.

LEWIS, D. D. 2004. Reuters-21578 text categorization test collection. http://www.daviddlewis.com/resources/testcollections/reuters21578/.

LIBERTY, E., WOOLFE, F., MARTINSSON, P., ROKHLIN, V., AND TYGERT, M. 2007. Randomized algorithms for the low-rank approximation of matrices. *Proc. Nat. Acad. Sci. 104*, 51, 20167.

MIRSKY, L. 1960. Symmetric gauge functions and unitarily invariant norms. *Quarterly J. Math. Oxford, Series 2*, 11, 50–59.

NGUYEN, N. H., DO, T. T., AND TRAN, T. D. 2009. A fast and efficient algorithm for low-rank approximation of a matrix. In *Proceedings of the 41st Annual ACM Symposium on Theory of Computing (STOC'09)*. ACM, New York, NY, 215–224.

PAPADIMITRIOU, C. H., TAMAKI, H., RAGHAVAN, P., AND VEMPALA, S. 1998a. Latent semantic indexing: a probabilistic analysis. In *Proceedings of the 17th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS'98)*. ACM, New York, NY, 159–168.

PAPADIMITRIOU, C. H., TAMAKI, H., RAGHAVAN, P., AND VEMPALA, S. 1998b. Latent semantic indexing: a probabilistic analysis. In *Proceedings of the 17th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS'98)*. ACM, New York, NY, 159–168.

PATEREK, A. 2007. Improving regularized singular value decomposition for collaborative filtering. In *Proceedings of KDD Cup and Workshop*.

ROKHLIN, V., SZLAM, A., AND TYGERT, M. 2009. A randomized algorithm for principal component analysis. *SIAM J. Matrix Anal. Appl. 31*, 3, 1100–1124.

SARLOS, T. 2006. Improved approximation algorithms for large matrices via random projections. In *Proceedings of the 47th Annual IEEE Symposium on Foundations of Computer Science (FOCS'06)*. IEEE Computer Society, Los Alamitos, CA, 143–152.

SUN, J., XIE, Y., ZHANG, H., AND FALOUTSOS, C. 2007. Less is more: Compact matrix decomposition for large sparse graphs. In *Proceedings of the SIAM International Conference on Data Mining (SDM)*.

SUN, J., XIE, Y., ZHANG, H., AND FALOUTSOS, C. 2008. Less is more: Sparse graph mining with compact matrix decomposition. *Statist. Anal. Data Mining 1*, 1, 6–22.

THOMAS, G. 2007. Fast Walsh-Hadamard transform. http://www.mathworks.com/matlabcentral/fileexchange/6879.

TJAHYADI, R., LIU, W., AND VENKATESH, S. 2004. Automatic parameter selection for eigenfaces. In *Proceedings of the 6th International Conference on Optimization: Techniques and Applications*.

TURK, M. AND PENTLAND, A. 1991. Eigenfaces for recognition. *J. Cogni. Neurosci. 3*, 1, 71–86.

VEMPALA, S. S. 2004. *The Random Projection Method*. DIMACS Series in Discrete Mathematics and Theoretical Computer Science, vol. 65, American Mathematical Society, Providence, RI.

YARLAGADDA, R. K. AND HERSHEY, J. E. 1997. *Hadamard Matrix Analysis and Synthesis: With Applications to Communications and Signal/Image Processing*. Kluwer Academic Publishers.

ZHANG, Z., ZHA, H., AND SIMON, H. 2001. Low-rank approximations with sparse factors I: Basic algorithms and error analysis. *SIAM J. Matrix Anal. Appl. 23,* 3, 706–727.

Received June 2010; accepted July 2010