Information Retrieval Through Various Approximate Matrix Decompositions

Kathryn Linehan, klinehan@math.umd.edu
Advisor: Dr. Dianne O'Leary, oleary@cs.umd.edu

Abstract

Information retrieval is extracting certain information from databases. In this paper we explore querying a document database using latent semantic indexing (LSI). We investigate and analyze the use of various matrix approximations to accomplish this task; specifically, we implement a nonnegative matrix factorization using the multiplicative update algorithm of Lee and Seung as found in [1] and the linear time Monte Carlo CUR decomposition of Drineas, Kannan, and Mahoney as found in [2]. We further implement variations on this CUR algorithm and use an implementation by G.W. Stewart of a deterministic CUR algorithm found in [3]. We test LSI performance on three common information retrieval data sets: MEDLINE, CISI and CRANFIELD.

We find that we can achieve LSI results that are almost as good as those produced by the SVD using much cheaper CUR decompositions. This is very important when a term-document matrix is too large to store or for which an SVD is too expensive to compute. In addition, we make improvements to the CUR decomposition of Drineas, Kannan, and Mahoney that give better matrix approximations without becoming significantly more expensive.

1 Introduction

The world is full of information, which would not be useful to us if we did not have a way of organizing and extracting it. In short, information retrieval is extracting certain information from databases. One example of information retrieval is the web search engine Google. The main issues associated with information retrieval are storage space, speed, and how "good" the results are, where "good" is a task specific measure. The objective of this project is to investigate querying a document database.

1.1 Approach

We implement the following approximate matrix decompositions: a nonnegative matrix factorization (NMF) computed by the multiplicative update algorithm of Lee and Seung as found in [1], and a linear time, Monte Carlo CUR decomposition by Drineas, Kannan, and Mahoney as found in [2].

We also investigate and implement an improvement to the CUR algorithm found in [2]. The improvement is a Compact Matrix Decomposition (CMD) by Sun, Xie, Zhang, and Faloutsos presented in [4]. In addition, we investigate "subspace sampling" for C and R, a method proposed by Drineas, Mahoney, and Muthukrishnan in [5], sampling without replacement, and a different method of computing U which we term the "optimal U". We use an implementation of a deterministic CUR algorithm by G. W. Stewart found in [3], which is based on a rank revealing QR decomposition.

We compare storage (number of nonzero elements for sparse matrices and number of matrix elements for dense matrices), runtime (seconds), and relative error (Frobenius norm) results for the above matrix approximations. We test the performance of the above matrix approximations in latent semantic indexing (LSI), investigate query time, and compare these results to those achieved using the SVD. All implementations are done in MATLAB.

The remainder of the paper is organized as follows. In section 1.2 we give information on related work to the topics covered in this paper. We present document retrieval background information in Section 2. In Section 3 we discuss the NMF problem and above algorithm, validate our implementation, and provide results on sparse initialization. Section 4 contains a description of all CUR algorithms that we implement, validation for all implementations, and a comparison of all implementations. In particular, Section 4.1.4 details sampling without replacement and why we feel it is a better strategy for CUR decompositions. In Section 5 and Appendices A and B we present LSI results using all of the above matrix approximations. We conclude in Section 6 and present ideas for future work in Section 7.

1.2 Related Work

There is much work on matrix approximations that is related to this paper; we highlight a limited number of results. When a matrix has very large dimensions, computing an SVD can be very expensive. In [6] Drineas, Kannan, and Mahoney present two algorithms for computing a Monte Carlo approx-

imate SVD, one of which provides a portion of the theoretical basis for the linear time CUR algorithm of Drineas, Kannan, and Mahoney in [2]. Also, in [7] Holmes, Gray, and Isbell present a Monte Carlo approximate SVD algorithm that uses sampling based on what they term "a cosine tree". These algorithms aim to give an approximate SVD that is not nearly as expensive to compute as the exact SVD.

We also mention that there are various other algorithms for computing the NMF of a matrix. Also presented in [1] are alternating least squares and gradient descent algorithms. In [8], Kim, Sra, and Dhillon present "Newton-type" methods for finding a NMF that they report to perform better than the multiplicative update algorithm of Lee and Seung.

We can approximate a matrix in other ways than using an SVD, NMF, or CUR. In [9] Boutsidis, Mahoney, and Drineas present an algorithm for choosing the most important k columns from a matrix which leads to a matrix approximation consisting of a projection of the original matrix onto the subspace spanned by the chosen columns. They accomplish this using a Monte Carlo step and then a deterministic step. In the CUR decomposition, we also aim to choose the most important columns of the original matrix.

There is also much work on applications of matrix approximations. In [10], Kolda and O'Leary analyze the use of the semidiscrete matrix decomposition in LSI. In [4], Sun, Xie, Zhang, and Faloutsos present results of the CMD (improvement on the CUR algorithm mentioned above) when used in anomaly detection. Kim, Sra, and Dhillon present an image processing application of nonnegative matrix factorization in [8] and Berry, Browne, Langville, Pauca, and Plemmons present a spectral analysis application of nonnegative matrix factorization in [1].

2 Background

2.1 Querying a Document Database

Given a document database, we want to be able to return documents that are relevant to given query terms. There are existing solutions to this information retrieval problem, such as literal term matching and latent semantic indexing (LSI). Background information and examples presented in Section 2 are taken from [10].

2.1.1 Problem Formulation

In real systems, such as Google, this problem is formulated in terms of matrices. An $m \times n$ term-document matrix, A, is created where entry a_{ij} represents the importance of term i in document j. Thus, each row in A represents a term and each column in A represents a document. Also, an $m \times 1$ query vector, q is created where q_i represents the importance of term i in the query. Different schemes for determining the entries in A and q are discussed in [10].

2.1.2 Literal Term Matching

In literal term matching, a relevance score is computed for each document as an inner product between q^T and the column of A that represents that document. The highest scoring documents are then returned.

The original term-document matrix is generally sparse; thus, unfortunately, literal term matching may not return relevant documents that contain synonyms of query terms, but not the actual query terms. We present an example of this below.

Example: Literal Term Matching

	Document				
Term	1	2	3	4	**Query**
Mark	15	0	0	0	1
Twain	15	0	20	0	1
Samuel	0	10	5	0	0
Clemens	0	20	10	0	0
Purple	0	0	0	20	0
Fairy	0	0	0	15	0
Score	30	0	20	0	

As seen in the above example, we have queried for the terms "Mark Twain". We notice that document 2 does not contain the terms "Mark Twain", but does contain the terms "Samuel Clemens" (who is the same person as Mark Twain) and is therefore relevant to the query. However, it has a relevance score of zero and thus is not returned as relevant. We would

like to have a document querying system in which this problem is solved.

2.1.3 Latent Semantic Indexing

In latent semantic indexing (LSI), an approximation to the term-document matrix is used to compute document relevance scores. Using a matrix approximation can introduce nonzero entries, possibly revealing relationships between synonyms, and thus relevant documents that may not have the exact query terms may be returned.

A commonly used matrix approximation in LSI is a rank-k singular value decomposition (rank-k SVD). Below, we present an example of LSI using a rank-2 approximation to the term-document matrix given in the literal term matching example.

Example: Latent Semantic Indexing

	Document				
Term	1	2	3	4	**Query**
Mark	3.7	3.5	5.5	0	1
Twain	11.0	10.3	16.1	0	1
Samuel	4.1	3.9	6.1	0	0
Clemens	8.3	7.8	12.2	0	0
Purple	0	0	0	20	0
Fairy	0	0	0	15	0
Score	14.7	13.8	21.6	0	

We see that in this example, by using LSI, document 2 now has a score of 13.8 and will therefore be returned as the third most relevant document, even though it did not contain the exact query terms "Mark Twain". This is an improvement over the relevance results of the literal term matching example.

In [10] a rank-k semidiscrete decomposition (SDD) is used in LSI. The rank-k SDD of an $m \times n$ matrix A is $A_k = X_k D_k Y_k^T$, where X_k is $m \times k$, D_k is $k \times k$, and Y_k is $n \times k$. Each entry of X_k and Y_k is one of $\{-1, 0, 1\}$ and D_k is diagonal with positive entries. The SDD is discussed in detail in [10].

In [10], an implementation of the SDD written by Tamara G. Kolda and Dianne P. O'Leary is used to determine that LSI using the SDD performs as well as LSI using the SVD, but does so using less storage space and query time.

An objective of this project is to test the performance of other matrix approximations when used in LSI.

2.1.4 Performance Measurement

To determine how well a document retrieval system performs, we use two measures: precision and recall. We define the following variables:

$Retrieved$ = number of documents retrieved
$Relevant$ = total number of documents relevant to the query
$RetRel$ = number of documents retrieved that are relevant.

Precision is defined as

$$P(Retrieved) = \frac{RetRel}{Retrieved}$$

and recall is defined as

$$R(Retrieved) = \frac{RetRel}{Relevant}.$$

We see that precision is the percentage of retrieved documents that are relevant to the query and recall is the percentage of relevant documents that have been retrieved.

3 Approximate Nonnegative Matrix Factorization

In general, a term-document matrix is nonnegative; thus, it is an interesting problem to find an approximate nonnegative decomposition. In [1] this problem is formulated as finding a W and H such that

$$f(W, H) = \frac{1}{2}\|A - WH\|_F^2 \tag{1}$$

is minimized, where A is the $m \times n$ matrix that we wish to approximate, W is $m \times k$, H is $k \times n$ and A, W and H are all nonnegative. For this problem, k is a rank parameter; $rank(WH) \leq k$.

3.1 Multiplicative Update Algorithm

We implement the multiplicative update algorithm of Lee and Seung as found in [1] to compute a W and H (for a given A) that satisfy Eq. (1). This algorithm is a gradient descent method derived from an alternating iteration. W and H are randomly initialized as dense matrices to begin. At each iteration W and H are updated using the following formulas as found in [1] using MATLAB notation:

$$\begin{aligned} H &= H. * (W^T A)./(W^T W H + 10^{-9}) \\ W &= W. * (A H^T)./(W H H^T + 10^{-9}). \end{aligned}$$

Each iteration includes six matrix multiplications, causing slow performance. Slight modifications, such as grouping certain matrix multiplications, can speed up the runtime [1].

Convergence is not guaranteed for this algorithm; fortunately though, in practice it is very common. However, when the algorithm does converge we are not guaranteed that the limit point is even a local minimum. If the limit point lies in the interior of the feasible region, it could be a local minimum or a saddle point. If the limit point lies on the boundary of the feasible region it may or may not be a stationary point [1].

3.2 Validation

Suppose A is our given $m \times n$ matrix. To validate our NMF implementation, we plot $\|A - WH\|_F / \|A\|_F$ (the relative error of the NMF approximation in the Frobenius norm) as a function of k. We expect that as

$$k \to rank(A), \;\; \|A - WH\|_F / \|A\|_F \to 0.$$

We also present a plot of time to compute the NMF of A (in seconds), and a plot of the storage needed for the NMF of A (in number of matrix elements), both as functions of k. In addition, for comparison purposes, we plot the relative error of the rank-k SVD (in the Frobenius norm), the time to compute the rank-k SVD of A (in seconds), and the storage needed for

the rank-k SVD of A (in number of matrix elements) all as functions of k on the respective above mentioned plots.

In the presented validation results, we first use a 5×3 random dense test matrix that has numerical rank 3. Call this matrix A. Further, all entries in A are contained in the interval $[0, 1]$. We use this as a test matrix because the NMF can be used on any matrix. We chose a test matrix with small dimensions due to the time-consuming computations (and number of iterations) involved in producing a NMF of full rank with very low relative error; using a small matrix allows us to explicitly show that we can achieve a very small relative error when the rank of our approximation equals the rank of A (in a practical amount of time).

For larger matrices, we generally need too many iterations to get a full rank NMF with a minuscule relative error. Also, in applications, we generally only need to continue the NMF iteration until a small enough relative error is achieved; thus, we can use a maximum number of iterations for the NMF algorithm. For these reasons, we present validation results for a 500×200 sparse matrix of numerical rank 200 with entries in $[0, 1]$ (call this matrix B), in which we iterate a maximum of 100 iterations. In this case, as $k \to rank(B)$ we still expect that the relative error of the NMF will decrease; however, it may not reach zero when $k = rank(B)$.

3.2.1 Validation Results

Our implementation of the multiplicative update algorithm of Lee and Seung as found in [1] provided the NMF results for matrix A and matrix B as seen in Figures 1 and 2 respectively. Due to the random initialization process, for each matrix we take the average relative error, runtime, and storage for the NMF over five runs for each value of k.

We first comment on Figure 1. For this test matrix, the dimensions are too small for a plot of the runtime or storage to be meaningful. Thus, we only provide a plot of the relative error.

In the relative error plot we get what we expect; as $k \to rank(A)$, $\|A - WH\|_F / \|A\|_F \to 0$. We also note that for this matrix the NMF achieves the optimal relative error produced by the SVD for each value of k (this behavior is generally not guaranteed for the NMF except for the value $k = rank(A)$).

We now comment on Figure 2, the results for B, the second test matrix. In the relative error plot we see the results that we expect: as $k \to rank(B)$

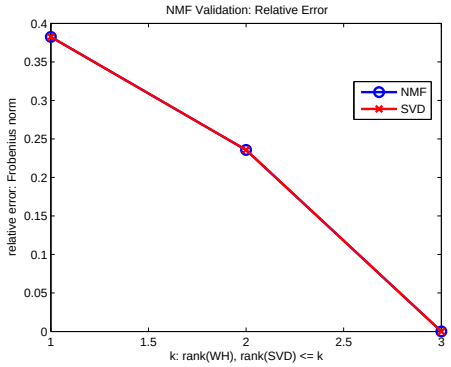

Figure 1: NMF validation results for matrix A (5×3)

the relative error of the NMF decreases; however, it does not reach zero when $k = rank(B)$. Also, we note that in this case, the NMF does not achieve the optimal relative error produced by the SVD for any value of k.

However, we also present a plot of relative error versus iteration number for matrix B (Figure 2: upper right). This plot gives information from one run of the NMF code using a rank parameter of 80. We plot the relative error for iterations 2 through 100 (iteration 1 has a much larger relative error as a result of the random initialization process). We see the trend we expect; relative error decreases at each successive iteration and approaches (but may never achieve) the rank 80 SVD relative error. This plot also illustrates another general trend of the NMF algorithm; we see a large improvement in relative error in the first iterations. After that, relative error improvements are very small.

The time plot is consistent with our expectations that as $k \rightarrow rank(B)$ the time to compute the NMF increases. For test matrix B, the NMF is more time consuming than the SVD for all values of k.

The storage plot is also consistent with expectations. Both the NMF and SVD store dense factors. Thus, we can measure the storage of the NMF as $k*(m+n)$ matrix entries and the storage of the SVD as $k*(m+n+1)$ matrix entries. We see this trend in the storage plot of Figure 2. Also, note that the NMF and the SVD require much more storage than the original sparse matrix.

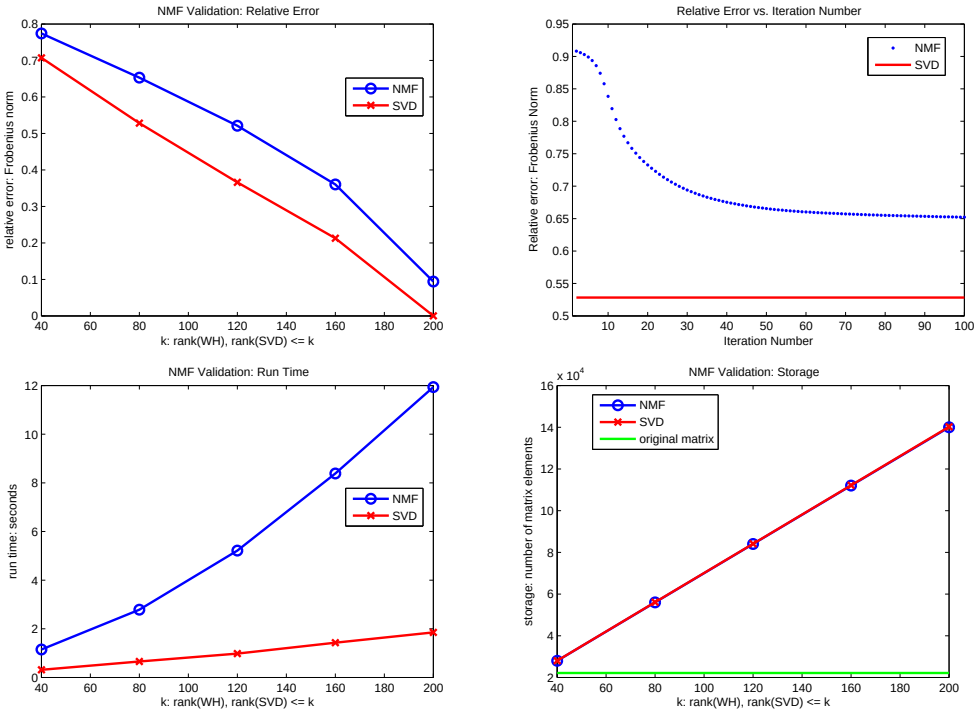

Figure 2: NMF validation results for matrix B (500×200)

3.3 Sparse Initialization

In addition to implementing the original multiplicative update algorithm of Lee and Seung as found in [1], we explored initializing W and H as sparse (instead of dense) matrices when the matrix to be factored was also sparse.

We again use B, the second validation test matrix. We randomly initialize W and H as sparse matrices with 25% nonzero elements each, and again iterate a maximum of 100 iterations per run. In Figure 3 we present plots of relative error, runtime, and storage (as functions of k) for this variation of the NMF and the rank-k SVD. We still measure storage as the number of matrix elements for the SVD, but now for the NMF storage measure we use the number of nonzero elements in W and H because W and H are stored as sparse matrices.

In Figure 3, in the relative error plot, we still have that as $k \to rank(B)$, $\|B - WH\|_F / \|B\|_F$ decreases. However, the sparse initialization penalizes the relative error of the NMF as compared to the dense initialization. We do not

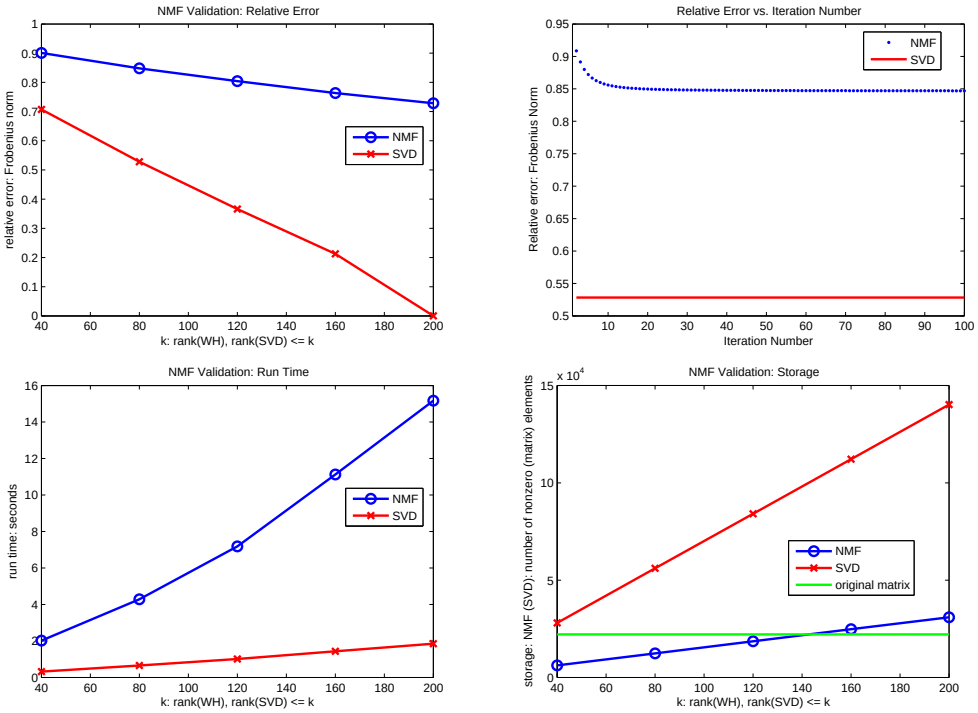

Figure 3: NMF validation results for matrix B (500×200) using sparse initialization

achieve near the relative error that we achieve when using dense initialization for any value of k.

Also in the upper right plot of Figure 3 we present a plot of relative error versus iteration number for matrix B (again for iterations 2 through 100). This plot gives information from one run of the NMF variation using a rank parameter of 80. We see the same trend in this plot as we do in the corresponding plot in 2; however, in this plot the decrease in relative error is much smaller and we do not see much decrease after iteration 10.

In the time plot of Figure 3, we see the same trend as in the time plot of Figure 2 except that the NMF variation takes a few more seconds to compute for each value of k. Again, the NMF variation is more time consuming than the SVD for all values of k.

In the storage plot of Figure 3 we see that we achieve about one third to one fourth of the storage of the SVD (and NMF with dense initialization) for all values of k. Also, the NMF variation uses less storage than the original

matrix for k up to about 140.

We see the above trends in Figure 3 because once we initialize W and H (as dense or sparse matrices) we can only update the nonzero entries to try and achieve a better approximation. Thus, with fewer possible entries to update, the NMF with sparse initialization does not produce as good of an approximation as NMF with dense initialization, and produces an approximation that takes far less storage than NMF with dense initialization.

In using NMF with sparse initialization, we achieve a large reduction in storage at the cost of relative error; thus, in the remainder of this paper we use the original NMF multiplicative update algorithm with dense initialization.

4 CUR Decomposition

The CUR decomposition is an approximation to an $m \times n$ matrix A: $A \approx CUR$, where C is $m \times c$, U is $c \times r$ and R is $r \times n$. The general idea behind the decomposition is as follows:

- C holds c sampled (and possibly rescaled) columns of A
- R holds r sampled (and possibly rescaled) rows of A
- U is computed using C and R.

Another property common to term-document matrices is sparsity. The CUR decomposition preserves sparsity in the factors C and R, whereas other decompositions, such as the SVD, do not preserve sparsity at all. This allows for not only less storage, but also a better physical interpretation of the decomposition. Each column from the product $CUR \approx A$, can be written as a linear combination of the columns of C; thus, the basis vectors are (rescaled) document columns from the original matrix.

4.1 CUR Algorithms

We implement six different Monte Carlo CUR algorithms and use a seventh deterministic CUR implementation by G. W. Stewart as found in [3]. There are two main decisions to make in a CUR algorithm: 1) how to choose the columns (rows) of A that form C (R); and 2) how to compute U.

The six Monte Carlo CUR algorithms that we implement can be described by their sampling techniques and computations of U, and for those algorithms

that use sampling without replacement, we also give a scaling descriptor. The Monte Carlo algorithms sample the columns and rows of A; the chosen columns and rows are rescaled and put into C and R respectively.

A list of the CUR implementations follow; we identify the implementations in the validation results by the abbreviations included in this list. Relevant references for the sampling scheme, computation of U, or both are given after each list item.

1. CN,L: Column (row) norm sampling with replacement, linear U (original CUR algorithm of [2])
2. CN,O: Column (row) norm sampling with replacement, optimal U [2]
3. S,L: "Subspace sampling "with replacement, linear U [5, 2]
4. S,O: "Subspace sampling" with replacement, optimal U [5]
5. w/o R, L, w/o Sc: Column (row) norm sampling without replacement, linear U, no scaling [2]
6. w/o R, L, Sc: Column (row) norm sampling without replacement, linear U, scaling [2]
7. D: Stewart's deterministic CUR [3]

In Section 4.1.1 we describe the original CUR algorithm of [2] that we began our research with. In Sections 4.1.2 through 4.1.5 we describe the remaining sampling techniques, computations of U, and scaling techniques (for those algorithms that use sampling without replacement) that are listed above. In Section 4.2 we validate each CUR implementation, and in Section 4.3 we compare all implementations.

In the remainder of this paper we make use of the pseudoinverse of a matrix. We denote the pseudoinverse of matrix M as M^+.

4.1.1 Original CUR Algorithm

The original CUR algorithm that we implement is a linear time CUR algorithm by Drineas, Kannan, and Mahoney as found in [2]. This algorithm uses

sampling with replacement according to column (row) norm probabilities, as seen below:

$$\begin{aligned} Prob(col\ j) &= \|A(:,j)\|_F^2/\|A\|_F^2 \\ Prob(row\ i) &= \|A(i,:)\|_F^2/\|A\|_F^2. \end{aligned}$$

Also, in this algorithm U is computed to approximately solve

$$\min_{\hat{U}} \|A - C\hat{U}\|_F, \tag{2}$$

where $\hat{U} = UR$, and $rank(U) \leq k$, where k is a rank parameter. We refer to U when computed as in [2] as a "linear U". Using the linear U, we are guaranteed that the rank of the CUR approximation is at most k.

We use the notation of [2] to describe the linear U. The linear U is computed using the SVD of C^TC and Ψ, an $r \times c$ matrix that holds sampled and rescaled rows of C. Specifically, let the SVD of C^TC be $\sum_{t=1}^{c} \sigma_t^2(C) y^t y^{t^T}$. Then,

$$U = \left(\sum_{t=1}^{k} \frac{1}{\sigma_t^2(C)} y^t y^{t^T} \right) * \Psi^T.$$

The solution to Eq. (2) is generally of the form $\hat{U} = (C^TC)^{-1}C^TA$. We see that the formula for the linear U is an approximation to this: 1) $\sum_{t=1}^{k} \frac{1}{\sigma_t^2(C)} y^t y^{t^T}$ is a rank-k approximation to $(C^TC)^+$; 2) Ψ^T is an approximation to C^T; and 3) R is an approximation to A and although it is not built into the formula for U, it appears as the last factor in $\hat{U}$. Thus, we see that for our approximation we get $C\hat{U} = CUR$. These approximations allow this algorithm to run in linear time (assuming $c \ll n$ and $r \ll m$) [2].

We implement an improvement to the original CUR algorithm of [2] as found in [4]. The improvement is a Compact Matrix Decomposition (CMD) by Sun, Xie, Zhang, and Faloutsos. The CMD uses the same sampling scheme and computation of U as the CUR algorithm of [2]. The CMD improvement is the following: remove repeated columns in C and repeated rows in R, then rescale the columns of C and rows of R appropriately, and then compute U using the new C and R. This improvement decreases storage space and runtime while achieving the same relative error as the original CUR algorithm [4].

We implement the CMD improvement for all CUR implementations that use sampling with replacement.

4.1.2 "Subspace Sampling"

We implement CUR algorithms that use sampling with replacement according to "subspace sampling" probabilities as found in [5]. Let the rank-k SVD of A be $A_k = U_k \Sigma_k V_k^T$ and the economy size SVD of C be $C = U_C \Sigma_C V_C^T$. The "subspace sampling" probabilities are given below:

$$\begin{aligned} Prob(col\ j) &= \|V_k(j,:)\|_F^2/k \\ Prob(row\ i) &= \|U_C(i,:)\|_F^2/c. \end{aligned}$$

The following discussion of "subspace sampling" is adapted from [5] (we also use the notation of [5]). The "subspace sampling" probabilities give us subspace information about A without giving us "size-of-A" information as found in the singular values of A [5]. They are inspired by the problem of finding a matrix X such that $\|A - CX\|_F$ is minimized, where C consists of sampled and rescaled columns of A. We can write a solution of this problem as $X = C^+ A$. We can also write the rank-k SVD of A as $U_k U_k^T A$.

Thus, we want to find a C such that the span of the columns of C is approximately the same subspace as the span of the columns of U_k. Let $M^{(i)}$ denote the i-th column of the matrix M, and let $rank(A) = \rho$. We have

$$A^{(i)} = U_k \Sigma_k \, (V_k^T)^{(i)} + U_{\rho-k} \Sigma_{\rho-k} \, (V_{\rho-k}^T)^{(i)}.$$

Therefore, "$\|(V_k^T)^{(i)}\|_F^2$ measures "how much" of the i-th column of A lies in the span of U_k, independent of the magnitude of the singular values associated with those directions" [5].

We can apply similar reasoning in a discussion of the row probabilities involved in "subspace sampling".

4.1.3 Optimal U

We also implement CUR algorithms that compute U as the "optimal U". We define the optimal U as the U that solves

$$\min_U \frac{1}{2} \|A - CUR\|_F^2 \tag{3}$$

given C and R. The least squares solution of the minimization problem in Eq. (3) is given by

$$U = (C^T C)^+ C^T A R^T (R R^T)^+.$$

The derivation of this solution follows.

Let $F(U) = \frac{1}{2}\|A - CUR\|_F^2$. Then,

$$\nabla F = C^T AR^T - C^T CURR^T.$$

Setting $\nabla F = 0$ we can solve for U from the equation $C^T CURR^T = C^T AR^T$.

Let $V_C \Sigma_C V_C^T$ be the SVD of $C^T C$. We now have

$$\begin{aligned} C^T AR^T &= C^T CURR^T \\ &= V_C \Sigma_C V_C^T URR^T \\ &= V_C \Sigma_C Y, \end{aligned}$$

where $Y = V_C^T URR^T$. The least squares solution of $C^T AR^T = V_C \Sigma_C Y$ is $Y = \Sigma_C^+ V_C^T C^T AR^T$.

Let $U_R \Sigma_R U_R^T$ be the SVD of RR^T. Now we have the following:

$$\begin{aligned} V_C Y &= URR^T \\ &= UU_R \Sigma_R U_R^T \\ &= X \Sigma_R U_R^T, \end{aligned}$$

where $X = UU_R$. The least squares solution of $V_C Y = X \Sigma_R U_R^T$ is $X = V_C Y U_R \Sigma_R^+$. This leads us to our final solution for U,

$$\begin{aligned} U &= XU_R^T \\ &= V_C Y U_R \Sigma_R^+ U_R^T \\ &= V_C \Sigma_C^+ V_C^T C^T AR^T U_R \Sigma_R^+ U_R^T \\ &= (C^T C)^+ C^T AR^T (RR^T)^+. \end{aligned}$$

The interested reader can find a minimal (Frobenius) norm solution to the minimization problem in Eq. (3) with an additional rank constraint on U in [11]. In addition, in [11] a proof is given that includes our proof as a special case.

4.1.4 Sampling Without Replacement

We implement two CUR algorithms that use sampling without replacement according to column (row) norm probabilities and the linear U from [2] (with

slight modifications). In this section we use the notation of [2]. The distinction between the two algorithms is the scaling involved in C, R, and Ψ (one of the matrices that is used in the computation of U).

First, we describe the modification of the linear U from [2] for our first CUR implementation that uses sampling without replacement. In [2], U is computed as

$$U = \left(\sum_{t=1}^{k} \frac{1}{\sigma_t^2(C)} y^t y^{t^T} \right) * \Psi^T,$$

where Ψ holds sampled and rescaled rows of C. We can write $C = AS_C D_C$, where S_C is an $n \times c$ matrix such that "$(S_C)_{ij} = 1$ if the ith column of A is chosen in the jth independent random trial and $(S_C)_{ij} = 0$ otherwise" [2] and D_C is a diagonal $c \times c$ matrix that holds the rescaling factors for the columns of C. Similarly, we can write $R = D_R S_R A$. Ψ is formally defined as $\Psi = D_R S_R C$ [2].

We propose to invert the scaling factor D_R for the matrix Ψ; in our first CUR implementation that uses sampling without replacement, we define $\Psi = D_R^{-1} S_R C$ and then use the same linear U formula from [2] with the new Ψ. We motivate this modification with the following demonstration:

Let A be a full rank $m \times n$ ($m \geq n$) matrix. Let $C = AD_C$ (ie. we have sampled without replacement n columns from A) and $R = D_R A$ (ie. we have sampled without replacement m rows from A). Note that both C and R are full rank as well. Now, take $\Psi = MC$ where M is a diagonal scaling matrix on Ψ. We calculate CUR as follows:

$$\begin{aligned} CUR &= (AD_C)((C^T C)^{-1} \Psi^T)(D_R A) \\ &= (AD_C)((D_C A^T A D_C)^{-1} D_C A^T M)(D_R A) \\ &= A(A^T A)^{-1} A^T M D_R A. \end{aligned}$$

We hope that after sampling all rows and columns of A, we get a CUR approximation with relative error zero. At this point it is clear that if we take $M = D_R^{-1}$ instead of $M = D_R$ we will get $CUR = A$, which is the desired result. This ends the demonstration.

For our second CUR implementation that uses sampling without replacement, we do not rescale C, R, or Ψ. Thus, we have $C = A_S C$, $R = S_R A$, and $\Psi = S_R C$.

Next, we explain why sampling without replacement is a better strategy than sampling with replacement. In general, sampling without replacement

brings in more information from the original matrix than sampling with replacement. For example, if we sample 20 columns from a matrix without replacement then we are guaranteed to have 20 unique columns; however, if we sample with replacement, we may only end up with 15 unique columns. It is reasonable to conjecture that if we use more information from the original matrix, we should get a better approximation (in terms of relative error).

We present the following argument to support the above conjecture. Let A be an $m \times n$ ($m \geq n$) matrix of rank n. Let $C = AD_C$ and let $R = D_R A_1$, where A_1 is the $k \times n$ matrix that consists of the first k rows of A. Also, let A_2 be the $(m-k) \times n$ matrix that consists of the last $m-k$ rows of A. Again, we use our updated formula for Ψ, which in this context is $\Psi = D_R^{-1} A_1 D_C$. We get the following CUR computation:

$$\begin{aligned} CUR &= AD_C((D_C A^T A D_C)^{-1} D_C A_1^T D_R^{-1}) D_R A_1 \\ &= A(A^T A)^{-1} A_1^T A_1 \\ &= A(A^T A)^{-1}(A^T A - A_2^T A_2) \\ &= A(I_n - (A^T A)^{-1}(A_2^T A_2)) \\ &= A - A(A^T A)^{-1}(A_2^T A_2). \end{aligned}$$

Thus, we see that the term $A(A^T A)^{-1}(A_2^T A_2)$ will contribute to the error in the CUR approximation. As we let k (the number of rows in A_1) go to m we notice that the relative error of the CUR approximation will go to 0 because the matrix A_2 goes to the zero matrix. We can conclude that once we have sampled m rows and n columns of the matrix, we indeed achieve a CUR approximation with relative error zero.

However, if we use the original CUR algorithm of [2] (in particular sample with replacement) and sample m rows and n columns of A we do not necessarily achieve this result. In fact, as long as we sample at least one column twice (which is very likely), C will have rank less than n causing the CUR approximation to have rank less than n. In this case, we have a lower rank approximation to our matrix A which will not have relative error zero.

By using sampling without replacement we avoid using redundant information from A and are thus able to produce a better CUR approximation. This trend is evident in practice; we provide results in the CUR comparison section, Section 4.3.

4.1.5 Stewart's Deterministic CUR Algorithm

We also use an implementation by G. W. Stewart of a deterministic CUR algorithm as found in [3]. A rank revealing QR decomposition of A (A^T) is used to determine which columns (rows) of A should be contained in C (R). U is computed by finding the minimizer of $\|A - CUR\|_F^2$ [3].

4.2 Validation

The validation of our CUR implementations is similar to the validation of the NMF implementation. Suppose A is our given $m \times n$ matrix. To validate our CUR implementations, we plot $\|A - CUR\|_F / \|A\|_F$ (the relative error of the CUR approximation in the Frobenius norm) as a function of k . We expect that as

$$k \to rank(A), \ \ \|A - CUR\|_F / \|A\|_F \to 0.$$

For comparison purposes, we also plot the relative error of the rank-k SVD in the Frobenius norm as a function of k.

Again we use test matrices A (5×3) and B (500×200) (the same test matrices from the NMF validation). We use test matrix A for validation of the CUR implementations that use sampling with replacement. We use A for these validations due to its small size. These CUR algorithms require us to sample many times over the columns and rows of A to show the desired validation results. Using a small matrix allows us to explicitly show this in a practical amount of time.

We also use a third test matrix for the CUR implementations. Call this matrix C. C is a 50×30 random sparse test matrix with entries in $[0, 1]$ that has numerical rank 30. We use this matrix to validate the CMD improvement of [4] that we implement for all CUR algorithms that use sampling with replacement.

We use test matrix B to validate the CUR implementations that use sampling without replacement. In this case, we can use a larger matrix because we only have to sample the matrix B according to its dimensions (ie. we sample no more than 500 rows and 200 columns from B).

In all validation plots we use a legend corresponding to the following:

1. Sampling choice. CN : column (row) norm sampling with replacement [2], S: "subspace sampling" with replacement [5], w/o R: column (row) norm sampling without replacement

2. U choice. L: linear U [2], O: optimal U

3. Scaling (only for use with sampling without replacement). Sc: rescaling of columns and rows used, w/o Sc: rescaling of columns and rows not used.

Stewart's deterministic CUR of [3] is given in the legend by D.

4.2.1 CMD Validation Results

We first present the validation results for the CMD improvement of [4]. The CMD improvement was intended for the original CUR algorithm of [2]. Thus, we present results for the CMD improvement on the CUR algorithm from [2]. Our implementation of the original CUR algorithm from [2] with and without the CMD improvement of [4] produced the results for test matrix C in Table 1 below. We use $k = 15$, $c = r = 30$ (ie. we sample 30 columns and 30 rows of A and use a CUR approximation that has rank at most 15). We compute the average runtime (seconds), storage (number of nonzero elements for sparse factors and number of matrix elements for dense factors), and relative error (Frobenius norm) over 10 runs due to the Monte Carlo approach of the CUR algorithm in [2] and the CMD algorithm in [4].

Algorithm	[2]	[2] with CMD
Runtime	0.008060	0.007153
Storage	880.5	550.5
Relative Error	0.820035	0.820035

Table 1: CMD results

We see that the CMD achieves improvements in runtime and storage, while achieving the same relative error as the original CUR algorithm found in [2]; these results were expected [4]. The CMD decreases runtime because computations involving C with fewer columns are faster. In all CUR algorithms that use sampling with replacement, we have implemented this CMD algorithm of [4].

4.2.2 Sampling with Replacement Validation Results

We now turn to validating all of our CUR implementations that use sampling with replacement. Figure 4 gives validation results for these implementations when using test matrix A (5×3). Due to the Monte Carlo approach of the CUR algorithms we compute the average relative error over five runs for each value of k.

In Figure 4 we present three plots; however, note that on the x-axis we do not have the variable k, instead we have the variables c and r where $c = r$. In each plot, we still use $k = 1, 2, 3$ (as we did when using test matrix A in the NMF validation). In the upper left plot of Figure 4 we set $c = r = 5k$; in the upper right plot of Figure 4 we set $c = r = 500k$; and in the lower left plot of Figure 4 we set $c = r = 100000k$.

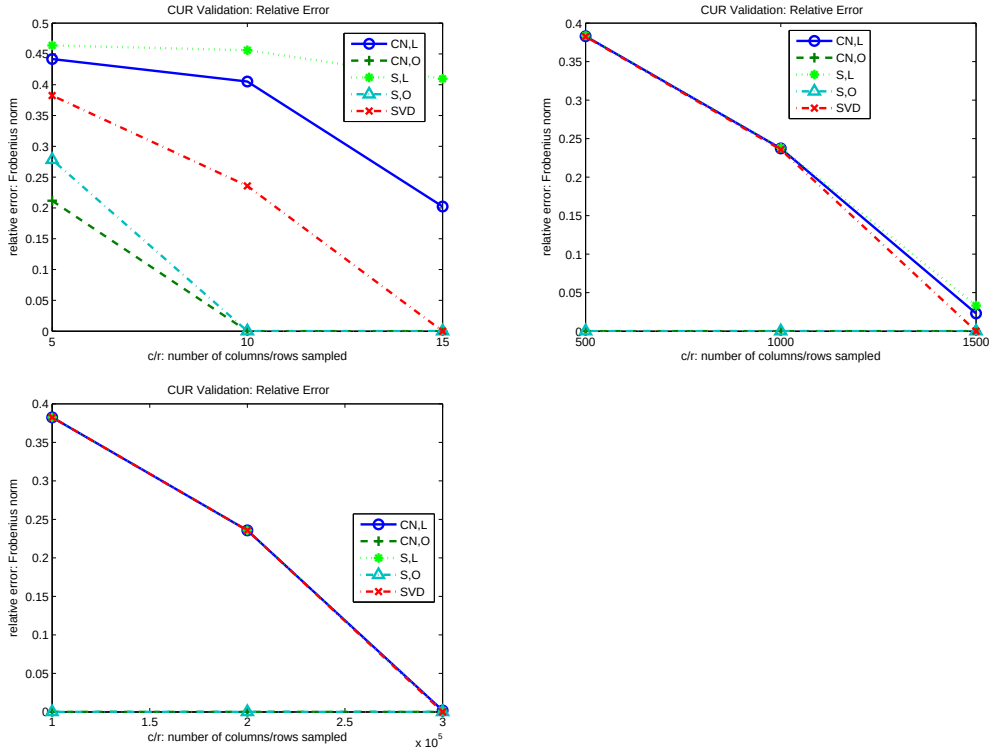

Figure 4: CUR: sampling with replacement validation results for matrix A (5×3)

First, we note that the CUR implementations that use the optimal U are

validated in the upper left plot of Figure 4. Once $c = r = 10$ it is evident that we have at least one (rescaled) copy of each column of A in C and each row of A in R at which time the relative error of the approximation is equal to zero; this is the expected behavior. We also note that these CUR implementations achieve a better relative error than the rank-k SVD for $k = 1, 2$. The optimal U computation does not involve a rank parameter; thus, the rank of a CUR implementation involving the optimal U can be larger than k (if c and r are both larger than k).

Now, we look at the CUR implementations that use the linear U of [2]. We see that even though the size of A is 5×3 we need to sample on the order of 10^5 columns and rows of A to get our desired validation results. We see this behavior because as we continue to sample, the scaling factors applied to C, R, and Ψ are updated, causing our approximations to have lower relative error. Once we reach a point when we most likely have a (rescaled) copy of each column of A in C and each row of A in R, we are still using the linear U (with the original scaling factor on Ψ as found in [2]) which is an approximation as described in section 4.1.1. Thus, we may not see a relative error of zero until we have sampled enough columns and rows of A.

4.2.3 Sampling without Replacement Validation Results

We now validate our CUR implementations that use sampling without replacement. Figure 5 gives validation results for these implementations when using test matrix B. Due to the Monte Carlo approach of the CUR algorithms we compute the average relative error over five runs for each value of k.

For each of these CUR implementations we use $r = 3k$ and $c = k$. This guarantees that when $k = 200$ (ie. $k = rank(B)$) that $r = 500$ (in the sampling without replacement case, we limit r to be less than or equal to the number of rows in the original matrix and similarly we limit c as well) and $c = 200$ (ie. we have sampled all of the rows and columns of B). In Figure 5 we see the desired validation result that also corresponds with the theory provided in section 4.1.4.

We also note that for this validation test, the CUR implementation that uses scaling and the CUR implementation that does not use scaling achieve about the same relative error for each value of k. While the two algorithms generally give very similar error results, it is not always the case that they are as similar as in Figure 5.

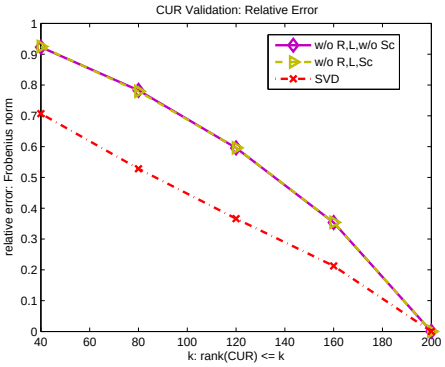

Figure 5: CUR: sampling without replacement validation results for matrix B (500×200)

4.3 CUR Comparison

Now, we compare all of the CUR implementations mentioned above in terms of relative error, runtime, and storage. Relative error is given in the Frobenius norm, and runtime is given in seconds. Storage is computed by summing the following: the number of matrix elements in the dense factors of the decomposition and the number of nonzero elements in the sparse factors of the decomposition. We use the test matrix B (500×200) and compute the average relative error, runtime, and storage for each value of k over five runs.

We present all comparisons in Figure 6. The left column of Figure 6 gives results for test matrix B when using $r = c = k$ and the right column gives results when using $r = c = 2k$.

We first comment on the left column of plots in Figure 6. We give the main trends for $k \leq 120$ as we generally use these decompositions to give a lower rank approximation to a matrix.

- CUR implementations that use sampling without replacement give better relative error results while using about the same amount of computation time and slightly more storage than the original CUR algorithm of [2].

- CUR implementations that use the optimal U achieve better relative error results than the original CUR algorithm of [2] while remaining inexpensive (when compared to the corresponding implementations that

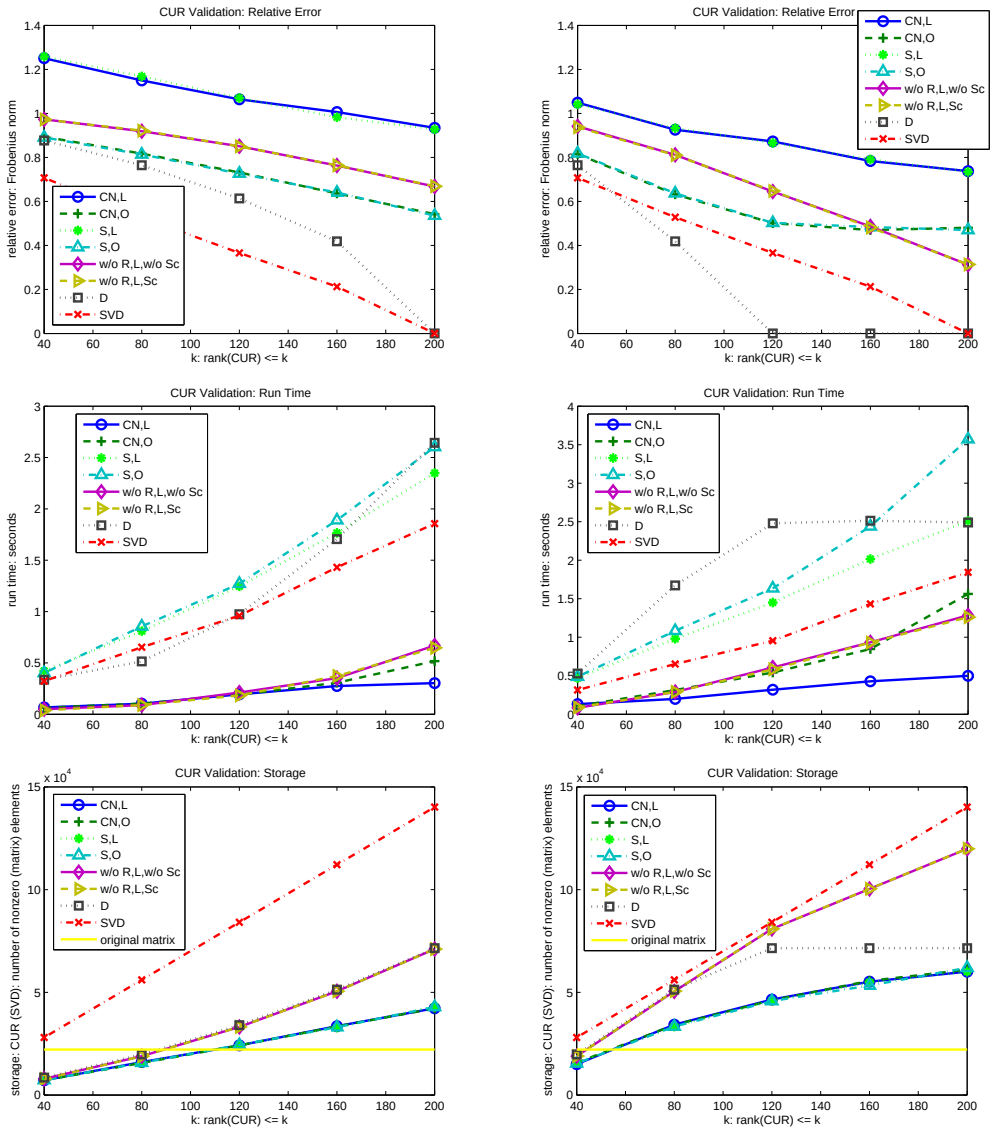

Figure 6: CUR comparison using test matrix B (500×200). Left: $r = c = k$. Right: $r = c = 2k$.

use the linear U of [2]).

- "Subspace sampling" ([5]) is expensive and appears to not give much error improvement over column norm sampling ([2]).

- Stewart's deterministic CUR implementation of [3] gives the best error results of the CUR implementations while not using much more storage or computation time than the original CUR algorithm of [2].

Now we comment on the right column of plots in Figure 6 and specifically compare with the plots in the left column. We note that even though $r = c = 2k$ in these plots, for each value of k such that $c = 2k > 200$, we set $c = 200$ (the number of columns in B is 200).

- As we sample more rows and columns than k we achieve better relative error results (this is consistent with results presented in [5]) at the price of more computation time and storage.

- CUR implementations that use sampling without replacement still have very fast computation times; however, they also use up to almost twice as much storage as the original CUR algorithm presented in [2].

- Stewart's deterministic CUR algorithm of [3] is very efficient. (Stewart's algorithm does not use rank constraint on U; thus, the rank of a CUR approximation using this algorithm depends on c and r. This explains why it achieves better relative error than the rank-k SVD.) It takes the same amount of computation time and storage once it obtains a full rank decomposition, which for test matrix B (of numerical rank 200) occurs for $k \geq 120$ ($r \geq 240$ and $c = 200$) [3].

In Section 5 and Appendices A and B we provide further comparisons of relative error, runtime and storage for CUR implementations.

5 LSI Results

In this section we test performance of the NMF and all of the above CUR decompositions in LSI. Similar to the example presented in Section 2.1.3, we explain how document scores are computed using the NMF and CUR decompositions.

Let A be an $m \times n$ term-document matrix, q an $m \times 1$ query vector, and s an $1 \times n$ score vector such that s_i is the relevance score for document i for the given query. For a given query, we compute s using a series of vector-matrix products as follows:

- (rank-k) SVD: $A \approx U_k \Sigma_k V_k^T$, $s = ((q^T U_k)\Sigma_k)V_k^T$
- NMF: $A \approx WH$, $s = (q^T W)H$
- CUR: $A \approx CUR$, $s = ((q^T C)U)R$.

We test the performance of the NMF and CUR decompositions in LSI using average precision and recall, where the average is taken over all queries in the data set, and compare to the performance of the SVD. We also compute average query time for each matrix approximation in the LSI process, where again the average is taken over all queries in the data set. We define query time to be the time it takes to score the documents and then sort them based on their scores.

We use three common information retrieval data sets: MEDLINE, CISI, and CRANFIELD. Each data set contains a term-document matrix, a term list, queries, and a list of relevant documents for each query. These LSI data sets can be found at www.cs.utk.edu/~lsi/ and are listed under MED, CISI, and CRAN. Each term-document matrix is large and sparse.

In all LSI results the legend abbreviates each matrix approximation. To be clear, each CUR algorithm that uses sampling with replacement is listed according to its sampling probabilities and U computation. The CUR implementations that use sampling without replacement are listed using 'w/o R' first and then yes or no according to whether the implementation uses scaling or not. Stewart's deterministic algorithm of [3] is listed under 'CUR: GWS' and using the original matrix (no approximation) is listed under 'LTM' (for literal term matching).

We also present results on the relative error (Frobenius norm), storage (computed as in Section 4.3) and runtime (seconds) of each matrix approximation (to the term-document matrix) that is used. We compute each matrix approximation only once and then use it to score each document for each query as described above. Results produced by using averages over five runs would produce similar results.

We present LSI results in Figure 7 for the MEDLINE data using rank 100 matrix approximations. The term-document matrix for this data set is 5831×1033. Also, for all CUR decompositions we use $r = c = 100$.

	Rel. Error (F-norm)	Storage (nz)	Runtime (sec)
SVD	0.8203	686500	22.5664
NMF	0.8409	686400	23.0210
CUR: cn,lin	1.4151	17242	0.1741
CUR: cn,opt	0.9724	16358	0.2808
CUR: sub,lin	1.2093	16175	48.7651
CUR: sub,opt	0.9615	16108	49.0830
CUR: w/oR,no	0.9931	17932	0.3466
CUR: w/oR,yes	0.9957	17220	0.2734
CUR: GWS	0.9437	25020	2.2857
LTM:	–	52003	–

Table 2: Matrix approximation results for MEDLINE term-document matrix (5831×1033)

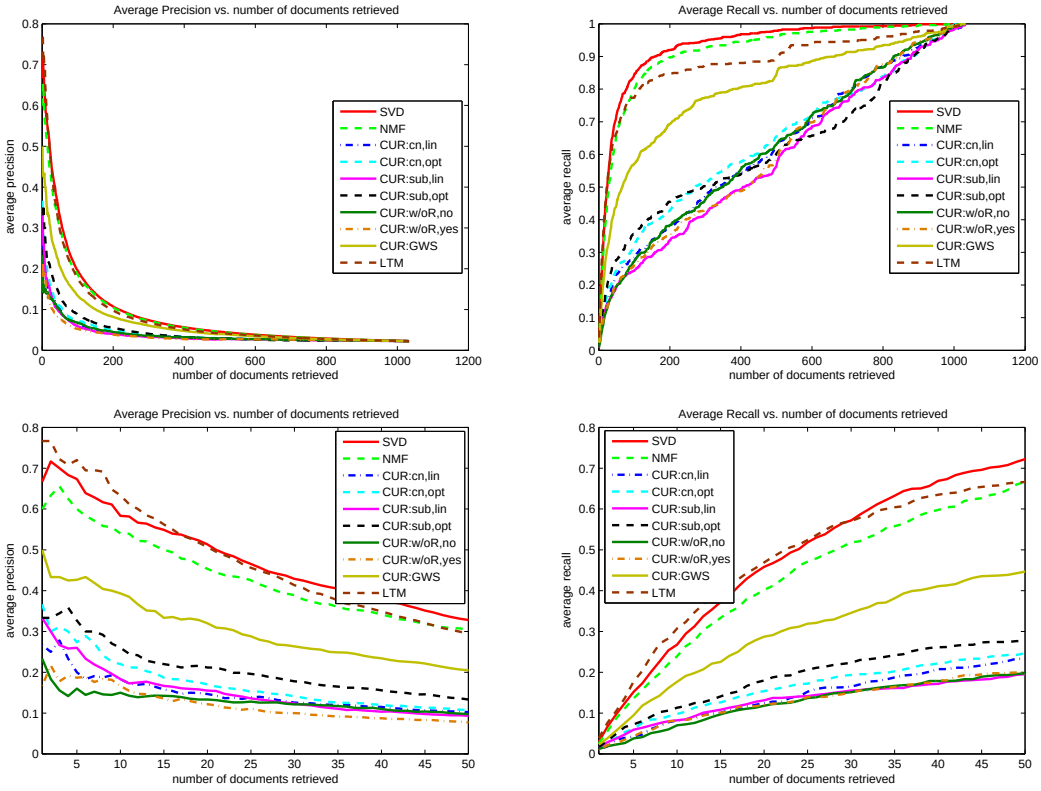

Figure 7: LSI Results for the MEDLINE data

We first note that results presented in Table 2 are consistent with results presented in Section 4.3. Also, average query time is less than 10^{-3} seconds for all matrix approximations (including the original matrix).

We now discuss the top two plots in Figure 7 in which we give results from returning 1 through 1033 documents. The SVD and NMF perform the best of the matrix approximations in terms of giving the best precision and recall (they also eventually outperform the original matrix (LTM)). We see that the Monte Carlo CUR implementations all give similar precision and recall results; although they do not perform as well as the SVD or NMF. Stewart's deterministic CUR algorithm of [3] performs well; it gives much better precision and recall than the Monte Carlo CUR algorithms. Also using the original matrix (LTM) gives better results than any CUR implementation.

In the bottom two plots of Figure 7 we give the same results but only from returning 1 through 50 documents; a user may only go through 20 to 30 documents in a search. Also, a user is most likely concerned with the precision measure: the percentage of documents returned that are relevant. For example, after returning 20 documents, we see that we have returned approximately:

- 10 relevant documents using SVD/NMF/original matrix
- 7 relevant documents using Stewart's deterministic CUR
- 4 relevant documents using Monte Carlo CUR decompositions.

Although the SVD and NMF give the best LSI results, they are also expensive matrix approximations in terms of storage and runtime. Stewart's deterministic CUR decomposition gives good results while remaining inexpensive. While the Monte Carlo CUR decompositions give results that are almost as good as those produced by Stewart's CUR; they are even more inexpensive (except those that use "subspace sampling").

Similar results from the CISI and CRANFIELD data sets are given in Appendices A and B. The analysis for each set of results is similar to that for the MEDLINE data set and is thus omitted.

6 Conclusions

We have made improvements on the original CUR algorithm of [2]: 1) using sampling without replacement (and changing the scaling factors used) and 2)

using the optimal U. The CUR implementations that used either improvement 1) or 2) showed substantial improvements in relative error over the original CUR algorithm of [2]. Furthermore, this can be achieved without substantial increases in storage or runtime.

We also have the following LSI conclusions. In real applications, we may not be able to store an entire term-document matrix or it may be too expensive to compute an SVD or NMF of a term-document matrix. However, we can achieve LSI results that are almost as good (as those produced by the SVD or NMF) using matrix approximations that are very cheap in terms of storage and computation time; namely, the Monte Carlo CUR decompositions that do not use "subspace sampling" and Stewart's deterministic CUR decomposition of [3].

In fact, Stewart's deterministic CUR gives LSI results that are very close to those of the SVD or NMF while only being slightly more expensive in terms of storage and runtime than the Monte Carlo CUR decompositions mentioned above. It is a happy medium between the two desired properties: giving good results and being cheap to compute.

7 Future Work

Future work may include continuing to explore the sparse initialization of the NMF in the multiplicative update algorithm of Lee and Seung. In particular, we would like to initialize W and H according to the sparsity pattern of the original matrix. We may also continue with the theoretical basis for why sampling without replacement is a better idea than sampling with replacement in CUR algorithms.

We may also look into implementing other matrix approximations such as the approximate SVD of Holmes, Gray, and Isbell found in [7] or the NMF algorithms of Kim, Sra, and Dhillon found in [8]. In addition, we may investigate these matrix approximations in LSI. Other applications for matrix approximations are also future possibilities.

8 Completed Project Goals

- Coded and validated NMF [1], CUR [2], and CUR variants [4, 5]

- Analyzed relative error, runtime, and storage of above matrix decompositions and Stewart's deterministic CUR of [3]
- Improved CUR algorithm of [2] (in a relative error sense) specifically using 1) sampling without replacement and changing scaling factors of [2] and 2) using the optimal U
- Analyzed use of above matrix decompositions in LSI

Deliverables: Code, final report

Appendices

A LSI Results: CISI Data

We have slightly modified the CISI data set; we have removed queries from this data set that have no relevant documents.

We present LSI results in Figure 8 for the CISI data using rank 100 matrix approximations. The term-document matrix for this data set is 5609×1460. Also, for all CUR decompositions we use $r = c = 100$. Average query time is less than 10^{-3} seconds for all matrix approximations (including the original matrix).

	Rel. Error (F-norm)	Storage (nz)	Runtime (sec)
SVD	0.8562	707000	26.2973
NMF	0.8809	706000	23.9639
CUR: cn,lin	1.3508	17552	0.2219
CUR: cn,opt	0.9835	17557	0.3317
CUR: sub,lin	1.2393	17309	52.5456
CUR: sub,opt	0.9732	17008	52.0564
CUR: w/oR,no	0.9951	19039	0.2247
CUR: w/oR,yes	0.9949	18687	0.2874
CUR: GWS	0.9607	30347	2.2091
LTM:	–	68240	–

Table 3: Matrix approximation results for CISI term-document matrix (5609×1460)

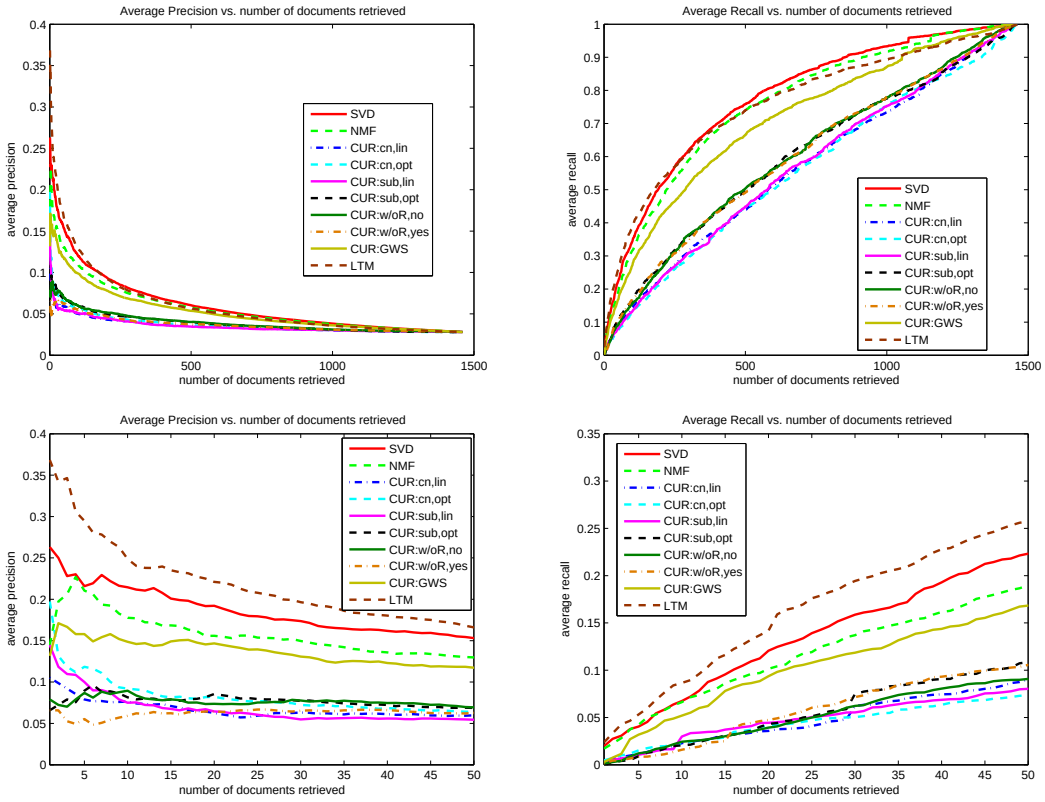

Figure 8: LSI Results for the CISI data

B LSI Results: CRAN Data

We present LSI results in Figure 9 for the CRAN data using rank 100 matrix approximations. The term-document matrix for this data set is 4612×1398. Also, for all CUR decompositions we use $r = c = 100$. Average query time is less than 10^{-3} seconds for all matrix approximations (including the original matrix).

	Rel. Error (F-norm)	Storage (nz)	Runtime (sec)
SVD	0.8130	601100	20.6018
NMF	0.8462	601000	20.7465
CUR: cn,lin	1.3233	21321	0.2440
CUR: cn,opt	0.9683	20438	0.3429
CUR: sub,lin	1.2014	20115	39.7348
CUR: sub,opt	0.9610	19817	39.9883
CUR: w/oR,no	0.9912	22250	0.2260
CUR: w/oR,yes	0.9888	22313	0.2347
CUR: GWS	0.9417	31743	2.0644
LTM:	–	83302	–

Table 4: Matrix approximation results for CRANFIELD term-document matrix (4612×1398)

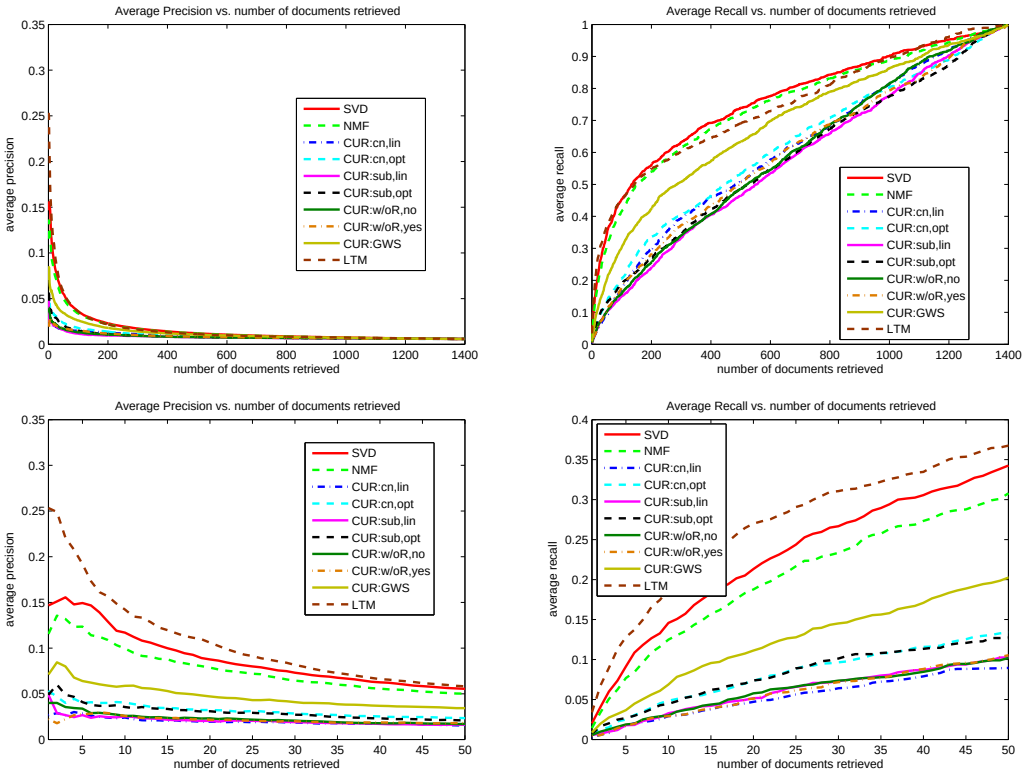

Figure 9: LSI Results for the CRANFIELD data

References

[1] Michael W. Berry, Murray Browne, Amy N. Langville, V. Paul Pauca, and Robert J. Plemmons. Algorithms and applications for approximate nonnegative matrix factorization. *Computational Statistics and Data Analysis*, 52(1):155–173, September 2007.

[2] Petros Drineas, Ravi Kannan, and Michael W. Mahoney. Fast Monte Carlo algorithms for matrices iii: Computing a compressed approximate matrix decomposition. *SIAM Journal on Computing*, 36(1):184–206, 2006.

[3] M.W. Berry, S.A. Pulatova, and G.W. Stewart. Computing sparse reduced-rank approximations to sparse matrices. Technical Report UMI-ACS TR-2004-34 CMSC TR-4591, University of Maryland, May 2004.

[4] Jimeng Sun, Yinglian Xie, Hui Zhang, and Christos Faloutsos. Less is more: Sparse graph mining with compact matrix decomposition. *Statistical Analysis and Data Mining*, 1(1):6–22, February 2008.

[5] Petros Drineas, Michael W. Mahoney, and S. Muthukrishnan. Relative-error CUR matrix decompositions. *SIAM Journal on Matrix Analysis and Applications*, 30(2):844–881, 2008.

[6] Petros Drineas, Ravi Kannan, and Michael W. Mahoney. Fast Monte Carlo algorithms for matrices ii: Computing a low-rank approximation to a matrix. *SIAM Journal on Computing*, 36(1):158–183, 2006.

[7] Michael P. Holmes, Alexander G. Gray, and Jr. Charles Lee Isbell. Quic-svd: Fast svd using cosine trees. In D. Koller, D. Schuurmans, Y. Bengio, and L. Bottou, editors, *Advances in Neural Information Processing Systems 21*, pages 673–680. 2009.

[8] Dongmin Kim, Suvrit Sra, and Inderjit S. Dhillon. Fast projection-based methods for the least squares nonnegative matrix approximation problem. *Statistical Analysis and Data Mining*, 1(1):38–51, February 2008.

[9] Christos Boutsidis, Michael W. Mahoney, and Petros Drineas. An improved approximation algorithm for the column subset selection problem. *CoRR*, abs/0812.4293, 2008.

[10] Tamara G. Kolda and Dianne P. O'Leary. A semidiscrete matrix decomposition for latent semantic indexing in information retrieval. *ACM Transactions on Information Systems*, 16(4):322–346, October 1998.

[11] Shmuel Friedland and Anatoli Torokhti. Generalized rank-constrained matrix approximations. *SIAM Journal on Matrix Analysis and Applications*, 29(2):656–659, March 2007.