Performance and Application of Matrix Decomposition and Inversion via Probabilistic Algorithms

Ethan Levine

April 7, 2017

Abstract

The advent and proliferation of "Big Data" has led to time efficiency issues of classical algorithms. One way of approaching this issue is by developing probabilistic algorithms that terminate more quickly than comparable deterministic ones. In this paper, we implemented probabilistic algorithms for computing the singular value decomposition and inverse of a matrix in R and C++. Further, we measured the speed and accuracy of these implementations against comparable, available deterministic algorithms. In order to keep all operations in memory, we developed "slice" matrix multiplication algorithm that progressively computed the product of two matrices. Finally, we apply these methods for fitting thin plate spline models for detecting land mines.

1 Introduction

A major focus of the analysis of algorithms is determining algorithmic efficiency. Specifically, one asks how long it takes for an algorithm to run, usually considering the average case. The length of time it takes for an algorithm to run is called its runtime, but it is often more important to understand its long term or asymptotic runtime behavior. First, to determine the simple runtime of an algorithm, one counts the number of scalar operations, such as addition, multiplication, and comparison, needed to perform the algorithm for a given input size. As an example, given two square matrices of dimension n^2, it takes $2n^3 - n^2$ scalar operations to perform classic matrix multiplication. Consider the following multiplication of two 3×3 matrices:

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix} \begin{pmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \\ b_{31} & b_{32} & b_{33} \end{pmatrix} = \begin{pmatrix} a_{11} * b_{11} + a_{12} * b_{21} + a_{13} * b_{31} & ... & ... \\ ... & ... & ... \\ ... & ... & ... \end{pmatrix}$$

In order to calculate each element of the product matrix, we must calculate three multiplications and two additions, which gives five scalar operations. Because there are nine elements of the product matrix, it takes 45 scalar operations to perform the complete matrix multiplication. For $n = 3$, the runtime function for matrix multiplication above agrees, as $2 * 3^3 - 3^2 = 45$.

To represent the asymptotic behavior of an algorithm, we use Big-O notation. We can consider two functions: $f(x)$ and $g(x)$. Then, $f(x) = O(g(x))$ as $x \to \infty$ if and only if there exist constants M and x_0 such that $|f(x)| \leq M|g(x)|$ for all $x \geq x_0$. This means that after a certain value of f, f is bounded by g. Consider again the runtime for matrix multiplication, which we can write as $f(x) = 2x^3 - x^2$. It is clear that f is bounded by $g(x) = 2x^3$ for at least all $x \geq 1$. Thus, if we consider $M = 2$, then we can say that $f(x) = O(g(x))$. That is, asymptotically, the runtime of matrix multiplication grows like x^3. Generally, if $f(x) = \sum_i c_i f_i(x)$, where the c_is are coefficients, then $O(f(x))$ will be the function $f_k(x)$ which grows fastest. If $f(x)$ is a polynomial, then this implies that $O(f(x))$ will be the highest degree term of $f(x)$.

An asymptotic growth rate on the scale of $O(n^3)$ poses significant issues to the efficiency of many algorithms. Multiplying two square matrices of dimension 100^2 takes about one million floating point (which are mostly the same as scalar) operations. For two matrices of size $1,000,000^2$, it takes about one quintillion or one billion billion operations. An Intel Core i7 processor, one of the most powerful, commercial processors, can perform about 110 billion floating point operations in a second (110 gigaflops). Thus, it would take such a processor about 10 million seconds to compute the product of two $1,000,000^2$ matrices, which is about 116 days.

"Big Data", which often considers data sets on the scales of those discussed or larger, can be defined as data big enough that it won't fit easily on a single machine (generally 1TB or larger). The advent and proliferation of "big data" has made using classic algorithms prohibitively expensive in regards to the time it takes for them to run.

This is especially important in the field of statistics and statistical programming. Matrix multiplication and other matrix operations, like matrix inversion and the singular value decomposition, which all have runtime complexity of $O(n^3)$ as well, are used extensively in statistical programming. For example, matrix inversion and multiplication are integral to performing ordinary least squares regression. Given a vector response y and matrix predictors X such that $y = X\beta + \epsilon$, where β is the vector of regression coefficients and ϵ is an error term, we find that $\hat{\beta} = (X^T X)^{-1} X^T y$ minimizes the the sum of the squared residuals between the regression line and the actual data. If our data set is exceedingly large, such as on the scale considered before, it can take a prohibitively long time to perform such an operation. Other algorithms are even more complicated and computationally expensive, exacerbating the issue with utilizing them on large data sets.

Thus, it is worthwhile to explore new algorithms that can perform such operations more efficiently. For example, Strassen's Algorithm and the Coppersmith-Winograd Algorithm can perform matrix multiplication in about $O(n^{2.81})$ and $O(n^{2.37})$ times respectively. However, there are drawbacks to using such algorithms, including the presence large coefficients hidden by Big-O notation that make these algorithms inefficient for small values of n and issues with regard to the numerical stability of these algorithms when implemented using limited precision floating point data. This paper will explore the use of probabilistic algorithms to derive efficiency gains in performing matrix operations.

At this point, all the algorithms discussed have been deterministic algorithms: ones that produce the same output for a given input every time they are run. A probabilistic algorithm is one that uses randomness in one or more of its steps. Thus, for a given input, the output of the algorithm will not always be the same. This paper will explore the utility of using probabilistic algorithms to perform the singular value decomposition and inversion of a matrix. We will consider the efficiency gains provided by using probabilistic algorithms, their stability regarding the accuracy of their outputs compared to the deterministic solutions, and applications to fitting thin-plate spline models.

2 Probabilistic Singular Value Decomposition

Given a real or complex $m \times n$ matrix A, there exists a factorization $A = U\Sigma V^*$, such that U is $m \times m$ and is unitary, Σ is an $m \times n$ diagonal matrix of the singular values of A, and V^* is $n \times n$ and unitary as well. This is referred to as the Singular Value Decomposition (SVD) of A. Typical uses of the SVD include computing pseudo-inverses of matrices in order to solve systems of linear equations, performing total least squares minimization, finding the range, rank, and nulls space of A, and constructing low-rank matrix approximations. Common ways of calculating the SVD of a matrix include (1) the classical two-step method of converting the matrix to a bidiagonal matrix using Householder reflections and applying a variant of the QR decomposition and (2) using an iterative low rank algorithmic approach.

The utility of being able to readily construct the SVD of a matrix is clear; however, for an $m \times n$ matrix, the runtime for constructing the SVD is $O(m^2n + mn^2 + n^3)$, which, for a square $n \times n$ matrix, would be $O(n^3)$, as mentioned before. Thus, for large matrices, performing singular value decompositions can be prohibitively expensive in terms of time.

An alternative to using the deterministic SVD calculation algorithm is to use a probabilistic algorithm, as presented in Halko, et al. (2010). If we have an $m \times n$ matrix A and a target number of singular vectors k, we can construct an approximate rank-k factorization $U\Sigma V^*$ of A. There are two stages to this algorithm. The first is to construct a matrix Q with a range that approximates the range of A. This means that $A \approx QQ^*A$ The second stage is to use this matrix Q to compute the SVD of A.

The algorithm is a follows:

Input: Matrix $A \in M_{n\times n}$, number of target singular vectors k, oversampling parameter l, and iteration parameter q

1. Stage 1
   - (a) Generate a $n \times (k + l)$ Gaussian matrix Ω
   - (b) Compute $Y_0 = A\Omega$
   - (c) Compute the QR factorization of $Y_0 = Q_0R_0$
   - (d) for $i = 1, 2, ..., q$
     - i. Compute $\tilde{Y}_i = A^*Q_{i-1}$ and its QR decomposition $\tilde{Y}_i = \tilde{Q}_i\tilde{R}_i$
     - ii. Compute $Y_i = A\tilde{Q}_i$ and its QR decomposition $Y_i = Q_iR_i$
   - (e) $Q = Q_q$
2. Stage 2
   - (a) Compute $B = QA$
   - (b) Construct the SVD of B: $B = \tilde{U}\Sigma V^*$
   - (c) $U = Q\tilde{U}$
   - (d) return (U, Σ, V^*)

We now have an approximate SVD of A, $U\Sigma V^*$. The step that makes this a probabilistic algorithm comes at the very beginning when we draw the random Gaussian matrix Ω. The iterative process in stage 1 creates matrix Q, which is an approximate basis of matrix A. At the same time, Q is created so that it has few columns; that is, so that it has low rank. Because Q has low rank, when we construct B in stage 2, it is efficient to compute its SVD using classical methods. Thus, it is the construction and utilization of the low rank basis matrix Q that makes this probabilistic algorithm potentially useful.

Before determining if an algorithm is more efficient than standard algorithms for matrices of large size in practice, we have to better understand the accuracy of the algorithm. That is, we would like to know about its average error, specifically $\mathbb{E}||A - QQ^*A||$. Halko, et al. (2010) provide the following average error bound for the probabilistic SVD algorithm:

$$\mathbb{E}||A - QQ^*A|| \leq (1 + \frac{4(k+l)^{1/2}}{l-1} \times n^{1/2})\sigma_{k+1}$$

All variables here are the same as defined previously and σ_{k+1} is the $k+1$ singular value of A. Additionally, the optimal error is given as:

$$||A - QQ^*A|| \approx \sigma_{k+1}$$

Thus, the expected error of the probabilistic algorithm lies within a polynomial of the optimal error. Further, by Gu and Eisenstat (1996), this error bound is actually sharper than that of comparable deterministic algorithms, like rank-revealing QR. Thus, not only does this probabilistic algorithm produce matrices Q that consistently approximate the basis of A, it permits less error than competitive deterministic algorithms, signaling that any error caused by the stochastic sampling will not be greater than error allowable within machine precision.

To determine the effectiveness of this algorithm in practice, we first needed to generate matrices on which to apply the algorithm. First, we generated lists of 5000 point coordinates with differing properties. One list was generated so that the points were equally spaced out in a grid. The other list was generated by drawing the x and y coordinates of the point from standard random uniform distributions. From these lists of coordinates, we generated covariance matrices with short, medium, and long range dependence structure by defining an exponential covariance function $exp(\frac{-d}{r})$ where d is the distance between the points and r is a chosen radius. An important feature of these different matrices is the differing rates at which their eigenvalues decay. The eigenvalues decay most quickly for the covariance matrix with long range dependence structure and least quickly for the covariance matrix with short range dependence structure.

We then applied the probabilistic SVD algorithm to these covariance matrices. We varied: the number of target singular vectors, so that $k \in \{5, 10, 15, 20, 100, 200, 5000\}$; the number of QR iterations, so that $q \in \{0, 1, 2, 3\}$; and the oversampling parameter, so that $l \in \{5, k\}$. We recorded the time taken for the algorithm to run and the error of the returned product of the algorithm, which was found by taking the Frobenius or point-wise norm of the given covariance matrix and the probabilistic low rank SVD of the matrix. As a basis of comparison, we also recorded the performance of a deterministic, iterative algorithm that is also used to approximate the SVD of a matrix.

The results make it clear that setting q to above 1 was almost never necessary as the error improvement wouldn't make up for the greater amount of time needed for the algorithm to run. Additionally, setting the over-sampling parameter l to the lower value of 5 was usually better than setting it to k as the higher setting didn't meaningfully improve the errors, while almost always increasing runtimes.

More important, though, was coming to understand how the probabilistic algorithm compared to the iterative one when the number of target singular vectors, k, increased. Consider the following graphs that show the time and error of the probabilistic algorithm, for select values of q and l, and the iterative algorithm for all values of k for the short, medium, and long covariance matrices:

In all three graphs, k increases from the top-left to the bottom-right, such that the runs with the most error and least time had the fewest number of target singular vectors and

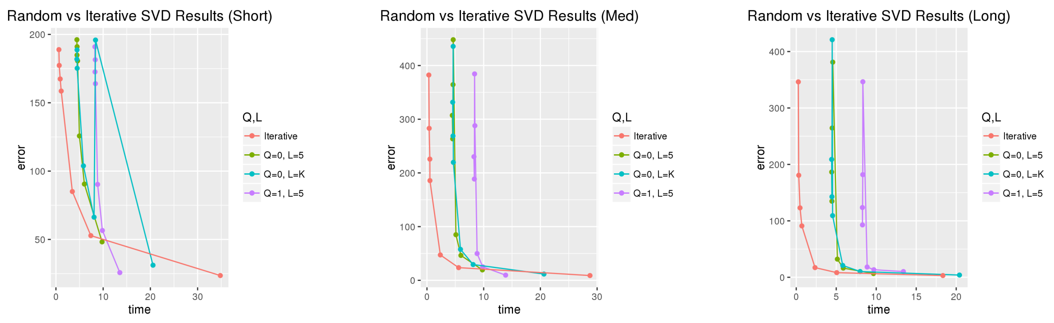


Figure 1: Probabilistic vs Iterative SVD Results

those with the least error and most time has the most number of target singular vectors. It is immediately clear that in the case of all three covariance matrices, when k is small, the iterative algorithm outperforms the probabilistic ones in regards to time and error for any choice of parameters. However, for small k, even though the time taken for the algorithms to run is small, the error produced by the algorithms can be sufficiently large so as to make the results unusable for application. In practice, a larger number of target singular vectors would be chosen so as to produce as accurate a result as possible, while improving or producing an equivalent run time.

For the short and medium covariance matrices, the probabilistic algorithm with set parameters of $q = 1$ and $l = 5$ outperforms all other algorithms when the number of singular target vectors is maximized, in that it produces a minimal error, while also not requiring an exorbitant amount of time to run the algorithm compared other options that take less time, but produce substantially more error. In the case of the long covariance matrix, the iterative algorithm almost always outperforms the probabilistic ones, though the probabilistic algorithm with $q = 0$ and $l = 5$ is competitive when k is maximized.

One important point to emphasize is that the preceding graphs are not all monotonically decreasing. The previous tests were performed multiple times and, on average, the results were monotonically decreasing; however, the aberrations that are present exemplify the variance observed in the run times. These aberrations can be caused by factors intrinsic to the algorithms, such as generated random matrices that are in some way pathological, or extrinsic, like using a shared machine to perform the run.

Our first implementation was developed in R. We later implemented this algorithm in C++ so as to try to improve its performance. The time versus error results for the R and C++ implementations are shown in the graphs below:

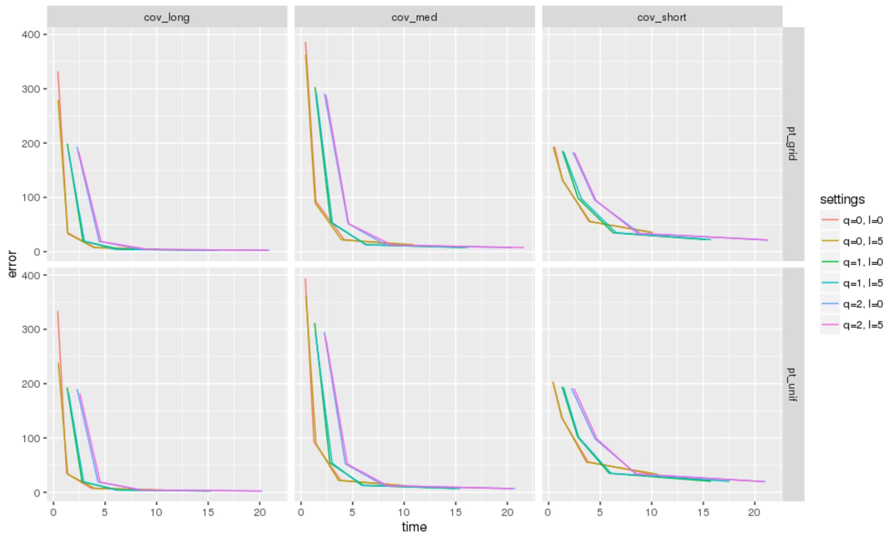


Figure 2: Time vs Error for R Implementation

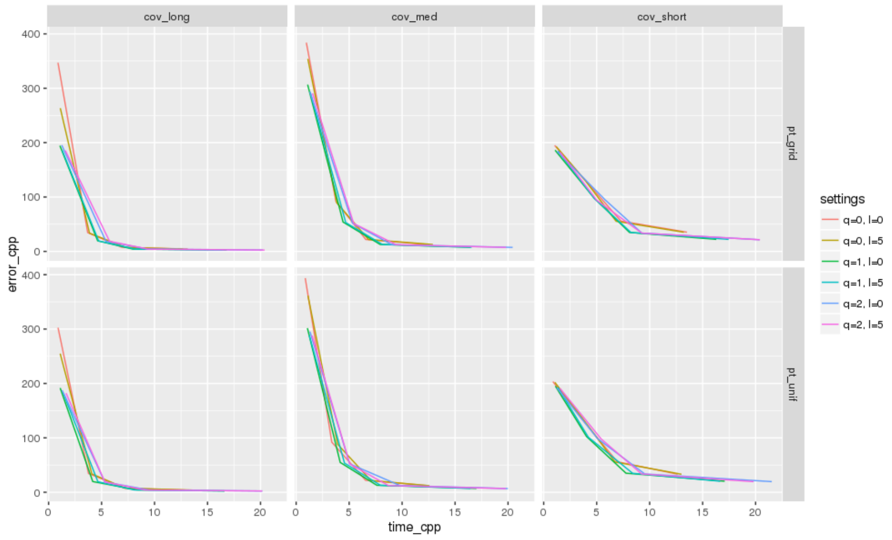


Figure 3: Time vs Error for C++ Implementation

As we can see by comparing these figures, the C++ implementation performed about the same as the R implementation regarding error, which was expected, as well as time, which was unexpected. Specifically and counterintuitively, the C++ implementation was about 0.85 times as fast as the R implementation, while having about the average amount of error. This is likely due to the rate limiting steps for both implementations depending on the low level linear algebra libraries lapack and blas used for the heavy lifting. The slowdown in the C++ implementation is likely due to the overhead of translating and copying R's matrix objects into a native C++ data structure. Thus, while useful in exercise, the C++ implementation did not yield any practical gains.

3 Probabilistic Matrix Inverse

Given a square matrix A, A^{-1} is said to be the inverse of A if $AA^{-1} = A^{-1}A = I$, where I is the identity matrix. Matrix inversions are found throughout statistical analysis with some examples being computing least square solutions to regression problems and determining posteriors for multivariate distributions in Bayesian analyses. There are a number of ways to compute the inverse of a square matrix. The classical way to do so is to use Gauss-Jordan elimination. Other common methods to find the inverse of a matrix are to use the LU decomposition, QR decomposition, and Cholesky decomposition.

Another means to finding the inverse of a matrix is to utilize the eigendecomposition of the matrix. Consider a square matrix A of dimension $N \times N$ that is full rank. We can find matrices Q and Λ, such that $A = Q\Lambda Q^{-1}$, such that the Q is $N \times N$ and its i^{th} column is the i^{th} eigenvector of A and that Λ is the diagonal matrix of the eigenvalues of A. One can determine the eigenvectors and eigenvalues of A by solving $Av = \lambda v$, where v is an eigenvector of A and λ is an eigenvalue of A. The eigendecomposition of a matrix has the useful property that $A^{-1} = Q\Lambda^{-1}Q^{-1}$. Thus given the eigendecomposition of a matrix, we need only find the inverse of its diagonal eigenvalue matrix to compute the inverse of the matrix in question. This too is simple as, $[\Lambda^{-1}]_{ii} = \frac{1}{\lambda_i}$. That is, we only need to take the reciprocal of each element of Λ to find its inverse and, thus, to find the inverse of A. Additionally, for a symmetric matrix A, Q will be orthogonal, so $Q^{-1} = Q^T$. This finally gives $A^{-1} = Q\Lambda^{-1}Q^T$.

Like the SVD, it is clear that finding the inverse of a matrix is useful in statistical computing. However, also like the SVD, computing the inverse of a matrix using classical algorithms, including using Gauss-Jordan elimination and finding the eigendecomposition, is inefficient as it is $O(n^3)$ for matrices of dimension $n \times n$. While there have been developments that have improved on the time complexity of matrix inversion, the remaining inefficiency of these algorithms (they are all less efficient than $O(n^2)$) provides the opportunity to develop a probabilistic algorithm for finding the inverse of a matrix, similar to how one was found to determine the SVD of a matrix.

The probabilistic matrix inverse algorithm is also given by Halko, et al. (2010) and is almost identical to the one given for the probabilistic SVD:

Input: Matrix $A \in M_{n\times n}$, number of target singular vectors k, oversampling parameter l, and iteration parameter q

1. Stage 1
   (a) Generate a $n \times (k + l)$ Gaussian matrix Ω
   (b) Compute $Y_0 = A\Omega$
   (c) Compute the QR factorization of $Y_0 = Q_0R_0$
   (d) for $i = 1, 2, ..., q$
      i. Compute $\tilde{Y_i} = A^*Q_{i-1}$ and its QR decomposition $\tilde{Y_i} = \tilde{Q}_i\tilde{R}_i$
      ii. Compute $Y_i = A\tilde{Q}_i$ and its QR decomposition $Y_i = Q_iR_i$
   (e) $Q = Q_q$
2. Stage 2
   (a) Compute $B = Q^TAQ$
   (b) Construct the eigendecomposition of B: $B = V\Lambda V^*$, where V is the matrix of $B's$ eigenvectors and Λ is the diagonal matrix of its eigenvalues

(c) Compute $U = QV$

(d) return (U, Λ) such that $A \approx U\Lambda U^T$

As with the probabilistic SVD algorithm, the probabilistic matrix inversion algorithm was implemented in R and C++ and was explored using various values of k, l, and q to find the inverse of the short, medium, and long range covariance matrices as inputs. The graphs below show the results of the R and C++ implementations:

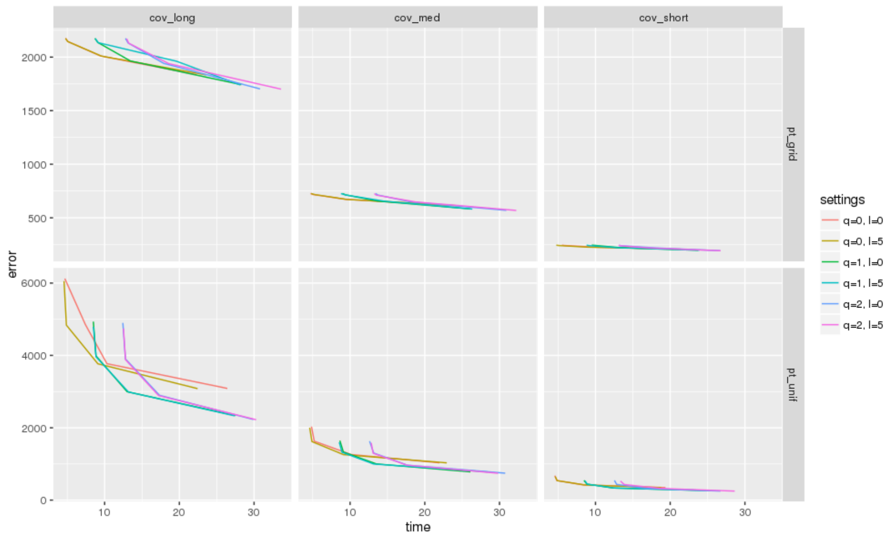


Figure 4: Time vs Error for R Implementation

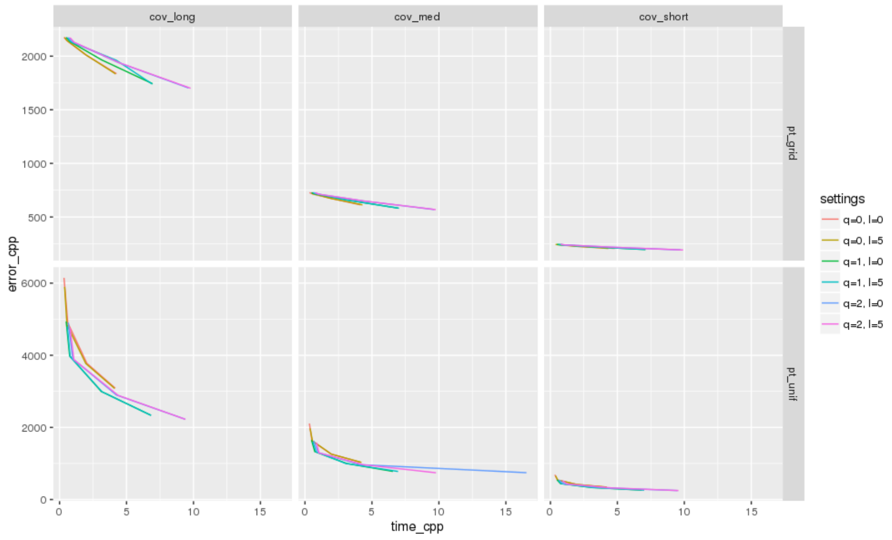


Figure 5: Time vs Error for C++ Implementation

While the graphs look similar, it is important to notice that the x-axis scales for the two sets of graphs are different. Specifically, the C++ implementation yielded a meaningful speed up in runtime that we did not see in the SVD example. On average, the C++ implementation of the matrix inversion algorithm was 8.73 times faster than the R implementation, with almost exactly the same error on average.

The probabilistic algorithm was also compared to deterministic matrix inverse algorithms, though, unlike the comparison made with the SVD algorithms, the latter algorithms were not iterative. The natural first deterministic algorithm to test was using the Cholesky Decomposition ($A = LL^*$ where L is is the lower triangular of A and L^* is its inverse). The runtime for this implementation was 2.538 seconds with negligible error, which means that it performed better, both time and error-wise, than the probabilistic algorithms. However, the downside of the Cholesky Decomposition is that it can only be made on Hermitian, positive definite matrices. This includes all correlation and covariance matrices (the latter of which are our test inputs), but excludes many other classes of matrices, including some matrices that will be considered later. Thus, while it exhibits the best performance, its specificity makes its usability limited compared to the probabilistic algorithm.

Other common algorithms used to compute inverse matrices are the Moore-Penrose Pseudo-Inverse, QR decomposition, and LU decomposition. For the relatively small covariance matrices considered, these inverses were only slightly slower than the probabilistic algorithm, but were much better performers in terms of error. However, as we consider matrices on the scale of 10000×10000 and larger, the error-performance between these methods become more similar, while the runtime differences between these deterministic algorithms and the probabilistic algorithm continue to increase in favor of the latter. The reason for this divergence is that the runtime of the deterministic algorithms are $O(n^3)$ (this is also true for the Cholesky decomposition, but its hidden coefficients are much smaller than the other algorithms'), while the runtime for the probabilistic algorithm is $O(n^2 log(k))$. Thus, the deterministic algorithms' runtimes will increase at a much faster rate than the probabilistic algorithm, even as the probabilistic algorithm becomes more accurate.

4 Slice Matrix Multiplication

An issue encountered while scaling up the previous algorithms was how to store and read in the actual data used in the algorithms. Generally, data is stored in two places in the computer: on the hard disk and in memory. Much more data can be held on the disk than in memory, usually on the scale of terabytes versus gigabytes, which is a difference of about three orders of magnitude. On the other hand, data can be accessed much more quickly from memory than from the disk. Data, as floating point values, can be sequentially accessed from memory about six times faster than from disk, while the difference is about 100,000 times if the data access is random (Jacobs, 2009). As our data sets can be exceedingly large, determining ways to efficiently use memory as opposed to disk is increasingly important in maintaining useful algorithms.

Further, after accessing data, there is the issue of actually operating on the data that was accessed. When discussing how much memory is needed to perform an operation or algorithm, we consider the space complexity of the algorithm. While Big-O notation was used to describe the computational efficiency (time complexity) of an algorithm, we will use Big-Θ notation to describe the space complexity of an algorithm. Given functions f and g, we say that $f(n) \in \Theta(g(n))$ if there exists n_0 such that for all $n > n_0$ there exist k_1, k_2 such that $k_1 g(n) \leq f(n) \leq k_2 g(n)$. That is, f is asymptotically bounded above and below by g. This is similar to Big-O notation, except that that constants for the above and below bound can be different. The space complexity of both matrix multiplication and matrix inversion, for matrices of size $n \times n$, is $\Theta(n^2)$. Thus, even if some "big data sets" could be read to memory, it could be unfeasible use them in algorithms while still working in memory.

From this point on, rather than considering square matrices in general, we will only consider distance and radial basis function matrices. First, consider a list of two-dimensional coordinates that is recorded in an $n \times 2$ matrix, called the location matrix. From this, a distance matrix can be constructed in the following way: take the coordinates from rows i and j from the location matrix and find the Euclidean distance between these locations. This distance will be the $[i, j]$ entry of the distance matrix. This can be done with all pairs of locations to fill out the entire distance matrix. Because the entries of these matrices are determined using the euclidean distance (a metric), they are symmetric because metrics are symmetric and that their diagonals are zero for all entries because metrics are positive-definite. From a given distance matrix, D, we can construct a matrix of radial basis functions, K, by: $K = D^2 log(D)$, where all operations are done element wise. For further clarification, a radial basis function is one whose value only depends on the distance from the origin (i.e. $f(x) = f(||x||)$). In this case, the specific radial basis functions being used are thin plate splines (TPS), which are a special type of polyharmonic splines. Sums of radial basis functions are often used in statistics in function approximation, such as kernel smoothing techniques, and in machine learning algorithms, such as support vector machines. In the context of the probabilistic algorithms described above, these matrices will act as inputs.

It is clear that, if given a sufficiently long list of coordinates, the respective distance matrix will become increasingly large, to the point that it cannot be wholly stored in memory. Further, even if they can be stored in memory, there is no guarantee that there will remain enough space in memory to perform matrix operations with them. In both algorithms described above, the input matrix A is used in multiplications in a number of steps of the first stages of the algorithms. If A is sufficiently large, then these matrix multiplications cannot be carried out in memory and would have to be done on disk, which would significantly slow down the algorithm. To avoid this problem, we implemented partial matrix generating functions for the distance and TPS matrices and a "slice matrix multiplication" algorithm. The former are functions that inputs a location matrix, a list of rows, and list of columns and outputs the indicated rows and columns of the distance/TPS matrix of the respective location matrix.

The latter "slice matrix multiplication" works by taking row slices of the distance/TPS matrix (e.g. slices of size $s \times n$) and right multiples them sequentially with a given, conformable matrix. After each multiplication, the result is vertically concatenated with the previous results, generating the resultant matrix product. A specific implementation for Stage 1, Step b of both probabilistic algorithms is as follows:

Input: location matrix locs, number of locations n, generating function gen_fcn, number of target singular vectors k, oversampling parameter l, and slice size slice

```
1. Generate empty matrix Y of dimension $n \times (k + l)$
2. Generate random number matrix $\Omega$ of dimension $n \times (k + l)$
3. Define start_row as the sequence from 1 to $n$ by size slice
4. Define end_row as following: take start_row and remove the first entry, subtract 1 from
   all remaining entries, and concatenate $n$ to the end of it
5. for $i$ in 1, 2, ..., length(start_row)
    (a) rows = start_row[$i$] = end_row[$i$]
    (b) Y[rows, ] = gen_fcn(locs, rows) * $\Omega$
6. return Y
```

As with the previous algorithms, the slice matrix multiplication algorithm was implemented in both R and C++. To get a better understanding of the efficiency of these implementations, we generated lists of 100, 1000, 10000, and 100000 coordinates to use as test inputs for the algorithms. For each of the lists of coordinates, a number of values for k, the target number of singular vectors, and slice were used (where $k \leq$ slice), while l was left constant at $l = 0$ as varying it would have the same effect as varying k. For example, for the 1000 coordinate list, slice was set to values of 10 and 100, while k was set to values of $5, 10$, and 100.

It is clear that when k is increased, the time taken for the algorithm to run would increase. Thus, it is more important to see how varying the slice size affects the algorithm's runtime. The graphs on the following page present the runtime for the slice matrix multiplication algorithm implemented in R and C++ for a list of 10000 coordinates for slice sizes set from 50 to 1000 incrementally increasing by 50 and k set to 5.

These graphs present two different stories. First, in general, the C++ implementation ran more quickly than the R implementation, averaging a runtime of about 2.87 times faster. With regards to the C++ implementation, other than the outlier for slice = 100, there appears to be a positive linear relationship between slice size and run time ($R^2 = .65$). It is important to note that the observed run time for slice = 100 is not an aberration. Each slice size was tested multiple times and the run time for slice = 100 was consistently much larger than the other run times. The cause of this is unknown. However, the linear relationship observed for the rest of the data is due to backend parallelization of the algorithm that distributes the many slice multiplications to various cores, which perform the multiplications at the same time. For small slice sizes, these multiplications occur more quickly. The speed gain due to the small slice sizes is further maintained when the parallelized jobs are brought back together to form the resultant product matrix, which accounts for the more faster run times for smaller slice sizes.

On the other hand, there is no clear pattern for the run times for the R implementation. The fastest run time is again observed when slice is minimized and the slowest run time is when slice is 100, but there is no linear relationship between slice size and run time (sans slice = 100, $R^2 = .01$). Again, each slice size was tested repeatedly, but still no pattern emerged. The cause for this sporadic behavior is unknown, but could be due to a trade off between the speed of each slice multiplication and the number of multiplications needed, issues with backend parallelization, and added run time for slice sizes that are not divisors of the number of locations.

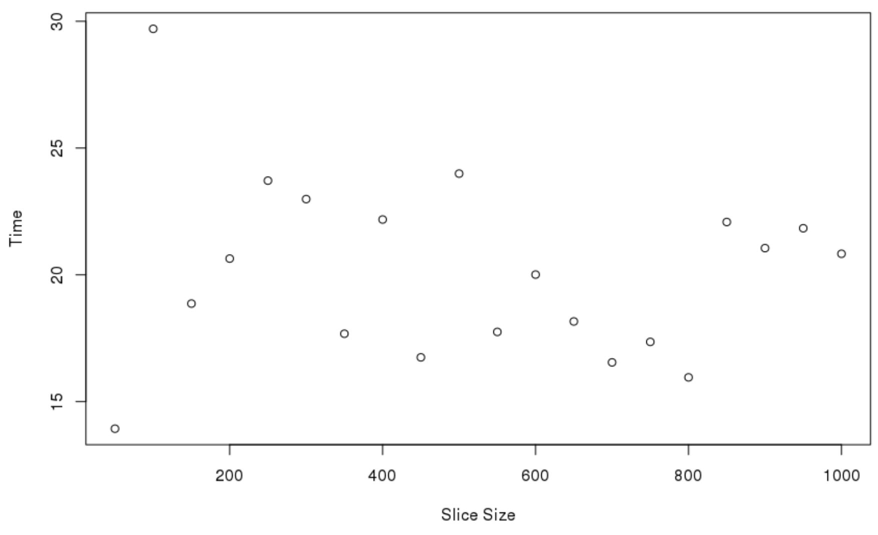


Figure 6: Slice Matrix Multiplication R Implementation

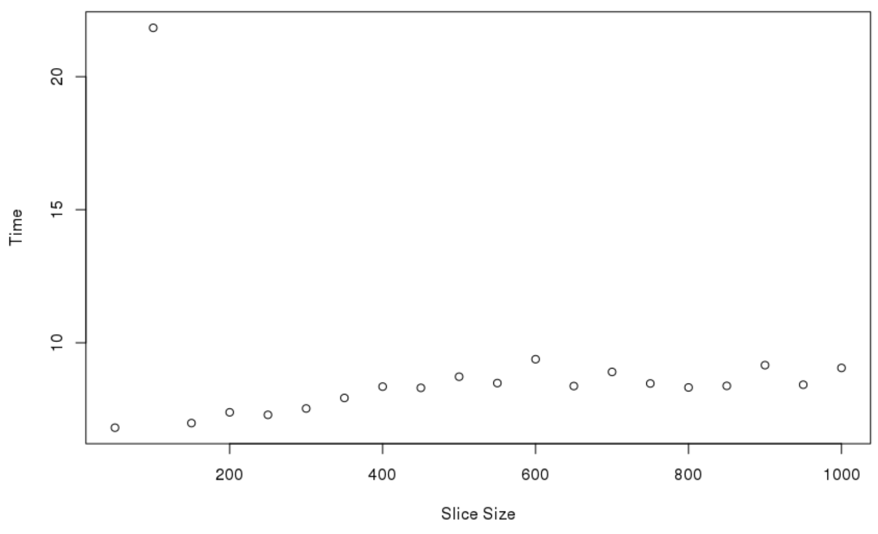


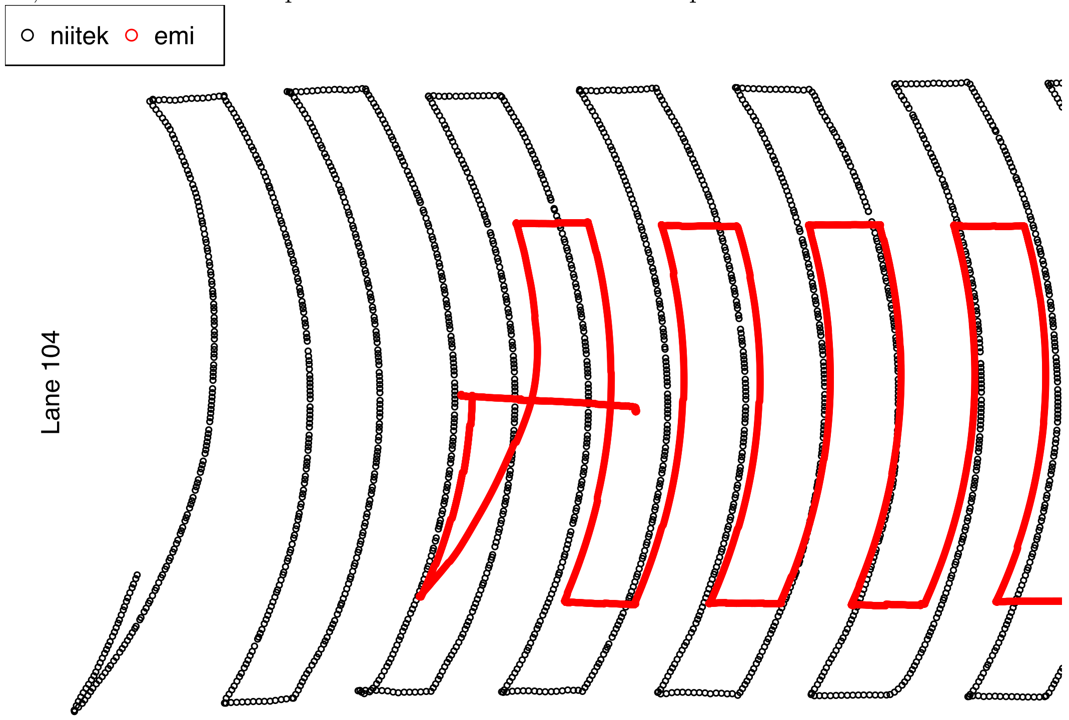
Figure 7: Slice Matrix multiplication C++ Implementation

5 Thin Plate Splines and Land Mines

Land mines are potent defensive explosive devices that are concealed or buried underground. They typically come in two varieties: anti-tank mines use to disable tanks and anti-personnel mines used to primarily injure people. The former are typically larger, require more pressure to set off, and are made of metal, while the latter are smaller (making them easier to conceal), require less pressure to set off, and are made of plastic. Anti-personnel mines are particularly known for their debilitating effects and there have been efforts to eliminate their use in the battlefield, notably through the drafting of the Ottawa Treaty. Unfortunately, due to their potency as an area denial weapon, both anti-tank and anti-personnel mines remain popular weapons.

In order to counter the potency of land mines, many countries have developed advanced minesweeping (detecting) and mine clearing (removing) technologies. In regards to the former, a minesweeper will often employ some sort of detecting sensor, often a metal detector. The sweeper will "sweep" the detector over the potential minefield. For example, the sweeper might take a number of measurements left to right over an arbitrary horizontal area. He would then sweep vertically while taking measurements before sweeping right to left. This zig-zag sweeping motion provides a voluminous number of discrete measurements, though most points on the ground will not be measured by the detector.

Current research and development into detector technology has focused partially on using a hybrid dual-sensor approach to detecting mines. Specifically, many organizations are seeking to integrate ground penetrating radar (GPR) and metal detecting sensors into a single instrument. The benefit of using both of these sensors is that they complement each other as, for example, metal detecting sensors often can't identify anti-personnel mines made of plastic. A major drawback of this hybrid approach is that the sensors are necessarily offset from each other, as shown in the image below. Thus, in regards to the sweeping described above, the two sensors will provide data at close but different points.



Thus, the hybrid detectors are left with two data issues: the data they collect is discrete and they collect two complementary, but offset sets of data. To correct for these issues, we want to fuse, interpolate, and smooth the data to create a single, continuous data set which would then be the basis of a detection algorithm for flagging potential land mines within an examined area. One tool used to do this is thin plate spline interpolation. In the previous section, thin plate splines were defined as radial basis functions of the form $f(r) = r^2 * log(r)$ where all operations are taking element-wise. Sums of radial basis functions are often used to approximate functions in a technique known as kernel smoothing.

In our consideration of land mines, we have data of the form (x_i, y_i, z_i) where we wish to predict z_i using x_i and y_i for all i. Thus, we seek to fit a regression model to this data, subject an additional following smoothness penalty:

$$\underset{f(x,y)}{\arg\min} \quad \sum_{i=1}^{n}(z_i - f(x_i, y_i))^2 + \lambda \int\int \left(\frac{\partial^2 f}{\partial x^2} + 2\frac{\partial^2 f}{\partial x\,\partial y} + \frac{\partial^2 f}{\partial x^2}\right) dx\,dy$$

This penalty is the 2D analog of the 1D smoothing spline. It has a natural solution of the form:

$$f(x, y) = \sum_{i=1}^{n} w_i \left((x - x_i)^2 + (y - y_i)^2\right) \log \sqrt{(x - x_i)^2 + (y - y_i)^2}.$$

This solution is a weighted sum of radial basis functions, specifically thin plate splines. The goal of this TPS model is to estimate the weights, w_i of the model. Regression type predictors can be added to this TPS model and their coefficients can be estimated at the same time as the TPS weights. This combined model is of the following form:

$$\underset{n'\times 1}{f(\boldsymbol{x}, \boldsymbol{y})} = \underset{n'\times p}{\boldsymbol{X}} \; \underset{p\times 1}{\boldsymbol{\beta}} + \sum_{i=1}^{n} w_i \left(\Big(\underset{n'\times 1}{\boldsymbol{x}} - x_i\Big)^2 + \Big(\underset{n'\times 1}{\boldsymbol{y}} - y_i\Big)^2\right) \log \sqrt{\Big(\underset{n'\times 1}{\boldsymbol{x}} - x_i\Big)^2 + \Big(\underset{n'\times 1}{\boldsymbol{y}} - y_i\Big)^2}.$$

To solve for the β's and w_i's, we can solve:

$$\begin{bmatrix} \underset{n\times n}{\boldsymbol{K}} & \underset{n\times p}{\boldsymbol{X}} \\ \underset{p\times n}{\boldsymbol{X'}} & \underset{p\times p}{\boldsymbol{0}} \end{bmatrix} \begin{bmatrix} \underset{n\times 1}{\boldsymbol{w}} \\ \underset{p\times 1}{\boldsymbol{\beta}} \end{bmatrix} = \begin{bmatrix} \underset{n\times 1}{\boldsymbol{z}} \\ \underset{p\times 1}{\boldsymbol{0}} \end{bmatrix}$$

Here $\boldsymbol{K}$ is the matrix of radial basis functions and $\boldsymbol{X}$ is the design matrix for the regression predictors. When we solve for w and β we get:

$$\begin{bmatrix} w \\ \beta \end{bmatrix} = \begin{bmatrix} K & X \\ X' & 0 \end{bmatrix}^{-1} \begin{bmatrix} z \\ 0 \end{bmatrix} = \begin{bmatrix} K^{-1} + K^{-1}X(-X'K^{-1}X)^{-1}X'K^{-1} & -K^{-1}X(-X'K^{-1}X)^{-1} \\ -(-X'K^{-1}X)^{-1}X'K^{-1} & (-X'K^{-1}X)^{-1} \end{bmatrix} \begin{bmatrix} z \\ 0 \end{bmatrix}$$

$$w = (K^{-1} + K^{-1}X(-X'K^{-1}X)^{-1}X'K^{-1})\,z$$
$$\beta = (-(-X'K^{-1}X)^{-1}X'K^{-1})\,z$$

Most of the above is not difficult to calculate. However, for large data, calculating K^{-1} will be time and space expensive. Current algorithms will typically sample a subset of points from the larger data set in order to achieve reasonable runtimes. This presents the obvious issue of accuracy deficiency as only some of the data is utilized. The probabilistic algorithms described above will hopefully allow for the inclusion of more data as they are more time and space efficient than similar deterministic algorithms.

In implementing the above model, we utilized two data sets called "NII" (36766 observations) and "EMI" (224458 observations). The NII data set has three variables: x coordinates, y coordinates, and a measurement using ground penetrating radar at that x,y location. The EMI set consists of x and y coordinates as well, but includes metal detector measurements rather than GPR measurements. Also considered are subsets of NII and EMI of size 4219 and 2665 observations respectively. These subsets are determined by choosing points from the full data sets that sit on a particular grid. They are utilized as training sets to train prediction models for the complete sets of data. Overall, the above model uses these discrete data to create smooth surface profiles for the GPR and metal detector measurements. Further modeling can fuse these smooth surfaces to create a single model to predict whether or not a land mine is present at a certain location.

The above model was implemented using the classical inverse and the random inverse algorithm described in section 3. The parameters for the probabilistic algorithm were set at: $q = 1$, $l = 5$, and $k \in \{100, 500, 1000\}$. We recorded the time it took for the implementations to run and the model prediction error using root mean squared error. Prediction was done in and out of sample, using the subset data and full data as the true values respectively. The results are presented in the tables below:

EMI

Type	k	Time	In Sample Error	Out of Sample Error
Probabilistic	100	2.680	.312	.550
Probabilistic	500	5.371	.454	.405
Probabilistic	1000	11.587	.307	.274
Deterministic	n/a	12.703	.055	.114

NII

Type	k	Time	In Sample Error	Out of Sample Error
Probabilistic	100	5.962	.391	.303
Probabilistic	500	10.522	.857	.657
Probabilistic	1000	19.599	1.41	1.69
Deterministic	n/a	18.912	.097	.142

Overall, the deterministic algorithm performed better error-wise, both in and out of sample, than the probabilistic algorithms regardless of the set value of k, while the probabilistic algorithms generally ran in less time than the deterministic one. An interesting observation made was that, while it was expected that accuracy would increase for the probabilistic algorithm as k increased, that was not always the case. There are few possible causes for this discrepancy. The first is that the subsets of data that we have are already highly structured as the points they sampled were chosen on a grid. The probabilistic algorithm, in effect, attempts to find structure in the data that it is given. Because the data is already highly structured, this exercise might lead to the probabilistic implementation being more inaccurate. Another possibility is that, when it comes to prediction, not all parts of the TPS matrix are equally important. Rather, certain parts, namely the diagonal, are more influential in prediction accuracy. Thus, random differences in the stochastic matrix could lead to better and worse TPS matrices for prediction based on what the diagonal ends up being.

Regardless, the results are fairly promising. The probabilistic algorithm ended up being faster than the deterministic algorithm and the errors were relatively similar enough. Further tuning and corrections can be made to improve the accuracy of the probabilistic algorithm without sacrificing time efficiency, which should make it an overall better option than the deterministic algorithm.

6 Next Steps

There are a couple of future directions that can be pursued to improve on the prediction of the previous implementation. First, to improve the issue regarding the diagonal of the inverse TPS matrix, we can utilize the Sherman-Woodbury-Morrison (SWM) Identity. The matrix inverse algorithm outputs the approximate eigenvectors and eigenvalues of the TPS matrix, such that $K \approx Q \Lambda Q^T$. Because we want to improve the approximation of the diagonal of K, we can find a diagonal matrix C such that $C = diag(K) - diag(Q\Lambda Q^T)$. Thus, $K \approx C + Q\Lambda Q^T$. However, inverting a matrix K of this form is not obvious. Fortunately, the SWM identity provides a concise way to invert it. It states that, given conformable matrices A, U, B, V,

$$(A + UBV)^{-1} = A^{-1} - A^{-1}U(B^{-1} + VA^{-1}U)^{-1}VA^{-1}$$

In our case, we would consider,

$$K^{-1} \approx (C + Q\Lambda Q^T)^{-1} = C^{-1} - C^{-1}Q(\Lambda^{-1} + Q^T C^{-1} Q)^{-1} Q^T C^{-1}$$

Because C and Λ are both diagonal matrices, there inverses are found simply by taking the reciprocal of their diagonal elements. Thus, the only new inverse that would need to be calculated would be: $(\Lambda^{-1} + Q^T C^{-1} Q)^{-1}$, which should be done reasonably efficiently using traditional algorithms because Q is low rank. Further, this approximate inverse TPS matrix should be more accurate because of the improved calculation of the diagonal of K.

Another future step could be using the probabilistic algorithm to seek out subspace structure in the full NII and EMI data sets. To do that, rather than using the subset data in the model creation, we would use the full data and set k to the number of observations in the subset data. Ideally, the probabilistic algorithm would be able to find structure in a smarter way than just choosing data points from a grid, which would lead to better predictive accuracy. This would also allow for the implementation of the slice matrix multiplication algorithm as using the full data set and such a large value for k would probably necessitate it to maintain time and space efficiency.

Taken together, these two steps should improve the predictive accuracy of the probabilistic algorithm, making it competitive with the deterministic algorithm. Once this competitiveness is achieved, issues regarding data fusion can be addressed.

References

[1] Gu, Ming, and Stanley C. Eisenstat. "Efficient Algorithms for Computing a Strong Rank-Revealing QR Factorization." SIAM Journal on Scientific Computing 17.4 (1996): 848-69. Web.

[2] Halko, N., P. G. Martinsson, and J. A. Tropp. "Finding Structure with Randomness: Probabilistic Algorithms for Constructing Approximate Matrix Decompositions." SIAM Review 53.2 (2011): 217-88. Web.

[3] Jacobs, Adam. "The Pathologies of Big Data." Queue 7.6 (2009): 10. Web.