To appear in Neural Computation

Methods for Binary Multidimensional Scaling

Douglas L. T. Rohde
School of Computer Science, Carnegie Mellon University,
and the Center for the Neural Basis of Cognition

Abstract

Multidimensional scaling (MDS), roughly speaking, is the process of transforming a set of points in a high dimensional space to a lower dimensional one while preserving the relative distances between pairs of points. Although effective methods have been developed for solving a variety of MDS problems, they mainly depend on the vectors in the lower dimensional space having real-valued components. For some applications, the training of neural networks in particular, it is preferable or necessary to obtain vectors in a discrete, binary space. Unfortunately, MDS into a low-dimensional discrete space appears to be a significantly harder problem than MDS into a continuous space. This paper introduces and analyses several different methods for performing approximately optimized, binary MDS.

1. INTRODUCTION

Recent approaches to artificial intelligence and machine learning have come to rely increasingly on data-driven methods that involve large vector spaces. One application of high-dimensional vectors that is particularly relevant today is in representing the contents of large collections of documents, such as all texts available on the internet. The similarity structure in these vector spaces can be exploited to perform a variety of useful tasks, including searching, clustering, and classification (see, e.g., Deerwester, Dumais, Furnas, Landauer, & Harshman, 1990; Berry, Dumais, & O'Brien, 1994). Other popular applications of vector spaces include representing the content of images (Beatty & Manjunath, 1997) and the meanings of words (Lund & Burgess, 1996; Burgess, 1998; Clouse, 1998).

However, it is often inefficient, if not intractable, to perform complex analyses directly in high-dimensional vector spaces. If one could reduce the set of high-dimensional vectors to a set of vectors in a much lower-dimensional space, while preserving their similarity structure, operations could be performed more efficiently on the smaller space with the

This research was supported by NIMH NRSA Grant 5-T32-MH19983 for the study of Computational & Behavioral Approaches to Cognition. Special thanks to David Plaut for his comments and guidance, Santosh Vempala for his brilliance, and to Daniel Clouse and Stephen J. Hanson for their helpful comments.

potential added benefit of improved results due to reduced noise and greater generalization. Scaling to a space with just one, two, or three dimensions also permits easy visualization of the resulting space, which can lead to a better understanding of its overall structure.

In order to place the current work in an historical framework, let us briefly trace the development of modern multidimensional scaling (MDS) techniques. Most applications of MDS, particularly in the psychological domains, have been in the analysis of human similarity ratings. Thus, rather than beginning with points in a high dimensional vector space, a more common starting point has been a matrix of pairwise comparisons of a set of items. Various types of comparisons might be used, including similarity or dissimilarity judgments, confusion probabilities, or interaction rates. One problem introduced by the use of measures of this sort is that it isn't clear how best to scale these ratings so that they correspond directly to distances in the vector space. Subjects' ratings may be quite skewed and are likely to be non-metric.

Perhaps the earliest explicit and practical MDS method was that of Torgerson (1952), which grew out of the work of Richardson (1938) and Young and Householder (1938), among others. Torgerson used a one-dimensional scaling technique to convert dissimilarity ratings to target distances and then attempted to find a set of points whose pairwise Euclidean distances best matched the target distances, according to mean-squared error. The initial scaling function might simply be a linear transformation, or could be a non-linear function, such as an exponential. While quite effective, the formal requirements of this technique are too strong for many applications and a serious drawback is that the proper scaling method is difficult to determine and may vary from one problem to the next.

The next major advance was made by Shepard (1962), who suggested that, rather than attempting to directly match scaled target distances, the goal of MDS should be to obtain a monotone relationship between the actual point distances and the original dissimilarities. Thus, the dissimilarities need not be scaled, and their values are actually discarded altogether. All that is retained is their relative ordering. Torgerson's earlier approach came to be known as *metric* MDS and this new technique as *non-metric* MDS. However, Shepard didn't provide a mathematically explicit definition of what constitutes a solution.

Kruskal (1964a, 1964b) further developed the method by explicitly defining a function, known as *stress*, relating the pairwise distances and the ranking of dissimilarities. Stress essentially involves the scaled sum-squared error between the pairwise distances and the best-fitting monotonic transformation of the original dissimilarity ratings. Iterative gradient descent was used to find the configuration with minimal stress.

The basic technique developed by Shepard and Kruskal has remained the standard for most applications of MDS to psychological phenomena (Shepard, Romney, & Nerlove, 1972; Borg & Groenen, 1997). Although possibly slower, gradient descent techniques have the advantage over matrix-algebra methods in that they can more easily tolerate missing or sparse data and can be used to minimize any differentiable measure of stress. Non-metric MDS is quite effective when similarity ratings involve unknown distortions. However, relying on rank order sometimes discards information that can't be recovered (Torgerson, 1965). This may be particularly true in cases where the structure of the data involves a number of tight clusters that are well separated (Shepard, 1966). Thus, metric methods may be more suitable for some types of data, but for the vast majority of problems of practical interest,

non-metric methods are likely to be as good, if not better.

A common feature of the MDS techniques discussed thus far is that they rely on the final vector space having real-valued components. However, some applications require vectors with discrete, usually binary, components. That is, the vectors should lie at the corners of a unit hypercube. An important application of this type is the development of representations for training neural networks. Increasingly, neural networks that serve as cognitive models are trained using inputs or targets derived from real data, rather than artificially generated vector sets. But those data sets may involve vectors of high dimensionality, possibly in the tens or hundreds of thousands, and it would be computationally intractable to train a network using such large vectors.

Furthermore, neural networks with thresholded outputs, particularly recurrent attractor networks (Pearlmutter, 1989; Plaut & Shallice, 1993), often learn better when their vector targets use binary components. It is harder for the network to accurately drive output units to intermediate levels of activation than to drive them to fully active or inactive states. Thus, it is sometimes necessary to scale a set of high-dimensional vectors to a relatively low-dimensional, binary vector space.

Binary MDS (BMDS) is a much harder problem than standard MDS. In fact, it has been shown that embedding a metric distance space in a bit space with minimal distortion is NP-complete (Deza & Laurent, 1997).[1] Thus, there is a good chance that no polynomial-time algorithm exists to compute an optimal set of BMDS vectors. However, it may still be possible to efficiently compute an approximation to the optimal BMDS solution.

This paper presents several methods for performing approximately optimal BMDS. The solutions fall in two broad classes: those that perform the optimization directly in bit space and so-called *hybrid* methods that perform the optimization in a real-valued space before converting to a bit space.

The first hybrid method is somewhat similar to Torgerson's linear-algebraic approach to MDS. It begins by computing the singular value decomposition of the matrix of initial vectors. The right singular vectors are then converted to bits using a unary encoding, with more bits assigned to vectors having larger singular values.

Two other hybrid methods are based on Shepard and Kruskal's technique for MDS. Gradient descent is performed in a real-valued vector space using the stress cost function before the vector components are converted to bits based on their signs. One of these variants is metric in that it uses the actual target values in computing the stress. The other method uses a monotonic transformation of the target values, rather than the values themselves.

The major problem with hybrid techniques is that important information can be lost in the discretization. A very good real-valued solution may turn into a very bad binary solution. An alternative approach is to perform the bulk of the optimization directly in bit space. The only known prior algorithm for computing BMDS directly in bit space is that

[1] Actually, it is NP-complete to decide whether a metric distance space is ℓ_1-embeddable with no distortion. It follows that finding a minimally distorted embedding of a metric space into a binary space under an ℓ_1 distance measure (such as Hamming distance) is NP-complete. However, the original set of distances in the BMDS problems considered here are more restricted than a metric space because they represent distances between pairs of points. Thus, the known proof may not apply. Nevertheless, it seems likely that deciding whether an ℓ_2- or ℓ_1-embeddable metric space is embeddable in an ℓ_1-space of lower dimensionality is also an NP-complete problem.

of Clouse and Cottrell (1996) and Clouse (1998). Their method began by creating a set of bit vectors for each item by thresholding values from the original high-dimensional vector. They then performed a random walk by repeatedly selecting bits at random, computing whether flipping the bit would improve the overall cost, and doing so only when beneficial. Once it was determined that no improvements could be made by flipping any one bit, the algorithm terminated.

One drawback of the Clouse and Cottrell method is that it is computationally inefficient. Without good record keeping, it is costly to determine whether a bit flip is advantageous. As the number of remaining good bits diminishes, the algorithm becomes less and less efficient because many bits must be tested before any progress can be made. The alternative, exhaustively testing all bits before deciding which to flip, is not much better. Thus, the algorithm doesn't scale well to larger problems.

The first fully-binary method presented here is an improved version of Clouse and Cottrell's algorithm. By using careful record keeping, it is able to keep track of the change in cost that would result from flipping any bit and to quickly find the bit that would result in the greatest improvement. Although there is some cost for the record keeping, it is more than made up for by the fact that the algorithm need never test a bit only to discover that flipping it would be counter-productive. A more effective, though less efficient, version of this algorithm minimizes the sum of the squared difference between actual and target distances, rather than the sum of the absolute differences.

The final method introduced in this study constructs the bit vectors sequentially, choosing the first bit in each vector, followed by the second bit in each vector, and so on. The algorithm then iterates several times, reassigning the bits in each dimension given the other bits previously chosen. Although simple and relatively easy to implement, this method is quite fast and effective.

The next section introduces the tasks and metrics used to evaluate the BMDS methods. Each method is then described in further detail, including the details of its implementation, advantages and disadvantages, and some possible variations. Finally, the methods are evaluated in terms of performance and running time. It is hoped that this study will prove useful to researchers interested in immediate applications of binary multidimensional scaling and that it will inspire future advances in these methods.

2. EVALUATION METRICS AND EXAMPLE TASKS

Before describing the actual BMDS algorithms, we begin by defining the scaling task more explicitly. The input is a set of N real-valued vectors of dimensionality M, representing N items. The output is a set of N bit-vectors with dimensionality D. The goal is for the relative distances between the final vectors to reflect the relative distances between the initial vectors as closely as possible. To make this more concrete, we must define the functions measuring pairwise distance in the original and final spaces and a measure of how well these two sets of distances agree.

There are a number of reasonable distance metrics for the original space, four of which are shown in Table 1. Euclidean distance or city-block distance are standard choices. However, they are dependent on the dimensionality, M, and the scaling of the vectors, making them somewhat inconvenient. Cosine is another reasonable choice. It is scale

Distance Measure	**Formula**
City-block	$\sum_k \lvert x_k - y_k \rvert$
Euclidean	$\sqrt{\sum_k (x_k - y_k)^2}$
Cosine	$0.5 - 0.5 \frac{\sum_k x_k y_k}{\sqrt{\sum_k x_k^2 \sum_k y_k^2}}$
Correlation	$0.5 - 0.5 \frac{\sum_k (x_k - \bar{x})(y_k - \bar{y})}{\sqrt{\sum_k (x_k - \bar{x})^2 \sum_k (y_k - \bar{y})^2}}$

Table 1: Some candidate distance measures.

invariant and is confined to a fixed range, $[-1, 1]$, which is more convenient than measures that depend on dimensionality and average value magnitude.

A fourth possibility, and the one used in this study, is to base the distance measure on Pearson's correlation. Like cosine, it is scale invariant and is confined to a fixed range, $[-1, 1]$. Computationally, correlation and cosine are identical except that cosine is calculated using the actual vector components while correlation is based on the differences between vector components and their mean. If components are evenly distributed between positive and negative values, their mean is usually close to zero and cosine and correlation are quite similar. But if components are constrained to be non-negative, cosine will be positive while correlation continues to use the full $[-1, 1]$ range. Correlation is thus a good initial choice for many scaling problems. In practice, using correlation has lead to better results than using the other three measures with several different tasks and BMDS algorithms.

In order to turn correlation into a distance measure, it is scaled by -0.5 and shifted by 0.5 so that a correlation of 1.0 becomes a distance of 0 and a correlation of -1.0 becomes a distance of 1.0. The set of all pairwise correlation distances was then scaled by a constant factor to achieve a mean value of 0.5, although this has little practical effect because the mean distance tended to be very close to 0.5 before scaling. This linearly transformed correlation will be referred to as *correlation distance*. Although it was not done here, one could scale the resulting values using an exponential with exponent greater than 1 to increase the influence of larger distances or less than 1 to enhance the smaller distances.

The simplest and most reasonable choice for the distance metric in bit space seems to be Hamming (city-block) distance. That is, the distance between two vectors is the number of bits on which they differ. Note that for bit vectors, Euclidean distance is just the square root of the Hamming distance. Likewise, if the bit vectors tend to have a roughly equal number of 1s and 0s, which seems to be the case with most of these BMDS methods in practice, Hamming distance is closely approximated by correlation distance (when scaled by D).

The third function that we must specify evaluates the agreement between the correlation distances in the original space and the Hamming distances in the final binary space. It's not entirely clear what is the best measure. One obvious choice is to use Kruskal's *stress* (Kruskal, 1964a):

$$\textit{metric stress} = \sqrt{\frac{\sum_{i<j} (d_{ij} - t_{ij})^2}{\sum_{i<j} d_{ij}^2}}$$

where i and j together iterate over all pairs of items, d_{ij} is the Hamming distance of i and j's bit vectors and t_{ij} is the correlation distance of i and j's initial vectors, scaled by the dimensionality of the bit vectors, D.

An alternative form of this measure, and the one actually used by Kruskal, is non-metric stress. In this case, the actual distances are not directly compared to the target distances but to the best monotonic transformation of the target distances:

$$\textit{non-metric stress} = \sqrt{\frac{\sum_{i<j}(d_{ij} - \hat{d}_{ij})^2}{\sum_{i<j} d_{ij}^2}}$$

where $\hat{d}_{ij}$ are those values that achieve minimal stress, under the constraint that the $\hat{d}_{ij}$ have the same rank order as the corresponding t_{ij}. Non-metric stress is a better measure if one is only concerned with preserving the rank-order relationship between pairwise distances. But if one is concerned with directly matching the target distances, metric stress is preferable.

An alternative method of evaluating the final vectors is to compute Pearson's correlation between the original set of pairwise distances among vectors and those of the final vectors. This measure is referred to here as *goodness* and is defined mathematically as follows:

$$\textit{goodness} = \frac{\sum_{i<j}(d_{ij} - \bar{d})(t_{ij} - \bar{t})}{\sqrt{\sum_{i<j}(d_{ij} - \bar{d})^2 \sum_{i<j}(t_{ij} - \bar{t})^2}}$$

Note that the optimal stress value is 0 and the optimal goodness value is 1; better vectors should result in lower stress but higher goodness. Goodness has the property that it is unaffected by linear transformations of the distances, so scaling and shifting the target distances has no effect on goodness. Because metric stress, non-metric stress, and goodness do not always agree on which is the best set of vectors, all three measures are reported in the analysis found in Section 9. In Section 9.3, the practical differences between these measures is discussed.

2.1. Example tasks

Two BMDS tasks were used in testing the algorithms presented here. The first, known as the *Exemplar* task, was completely artificial. It consisted of 4,000 vectors of dimensionality 1,000, generated in the following way. First, 10 random bit vectors of length 50 were produced. Each of the 4,000 vectors was created by taking one of the 10 exemplars, flipping each bit with 10% chance, and then doing a random projection to real-valued 1,000-dimensional space. The resulting vector set has a basically simple similarity structure with a good deal of randomness superimposed. Because, prior to the random projection, the vectors occupied a 50-dimensional bit space, the vectors should be quite compressible.

The second task, the *Word* task, involves 5,000 vectors of length 4,000 representing word meanings. The vectors were generated using a method similar to HAL (Lund & Burgess, 1996). Word co-occurrences were gathered over a large corpus of Usenet text. Raw co-occurrence counts were converted to a ratio of the conditional probability of one word occurring in the neighborhood of another to the word's overall probability of occurrence.

The first 4,000 values of each word's vector, reflecting its co-occurrences with the 4,000 other most frequent words, was taken as the word's meaning vector. This set has a more complex similarity structure than the Exemplar set.

Note that these problems are considerably larger than those to which MDS is typically applied, which generally involve no more than a few hundred items. Clouse and Cottrell (1996) reported an example task involving 233 words. Because any reasonable MDS algorithm will likely have a running time that is at least $O(N^2)$, the tasks studied here are effectively several hundred times more complex. Of critical concern will be not only the ability to achieve low stress or high goodness, but the running time of the various algorithms.

3. SVD: THE SINGULAR-VALUE DECOMPOSITION METHOD

The singular value decomposition (SVD) is the foundation for the Latent Semantic Analysis technique for document indexing and retrieval (Deerwester et al., 1990). It has also recently received considerable attention for its use in efficient clustering methods (Frieze, Kannan, & Vempala, 1998). It therefore seems natural to consider designing a BMDS algorithm using the singular value decomposition. This first method, which is based on computing the SVD of the item vector matrix, is somewhat related to the metric MDS technique of Torgerson (Torgerson, 1952).

Any real matrix, A, has a unique singular value decomposition, which consists of three matrices, $U\Sigma V$, whose product is the original matrix. The first of these, U, is composed of orthonormal columns known as the *left singular vectors* and the last, V, is composed of orthonormal rows known as the *right singular vectors*. Σ is diagonal and contains the *singular values*. The singular vectors reflect principle components of A and each pair has a corresponding value, the magnitude of which is related to the variance accounted for by the vector. If A is symmetric and positive semi-definite, the left and right singular vectors will be identical and equivalent to its eigenvectors and the singular values will be its eigenvalues.

Non-binary multidimensional scaling can be performed using the SVD as follows. Let A be the $M \times N$ matrix whose columns are the original item vectors. The SVD is computed and the right singular vectors are sorted by decreasing singular value. Only the first D vectors and values are retained. The new representation of item i is the vector composed of the ith value in each of the D highest right singular vectors, scaled by its corresponding singular value.

3.1. Discretization

In order to perform binary MDS, the values must be converted to bits. One could simply use the first D right singular vectors and assign a single bit to each component. But this would not be very effective because the vectors with highest value contain most of the useful variance. Furthermore, there are often fewer than D non-zero singular values. Thus, we may need to assign more than one bit to each right singular vector. The method found to be most effective is to assign the bits roughly in proportion to the magnitudes of the singular values. A deterministic procedure is used to accomplish this. The first bit is assigned to the vector with the largest singular value. Its value is then halved. The second bit is assigned to the vector with the singular value that is now the largest. Once two bits

have been assigned to a vector, its value is set to 1/3 of its original value. For three bits, its value is 1/4 of the original, and so on.

Assume, for example, that we had three singular vectors with values 12, 9, and 5 and we were to assign 5 bits ($D = 5$). The first bit goes to the first vector and its value is reduced to 6. The second bit goes to the second vector and its value is reduced to 4.5. The third bit goes to the first vector again, because it is once again the largest value. Its value is set to 4 (12/3). Bit 4 goes to the third vector, because it is now the largest, and its value is reduced to 2.5. The remaining values are 4, 4.5, and 2.5 and the last bit goes to the second vector. In the end, the first two vectors have been assigned two bits and the last vector one.

The bits are then given values using a unary encoding. If a vector has three bits assigned to it, the possible codes are 000, 001, 011, and 111. This may not seem to be the most efficient use of the bits, but it is the only method that has the appropriate similarity structure between codes. The codes are assigned to items as follows. Each right singular vector has N components corresponding to the N items. These are sorted and partitioned evenly into the same number of bins as there are unary codes. With three bits there would be four bins. The items in the first bin would receive the bits 000. Those in the second bin would receive the bits 001, and so on.

3.2. Running time

A major problem with the SVD method is that computing the SVD is quite slow. Because the matrices are dense, computing the SVD takes $\Theta(N^2(N+M))$ time. If N is larger than M, the matrix A can be transposed and the left singular vectors used, giving a running time of $\Theta(M^2(M+N))$. However, it is no faster to run the SVD algorithm with small D than with large D. Using a 500MHz 21264 Alpha processor, it takes 25 minutes to compute the SVD on the Exemplar task. But on the Word problem, with $N = 5000$ and $M = 4000$, it runs for nearly a day.

3.3. Variants

If one knows that a certain subset of items is representative of the others, it is possible to speed up the SVD computation by only using that subset to generate the singular vectors. Although this might enable the SVD method to tackle much larger problems, it hinders performance. As we'll see in Section 9, the SVD method of BMDS does quite poorly to begin with. Several alternative discretization methods have been tested, but were not found to be as effective.

4. MGD: THE METRIC GRADIENT DESCENT METHOD

The second hybrid method uses gradient descent to optimize the item vectors in a real-valued space before discretizing them. It is similar to more traditional MDS in the use of the stress cost function and gradient descent, but differs from them in that it is metric. The stress measure uses linearly transformed target distances, rather than monotonically transformed targets.

4.1. Initialization

The first step of the gradient descent method is to create an initial set of N real-valued vectors of dimensionality D. The vectors could be assigned randomly, as is typically done in standard MDS. However, this tends to result in an unnecessarily long minimization process. A better approach is to use a fast method to create moderately good initial vectors. One nice way to produce good initial vectors is with a random projection.

First, D random *basis vectors* of dimensionality M are generated. The elements of the basis vectors are draw independently from a Gaussian distribution. For each item, the correlation between its original vector, having dimensionality M, and each of the D basis vectors is computed, and these D correlations form the components of the item's initial vector in the smaller space.

This random projection is reasonably fast, $\Theta(NMD)$, and preserves much of the information in the original vectors, especially if D is large. Even without the minimization phase, the random projection goes a long way toward solving the BMDS problem. However, there is still considerable room for improvement to justify the more expensive optimization process.

4.2. The cost function and its derivative

The goal of the optimization phase is to minimize the stress between the actual vector distances and the scaled target vector distances:

$$S = \sqrt{\frac{S^*}{T^*}} = \sqrt{\frac{\sum_{i<j}(d_{ij} - b\ t_{ij})^2}{\sum_{i<j} d_{ij}^2}}$$

i and j together iterate over all pairs of items, d_{ij} is the city-block distance of i and j's vectors in the new space, t_{ij} is the correlation distance of i and j's initial vectors, and b is an adjustable scaling factor.

At the start of each step of the iteration, the value of b is computed that results in minimal stress. This method is commonly known as *ratio* MDS, as it seeks to minimize the discrepancy between actual and target distance ratios. The optimal value of b is given by:

$$b = \frac{\sum_{i<j} d_{ij} t_{ij}}{\sum_{i<j} t_{ij}^2}$$

Next, the derivative of the stress with respect to each of the ND vector components is computed. This derivative is given by the following formula:

$$\frac{\partial S}{\partial i_k} = \sum_j \left(\frac{d_{ij} - b\ t_{ij}}{\sqrt{S^* T^*}} - \frac{d_{ij}\sqrt{S^*}}{T^*\sqrt{T^*}} \right) \frac{\partial d_{ij}}{\partial i_k}$$

When using city-block distance, the derivative of distance d_{ij} w.r.t. component i_k is simply $sgn(i_k - j_k)$, or 1 if $i_k > j_k$ and -1 otherwise.

4.3. Component updates

Once the ND derivatives have been accumulated over all item pairs, the vector components are updated by taking a small step down the direction of steepest descent. The size of the step is scaled by a learning rate parameter, α. Following Kruskal (1964b), it is also scaled by the r.m.s. value of all vector components and the inverse of the r.m.s. derivative. The reason for this scaling is that it reduces the need to otherwise adapt the learning rate to the size of the overall problem. The component update formula is as follows:

$$i_k = i_k - \alpha \frac{\sqrt{\sum_{i,b} i_b}}{\sqrt{\sum_{i,b} \frac{\partial S}{\partial i_b}}} \frac{\partial S}{\partial i_k}$$

In order to prevent the vector components from shrinking or growing to a point where precision is lost, they are normalized following each update to maintain an r.m.s. value of approximately 1.

At the end of the minimization, the real-valued components are converted to bits based on their signs. Negative components become 0s and positive components become 1s.

4.4. Learning rate adaptation and stopping criterion

Despite the learning rate scaling factors, it is still necessary to adapt the learning rate as the minimization proceeds. In general, a larger learning rate is used initially and then progressively reduced as a minimum is approached. The initial value of the learning rate was 0.2, which has proven to be a good choice for tasks of widely varying size.

The general problem of automatically adapting the learning rate during a gradient descent to achieve the best performance is an interesting and difficult one. The following method is based on observations of what experienced humans do when adjusting a learning rate by hand. It is by no means optimal, but it does seem to work quite well for any gradient descent that has a smooth, fairly stable error function, such as the current problem or when training neural networks under batch presentation.

The learning rate and termination criterion are based on two measures, known as *progress* and *instability*. Progress, is the percent change in overall stress following the last weight update, and is defined as:

$$progress = P_t = \frac{S_{t-1} - S_t}{S_{t-1}}$$

where S_t is the current stress and S_{t-1} is the previous stress. A positive P value indicates that the stress is being reduced, which is good. If P ever becomes negative, the learning rate is immediately scaled by 0.75. This normally results in a return to positive progress on the next update.

Instability is a time-averaged measure of the consistency of the progress, and is defined as follows:

$$instability = I_t = 0.5 * \left(I_{t-1} + \left| \frac{P_{t-1} - P_t}{P_{t-1}} \right| \right)$$

Steady progress results in low instability. Whenever the learning rate changes, instability is reset to 10. If progress is high and the instability is low, things are proceeding well

and no changes are needed. If progress is unstable, it often indicates that the learning rate is a bit too high. But it is usually not worth lowering the rate unless negative progress is made. The only case where it is generally a good idea to increase the learning rate is when progress is slow and stable. Thus, whenever the progress is less than 0.02 (2%) and the instability is less than 0.2, the learning rate is scaled by 1.2.

The minimization terminates when the progress remains below 0.001 (0.1%) for 10 consecutive updates. On the example tasks used here, the algorithm generally terminates in between 50 and 250 updates, depending on the values of N and D.

4.5. Running time

As with any gradient descent technique, it is difficult to predict in advance how many updates will be required. In general, the length of the settling process increases a bit with larger N and D. The cost for each update is $\Theta(N^2D)$. Therefore, the algorithm tends to be somewhat worse than quadratic in N and somewhat worse than linear in D. The running time is evaluated empirically in Section 9.4.

4.6. Variants

One of the advantages of gradient descent methods is that they are extremely flexible and permit endless variation. In addition to the stress cost function and the city-block distance function, summed and sum-squared cost have also been tested, as well as Euclidean distance. None of these alternatives performed as well with metric gradient descent on the current tasks as did stress and city-block distance.

Furthermore, various methods of scaling the target distances have also been tested. Rather than scaling the targets by an adaptive factor, b, they could simply be used in their raw form of correlation distances scaled by D. Alternately, one could transform the distances by $a + b\, t_{ij}$, where a and b are both adjusted to minimize the overall cost. This is known as an *interval* scale. Using an interval scale, although more flexible, proved slightly less effective on the current problems. One could further loosen the restrictions on the target distances by using the optimal monotonic transformation, as in Kruskal and Shepard's non-metric MDS technique, but that is the subject of the next algorithm.

It may be possible to improve the rate of convergence with a better automated procedure for adjusting the learning rate. Another promising addition would be the use of a momentum term on the component update step, as is often done in training neural networks. Momentum adds a fraction of the previous step in vector space to the current step, which can often speed learning.

Finally, as one might expect, a major problem with this method of BMDS is that significant information is lost in the discretization step. Although the city-block distances between pairs of vectors may match the target distances very well, those city-block distances will not accurately reflect the Hamming distances of the corresponding bit vectors unless the real-valued components are close to -1 or 1.[2] Thus, it may be beneficial to introduce an additional cost term that penalizes components for being far away from -1 or 1. A simple

[2] If city-block distances between vectors with components that are expected to fall close to -1 or 1 are to be compared to Hamming distances of bit vectors formed from the signs of the components, the city-block distances must actually be scaled by 1/2.

polarizing cost function is the absolute value of the distance between the value and -1 or 1, whichever is closer. This would add a constant-size term to the value on each update, having the effect of pulling the value towards the closer of 1 and -1. Like the learning rate, the size of that step is an adjustable parameter.

A somewhat more sophisticated polarizing cost function is $(i_k^4 - 2i_k^2 + 1)/4$. This is shaped like a smooth W. It has concave upward minima at 1 and -1 and a concave downward maximum at 0. At values larger than -1 or 1 it increases rapidly, heavily penalizing large values. It is nice because there are no discontinuities, which can disrupt the gradient descent. The derivative of this function is simply $i_k^3 - i_k$. Thus, at each weight step, $i_k^3 - i_k$, multiplied by the cost parameter, is subtracted from each value.

Experimentation with these cost functions indicates that they are quite effective in speeding the convergence of the gradient descent. However, when using metric gradient descent they don't seem to do much to improve the quality of the resulting vectors.

5. OGD: THE ORDINAL GRADIENT DESCENT METHOD

This next method is based more closely on Shepard and Kruskal's gradient descent technique for MDS (Shepard, 1962; Kruskal, 1964a). Rather than using linearly scaled target values in computing the stress, the best-fitting monotonic transformation of the target values is used. Thus the method is non-metric, or ordinal, in that the actual target values are not important, only their rank ordering. Except where noted, all aspects of this method are identical to those for MGD, including the initialization step, the update step, and the learning rate adjustment.

5.1. Non-metric stress

Ordinal gradient descent uses as its cost function the non-metric stress measure:

$$S = \sqrt{\frac{S^*}{T^*}} = \sqrt{\frac{\sum_{i<j}(d_{ij} - \hat{d}_{ij})^2}{\sum_{i<j} d_{ij}^2}}$$

where $\hat{d}_{ij}$ are those values that minimize the stress, under the constraint that the $\hat{d}_{ij}$ have the same rank order as the corresponding t_{ij}. The derivative of this function w.r.t. component i_k is the same as for metric stress, except that $b\, t_{ij}$ is replaced by $\hat{d}_{ij}$.

The optimal $\hat{d}_{ij}$ values are computed on each iteration using the *up-down* algorithm of Kruskal (1964b). In terms of running time, this process is linear in the number of distance values, $O(N^2)$, and is thus significantly less costly than computing the component derivatives.

5.2. Sigmoidal components

As mentioned before, a major problem with the gradient descent method is the distortion introduced in discretizing the real values into bits. If the real values are either exactly -1 or 1, the city-block distance between real-valued vectors will exactly correspond to the Hamming distance of the bit vectors and there will be no loss of information. However, if the real values are much less than or greater than 1 in magnitude, the discretization will

introduce noise. This lead to the idea of adding a cost function that encourages the real values to be close to 1 or -1. However, such cost functions were not found to be terribly helpful in practice.

An alternative is to transform the vector components using a *sigmoid*, or *logistic*, function, which limits values to the range [0,1] and makes it easier for the gradient descent to achieve nearly-discrete values (Rumelhart, Hinton, & Williams, 1986). The sigmoid function has the following formula:

$$s(i_k) = \frac{1}{1 + e^{-g i_k}}$$

This function is shaped like a flattened S. If $i_k = 0$, $s(i_k) = 0.5$. As i_k increases above 0, $s(i_k)$ approaches 1. As i_k decreases below 0, $s(i_k)$ approaches -1. The parameter g is known as *gain* and controls how rapidly the sigmoid approaches its limits. The advantage of the sigmoid is that it is quite easy for the gradient descent to drive the effective vector coordinates, $s(i_k)$, close to 1 or 0 by driving the actual coordinates to large positive or negative magnitudes.

When using the sigmoid transformed components, the vector distance function and its partial derivative become:

$$d_{ij} = \sum_k |s(i_k) - s(j_k)|$$

$$\frac{\partial d_{ij}}{\partial i_k} = g \; s(i_k)(1 - s(i_k)) \; sgn(s(i_k) - s(j_k))$$

5.3. Polarizing cost

The sigmoid is only helpful if a good proportion of the vector components actually grow fairly large (and thus approach 0 or 1 when put through the sigmoid). In order to encourage this, an extra polarizing term is added to the cost function. In the absence of the sigmoid, such a cost function must be somewhat complex, as it should have the effect of pulling the values towards -1 or 1, but not beyond them. Two such functions were mentioned in Section 4.6. But when the sigmoid is used, the cost function need only push the raw values away from 0. Therefore, the simple linear cost function is used. This has the effect of adding a small constant to the positive values and subtracting a small constant from the negative values on each weight update. In evaluating this method, a constant of 0.05 was used.

Because the sigmoid avoids the problem of overly-large components and the polarizing term prevents an overall shrinking of components, there is no need to re-normalize the values periodically.

5.4. Running time

Because they involve an exponential, computing the sigmoids exactly can substantially slow down this algorithm. Therefore, a fast approximation to the sigmoid function was used. The sigmoids of 1024 values evenly distributed in the range [-16,16] were computed

in advance and stored in a table. In computing the sigmoid of a new value, the sigmoids of the two closest values in the table are linearly interpolated. This method is quite fast and is accurate to within 2×10^{-7} of the correct value.

Despite being somewhat more complex than MGD, the asymptotic running time of this method remains $\Theta(N^2D)$ per update. Because it tends to converge faster than the metric method, however, the overall running time is somewhat less.

5.5. Variants

The gain term in the sigmoid function determines how sharp the sigmoid is and thus how polarized the values are. A higher gain draws the resulting values closer to 0 or 1, and thus reduces the noise introduced in the discretization. However, a high gain can also impede learning in the minimization phase. Thus, one might think of starting with a small gain and gradually increasing it as the minimization progresses. However, attempts to do this resulted in no improvement over using a fixed gain of 1.0. Using other fixed gain values also seemed to make little difference.

6. GBF: THE GREEDY BIT-FLIP METHOD

This next method is quite similar to that of Clouse and Cottrell (1996), but the algorithm has been altered to achieve an asymptotically faster running time. Like the gradient descent techniques, this method performs a gradual minimization. However, rather that working in a real-valued space, it operates directly in bit space.

The optimization proceeds by flipping individual bits, in an attempt to minimize the linear cost function:

$$cost = \sum_{i<j} |d_{ij} - t_{ij}|$$

where d_{ij} is the Hamming distance between the bit vectors for items i and j and t_{ij} is the correlation distance between their original vectors, scaled by D, the number of bits per vector. The advantage of the linear cost function is that the contribution of individual bits in a vector to the cost are often independent of one another, which allows the algorithm to be much more efficient. In the next method, GBFS, we consider the effect of using squared rather than absolute cost.

6.1. Initialization

The initial bit vectors are formed using a random projection, as in the gradient descent methods. However, rather than using the actual correlations with the basis vectors as the components of the initial vectors, these correlations are converted to bits based on their sign. Negative correlations become 0s and positive correlations become 1s. This method has the property that 1s and 0s are expected equally often.

6.2. Minimization

The minimization phase is conceptually very simple. It operates by repeatedly flipping individual bits in the $N \times D$ matrix of bit vectors, provided that those flips lead to immediate

improvements in the overall cost function defined above. The bit that is flipped is always the one that leads to the greatest immediate improvement in the cost, hence the name *greedy*.

The Clouse and Cottrell (1996) algorithm differed in that it selected bits at random and then tested to see if flipping the selected bit would decrease the cost. This is fairly inefficient near the end of the minimization process where there are very few bits worth flipping.

The key to performing the minimization quickly is to keep track, at all times, of the change in overall cost that would result from flipping each bit. This is referred to as the bit's *gain*. A positive gain means the overall cost would be reduced by changing the bit. All bits with positive gains are stored in an *implicit heap* (Williams, 1964). This standard priority queue data structure allows the bit with the highest gain to be accessed in constant time.

Whenever the gain of a bit is changed (because it or another bit is flipped), the heap must be adjusted. In theory this adjustment takes $O(\log(ND))$ time. However, the $O(\log(ND))$ bound is very loose. Because the gain changes tend to be small, heap updates rarely involve more than a few steps through the heap. Furthermore, because only bits with positive gains are maintained in the heap and the vast majority of bits have negative gains, especially toward the end of the process, there are usually far fewer than ND bits actually in the heap. Therefore, the gain update step is, in practice, quite close to a constant-time operation.

Along with the gain heap, we also maintain the target distance, t_{ij}, and the current distance, d_{ij}, for each pair of vectors. If the actual distance is at least 1 less than the target distance, we would like to make the two vectors more different. Therefore, for any dimension k, if bits i_k and j_k are the same, the overall cost would be reduced by 1 if we flipped either of those bits. If i_k and j_k are different, the cost would increase by 1 if we flipped either of those bits.

Likewise, if the actual distance is at least 1 larger than the target distance, the contribution of these two vectors to the overall cost will be reduced by 1 if we flip any bit that makes them more similar and increase by 1 if we flip any bit that makes them more different. If flipping a bit causes d_{ij} to change from larger than t_{ij} to smaller than t_{ij}, the change in cost will be $1 - 2(d_{ij} - t_{ij})$. Likewise, if d_{ij} grows larger than t_{ij} with a bit flip, the change in cost is $1 - 2(t_{ij} - d_{ij})$.

Of course, the gain for flipping a particular bit for item i is not dependent on just one other item, j. It is summed over all other items. When any bit is flipped, the gains for some other bits must be adjusted. One factor that improves the efficiency of the GBF algorithm is that we do not need to update the gain for every other bit. As long as we are using the linear cost model, the gain for most other bits remains unchanged.

If the bit i_k has just been flipped, we will definitely need to update the gain for the other bits in vector i. We will also need to update bit k for all of the other vectors. However, we don't necessarily need to update the other bits for the other vectors. We only need to do so if $|t_{ij} - d_{ij}| < 1$, either before or after the flip.

6.3. Running time

Each of the bit updates takes constant time (assuming a constant heap update). So the cost of flipping a bit is somewhere between $\Theta(N + D)$ and $\Theta(ND)$, depending on how many other bits must be updated. In practice, all the bits of a vector are usually updated roughly 1/3 of the time. However, most of these cases occur toward the end of the minimization as the actual distances grow close to the target distances. Therefore, the bulk of the minimization occurs in a relatively short time and, if time were a factor, the minimization could be stopped well before it is complete without significant degradation in the resulting vectors.

Unfortunately, as with the gradient descent methods, it isn't possible to predict exactly how long the minimization process will take. In theory, there could be an exponentially large number of flips before the algorithm terminates. One could, of course, terminate early once a minimum gain, a minimum number of bits in the gain heap, or a time limit has been reached. Or one could test the bit vectors periodically and stop when significant further progress seems unlikely. However, in practice the algorithm tends to terminate on its own in a consistent number of flips, varying by at most a few percent between trials.

6.4. Variants

One concern in performing greedy minimization, always flipping the bit that provides the greatest immediate gain, is that the algorithm may be more likely to fall into bad local minima. A better option may be to select random bits from the gain heap as did Clouse and Cottrell (1996), effectively. In practice, performing random optimization can lead to very slightly better solutions than greedy optimization on most, but not all, tasks. However, it tends to require about twice as many flips because they are, on average, less effective.

Another possibility is to start with a completely random initial configuration, rather than one produced by the random projection technique. Again, an optimization starting from a random initial configuration tends to take about twice as long. When D is large, there is little or no difference in performance. Interestingly, when D is small, starting from a random initial configuration leads to significantly better results on the Exemplar task but much worse results on the Word task.

An additional thought is that, rather than flipping bits one at a time, several bits could be flipped at once. This would make the algorithm more like a discrete version of the gradient descent methods, in which all vector components are updated simultaneously. Perhaps flipping several bits at once would add noise that could help propel the minimization past local minima.

Several variants of this idea were tested. In the first, a random subset of bits in the gain heap were flipped simultaneously. Each bit in the heap was chosen with a probability that ranged from 5% to 25% across trials. Once there were less than about 10 bits in the heap, it was necessary to revert to the single-bit method or the minimization would never terminate. A second variant flipped the n bits with highest gain, where n was a specified fraction of the total number of bits with positive gain. Unfortunately, these methods produced very similar results to the single-bit method, but were somewhat slower.

7. GBFS: THE GREEDY BIT-FLIP WITH SQUARED COST METHOD

GNFS is identical to GBF, except that a squared cost function is used, as recommended by Clouse and Cottrell (1996). This provides a greater penalty for vector pairs that are very far from their target distances. The disadvantage of the squared cost function is that we cannot assume independence when updating bit gains. Thus, when a bit is flipped we must update the gains for all ND bits, slowing the algorithm considerably.

If one wishes to run the algorithm until a local minimum is reached, this method will be more efficient than the Clouse and Cottrell (1996) technique because the latter suffers from inefficiency when few bad bits remain. However, if the algorithm is to be terminated well short of convergence, the Clouse and Cottrell (1996) method will most likely be faster because it avoids the overhead of maintaining the gain heap.

8. GMC: THE GREEDY MAX CUT METHOD

The final algorithm, known as the greedy max cut (GMC) method, also operates primarily in bit-space. It starts with an empty $N \times D$ matrix of bit vectors and iterates through the columns of bits, choosing the value of the first bit for each vector, then the second bit for each vector, and so on. Once all of the bits have been chosen, it iterates through the columns of the matrix again several times, adjusting the bits where necessary.

Aside from the difference in initialization, this algorithm can be distinguished from GBF and GBFS, and the earlier method of (Clouse & Cottrell, 1996), primarily in how it identifies bits that need to be flipped. Rather than selecting bits greedily or at random, the GMC algorithm systematically tests each bit in the matrix. This requires simpler record keeping than GBF, as we no longer need to maintain the gain of each bit. Like the Clouse and Cottrell method, GMC only maintains the current Hamming distances and target distances between all pairs of vectors.

As in GBFS, the cost function minimized in this algorithm is the sum-squared difference between the actual Hamming distances, d_{ij}, and the correlation distances of the original vectors, scaled by D:

$$cost = C = \sum_{i<j} (d_{ij} - t_{ij})^2$$

8.1. Filling the columns

At all times, the algorithm maintains the current set of Hamming distances between vector pairs. It begins with vectors of all 0s. It then cycles through the D columns in the bit-vector matrix, filling each column so as to minimize the overall cost.

This is called the "greedy max cut" method because the process of filling the columns chooses each bit so as to greedily reduce the overall cost and is related to the well-known Maximum Cut graph partitioning problem. Consider a complete, undirected graph with N vertices, corresponding to the N items, with the weight of edge ij equal to $1 - 2(t_{ij} - d_{ij})$. The problem of finding the assignment of bits to column k that minimizes the squared cost is equivalent to finding the partitioning of the graph that maximizes the weight on edges crossing the partition. The items on the same side of the partition receive the same value

for bit k. This is the Maximum Cut problem, which is known to be NP-complete (Karp, 1972).[3]

Therefore, it seems likely that no algorithm exists to produce an optimal solution to the problem in polynomial time. However, several fast approximation algorithms are known that will produce solutions to the Maximum Cut problem guaranteed to be within a certain percentage of the optimal value. The earliest such approximation algorithm is that of Sahni and Gonzales (1976), which guaranteed that the solution found would be at least half of the optimal value. The method employed here, in its first pass, is quite similar to the Sahni and Gonzales algorithm.

When filling column k, the first item receives a random bit. Each subsequent item is given the bit value that results in a lower overall cost, computed over the preceding items. The contribution to the overall cost of item i which results from selecting the value $i_k = 0$ will be:

$$C^0_{i_k} = \sum_{j<i, j_k=0} (d_{ij} - t_{ij})^2 + \sum_{j<i, j_k=1} (d_{ij} + 1 - t_{ij})^2$$

where j iterates over all items for which bit k has been chosen. The first summation includes only those items for which bit j_k is 0 and the second summation includes only those items for which bit j_k is 1. Thus, if $i_k = 0$, the distance to items for which $j_k = 1$ will increase by one.

Similarly, the cost for choosing $i_k = 1$ will be:

$$C^1_{i_k} = \sum_{j<i, j_k=0} (d_{ij} + 1 - t_{ij})^2 + \sum_{j<i, j_k=1} (d_{ij} - t_{ij})^2$$

If $C^0_{i_k} < C^1_{i_k}$, i_k is set to 0, otherwise to 1. In practice, it would be inefficient to compute those entire expressions. It is better just to compute their difference, which is given by:

$$C^0_{i_k} - C^1_{i_k} = \sum_{j<i} (2j_k - 1)(2(d_{ij} - t_{ij}) + 1)$$

8.2. Adjusting the columns

However, this single-pass assignment of the bits in a column can only achieve a rough approximation to the optimal partitioning. It can be further refined by iterating through the items and flipping their bits when doing so results in lowered cost. In this adjustment phase, the bits are not cleared in advance and the cost function for item i is computed over *all* other items, not just the preceding items. Thus, the value of each bit in the column is reconsidered given the values of the other bits. Subsequent adjustments will further refine the assignment, but the number of bits that are changed each time gradually decreases. It is useful to perform at least two of these *primary adjustments* to each column before filling the next column.

[3]Technically, our bit assignment problem does not exactly reduce to Maximum Cut because the latter normally only permits positive edge weights, whereas our graph has both positive and negative weights. Although this form of reduction doesn't prove that the bit assignment problem is NP-hard, it is suggestive of the difficulty of the problem.

Once all of the columns are filled, it is helpful to cycle through them several more times, re-adjusting the bits in each one when the cost improves. These are known as *secondary adjustments*. To clarify, the primary adjustments occur to each column before the next column is filled. The secondary adjustments occur once all of the columns have been filled. In evaluating this algorithm, 2 primary adjustments and 8 secondary adjustments were used.

8.3. Running time

Unlike the gradient descent or bit flipping methods, the running time of this algorithm is easily predicted. Filling or adjusting each column requires $\Theta(N^2)$ operations. If the total number of primary and secondary adjustments is a, the running time of the algorithm is $\Theta(N^2D(a+1))$.

8.4. Variants

The obvious parameters affecting this method are the number of primary and secondary adjustments. The first few adjustments result in significant improvement, but further adjustments have greatly diminishing returns. There is a tradeoff in balancing the number of primary and secondary adjustments. One could rely on all-primary or all-secondary adjustments, but performance is better with some of each. Holding the total number of adjustments fixed, it is generally best to use 2 or 3 primary adjustments and a greater number of secondary ones.

As described, the GMC algorithm is completely deterministic. Multiple runs on the same set of items will result in effectively the same vectors, although some columns will have all of their bits reversed because the first bit in each column was selected randomly. It seems plausible that this method could tend to hit local minima because the bits are always updated in the same order in the adjustment phases. A reasonable variant would be to adjust the columns in randomly permuted order. However, experiments with this on the Word task found equivalent or slightly worse performance than the simpler, deterministic method.

Another possibility is to alter the order of traversal in the secondary adjustment phase. Rather than traversing the columns, one might traverse the rows. This has the effect of adjusting a single point relative to all of the other points before moving the next point. In contrast, column traversal adjusts all points along a single dimension before considering the next dimension. One might expect these two adjustment methods to produce different results, but in practice they seem to result in virtually identical performance.

An alternative is to select bits for possible adjustment at random, as in the Clouse and Cottrell (1996) algorithm. Again, one might expect this to better avoid local minima. However, equating for the number of bits tested, random flipping has proved to be slightly, but not much, worse than either of the systematic updating methods. Thus, in its secondary phase, GMC doesn't differ significantly from the earlier method. The most important difference between the algorithms is the method of initializing the bit matrix. Replacing the primary bit-assignment phase of GMC with a random assignment of bits, but still equating for the total number of bits tested, results in significantly worse performance.

Replacing it with a random projection, as in GBF, also degrades performance, but less so than random initialization.

This algorithm can be easily adapted to any cost function that has the form of a sum over independent contributions from each item pair. It could efficiently handle the stress measure if the numerator and denominator were stored and updated incrementally. It would be more difficult to adapt the algorithm to a non-metric cost function. One would need a fast, incremental version of the up-down algorithm to maintain the optimal monotonic transformation of the vector distances as bits are flipped. When amortized over multiple flips, this may still be more than a constant time operation and would thus add to the asymptotic complexity.

Finally, as in the GBF and GBFS methods, it is easy to add a term to the cost function that severely penalizes duplicate vectors and thus ensures that each item has a unique representation, which is sometimes necessary.

9. PERFORMANCE ANALYSIS

In this section, we present comparisons of the six implemented algorithms: SVD, MGD, OGD, GBF, GBFS, and GMC. The methods were applied to the Exemplar and Word tasks with varying bit-vector dimensionalities, D, and were evaluated using three measures of the agreement between the original pairwise distances and the final bit-vector Hamming distances: goodness, metric stress, and non-metric stress. The average running times of the algorithms were also measured and are reported in Section 9.4.

Three trials of each condition were run on a 500MHz 21264 Alpha processor and the results averaged. The only exceptions were the few trials of SVD and GBFS that lasted more than 10 hours, which were only run once. In general, the results of the methods were all very consistent across trials. The SVD and GMC algorithms are deterministic, and thus achieve the same results every time. The other algorithms achieve goodness or stress ratings that vary about 1% between trials for small values of D and less than 0.2% for $D >= 100$. Because of the small variance, limited number of trials, and general clutter of the figures, error bars are not shown.

9.1. The exemplar task

The average goodness ratings of the six algorithms on the Exemplar task, as a function of D, are shown in Figure 1. With the exception of SVD, the goodness increases monotonically with the size of the vectors. SVD is clearly the worst of the methods. Interestingly, in that case the goodness actually decreases with D values over 20. This is presumably because, with larger D, the SVD method begins to rely on less important singular vectors which convey little useful information.

There is no clear winner between OGD and MGD. OGD may be better for lower D but worse for higher D. The three best algorithms, according to goodness, are the bit-space optimizing methods: GBF, GBFS, and GMC. GBF does not do as well for small D. With the exception of $D = 10$, GMC achieves the best goodness in every case, although the differences with GBF and GBFS are inconsequential for large D.

The metric and non-metric stress on the Exemplar task are shown in Figures 2 and 3, respectively. SVD does so poorly according to metric stress (from 18.6 for $D = 10$ to 0.34

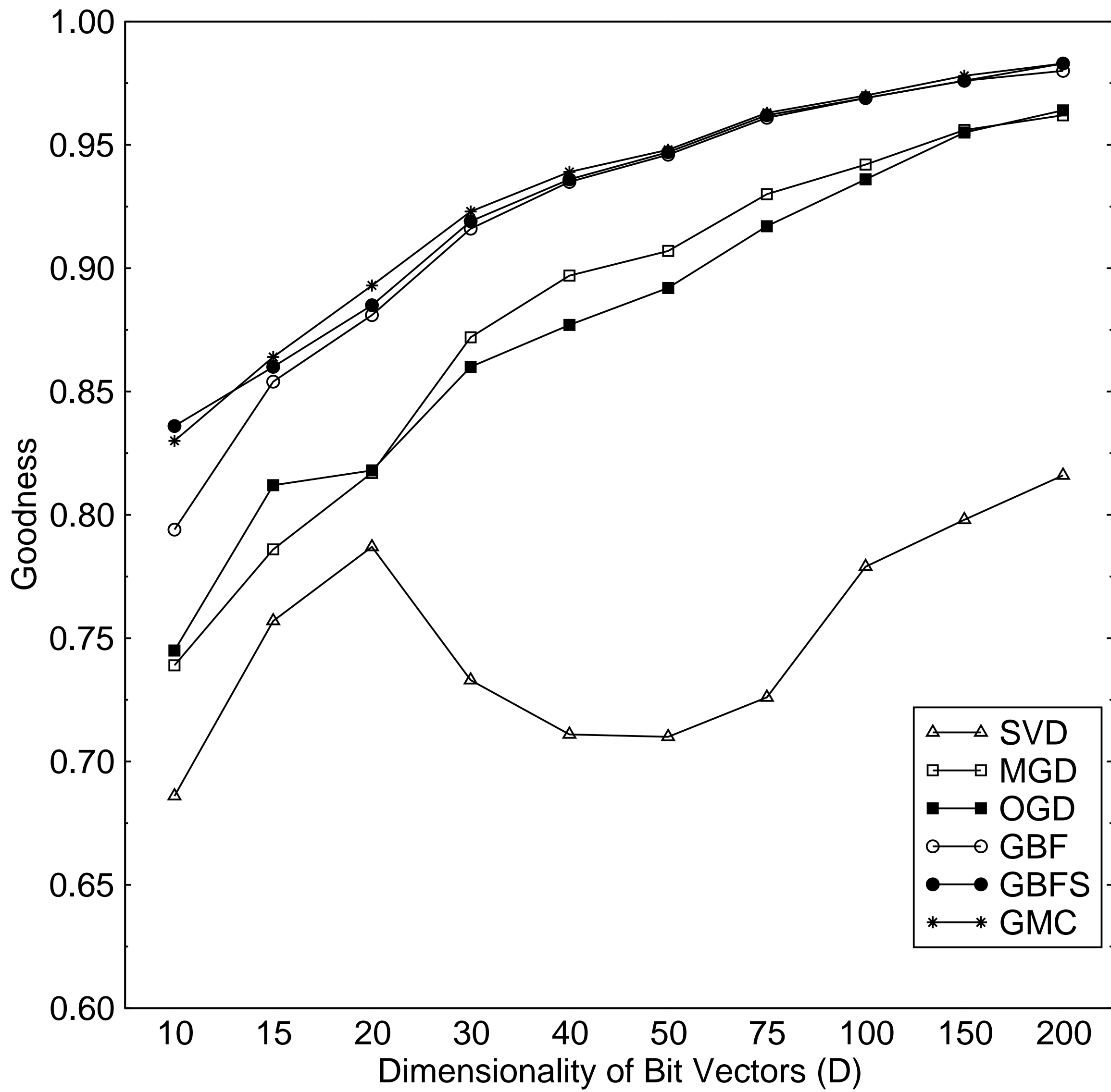

Figure 1. Goodness on the Exemplar task.

for $D = 200$) that it doesn't appear on the graph. SVD also does quite poorly according to non-metric stress, although it is at least comparable to the other methods. Interestingly, although it did not according to goodness, the performance of SVD monotonically improves with D according to the stress measures.

MGD is mediocre according to both measures. OGD does rather poorly according to metric stress. This should not be too surprising since OGD is only optimizing non-metric stress and its resulting pairwise distances are unlikely to directly match the target distances. However, OGD is still not as good as the bit-space methods even on non-metric stress. GBFS and GMC are fairly indistinguishable according to either measure, although, with the exception of $D = 10$, GMC is slightly better in all cases. GBF is worse than the other two for $D = 10$, but is nearly as good at higher dimensionality.

9.2. The word task

The Word task has a much more complex similarity structure than the artificial Exemplar task, and thus may provide a better measure of the ability of the BMDS methods on other scaling problems involving natural data. Because it has 5,000 items and the original vectors have 4,000 dimensions rather 1,000, the Word task is computationally harder as well. The goodness measure is shown in Figure 4 and the stress measures are depicted in Figures 5 and 6.

Once again, SVD does quite poorly. It makes little improvement in goodness with higher dimensionality. It is off of the metric stress scale, except for the case of $D = 200$, and its non-metric stress is much worse than that of the other measures. Again, however, it does make steady improvement with higher D according to the stress measures, but not according to goodness.

OGD is the clear winner on the goodness scale, especially for small D. This is surprising since it performed quite poorly on the Exemplar task. According to metric stress, OGD again doesn't do very well, but it is the best measure by non-metric stress for $D <= 50$. Nevertheless, based on non-metric stress, it is not clearly dominant over the other measures as it is on the goodness scale. The next section addresses this apparent disparity between goodness and the stress measures.

As measured by goodness, MGD does quite well but is not close to the performance of OGD. According to stress, MGD is just average. GMC and GBFS have quite similar performance, but GMC is consistently a little bit better on all three measures at all values of D. GBF is reasonably good at high values of D, but its performance on all measures drops off with small D.

9.3. Stress vs. goodness

It is interesting that the stress and goodness measures do not always agree. At times, one method will perform better than another according to goodness but worse according to stress. For example, on the Word task with 50 bits, OGD achieves an average goodness of 0.843 while the GMC vectors only have a goodness of 0.741. However, the GMC vectors have a metric stress of 0.109, which is better than the 0.133 of the OGD vectors. According to non-metric stress, GMC and OGD are very similar (0.104 vs. 0.102). Because the measures

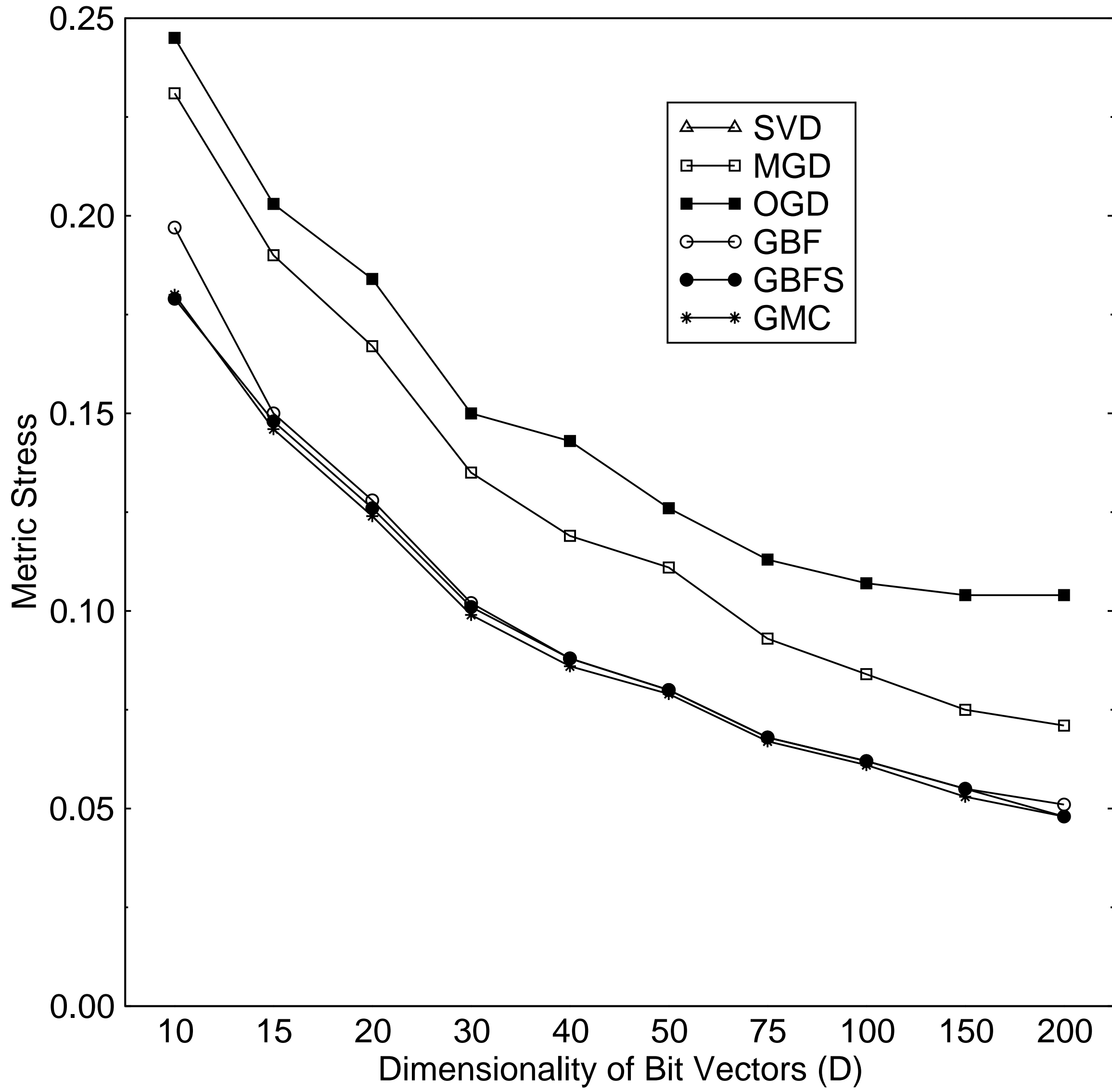

Figure 2. Metric stress on the Exemplar task.

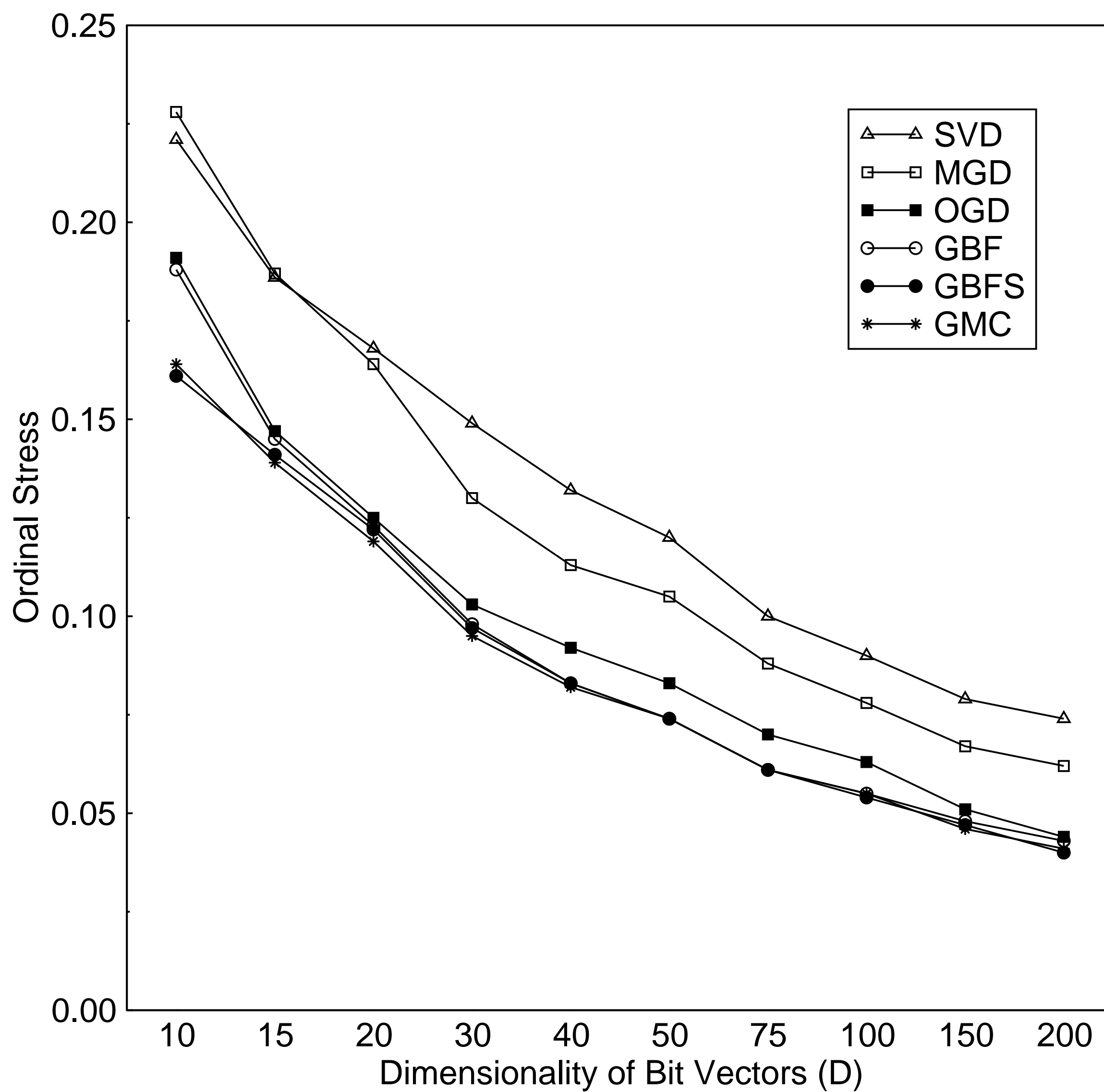

Figure 3. Non-metric stress on the Exemplar task.

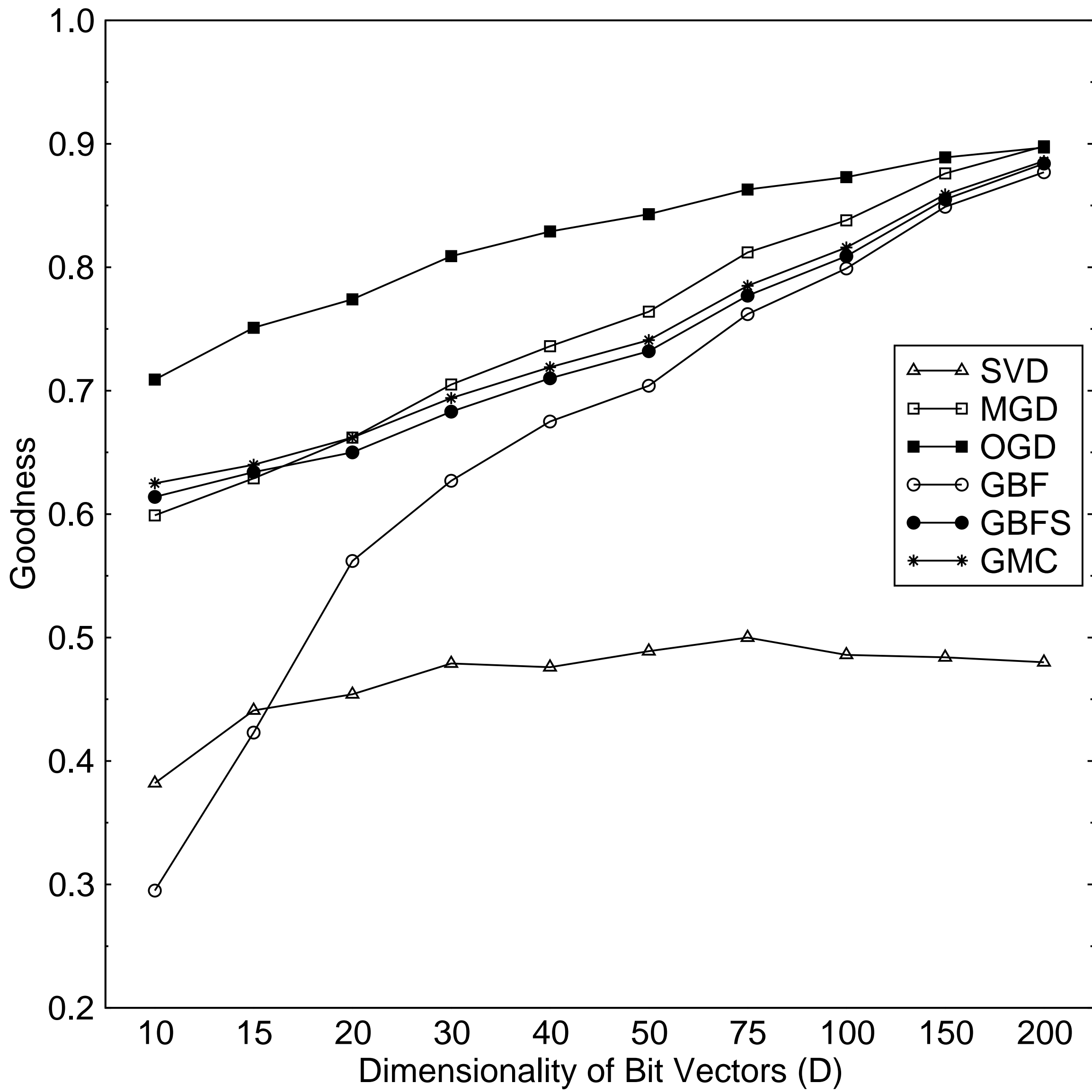

Figure 4. Goodness on the Word task.

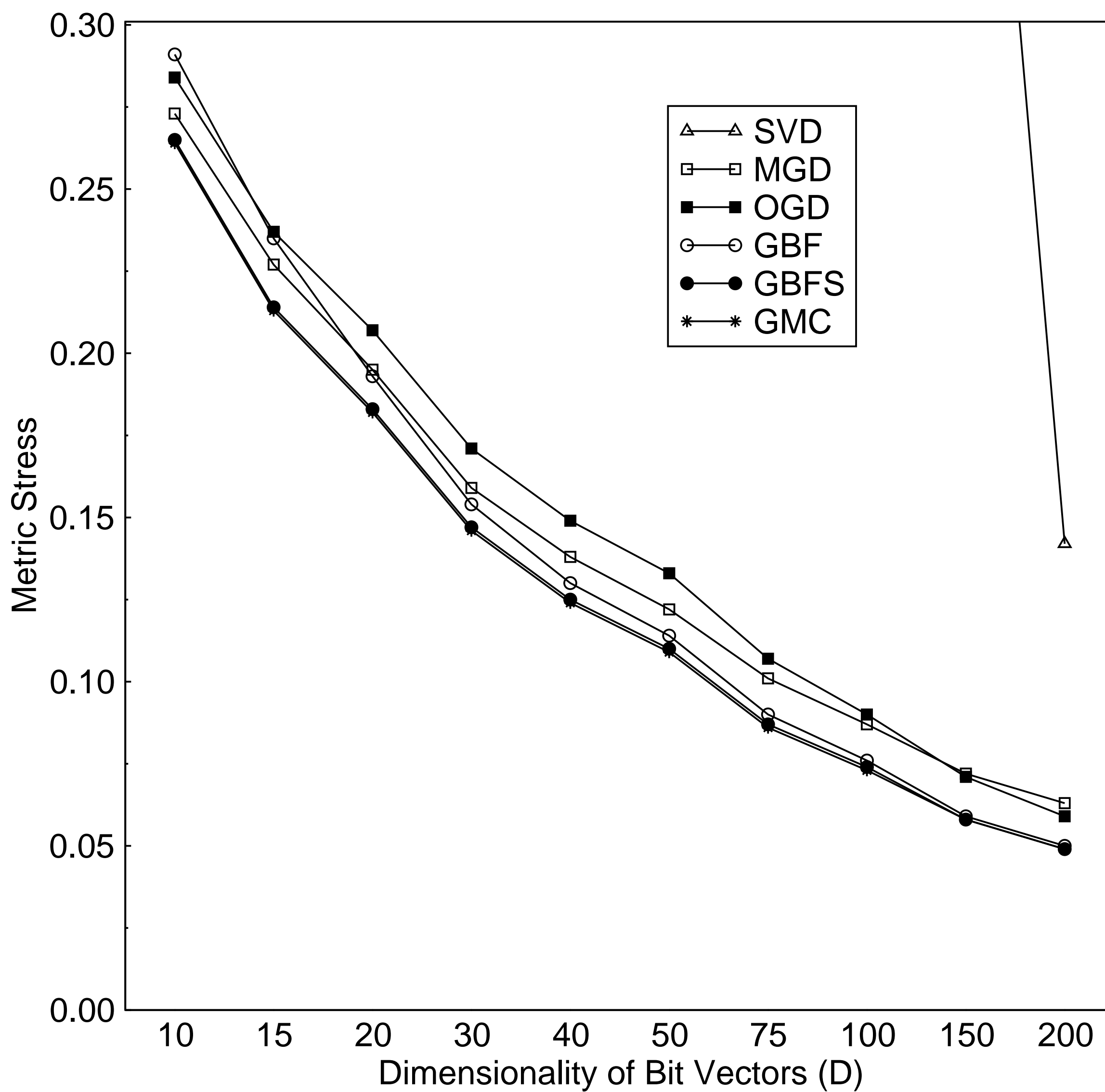

Figure 5. Metric stress on the Word task.

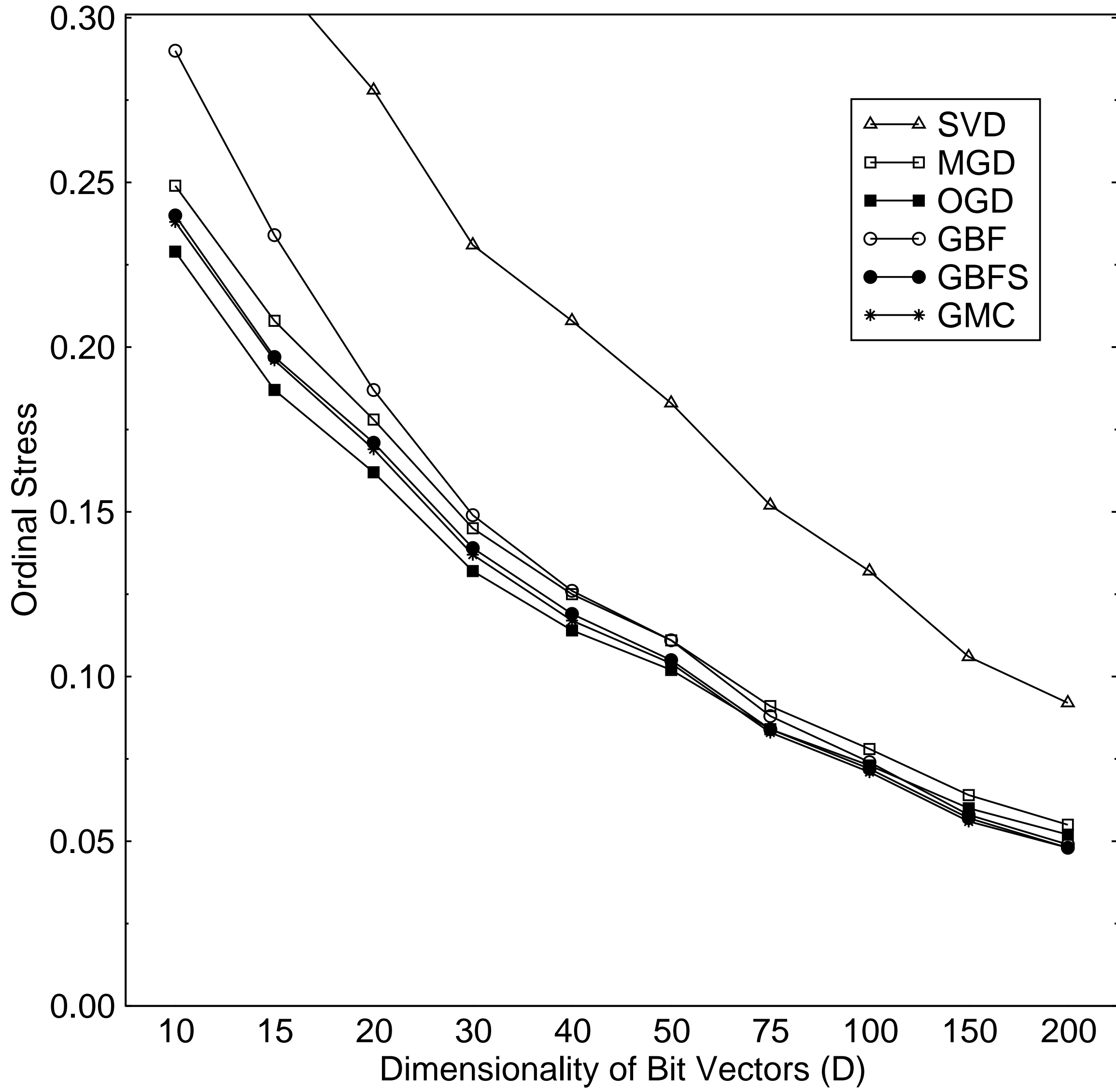

Figure 6. Non-metric stress on the Word task.

don't always agree, it is important to choose an evaluation measure that is appropriate for a given task. So let's briefly compare the properties of these measures.

We begin by trying to understand what aspects of the GMC and OGD solutions might have lead to the disagreement among evaluation measures. One common way to visually evaluate MDS results is through the use of a *Shepard diagram*. This is a scatter plot with one point for every pair of items. The horizontal axis represents the distance between the original pair of vectors, or subjects' similarity ratings if that is the starting point. The vertical axis represents the actual distance between the vectors in the reduced space. Ideally, the points should fall on the identity line.

However, because the Word task involves 5,000 items, a standard Shepard plot would contain 12.5 million points. So many points are hard to handle and estimating their density is difficult. Therefore, a modified version of a Shepard plot is used here which is something like a 2D histogram. The graph is partitioned into a grid of cells and the number of points falling into each cell is counted. Then the columns of cells are normalized so that the values in each column sum to 1. Each cell is then plotted as a circle whose area is proportional to the normalized cell value. The ideal graph will be linear and have little vertical spread. These "Shepard-histograms" for one run of the GMC and OGD algorithms are shown in Figures 7 and 8.

It should be noted that this method of normalizing columns of cells disguises the fact that the vast majority of pairs have target distances close to 25, indicating that their original vectors were uncorrelated. Therefore, most of the datapoints in the graph actually fall in this middle range. This can be seen by performing the normalization across all cells, not just along the individual columns. Figure 9 displays the same data as Figure 8, but with normalization across all cells. The cells representing very small or large correlation distances are barely visible because they contain so few points.

Figure 7 shows that the GMC algorithm finds a fairly linear solution. However, the solution appears to have a negative zero-crossing and is somewhat warped for small target distances. Thus, vectors that were originally quite similar tend to be even more similar in the bit-space. There is also greater variance for the columns on the left, although this partially results from the fact that they contain very few points.

The plot of the OGD solution in Figure 8 is noticeably less linear, having a more pronounced sigmoidal shape. Like the GMC solution, small distances tend to be too-small in the final space. However, in this case, large distances tend to be exaggerated as well. This partially explains why OGD does better according to goodness, or correlation, worse according to metric stress and almost equivalently according to non-metric stress.

Non-metric stress is essentially indicative of the variance of the columns in the Shepard diagram, but is insensitive to the mean value in each column. In this case, the two methods have fairly similar variance, resulting in similar non-metric stress. Metric stress, on the other hand, measures the disparity between the points and the identity function. Any deviation from this line, whether for large or small target distances, contributes equally to the stress. Metric stress is sensitive to both noise and monotonic distortions, the latter having a relatively strong effect. Thus, the OGD solution has a higher metric stress.

The correlation measure is a bit harder to characterize analytically. But some simple empirical tests involving a few artificial datasets show that correlation is actually fairly insensitive to monotonic distortions. First, a set of random numbers evenly distributed

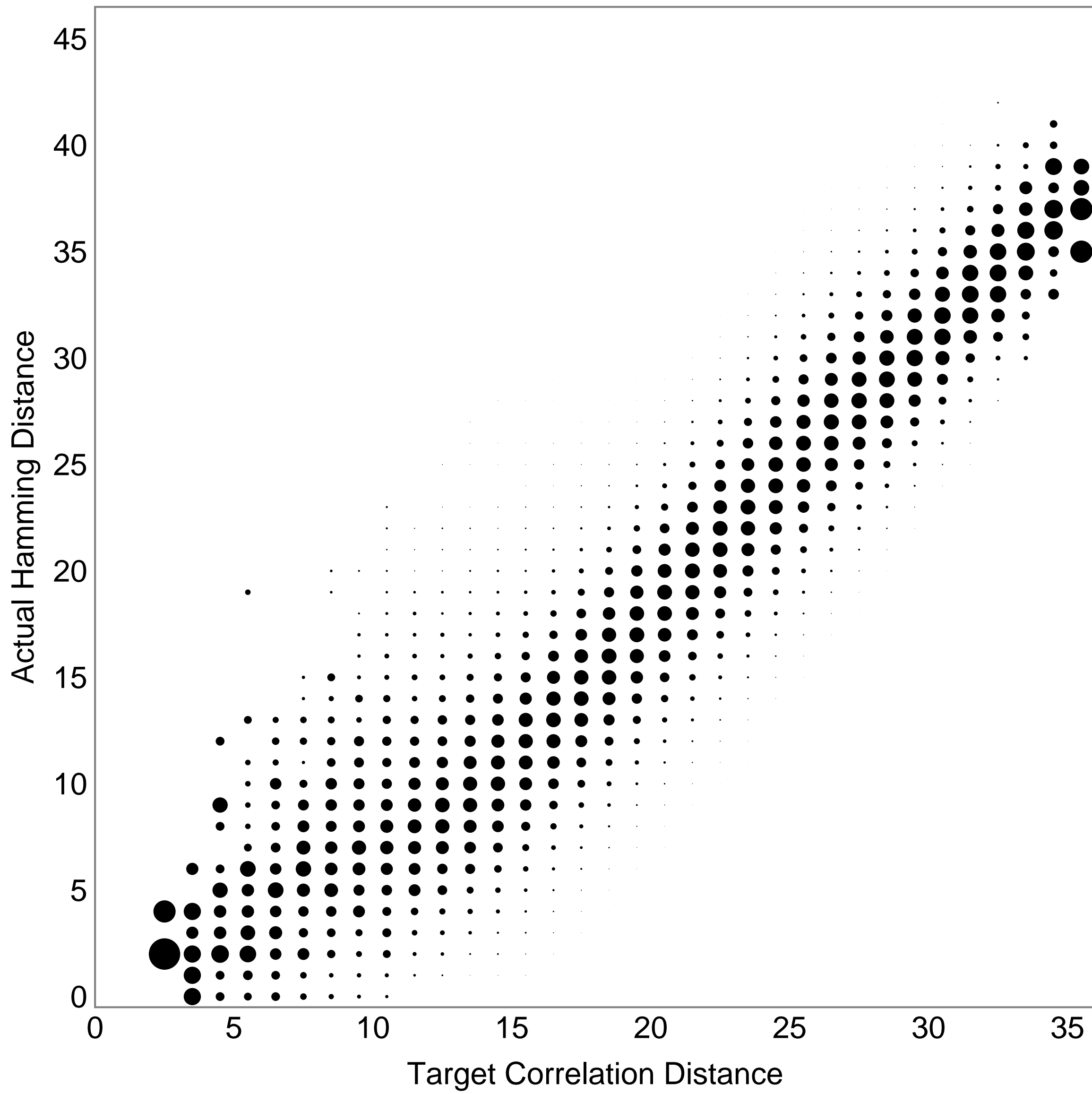

Figure 7. Distribution of pairwise bit vector Hamming distances versus original distances for a run of the GMC method on the Word task with 50 bits. Cells are normalized by columns.

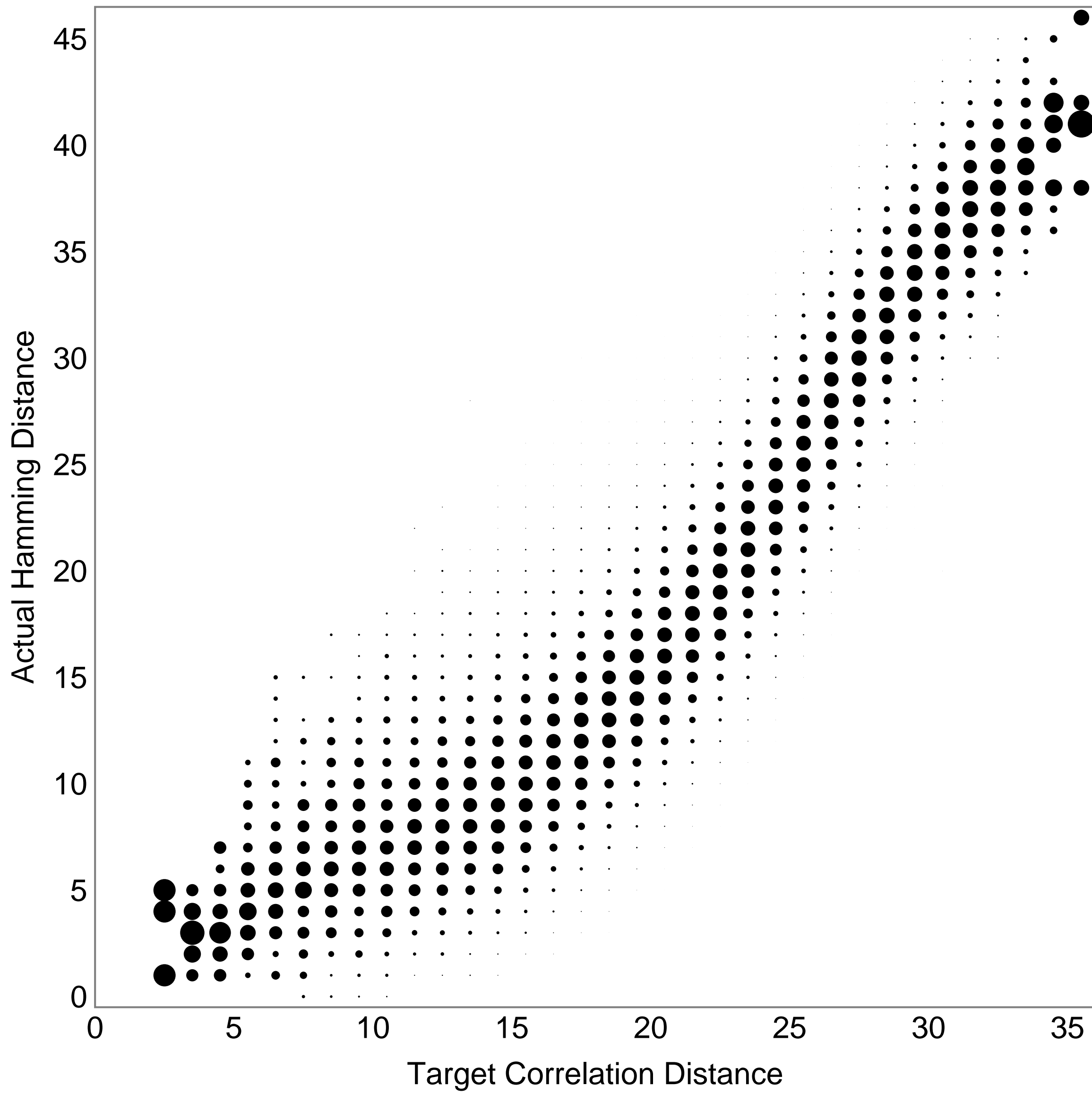

Figure 8. Distribution of pairwise bit vector Hamming distances versus original distances for a run of the OGD method on the Word task with 50 bits. Cells are normalized by columns.

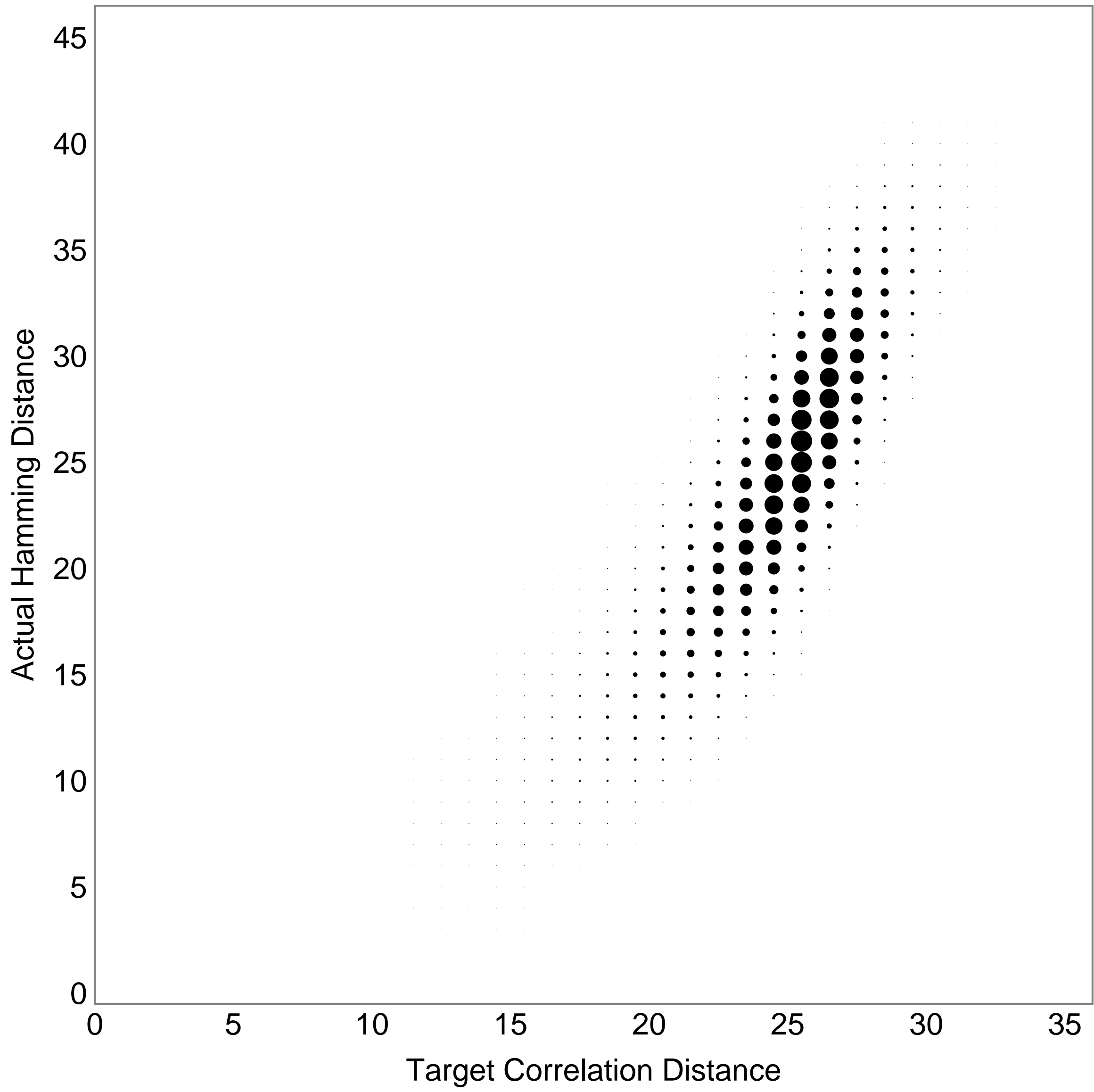

Figure 9. The same data as in Figure 8 normalized across all cells, rather than by columns.

between 0 and 1 were generated. A second set of numbers was produced by transforming each value in the first set and the correlation and stress of the number pairs was measured.

Monotonic distortions have a greater effect on stress. If the second set is composed of the square roots of the first set, there is little effect on the correlation, which is 0.98, but the stress rises to 0.098. If cube roots are used, the correlation is still fairly high, 0.959, but the stress is 0.229. Finally, if a sigmoid centered around 0.5 with a gain of 2 is applied to the numbers to create an S-shaped transformation as in Figure 8, the stress is moderate, 0.067, but the correlation is virtually unchanged at 0.9998.

In contrast, adding noise to the values has a larger effect on correlation than on stress. If 0.15 is either added to or subtracted from each value with equal chance, the correlation drops to 0.887 but the stress is again 0.067. Thus, in this case of noise, the stress is equal to or lower than it was with the monotonic transformations, but the correlation is much worse. The implication of this is that the correlation measure is fairly ordinal in its behavior. Indeed, measuring the correlation on the example tasks using monotonically transformed targets, $\hat{d}_{ij}$, rather than the actual targets, t_{ij}, generally results in only a slight improvement.

9.4. Running time

Finally, we turn to the issue of the running time of the various algorithms. Regardless of how good the results may be, an algorithm is only useful if it can solve a given task in an acceptable time frame. The running times of SVD and GMC are fairly easy to analyze because they're deterministic. The method used here to compute the SVD is $\Theta(N^2(N+M)+ND)$. The ND term is for assigning the bits and is inconsequential. If $M < N$, the matrix is inverted, resulting in a $\Theta(M^2(M+N))$ algorithm.

The other algorithms require that all pairwise distances between the vectors are computed, which is a $\Theta(N^2M)$ process. However, because that step is common to all of them, it was done in advance and the distances stored. It is therefore not shown in the measured running times. It is worth noting that, although they are both $\Theta(N^2M)$, computing the vector distances is much quicker than computing the SVD due to the much improved constant.

Following the distance computation, the GMC algorithm is $\Theta(N^2D)$, assuming the number of adjustments is treated as a constant. The gradient descent and bit-flipping algorithms, on the other hand, are difficult to analyze because it isn't clear when they will terminate. They are suspected to be roughly $\Theta(N^2D)$, however, and we turn to some empirical measures to verify this.

Figures 10 and 11 show the scaled running times of the methods as a function of D on the Exemplar and Word tasks. Because the times were expected to be roughly linear in D, they were all divided by D in producing the graph. Therefore, a flat line indicates a truly linear algorithm. SVD, because its running time is essentially constant, has decreasing curves in both graphs. SVD was so slow on the Word task, however, that it only appears on the graph for $D = 200$.

MGD and OGD seem to be fairly linear in D. Both of them are a bit slower for very small D on the Exemplar task. Possibly this is because they had difficulty settling on a good solution with so few bits. OGD was significantly quicker on the Word task, but slower on the Exemplar task. The GBF method appears to be somewhat worse than linear, its

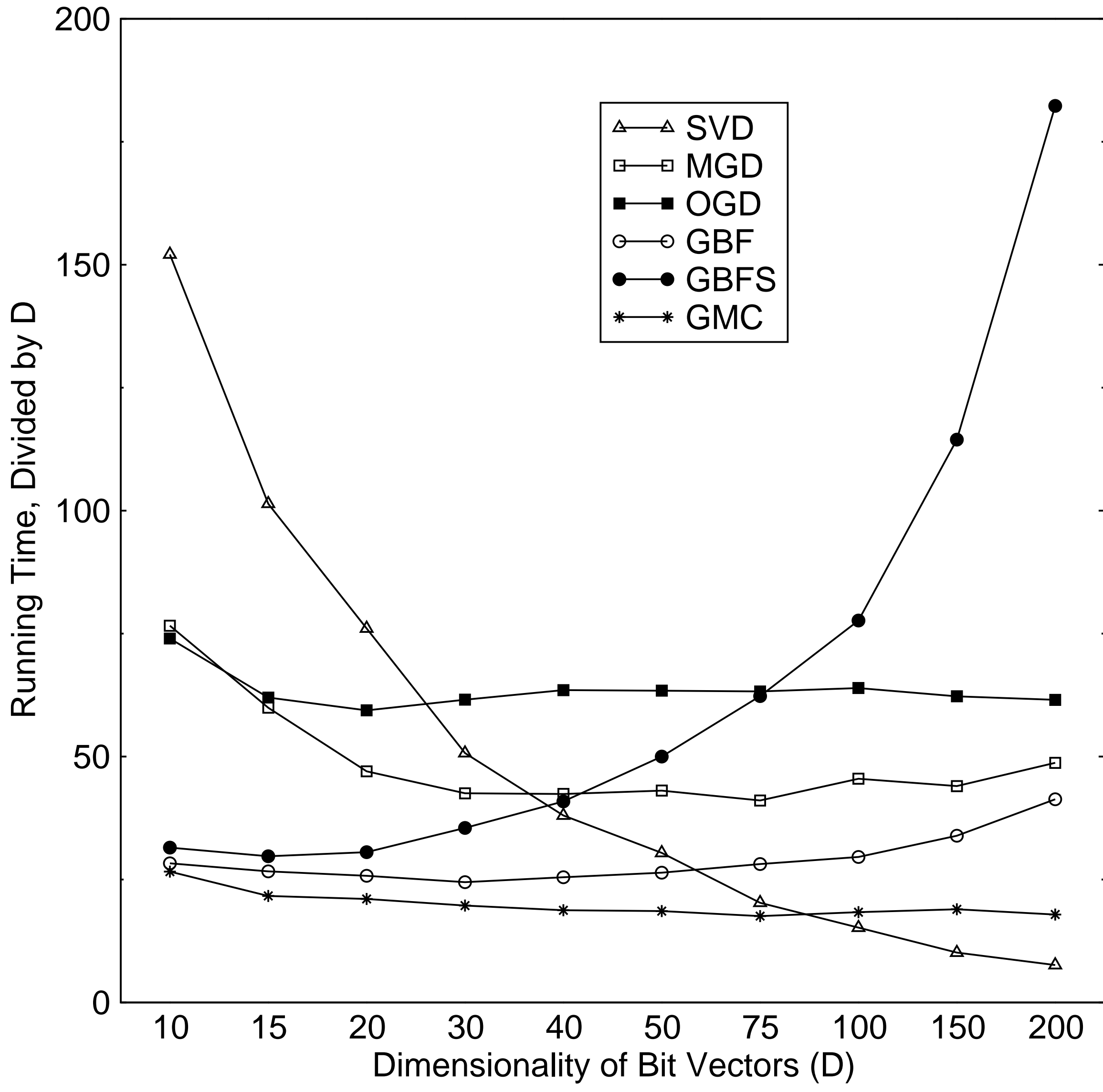

Figure 10. Running time as a function of D on the Exemplar task (scaled by $1/D$).

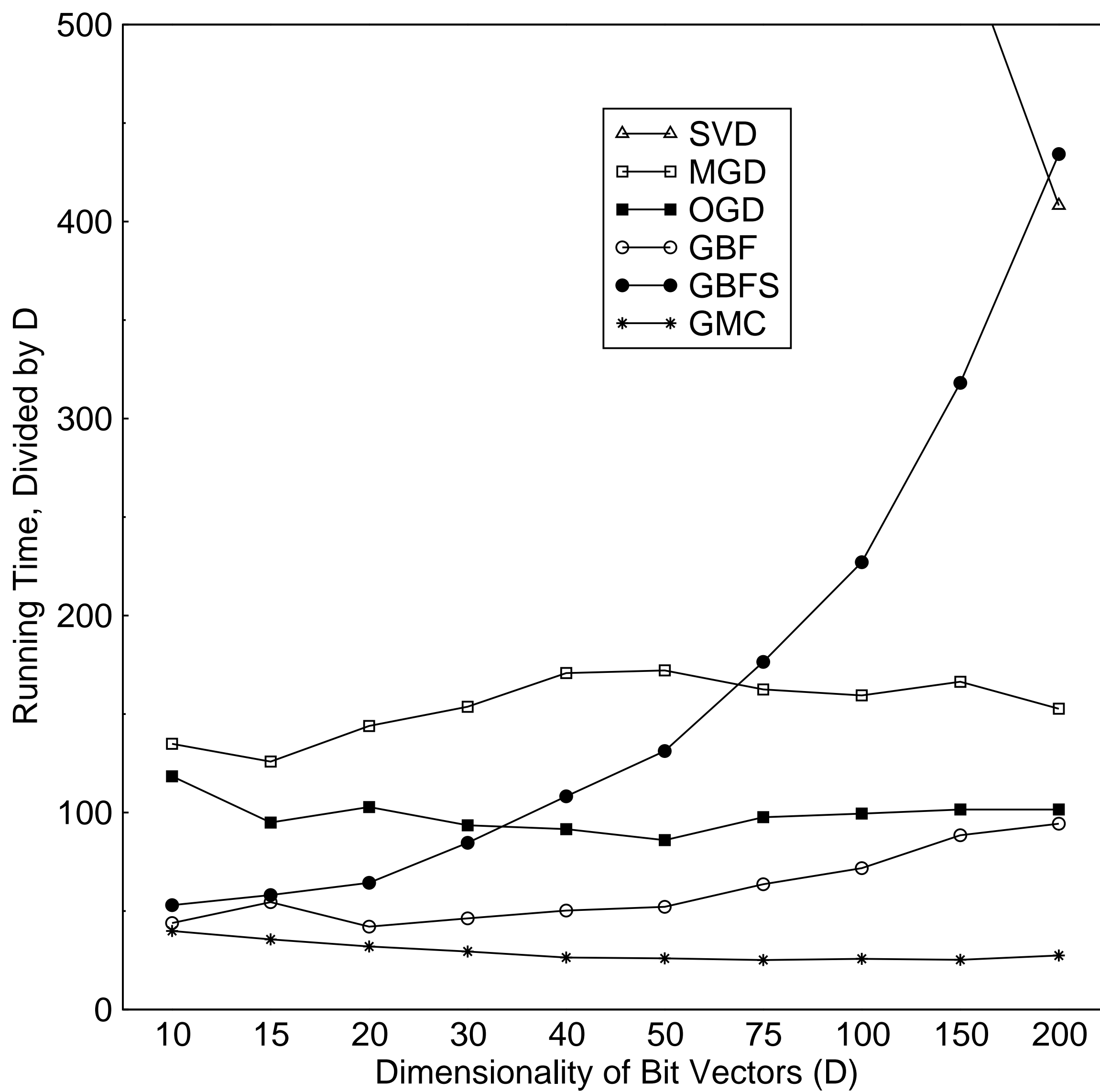

Figure 11. Running time as a function of D on the Word task (scaled by $1/D$).

curves noticeably rising on the right side. GBFS, on the other hand, is essentially quadratic in D.

The algorithm that is consistently fastest, other than the ineffective SVD method, is GMC. On the Word task with 200 bits per vector, GMC is almost three and a half times faster than the next fastest method, GBF. Although GMC is known to be truly linear in D, its scaled running times actually decrease with larger D. This reflects the fact that lower-order terms, such as the N^2 cost of loading the pairwise distances, have a relatively diminishing effect on the overall time. This indicates that the other methods that appeared to be linear due to flat lines may actually be slightly worse.

Figure 12 depicts the running times of the methods for varying numbers of items, N, on the Word task. In this case, the running times have been divided by N^2. GMC is known to be quadratic in N. Therefore, the slight rise in its line is either due to lower-order terms or caching inefficiencies resulting from the increased memory requirements of the larger problems. GBF has a similar profile and is thus nearly quadratic in N as well. GBFS and MGD, on the other hand, are clearly worse than N^2. OGD may be slightly worse than quadratic, but it's not clear. SVD ranged from 4.3 times slower than MGD for $N = 500$ to 9.5 times slower for $N = 5000$.

10. DISCUSSION

Although multidimensional scaling techniques have been studied for over half a century, binary multidimensional scaling, which was inspired by the need to develop representations usable in training neural networks, is a relatively new, yet intriguing, problem. This study has introduced and evaluated several algorithms for performing binary multidimensional scaling. It is hoped that the better methods will prove useful to researchers in their current forms and that the inclusion of the less effective methods in this report will help to direct future attempts to improve on these techniques.

10.1. SVD

Although it is so useful in other types of scaling problems, the SVD method is simply not a good choice for BMDS. It consistently achieved the worst performance. For the Word task, this came at the greatest cost, in terms of running time. Although it is possible that improved discretization methods could achieve better BMDS performance using the SVD, there is still the issue of the running time. Unless either the number or dimensionality of the original vectors is quite small, simply computing the SVD is prohibitively expensive.

10.2. MGD and OGD

The gradient descent methods, which are the most closely related to techniques currently in use in standard MDS, show some promise for BMDS. They have the advantage that they can be used with any differentiable cost function, and are thus extremely flexible. Although they were slower than GBF and GMC on these tasks, most of the improvement in their results occurs early in the gradient descent and the process can be cut short with relatively little effect on performance.

On the Exemplar task, OGD and MGD were not as good as the bit-space methods by any measure. However, they performed very well according to the goodness measure

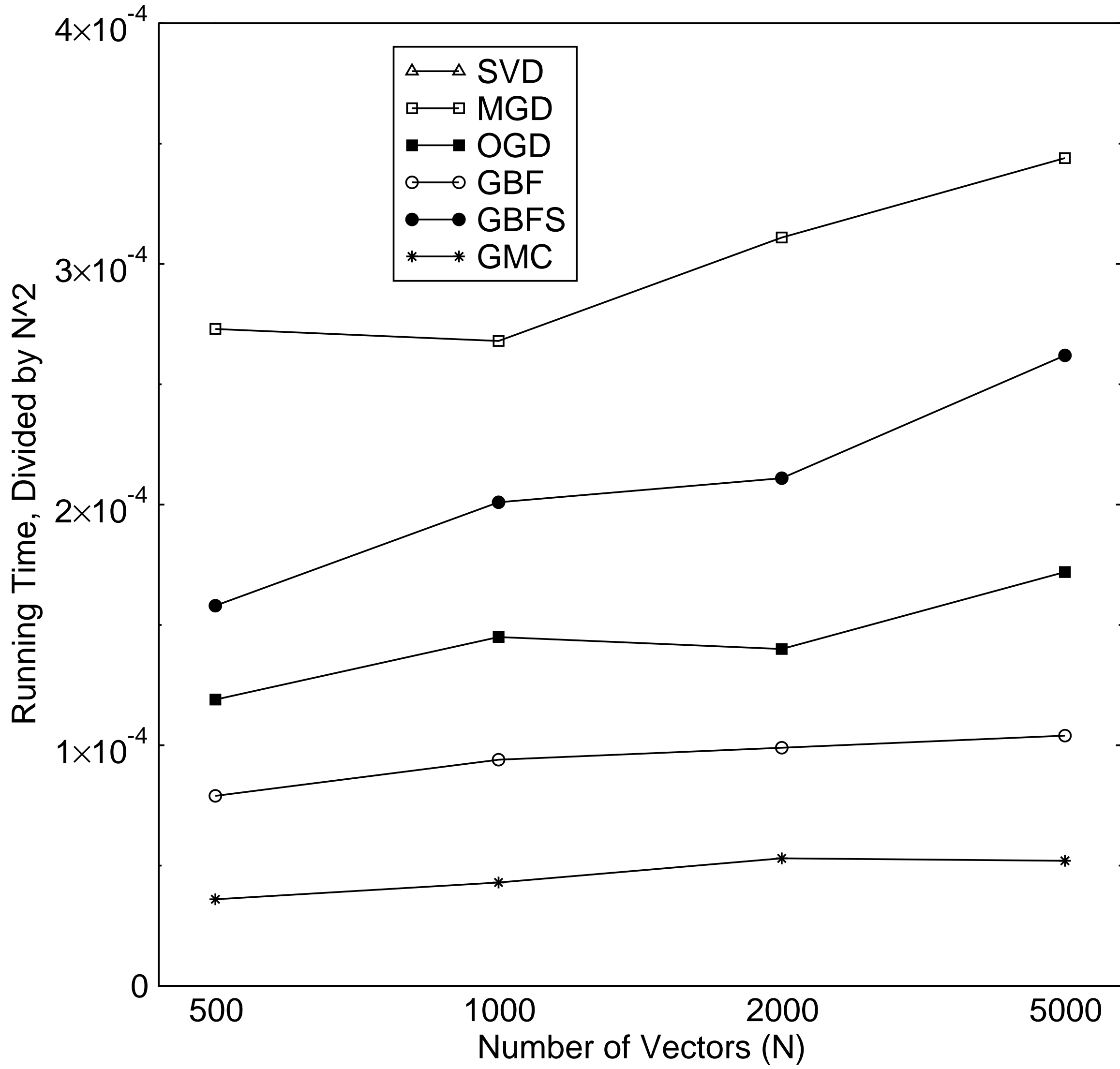

Figure 12. Running time as a function of N on the Word task (scaled by $1/N^2$).

on the Word task, especially OGD. OGD was also the best according to non-metric stress for small D on the Word task. Unless one is concerned with using a non-metric method, OGD seems to be a better choice than MGD. It generally achieves superior performance and also converges more quickly. This is partially due to the fact that it is non-metric and partially due to the use of sigmoidally-transformed values in computing the vector distances. It is true, however, that time to convergence of MGD could be reduced with the use of a polarizing cost function.

Innumerable variants of these methods are possible and it is likely they could both be improved with further work.

10.3. GBF and GBFS

The GBFS method is essentially the one used by Clouse and Cottrell (1996), although the current implementation begins with a random projection and is operated in a greedy fashion rather than by flipping random bits with positive gain. GBFS consistently produces very good solutions. Unfortunately, it suffers from being quadratic in D and more than quadratic in N.

Because it uses a linear cost function, the GBF method is able to cut corners and run much more quickly, with only a modest loss in performance on the Exemplar task. On the Word task, GBF does not do as well, and has particular trouble with small D. Nevertheless, both GBF and GBFS appear to be strictly worse than GMC, in terms of running time as well as performance. Although the speed of both methods could be improved by terminating the optimization early, this would only hurt performance.

10.4. GMC

The GMC algorithm seems to be currently the best overall choice for BMDS. It is the fastest of the algorithms and produces the best or nearly the best results according to the stress measures and also achieved the best goodness scores on the exemplar task. But it should be noted that OGD did achieve better goodness ratings on the Word task, and thus may be preferable in cases where non-metric solutions are acceptable and the similarity structure is relatively complex.

The GMC algorithm has a number of other advantages. Its running time is well understood. Unlike the gradient descent and bit flipping methods, GMC runs for a consistent and predictable amount of time. As with GBF and GBFS, but unlike the gradient descent methods, GMC can be modified to produce only unique vectors by simply adding a term to the cost function. This may be a requirement for some applications of BMDS. For example, it could be problematic if two different words are assigned exactly the same meaning.

GMC can also be used with a variety of cost functions, although it is not quite as flexible as the gradient descent methods in this regard because the cost must be incrementally calculable. Finally, GMC is very easy to implement. Unlike the gradient descent methods, there are no learning rates or other parameters to adjust, nor complex derivatives to compute. Unlike the bit flipping methods, the algorithm is simple and straightforward with minimal record keeping.

10.5. Conclusion

With the exception of SVD and possibly GBFS, the binary multidimensional scaling methods presented here are capable of handling problems of reasonably high complexity. However, even GMC, with a running time of $\Theta(N^2(M + D))$ will not scale up well to problems with hundreds of thousands of items or dimensions. To solve such large problems, more efficient, though perhaps less effective, techniques will be needed. One possibility is to use a limited set of R reference items. All items are positioned relative to those in the reference set, but not necessarily relative to one another. If the dimensionality of the final space is not too large, the reference vectors may sufficiently constrain the positions of the other items relative to one another to produce a good overall solution. Variants of this idea are possible with all of the algorithms presented here, although not always conjointly with the uniqueness constraint offered by the bit-space methods.

Code for any of these methods can be obtained by contacting the author.

REFERENCES

Beatty, M., & Manjunath, B. S. (1997). Dimensionality reduction using multidimensional scaling for image search. In *Proceedings of the IEEE International Conference on Image Processing, Vol. II* (pp. 835–838). Santa Barbara, California.

Berry, M. W., Dumais, S. T., & O'Brien, G. W. (1994). *Using linear algebra for intelligent information retrieval* (Tech. Rep. No. CS-94-270). Knoxville, TN: University of Tennessee.

Borg, I., & Groenen, P. (1997). *Modern multidimensional scaling.* New York: Springer-Verlag.

Burgess, C. (1998). From simple associations to the building block of language: Modeling meaning in memory with the HAL model. *Behavior Research Methods, Instruments, and Computers, 30*, 188-198.

Clouse, D. S. (1998). *Representing lexical semantics with context vectors and modeling lexical access with attractor networks.* Unpublished doctoral dissertation, University of California, San Diego.

Clouse, D. S., & Cottrell, G. W. (1996). Discrete multi-dimensional scaling. In *Proceedings of the 18th annual conference of the Cognitive Science Society* (pp. 290–294). Mahwah, NJ: Lawrence Erlbaum Associates.

Deerwester, S. C., Dumais, S. T., Furnas, G. W., Landauer, T. K., & Harshman, R. A. (1990). Indexing by latent semantic analysis. *Journal of the American Society of Information Science, 41*, 391–407.

Deza, M. M., & Laurent, M. (1997). *Geometry of cuts and metrics.* Berlin: Springer-Verlag.

Frieze, A., Kannan, R., & Vempala, S. (1998). Fast Monte-Carlo algorithms for finding low-rank approximations. In *Proceedings of the 39th ieee symposium on foundations of computer science.* Los Alamitos, CA: IEEE Computer Society Press.

Karp, R. M. (1972). Reducibility among combinatorial problems. In R. Miller & J. Thatcher (Eds.), *Complexity of computer computations* (pp. 85–103). New York: Plenum Press.

Kruskal, J. B. (1964a). Multidimensional scaling by optimizing goodness of fit to a nonmetric hypothesis. *Psychometrika, 29*, 1–27.

Kruskal, J. B. (1964b). Nonmetric multidimensional scaling: A numerical method. *Psychometrika, 29*, 115–129.

Lund, K., & Burgess, C. (1996). Producing high-dimensional semantic spaces from lexical co-occurrence. *Behavior Research Methods, Instruments, and Computers*, *28*, 203–208.

Pearlmutter, B. A. (1989). Learning state space trajectories in recurrent neural networks. *Neural Computation*, *1*, 263–269.

Plaut, D. C., & Shallice, T. (1993). Deep dyslexia: A case study of connectionist neuropsychology. *Cognitive Neuropsychology*, *10*, 377–500.

Richardson, M. W. (1938). Multidimensional psychophysics. *Psychological Bulletin*, *35*, 659.

Rumelhart, D. E., Hinton, G. E., & Williams, R. J. (1986). Learning internal representations by error propagation. In D. E. Rumelhart, J. L. McClelland, & the PDP Research Group (Eds.), *Parallel distributed processing: Explorations in the microstructure of cognition. Volume 1: Foundations* (pp. 318–362). Cambridge, MA: MIT Press.

Sahni, S., & Gonzales, T. (1976). P-complete approximation problems. *Journal of the ACM*, *23*, 555–565.

Shepard, R. N. (1962). The analysis of proximities: Multidimensional scaling with an unknown distance function. *Psychometrika*, *27*, 125–139, 219–246.

Shepard, R. N. (1966). Metric structures in ordinal data. *Journal of Mathematical Psychology*, *3*, 287–315.

Shepard, R. N., Romney, A. K., & Nerlove, S. B. (1972). *Multidimensional Scaling, Volume I: Theory.* New York: Seminar Press.

Torgerson, W. S. (1952). Multidimensional scaling: I. Theory and method. *Psychometrika*, *17*, 401–419.

Torgerson, W. S. (1965). Multidimensional scaling of similarity. *Psychometrika*, *30*, 379–393.

Williams, J. W. J. (1964). Algorithm 232: Heapsort. *Communications of the ACM*, *7*, 347–348.

Young, G., & Householder, A. S. (1938). Discussion of a set of points in terms of their mutual distances. *Psychometrika*, *3*, 19–22.