NONLINEAR DIMENSIONALITY REDUCTION APPLIED TO THE BINARY CLASSIFICATION OF IMAGES

AUTHOR:
CHAE CLARK
DEPARTMENT OF MATHEMATICS
UNIVERSITY OF MARYLAND, COLLEGE PARK

ADVISOR:
DR. KASSO OKOUDJOU
DEPARTMENT OF MATHEMATICS
UNIVERSITY OF MARYLAND, COLLEGE PARK

Abstract. In this project we are interested in the reduction of high-dimensional data points x from a space $\mathbb{R}^D$ to a lower dimensional space $\mathbb{R}^d$ (where $d << D$) in a way that preserves certain important characteristics. In particular, we are interested in reducing the size of high-dimensional points for their application to the classification of signals. We first examine a MatLab implementation of the Locally Linear Embeddings [LLE] algorithm, and apply it to a specific image database. We then use the output of the LLE and the original dataset to test and compare the performance ability of a MatLab implementation of a support vector machine [SVM].

1. Background & Introduction

We start by introducing the necessary background and motivation for this project. We first describe how high-dimensional datasets can cause problems when attempting to process. We then give a brief overview of Linear and Nonlinear Dimensionality Reduction techniques. In the following section, the particular dimension reduction technique discussed in this paper is presented. An introduction to Support Vector Machines follows. And finally, we present our results, conclusions, further work, and references.

1.1. High Dimensional Data.

At a basic level, the field of dimensionality reduction is concerned with taking high-dimensional data points and representing them with fewer elements while preserving the important features of the data. In current data collection and processing applications, an enormous amount of data can now be stored easily and efficiently. This allows for tons of information to be collected on any specific task. The issue then becomes the usefulness of this high-dimensional data. "Handling" data points in higher dimensions requires more computing power, but it may not be the case that all of the collected data is meaningful. Redundant and unnecessary information may be present. This is the curse of dimensionality; we want to collect more information to create more accurate model predictions, but we sometimes are not sure which information is relevant to the application at hand. Errors in our measurements can also lead to problems when dealing with high dimensions.

1.2. Dimensionality Reduction.

To combat the curse of dimensionality, various techniques and algorithms have been developed to choose only the important features from a high-dimensional point. There are two fields in dimension reduction, linear techniques, that use a linear mapping to reduce the dimension, and nonlinear techniques, which are the focus of this project, that make the assumption that the data available is embedded on a manifold (or surface in

Date: May 14, 2013.

lower dimensional space). The data is then mapped onto a lower-dimensional manifold for more efficient processing.

1.3. **Linear Dimensionality Reduction [LDR].** With Linear Dimension Reduction techniques [LDR], we are searching for a matrix A_{LDR}, such that when we apply it to our dataset $X \in \mathbb{R}^{D\times N}$, we get a "faithful" lower dimensional representation $Y \in \mathbb{R}^{d\times N}$ ($d << D$).

$$A_{LDR} \cdot X = Y \tag{1}$$

"Faithful," in this context can be any structure, or properties of your data that you wish to retain. Principal Component Analysis [PCA] is an example of a Linear Dimension Reduction technique. We reduce the dimension of a dataset by retaining only the largest eigenvalues/eigenvectors of X and set the remaining eigenvalues to 0.

Linear Dimension Reduction techniques are, in general, faster to implement than other reduction methods, but they make the key assumption that the data can be viewed as lying on a linear manifold. This is rarely the case in real-world data.

Some other examples of LDR techniques are: Linear Discriminant Analysis [LDA], Independent Component Analysis [ICA], etc.

1.4. **Nonlinear Dimensionality Reduction [NLDR].** For cases when we cannot assume that our data lies on a linear manifold, we must resort to more powerful techniques. Any dimensionality reduction method that cannot be written in the form $A \cdot X = Y$, is considered a Nonlinear Dimensionality Reduction technique [NLDR]. Here, instead of looking for a linear mapping to a lower dimensional space, we look for an optimal mapping subject to some objective (usually to preserve some geometric properties).

Nonlinear Dimensionality Reduction techniques allow us to accurately represent a wider range of data in low dimensions, but at the cost of computational efficiency in the best case (and stability in the worst). Specifically for this project, we only consider the nonlinear reduction scheme, Local Linear Embeddings (LLE). In this scheme each high-dimensional data point $x \in \mathbb{R}^D$ is represented by a linear combination of its nearest neighbors (in euclidean distance). A lower-dimensional point $y \in \mathbb{R}^d$ ($d << D$) is then constructed to preserve local properties in an optimal way. This scheme is computationally simple and has other qualities that render it useful in a variety of signal processing applications.

It is important to note that many NLDR schemes rely heavily on the assumption that the local structure of a dataset X is approximately linear. Furthermore, many NLDR methods come about as a modification of a linear method. As an example, take the Linear Dimensionality Reduction method of taking the inner products of each data point and a number of basis elements. If we truncate our result, this scheme can be written as

$$I_{d\times N} \cdot (A \cdot X) = Y \tag{2}$$

What we have written here is a linear dimensionality reduction method. If we further add the requirement that instead of using the first d components of our projection (the matrix $I_{d\times N}$), we keep the largest d components for each data point on our set X, we have now created a nonlinear dimensionality reduction technique. We are not able to express this requirement as a matrix operation, and thus, it is not a linear technique.

More information on nonlinear dimensionality reduction techniques, refer to [**Lee1**].

2. Locally Linear Embeddings [LLE]

Locally Linear Embeddings [LLE] is a non-linear dimension reduction scheme that takes a data point lying in a high dimension and maps the point in a low dimension. Developed by Roweis and Saul and published in the journal Science in 2000 [**Sau1**], LLE is described as a

neighborhood preserving embedding algorithm. The data points are assumed to lie on a well-behaved manifold where we further assume that any small patch of the manifold is approximately linear. Implicitly, it is also assumed that the data points can actually be efficiently represented in a lower dimensional space. This is the equivalent of saying the data point in high dimensional space has irrelevant or redundant information.

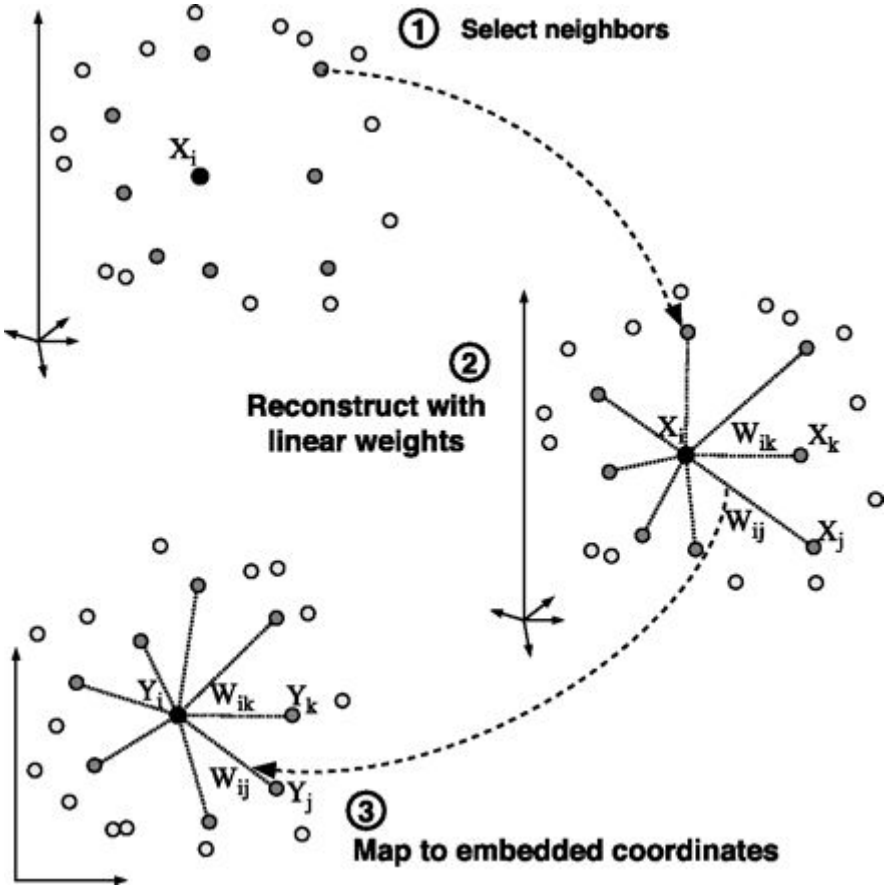


FIGURE 1. This figure illustrates the steps of the LLE algorithm. It was provided by the Sam Roweis from his LLE webpage. **http://www.cs.nyu.edu/∼roweis/lle/algorithm.html**

More formally, given a set of data points $X = \{x_i \,|\, x_i \in \mathbb{R}^D, \text{ for } i = 1, 2, \ldots, N\}$ where D is large, the LLE algorithm finds a corresponding set of data points $Y = \{y_i \,|\, y_i \in \mathbb{R}^d, \text{ for } i = 1, 2, \ldots, N\}$ where d is small and the relative distance between points is preserved from $\{x_i\}$ to $\{y_i\}$ (the distance metric here is Euclidean). This mapping is accomplished through three steps that are detailed below.

Step 1: Given a data set X, where $|X| = N$, we must first find K nearest neighbors for each point in our data set. These neighbors form the structure we wish to preserve. The nearest neighbors of each data point x_i are denoted $\{\eta_{ij}\}_{j=1}^{K}$. The double subscript (ij) refers to the j-th nearest neighbor of the i-th data point. It's important to note that the set of nearest neighbors does not include the i-th data point itself.

Step 2: With the set of nearest neighbors for each data point, we now want to approximate these points with a linear combination of their nearest neighbors, and we want the difference of the approximation and the data point to be as small as possible. For each neighbor η_{ij}, we denote the coefficient of the linear combinations w_{ij}. We also denote the matrix of these values (from here on referred to as the weights matrix) $W = [w_{ij}]$. We can model this as a constrained minimization problem.

$$\min_W : E(W) = \sum_{i=1}^{N} \left\| x_i - \sum_{j=1}^{K} w_{ij}\eta_{ij} \right\|_2^2 \tag{3}$$

The first constraint of this problem is that

$$w_{ij} = 0 \tag{4}$$

for any indices (ij), where x_j is not a nearest neighbor of x_i, or more succinctly, when $x_j \notin \{\eta_{ik}\}_{k=1}^{K}$. This constraint ensures that the algorithm only attempts to preserve the structure between data points and their nearest neighbors. It's also worth noting that for all i,

$$w_{ii} = 0 \tag{5}$$

The second constraints is that

$$\sum_{j=1}^{N} w_{ij} = 1, \text{ for } i = 1, 2, \ldots, N \tag{6}$$

This constraints ensures the rows of the weight matrix W sum to one. It's required to ensure that the solution is invariant to the scaling, rotation, and translation of data points. With this constraint, the data is no longer dependent on the current frame of reference. Instead, the weights will only represent relationships between data points.

Step 3: Now that we have the optimal weight matrix, we must find the lower-dimensional representation $Y = \{y_i \,|\, y_i \in \mathbb{R}^d\}$ for the high-dimensional data points $X = \{x_i \,|\, x_i \in \mathbb{R}^D\}$. As we wish to maintain the same structure in this mapping, we can model our search as another constrained optimization problem. Here, for clarity, we denote $\{\rho_{ij} \,|\, \rho_{ij} \in \mathbb{R}^d\}_{j=1}^{K}$ to be the low dimensional data points corresponding to the nearest neighbor sets $\{\eta_{ij}\}_{j=1}^{K}$.

$$\min_Y : e(Y) = \sum_{i=1}^{N} \left\| y_i - \sum_{j=1}^{N} w_{ij}\rho_{ij} \right\|_2^2 \tag{7}$$

The first constraint is that

$$\sum_{i=1}^{N} y_i = 0 \tag{8}$$

This constraint centers the points around the origin (to remove translational invariance). The other constraint is that

$$\frac{1}{N} \sum_{i=1}^{N} y_i^T y_i = I_d \tag{9}$$

This constraint removes the rotational invariance. It's worth noting that $y_i^T y_i$ is an outer product.

Once this problem has been solved, the solution Y is the set of lower-dimensional representations of the original data points. Using details from the original paper by Roweis and Saul [**Sau1**], the book by Theodoridis and Koutroumbas [**Kou1**], and the book by O'Leary [**Ole1**], solution methods are presented in the following subsections.

2.1. **Nearest Neighbor Search.** Finding the K nearest neighbors is the first step in the LLE algorithm. To do this, we will use a full enumeration scheme which involves computing the pairwise distances between the data points, then selecting the K closest points. An algorithm is presented below.

Algorithm 1 Nearest Neighbor Search by Full Enumeration

1: **for** each point $x_i \in X$ **do**
2: compute the distance between x_i and each other point $x_j \in X$
3: sort distances in ascending order (keeping track of the original indices)
4: choose the K indices with the smallest distances
5: return the indices of the K nearest neighbors
6: **end for**

2.2. **Validation.** To validate our implementation of the nearest neighbor search, we will use MatLab's implementation. The comparison results are now presented. Let $S_1(q, Q)$ be the ordered set of nearest neighbors to point q residing in point-set Q resulting from the MatLab code knnsearch.m. And let $S_2(q, Q)$ be the ordered set of nearest neighbors to point q residing in point-set Q resulting from the MatLab code knn.m (our implementation). We apply both functions to random datasets and compare based on two metrics:

1. [Presence] Are the same neighbors present in both sets ($S_2(q, Q) \subset S_1(q, Q)$).
2. [Preservation] Is the neighbor order preserved (the same) in both sets ($S_1(q, Q) = S_2(q, Q)$).

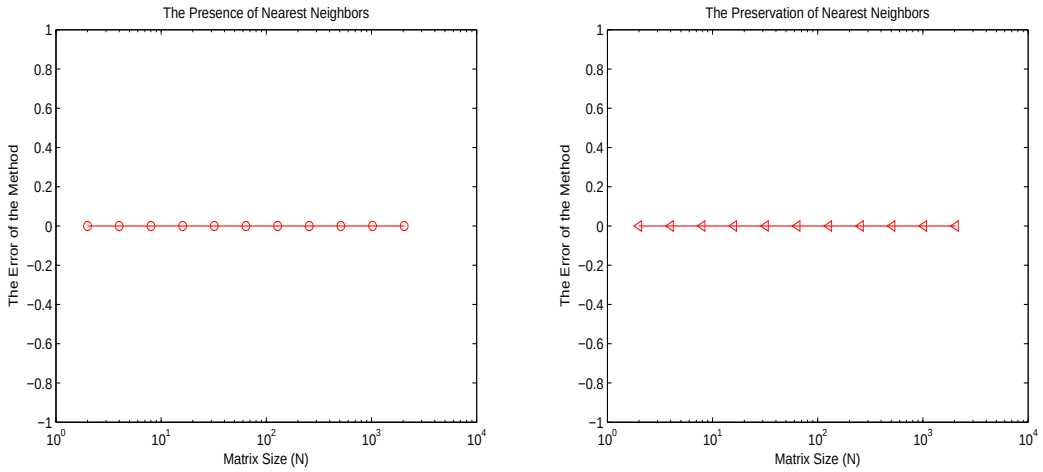


FIGURE 2. As the two graphs show, there is no difference in the output of the methods in the preservation or presence of nearest neighbors.

2.3. **Minimal Reconstruction Error.** After the nearest neighbors have been found, the weights that reconstruct each point as a linear combination of its neighbors must be found. This is accomplished by solving

$$\min_W : \sum_{i=1}^{N} \left\| x_i - \sum_{j=1}^{K} w_{ij}\eta_{ij} \right\|_2^2 \tag{10}$$
$$\text{st.: } w_{ij} = 0 \quad \text{for } x_j \text{ not a neighbor of } x_i$$
$$\sum_{j=1}^{N} w_{ij} = 1, \text{ for } i = 1, 2, \ldots, N$$

There is a closed formula solution to this problem, and the following algorithm implements this solution

Algorithm 2 Weight Construction

1: **for** each point $x_i \in X$ given its neighbors $\{\eta_{ij}\}$ **do**
2: center the points, $\widetilde{\eta}_{ij} = \eta_{ij} - x_i$
3: compute the Gram matrix $G = [G_{jk}]$ whose elements are $\langle \widetilde{\eta}_{ij}, \widetilde{\eta}_{ik} \rangle$
4: solve the system $Gw_i' = \frac{1}{2}\mathbf{1}_K$ for w_i'
5: compute the Lagrange multiplier $\lambda = \frac{1}{\sum_{j=1}^{K} w_j'}$
6: set $w_i = \lambda \cdot w_i'$
7: return the weights w_i
8: **end for**

2.4. **Validation.** From the algorithms above, MatLab code was implemented (weights.m). The validation of this code is incorporated into other tests and so we only present figures showing that the constraints are satisfied in our code. For a derivation of the algorithm, see [**Unk1**] or Appendix (**A**).

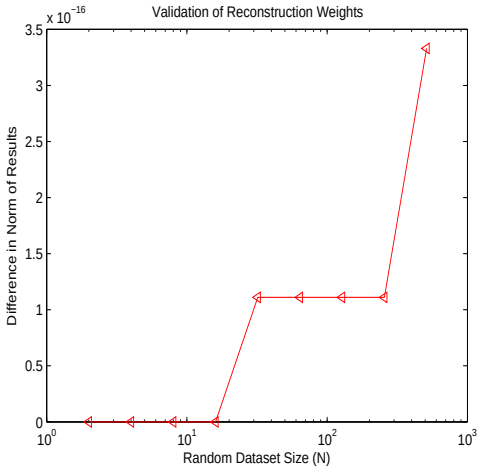


FIGURE 3. This graph shows the difference, in norm, of the actual optimal weights and the output of weights.m.

2.5. **Reconstruction in Low Dimension.** The final step in the Locally Linear Embeddings Algorithm is the computation of the lower dimensional embedding Y. This is accomplished by solving

$$\min_{Y} : \sum_{i=1}^{N} \left\| y_i - \sum_{j=1}^{N} w_{ij}\rho_{ij} \right\|_2^2 \tag{11}$$

$$\text{st.: } \sum_{i=1}^{N} y_i = 0$$

$$\frac{1}{N}\sum_{i=1}^{N} y_i^T y_i = I_d$$

It has been shown by [**Unk1**], [**Sau1**], and [**Sau2**] that solving the above problem is equivalent to performing an eigen-decomposition of $(W-I_N)^T(W-I_N)$. We denote the matrix whose columns are eigenvectors $V = [v_1\, v_2\, \ldots\, v_D]$. Our lower-dimensional dataset (whose rows are data points) is then set to be $Y = [v_1\, v_2\, \ldots\, v_d]$, where the eigenvectors correspond to the smallest eigenvalues (this assumes the eigenvalue/eigenvector pair corresponding to an eigenvalue of 0 is removed). An algorithm is presented below.

Algorithm 3 Embedding Construction

1: construct the matrix $(W - I_N)^T(W - I_N)$
2: compute the eigenvalues and eigenvectors of $(W - I_N)^T(W - I_N)$
3: remove the eigenvalue/eigenvector pair corresponding to the eigenvector of 0
4: collect the eigenvectors corresponding to the lowers d eigenvalues
5: return the eigenvectors $[v_1\, v_2\, \ldots\, v_d]$ as the dataset Y

2.6. **Validation.** From the algorithm above, MatLab code was implemented (embd.m). The validation of this code is incorporated into the full LLE test, and so we only present figures that show that the code satisfies the constraint requirements.

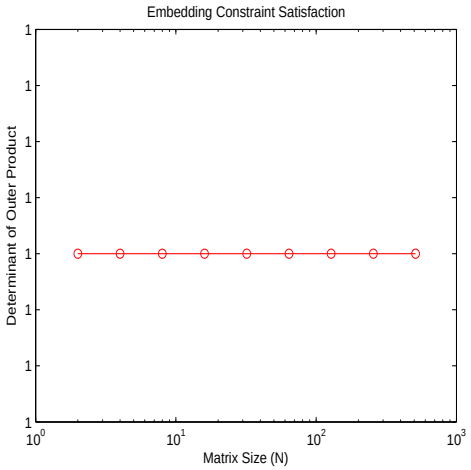


FIGURE 4. This graph shows that the constraints for problem (7) are enforced.

3. Eigenvalues & Eigenvectors

In finding the lower dimensional embedding, we must perform an eigen-decomposition, so we also implement an eigen-solver for real, symmetric, and positive definite matrices. Because we plan to use this eigen-solver (eigQR.m) to find eigenvectors for large datasets, an iterative method is employed. To find our eigenvalues, we wish to decompose our matrix A into VDV^T. We can

accomplish this decomposition, we factor A into QR, where Q is an orthogonal matrix, and R is an upper triangular matrix. We then multiply A by Q such that $A \leftarrow QA$. By repeatedly performing these two operations, A approaches D, and the product of our Q's converges to V. The algorithm is shown below, but for more background, see [**Pan1**].

Algorithm 4 Iterated QR Method

1: designate the real, symmetric, positive definite matrix A
2: $V = I_N$ (the approximation of the eigenvectors)
3: $D = A$ (the approximation of the eigenvalues)
4: **for** k=1,2,... **do**
5: $[Q, R]$ = the QR factorization of D
6: $V = V \cdot Q$
7: $D = R \cdot Q$
8: **end for**

3.1. **Complexity & Convergence.** We now turn to the analysis of the complexity of this algorithm. It requires three main operations, a QR factorization, and two matrix multiplications. To analyze the complexity of the QR factorization, we take the view that it is accomplished through Gram-Schmidt Orthogonalization [**Olv1**]. In brief, the Gram-Schmidt process takes a set of vectors [here, represented as a matrix $A \in \mathbb{R}^{N\times N} = [a_1, a_2, \ldots, a_N]]$ and iteratively transforms them into the orthonormal basis vectors of $Q = [q_1, q_2, \ldots, q_N]$. The matrix R is then formed by taking the inner-product of the columns of Q and the columns of A, such that for $R = (r_{ij})$, $r_{ij} = \langle q_i, a_j \rangle$ for $j \geq i$, and $r_{ij} = 0$ otherwise.

Now we count the number of inner-products required to factor $A \in \mathbb{R}^{n\times n}$. There are n iterations in the Gram-Schmidt process, and at the p-th iteration, we require $(p-1)$ inner-products and a norm computation (the inner-product of a vector and itself). We then perform $(n)(n+1)/2$ inner-products to fill the upper-triangular matrix R. By summing these operations, we achieve a complexity of

$$O\left[\frac{1}{2}\cdot(n^2+n)+\frac{1}{2}\cdot(n^2+n)\right] = O\left[n^2\right] \tag{12}$$

The number of inner-products required to update V and D is n^2 and n^2 respectively, which results in a complexity of

$$O[n^2+n^2] = O[n^2] \tag{13}$$

Combining these, we get a complexity of

$$O[n^2] \tag{14}$$

If we go further, we can account for multiplications as well, by adding an order of magnitude to our result, which gives us

$$O[n^3] \tag{15}$$

If we now try to analyze to complexity of the full algorithm, we must supply a stopping iteration, but supplying this doesn't guarantee convergence. To bypass this stumbling block, we can try to empirically divine the amount, and below, we present graphs exploring the number of iterations required to converge. Convergence is defined to be the maximum absolute value of the off-diagonal elements being below a tolerance (set to be 1E-5).

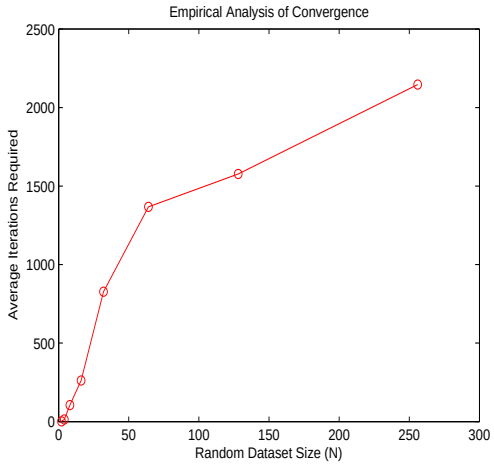


FIGURE 5. This graph displays some of the convergence properties of the iterative QR method.

Looking at this graph, we may make the observation that the number of iterations required to converge (m) is related to the size of A (which we designate n). This relation seems to be relatively linear,

$$m = C \cdot n \tag{16}$$

where C in this case appears to be approximately 10. This would give us a total complexity of

$$O[m \cdot n^3] = O[10n \cdot n^3] = O[n^4] \tag{17}$$

required multiplications.

3.2. **Validation & Testing.** We now present some results that show the correctness of our code. We use random datasets of various sizes.

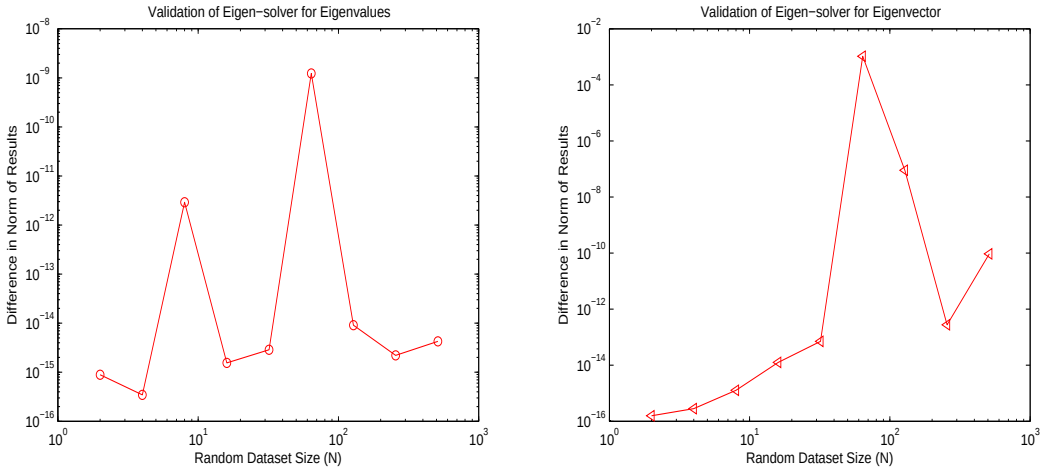


FIGURE 6. The left figure shows the difference in eigenvalue of our function eigQR.m and MatLab's eigs.m. The right figure shows the inner product between eigenvectors. The tolerance here is 1E-5.

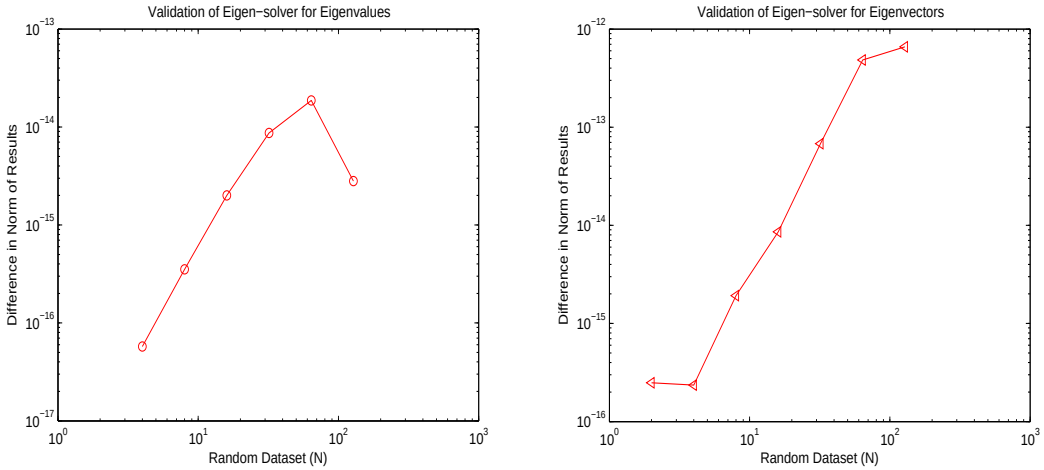


FIGURE 7. The left figure shows the difference in eigenvalue of our function eigQR.m and MatLab's eigs.m. The right figure shows the inner product between eigenvectors. The tolerance here is 1E-16.

As a further comparative measure, we present timing results of our function (eigQR.m) versus MatLab's eigs.m function.

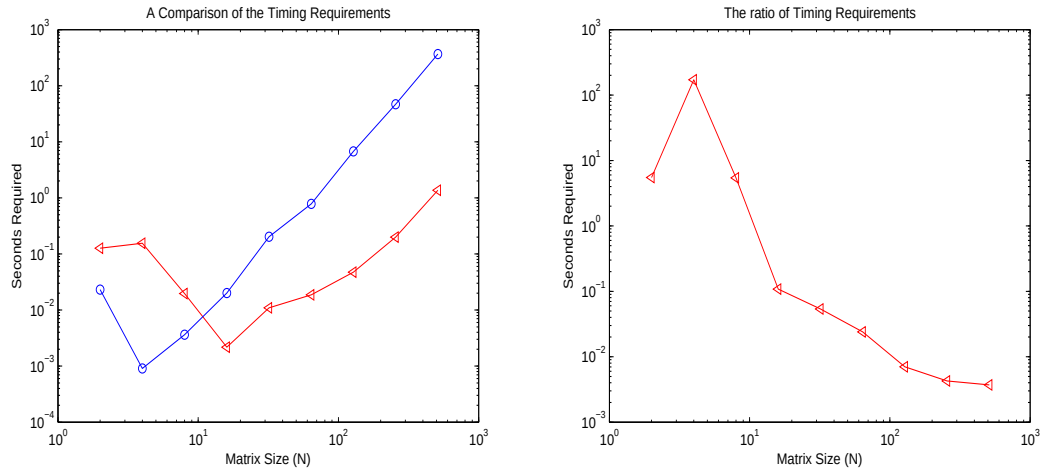


FIGURE 8. The left figure shows the times required per dataset size.

4. Parallelization of LLE

One thing to consider in the LLE algorithm is its ability to be parallelized. To parallelize an algorithm, there must be elements that can be decoupled from each other, or there must be certain notion of independence in its steps. This is achievable when determining the optimal weights. For each point, we must solve a linear system, and each of these are independent, which implies that we can parallelize this step. This isn't as possible with determining the eigenvalues of our matrix (although with additional information, this may be possible).

With our LLE algorithm now fully formed, we turn our attention to image classification, starting with the machine learning algorithm Support Vector Machines [SVM].

5. Support Vector Machines [SVM]

The specific application of this proposed project is in image classification. Given a set of images $\{\mathcal{I}_i\}$, we want to be able to correctly distinguish between their various properties. Here we take the simple case of binary classification, where we assume that any image considered is either in a class $\mathcal{A}$ or class $\mathcal{B}$. One of the main goals in pattern recognition is to be able to detect the differences between various objects. There are a variety of ways to do this, but one of the most popular utilizes Support Vector Machines [SVM]. Various features of the image under study are arranged into a vector (data point x_i) that is a representation of its properties. Each data point is then assigned one of the classes above. For notational convenience, we will designate z_i as the class designation of a data point x_i, where $z_i \in \{-1, 1\}$.

With support vector machines, a number of these data points are collected and called a training set, as their classes are known. These are used to find a hyperplane $\mathcal{H} = \langle u, u_0 \rangle$ that separates the two classes. Here, u is the normal vector to the hyperplane. u_0 is the offset of the hyperplane from the origin. In general there are an infinite number of hyperplanes that have this property, but the hyperplane desired is the one that maximizes the distance between elements in a class and the hyperplane itself (this is called the maximal margin hyperplane). The objective for this problem becomes finding the hyperplane whose normal vector u accurately classifies each training vector and has maximal margin. Our problem can be solved through constrained optimization, formulated below.

$$\arg\min : \frac{1}{2}\|u\|^2 \tag{18}$$

This problem is subject to the constraint that all of the data points x_i are labeled correctly, or

$$z_i(u^T x_i + u_0) \geq 1 \tag{19}$$

This problem will only have a solution if there exists a hyperplane that separates the data. To account for the case where our data is not separable in this way, we can relax the constraints, or form the Lagrangian problem, presented below.

$$\arg\min L(v, v_0, \mu) = \frac{1}{2}\|u\|^2 - \sum_{i=1}^{N} \mu_i[z_i(u^T x_i + u_0) - 1] \tag{20}$$

With this, we can test classification accuracy, when the dataset output for LLE is applied.

6. Software & Hardware

The algorithms stated above are to be implemented in the programming language MatLab. This decision is primarily due to the languages' flexibility in syntax, its ubiquitous use by the scientific community, and the wide availability of support and toolboxes. In particular, the optimization, linear algebra, and sparse matrix support, make it an ideal choice for scientific computing tasks.

The main machine used for development, validation, and testing was a Dell PC with Windows 7. The 64-bit machine has 6GB of RAM, and 4 AMD Phenom II 3.00 GHz processors.

7. Outside Code & Toolboxes

A few outside toolboxes and code written by other authors (not included in MatLab) are used in development, validation, and testing.

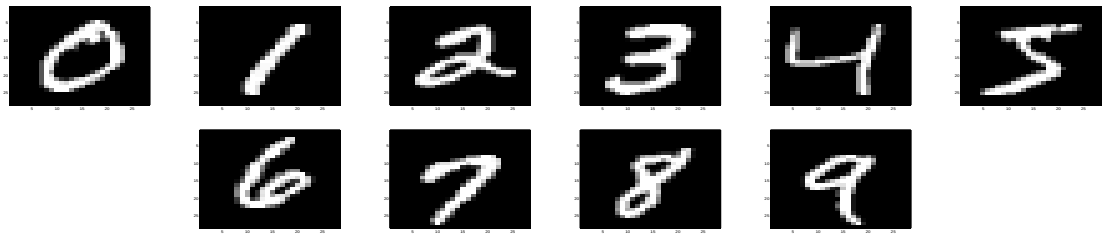
7.1. **MatLab's Dimensionality Reduction Toolbox.** Available from the LLE author's website (http://homepage.tudelft.nl/19j49/Matlab_Toolbox_for_Dimensionality_Reduction.html) there is an implemented and validated code set for performing data dimensionality reduction as well as some associated Locally Linear Embedding. Using this code, and some of the test function mentioned in Standard Topological Manifolds, the results can be compared to ensure an accurate implementation.

7.2. **Author's Locally Linear Embedding Code.** Available from the LLE author's website (http://www.cs.nyu.edu/ roweis/lle/) there is an implemented and validated code set for performing Locally Linear Embedding. Using this code, and some of random datasets, the results can be compared to ensure an accurate implementation.

7.3. **Databases.** A database of handwritten images (of the digits 1-9) is used in the following testing. The full dataset contains 60,000 (28×28) training images, and 10,000 (28×28) testing images. In the processing of these images, we stack the columns into single 784-length data points.

This dataset is provided by the National Institute of Standards and Technology [NIST] and is available at:

http://yann.lecun.com/exdb/mnist/



8. LLE Validation & Testing

Given that we have now validated the constituent pieces of the LLE algorithm, we can now validate the algorithm in full, as well as perform some tests. We first compare the preservation properties of our implementation of LLE (LLE.m) against the implementation of the author (Sam Roweis). Our datasets, in these tests, consist of randomly generated matrices $X \in \mathbb{R}^{D \times N}$ where $D = N$ and N is varied. The number of nearest neighbors (K) used is set to be some scalar of the log of the dimension. And the target dimension (d) is set in a similar way.

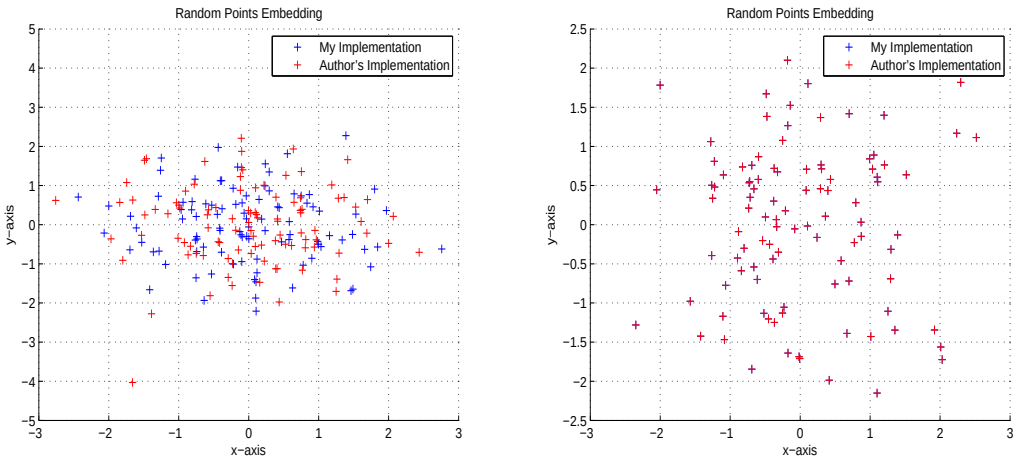


FIGURE 9. (Left) Random Data Embedding with a norm difference of absolute value $1.1165e-11$. (Right) Random Data with a norm difference of absolute value $2.2495e-12$

8.1. **Topological Surfaces.** We also validated the algorithm with a few standard topological surfaces (namely the Swiss Roll, Twin Peaks, Gaussian, and Logarithmic). For these functions, the 2 dimensional manifold is known, which allows us to check our LLE output against the known 2-D manifolds of these surfaces.

Swiss Roll:

$$F : (x, y) \to (x\cos(x), y, x\sin(x)) \tag{21}$$

Gaussian Distribution:

$$f(x, y) = \frac{1}{\sqrt{2\pi}} e^{-\left(\frac{x^2+y^2}{2}\right)} \tag{22}$$

Twin Peaks Function:

$$g(x, y) = x^4 + 2x^2 + 4y^2 + 8x \tag{23}$$

Logistic Function:

$$h(x, y) = \frac{1}{1 + e^{-x}} \tag{24}$$

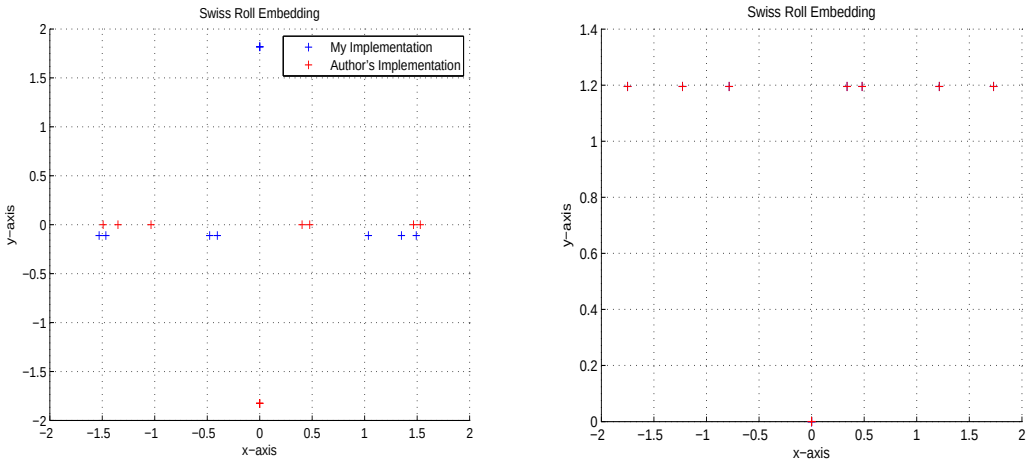


FIGURE 11. (Left) Swiss Roll Embedding with a norm difference of absolute value $5.2766e-13$. (Right) Swiss Roll with a norm difference of absolute value $5.9560e-13$

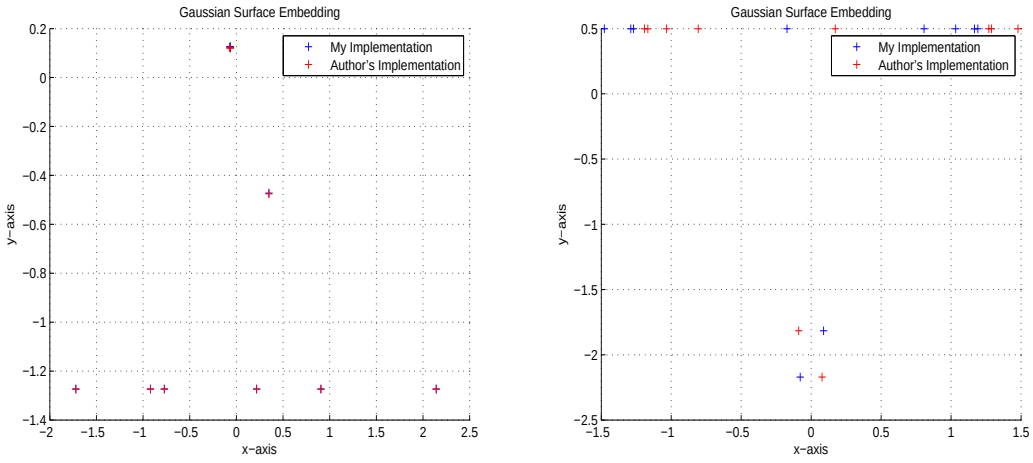


FIGURE 10. (Left) Gaussian Embedding with a norm difference of absolute value $7.3131e-10$. (Right) Gaussian Embedding with a norm difference of absolute value $5.1161e-12$

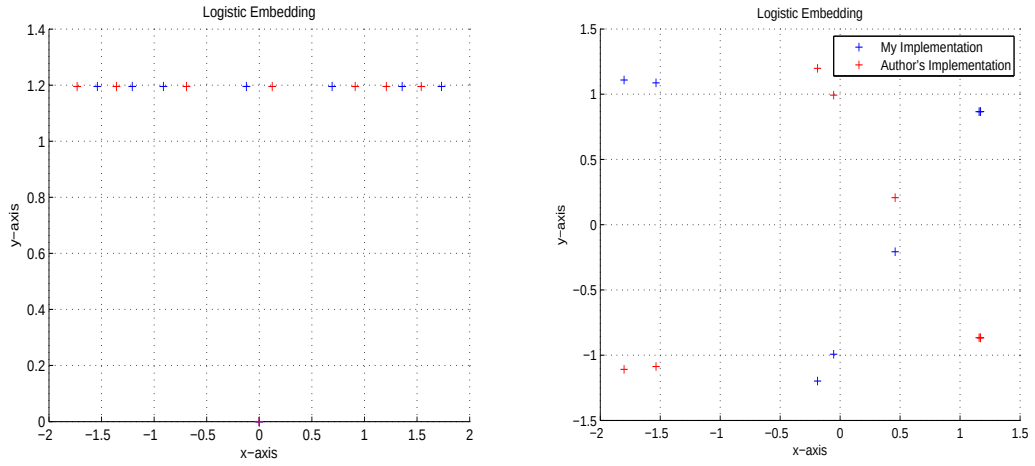


FIGURE 12. (Left) Logistic Embedding with a norm difference of absolute value $1.4568e-12$. (Right) Logistic Embedding with a norm difference of absolute value $1.3797e-9$

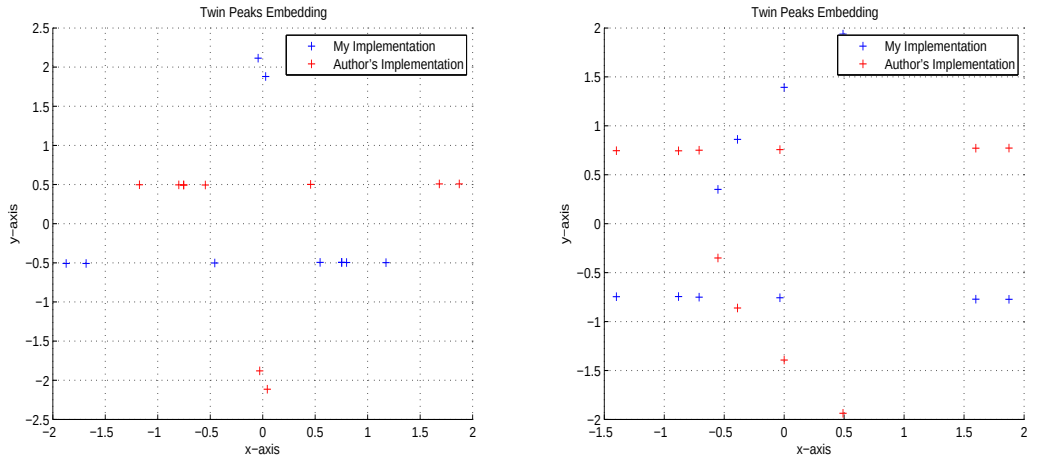


FIGURE 13. (Left) Twin Peaks Embedding with a norm difference of absolute value $1.8453e-9$. (Right) Twin Peaks Embedding with a norm difference of absolute value $4.6439e-13$

The reader may note that the embedded points, resulting from the two algorithms, do not exactly match. This is due to the two different eigen-decompositions used. The eigenvectors are orthonormal, but multiplying elements by -1 retains orthonormality and is different from the original eigenvector. This is why taking the absolute value of the points first provides a small difference in norm.

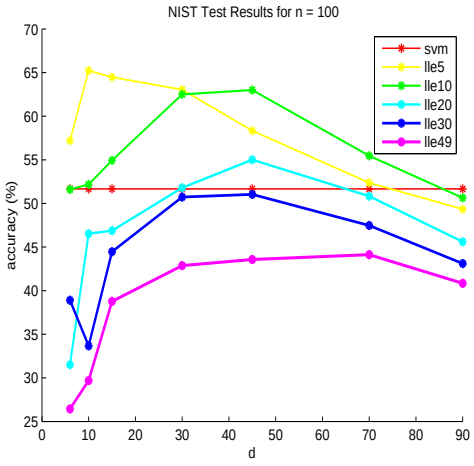
9. RESULTS

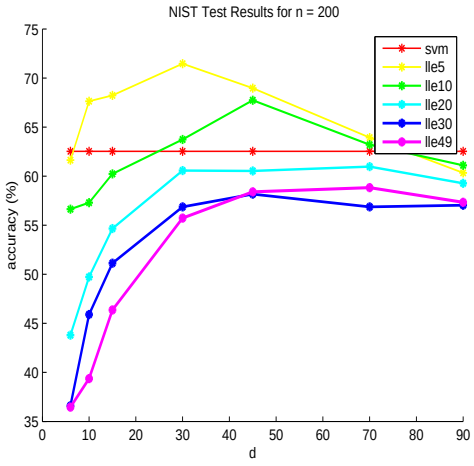
We now turn to the goal we stated in the introduction. We wish to test the ability of the dimensionality reduction algorithm LLE to pre-process images to allow for more accurate classification (in one case), and mitigation of accuracy loss (in the other).

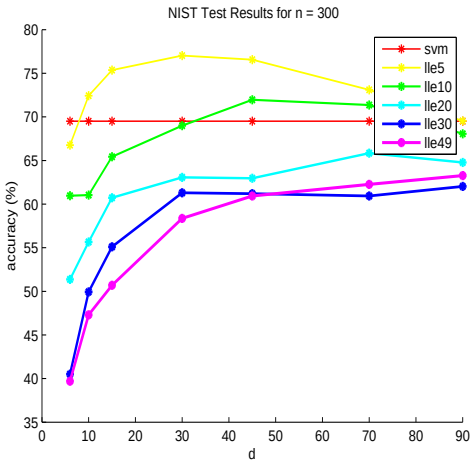
For these test, a training set of images is used to train an SVM classifier. We then take a testing set, and see how well our SVM training step performed. We then take the same training and testing sets, and apply the LLE algorithm. We train an SVM with the embedded training

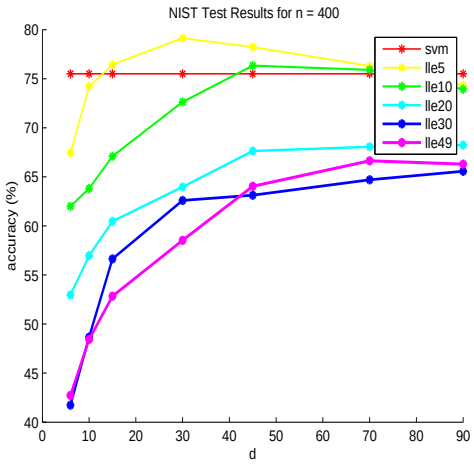
set, and collect the accuracy of the trained SVM on the embedded testing set. With these two results, we can then compare and analyze the effect of pre-processing with LLE.

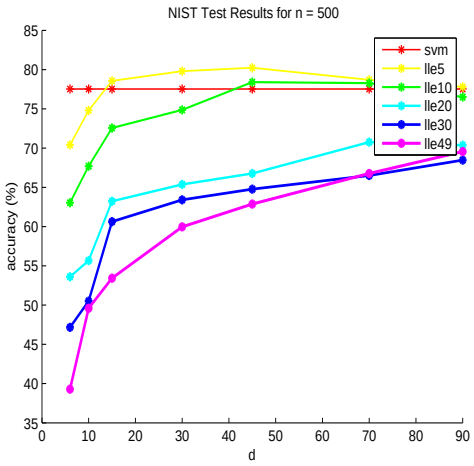
We start with small values of N for the training set size, and increase them. We take a constant M throughout all tests ($M = 3000$) for the testing set size. In these tests, we also take various numbers of nearest neighbors K (these are labeled in the figures as a number next to the algorithm name lle15).

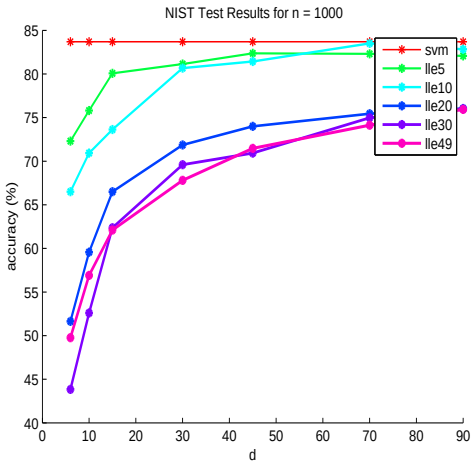


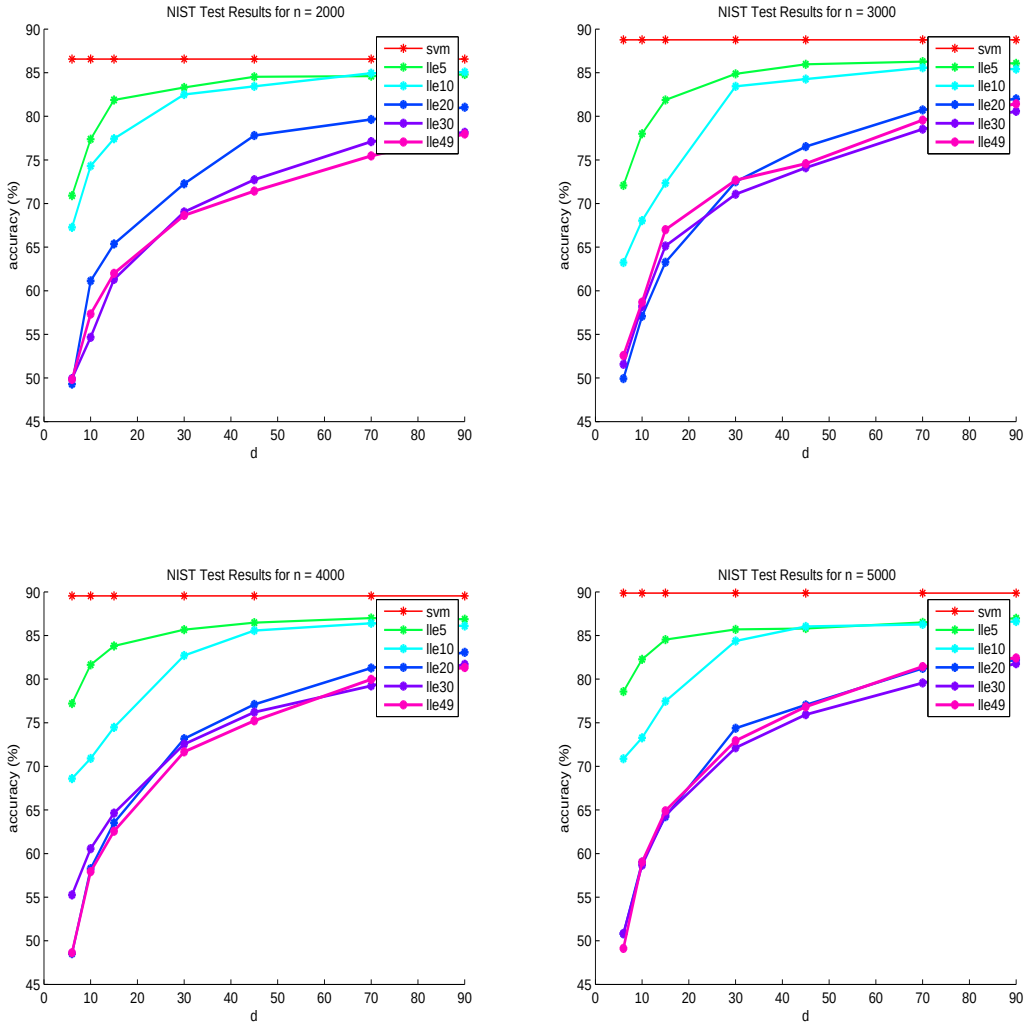












9.1. **Conclusion.** The first observation we make is with regards to the classification accuracy of the embedded dataset for small training sizes. We have cases where the embedded data obtains a higher classification accuracy than the unprocessed data. This happens for two main reasons. The first is that for small training set sizes, finding a good separating hyperplane becomes increasingly difficult (as the dimension D increases). This results in a poor classification accuracy for the testing set. The other reason for this result, is that the LLE processed datasets map close points, closer together, resulting in a better separating hyperplane.

It's interesting to note that for larger training set sizes, we see a steady increase in the classification accuracies of the SVM's, where the increase is more rapid for the unprocessed dataset. In this case, the lost information (during the embedding) becomes the limiting factor for the embedded data.

In summary, considering the results as a whole, we can make the following cursory conclusion. When very few data points are available for training it may be wise to pre-process the data and accept the small information loss. But for larger datasets this may not be as sound an idea.

10. Deliverables

The deliverables for this project are the MatLab code that implements the LLE algorithm and any code used for testing (i.e. scripts for the surface creation, MatLab toolbox files, and other

pre-packaged code). The code will be optimized for performance and effective memory management, as well as being fully documented. Reports at various stages throughout the course will detail the approach, implementation, validation, testing, and extensions of the algorithm. With this information, a researcher will be able to reproduce any results present in our reports.

11. References

[Sau1] Sam Roweis and Lawrence Saul, Nonlinear Dimensionality Reduction by Locally Linear Embeddings, Science v.290 no.5500, Dec.22, 2000. pp.2323–2326.

[Kou1] Sergios Theodoridis and Konstantinos Koutroumbas, Pattern Recognition, Fourth Edition, Academic Press 2008.

[Kou2] O. Kouropteva and M. Pietikainen. Incremental locally linear embedding. Pattern Recognition, 38:1764–1767, 2005.

[Fab1] Boschetti and Fabio, Dimensionality Reduction and Visualization of Geoscientific Images via Locally Linear Embedding, Comput. Geosci., July, 2005, 31,6, 689–697.

[Yeu1] Hong Chang and Dit-yan Yeung, Robust Locally Linear Embedding, 2005.

[Chi1] Chang, Chih-Chung and Lin, Chih-Jen, LIBSVM: A library for support vector machines, ACM Transactions on Intelligent Systems and Technology, 2, 3, 2011, 27:1–27:27.

[Kri1] Georghiades, A.S. and Belhumeur, P.N. and Kriegman, D.J., From Few to Many: Illumination Cone Models for Face Recognition under Variable Lighting and Pose, IEEE Trans. Pattern Anal. Mach. Intelligence, 2001, 23, 6, 643-660.

[Wan1] Zhang Z, Wang J (2007) MLLE: Modified locally linear embedding using multiple weights. Advances in Neural Information Processing Systems (NIPS) 19, eds Scho lkopf B, Platt J, Hofmann T (MIT Press, Cambridge, MA), pp 1593–1600.

[Sau2] Sam Roweis and Lawrence Saul, An Introduction to Locally Linear Embedding.

[Pan1] Maysum Panju, Iterative Methods for Computing Eigenvalues and Eigenvectors, Waterloo Mathematics Review, University of Waterloo.

[Lee1] John A. Lee, Michel Verleysen, Nonlinear Dimensionality Reduction, Information Science and Statistics, Springer Science + Business Media 2007, New York, NY 10013.

[Unk1] Nonlinear Dimensionality Reduction I: Local Linear Embedding, 36-350, Data Mining. October 2009.

[Olv1] Peter J. Olver, Chehrzad Shakiban, Applied Linear Algebra, Pearson Prentice Hall, Upper Saddle River, NJ 07458, 2006.

[Ole1] Dianne P. O'Leary, Scientific Computing with case studies, SIAM, 3600 Market Street, 5th Floor, Philadelphia, PA, 19104-2688, 2009.

12. Appendix A (Weights Derivation)

The find the weights for the data point x_i, we minimize

$$\min_W : E(W) = \sum_{i=1}^{N} \left\| x_i - \sum_{j=1}^{K} w_{ij}\eta_{ij} \right\|^2$$

Our weights sum to one, implying that

$$\sum_{j=1}^{K} w_{ij}x_i = x_i$$

We can now write our objective function as

$$\min_W : E(W) = \left\| \sum_{j=1}^{K} w_{ij}(x_i - \eta_{ij}) \right\|^2$$

which we can further re-write as

$$\min_W : E(W) = \left\| \sum_{j=1}^{K} w_{ij}(\eta_{ij} - x_i) \right\|^2$$

Let $\tilde{z}_i = [\eta_{i1} - x_i \, , \, \eta_{i2} - x_i \, , \dots]$. Expanding the norm, we have

$$E(W) = w_i^T \tilde{z}_i^T \tilde{z}_i^T w_i = w_i^T G w_i$$

Forming the Lagrangian with our sum-to-one constraint, we have

$$L(w_i, \lambda) = w_i^T G w_i - \lambda(1_K^T w_i - 1)$$

$$\nabla_{w_i} L = 2Gw_i - \lambda 1_N = 0_N$$

This is now the system presented in the weights construction algorithm.