Overcoming the Curse of Dimensionality

Matthew Rossi

December 16th, 2021

1 Introduction

More often than not, datasets used for machine learning consist of high-dimensional data. Consider the famous MNIST[1] dataset, which contains 70,000 28x28 grayscale images of handwritten digits 0-9. Each image, then, can be represented by a vector of length 784 (where each entry is the brightness value between 0 and 255 for a particular pixel). Letting N be the number of images we want to consider at once, we can view our dataset as a matrix with N rows and 784 columns.

Working in 784-dimensional space, and high-dimensional space in general, poses problems for several reasons. One potential solution to these problems is dimensionality reduction, an approach that seeks to represent the meaning inherent in data with fewer features, i.e., in fewer dimensions. Dimensionality reduction has many applications, including noise reduction, visualization, cluster analysis, or as an intermediate step to facilitate additional analysis.

There are many algorithms for dimensionality reduction, both linear and nonlinear, including t-distributed Stochastic Neighbor Embedding (t-SNE), several types of Principal Component Analysis (PCA), discriminant analysis (LDA and GDA), non-negative matrix factorization (NMF), and more. These methods involve no small amount of linear algebra, many relying on the Singular Value Decomposition (SVD).

2 The Curses of Dimensionality

Working with high-dimensional data poses several problems. First, there's so-called the curse of dimensionality: as the number of dimensions increases, the volume of the space increases so quickly that any amount of data is sparse within the high-dimensional space. In essence, the amount of data required to obtain a reliable result grows exponentially as the number of dimensions increases. One byproduct of this is that, in high-dimensional space, the Euclidean distance between samples becomes very similar for all pairs of samples. In other words, given a reference point Q and a fixed number n of randomly chosen samples

[1]http://yann.lecun.com/exdb/mnist/

$P_1, \ldots, P_n$, the difference between the maximum distance between Q and some P_i and the minimum distance (between Q and some P_j), considered relative to the minimum distance, tends to 0. That is, if d is the number of dimensions:

$$\lim_{d \to \infty} \frac{|dist_{max}(d) - dist_{min}(d)|}{dist_{min}(d)} \to 0$$

As a further consequence, analyzing high-dimensional data, if not computationally intractable, is often computationally expensive. The computational complexity of most machine learning algorithms scales with the number of features. Machine learning techniques that divide up a feature space (like decision trees, for example), are more costly in a vast, high-dimensional space. Further, requirements for storage space scale with the number of features. If a dataset has millions of data points, each additional feature adds millions of values that must be stored. Since reading from memory can be a bottleneck for data analysis techniques, this is undesirable.

Additionally, high-dimensional data may have redundant or useless features that introduce irrelevant noise that complicates data analysis and machine learning. Finally, high-dimensional data cannot be visualized, and humans have a hard time grasping high-dimensional space because we live in a three-dimensional world. For these reasons and others, dimensionality reduction is commonly used for preparing datasets for analysis.

3 Applications of Dimensionality Reduction

Dimensionality reduction can be used to remove some noise from data.[2] For example, using PCA, one can reduce to a smaller number of components and then perform the inverse transformation up to the original number of components and the output is a smoothed approximation of the original data.

Additionally, dimensionality reduction that preserves relationships between data points is useful for visualizing data, which is often helpful for analyzing the output of clustering algorithms. Finally, dimensionality reduction is commonly an important step in preparing data for machine learning tasks like classification or regression. These tasks can be performed more accurately in the reduced space than in the original space.

4 The Theory of Dimensionality Reduction

First, it is important to note that dimensionality reduction is not performed on individual values in a vacuum. It must be performed on a representative dataset so that a successful reduction maintains the relationships between different data points in the set.

With that in mind, dimensionality reduction necessarily assumes that the information contained in high-dimensional data can be represented or captured

[2]https://scikit-learn.org/stable/auto_examples/applications/plot_digits_denoising

in fewer features, i.e., in fewer dimensions. For this to be possible, the dataset or function must have an intrinsic dimension that is less than its full dimension. Intrinsic dimension is the number of variables needed in a minimal representation of the data. For example, a two-variable function $f(x_1, x_2)$ might be equal to $g(x_1)$ for some one-variable function g. Then, we would say that the intrinsic dimension of f is one.

The measured dimension of data can be higher than its intrinsic dimension for a number of reasons. For one, there might be noise introduced into the data during measurement. However, most often, there are redundant features. If two features are highly correlated, they may both represent the same underlying information. At the least, the data can be represented without one of these redundant features.

This illustrates the principle of feature selection. Given data in N variables, choose the most salient K features with $0 < K < N$. In forward feature selection, we start with zero features and, at each iteration, add the most useful feature until the data is sufficiently represented. In backward feature elimination, we start with all features and, at each iteration, eliminate the feature that contributes the least.

Feature selection is contrasted with feature extraction, in which variables are combined to build derived features that are informative and non-redundant. The latter is the general strategy for dimensionality reduction—rather than simply removing variables that contain little data, the goal is to distill the meaning from many variables into fewer variables.

However, even if we could easily identify the minimum number of features required to represent the data, using extraction to reduce to that number of features always carries the potential for information loss. Typically, the combination of variables into a feature only approximates, but does not perfectly capture, all the information contained in the raw variables.

Hence, in this project, I examine various algorithms for dimensionality reduction and perform experimental tests (using the MNIST dataset) to evaluate their performance. The most effective algorithm will have the least information loss (as determined by accuracy on a classification task) when reducing to a fixed number of components.

5 Dimensionality Reduction Algorithms

5.1 Principal Component Analysis (PCA)

The task of dimensionality reduction requires algorithms to discover a new, smaller set of features that contain as much information about the underlying data as possible. Principal component analysis[3] (PCA) assumes that the most informative features are the ones that have the greatest variance over the dataset. As such, it finds a set of p orthogonal unit vectors (called principal

[3]https://en.wikipedia.org/wiki/Principal_component_analysis

components), where each vector is the direction of best fit for the data and is orthogonal to all the previously chosen vectors.

A line of “best fit” means the line that has the minimum error, where error is the average distance (or squared distance) from the data points to the line. Alternatively (and equivalently), the line of “best fit” can be defined as the direction that maximizes variance. From either perspective, the resulting algorithm is the same: either way, it can be shown that the principal components are the sorted eigenvectors of the data’s covariance matrix. In practice, the principal components can be computed by eigendecomposition of the covariance matrix, or by singular value decomposition of the original data matrix.

With the principal components in hand, the dimension of the data can be reduced by projecting all data points onto the first p principal components. As an example, keeping only the first two principal components projects data onto the two-dimensional plane along which the data is most spread out—i.e., the plane along which the data has the greatest variance.

If the data has N samples in d dimensions, then the principal components decomposition of $\mathbf{X}$ is:

$$\mathbf{T}_p = \mathbf{X}\mathbf{W}_p$$

where $\mathbf{X}$ is an N-by-d matrix, $\mathbf{W}$ is a d-by-p matrix whose columns are the eigenvectors of $\mathbf{X}^T\mathbf{X}$, and $\mathbf{T}$ is a projection of $\mathbf{X}$ onto the first p principal components.

The full principal components decomposition then has $p = d$, and it is a change of coordinates to a new space of variables which are linearly uncorrelated over the data set. To reduce the number of dimensions, the transformation is performed with $p < d$, which truncates the least significant $(d - p)$ variables from the transformed $\mathbf{T}$ matrix.

The principal components transformation can be implemented using the singular value decomposition

$$\mathbf{X} = \mathbf{U}\mathbf{\Sigma}\mathbf{W}^T$$

where $\mathbf{\Sigma}$ is an n-by-p rectangular diagonal matrix whose entries are the singular values of $\mathbf{X}$; $\mathbf{U}$ is an n-by-n orthonormal matrix of left singular vectors; and $\mathbf{W}$ is a p-by-p orthonormal matrix of right singular vectors.

Using SVD, $\mathbf{T}$ can be written as

$$\mathbf{T} = \mathbf{X}\mathbf{W} = \mathbf{U}\mathbf{\Sigma}\mathbf{W}^T\mathbf{W} = \mathbf{U}\mathbf{\Sigma}$$

of course, $\mathbf{W}^T\mathbf{W} = \mathbf{I}$ since $\mathbf{W}$ is orthonormal, and the dot product of any two orthogonal vectors is 0, while a unit vector dotted with itself yields 1.

Thus, each column of $\mathbf{T}$ is given by one of the left singular vectors of $\mathbf{X}$ multiplied by the corresponding singular value. SVD relies on the eigendecomposition of the matrix $\mathbf{X}^T\mathbf{X}$ to find the singular vectors and values. However, there are algorithms that can efficiently calculate the SVD of $\mathbf{X}$ without having to form the matrix $\mathbf{X}^T\mathbf{X}$, so, in practice, this is the method used to compute PCA.

To perform dimensionality reduction, the transformed matrix $\mathbf{T}_p$ is obtained by considering only the p largest singular values and their corresponding singular vectors:

$$\mathbf{T}_p = \mathbf{U}_p \mathbf{\Sigma}_p = \mathbf{X}\mathbf{W}_p$$

This is the projection of the dataset onto the first p principal components.

I use the implementation of PCA provided by scikit-learn.[4] It first centers, but does not scale, the input data for each feature before applying SVD. It uses the LAPACK implementation of the full SVD or a randomized truncated SVD by the method of Halko[5] et al. 2009, depending on the shape of the input data and the number of components to extract.

5.2 Kernel Principal Component Analysis (KPCA)

Traditional PCA is a linear dimensionality reduction algorithm. Kernel PCA[6] extends the logic of PCA using kernel methods. Kernels can be thought of as similarity functions in Euclidean space. For example, $\mathbf{K}(\mathbf{x}, \mathbf{y}) = \mathbf{x}^T \mathbf{y}$ is the linear kernel.

Kernel methods are useful for the applications of PCA because, in general, N points cannot be linearly separated in $d < N$ dimensions. However, they can almost always be linearly separated in $d \geq N$ dimensions. That is, if we map the data set into N-dimensional space with an arbitrary function $\Phi : \mathbb{R}^d \to \mathbb{R}^N$, it is easy to construct a hyperplane that separates the data into any two arbitrary clusters.

Kernel PCA uses kernels to utilize this effect without having to do any work in the high dimensional Φ-space. In fact, Φ is never calculated explicitly, but instead we use the N-by-N kernel matrix:

$$K = k(\mathbf{x}, \mathbf{y}) = (\Phi(\mathbf{x}), \Phi(\mathbf{y})) = \Phi(\mathbf{x})^T \Phi(\mathbf{y})$$

This kernel represents the inner product space of the otherwise intractable feature space. Then, rather than calculating the principal components themselves (which would require operations in the high-dimensional Φ-space), kernel PCA computes the projections of the data onto those components. The projection of a point $\Phi(x)$ in the high-dimensional space onto the k^{th} principal component V^k is given by:

$$\mathbf{V}^{k^T} \Phi(\mathbf{x}) = \left(\sum_{i=1}^{N} \mathbf{a_i}^k \Phi(\mathbf{x_i}) \right)^T \Phi(\mathbf{x})$$

where each $\mathbf{a_i}^k$ can be found by solving the eigenvector equation $N\lambda\mathbf{a} = K\mathbf{a}$ where N is the number of points in the dataset and λ and $\mathbf{a}$ are the eigenvalues and eigenvectors of K. The eigenvectors $\mathbf{a}^k$ can be normalized by requiring that $(\mathbf{V}^k)^T \mathbf{V}^k = 1$.

[4]https://scikit-learn.org/stable/modules/generated/sklearn.decomposition.PCA
[5]https://epubs.siam.org/doi/10.1137/090771806
[6]https://en.wikipedia.org/wiki/Kernel_principal_component_analysis

I use the Kernel PCA[7] implementation from scikit-learn, which offers the different kernel options 'linear', 'poly', 'rbf', 'sigmoid', 'cosine', 'precomputed', but I use the default linear kernel.

5.3 Non-Negative Matrix Factorization

An alternative approach to dimensionality reduction is non-negative matrix factorization[8] (NMF), which aims to decompose the non-negative matrix **X** into two non-negative matrics **W** and **H**. NMF can be used to reduce the dimensionality in the following way. If **X** is an N-by-d matrix, then, when decomposed, **W** is an N-by-p matrix, and **H** is a p-by-d matrix. There are usually many ways that **X** could be decomposed, and if p can be chosen to be significantly smaller than both N and d, then **WH** is a much simpler representation of the information in **X**.

Typically, such a decomposition is found by approximation: finding **W** and **H** that minimize the error function

$$\|V - WH\|_F \,, \text{ subject to } W \geq 0, H \geq 0.$$

There are several algorithms used to find **W** and **H**. For example, Lee and Seung's[9] multiplicative update rule offers an iterative update strategy to generate the approximation and requires no special initialization for **W** and **H** other than being non-negative. Other methods use a non-negative least squares method to minimize the loss, in which each iteration alternates with one of the two matrices fixed and calculates the other using least squares.

I use the NMF algorithm implemented in scikit-learn, which has the following objective function to be minimized:

$$\begin{aligned} 0.5 * ||X - WH||_{loss}^2 \\ + alpha\_W * l1_{ratio} * n\_features * ||vec(W)||_1 \\ + alpha\_H * l1_{ratio} * n\_samples * ||vec(H)||_1 \\ + 0.5 * alpha\_W * (1 - l1_{ratio}) * n\_features * ||W||_{Fro}^2 \\ + 0.5 * alpha\_H * (1 - l1_{ratio}) * n\_samples * ||H||_{Fro}^2 \end{aligned} \tag{1}$$

where:
$||A||_{Fro}^2 = \sum_{i,j} A_{ij}^2$ (Frobenius norm)
$||vec(A)||_1 = \sum_{i,j} abs(A_{ij})$ (Elementwise L1 norm)

5.4 t-distributed Stochastic Neighbor Embedding

t-distributed Stochastic Neighbor Embedding[10] (t-SNE) is a dimensionality reduction tool typically used for visualizing high-dimensional data. It works by creating a similarity measurement between pairs of points.

[7] https://scikit-learn.org/stable/modules/generated/sklearn.decomposition.KernelPCA
[8] https://en.wikipedia.org/wiki/Non-negative_matrix_factorization
[9] https://www.cs.cmu.edu/~11755/lectures/Lee_Seung_NMF.pdf
[10] https://jmlr.org/papers/volume9/vandermaaten08a/vandermaaten08a.pdf

For $i \neq j$, define $p_{j|i}$ according to the following equation:

$$p_{j|i} = \frac{\exp(-\|\mathbf{x}_i - \mathbf{x}_j\|^2/2\sigma_i^2)}{\sum_{k\neq i}\exp(-\|\mathbf{x}_i - \mathbf{x}_k\|^2/2\sigma_i^2)}$$

As Van der Maaten and Hinton, the creators of t-SNE, explain, "The similarity of datapoint x_j to datapoint x_i is the conditional probability, $p_{j|i}$, that x_i would pick x_j as its neighbor if neighbors were picked in proportion to their probability density under a Gaussian centered at x_i."

Then, define p_{ij} as

$$p_{ij} = \frac{p_{j|i} + p_{i|j}}{2N}$$

where N is the total number of datapoints. t-SNE aims to learn a d-dimensional map that reflects these similarity measurements as well as possible. A similarity measurement q_{ij} is defined like the one above but for all pairs of points in the mapping. Then, a heavy-tailed Student t-distribution (with one degree of freedom) is used to measure similarities between low dimensional points. This allows dissimilar objects to be modeled far apart in the mapping. This strategy directly solves for the aforementioned distance issues in high-dimensional space, wherein the Euclidean distances between points become indistinguishable as the number of dimensions tends toward infinity.

The locations of the points in the map are determined by minimizing the Kullback-Leibler divergence of the distribution P from the distribution Q. That is:

$$\mathrm{KL}\,(P \parallel Q) = \sum_{i\neq j} p_{ij} \log \frac{p_{ij}}{q_{ij}}$$

This minimization is performed using gradient descent. Since this cost function is not convex, t-SNE yields different results with different initializations.

I use the implementation of t-SNE[11] from scikit-learn, which provides two algorithms. The default implementation is the faster, approximate Barnes-Hut[12] algorithm, which runs in $O(N \log N)$ time, but requires that the number of components being reduced to be 3 or fewer. The slower, exact algorithm runs in $O(N^2)$ time, and as such cannot scale to millions of examples.

The scikit-learn page also recommends using another dimensionality reduction method (like PCA) to pre-reduce the number of dimensions to a reasonable amount (e.g. 50) before applying t-SNE.

6 Experiments

I performed dimensionality reduction experiments on the famous MNIST data set, which contains 28x28 pixel grayscale images of the handwritten digits 0-9. I imported the dataset using `fetch_openml` from sklearn. Each image is a 28x28 matrix with integer values 0-255 each representing the brightness of the pixel at

[11]https://scikit-learn.org/stable/modules/generated/sklearn.manifold.TSNE
[12]https://arxiv.org/pdf/1301.3342.pdf

a specific location in the image. I first scaled the data into the range $[0, 1]$ by dividing by 255. Then, I reshaped the data from shape (N, 28, 28) to (N, 784), where N is the number of data points in the set.

Then, I divided the data and its labels into a 5000 point training set, a 1000 point validation set, and a 1000 point test set. This is standard machine learning practice: models can only be fairly evaluated on data they have not seen before (i.e., the validation and testing sets). Optimization is performed on the validation set so that the test set is only used for final testing to report results.

Next, I wrote what I call "the k-means classifie". It's quite similar to the naive k-Nearest Neighbors classifier that is a common baseline for machine learning tasks like this, but mine is more idiotic. Regardless, it gives an accuracy reading that can be used to create "degradation curves", which show the rate at which data is lost by dimensionality reduction as we reduce to fewer and fewer components.

The classifier first clusters the data into the appropriate number of classes (in the case of the MNIST data, it's 10 clusters). However, the cluster that contains datapoints representing a digit will be labeled as a different cluster (for example, images of zeros might be clustered into cluster 7).

So, I wrote the function `build_mapping` that decides what clusters are which classes. It builds a confusion matrix, where each row represents one cluster, and each column represents one class. Then, it assumes that each cluster represents the class for which it has the most examples (which is the position of the largest value in that cluster's row). This often leads to the rather awkward result where multiple clusters decide that they represent the same class. For example, clusters 2 and 4 might both believe that they represent images of sevens, because they each have more sevens than they have anything else. The classifier does nothing to handle this case, and asserts instead that some digits are actually missing from the dataset—for example, it might predict no nines as a result.

Then, I use my function `assign_labels` to use the generated mapping to predict the labels of each point in the dataset. Lastly, my function `accuracy` calculates the proportion of the dataset that is correctly classified.

When testing the model, I track:

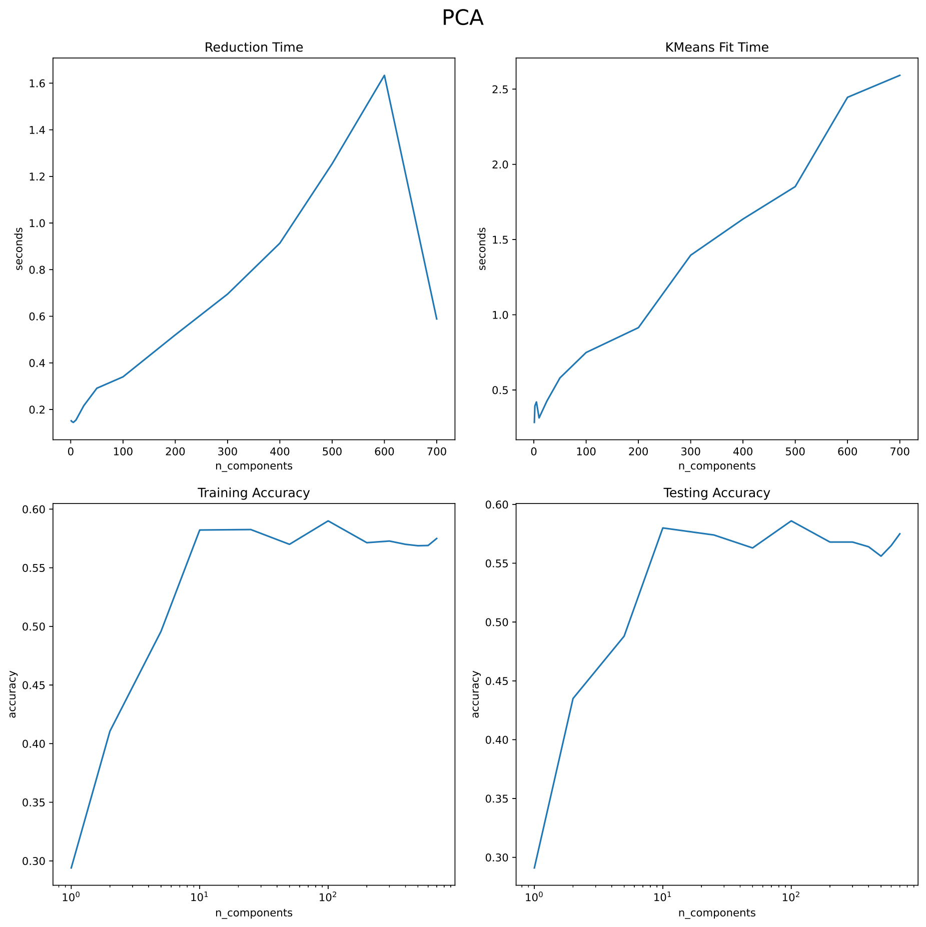
1. the time it takes the dimensionality reducer to transform the data
2. the time it takes k-means to fit to the transformed data
3. the classification accuracy on the training set
4. the classification accuracy on the testing set

I test the following dimensionality reduction techniques: PCA, Kernel PCA, NMF, and t-SNE, letting the number of components reduced to vary accordingly. For PCA, Kernel PCA, and NMF, `n_components` was chosen from the set [1, 2, 5, 10, 25, 50, 100, 200, 300, 400, 500, 600, 700]. t-SNE, however, uses the Barnes-Hut algorithm, which limits reduction to 1, 2, or 3 components, since t-SNE is primarily for visualization.

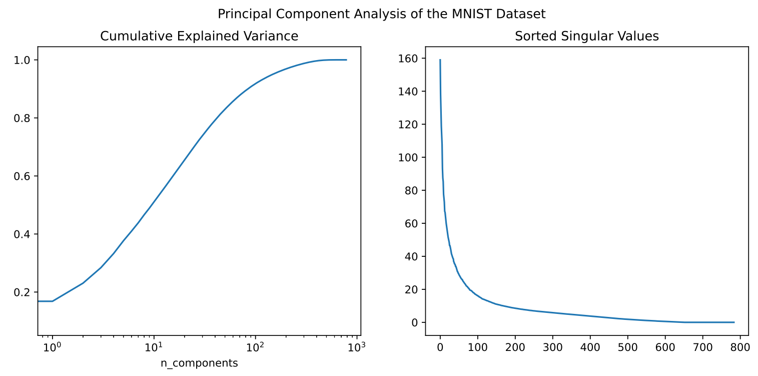
As a baseline, I tested the classifier with no dimensionality reduction and attained the following results:

- Fitting to training data took 2.76 seconds
- Training accuracy: 0.5716
- Predicting labels for test data took 0.02194166 seconds
- Testing accuracy: 0.568

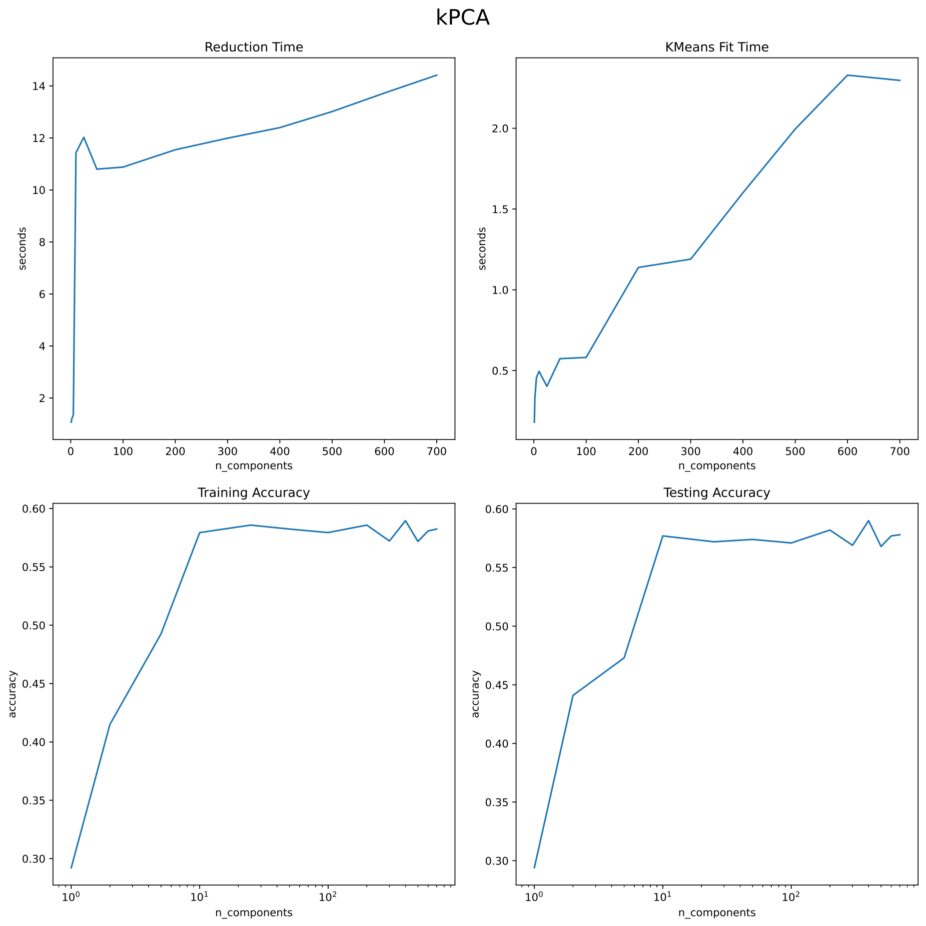
At the outset of this project, I wanted to see if any dimensionality reduction algorithms could transform the data in such a way that the clusters are more easily separable and therefore the classifier more accurate. That is, I wanted to see if the transformation can reduce the distance between data points of the same class and increase the distance between data points of different classes. But let's not get ahead of ourselves: the first result we should expect is that, as the number of components in the reduced data set decreases, accuracy should decrease as information is lost. I refer to this phenomenon as "degradation". This is exactly what we see for the classifiers, but the degradation curves also have some interesting quirks.



Traditional PCA shows generally what we expect. I believe the drop in the reduction time for `n_components`=700 is attributable to a change in the choice of SVD algorithm, as the choice is made by the PCA algorithm automatically based on the shape of the input. The training accuracy and test accuracy flatline when we use about 10 or more components. However, for fewer than 10 components, the accuracy rapidly degrades. The best accuracy attained is no better than that from the baseline, but this experiment does show that the baseline accuracy can be attained with radically fewer features. Also, note that PCA is quite fast. The transformations take on the order of 0-2 seconds—shorter than performing k-means on the transformed data.

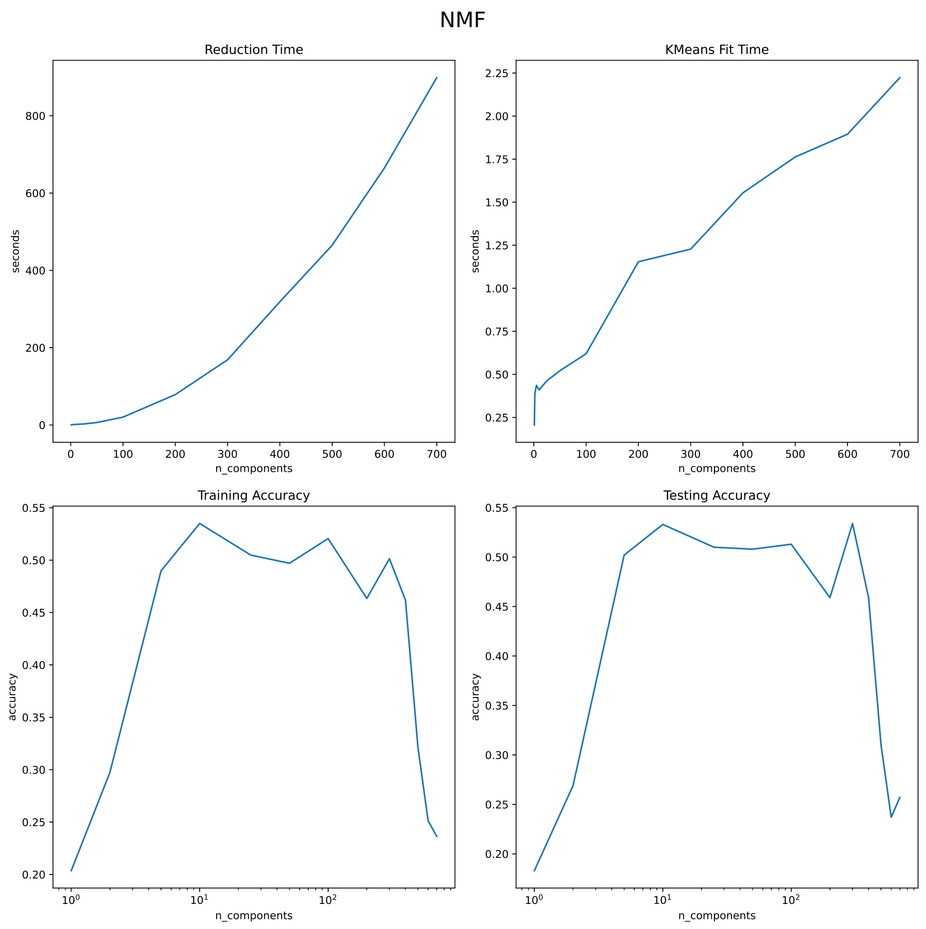


To better understand what's going on, I plotted the cumulative explained variance curve, which shows what proportion of the variance in the full dataset is captured by the first `n_components`. Interestingly, the first 10 components explain only about half ($\sim$0.49) of the variance in the data set, and yet using only 10 performs about as well as using the entire data set on the classification task. It's possible that this is a quirk of the k-means classifier; a better classifier might be able to take advantage of the additional data present in more components and could show better performance. Additionally, most singular values are small compared to the largest singular values. In general, the most important components represent a significantly outsized proportion of the meaning in the data. This is a good indication that the intrinsic dimension of the MNIST data is much lower than 784.



Next, I tested Kernel PCA with a similar set of experiments. The reduction time is almost instantaneous for 1, 2, and 5 components, although I'm not sure why. However, after that, it stays in a pretty tight range between ∼11 and ∼14 seconds, while increasing slightly as the number of components increases. The KMeans Fit Time curve looks very similar to that of PCA, which is what we expect; in both cases, k-means is being asked to fit a matrix of the same shape.

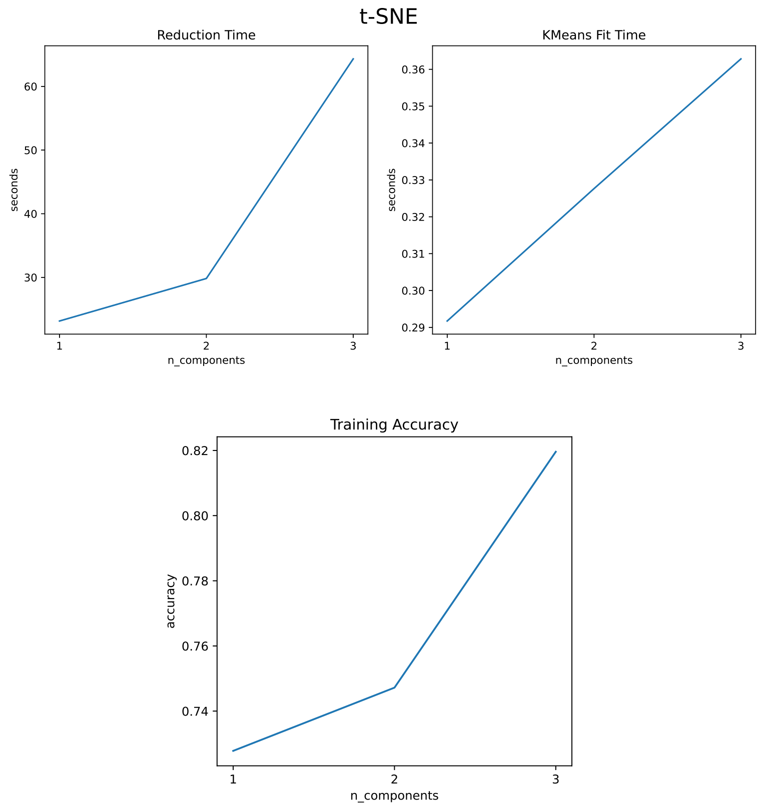
The accuracy curves are also almost identical. It seems that, in this instance, when compared to PCA, Kernel PCA does not offer an advantage against degradation with fewer components and does not offer better accuracy with more components. This could again be a result of the classifier. It's possible that Kernel PCA does offer an edge that only a more sophisticated classifier could take advantage of.



Next, I tested NMF, which took much longer. It's worth noting that the algorithm did not converge for `n_components` ≥ 10. Maximum iterations is set to 200 by default, and I didn't change it, as even that number of iterations took a long time. The KMeans curve again looks the same as before—no surprises there. The accuracy curves have an interesting shape, however. The best accuracy attained is less than 0.55, so it's a little bit worse than doing nothing. However, if I had let the algorithm converge and generated a better matrix approximation, it's possible that the accuracy might have been the same. Even though the algorithm converges for small numbers of components, these approximations are not very good, and there is significant degradation. After reaching a peak around 10 components, the accuracy slowly drops, then drops off massively for 600 and 700 components. I suspect that this is due to convergence issues—when using more components, the algorithm needs to run for much longer to generate a good approximation of the original matrix. Since the approximations get worse, the accuracy decreases.

Lastly, I tested t-SNE, which required some unique considerations. t-SNE is primarily used for data visualization, and is arguably out-of-place in this task, but I wanted to try it anyway. In scikit-learn, t-SNE is implemented with the Barnes-Hut algorithm, which only supports 1, 2, or 3 components for the transformed data. Additionally, t-SNE can only transform data that it has been fit to, so it cannot be used on unseen data for this task. However, I still was curious to see how it would perform on the training data.

As suggested by scikit-learn, I used PCA to pre-reduce the data into 50 dimensions before applying t-SNE.



With so few components, K-Means converged almost instantly, but the time increased linearly with the number of components. The reduction time is not insignificant, but much less expensive than NMF. The most striking result, however, is the accuracy. Reduction to 1, 2, or 3 components actually increased accuracy on the training set significantly. For 3 components, the accuracy reaches 82%, where the baseline was 57%. That's an enormous improvement.

I hypothesize that this result is attributable to t-SNE's use of the heavy-tailed t-distribution to create additional separation between dissimilar points when the mapping is performed. This suggests that the issues with Euclidean distance in high-dimensional space are an obstacle to classification on the MNIST data set. The features of t-SNE designed to solve that issue thus improve classifier performance.

7 Conclusion

In this project, I set out to study different techniques for dimensionality reduction. The best of these techniques perform feature extraction by creating mappings or projections to fewer variables (i.e., lower dimensional space) that preserve much of the information intrinsic to the data set.

Experimentally, I confirmed that the MNIST dataset has a quite low intrinsic dimension (about 10), and that using fewer components than this intrinsic dimension causes performance to degrade quite quickly, while using more components improves performance only marginally. Additionally, I was able to attain a positive result: dimensionality reduction using t-SNE actually improved performance on the classification task by creating additional separation between the natural clusters in the data.