Computational and Theoretical Analysis of Novel Dimensionality Reduction Algorithms in Data Mining

Brandon Guo
Monta Vista High School
Cupertino, California

Contents

Acknowledgement

I would like to thank Professor Guangliang Chen of San Jose State University, who graciously offered me guidance and support in the creation of this article. Additionally, he invited me to study his course materials in order to better establish a foundation for the material presented here. Without his assistance, I would have neither the intellectual maturity nor motivation to pursue such a topic.

1 Introduction

Data mining and analysis are ubiquitous skills that are used in practically every modern industry. The sheer quantity of data available to industrial experts provides unbounded access, and the existence of advanced programming languages and packages allows for simple development of models and predictions.

However, it seems as with two steps forward data mining has taken one step back. Simply using the data presented, unfiltered, can be dangerous for a few reasons. Firstly, the absence of dimension reduction makes computer training more memory-intensive and less efficient (Wang and Carreria-Perin 2014). Additionally, from an industry perspective, having fewer dimensions to describe a result can be a lot more instructive as to which factors influence changes in the response (Burges 2010). Most importantly, however, is the effect dimensionality reduction has on inference performance. Inferential models perform better with reduced dimensions in two ways: faster convergence and prevention of overfitting (Srivastava et al. 2014). Stone (1982) proved that under the assumption of regular data (independently and identically distributed) the rate of convergence ρ varies with $m^{-x/(2x+d)}$, where m is the number of training samples, the regression function is x times differentiable, and d is the number of dimensions.

For large datasets, such as gene expression datasets, the value of d is in the thousands (An and Chen 2006), implying $d \gg x$. Thus, by simply reducing the number of dimensions d into d' where $d' = \frac{d}{k}$:

$$\rho \propto m^{\frac{-x}{(2x+d/k)}} \approx (m^{(-x/(2x+d))})^{\frac{1}{k}} = \sqrt[k]{m^{\frac{-x}{(2x+d)}}} \tag{1}$$

which is a big difference, especially for large $m^{-x/(2x+d)}$, k, or either. Since reducing m would hurt our predictive capabilities (Blumer et al. 1989; Ehrenfeucht et al. 1989), we are uniquely interested in reducing d.

Secondly, overfitting is a problem that arises with numerous dimensions in the data (Srivastava 2014). Especially for high-dimensional, low sample size problems (HDLSS) such as gene data, overfitting has been shown to be a problem with model training (Subramaniam 2013; Simon et al. 2003). The more dimensions that exist in the data, the more sparse the points tend to be. Simon et al. (2003) warns that training with such large dimensions could lead to deceiving and misreported accuracy results, and subpar performance on novel data. Simply by increasing the dimension by one, the volume of the datapoints increases exponentially and can become much harder to predict; this is otherwise known as the *curse of dimensionality* (Bellman 1957, Keogh et. al 2017).

Simply wishing to reduce the number of dimensions is one thing, however. How does one actually go about choosing which dimensions to keep, and which to ignore? Concretely, how does an algorithm ensure that dimensions are chosen in a way that preserves the behaviors of meaningful features over less meaningful features? What determines if a dimension is "meaningful," anyway? Ultimately, the answer to this question lies in the algorithms themselves: different dimension-dropping techniques approach the curse of dimensionality differently.

2 Algorithms

This article will evaluate a series of dimensionality reduction techniques. This section will review the theories and motivations regarding such algorithms. Based on the intuition established in this section, one may be able to predict the shortcomings of certain algorithms over others. Here are the algorithms that are to be elaborated upon in this section, and used in practice in the next:

- Principal Component Analysis (PCA)
- Nonnegative Matrix Factorization (NMF)
- Kernel Component Analysis (KCA)
- Independent Component Analysis (ICA)
- t-Distributed Stochastic Neighbor Embedding (t-SNE)

2.1 Overview: Notation and Style

Algorithms will be presented and described with respect to the most popular dimensionality reduction algorithm, PCA (Principal Component Analysis), simply because it is well-known and intuitive. As a result, PCA will be wholly derived and presented with an assumption of no prior knowledge in reduction techniques, as means of elucidating future algorithms and their comparisons.

Unless otherwise stated, matrices will be presented with capital, boldface letters: for example $\mathbf{A}$. Vectors will be presented with lowercase, boldface letters, such as $\mathbf{u}$. Eigenvalues for certain techniques that solve eigenproblems will be marked with λ.

2.2 Principal Component Analysis

Principal Component Analysis (PCA) is one of the most popular techniques used to reduce the dimensions of a certain training set (Shlens 2005). The idea of the algorithm is, based on a fixed number of dimensions, to find a new set of variables of that size that still retains a certain amount of the original information of the dataset. Concretely, given a dataset with d dimensions, the problem is to find $k < d$ linear projections of the datapoints that maximize $x\%$ of the variance, where k, and hence x is ultimately up to the user.

Prior to PCA, there exist m d-dimensional vectors that represent each of the training points in the data. We wish to project these vectors into k dimensions, or the *principal components* of the data. The principal components are then ordered by the amount of variance they capture, so the k^{th} principal component captures the k^{th} least amount of variance.

Prior to any computation, we are going to assume that the data has been centered (such that the mean is 0), with n entries per column (or rows). This shouldn't change anything, since one would just shift the entire sample, but it

does simplify calculations. Since the data is centered, the dataset $\mathbf{X}$ is equal to its deviation matrix $\mathbf{X_d}$ since the mean of each column is 0. By definition, we can calculate the covariance matrix $\mathbf{C}$ of $\mathbf{X}$:

$$\mathbf{C} = \frac{1}{n}\mathbf{X_d^T X_d} = \frac{1}{n}\mathbf{X^T X} \implies \mathbf{X^T X} = n\mathbf{C} \tag{2}$$

which is a valuable insight for later computations.

To mathematically represent the principal components, we introduce a matrix $\mathbf{W}$ that is composed of unit vectors $\mathbf{w_i}$ that lie directly on the components of interest. Notice that a unit vector suffices simply because we are projecting the data onto its line, which stretches the entire plane, and we have the nice property that $||\mathbf{w_i}|| = 1$.

To understand the goal and motivation behind PCA, we will first look at the problem in the one dimensional case. Concretely, given just one principal component $\vec{w}$, for each data vector $\vec{x_i}$, we can project the vector onto the line containing $\vec{w}$ by computing $\vec{w}(\vec{x_i} \cdot \vec{w})$. [1] We call this new vector the projection of the vector, which by nature is an estimation and must have some error. To quantify this error, we square the magnitude of the vector difference $\vec{x_i}$ and the projected vector:

$$||\vec{x_i} - (\vec{x_i} \cdot \vec{w})\vec{w}||^2 = ||\vec{x_i}||^2 - 2(\vec{x_i} \cdot \vec{w}) + (\vec{x_i} \cdot \vec{w})^2||\vec{w}||^2 \tag{3}$$

Since $\vec{w}$ is a unit vector, $||\vec{w}||^2 = 1$; hence, Equation (3) can be simplified nicely into

$$||\vec{x_i}||^2 - (\vec{x_i} \cdot \vec{w}) \tag{4}$$

for each value of $\vec{w}$. Since $\vec{x_i}$ cannot be changed (it is the nature of the data), it suffices to maximize the second term to minimize the individual errors. Holistically, the problem is maximizing the sums of this term, or

$$\frac{\sum_{i=1}^{n}(\vec{x_i} \cdot \vec{w})^2}{n} \tag{5}$$

Which is the sample mean of the squares of $\vec{x_i} \cdot \vec{w}$. However, if we remember that the mean of $\vec{x_i}$ and thus $\vec{x_i} \cdot \vec{w}$ are both 0, then we can relate the result from Equation (5) to the variance of all the vectors $\vec{x_i} \cdot \vec{w}$. Concretely,

$$\hat{\sigma}^2 = \frac{\sum_{i=1}^{n}(\vec{x_i} \cdot \vec{w} - \mu)}{n} = \frac{1}{n}\sum_{i=1}^{n}(\vec{x_i} \cdot \vec{w})^2 \tag{6}$$

because $\mu = 0$. Therefore, maximizing Equation (5) is the same thing as maximizing the $\hat{\sigma}^2$ (of $\vec{x_i} \cdot \vec{w}$). For each dimension $\vec{w}$, we wish to minimize the cumulative sum of all the variances. To accomplish this, we will stack x_i and w_i as follows: We define $\mathbf{X} \in \mathbb{R}^{a \times b}$ as the data matrix given and $\mathbf{W} \in \mathbb{R}^{b \times p}$,

[1] We can check that this indeed gives us the correct magnitude by the definition of the dot product: $||\vec{a} \cdot \vec{b}|| = (||a||)(||b||)\cos\theta$, which is indeed the component of $\vec{a}$ that lies on the axis defined by $\vec{b}$.

where p is the number of dimensions we eventually want to reduce the problem into. We can then represent the variance in question with matrices:

$$\hat{\sigma}^2 = \frac{1}{n}(\mathbf{XW})^T(\mathbf{XW}) = \frac{1}{n}\mathbf{W}^T\mathbf{X}^T\mathbf{XW} \tag{7}$$

Applying the result of Equation (2), Equation (8) can be represented by only the directions and the covariance matrix $\mathbf{C}$ and the directions $\mathbf{W}$:

$$\hat{\sigma}^2 = \mathbf{W}^T\mathbf{CW} \tag{8}$$

We want to find a way to maximize the result from (8), but we need to establish some limiting condition, because currently, some large $\mathbf{W}$ would make $\hat{\sigma}^2$ unboundedly large, which cannot be true. However, since each of the $\vec{w}$ that make up $\mathbf{W}$ are unit vectors, or $||\vec{w}||^2 = 1$, it follows that $\mathbf{W}^T\mathbf{W} = \mathbf{1}$, which can be written as a function $g(x) = \mathbf{W}^T\mathbf{W} - \mathbf{1}$. Given this constraint, we can then construct the Lagrange expression $\mathcal{L}$ with the Lagrange multiplier λ as such:

$$\mathcal{L}(\lambda, \mathbf{W}) = \mathbf{W}^T\mathbf{CW} - \lambda \cdot g(x) \tag{9}$$

Note that the $\mathbf{1}$ in the definition of $g(x)$ represents the $p \times p$ identity matrix. When we take the partials of $\mathcal{L}$, we know that the extrema of the variance lie in the zeros of the partial derivatives. Taking partials with respect to λ and $\mathbf{W}$ yields

$$\frac{\partial \mathcal{L}}{\partial \lambda} = \mathbf{W}^T\mathbf{W} - \mathbf{1} \tag{10}$$

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}} = 2\mathbf{CW} - 2\lambda\mathbf{W} \tag{11}$$

Both partials are equal to zero when the variance is optimized, so we have

$$\mathbf{W}^T\mathbf{W} = 1 \tag{12}$$

$$\mathbf{CW} = \lambda\mathbf{W} \tag{13}$$

Equation (12) is just a restatement of the constraint established earlier, but Equation (13) is just a restatement of the eigenvector problem, where $\mathbf{W}$ is the eigenvector associated with eigenvalue λ. We know that the covariance matrix $\mathbf{C}$ has dimensions $b \times b$, so therefore there should be at most b distinct eigenvectors, and thus eigenvalues. If we combine Equation (13) with Equation (8) with hopes to maximize variance:

$$\hat{\sigma}^2 = \mathbf{W}^T\mathbf{CW} = \lambda\mathbf{W}^T\mathbf{W} \tag{14}$$

From this, we can get two telling results. First, all the eigenvectors should be orthogonal to each other in b-space, which mean that they should span the entire plane. This means, in theory (though not very helpful in practice), projecting the data into d dimensions where d is the original number of dimensions with PCA would still capture every single of the (at most) b eigenvectors that make

up the original space. Thus, the resulting projection would still capture 100 percent of all the variance. This is a good sanity check that all the information in the data is still retained in projection, but it is up to the user to decide how much is needed.

Secondly, Equation (14) shows that the variance we wish to maximize is directly proportional to the value of λ, so there exists a very systematic way to choose dimensions that capture the most variance. Since $\mathbf{C}$ has all positive values (by definition of a covariance matrix), we know that all λ must be positive. Therefore, one can simply order the resulting eigenvalues by magnitude, since the largest eigenvalue should lead to the largest variance. When the user wishes to choose the best n components, one would simply return the resulting n eigenvectors that result in the n largest λ to best represent the data. The first principal component would be the directional vector $\mathbf{W}$ that has the largest magnitude λ solution, and the second $\mathbf{W}$ with the second largest λ, and so on.

Ultimately, it is up to the user regarding how many principal components p they wish to include in their model testing. Through deriving the validity of PCA, we have developed a bit of an intuition regarding how PCA performs with respect to the number of components the user chooses. At zero components, trivially, none of the variance is captured. The difference between the variance at $p = 1$ and $p = 0$ should be the largest. Between each $p = k$ and $p = k+1$, the variance should increase by a value that is no larger than the variance difference between $p = k - 1$ and $p = k$. This change in variance should get smaller and smaller, such that if we define a function of the percent variance captured by our dimensions as a function of the number of components used as $F(k)$, we suspect that $\lim_{p\to\infty} F(k) = 100$, and $\frac{d^2}{dk^2}F(k) < 0$ for all $k \geq 0$.

Thus, PCA is simply a process by which

1. The user chooses the number of dimensions, d to reduce the problem into
2. Equation (13) is solved with the covariance of the data matrix
3. The largest d eigenvalue solutions to (13) are selected.
4. Using these eigenvalues, all d projection vectors W are selected.
5. The data is projected onto these new W vectors as axes

2.3 Nonnegative Matrix Factorization

Like Principal Component Analysis, Nonnegative Matrix Factorization (NMF) is a method of decomposing in an unsupervised manner, a large matrix of data (Foldilak and Young 1995). NMF seeks to solve a matrix multiplication problem. Given a data matrix $\mathbf{X}$ (hence the name of the algorithm, all elements of $\mathbf{X}$ are nonnegative, we wish to approximate two factor matrices $\mathbf{A}$ and $\mathbf{B}$ such that

$$\mathbf{X} \approx \mathbf{AB} \tag{15}$$

, and $\mathbf{X} \in \mathbb{R}^{a\times b}, \mathbf{A} \in \mathbb{R}^{a\times n}, \mathbf{B} \in \mathbb{R}^{n\times b}$, and $n < a, b$, resulting in a compressed data matrix.

To understand the significance of this reconstruction, we look at one data vector x_i of the training data. We can then express $x_i = Ab_i$, where b_i is a similarly defined vector of matrix $\mathbf{B}$. By that definition, every x_i is really just a linear combination of $\mathbf{A}$ and the "weights" defined by b_i. The problem then becomes to find the most accurate $\mathbf{A}$ and $\mathbf{B}$ for learning.

As with many optimization problems, we define a cost function that measures how accurate the approximation is. The value we wish to minimize is simply the discrepancy between the two matrices. Thus we define a cost function C as

$$C = ||\mathbf{X} - \mathbf{AB}||^2 \tag{16}$$

To optimize C, Lee and Seung (2001) proposed an upx rule to minimize this cost function. For each iteration,

$$\mathbf{A} \leftarrow \frac{\mathbf{XB}^T}{\mathbf{ABB}^T} \circ \mathbf{A} \tag{17}$$

$$\mathbf{B} \leftarrow \frac{\mathbf{A}^T\mathbf{X}}{\mathbf{A}^T\mathbf{AB}} \circ \mathbf{B} \tag{18}$$

where $\circ$ is the operation for element-wise multiplication of two matrices. The validity of this update rule to ensure $\mathbf{A}$ and $\mathbf{B}$ are non-decreasing is proven by Burred (2014) to show that such updating will result in a local minima of C. Since it has already been established that all matrices of Equation (15) are nonnegative, it follows that any updated versions of $\mathbf{A}$ and $\mathbf{B}$ are nonnegative, as well. Something else worth checking is the equality case of Equation (15). If we let $\mathbf{X} = \mathbf{AB}$, it follows that the multiplicative update factor is simply the identity matrix.

Because the cost function C is decreasing for every iteration of the multiplicative updating, the final result is the product that is closest to the true value of $\mathbf{X}$. Because n, the new dimensions in both vectors $\mathbf{A}$ and $\mathbf{B}$ is smaller than the values of a and b, the number of dimensions to train is reduced, and this leads to a more concise dataset and fewer dimensions to have to map for an algorithm.

2.4 Kernel Component Analysis

Kernel Component Analysis is an extension of the Principal Component Analysis technique that seeks to deal with the potential nonlinear relationships that exist in the data which otherwise would not be accurately accommodated for by PCA. The key, seemingly counter-intuitive step of Kernel PCA is to map the dataset into *more dimensions*, and then use kernels to find the principal components of the new, translated data. Obviously, it seems like this type of analysis is a little more inefficient, simply because data is translated into more dimensions, but this allows for accurate mapping for data that is not linearly defined (more often the case with data in nature).

To being Kernel Analysis, we will start similarly as we did during PCA, by establishing some conditions as well as the covariance matrix. In PCA, we

intialized our data vectors as x_i, which were d dimensional. For Kernel, we will define a function $\gamma(x_i)$ as a mapping of each x_i to a larger feature space $\mathbb{R}^f$ where $f > d$. Given this, we define the new covariance matrix

$$\mathbf{C} = \frac{1}{n}\sum_{i=1}^{n}\gamma(x_i)(\gamma(x_i)^T) \tag{19}$$

By the result from Equation (13), we are interested in solving the eigenproblem of the covariance matrix:

$$\mathbf{C}\mathbf{v} = \lambda\mathbf{v} \tag{20}$$

After plugging in the value of $\mathbf{C}$ manipulating for the eigenvector, we find that

$$\mathbf{v} = \frac{1}{\lambda n}\sum_{i=1}^{n}(\gamma(x_i)\cdot\gamma(x_i)^T\cdot\mathbf{v}) \tag{21}$$

We can then define $a_{n,i}$ iteratively for each eigenvalue λ_i and its eigenvector $\mathbf{v}$

$$a_{n,i}\frac{1}{\lambda_i N}(\gamma(x_i))\cdot\mathbf{v} \tag{22}$$

With that substitution, we can write Equation (21) in a more simple fashion as

$$\frac{1}{N}\sum_{i=1}^{N}\gamma(x_i)\gamma(x_i)^T\sum_{j=1}^{N}a_{k,j}\gamma(x_j) = \lambda_i\sum_{k=1}^{N}a_{k,i}\gamma(x_i) \tag{23}$$

Multiplying both sides of Equation (23) by $\gamma(x_h)^T$, where x_h represents another dimension, still bounded by $1 \leq h \leq N$, Equation (23) becomes

$$\frac{1}{N}\sum_{i=1}^{N}\kappa(x_h,x_i)\sum_{k=1}^{N}a_{n,k}\kappa(x_i,x_k) = \lambda_n\sum_{i=1}^{N}a_{n,i} \tag{24}$$

$$\kappa(x_h,x_i)\kappa(x_h,x_i) = \gamma(x_h)^T\gamma(x_i) \tag{25}$$

If we stack all the kernels κ into one large matrix $\mathbf{K}$, and we stack the $a_{n,i}$ into a matrix $\mathbf{A_n}$, we can construct a matrix equation from the result in Equation (24).

$$\frac{1}{N}\mathbf{K}^2\mathbf{A_n} = \lambda_n\mathbf{K}\mathbf{A_n} \tag{26}$$

When checking the dimensions of this equation, it is worth noting that stacking $a_{n,i}$ row-wise does not multiply properly in Equation. However, $\mathbf{A_n} = [a_{n,1},\ldots,a_{n,N}]^T$ preserves the dimensions as well of the logic of the equation. This equation nicely simplifies to

$$\mathbf{K}\mathbf{A_n} = N\lambda_n\mathbf{A_n} \tag{27}$$

which is just the eigenproblem for the matrix $\mathbf{K}$! What this means is given a function for the kernel function κ, one can solve the dimension reduction problem with a similar logic as PCA. In the last section, we proved that taking the largest eigenvectors of this problem would yield to the most variance captured in the fewest dimensions. The proof of this fact in multiple dimensions is very similar here through the dimension mapping function γ. Once the eigenvalues are achieved, the user has the liberty to choose the number of eigenvalues to keep based on the amount of variance intended. The user can also choose the type of kernel function they would like, namely how to perform the κ transformation, such as Gaussian, exponential, and Poisson. The testing with this algorithm at the end of the review will use the Gaussian kernel transformation.

2.5 Independent Component Analysis

Independent Component Analysis is a dimension reduction problem that seeks to separate independent sources that are mixed together in the data. The most commonly used analogy for this technique is the *cocktail party problem*, in which n speakers pick up n different voices, but each speaker picks up some confounding group of voices. The goal, therefore, is to separate the recorded data into as independent of observations as possible.

To quantify the problem, suppose we have a n individual sources $s_1, s_2, \ldots, s_n$ that make up the vector $\mathbf{s}$, but we observe, from n "microphones", n different mixtures of the data $\mathbf{x}$. Then, we define the mixing matrix $\mathbf{A}$ such that

$$\mathbf{x} = \mathbf{A}\mathbf{s} \tag{28}$$

Because $\mathbf{x}$ and $\mathbf{s}$ have the same dimensions, we know that $\mathbf{A}$ is a square matrix, and hence has an inverse. Given that, we can define

$$\mathbf{W} = \mathbf{A}^{-1} \implies \mathbf{s} = \mathbf{W}\mathbf{x} \tag{29}$$

This means, that if we can estimate the values of $\mathbf{A}$ and $\mathbf{x}$, then we can find the inverse of $\mathbf{A}$ to systematically compute each of the individual sources $\mathbf{s}$. How do we solve for $\mathbf{W}$? The goal of Independent Component Analysis is: given Equation (28), to choose a mixing matrix $\mathbf{A}$ such that the sequences resulting from the calculation $\mathbf{W}\mathbf{x}$ are most statistically independent.

We define independence as a pairwise measure of how the values of random variable x_1 affects the value of x_2. Concretely, if we let $\rho(x_1, x_2)$ as the joint probability density function (pdf) of a random x_1, x_2. We can then define the two marginal probability distributions $p(x_1)$ and $p(x_2)$ as such:

$$p(x_1) = \int \rho(x_1, x_2) dx_1 \tag{30}$$

$$p(x_2) = \int \rho(x_1, x_2) dx_2 \tag{31}$$

Indpendence between x_1 and x_2 holds if and only if:

$$\rho(x_1, x_2) = p(x_1)p(x_2) \tag{32}$$

The result of ICA constrains itself to returning independent components, as shown in the definition above. Another key observation necessary to performing the analysis is to assume that the data is non-Gaussian. The reason for this is because Gaussian data has a pdf of

$$\rho(x_1, x_2) = \frac{1}{2\pi} e^{-\frac{x_1^2 + x_2^2}{2}} \tag{33}$$

However, since this data is completely symmetric, there are no direction vectors that can be used to estimate the vector $\mathbf{A}$, which invalidates the problem. This exception to the rule is definitely nontrivial, because data can often tend to a normal curve.

Because one cannot change how "Gaussian" data is, but rather simply be aware of its tendency, *kurtosis* will be used as a measure of how normal the distribution of data is.

Kurtosis, or sometimes referred as the fourth moment (of the generating function), [2]

of x is defined as $K(x)$

$$K(x) = E(x^4) - 3 \tag{34}$$

which is nonzero when x is not Gaussian, which is what we want.

Not only is it essential that non-Gaussianity occurs for the sake of the independence argument, but non-Gaussianity is actually one metric by which the ICA model can be optimized.

Indeed, the ICA technique analyzed will aim to maximize the absolute value of kurtosis. The problem searches for solutions in the form $\mathbf{s} = \mathbf{w}^T x$, where x composes the whitened data matrix. This is the same matrix problem we were trying to solve in (29). Hyvrinen and Oja. (1997) maximizes the kurtosis by calculating the gradient of the kurtosis function with respect to w, leading to the following update rule for the weights vectors $\mathbf{w}$:

$$w \equiv E(\mathbf{x}(\mathbf{w^T x})^3) - 3||\mathbf{w}||^2\mathbf{w} \tag{35}$$

until convergence. This descent method of maximizing kurtosis is more colloquially known as "FastICA". Since this final matrix $\mathbf{W}$ is as independent as possible, the projections of the data via the mixing matrix $\mathbf{s}$ should capture as much variance as possible. Obviously, the most independent n projections of ICA need not be related to the first n projections given by PCA, since they are used to describe two different nuances of the data.

[2] The moment generating function is a polynomial used to describe analytically the probability distribution of a random variable X. For all integral, positive t, we define the t^{th} term of the polynomial as $E[e^{tX}]$. By the Taylor series, one can expand e in terms of $t!$ for each term. This is extremely useful because it yields the nice property that the n^{th} derivative of the moment function, evaluated at $t = 0$, yields the n^{th} moment of the random variable. $n = 1$ yields the mean, $n = 2$ yields the standard deviation, and thus $n = 4$ is the kurtosis of X.

2.6 t-Distributed Stochastic Neighbor Embedding

t-SNE is a visualization of multi-dimensional data that improves on the related algorithm SNE. SNE, or Stochastic Neighbor Embedding, maps data from multiple dimensions to a reduced number of dimensions by analyzing the similarities and differences of data observations in a pairwise fashion. Developed by Hinton and Roweis (2002), this algorithm is used to visualize data that may not be contingent on one location or element (in the case of PCA, the eigenvectors). Additionally, a probabilistic framework is used as means of subverting certain obvious differences between two inherently similar data values.

Our problem is very general one, given a dataset of $x_i \in \mathbb{R}^n$, we are interested in computing $y_i \in \mathbb{R}^d$, where $n \gg d$.

Before describing the motivation behind the t-Distribution, it is wise to first explain the math of the SNE algorithm. The algorithm converts a high-dimensional subspace into conditional probabilities. For each point x_i, if that point picked a neighbor point based on the distribution of a Gaussian model centered at itself, the probability it picks a specific other point x_j is defined as:

$$p_{j|i} = \frac{\exp(-||x_i - x_j||^2/2\sigma_i^2)}{\sum_{k \neq i} \exp(-||x_i - x_k||^2/2\sigma_i^2)} \tag{36}$$

The goal of the problem is to map x_i to y_i such that the conditional probability of $y_{i,j}$, which we call $q_{j|i}$, is as identical to $p_{j|i}$ as possible. $q_{j|i}$ will be calculated similarly to $p_{j|i}$, except we will set the value of $\sigma_i = \frac{1}{\sqrt{2}}$ for simplicity. We can do this according to Hinton (2002) because for small dimensions, fixing the variance of a dataset will simply scale the values, which does not affect our probability calculations in this problem. With that assumption, we can then establish:

$$q_{j|i} = \frac{\exp(-||y_i - y_j||^2)}{\sum_{k \neq i} \exp(-||y_i - y_k||^2)} \tag{37}$$

To compare p and q, one can use relative entropy, or the Kullback-Liebler divergence to establish the cost function.

$$C(p, q) = \sum_i \sum_j p_{j|i} \log \frac{p_{j|i}}{q_{j|i}} \tag{38}$$

Gradient descent can then be performed on the cost function by taking the partial with respect to y_i:

$$\frac{\partial C}{\partial y_i} = 2 \sum_j (p_{j|i} - q_{j|i} + p_{i|j} - q_{i|j})(y_i - y_j) \tag{39}$$

From a physics perspective, the gradient is the resultant force on the point y_i if it were connected to each other point by Hookian springs. The spring force, is proportional to the displacement of the spring, calculated by $y_i - y_j$. The length of the spring then affects the spring constant, calculated by $p_{j|i} - q_{j|i} + p_{i|j} - q_{i|j}$.

Summing all these vectors would yield the net force on the point y_i. To ascertain this, a large value of the first multiple means that y_i and y_j are very close, and it is a well-known result that a shorter spring yields a large spring constant force. [3]

However, the optimization process of the SNE algorithm suffers from something called the "crowding problem", first observed by Maaten (2008). When certain points are equally close to each other, and the SNE algorithm attempts to reduce their dimensions, the points eventually become pulled together and destroy the gaps between clusters of points. Concretely, given n data points that that mutually equidistant on a $n-1$ dimensional manifold, the values of $p_{j|i}$ would all be equal. For larger n, the attractive force given by the gradient calculation would be very small. When many of these small forces show up, the data values are pulled into the center of the map, and do not differentiate well.

To subvert the unwanted attraction of dissimilar points, one solution is to use distributions with heavier tails. One that quickly comes into mind is the Student-t distribution. This is a lucrative choice, because it also does not involve an exponential term, which makes calculation all the more quicker. For simplicity, we use the t-distribution with one degree of freedom, which yields the following probability distribution for q:

$$q_{j|i} = \frac{(1 - ||y_i - y_j||^2)^{-1}}{\sum_{k \neq i}(1 - ||y_k - y_i||^2)^{-1}} \tag{40}$$

We will not follow a similar process as we did with the regular SNE algorithm, using our probability function p defined in Equation (35) as well as the newfound Student probability function as q. Again, the Kullback-Leibler divergence is used to obtain the gradient with respect to y_i:

$$\frac{\partial C}{\partial y_i} = 4 \sum_j (p_{j|i} - q_{j|i})(1 + ||y_i - y_j||^2)^{-1}(y_i - y_j) \tag{41}$$

The one other parameter that is necessary for the descent is the specification of σ_i used in the probability distribution p. It doesn't make sense to keep this value constant, because density of data is quite variate across dimensions. In most algorithms, the user will fix the perplexity, which grows as entropy grows, to find the value of σ_i based on perplexity. Obviously, perplexity, entropy, and σ_i are all positively correlated. For the sake of this experiment perplexity is fixed at 30, which is the industry standard for the t-SNE algorithm.

[3] Let us recall the definition of a spring constant k, given by $\frac{F}{\Delta x}$. Say the spring is stretched some x_1, provoking a restoring force F_1 from the spring. Suppose the spring is held still in the middle, and the left side of the spring is allowed to relax. Now, the spring is stretched x_2, which causes a force F_2. But, the center of mass of the spring has stayed constant, so there cannot have been a net force. So, $F_1 = F_2$. But by symmetry, $x_1 = 2x_2$, which means the spring force k_2 must have doubled. This same argument can be applied to any reduction of the length of a Hookian spring.

3 Experimentation and Results

3.1 Dataset

The techniques presented above will now be implemented on a dataset for analysis of accuracy and efficacy of such algorithms. For brevity's sake, this dataset will be marked as Ω

Ω will be a dataset taken from the Lung Genomics Research Consortium, which maps the entire human genome for individual patients to whether or not they carry lung disease. Obviously, the number of dimensions for this problem is quite exhaustive, with $16,271$ total dimensions.

As to how many dimensions the algorithm should strive to reduce the data to, there is no one correct answer. So, we will fix these values based merely on sensibility. Ω should be reduced to 400 dimensions. There is no mathematical significance of this relationship between d and n, but it exists simply because it yields logical results for both the large and the small datasets. Deviations in this mathematical relationship should not affect the sanctity of the experiment much, since this expectation is being held constant for all the algorithms, which is the purpose of the experiment.

3.2 Experimental Setup

The dataset will be run with all five algorithms previously introduced, and the performance and runtime of each algorithm will be tabulated. The five algorithms will be assigned a variable for brevity based on the table below:

Variable	Name of Algorithm
α	Principal Component Analysis
β	Nonnegative Matrix Factorization
γ	Kernel Component Analysis
δ	Independent Component Analysis
ϵ	t-Distributed Stochastic Neighbor Embedding

To calculate performance, the Pearson's coefficient (r) will be calculated for a multivariate linear regression model of the reduced predictors. This will be the case for all algorithms except for t-SNE, simply because it is a manifold algorithm is relies more on visualization and cannot reduce to 400 dimensions. K-means clustering will be used in lieu of regression, and the r value will be calculated based on the clustering results. It is also in our experimental interest to calculate confidence intervals for the value of r, which we cannot do directly.

To calculate intervals, we will convert our coefficients r_i into their respect Fisher's coefficients z_i, for which we can assume takes a normal distribution with a known standard error σ_z (assuming independent and identical distribution of points), allowing us to calculate the confidence intervals for z_i, converting back to r_i. Concretely, Fisher's z-transformation defines a normal z such that

$$z = \frac{1}{2}\ln(\frac{1+r}{1-r}) \tag{42}$$

with the fixed property that

$$\sigma_z = \frac{1}{\sqrt{N-3}} \tag{43}$$

where N is the number of data values. Under the normal assumption, it is not difficult to find the 95% confidence intervals for the z coefficient. Rearranging Equation (41) gives us the closed form for r in terms of z.

$$r = \frac{e^{2z}-1}{e^{2z}+1} \tag{44}$$

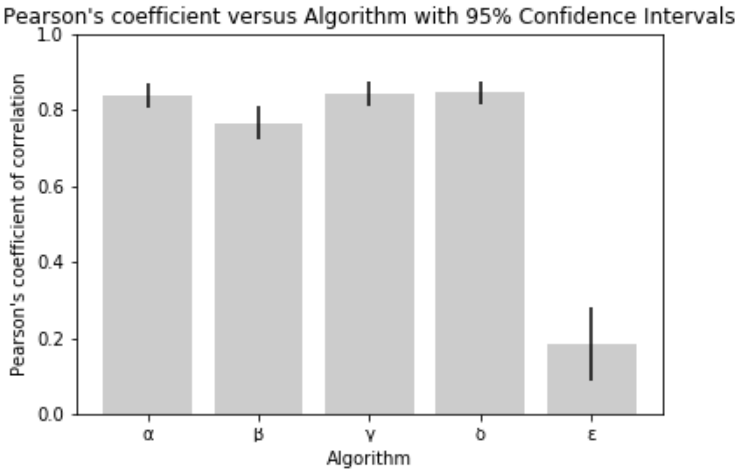
It is worth noting that Equation (43) has no solution for $r = 1$, and Equation (41) also falls under that assumption. Fortunately, data is rarely ever oriented in such a manner, and the $r = 1$ case need not be a concern for our calculations. For each permutation of algorithm and dataset, this calculation will be done twice, for a two-sided confidence interval with 95% confidence (using a z-score of 1.96). Such intervals will be used later to determine statistical significance of the variation of results.

3.3 Results

The table below shows the resulting r coefficients for each of the algorithms.

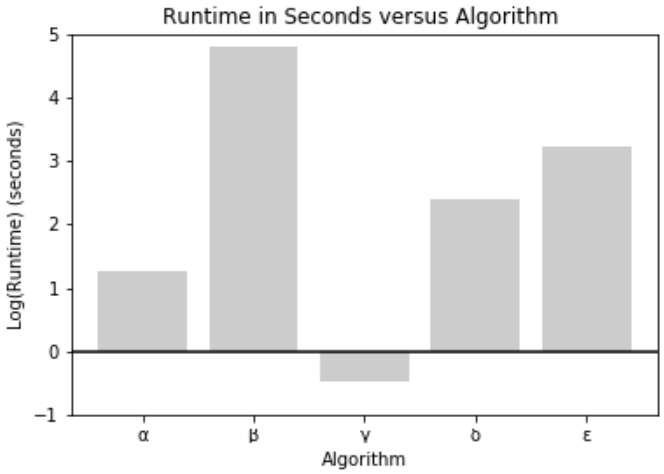
Variable	r
α	0.838
β	0.765
γ	0.842
δ	0.845
ϵ	0.184

The values of each of the algorithms will be visualized via bar plot, with error bars representing the 95% confidence intervals for each algorithm.



There is a generally consistent performance between the first four algorithms, which we can delve in further with some statistics, but there seems to be a very large drop when it comes to Algorithm ϵ, which is the t-SNE algorithm. From the graph alone, it appears that this algorithm is ill-equipped to conduct dimension reduction.

Additionally, the natural logarithm of run-time for each of the algorithms is graphed here:



Curiously, kernel PCA takes significantly less time than its arguably simpler counterpart, PCA. It is difficult to ascertain why exactly this is the case, since both of the algorithms solve a very similar eigenproblem. Unsurprisingly, ICA and t-SNE, both of which implement gradient descent methods, are run until convergence and exhibit much larger run-times. Algorithm β exhibits a very large runtime, but upon inspection of the dataset size, the magnitude seems sensible. The long run-time is also paired by a significantly lower accuracy (at least, according to the confidence intervals). This suggests that the NMF algorithm is more effective with more compact datasets, and perhaps should follow a PCA reduction for better results. Natural log-scaling of the time variable is used to better demonstrate the differences between small run-times, notably that of α and γ, the two PCA variants.

3.3.1 A Case for t-SNE

A primary difference between t-SNE and the other four algorithms is that t-SNE is a visual algorithm, and thus can only reduce datasets to two or three dimensions. For each of the other algorithms, the problem was reduced to 400 dimensions.

This discrepancy is quite significant, because given two dimensions, the projections of t-SNE can only take two values. Since this is a binary classification problem, these two values are forced at $0, 1$. This means that the linear regression model must fit a line to model data that only takes two values. A

much better way to model the predictions of t-SNE, perhaps, is to use logistic regression as a model.

More importantly, however, is the issue of "rounding" that happens when data is forced into few clusters. The regression models of the other four algorithms can return any real number between 0 and 1, whereas t-SNE only returns 0 and 1. Say the regression model comes up with the answer 0.54, but the real value is 0. The t-SNE algorithm would return 1, which obviously has a much larger residual when the r coefficient is considered. This leads to a much more sparse prediction, drastically reducing the r value.

To more equitably measure the efficacy of t-SNE, two possible continuations of this experiment could be considered:

1. t-SNE is a manifold algorithm, which differs from the general decomposition nature of the other algorithms. t-SNE ought to be compared with similar algorithms, such as Isomaps, Spectral Embedding, or Multi-Dimensional Scaling.

2. Instead of using regression as a metric of accuracy, classification algorithms, such as K-means Classification, ought to be used, so that the aforementioned rounding error does not harm the experiment.

It is also worth noting that both of these improvements to the experiment could be simultaneously considered for t-SNE. Likewise, the statistical implications of this result of t-SNE will not be seriously considered in analysis.

3.3.2 On Efficiency

Obviously, for algorithms with similar accuracies but differing run-times, for example, in γ and δ, it appears that the Algorithm γ is strictly better. If one grossly estimates efficiency as the dividend of accuracy and runtime, this seems to be the case. But is it really better?

The answer lies in the math of the algorithms. It seems like γ and δ achieve roughly the same thing when it comes to prediction, but δ goes one step further in the math to identify independent vectors to project the data on. These vectors can obviously be separated to better understand the nuances of the data. γ, on the other hand, compresses the data into statistically meaningful vectors, at the cost of some predictive efficiency.

4 Conclusion

Based on the results of dimension reduction on this dataset, it appears that PCA and its variant, Kernel PCA, tend to be the most efficient (when accuracy and time are both taken to account). Nonnegative Matrix Factorization suffers from a large data matrix, shown in both its runtime and accuracy. ICA performs roughly similarly to PCA, but finding independent vectors proves to be much less computationally efficient. Lastly, t-SNE suffers greatly in this analysis, because

of the clustering nature of the algorithm, and no meaningful conclusions of the algorithm itself can be drawn from this experiment.

This work is by no means objective, and does suffer from some experimental shortcomings. Firstly, these results stem from one dataset, and it may be more wise to include more datasets of differing dimensional structures (large dimension, small sample size; small dimension, large sample size) to receive a more holistic set of results from which to draw conclusions. Moreover, the linear regression model and Pearson's coefficient is used as a metric of efficacy of the reduction algorithms, which is by no means the most optimized. Clustering and purity, as an algorithm and metric, respectively, can also be used to evaluate the algorithms. Lastly, it would be most wise to separate decomposition algorithms, such as PCA and ICA, from manifold algorithms, such as t-SNE and Isomaps, in the analyses. This will help better avoid anomalous results like that of t-SNE in this experiment.

Dimension reduction is a very promising field of study for a scientific society that has too much data, and not enough predictive power. By being able to understand how certain algorithms work and perform in general conditions, the scientific community can make great strides toward harnessing this surplus of data as a blessing, rather than an obstacle.

5 References

An and Chen 2006: Jiyuan An, Yi-Ping Phoebe Chen, Finding rule groups to classify high dimensional gene expression datasets, Computational Biology and Chemistry, Volume 33, Issue 1,2009, Pages 108-113,ISSN 1476-9271.

Bellman RE (1957) Dynamic programming. Princeton University Press, Princeton.

Blumer, Anselm & Ehrenfeucht, Andrzej & Haussler, David & K. Warmuth, Manfred. (1989). Learnability and the Vapnik-Chervonenkis Dimension. J. ACM. 36. 929-965. 10.1145/76359.76371.

Burges, C. J. C. (2009). Dimension Reduction: A Guided Tour. Foundations and Trends in Machine Learning, 2(4), 275364. https://doi.org/10.1561/2200000002

Burred. (2014). Detailed derivation of multiplicative update rules for NMF.

Foldiak, P & Young, M (1995). Sparse coding in the primate cortex. The Handbook of Brain Theory and Neural Networks, 895- 898. (MIT Press, Cambridge, MA).

Hinton and Roweis 2002: Hinton, Geoffrey & Roweis, Sam. (2003). Stochastic Neighbor Embedding. 15.

Hyvarinen: A.Hyvrinen and E.Oja. A Fast Fixed-Point Algorithm for Independent Component Analysis.Neural Computation, 9 (7): 1483-1492, October 1997.

Lee and Seung 2001: Lee, Daniel & Seung, Hyunjune. (2001). Algorithms for Non-negative Matrix Factorization. Adv. Neural Inform. Process. Syst.. 13.

Lung Genomics Research Consortium. Lung Genomics.

Keogh, E., & Mueen, A. (2017). Curse of Dimensionality. In Encyclopedia of Machine Learning and Data Mining (pp. 314315). Springer US. https://doi.org/10.1007/978-1-4899-7687-1192

Maaten, L.V., & Hinton, G.E. (2008). Visualizing Data using t-SNE. Simon R, Radmacher MD, Dobbin K, McShane LM. Pitfalls in the use of DNA microarray data for diagnostic and prognostic classification. J Natl Cancer Inst 2003;95:148.

Python Software Foundation. Python Language Reference, version 3.0. Available at http://www.python.org.

Shlens, J. (2005). A Tutorial on Principal Component Analysis.CoRR, abs/1404.1100.

Srivastava, Nitish & Hinton, Geoffrey & Krizhevsky, Alex & Sutskever, Ilya & Salakhutdinov, Ruslan. (2014). Dropout: A Simple Way to Prevent Neural Networks from Overfitting. Journal of Machine Learning Research. 15. 1929-1958.

Stones 1982: Stone, C. (1982). Optimal Global Rates of Convergence for Nonparametric Regression.The Annals of Statistics,10(4), 1040-1053. Retrieved from http://www.jstor.org/stable/2240707

Subramanian, J., & Simon, R. (2013). Overfitting in prediction models Is it a problem only in high dimensions? Contemporary Clinical Trials, 36(2), 636641. https://doi.org/10.1016/j.cct.2013.06.011

Wang, Weiran & . Carreira-Perpin, Miguel. (2014). The role of dimensionality reduction in classification. Proceedings of the National Conference on Artificial Intelligence.