Data-Dimensionality Reduction

Using PCA, t-SNE, Sammon map and Autoencoder

Lori Sheng

May 11, 2019

Abstract

In modern society, we are surrounded by a large amount of data from various fields: digital communication, education, economy, medical care and so on. We want to get valuable information from the data in order to evaluate and to predict results. However, raw data can be hard to interpret and harder to model as we may have missing data, data that contains a lot of noise or data that is extremely high dimensional. In order to deal with extremely high-dimensional data, we wish to somehow map the data to low dimensions (two or three dimensions) in order to visualize it. In this project, we will compare and explore several methods: Principal Component Analysis (PCA), t-Stochastic Neighbor Embedding(t-SNE), Sammon Map, and the autoencoder (a type of neural network).

Acknowledgements

I would like to express my gratitude to my supervisor, Professor Hundley, who has led me through this project, and to my peer-review partner, Chaoyi Lou for helping me edit my project. I am grateful for the time and effort my professors, classmates, and friends have put in my project-My project would not be possible without their support. Additionally, I want to thank Professor Keef, who teaches the senior project class, for teaching me how to write a professional math paper, giving me guidance throughout my presentations, for his constant support. Throughout my time at Whitman College, the math department has constantly pushed me to become a better student, exposed me to different areas of math, and help me find a future career. I am incredibly thankful for my time here.

Contents

1 Introduction

In the last a few decades, copious amounts of data has gushed into our lives. Human beings create data every day. Data can be complicated, containing tons of features that may be useful or useless. Statistical and machine reasoning methods have difficulty when dealing with such high-dimensional data sets. If we can understand the data we produce, the benefits to our society will be endless. To better visualize the data, we can use data-dimensionality reduction. As an example of what we can do with dimensionality reduction: the breast cancer data we will look at has over 30 numerical features, meaning that it has over 30 dimensions. Invasive Ductal Carcinoma (IDC) is the most common subtype of all breast cancers. We can also load images in pixels and reduce their dimensionality in order to focus on the regions which contain the IDC. Further, we can do some predictions on detecing IDC. In order to extract important features from the data set or to get crucial information that we need, there are mainly two ways: (1) **feature extraction**: we can keep the most relevant information from the original dataset or (2) **dimensionality reduction**: we can exploit the redundancy of the original data and find a smaller set of new variables, each being a combination of the original variables, that contain basically the same information as the original variables. In this project, we will use the latter, applying some mapping methods to manipulate the data set, to achieve data-dimensionality reduction. We will discuss four mapping methods in total: PCA, t-SNE, Sammon Map, and Autoencoder. There are many benefits on reducing data-dimentionality, including:

- Model accuracy improves without redundant data
- Less computing with less time to train the algorithm
- Less storage space needed with less data
- Remove redundant data and noise

In Section 2, we will explore the oldest method in the four methods we mentioned above: Principal Component Analysis, or PCA. It was first described by Pearson in 1901, and then was developed independently by Hotelling in 1933. PCA is based in linear algebra, so we will follow the development found in *Linear Algebra and its applications* by David C. Lay [4]. Our explorations on the other three methods are all based on their original papers: SNE paper by Geoffrey Hinton and Sam Roweis[3], t-SNE paper by Laurens van der Maaten and Geoffrey Hinton[5], t-SNE documentation in Matlab, Sammon map paper by John W. Sammon, JR[6]. In the third section, we will introduce Sammon Map. In the forth section, we will dig into t-SNE by briefly introducing SNE first and looking at why t-SNE is more advanced

than SNE. In the fifth section, we will introduce Autoencoder. Then in the last section, we will compare these four methods using `MNIST` data set and see when we should use one over the other. At the end of the paper, we will include all the definitions, details of data sets, and references located in the appendix. By reading this project, we hope that our readers can have some idea of how each method works and how to choose the suitable one for data sets.

2 Principal Component Analysis (PCA)

Principal Component Analysis is a method based on the **Spectral Theorem** in linear algebra. We will talk about the usefulness of the Spectral Theorem later. Through PCA, we are able to compare two events and measure their similarity. Sometimes, when the data set has a high dimensionality and many variables have similar effects, we can replace a group of variables with a single new variable. Principal Component Analysis is a method for achieving this simplification. It is a mathematical procedure that transforms a number of correlated variables into a smaller number of uncorrelated variables called **principal components**. The **first principal component** and the **second principal component** account for as much of the variability in the data as possible, and each succeeding component accounts for as much of the remaining variability as possible. We will talk about their mathematical definitions in later sections.

2.1 Intuitive motivation

We first want to explain the method from a non-mathematical perspective. The data points are falling all over the space. We want to find a mapping method to project those data points on a $2-$dimensional space. Then the plot will show that similar samples will cluster together and show which new variable accounts for the most of the variability of the original data set by the following steps:

1. Calculate the center of the data for each variable by taking corresponding average values in order to make data points more concentrated.

2. Shift the data points by the amount of average values so that the center is on the origin of the graph.

3. Fit a line.

   (a) Start by drawing a random line that goes through the origin.

(b) Rotate the line until it fits the data set the best while keeping the line still going through the origin. However, how can we decide which line is the best?

- Project data points on the line.
- Measure the distance from data points to the line.
- Find the line that maximizes the distances from the projected points to the origin, calling distances $d_1, d_2, d_3, ..., d_n$. Then, square $d_1, d_2, d_3, ..., d_n$ to cancel negative values. We then compute the sum of the squared distances, ss below:

$$ss = {d_1}^2 + {d_2}^2 + ... + {d_n}^2$$

The line with the largest ss is the best line, and is called Principal Component 1 ($PC1$).

(c) The eigenvector for the first principle component is the (unit) vector pointing in the direction of the line. The eigenvalue is the variance of the data projected to that line, computed via the sum of squared distances.

4. Find the second principal component, $PC2$, which is the perpendicular to $PC1$ and has the next largest sum of squared distances.

5. Rotate the whole graph so that $PC1$ is the x-axis and $PC2$ is the y-axis. We could use those projected points to represent original data points on this 2-dimensional space.

6. We might note, and we'll define it in the next section that if we have n data points that have been centered, then dividing the sum of squared distances by $n - 1$ gives us the variance of the data projected to that line. Therefore, the first principle component is the vector on which the variance of the data is maximized. The second PC is the vector, perpendicular to $PC1$ that gives the next best variance, and so on.

This part provides an overview of how PCA works. If we have a high dimensional space instead of $2 - D$ in this example, the line we look for will be a $1-$dimensional subspace.

2.2 Statistical definitions

We will now explain how to transform data points from a high-dimension to a low-dimension and how to visualize it using PCA in a mathematical way. We must begin by defining principal components. To do so, we need to explain some terms first:

Let $\boldsymbol{x}$ represent a vector and

$$\boldsymbol{x} = \{\boldsymbol{x_1}, \boldsymbol{x_2}, ..., \boldsymbol{x_n}\}$$

and $\boldsymbol{y}$ represent a vector and

$$\boldsymbol{y} = \{\boldsymbol{y_1}, \boldsymbol{y_2}, ..., \boldsymbol{y_n}\}$$

Definition 2.1. Variance is denoted as S_x^2 and defined as

$$S_x^2 = \frac{1}{n-1}\sum_{i=1}^{n}(x_i - \bar{x})^2$$

where $\bar{x}$ is the mean of the vector $\boldsymbol{x}$.

Definition 2.2. Covariance of x_i and y_i is denoted as S_{xy}^2 and defined as

$$S_{xy}^2 = \frac{1}{n-1}\sum_{i=1}^{n}(x_i - \bar{x})(y_i - \bar{y})$$

where $\bar{x}$ is the mean of the vector $\boldsymbol{x}$ and $\bar{y}$ is the mean of the vector $\boldsymbol{y}$.

Definition 2.3. Mean-deviation form : Suppose we have a $m \times n$ matrix X. We first calculate the mean value of X. This can be either from taking the average over all rows (and ending up with a row, $1 \times n$), or by taking the average over all columns (and ending up with a column, $m \times 1$). Define M_x as the matrix created by replicating the row mean m times down or the column mean n times across, and this will be M_x (an $m \times n$ matrix). Then

$$B = X - M_x$$

is said to be in the mean-deviation form.

Definition 2.4. Covariance matrix. Let B be the $m \times n$ array of data in mean-deviation form (data with the mean subtracted). Then the **covariance matrix** for the data is the $m \times m$ matrix C defined by

$$C = \frac{1}{n-1}BB^T$$

For example, C_{ij} is the covariance between the i^{th} and j^{th} row of B.

Mathematically, the principal components of the data are unit eigenvectors $\boldsymbol{u_1}, \boldsymbol{u_2}, ..., \boldsymbol{u_n}$ of the covariance matrix C. Each principal component is a linear combination of the original variables. The **first principal**

component is the eigenvector corresponding to the largest eigenvalue of C. The **second principal component** is the eigenvector corresponding to the second largest eigenvalue of C. The principal components as a whole form an orthonormal basis for the space of the data set containing no redundant information. For 2−dimensional visualization, we only look at the first two principal components and construct a 2×2 matrix containing eigenvectors corresponding to the two largest eigenvalues because they could potentially be representatives of the original data set.

2.3 Mathematical explanation

As the principle components are eigenvectors, we need a theory to help compute them. We know that, for a general $m \times n$ matrix, eigenvectors are not defined. However, we will not be using a general matrix- we will be using a covariance matrix. The $m \times m$ covariance matrix is **symmetric**, which gives us a key to the whole computation.

Theorem 2.5. ***Spectral Theorem*** *[4]: If A is a real $n \times n$ symmetric matrix, meaning that $A^T = A$, then*

1. *A has n real eigenvalues, counting multiplicities.*
2. *For each distinct λ, the algebraic and geometric multiplicities are the same. (Note: algebraic multiplicity is the number of repetitions of eigenvalues, and geometric multiplicity is the dimension of the eigenspace).*
3. *The eigenspaces are mutually orthogonal.*
4. *A is orthogonally diagonalizable. If U is the matrix whose columns are (orthonormal) eigenvectors of A, then*

$$A = UDU^T$$

Proof. Before proving any of the above statements of spectral theorem, we need to state one fact first and then we can use it in further proofs.
Suppose $\boldsymbol{u}$ and $\boldsymbol{v}$ are two distinct eigenvectors of a matrix A and λ and μ are corresponding eigenvalues.

$$A\boldsymbol{u} = \lambda\boldsymbol{u}, A\boldsymbol{v} = \mu\boldsymbol{v}$$

Since $\boldsymbol{u}^T A\boldsymbol{v} = (A^T\boldsymbol{u})^T\boldsymbol{v}$,

$$\boldsymbol{u} \cdot A\boldsymbol{v} = A^T\boldsymbol{u} \cdot \boldsymbol{v}$$

and

$$\begin{aligned}\boldsymbol{u} \cdot A\boldsymbol{v} &= \boldsymbol{u} \cdot \lambda \boldsymbol{v} \\ &= \lambda(\boldsymbol{u} \cdot \boldsymbol{v})\end{aligned}$$

Then

$$A^T\boldsymbol{u} \cdot \boldsymbol{v} = \boldsymbol{u} \cdot \lambda \boldsymbol{v}$$

Since A is symmetric, meaning that $A = A^T$

$$\begin{aligned}\lambda(\boldsymbol{u} \cdot \boldsymbol{v}) &= A^T\boldsymbol{u} \cdot \boldsymbol{v} \\ &= A\boldsymbol{u} \cdot \boldsymbol{v} \\ &= \mu\boldsymbol{u} \cdot \boldsymbol{v} \\ &= \mu(\boldsymbol{u} \cdot \boldsymbol{v})\end{aligned}$$

Next, we move $\mu(\boldsymbol{u} \cdot \boldsymbol{v})$ to the other side, and we get

$$(\lambda - \mu)(\boldsymbol{u} \cdot \boldsymbol{v}) = 0$$

Since $\lambda \neq \mu$, we have $\boldsymbol{u} \cdot \boldsymbol{v} = 0$, meaning that $\boldsymbol{u}$ and $\boldsymbol{v}$ are orthogonal.
We can start proving the first statement using this fact.

1. We are going to prove prove part(1) and part(4) together because the result in part(1) is a part of the proof of part(4).

   To prove A has n real eigenvalues, we use contradiction. To compute eigenvalues and eigenvectors, we need to calculate the characteristic polynomial: $det(A - kI)$. The factors of this polynomial is

   $$(\lambda_1 - k)(\lambda_2 - k)(\lambda_3 - k)...(\lambda_n - k)$$

   Let f be the polynomial. Suppose the polynomial has a complex root, $\lambda_j = a + bi$ for some j. Let the

corresponding eigenvector be $\boldsymbol{x} + i\boldsymbol{y}$ with $\boldsymbol{x}$ and $\boldsymbol{y}$ each being real vectors.

$$A(\boldsymbol{x} + i\boldsymbol{y}) = (a + bi)(\boldsymbol{x} + i\boldsymbol{y})$$

Taking complex conjugates of two sides,

$$A(\boldsymbol{x} - i\boldsymbol{y}) = (a - bi)(\boldsymbol{x} - i\boldsymbol{y})$$

so $\boldsymbol{x} - i\boldsymbol{y}$ is also an eigenvector with eigenvalue of $a - bi$.
If λ_j is complex, then $b \neq 0$ and $a + bi \neq a - bi$. By the orthogonality of the eigenvectors, we must have that

$$(\boldsymbol{x} + i\boldsymbol{y}) \cdot (\boldsymbol{x} - i\boldsymbol{y}) = 0$$

which contradicts the following

$$(\boldsymbol{x} + i\boldsymbol{y}) \cdot (\boldsymbol{x} - i\boldsymbol{y}) = |\boldsymbol{x}|^2 + |\boldsymbol{y}|^2$$

since $\boldsymbol{x}, \boldsymbol{y}$ are real entries. Therefore, the polynomial only has real roots.
Now we need to prove there are n of the real eigenvalues. Suppose U is a $n \times n$ matrix, then we can partition the matrix as we show below, where $\boldsymbol{u_1}$ is $n \times 1$ and $\hat{\boldsymbol{u}}$ is $n \times (n-1)$. We note similarly the dimensions of U^T, where $\boldsymbol{u_1}^T$ is $1 \times n$ and $\hat{\boldsymbol{u}}^T$ is $(n-1) \times n$.

$$U = \begin{bmatrix} \boldsymbol{u_1} & \hat{\boldsymbol{u}} \end{bmatrix} \qquad U^T = \begin{bmatrix} \boldsymbol{u_1}^T \\ \hat{\boldsymbol{u}}^T \end{bmatrix}$$

Then multiplying, we have four blocks.

$$\begin{aligned} U^T A U &= \begin{bmatrix} \boldsymbol{u_1}^T \\ \hat{\boldsymbol{u}}^T \end{bmatrix} A \begin{bmatrix} \boldsymbol{u_1} & \hat{\boldsymbol{u}} \end{bmatrix} \\ &= \begin{bmatrix} \boldsymbol{u_1}^T A \boldsymbol{u_1} & \boldsymbol{u_1}^T A \hat{\boldsymbol{u}} \\ \hat{\boldsymbol{u}}^T A \boldsymbol{u_1} & \hat{\boldsymbol{u}}^T A \hat{\boldsymbol{u}} \end{bmatrix} \end{aligned}$$

Looking at each block more closely, the upper-left corner, is a scalar:

$$\begin{aligned} \boldsymbol{u_1}^T A \boldsymbol{u_1} &= \boldsymbol{u_1}^T \lambda_1 \boldsymbol{u_1} \\ &= \lambda_1 \boldsymbol{u_1}^T \boldsymbol{u_1} \\ &= \lambda_1 \end{aligned}$$

For the lower left block, we have an $(n-1) \times 1$ matrix that simplifies because $\mathbf{u}_1$ is orthogonal to the vectors in $\hat{\mathbf{u}}$.

$$\hat{\boldsymbol{u}}^T A \boldsymbol{u_1} = \lambda_1 \hat{\boldsymbol{u}}^T \boldsymbol{u_1} = \lambda_1 0 = 0$$

The upper right corner is irrelevant for now, but we'll revisit this momentarily.

Now the lower right corner is $(n-1) \times (n-1)$ matrix. Let $A_1 = \hat{\boldsymbol{u}}^T A \hat{\boldsymbol{u}}$.

We can repeatedly do the same thing and get $\lambda_2, \lambda_3, ..., \lambda_n$ by finding $A_2, A_3, ..., A_n$ in the square matrix. Through iterations, we can find out that there are n real roots.

Continuing to prove part(4), for now we only know that $\hat{U}^T A \hat{U}$ is an upper triangular matrix, calling it R, but we do not know what other positions of R are except λ's. Thus we move U and U^T to the other side and get

$$A = URU^T$$

Since A and U are symmetric matrices, the upper triangular matrix R is a symmetric matrix. Therefore, R is a diagonal matrix and we complete the proof.

2. For each distinct λ, the algebraic and geometric multiplicities are the same.
   We can prove part(2) using part(4). Since we have proved for part(4) that A is orthogonally diagonalizable and if U is the matrix whose columns are (orthonormal) eigenvectors of A, then

$$A = UDU^T$$

where D is a diagonal matrix. Let λ_1 has algebraic multiplicity of k. Since the columns of U are linearly independent, then we have k linearly independent columns of U, meaning that we have k geometric multiplicities.

3. Let $\boldsymbol{v_1}$ and $\boldsymbol{v_2}$ be eigenvectors from distinct eigenvalues, λ_1, λ_2. We want to show that $\boldsymbol{v_1} \cdot \boldsymbol{v_2} = 0$:

$$\begin{aligned}\lambda_1 \boldsymbol{v_1} \cdot \boldsymbol{v_2} &= (X\boldsymbol{v_1})^T \boldsymbol{v_2} \\ &= \boldsymbol{v_1}^T X^T \boldsymbol{v_2} \\ &= \boldsymbol{v_1}^T (X\boldsymbol{v_2}) \\ &= \lambda_2 \boldsymbol{v_1} \cdot \boldsymbol{v_2}\end{aligned}$$

We can shift variables around and get

$$(\lambda_1 - \lambda_2)\boldsymbol{v_1} \cdot \boldsymbol{v_2} = 0$$

From spectral theorem, we can get many lovely results. If A is any $m \times n$ matrix of real numbers, then the $m \times m$ matrix AA^T and $n \times n$ matrix A^TA are both symmetric. This result leads to an important proposition:

Proposition 2.6. ***Proposition*** *The matrices AA^T and A^TA share the same nonzero eigenvalues.*

Proof. Let $\boldsymbol{v}$ be a nonzero vector of A^TA with eigenvalue $\lambda \neq 0$. This means that

$$(A^TA)\boldsymbol{v} = \lambda \boldsymbol{v}.$$

Now, if we multiply both sides by A, we can get:

$$AA^T(A\boldsymbol{v}) = \lambda(A\boldsymbol{v}).$$

Therefore, we have proved that $A\boldsymbol{v}$ is a eigenvector of AA^T.

The theorem and the proposition are essential as they provide us with a simple way to find eigenvalues of the $n \times n$ matrix of AA^T no matter the difference in size of m versus n. After finding all the eigenvalues of A^TA, which is $m \times m$, and the rest $m - n$ eigenvalues of AA^T are all zero. Furthermore, by proving that $A\mathbf{v}$ is a eigenvector of AA^T, we have the connection to eigenvectors we need.
Another benefit of Spectral Theorem to PCA is that it guarantees that we can always construct an orthonormal basis of eigenvectors. Like the fact we find out from the Spectral Theorem, if X is a

symmetric matrix, then any two eigenvectors from different eigenspaces are orthogonal. We will now discuss a crucial concept that appears in the above statement: Orthonormal basis .

Definition 2.7. Orthonormal A set of vectors is orthonormal if the vectors are orthogonal to each other and the length of each vector is 1.

Suppose matrix U is formed using orthogonal columns, $U = [\boldsymbol{v_1}, \boldsymbol{v_2}, ..., \boldsymbol{v_k}]$. Then

$$U^T U = \begin{bmatrix} \boldsymbol{v_1}^T \\ \boldsymbol{v_2}^T \\ \vdots \\ \boldsymbol{v_k}^T \end{bmatrix} [\boldsymbol{v_1}, \boldsymbol{v_2}, ..., \boldsymbol{v_k}] = I$$

So that, if U is square, then $U^{-1} = U^T$. The columns of U would form a basis for $\mathbb{R}^n$. Therefore, for any $\mathbf{x} \in \mathbb{R}^n$, we can write

$$\boldsymbol{x} = c_1 \boldsymbol{v_1} + c_2 \boldsymbol{v_2} + \cdots + c_n \boldsymbol{v_n}$$

Since we have an orthonormal basis, the coefficients are computed via orthogonal projection

$$c_i = \frac{\boldsymbol{v_i} \cdot \boldsymbol{x}}{\boldsymbol{v_i} \cdot \boldsymbol{v_i}} = \boldsymbol{v_i} \cdot \boldsymbol{x}$$

If we decided to only use the first two basis vectors to represent $\mathbf{x}$, we would introduce an error. That is, if we represent $\mathbf{x} = (c_1, c_2)$ then we define the error in reconstructing $\mathbf{x}$ as:

$$Error = ||c_3 \boldsymbol{v_3} + c_4 \boldsymbol{v_4} + \cdots + c_k \boldsymbol{v_k}||^2 = ||\boldsymbol{x}_{error}||^2$$

We note that finding $\mathbf{v}_i$ that minimizes the error is the same as maximizing the quantity

$$||c_1 \boldsymbol{v_1} + c_2 \boldsymbol{v_2}||^2 = c_1^2 + c_2^2$$

In fact, we'll proceed by just looking for $\mathbf{v}_1$ first, then we'll compute $\mathbf{v}_2$, and so on.

2.4 The Principle Components

Given m points in $\mathbb{R}^n$ arranged in an $n \times m$ array X, we first put X in mean-deviation form by subtracting the mean of the data from every row, then we form the covariance matrix C, which is symmetric. Thus, we

can apply the Spectral Theorem to C, and order the eigenvectors according to the set of real eigenvalues, getting an orthogonal diagonalization of $C = U\Lambda U^T$. The columns of U are the principle components of the data.

Before proceeding further, we break here and look at some examples.

2.5 Examples

Here are two examples of PCA from different perspectives: statistics with numerical data processed using R-studio, and linear algebra with image data processed using Octave.

Example 2.8 (Example 1)**.**

My friend, Sarah Rothschild, and I did a principal component analysis on a heart disease data set from UCI. We tried to find similarities among 13 variables (details in appendix). We want to compare variables and their similarities and then summarize the set with a smaller number of representative variables that collectively explain most of the variability in the original set.

We first download the data set and import it into R-studio. We use `pr.out=prcomp()` function to center variables around the mean 0 with deviation being 1. The function projects $\mathbb{R}^{270}$ to $\mathbb{R}^2$ by finding eigenvectors corresponding to the two largest eigenvalues. Then we use `pr.out$rotation` to get principal component loading vectors and use `biplot(pr.out, scale=0)` to get a plane where $PC1$ is the x-axis and $PC2$ is the y -axis showing the relationship among variables.

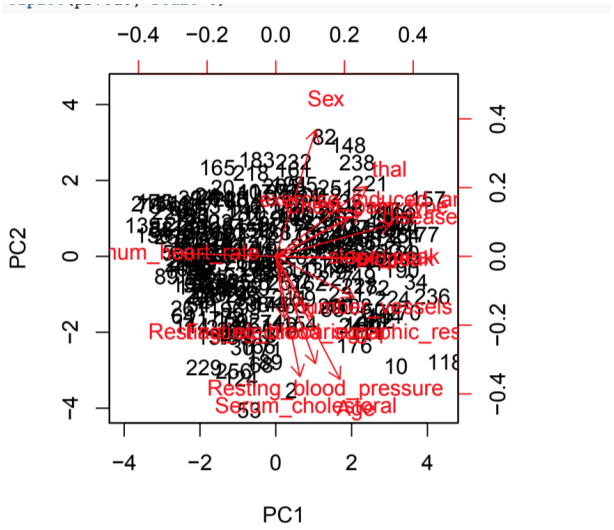

Figure 1: PCA plot on Heart Disease data set

Vectors that have similar directions and lengths have about the same effects on the response variable. The Figure 3 below are eigenvectors in 2-dimension.

```
##                                          PC1         PC2
## Age                                0.23338260 -0.44708879
## Sex                                0.14591631  0.45971778
## Chest_pain_type                    0.26388139  0.14016061
## Resting_blood_pressure             0.14421824 -0.38953877
## Serum_cholestoral                  0.08778353 -0.43662210
## Fasting_blood_sugar                0.03129551 -0.22473998
## Resting_electrocariographic_results 0.13405994 -0.22391028
## maximum_heart_rate                -0.34789716  0.01020265
## exercise_induced_angina            0.31168858  0.16054485
## oldpeak                            0.34981542 -0.01206693
## slope_peak                         0.31516772 -0.00459286
## number_vessels                     0.28420642 -0.14831297
## thal                               0.33113981  0.25380693
```

Figure 2: PC1 and PC2

It turns out that PCA is not the best method to analyze this data set because only around 27% of the data can be explained by the first two components.

```
##  [1] 0.25690984 0.11635208 0.08883022 0.08523647 0.07296537 0.06496323
##  [7] 0.06011651 0.05378470 0.04877160 0.04118226 0.03239827 0.03025938
## [13] 0.02446322 0.02376684
```

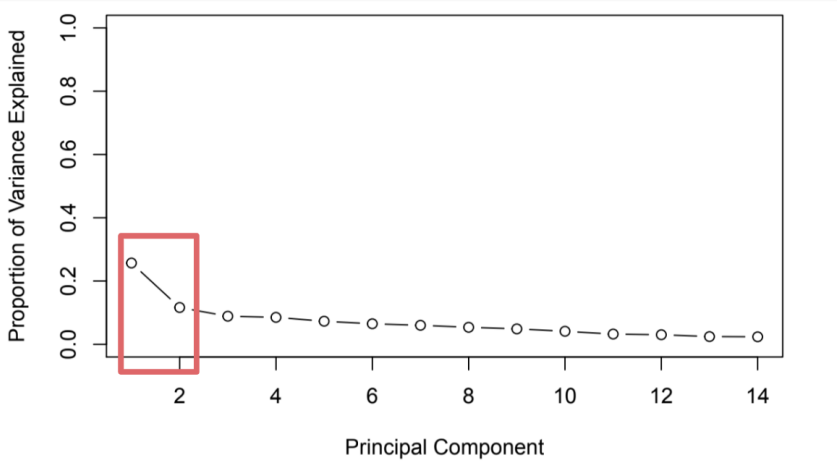

Figure 3: Performance of PC1 and PC2

Example 2.9 (Example 2)**.**

We also apply PCA to reduce dimensionality of a face image data set. The data set, called H, is a

1295504×15 matrix, where there are 15 face images with 1204×1076 for each.

1. `load FacesBW`;

2. `m=1204; n=1076;`

   set dimension for each image.

3. `H_m=mean(H,2);`

   find the mean of each row.

4. `imagesc(reshape(H_m(:,1),m,n));`

   reshape the first row of new matrix into a $m \times n$ matrix and display the data in what we have reshaped as an image that uses the full range of colors in the colormap. The mean face is shown as Figure 5.

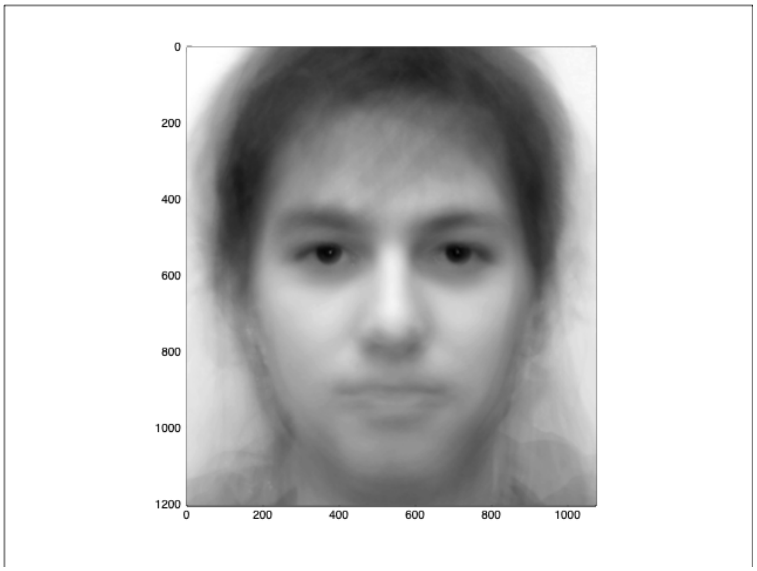

Figure 4: Mean Face

5. `colormap(gray);axis equal;`

   change the full range of colors of image to grey.

6. `H_s=H-H_m;`

   have a new matrix `H_s` that subtracts the mean and moves the data to the center.

7. `[u,s,v]=svd(H_s,'econ');`

produce an economy-size decomposition of $m \times n$ matrix `H_s` where `H_s` $= u \cdot s \cdot v'$. If $m > n$, then only the first n columns of u are computed, and s is $n \times n$. If $m = n$, then `svd(H_s,'econ')` is the same as `svd(H_s)`. If $m < n$, then the first m columns of v are computed, and s is $m \times m$.

8. `figure`

   plot the image.

9. `imagesc(reshape(u(:,1),m,n));`

   reshape the first row of u into a $m \times n$ matrix and display the data in what we have reshaped as an image that uses the full range of colors in the colormap. This is the eigenvector that correponds to the largest eigenvalue.

10. `colormap(gray);axis equal;`

    change the full range of colors of image to grey. The image of face of PC1 is shown below as Figure 6.

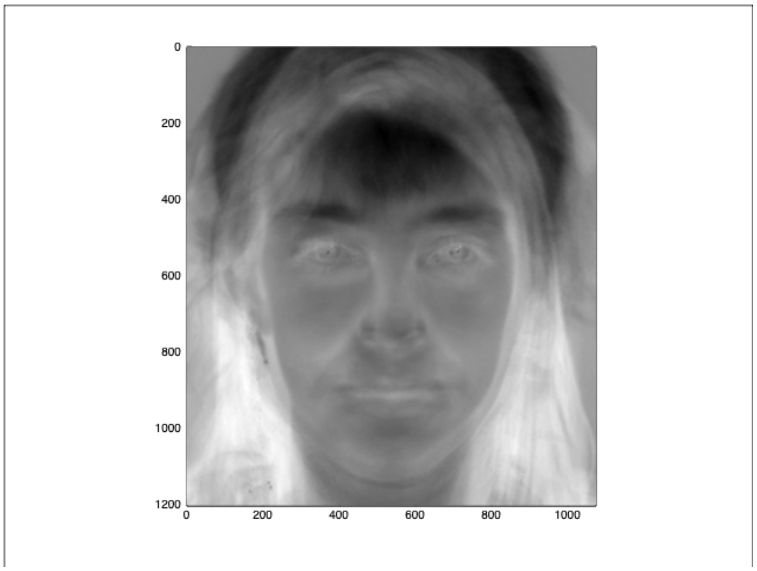

Figure 5: Face with PC1

11. `figure`

    plot the image.

12. `imagesc(reshape(u(:,2),m,n)); colormap(gray);axis equal;`

    reshape the second row of u into a $m \times n$ matrix and display the data in what we have reshaped as an image that uses the full range of colors in the colormap and change the full range of colors of

image to grey. This is the eigenvector that correponds to the second largest eigenvalue. The image of face of PC2 is shown as Figure 7.

Figure 6: Face of PC2

3 Sammon Map

In this section, we are going to introduce another algorithm to reduce data-dimensionality: Sammon Map[6]. The main idea of this method is to keep the structure that data points create in high-dimensional space the same as much as possible in the low-dimensional space. The Principal Component Analysis only has the ability to preserve linear projection, but is unable to keep preserve "complex" structures. Sammon Map minimizes the differences between corresponding inter-point distances in the low-dimensional space. The transformation is selected if the distance is very similar to the distance between two points in high-dimensional space. Additionally, Sammon Map needs to make sure that the mapping does not affect the overall structure of data points.

3.1 Mathematical explanation

Here is a mathematical explanation of Sammon Map. We want to map N points from X_i, where $i = 1, 2, 3, ..., N$ with $L-$dimension, to Y_j, where $j = 1, 2, 3, ..., N$ with $d-$dimension, and $d < L$. Let $d_{ij} =$ pairwise distance between Y_i and Y_j and $d^*_{ij} =$ pairwise distance between X_i and X_j. In order to know if the mapping preserves the structure of data points in high-dimensional space in low-dimensional space, we

want to look at the error function:

$$E = \frac{1}{\sum_{i<j} d_{ij}^*} \sum_{i<j}^{n} \frac{(d_{ij}^* - d_{ij})^2}{d_{ij}^*}$$

The reason why we set $i < j$ is to make sure each pairwise distance is connected only once. The tendency to preserve topology is due to the factor of d_{ij}^* in the denominator, which makes sure if the distance between two points is small, then the weight is greater. Our goal now is to minimize error function E so then we can have an optimal transformation.

The process of Sammon Map is:

1. Initialize Y_i by performing PCA on original data.

2. Update repeatedly Y_i with steepest descent until convergence occurs.

   Here, we only want to show the basic idea of updating Y_i with steepest descent, so we are going to do the first one. We know that error function is

   $$E = \frac{1}{\sum_{i<j} d_{ij}^*} \sum_{i<j}^{n} \frac{(d_{ij}^* - d_{ij})^2}{d_{ij}^*}$$

   Since $\frac{1}{\sum_{i<j} d_{ij}^*}$ is a constant, then we set this part to $\frac{1}{c}$, which is

   $$E = \frac{1}{c} \sum_{i<j}^{n} \frac{(d_{ij}^* - d_{ij})^2}{d_{ij}^*}$$

   Then we do partial derivative of y_{pq} in the error function

   $$\frac{\partial E}{\partial y_{pq}} = \frac{2}{c} \sum_{j=1}^{N} \frac{(d_{pj}^* - d_{pj})^2}{d_{pj}^*} \cdot \frac{-\partial d_{pj}}{\partial y_{pq}}$$

Now we need to figure out $\frac{-\partial d_{pj}}{\partial y_{pq}}$. As we set above,

$$\begin{aligned} d_{pj} &= d(Y_p, Y_j) \\ &= \sqrt{\sum_{k=1}^{q}(y_{pk} - y_{jk})^2} \\ &= \sqrt{(y_{p1} - y_{j1})^2 + (y_{p2} - y_{j2})^2 + ... + (y_{pq} - y_{jq})^2} \end{aligned}$$

We can find the derivative on d_{pj}

$$\begin{aligned} \frac{\partial d_{pj}}{\partial y_{pq}} &= \frac{1}{2}((y_{p1} - y_{j1})^2 + (y_{p2} - y_{j2})^2 + ... + (y_{pq} - y_{jq})^2)^{-\frac{1}{2}} \cdot \frac{\partial}{y_{pq}}(y_{pq} - y_{jq})^2 \\ &= \frac{2(y_{pq} - y_{jq})}{2\sqrt{(y_{p1} - y_{j1})^2 + (y_{p2} - y_{j2})^2 + ... + (y_{pq} - y_{jq})^2}} \\ &= \frac{y_{pq} - y_{jq}}{d_{pj}} \end{aligned}$$

We plug $\frac{\partial d_{pj}}{\partial y_{pq}}$ back to $\frac{\partial E}{\partial y_{pq}}$, which becomes

$$\frac{\partial E}{\partial y_{pq}} = -\frac{2}{c}\sum_{j=1}^{N}\frac{(d^*_{pj} - d_{pj})^2}{d^*_{pj}d_{pj}}(y_{pq} - y_{jq})$$

3.2 Examples

We still use our helix data set.

1. `[X, labels] = generate_data('helix', 2000);`

   Create a helix dataset with 2000 labels. (Note: The function `generate_data()` is a function built in the Matlab Toolbox of data-dimensionality reduction.)

2. `[mappedX,E] = sammon(X, 2);`

   Use the Sammon Map computing function that is built in the toolbox. The function will take data X in and reduce its dimensionality to 2, the second parameter of the function. This line of code will give us `mappedX`, the low dimensional space we map on, and an error function, `E`.

3. `plot(mappedX(:,1),mappedX(:,2),'.')`

Plot the first and the second dimension of `mappedX`.

Figure 13 is the graph we get after applying Sammon Map to the orignial data set.

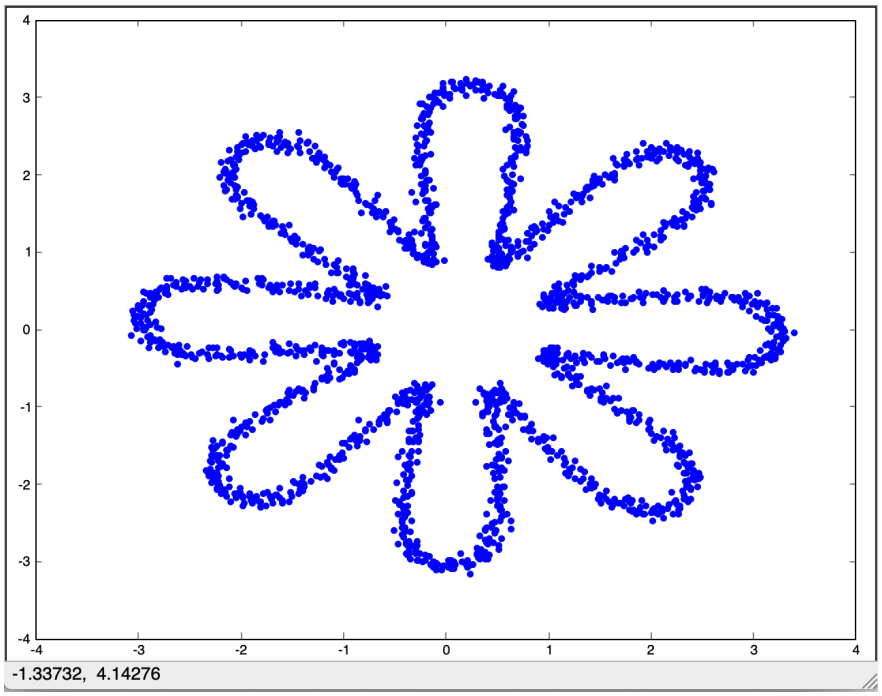

Figure 7: Applying Sammon Map on helix data set

We can notice that the graph we get from Sammmon Map is similar to the one from PCA. It is because the algorithm of Sammon Map starts with PCA and the helix data set only has 3 dimensions, the algorithm of Sammon Map stops with PCA and does not go any further.

4 t-Stochastic Neighbor Embedding (t-SNE)

t-Stochastic Neighbor Embedding is another way of placing objects in a high-dimension into a matrix of pairwise similarities and visualizing the resulting similarity data.

4.1 Stochastic Neighbor Embedding (SNE)

t-Stochastic Neighbor Embedding is a modified version of **SNE**, which stands for Stochastic Neighbor Embedding brought up by Geoffrey Hinton and Sam Roweis in 2002[3]. The low-dimensional space that data points convert on preserves neighbor identities. The purpose of embedding is to approximate the distribution for having this operation perform well in the low-dimensional images of the objects. The

model is a **Gaussian function**, which is centered on each data point in the high-dimensional space. The densities under this function is for identifying a probability distribution over all the potential neighbors of the data point, which means that the distribution of potential neighbors of the data point follows a normal distribution. We will talk about Gaussian function in the next paragraph. The natural cost function is the sum of **Kullback-Leiber divergences (KLD)** that will be presented with the Gaussian function. The KLD measures error so we want to minimize it.
SNE has some advantages over PCA:

- It allows multiple different low-dimensional images of each data point, where other methods requires a data point in high-dimension only to associate with one location in low-dimensional space. The feature is necessary because a single ambiguous object belongs in many separate locations in the low-dimensional image.
- By using SNE, even if data points are widely separated in the low-dimensional space, they will not be pressed as neighbors. The cost function enables to make nearby points stay close together and keep widely separated data points far apart.

Now, we can dive into the Gaussian function and the cost function.

1. **Gaussian function**:

   In the high-dimensional space, the conditional probability $p_{j|i}$, which is also called a Gaussian neighborhoods function, is the similarity of data point x_j to the data point x_i. This means that the probability that point x_i would pick x_j as its neighbor given that neighbors would be picked based on the proportion to their probability density under a Gaussian that is centered around x_i. For data points that are close to x_i, we expect a high $p_{j|i}$; for data points that are far away from x_i, we expect a infinitesimal $p_{j|i}$ as $p_{j|i}$ is defined as

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

   where $||x_i - x_k||^2/2\sigma_i^2$ is the dissimilarities and σ_i is the standard deviation for each high-dimensional point x_i. The reason why we use this **scaled squared Euclidean distance** is that clusters have different densities and different densities may have wider or more narrow distributions. In the high-dimensional space, σ_i is computed by a binary search. The binary search is looking for the σ_i that makes the **entropy** of the neighbors' distribution equal to $\log k$, where k is the **perplexity** that

is selected manually. In order to understand the overall idea, we need to know some terms first:

Definition 4.1. The **information** h of an event E is

$$h(E) := h(P(E)) = -\log_b(P(E))$$

We will use $b = 2$ as it is convenient to store information in 'bits'.

Definition 4.2. Let X be a random variable. Then the **entropy** is

$$H(X) := E[h(X)] = -\sum_{j=1}^{J} p_j \log_2 p_j$$

where $p_1, ..., p_J$ is the set of discrete probabilities of events in F.

The graph of $p_j \log_2 p_j$ is shown as Figure 8:

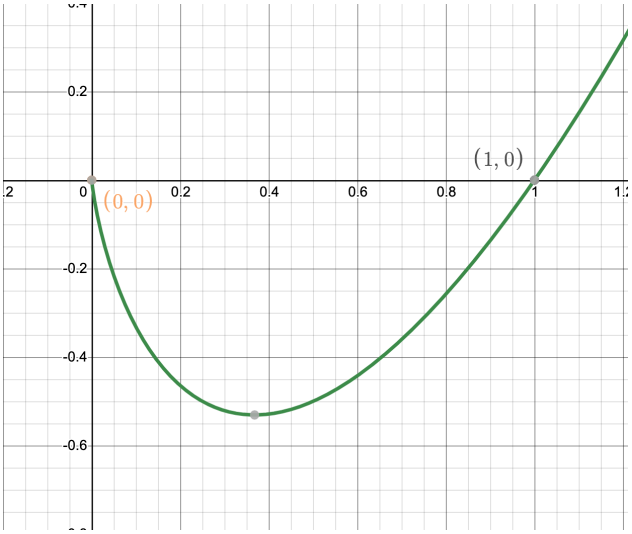

Figure 8: $p_j \log_2 p_j$

Note that as $\log_2 p_j$ approaching 0, $p_j \log_2 p_j$ approches 0. The equation reaches its minimum at $p_j = \frac{1}{e}$.

Definition 4.3. Perplexity is a measure of effective number of neighbors and its defined function is:

$$Perp(P_i) = 2^{H(P_i)}$$

where P_i is the fixed perplexity chosen by hand and $H(P_i)$ is the Shannon **entropy** of P_i.

In the low-dimensional space, we have a similar conditional probability $q_{j|i}$. The conditional probability $q_{j|i}$ is a function of the images of y_i in low dimension, finding the probability that point y_i would pick y_j as its neighbor given that neighbors would be picked based on the proportion to their probability density under a Gaussian that is centered around y_i. We set the standard deviation (σ) of the Gaussian of $q_{j|i}$ to $\frac{1}{\sqrt{2}}$ (Note: Setting the variance in the low-dimensional space to one value only results in a rescaled version of the final map. Once we use the same standard deviation for every data point in the low-dimensional space, we lose the property that the data is a perfect model of itself if we embed it in a space of the same dimensionality, because we use different standard deviations in the high- dimensional space). Here is the function of $q_{j|i}$:

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

2. **Cost function**:

   Our goal is to make $p_{j|i}$ and $q_{j|i}$ match as well as possible. One way to measure the differences and to achieve our goal is to use cost function: **Kullback-Leibler divergences**. It is defined as:

$$C = \sum_i KL(P_i||Q_i) = \sum_i \sum_j p_{j|i} \log \frac{p_{j|i}}{q_{j|i}}$$

   One way to minimize the cost function is using gradient descent method. Figure 8 illustrates the idea of gradient descent.

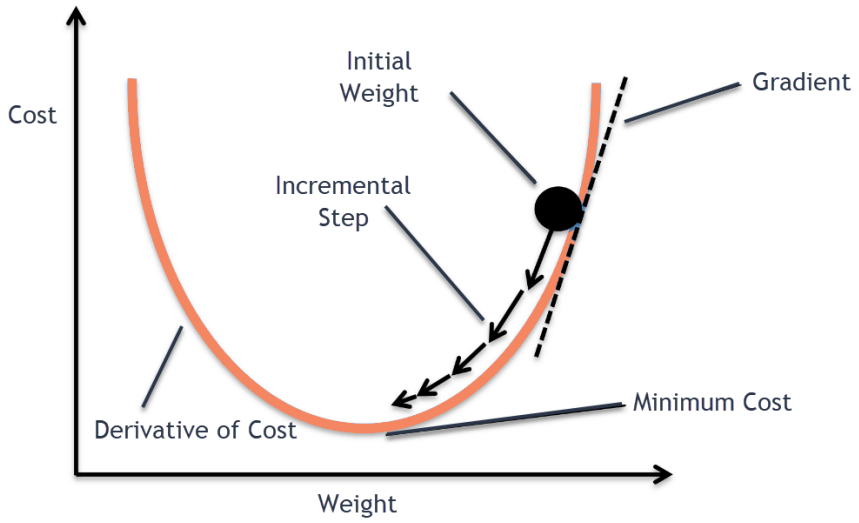

Figure 9: Gradient descent

Definition 4.4. Gradient descent is an optimization algorithm used to minimize some function by iteratively moving in the direction of steepest descent as defined by the negative of the gradient.

We start at a random position, and then we take our step downward in the direction of the negative gradient. Next we calculate the negative gradient (passing in the coordinates of our new point) again and take another step in the direction it determines. We continue this process repeatedly until we get to a local minimum, where we have nowhere to go.
Using the gradient descent method, we can know how to minimize C. The gradient of C is:

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

We can think of the gradient being a resultant force created by a set of springs between the point y_i and all other points y_j. All springs add a force on the direction $(y_i - y_j)$. If two points in the high-dimensional space are not similar, which means that the distance of these two points is too large, then the spring between y_i and y_j will repel the mapping points. If two points in high-dimensional space are similar, then the spring between y_i and y_j attracts the mapping points. Therefore the mapping points in low-dimensional space will keep the similar structure of data points from high-dimensional space and create a more clear cluster of data points. The force exerted on the string is proportional to its length and its toughness. In the above function, $(p_{j|i} - q_{j|i} + p_{i|j} - q_{i|j})$

represents the mismatch between the pairwise similarities of the data points and the mapping points, which determines the force on the string we talked before.

To start, we put all the low-dimensional images in random locations around the origin. In order to make the optimization faster and not to get stuck in poor local optima, we add a large momentum term to the gradient that decreases with time and gives us a better local optima.

When we begin our optimization, we add Gaussian noise to the map points after each iteration. Then we reduce the variance of noise gradually in order to get over the poor local minimum in the cost function. The stochastic neighbor embedding will find a better global organization if the variance of noise changes very slowly at the critical point. This method does have some advantages on finding an optimization method that gives good results without requiring the extra computation time and parameter choices introduced by the simulated annealing. However, it requires us to have a good amount of initial Gaussian noise and the step size in the gradient descent. Therefore, we need to run the optimization several times to find the optimal parameters.

4.2 t-SNE

As we said before, t-SNE is a modified version of SNE. Since SNE has some disadvantages in finding the optimal cost function and encountering **crowding problem**. In order to avoid these problems, Hinton and Roweis sightly changed the cost function of t-SNE in 2002[5]. Two major developments are:

- We use **symmetric version of SNE** in order to have a simpler gradient descents.
- We use **Student t-distribution** in the low-dimensional space instead of normal distribution (Gaussian distribution) in order to avoid crowding problem by having a heavier tail.

Several new terms were mentioned in the above statements. We are going to explain their meanings and functions below.

1. **Symmetric SNE**

   Instead of minimizing the cost function of Kullback-Leibler divergences between $p_{j|i}$ and $q_{j|i}$ (the sum of distance between conditional probabilities $p_{j|i}$ and $q_{j|i}$), we can minimize a *single* Kullback-Leibler divergences between a joint probability, P, in the high-dimensional place and a joint probability, Q, in the low-dimensional space:

$$C = KL(P||Q) = \sum_i \sum_j p_{ij} \log \frac{p_{ij}}{q_{ij}}$$

where we also set p_{ii} and q_{ii} to 0. We call this a symmetric SNE because for every i and j, $p_{ij} = p_{ji}$ and $q_{ij} = q_{ji}$.

The pairwise similarities in the high-dimensional space, p_{ij}, are defined as

$$p_{ij} = \frac{exp(-||x_i - x_j||^2/2\sigma^2)}{\sum_{k \neq l} exp(-||x_k - x_l||^2/2\sigma^2)}$$

The pairwise similarities in the low-dimensional space map, q_{ij}, are defined as

$$q_{ij} = \frac{exp(-||y_i - y_j||^2)}{\sum_{k \neq l} exp(-||y_k - y_l||^2)}$$

However, an issue will arise if x_i is an outlier in high-dimensional space. If x_i is an outlier, then p_{ij} will be extremely small for all j; the cost function will not be affected by y_i in the low-dimensional space. In order to avoid this problem, we set $p_{ij} = \frac{p_{j|i} + p_{i|j}}{2n}$, which makes each x_i contribute significantly to the cost function because $\sum_j p_{ij} > \frac{1}{2n}$. The symmetric version of SNE gives us a way to find simpler gradients:

$$\frac{\delta C}{\delta y_i} = 4 \sum_j (p_{ij} - q_{ij})(y_i - y_j)$$

2. **The crowding Problem**

Definition 4.5. The Crowding Problem: The area of the two-dimensional map that is available to accommodate moderately distant datapoints will not be large enough for the area needed.

We only briefly address this problem. It happens to other multidimensional scaling method too, such as Sammon Map.

3. **Student t-distribution**

Since we use the symmetric version of SNE, we can avoid crowding problem automatically. In high-dimensional space, we transform from distances to probabilities by using Gaussian (normal) distribution, and then in low-dimensional space, we convert back from probabilities to distances by using Student t-distribution, which has a heavier tails than normal distribution. Since student t-distribution does not contain exponents, it is faster to compute the density of a point under student t-distribution than Guassian distribution. Here, we use student t-distribution with one degree of

freedom, then the probability of q_{ij} becomes:

$$q_{ij} = \frac{(1 + ||y_i - y_j||^2)^{-1}}{\sum_{k \neq l}(1 + ||y_k - y_l||^2)^{-1}}$$

This has a helpful property: for map points that are far away from the cluster, $(1 + ||y_i - y_j||^2)^{-1}$ makes the map's representation of joint probabilities stay unchanged to the scale of the map.

In this case, the gradient of the cost function is:

$$\frac{\delta C}{\delta y_i} = 4 \sum_j (p_{ij} - q_{ij})(y_i - y_j)(1 + ||y_i - y_j||^2)^{-1}.$$

4.3 Examples

For t-SNE, we generate a set of data points that give us a helix-shape and we define the matrix as X being 2000×3.

Codes in Octave are very simple:

1. `[X, labels] = generate_data('helix', 2000);`

   Create a helix dataset, which is a function built in the Matlab Toolbox for Dimensionality Reduction.

2. `mappedX = tsne(X, labels);`

   Use t-SNE computing function that is built in the toolbox, too.

Figure 9 is an image corresponding to the original data set:

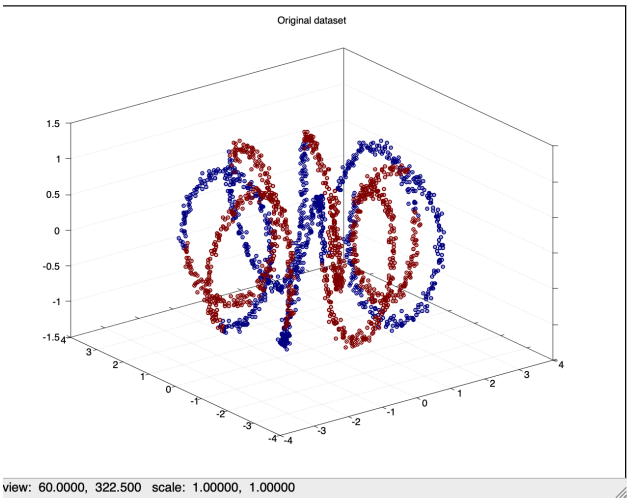

Figure 10: Original helix data

We first use t-sne to reduce its dimensionality as shown on Figure 10:

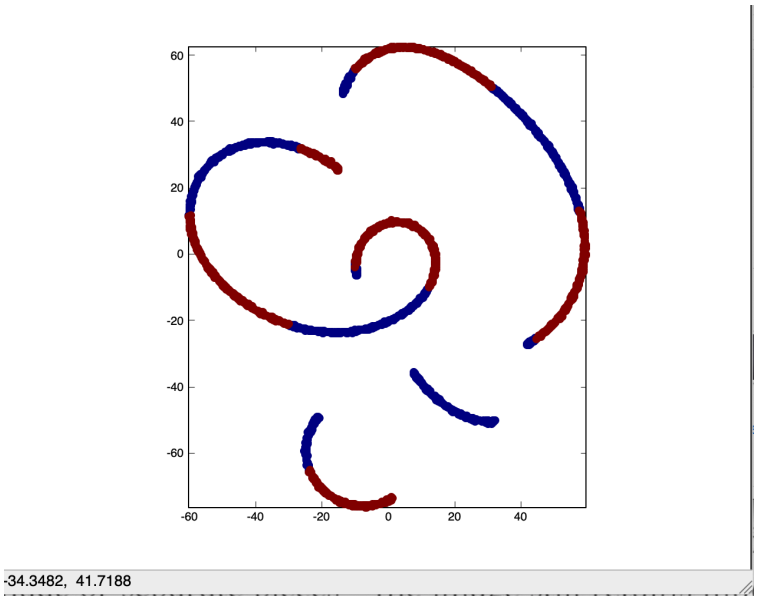

Figure 11: Applying t-SNE with default perplexity on helix data set

It is a long process to get this final image because t-sne is evaluating and deciding for each point if all other points can be its neighbor. Here is the link for the almost whole process: **Helix with t-SNE (**$perp = 30$**) (Click the link)**.

In order to have a better knowledge of perplexity, we change the perplexity (default value is 30) to 1 and 100.

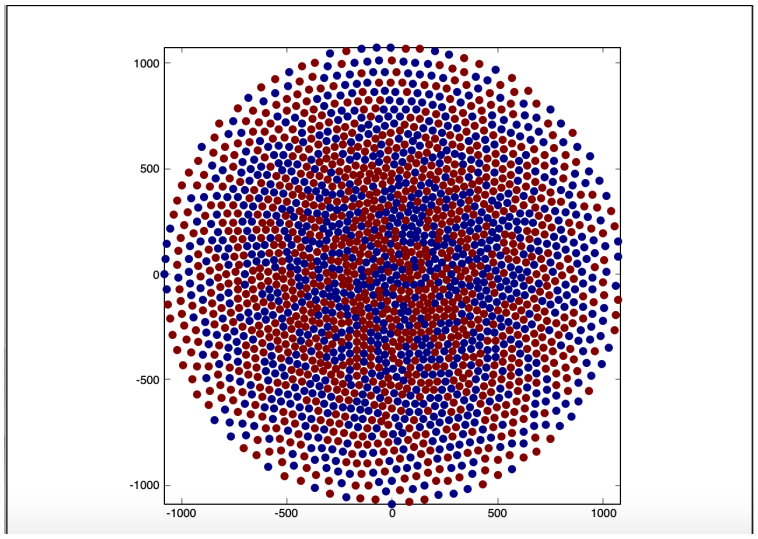

Figure 12: Applying t-SNE with perplexity of 1 on helix data set

Figure 11 is the data-dimensionality reduced graph when we change perplexity to 1. Even though we use t-SNE on the same data set, their outputs are significantly different because data points in Figure 11 tend to be more separated than data points in Figure 10. Since the perplexity is so small, the probability that one data point regards another data points as its neighbor is very low. That is why data points seem to be apart on the graph. By viewing the GIF below, you can find out that the graph has tiny changes each time. Here is the link for the almost whole process of perplexity being 1: **Helix with t-SNE($perp = 1$) (Click the link)**.

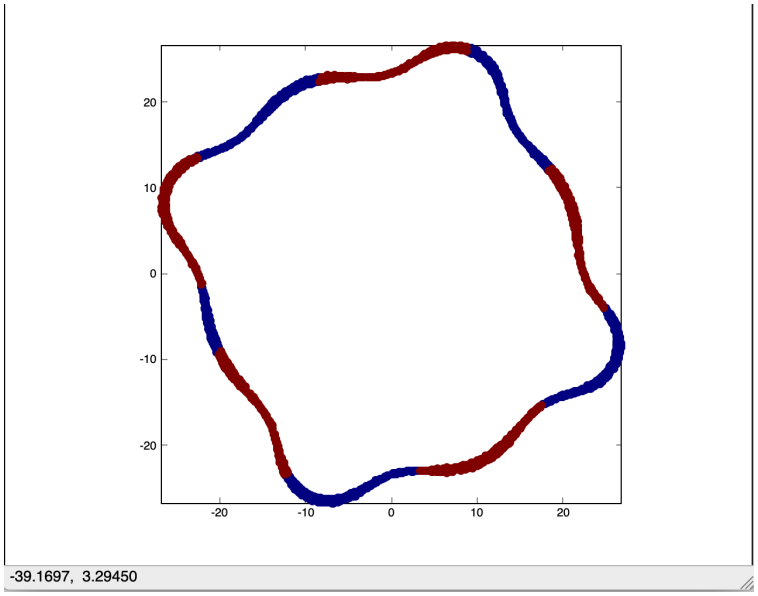

Figure 13: Applying t-SNE with perplexity of 100 on helix data set

Figure 11 is the graph we get when we assign perplexity to 100. Perplexity being 100 is relatively large. The probability of one data point regarding other data points to be its neighbor is very high, so data points in the low-dimensional space tend to be connected. By viewing the video, we can see that data points find their neighbors very fast and cluster together!

Here is the link for the almost whole process of perplexity being 100: **Helix with t-SNE($perp = 100$) (Click the link)**.

We also use PCA to reduce dimensionality of helix dataset in order to make some comparisons between results from PCA and t-SNE:

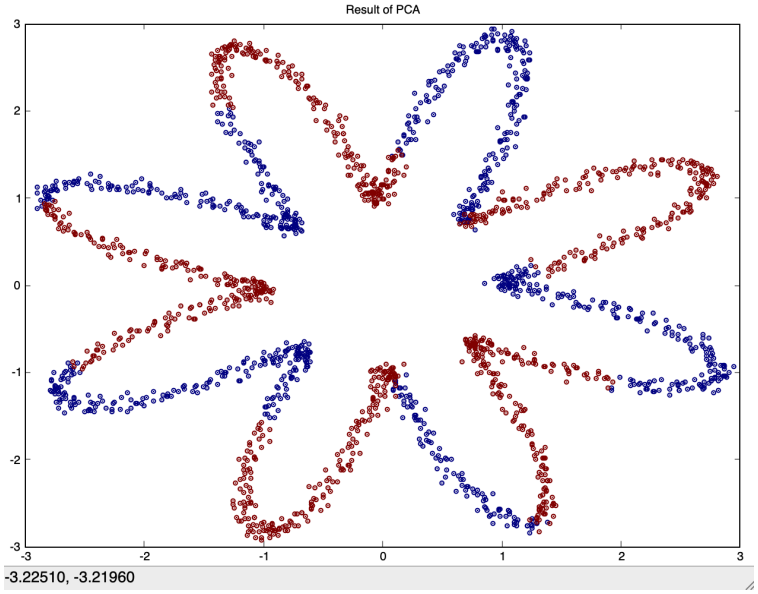

Figure 14: Applying PCA on helix data set

As seen in Figure 12, we get a flower-shape. From those two images, we could see that data points from t-SNE are more concentrated and in a irregular shape.

5 Autoencoder

Autoencoder is an abbreviation of **autoencoder neural network**[2]. It is an unsupervised learning, meaning that we are using unlabeled data. In order to understand autoencoder, we need to know what **Neural Nets** are.

5.1 History of Artificial Neural Networking (ANN)

Neural Nets first appeared in the 1940's and 1950's. They are based on an artificial neuron: the inputs and output are quantitative and each input connection is associated with a weight.

Definition 5.1. Neuron is an abbreviation of **artificial neuron**. The artificial neural network is based on a collection of connected units or nodes, which is called neuron.

Definition 5.2. Weights are usually denoted as W_i for some integer i. It represents the strength of connections between neurons.

The model computes a weighted sum of its inputs

$$y = W_1x_1 + W_2x_2 + ... + W_nx_n = W^T \cdot x$$

then we apply a *step function* to that sum and outputs the result, where the step is at trigger:

$$Y_W(x) = step(y) = step(W^T \cdot x)$$

This is not a good function because it is not continuous and not differentiable at the trigger. The progression of developing neural nets was hindered by this weakness. Later in 1970's, Stephen Grossberg, a group in Finland led by T. Kohonen, and several researchers tried to find methods to train networks that could be put in parallel. Researchers replaced the step function by any function that is increasing, differentiable and has finite horizontal asymptotes at ∞ and $-\infty$, which is **sigmoidal function**.

Definition 5.3. Sigmoid function is usually denoted as $\sigma(r)$ and is defined as

$$\sigma(r) = \frac{1}{1 + e^{-r}}$$

In 1986, D.E.Rumelhart et al introduced **back-propagation** algorithm to the public.

5.2 Back-propagation

The algorithm of back-propagation starts with the **feed-forward net**, feeding each training instance to the network and computing the output of every neuron in each consecutive layer.

Definition 5.4. Feed-forward net is a kind of neural network that let the information flow through the function being evaluated from input x, through the intermediate computations used to define function f, and finally to the output y.

Suppose we have l neurons, then we have a linear mapping from $\mathbb{R}^n$ to $\mathbb{R}^l$ and W_1 is a $l \times n$ matrix. We have our input $\boldsymbol{x}$ and we feed it into the linear function:

$$\boldsymbol{x} \longrightarrow W_1\boldsymbol{x} + b_1$$

where b_1 is a bias term we added to the model. We call this the *prestate*, denoted as P_1, of the layer of neurons. After putting our input in a linear model, we feed it to a nonlinear function, $\sigma(r)$. We will get a

vector in $\mathbb{R}^l$ and we will call it *statestate*, denoted as S_1 of the layer of neurons. After going through this layer, our input $\boldsymbol{x}$ is transferred to

$$\boldsymbol{x} \longrightarrow P_1 = W_1\boldsymbol{x} + b_1 \longrightarrow S_1 = \sigma(P_1)$$

Our input becomes a vector in $\mathbb{R}^l$, and now we want to transform it into $\mathbb{R}^m$. Thus, we use a $m \times l$ weights matrix, W_2, and a bias term, $b_2 \in \mathbb{R}^m$ to get the output $y \in \mathbb{R}^m$

$$\begin{aligned} P_0 = \boldsymbol{x} &\longrightarrow S_0 = P_0 \\ P_1 = W_1S_0 + b_1 &\longrightarrow S_1 = \sigma(P_1) \\ P_2 = W_2S_1 + b_2 &\longrightarrow S_2 = P_2 \end{aligned}$$

The first formula is in the **input layer**; the second formula is in the **hidden layer**; and the last formula is in the **output layer**.

Definition 5.5. Input layer of a neural network is composed of artificial input neurons, and brings the initial data into the system for further processing by subsequent layers of artificial neurons.

Definition 5.6. Hidden layer in an artificial neural network is a layer in between input layers and output layers, where artificial neurons take in a set of weighted inputs and produce an output through an activation function.

Definition 5.7. Output layer in an artificial neural network is the last layer of neurons that produces given outputs for the program.

By combining all these formulas together, we have a function F:

$$F(\boldsymbol{x_i}) = W_2(\sigma(W_1\boldsymbol{x_i} + b_1)) + b_2$$

Therefore, F becomes a function of the **weights** W_1, W_2 and biases b_1, b_2. This is just a simple example with only one hidden layer. In real-life work, we might need to use multiple hidden layers.
After feed-forward network, the back-propagation measures the network's output error, which is the difference between the desired output and the actual output of the network.
We still use the model with only one hidden layer as an example.

Definition 5.8. Architecture of the neural network is typically defined by stating the number of neurons in each layer. For example, a $3-5-4-3$ network has two hidden layers of 5 and 4 neurons, and maps from $\mathbb{R}^3$ to $\mathbb{R}^3$.

The architecture of $1-1-1$ network is

$$\boldsymbol{x} \longrightarrow \boldsymbol{y} = W_2(\sigma(W_1\boldsymbol{x_i} + b_1)) + b_2$$

Given a target t, the error is

$$E(W_1, W_2, b_1, b_2) = \frac{1}{2}(t - \boldsymbol{y})^2 = \frac{1}{2}\Big(t - (W_2\sigma(W_1\boldsymbol{x_i} + b_1)) + b_2)\Big)^2$$

We will generalize the error function for models that have more than one hidden layers.
The back-propagation computes how much each neuron in the last hidden layer contributed to each output neuron's error. It then proceeds to measure how much of these error contributions came from each neuron in the previous hidden layer and so on until the algorithm reaches the input layer. In order to minimize the error overall, we will move in the opposite direction of the gradient. There are two major parameters in our model, weights and biases. Suppose we let the symbol u denote a parameter (either a weight or a bias). We use gradient descent again here. The new parameter u is updated by

$$u_{new} = u_{old} - \alpha\frac{\partial E}{\partial u} + \alpha\Delta u$$

where α is the **learning rate**.

Definition 5.9. Learning rate is a hyper-parameter that controls how much we are adjusting the weights of our network with respect the cost gradient. The lower the value, the slower we travel along the downward slope.

We use the chain rule on the error to compute Δu. In particular,

$$\Delta u = -\frac{\partial E}{\partial y} \cdot \frac{\partial y}{\partial u} = -(t - y) \cdot -\frac{\partial y}{\partial u} = (t - y)\frac{\partial y}{\partial u}$$

Specifically, here we compute some partial derivatives for the one-hidden-layer example.

$$\begin{aligned} \boldsymbol{y} &= W_2 S + b_2 \\ \frac{\partial \boldsymbol{y}}{\partial W_2} &= S \qquad \frac{\partial \boldsymbol{y}}{\partial b_2} = 1 \\ \Delta W_2 &= (t - \boldsymbol{y})S \qquad \Delta b_2 = (t - \boldsymbol{y}) \end{aligned}$$

Then we compute the other parameters in the model,

$$\begin{aligned} \boldsymbol{y} &= W_2 \sigma(P) + b_2 \\ \frac{\partial \boldsymbol{y}}{\partial W_1} &= W_2 \sigma^{'}(P) \cdot x \qquad \frac{\partial \boldsymbol{y}}{\partial b_1} = W_2 \sigma^{'}(P) \\ \Delta W_1 &= (t - \boldsymbol{y}) W_2 \sigma^{'}(P) \cdot x \qquad \Delta b_1 = (t - \boldsymbol{y}) W_2 \sigma^{'}(P) \end{aligned}$$

We can minimize the error function by optimizing parameters.
For general model building, we assume that we have p data pairs, $(\boldsymbol{x_1}, \boldsymbol{t_1}), (\boldsymbol{x_2}, \boldsymbol{t_2}), ..., (\boldsymbol{x_p}, \boldsymbol{t_p})$, where t stands for *target* as we defined before. Our goal is to build a function F where

$$F(\boldsymbol{x_i}) = \boldsymbol{t_i}$$

for $i = 1, 2, 3, ..., p$. We will add **error** term to the model because in real-life, the equation is not perfect. The error ϵ_i follows a normal distribution. Let $\boldsymbol{y_i}$ denote the output of the model so

$$\boldsymbol{y_i} = F(\boldsymbol{x_i})$$

and

$$\boldsymbol{t_i} = \boldsymbol{y_i} + \epsilon i$$

The error is the averaged difference between the value our model gives and the target. Thus, we define the error function as

$$E = \frac{1}{p} \sum_{i=1}^{p} ||\boldsymbol{t_i} - \boldsymbol{y_i}||^2$$

The error function E contains parameters x in F. There is an name for this error function: **Mean Squared Error**.

As we went through before, we will use gradient descent once again to find the direction of minimizing the error function. By using gradient descent

$$\frac{\partial}{\partial \alpha}(||\boldsymbol{t}-\boldsymbol{y}||^2) = -2(\boldsymbol{t}-\boldsymbol{y}) \cdot \frac{\partial \boldsymbol{y}}{\partial \alpha}$$

The representation of the output y in terms of weights and biases is

$$\boldsymbol{y} = W\boldsymbol{x} + \boldsymbol{b} = \begin{bmatrix} W(1,:)\boldsymbol{x} + b_1 \\ W(2,:)\boldsymbol{x} + b_2 \\ \vdots \\ W(p,:)\boldsymbol{x} + b_p \end{bmatrix} \Rightarrow \frac{\partial \boldsymbol{y}}{\partial W_{ij}} = \begin{bmatrix} 0 \\ \vdots \\ 0 \\ x_j \\ 0 \\ \vdots \\ 0 \end{bmatrix}$$

where x_j is in the i^{th} coordinate position. By putting things altogether

$$\frac{\partial}{\partial W_{ij}}(||\boldsymbol{t}-\boldsymbol{y}||^2) = -2(t_i - y_i)x_j$$

The back-propagation efficiently measures the error gradient across all the connection weights in the network by passing the error gradient backward in the network.

5.3 Autoencoder

An **autoencoder** is a kind of artificial neural network used to learn efficient data codings in an unsupervised manner.

Definition 5.10. Codings are the efficient representations of the input data that the artificial neural networks are learning of.

We can view Autoencoders as feature detectors. Autoencoders work by training a neural network to be the identity mapping (so the output is identical to the input). What makes the mapping non-trivial and dimensionality-reducing is that the middle layer will be a small number of neurons (2 or 3 for visualization

purposes). Figure 15 gives us an autoencoder network with one input layer, one output layer and five hidden layers. The autoencoder always includes two parts: **encoder**, which is the mapping from the input layer to the middle layer, and the **decoder**, which is the mapping from the middle layer to the output layer.

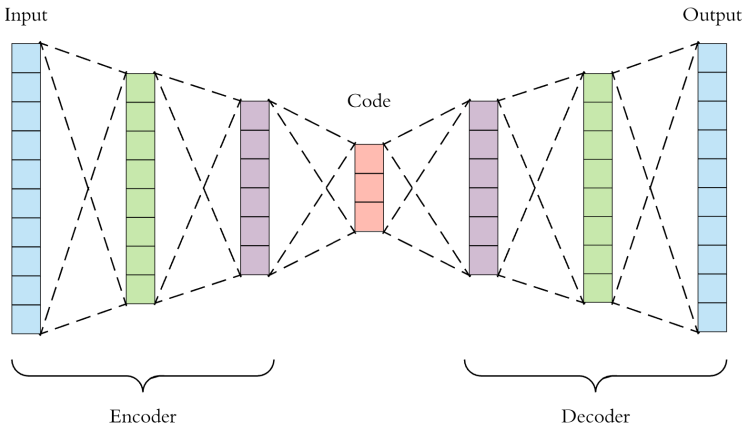

Figure 15: Autoencoder with the encoder half to the left and the decoder half to the right.

Definition 5.11. Encoder is a recognition network that converts the inputs to an internal representation.

Definition 5.12. Decoder is a generative network that converts the internal representation to the outputs.

We might note that, if the autoencoder used linear mappings (rather than applying a sigmoidal function), then the representation of the data on the middle layer would be identical to what we would obtain by performing Principal Component Analysis.

Definition 5.13. Stacked Autoencoders (or deep Autoencoders) is when Autoencoders have multiply hidden layers.

The architecture of a stacked Autoencoder is symmetric with regards to the coding layer. Suppose there are k hidden layers, the first half part (from input layer to the $\frac{k+1}{2}$th hidden layer) is the encoder. For data-dimensionality reduction, we will focus on **encoder** part and we are interested in the internal representation. Take the network in Figure 14 as an example of a stacked Autoencoder, Encoder maps the input from $\mathbb{R}^{10}$ to $\mathbb{R}^3$.

5.4 Example

We used Matlab/Octave for the last three methods, but for Autoencoders, we will use Python in jupyter notebook with TensorFlow. Tensorflow is a free and open-source software library for dataflow

programming. It is very powerful especially in the Machine Learning field.

We will start with a simple data set with only 3 dimensions, and we want to use Autoencoder to project the data points onto a 2-dimensional plane. We are going to go through the code and examine what Autoencoder does.

We first build the $3 - D$ model:

1. `import numpy.random as rnd`

   Import `numpy.random` in order to generate random real numbers.

2. `rnd.seed(4)`

   Keep the randomized numbers the same next time by giving it a name(4).

3. `m = 200, w1, w2 = 0.1, 0.3, noise = 0.1`

   Set the parameters: m is the number of points; `w1,w2` are the weights that we use in the function; `noise` is very intuitive itself.

4. `angles = rnd.rand(m) * 3 * np.pi / 2 - 0.5`

   Get the variable `angles`. Generate `m` random real numbers, multiply them by 3π, divided by 2 and then minus 0.5.

5. ```
   data = np.empty((m, 3))
   data[:, 0] = np.cos(angles)+np.sin(angles)/2 + noise * rnd.randn(m) / 2
   data[:, 1] = np.sin(angles) * 0.7 + noise * rnd.randn(m) / 2
   data[:, 2] = data[:, 0] * w1 + data[:, 1] * w2 + noise * rnd.randn(m)
   ```

   Construct a $m \times 3$ matrix called `data` and assign each column of `data` to some functions.

It is very important to normalize the data:

1. `from sklearn.preprocessing import StandardScaler`

   Import `StandardScaler`.

2. `scaler = StandardScaler()`

   Set up the scaler using `StandardScaler`.

3. `X_train = scaler.fit_transform(data[:100])`

   Assign the first 100 data points in `data` into the train set and scale them.
   ```

4. `X_test = scaler.transform(data[100:])`

   Assign the data points from 100 till the end of the `data` into the test set and scale them.

Now we are ready to build our Autoencoder:

1. `import tensorflow as tf`

   Import `TensorFlow`.

2. `reset_graph()`

3. `n_inputs = 3`

   `n_hidden = 2`

   `n_outputs = n_inputs`

   Assign the number of the input layer to 3 because our data is 3 dimensional;assign the number of the hidden layer (codings) to 2 because we want to transform the data set into 2 dimensional; then set the number of the output layer to be the same as the number of the input layer.

4. `learning_rate = 0.01`

   Set `learning_rate` as 0.01.

5. `X = tf.placeholder(tf.float32, shape=[None, n_inputs])`

   `hidden = tf.layers.dense(X, n_hidden)`

   `outputs = tf.layers.dense(hidden, n_outputs)`

   Let X be the placeholder for the input variable, which only takes in "float". Set up the hidden layer and the output layer.

6. `reconstruction_loss = tf.reduce_mean(tf.square(outputs - X))`

   Call the MSE cost function `reconstruction_loss`.

7. `optimizer = tf.train.AdamOptimizer(learning_rate)`

   `training_op = optimizer.minimize(reconstruction_loss)`

   `init = tf.global_variables_initializer()`

   Use `AdamOptimizer` to be the optimizer using `learning_ratet` defined earlier. Set up the training operation using the `optimizer` on the error function `reconstruction_loss`.

After establishing the Autoencoder, we can take our data set in and train it.

1. `n_iterations = 1000`

   `codings = hidden`

   Set the number of training iterations to be 1000 and the hidden layer to be `codings`.

2. `with tf.Session() as sess:`

   `init.run()`

   `for iteration in range(n_iterations):`

   `training_op.run(feed_dict={X: X_train})`

   `codings_val = codings.eval(feed_dict={X: X_test})`

   Take the training set in and train the model. Evaluate the result by taking the testing set in.

The last step is to plot the $2-D$ graph:

1. `fig = plt.figure(figsize=(4,3))`

   `plt.plot(codings_val[:,0], codings_val[:, 1], "b.")`

   `plt.xlabel("$z_1$", fontsize=18)`

   `plt.ylabel("$z_2$", fontsize=18, rotation=0)`

   `plt.show()`

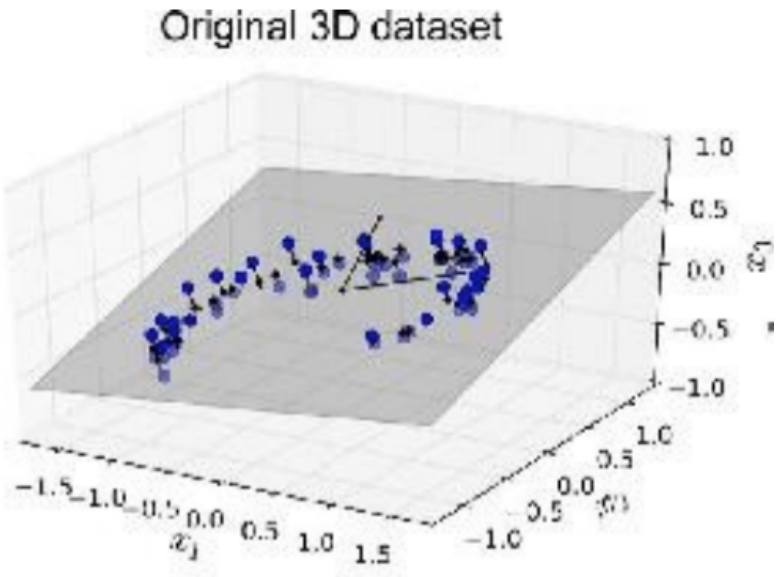

Figure 16: Original data points on $3-D$

Figure 16 is the plot of the original data set in $3-D$.

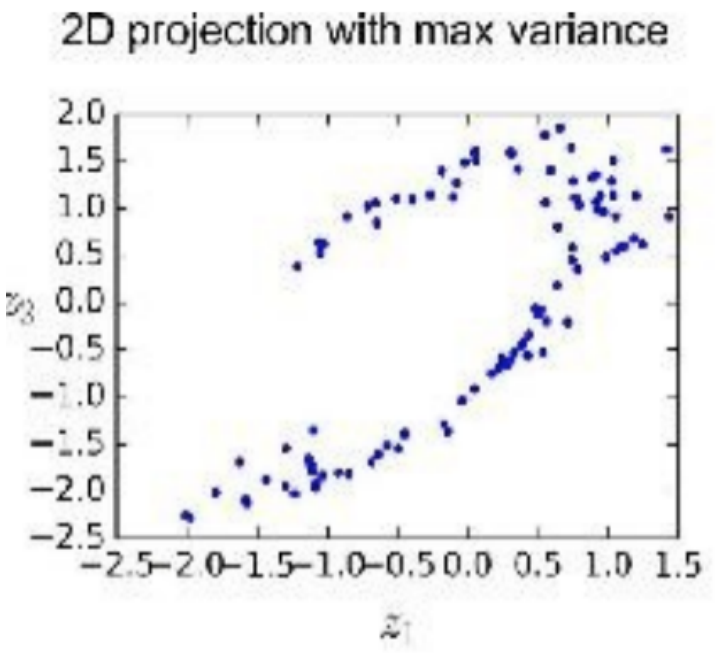

Figure 17: Output of Autoencoder's coding layer

Figure 17 is the plot of the $2-D$ plane that we project the original data points on.
Similar to PCA, the Autoencoder looks for the best $2-D$ mapping by preserving as much variance as it could.

6 Comparison and Conclusion

We will apply PCA, t-SNE, Sammon Map and Autoencoder on `MNIST` data set. It is a computer vision data set, consisting of 28×28 pixel images of handwritten digits from 0 to 9. The details of the data set are included in the appendix. Then we will compare how they are different and when we will choose one over the others.
In Colah's blog[1], he uses MNIST data set on PCA, t-SNE and Sammon Map and provides interactive programs so that users can visualize the process of data points clustering together. We will then use Autoencoder to map `MNIST` data set into $2-D$ in python. By applying the four methods on the same data set, we can easily notice how they differ from each other. It does not mean that one method is better than the others, but we need to utilize the most the suitable method for the specific project.

6.1 MNIST with PCA

As we talk about in section 2, we map data points in high-dimensional space onto a $2-D$ space with 2 principal components. For `MNIST` data set, here are two directions (horizontal and vertical) PCA chooses:

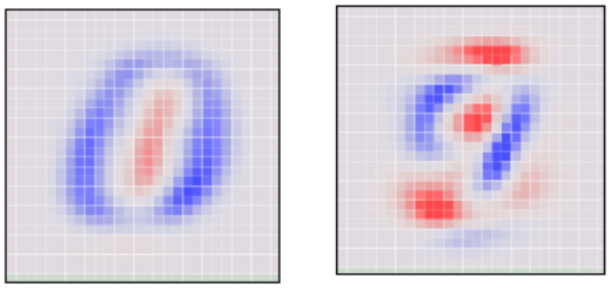

Figure 18: Two principal components of MNIST

Blue and red are used to denote what the trend is for that pixel. Red represents dragging a pixel's dimension to one side, blue to the other. Horizontally, if the pixels are mainly highlighted by red in the graph, then it pushes the data point toward left in the $xy-$ plane. Otherwise, it pushes the data point to the right. Figure 18 is the graph of visualizing `MNIST` with PCA.

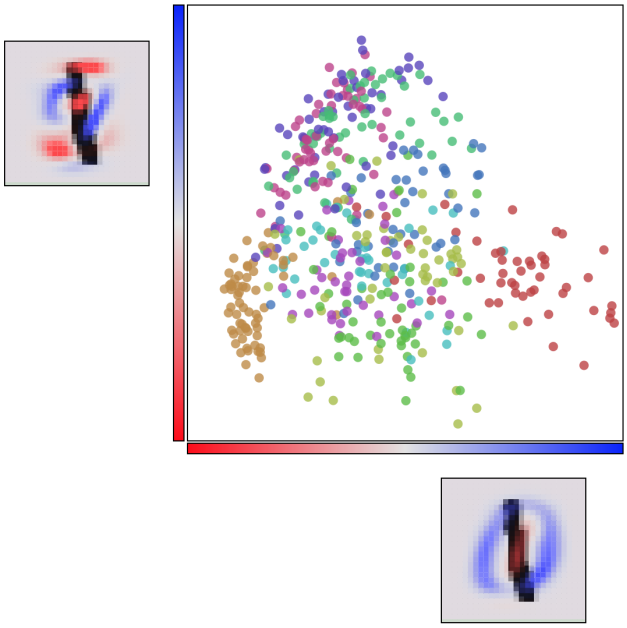

Figure 19: MNIST with PCA

However, the result turns out to be a scattered. We cannot divide digits or cluster them. `MNIST` data doesn't line up orderly for a pleasing visualization. The reason why it does not work is that in the high-dimensional space, `MNIST` is too complex for PCA, a linear projecting tool, to map on a $2 - D$ space.

6.2 MNIST with Sammon Map

As we talked in the section of Sammon Map, its algorithm starts with PCA. In the helix data set, we do not have a chance to see how Sammon Map works because there are only 3 dimensions in that data set. Luckily, the `MNIST` data set has 28×28 dimensions, so we can take a look at how Sammon Map works on dimensionality reduction.

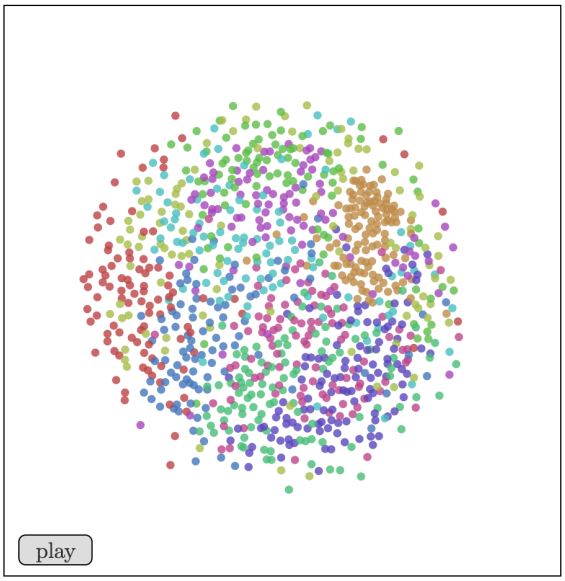

Figure 20: MNIST with Sammon Map

Figure 20 is the $2-D$ space in which Sammon Map plots the original data points on. What Sammon Map does is to preserve the distances between data points in the high-dimensional space in the low-dimensional space and also to preserve the overall data structure. We can vaguely see some clusters but Sammon Map does not divide each cluster clearly. Therefore, Sammon Map does not do a good job on complex data sets.

6.3 MNIST with t-SNE

The last two methods do not perform very well because they tend to map simple (linear) data structures; on the other hand, t-SNE is a method that can deal with complex data structures. What t-SNE tries to do is to preserve the neighbor identities. For every point, t-SNE evaluates and chooses neighbors for each data point, and then tries to make all points have the same number of neighbors. Let's take a look at the $2-D$ graph that `MNIST` maps on with t-SNE:

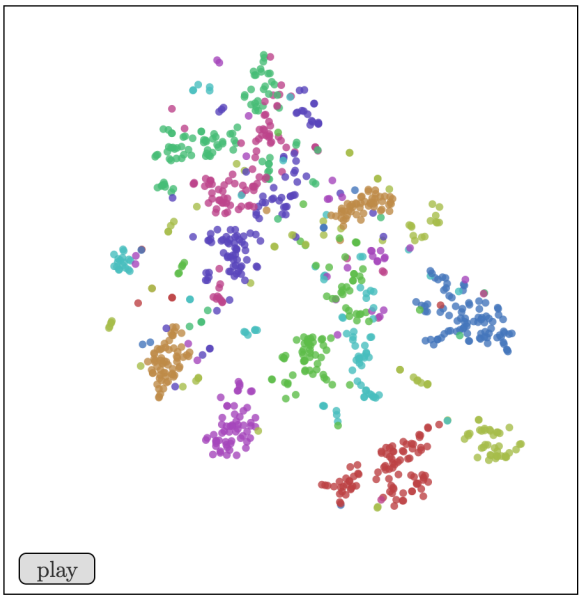

Figure 21: 1000 points of MNIST with t-SNE

From Figure 21, we can see t-SNE is very successful at revealing clusters in data by putting data points in one category together and keeping each cluster apart. It is a good method for visualization. However, it does have one weakness: getting stuck in local minimum as shown in Figure 22.

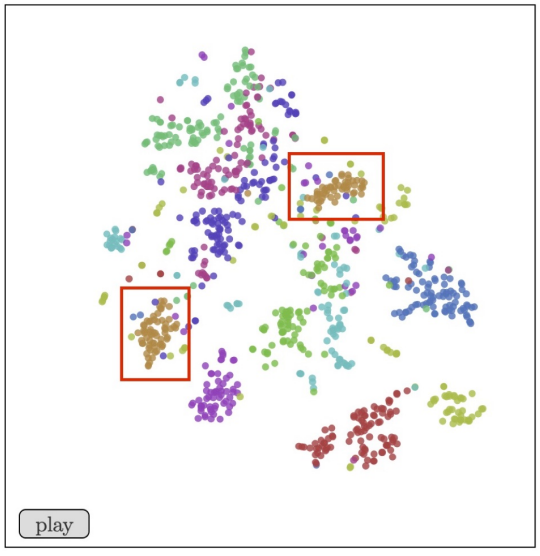

Figure 22: t-SNE stuck in local-minimum

There are two brown clusters on the graph and they are far apart. It means that the data points in this class get stuck in local minimum when t-SNE runs is algorithm. There are several ways that can help us get over the local minimum:

1. Add more instances
2. Use simulated annealing and carefully select a number of hyperparamters

Figure 23 is the graph when we use the full $50,000$ data points and it looks much better than the one above using only 1000 points.

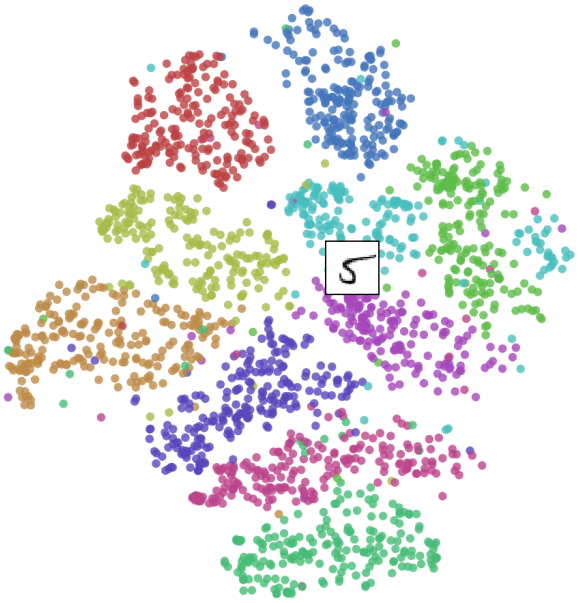

Figure 23: Full MNIST with t-SNE

6.4 MNIST with Autoencoder

Similarly to t-SNE, Autoencoder is another method that is capable of reducing dimensionality on complex data sets and more. By using artificial neural network, Autoencoder trains the model itself. If we have a new data point, we can put the point into Autoencoder and map it onto low-dimensional space easily because we have already built a model for the algorithm. However, adding a new data point into t-SNE will mess up the result we already got because t-SNE needs to reevaluate and choose neighbors for every data point again. Here is an example of how Autoencoder works:

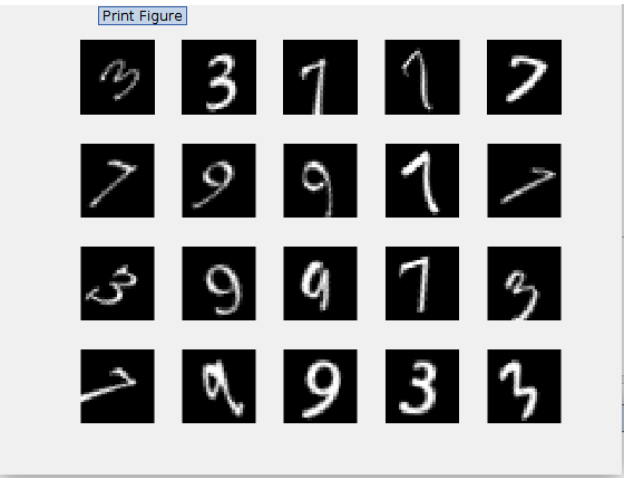

Figure 24: MNIST with Autoencoder

Figure 24 is the original set of images of handwritten 3,7 and 9. We decide to have 3 hidden layers. The hidden layer has 100 nodes, meaning that each of the 100 neurons has 784 numbers attached to it. The second hidden layer, which is the coding layer, has 50 layers. The third hidden layer, which has to have the same number of nodes as the first hidden layer. First, we ask Matlab to plot the first hidden layer as shown in Figure 25:

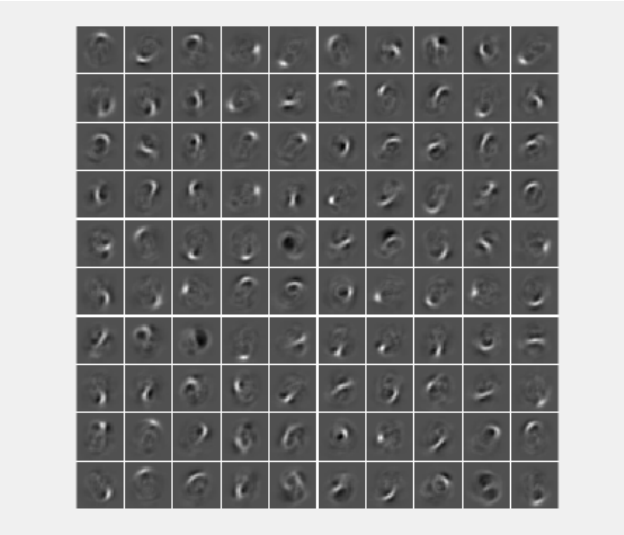

Figure 25: Plot of coding layer

The input layer has 784 nodes. After the first hidden layer, the number of nodes decreases to 100. We can visualize each of those 100 sets of weights as an image, and that is why Figure 25 contains 100 images. We also set the number of nodes in coding layer to be 2, so all the data points can be plotted on the

$xy-$plane. Figure 26 shows the results

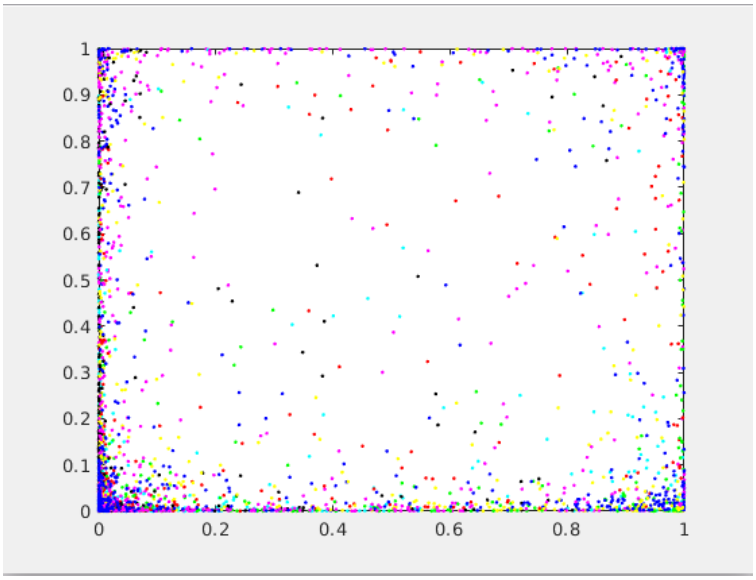

Figure 26: MNIST with Autoencoder (2-D)

It could be a useful tool for data dimensionality reduction but it does not have a meaningful representation in $2-D$ plot by looking at the weights in $2-D$.

6.5 Conclusion

We have shown all of the four methods of data-dimensionality reduction and the differences among them. We cannot say if one method is definitely better the other, however, we need to choose the one that fits the specific data set. Principal Component Analysis tends to cluster data points of the same class together in a long but narrow band. Sammon Map can preserve the local structure. Gaps in t-SNE are formed between different classes for better clustering and visualization. Autoencoder is extremely helpful when the data set has a very complex structure and we need to add new data points in the model.

Principal Component Analysis and Autoencoder have exited formulas so if we have new data points, we can feed into the formulas directly and then they will generate the new points in low-dimensional space. However, if we want to feed new data points into Sammon Map or t-SNE, we have to run the model again because new points will ruin the existing mappings.

Principal Component Analysis is the basic in data-dimensionality reduction. Many other methods' algorithms, like Sammon Map, start from PCA. One thing that we need to be aware of is that PCA may eliminate many information we actually want, which makes the further dimensionality reduction

undesirable.

It is also possible to apply two or more data-dimensionality reduction techniques sequentially, such as t-SNE and Autoencoder. t-Stochastic Neighbor Embedding can help us drag each clusters apart but sometimes, it will get stuck in the local minimum. By applying Autoencoder before t-SNE, the model can produce a better ordered output, which are the weights of each data point, that t-SNE can take in later. By understanding how different methods work, their strengths and weaknesses, later when we have different cases in hand, we can pull these techniques from our toolbox. We are looking forward to use data-dimensionlity reduction techniques in disease detection and related health fields.

References

[1] Colah. Visualizing mnist: An exploration of dimensionality reduction, Oct 2014.

[2] A. Géron. *Hands-on machine learning with Scikit-Learn and TensorFlow: concepts, tools, and techniques to build intelligent systems.* OReilly, 2017.

[3] G. Hinton and S. Roweis. Stochastic neighbor embedding - new york university, Jun 2003.

[4] D. C. Lay. *Linear algebra and its applications.* Addison-Wesley, 4 edition, 2012.

[5] L. v. d. Maaten and G. Hinton. Visualizing data using t-sne, Nov 2008.

[6] J. W. Sammon. A nonlinear mapping for data structure analysis, May 1969.

Appendix A Data sets

A Heart disease data set

The data set is from UCI. It has characteristic of Multivariate, containing 303 instances. There are 75 features (dimensions) in total in this data set but only 14 attributes used:

1. age
2. sex
3. cp
4. trestbps
5. chol
6. fbs
7. restecg
8. thalach
9. exang
10. oldpeak
11. slope
12. ca
13. thal
14. num (the predicted attribute)

Creators:

1. Hungarian Institute of Cardiology. Budapest: Andras Janosi, M.D.
2. University Hospital, Zurich, Switzerland: William Steinbrunn, M.D.
3. University Hospital, Basel, Switzerland: Matthias Pfisterer, M.D.
4. V.A. Medical Center, Long Beach and Cleveland Clinic Foundation: Robert Detrano, M.D., Ph.D.

Donor: David W. Aha (aha '@' ics.uci.edu) (714) 856-8779

Access: https://archive.ics.uci.edu/ml/datasets/Heart+Disease

B MNIST data set

The MNIST database of handwritten digits has $70,000$ instance in total, containing a training set of $60,000$ examples, and a test set of $10,000$ examples. It is a subset of a larger set available from `MNIST`. The digits have been size-normalized and centered in a fixed-size image 28×28.

Creators:

1. Yann LeCun, Courant Institute, NYU
2. Corinna Cortes, Google Labs, New York
3. Christopher J.C. Burges, Microsoft Research, Redmond

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

Alphabetical Index

List of Figures