Project 1 Report: Dimensionality Reduction

Mingquan Feng 517030910373

Mingjie Li 517030910344

Shangning Xu 517030910384

***Abstract*—The rise of deep learning means that feature extraction is no longer manual labor, but depending on network design, deep learning approaches tend to extract a large number of features, calling for effective dimensionality reduction methods. Based on preextracted deep learning features from the AWA2 dataset, we survey current dimensionality reduction methods, including feature selection by variance, PCA, kernel PCA, LDA, FA, t-SNE, LLE, MDS and AE, and evaluate them on SVM classifiers equipped with different kernels. Our experiment shows that t-SNE with LDA preprocessing gives the best performance, achieving 95.40% accuracy with only two dimensions. We also analyse each method's performance in terms of its approach to dimensionality reduction.**

***Index Terms*—dimensionality reduction, AwA2, PCA, kernel PCA, LDA, FA, t-SNE, LLE, MDS, AE**

I. Introduction

There are 3 main approaches to reduce dimensionality. The most naive approach is feature selection, which reduce the dimensions by selecting subsets of features according some criteria. Greedy methods like forward search or backward search repitively add new qualifying features or discard disqualified ones. The popular dimensionality reduction algorithms like PCA [1], kernel PCA [2], LDA [3], FA [4] etc. try to find lower dimensional linear combinations of the original features by learning a projection matrix W. These methods belong to another paradigm called feature projection. Nowadays, the third approach to dimensionality reduction, namely feature learning, is growing rapidly and have taken over the state-of-the-art of some areas like visualization. They manage to retain some characteristics of the training data, such as the data distribution (SNE, t-SNE [5]), manifold structure (LLE [6], MDS [7], IsoMap), or use deep learning method (Auto-encoder).

In this project report, we evaluated the performance of some popular dimensionality reduction algorithms on Animals with Attributes (AwA2) dataset [8], including variance feature selection, PCA, kernel PCA, FA, LDA, t-SNE, LLE, MDS and AE. The report is organized as follows: In Section II, we give a comprehensive review of methods evaluated in this report. Section III presents our evaluation of these methods and we analyse each method in terms of its approach to dimensionality reduction and compare the results with visualization in Section IV.

II. Method

A. Feature Selection

Feature selection is the most basic method for dimensionality reduction and its result remains easy to interpret, due to its intuitiveness. There are several types of feature selection methods, notably the wrapper method, where we repetitively select a subset of features and evaluate, until some stopping criteria are met, and the filtering method, where a score is computed for each feature and we filter out features that fail to met some threshold.

There are two approaches to the wrapper method, namely forward search and backward search. In forward search, we start with an empty set and greedily add features to our set, while in backward search we start with all features and greedily remove features. We experiment with and evaluate the selection-by-variance method, where the variance for each feature is computed and the top k features with largest variance are selected. It can be considered both as forward search and backward search.

B. Principal Component Analysis

Principal component analysis (PCA) was invented in 1901 by Karl Pearson [1]. Intuitively, PCA can be thought of as only keeping the most informative directions of data space, and therefore the unit vectors of these directions are named as principal components. The amount of information is measured by data variance after projection to principal component. This intuition leads to the object function as below:

$$\begin{aligned} &\max_{\mathbf{v}} \mathbf{v}^T\mathbf{X}\mathbf{X}^T\mathbf{v} \\ &s.t. \quad \mathbf{v}^T\mathbf{v} = 1 \end{aligned}$$

where $\mathbf{v}$ is principal component, $\mathbf{X}$ is matrix of data, $\mathbf{v}^T\mathbf{X}\mathbf{X}^T\mathbf{v} = \sum_{i=1}^n \left(\mathbf{v}^T\mathbf{x}_i\right)^2$ is proportional to projection variance. Using Lagrange multiplier to solve the optimization problem gives:

$$\begin{aligned} \mathcal{L}_{\mathbf{v}} &= \mathbf{v}^T\mathbf{X}\mathbf{X}^T\mathbf{v} + \lambda\left(1 - \mathbf{v}^T\mathbf{v}\right) \\ \frac{\partial \mathcal{L}_{\mathbf{v}}}{\partial \mathbf{v}} &= \mathbf{X}\mathbf{X}^T\mathbf{v} - \lambda\mathbf{v} = \mathbf{0} \\ \mathbf{X}\mathbf{X}^T\mathbf{v} &= \lambda\mathbf{v} \\ \mathbf{v}^T\mathbf{X}\mathbf{X}^T\mathbf{v} &= \mathbf{v}^T\lambda\mathbf{v} = \lambda \end{aligned}$$

Therefore, the principal components are actually normalized eigenvectors. To maximize variance with K dimensions, the eigenvectors corresponding to top-K eigenvalues are kept.

C. Kernel PCA

Kernel PCA [2] was proposed to reduce dimension in a non-linear method. Suppose the size of dataset is N, with d features. Notice that in general, when $d < N$, the N data points can not be linearly separated, which results in

limitation of linear PCA. But when $d \geq N$ they can almost always be linearly separated. Therefore, a mapping function $\Phi\left(\mathbf{x}_{i}\right)$,where $\Phi: \mathbb{R}^{d} \rightarrow \mathbb{R}^{N}$, is applied to first non-linearly increase data dimension, then it will be easier to linearly reduce dimension in the new feature space. However, in most of cases, it is expensive or infeasible to explicitly calculate $\Phi(\mathbf{x})$, therefore kernel trick is applied here. For data matrix $\mathbf{X}$, only the inner product $K=\Phi(\mathbf{X})^{T} \Phi(\mathbf{X})$ is calculated, where K is a $N * N$ matrix named kernel.

The formulation of kernel PCA is derived from PCA by replacing $\mathbf{X}$ with $\Phi(\mathbf{X})$, and replacing $\mathbf{v}$ with $\Phi(\mathbf{X}) \alpha$. Then $\mathbf{X X}^{T} \mathbf{v}=\lambda \mathbf{v}$ becomes:

$$
\begin{aligned}
\phi(\mathbf{X}) \phi(\mathbf{X})^{T} \phi(\mathbf{X}) \boldsymbol{\alpha} &= \lambda \phi(\mathbf{X}) \boldsymbol{\alpha} \\
\phi(\mathbf{X})^{T} \phi(\mathbf{X}) \phi(\mathbf{X})^{T} \phi(\mathbf{X}) \boldsymbol{\alpha} &= \lambda \phi(\mathbf{X})^{T} \phi(\mathbf{X}) \boldsymbol{\alpha} \\
\mathbf{K K} \boldsymbol{\alpha} &= \lambda \mathbf{K} \boldsymbol{\alpha} \\
\mathbf{K} \boldsymbol{\alpha} &= \lambda \boldsymbol{\alpha}
\end{aligned}
$$

which gives that α is eigenvector of $\mathbf{K}$. The projected data points are $\phi(\mathbf{X})^{T} \mathbf{v}=\phi(\mathbf{X})^{T} \phi(\mathbf{X}) \alpha=\mathbf{K} \alpha=\lambda \alpha$.

The choice of kernel can be arbitrary, and we choose 4 kernels used in sklearn KernelPCA module: Cosine similarity,Polynomial kernel,Sigmoid kernel and RBF kernel, whose expressions are listed below:

1) Cosine similarity: $k(x, y)=\frac{x y^{\top}}{\|x\|\|y\|}$
2) Polynomial kernel: $k(x, y)=\left(\gamma x^{\top} y+c_{0}\right)^{d}$
3) Sigmoid kernel: $k(x, y)=\tanh \left(\gamma x^{\top} y+c_{0}\right)$
4) RBF kernel: $k(x, y)=\exp \left(-\gamma\|x-y\|^{2}\right)$

D. Linear Discriminative Analysis

Linear Discriminative Analysis (LDA) [3] is a supervised dimension reduction method, which maximizes the inter-class variance and minimizes the intra-class variance. Here we use 2-class Fisher's linear discriminant, which is original form of LDA, to illustrate this idea formally as below:

$$
\begin{aligned}
J(\mathbf{v}) &= \frac{\left(\mathbf{v}^{T} \boldsymbol{\mu}_{1}-\mathbf{v}^{T} \boldsymbol{\mu}_{2}\right)^{2}}{\boldsymbol{\sigma}_{1}^{2}+\boldsymbol{\sigma}_{2}^{2}} \\
&= \frac{\left(\mathbf{v}^{T} \boldsymbol{\mu}_{1}-\mathbf{v}^{T} \boldsymbol{\mu}_{2}\right)^{2}}{\sum_{i=1}^{C_{1}}\left(\mathbf{v}^{T} \mathbf{x}_{1, i}-\mathbf{v}^{T} \boldsymbol{\mu}_{1}\right)^{2}+\sum_{i=1}^{C_{2}}\left(\mathbf{v}^{T} \mathbf{x}_{2, i}-\mathbf{v}^{T} \boldsymbol{\mu}_{2}\right)^{2}} \\
&= \frac{\mathbf{v}^{T}\left(\boldsymbol{\mu}_{1}-\boldsymbol{\mu}_{2}\right)\left(\boldsymbol{\mu}_{1}-\boldsymbol{\mu}_{2}\right)^{T} \mathbf{v}}{\mathbf{v}^{T}\left(\boldsymbol{\Sigma}_{\mathbf{2}}+\boldsymbol{\Sigma}_{\mathbf{1}}\right) \mathbf{v}} \\
&= \frac{\mathbf{v}^{T} \mathbf{S}_{B} \mathbf{v}}{\mathbf{v}^{T} \mathbf{S}_{W} \mathbf{v}}
\end{aligned}
$$

where $\boldsymbol{\Sigma}_{\boldsymbol{j}}=\sum_{i=1}^{C_{j}}\left(\mathbf{x}_{2, i}-\boldsymbol{\mu}_{2}\right)\left(\mathbf{x}_{2, i}-\boldsymbol{\mu}_{2}\right)^{T}$ denotes single inner class variance, $\mathbf{S}_{B}$ denotes variance between classes, $\mathbf{S}_{W}$ denotes variance within classes. Since the scale of $\mathbf{v}$ does not affect $J(\mathbf{v})$, we add constraint $\mathbf{v}^{T} \mathbf{S}_{W} \mathbf{v}=1$, then the optimization problem becomes:

$$
\max _{\mathbf{v}} \mathbf{v}^{T} \mathbf{S}_{B} \mathbf{v}, \quad \text { s.t. } \quad \mathbf{v}^{T} \mathbf{S}_{W} \mathbf{v}=1
$$

Apply Lagrange multipliers for this constrained maximization.

$$
\begin{aligned}
L &= \mathbf{v}^{T} \mathbf{S}_{B} \mathbf{v}-\lambda\left(\mathbf{v}^{T} \mathbf{S}_{W} \mathbf{v}-1\right) \\
\frac{\partial L}{\partial \mathbf{v}} &= 2 \mathbf{S}_{B} \mathbf{v}-2 \lambda \mathbf{S}_{W} \mathbf{v}=0 \\
S_{W}^{-1} S_{B} \mathbf{v} &= \lambda \mathbf{v}
\end{aligned}
$$

which gives that $\mathbf{v}$ is eigenvector of $S_{W}^{-1} S_{B}$. The above 2-class LDA can also be extended to multi-class LDA by defining S_{B} as below:

$$
S_{B}=\frac{1}{C} \sum_{i=1}^{C}\left(\mu_{i}-\mu\right)\left(\mu_{i}-\mu\right)^{T}
$$

where C is number of classes, μ_{i} is mean value of class C_{i} and μ is global mean of data. Since the rank of S_{B} is at most $C-1$, the rank of S_{W} is at most $n_feature$, therefore the number of non-zero eigenvector should satisfy $n_{eigenvector}=\min (C-1, n_feature)$. From this we derive that the maximal dimension after LDA should not exceed $\min (C-1, n_feature)$.

E. Factor Analysis

Factor Analysis (FA) is originated from Psychology [4]. In *sklearn* document [1], FA is illustrated as continuous latent variable model:

$$
\mathbf{X}=W \mathbf{H}+\mathbf{M}+\mathbf{E}
$$

where $\mathbf{X}$ is observed dataset, $\mathbf{H} \sim \mathcal{N}(0, \mathbf{I})$ is unobserved latent variable, W is weight of $\mathbf{H}$, also called factor loading matrix, $\mathbf{M}$ is mean value of $\mathbf{X}$, $\mathbf{E} \sim \mathcal{N}(0, \Psi)$ is Gaussian noise. If $\mathbf{H}$ is known, the posterior distribution of $\mathbf{X}$ can be derived as below, and we assume $\mathbf{X}$ is centered thus $\mathbf{M}=0$:

$$
\begin{aligned}
E\left(\mathbf{X} | \mathbf{H}=h_{i}\right) &= W h_{i}+E(\mathbf{E})=W h_{i} \\
Cov\left(\mathbf{X} | \mathbf{H}=h_{i}\right) &= E\left(\left(\mathbf{X}-E\left(\mathbf{X} | \mathbf{H}=h_{i}\right)\right)\left(\mathbf{X}-E\left(\mathbf{X} | \mathbf{H}=h_{i}\right)\right)^{T}\right) \\
&= \Psi \\
P\left(\mathbf{X} | \mathbf{H}=h_{i}\right) &= \mathcal{N}\left(W h_{i}, \Psi\right)
\end{aligned}
$$

Also notice that we assume $\mathbf{H}$ is unit Gaussian prior, thus the marginal distribution of $\mathbf{X}$ is:

$$
\begin{aligned}
E(\mathbf{X}) &= W E(\mathbf{H})+E(\mathbf{E})=0 \\
Cov(\mathbf{X}) &= E\left(\mathbf{X X}^{T}\right) \\
&= E\left(W \mathbf{H H}^{T} W^{T}+\mathbf{E H}^{T} W^{T}+W \mathbf{H E}^{T}+\mathbf{E E}^{T}\right) \\
&= W W^{T}+\Psi \\
P(\mathbf{X}) &= \mathcal{N}\left(0, W W^{T}+\Psi\right)
\end{aligned}
$$

Here we assume $\Psi=\operatorname{diag}\left(\psi_{1}, \psi_{2}, \ldots, \psi_{n}\right)$, and we can formulate fitting of FA as an maximum log-likelihood problem:

$$
\begin{aligned}
& \max _{W, \Psi} \quad \sum_{x_{i}} \log p\left(x_{i}\right) \\
\Rightarrow & \max _{W, \Psi} \quad \sum_{x_{i}}\left(-\frac{1}{2} \log \left|W W^{T}+\Psi\right|-\frac{1}{2} \log x_{i}\left(W W^{T}+\Psi\right)^{-1} x_{i}^{T}\right. \\
\Rightarrow & W W^{T}+\Psi=\sum_{x_{i}} x_{i} x_{i}^{T}=\mathbf{X X}^{T}
\end{aligned}
$$

[1] https://scikit-learn.org/stable/modules/decomposition.html

To solve the problem above, we can iteratively update W, Ψ by:

1) set some initial value of Ψ
2) estimate W as $W = \left(\sqrt{\lambda_1}\eta_1, \sqrt{\lambda_2}\eta_2, \cdots, \sqrt{\lambda_m}\eta_m\right)$, where m is dimension of $\mathbf{H}$, λ, η is eigenvalue and eigenvector of $\mathbf{X}\mathbf{X}^T$.
3) update Ψ based on W, and go to step 2 if not converge.

Suppose $\mathbf{X}$ is $n \times p$ matrix, then W is $n \times m$. When $m < p$, W is a feature reduced form of $\mathbf{X}$. Note that different assumption of error covariance Ψ leads to different models:

1) $\Psi = \sigma^2 \mathbf{I}$ leads to probabilistic model of PCA, or *PPCA*
2) $\Psi = \text{diag}\left(\psi_1, \psi_2, \ldots, \psi_n\right)$ leads to Factor Analysis.

Both models essentially estimate a Gaussian with a low-rank covariance matrix, therefore Factor analysis can produce similar components as PCA. The main advantage for Factor Analysis over PCA is that it can model the variance in every direction of the input space independently (heteroscedastic noise)

F. T-Distributed Stochastic Neighbor Embedding (t-SNE)

T-distributed Stochastic Neighbor Embedding (t-SNE) is a machine learning algorithm for visualization. [5] It is a nonlinear dimensionality reduction technique well-suited for embedding high-dimensional data for visualization in a low-dimensional space of two or three dimensions. What t-SNE does is to find a way to project data into a low dimensional space, so that the clustering in the high dimensional space is preserved. In mathematics, it is an unsupervised learning algorithm to construct a data distribution in low dimensional space which has a similar distribution in high dimensional space.

That is, given a set of N high-dimensional data points $\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_N$ $(\mathbf{x}_i \in \mathbb{R}^m)$, t-SNE first calculate the conditional probability to represent similarity of two data points, based on the distance between two points, as follows,

$$p_{j|i} = \frac{\exp(-\|\mathbf{x}_i - \mathbf{x}_j\|/2\sigma_i)}{\sum_{k \neq i} \exp(-\|\mathbf{x}_i - \mathbf{x}_j\|/2\sigma_i)} \tag{1}$$

In Equation 1, the σ_i is a hyper-parameter determined by a predefined perplexity $Perp$. That is, it should satisfy

$$Perp = 2^{H(p_i)}, \text{ where } H(p_i) = -\sum_j p_{j|i} \log p_{j|i} \tag{2}$$

T-SNE aims to learn a low dimension representation $\mathbf{y}_1, \mathbf{y}_2, \ldots, \mathbf{y}_N$ $(\mathbf{y}_i \in \mathbb{R}^d, d < m)$ that reflects the similarities $p_{j|i}$ as well as possible. To this end, similarities in low dimensional space is also measured, but in a different way.

$$q_{j|i} = \frac{(1 + \|\mathbf{x}_i - \mathbf{x}_j\|)^{-1}}{\sum_{k \neq i}(1 + \|\mathbf{x}_i - \mathbf{x}_k\|)^{-1}} \tag{3}$$

The locations of the points $\mathbf{y}_i$ in the map are determined by minimizing the (non-symmetric) Kullback–Leibler divergence of the distribution.

$$C = \sum_i KL(P_i \| Q_i) = \sum_i \sum_j p_{j|i} \log \frac{p_{j|i}}{q_{j|i}} \tag{4}$$

There are two things that need to be noted, which guarantee a good clustering effect. The first is the t-distribution. As we can see in Fig. 1, data points with higher similarity tend to be nearer and data points with lower similarity tend to be farther. The second is inside the cost function. It tends to penalize more on points with large distance in low-dim space but small distance in high-dim space (p big q small). This means t-SNE tends to preserve local distribution.

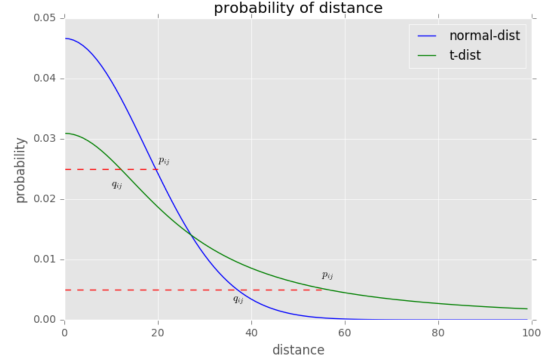


Fig. 1. Comparison between normal distribution and t-distribution

G. Locally Linear Embedding (LLE)

Locally Linear Embedding (LLE) is an unsupervised learning algorithm that computes low-dimensional, neighborhood-preserving embeddings of high-dimensional inputs. [6] "Locally linear" means to use nearest neighbors to linearly reconstruct the target data point. "Embedding" means to learn a representation in low-dimensional space to preserve the neighborhood relationship. Fig. 2 shows an illustration of the neighborhood-preserving mapping discovered by LLE, which discovers the global internal coordinates of the manifold without signals that explicitly indicate how the data should be embedded in two dimensions.

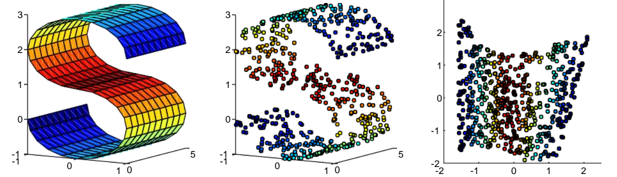


Fig. 2. Illustration of the neighborhood-preserving mapping discovered by LLE

Mathematically, given a set of N high-dimensional data points $\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_N$ $(\mathbf{x}_i \in \mathbb{R}^m)$, LLE aims to obtain a low-dim representation $\mathbf{y}_1, \mathbf{y}_2, \ldots, \mathbf{y}_N$ $(\mathbf{y}_i \in \mathbb{R}^d, d < m)$. It can be divided into 3 steps: (i) finding nearest neighbors (ii) linear reconstruction (iii) low dimensional embedding.

Step 1: Finding nearest neighbors

For every data point $\mathbf{x}_i$, select its $k-$nearest neighbors, based on Euclidean distance. k is a hyper-parameter in this algorithm.

Step 2: Linear reconstruction

Let the set of the $k-$nearest neighbors of $\mathbf{x}_i$ be $\mathcal{N}_k(\mathbf{x}_i)$. Initialize a matrix $M \in \mathbb{R}^{N\times N}$, for $\mathbf{x}_j \notin \mathcal{N}_k(\mathbf{x}_i)$, let $W_{ij} = 0$. LLE first optimizes this reconstruction error to get the optimal reconstruction.

$$\begin{aligned} \min_{W} \quad & \epsilon(W) = \sum_{i=1}^{N} \|\mathbf{x}_i - \sum_{j\in\mathcal{N}_k(\mathbf{x}_i)} W_{ij}\mathbf{x}_j\|^2 \\ s.t. \quad & W\mathbf{1} = \mathbf{1} \end{aligned} \tag{5}$$

Step 3: Low-dimensional Embedding

After calculating the local relationship in the high dimensional space, LLE will learn a data representation in low-dim space to preserve this relationship. *I.e.*, LLE will minimize the equation below

$$\begin{aligned} \min_{Y} \quad & \Phi(Y) = \sum_{i=1}^{N} \|\mathbf{y}_i - \sum_{j=1}^{N} W_{ij}\mathbf{y}_j\|^2 \\ s.t. \quad & \mathbf{y}_i^\top \mathbf{y}_i = 1 \\ & \mathbf{y}_i^\top \mathbf{1} = 0 \end{aligned} \tag{6}$$

We need the first constraint because $\mathbf{y}_i = \mathbf{0}$ is a trivial solution. The second constraint is a zero-mean condition. Since $W\mathbf{1} = \mathbf{1}$, adding a bias to $\mathbf{y}_i$ will not affect the optimality.

H. Multi-Dimensional Scaling (MDS)

Multidimensional Scaling (MDS) [7] is also a manifold learning method. In this project, we only discuss classical MDS, which is also known as Principal Coordinates Analysis (PCoA). In MDS with classical scaling, the inputs are projected into the subspace that best preserves their pairwise distance. It takes an input matrix giving dissimilarities between pairs of items and outputs a coordinate matrix whose configuration minimizes a loss function called “strain”.

In mathematics, this problem can be formulated as given the distance matrix of N data points $D = (d_{ij}) \in \mathbb{R}^{N\times N}$, with $d_{ij} = dist(\mathbf{x}_i, \mathbf{x}_j)$. MDS aims to find the coordinates of these data points $\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_N$ ($\mathbf{x}_i \in \mathbb{R}^d$). To this end, MDS transforms “approximating distances” into “approximating inner products”.

Below shows the mathematical derivations in MDS. Let $B = (b_{ij}) \in \mathbb{R}^{N\times N}$ be the inner product matrix of the N data points, *i.e.* $b_{ij} = \mathbf{x}_i^\top \mathbf{x}_j$. It is obvious that

$$d_{ij}^2 = b_{ii} + b_{jj} - 2b_{ij} \tag{7}$$

MDS aims to represent B in the form of D. The basic assumption is that the N data points are decentralized.

$$\sum_{i=1}^{N} \mathbf{x}_i = \mathbf{0} \tag{8}$$

Then we have, the sum of any row or column of B is zero.

$$\sum_{i=1}^{N} b_{ij} = \sum_{i=1}^{N} \mathbf{x}_i^\top \mathbf{x}_j = \left(\sum_{i=1}^{N} \mathbf{x}_i^\top\right) \mathbf{x}_j = 0 \tag{9}$$

Based on Eq. 7, Eq. 8 and Eq. 9, we have

$$\begin{aligned} \sum_{i=1}^{N} d_{ij}^2 &= \sum_{i=1}^{N} b_{ii} + N b_{jj} \\ \sum_{j=1}^{N} d_{ij}^2 &= \sum_{i=1}^{N} b_{ii} + N b_{ii} \\ \sum_{i=1}^{N}\sum_{j=1}^{N} d_{ij}^2 &= 2N \sum_{i=1}^{N} b_{ii} \end{aligned} \tag{10}$$

Then we can calculate b_{ij}

$$b_{ij} = -\frac{1}{2}\left(d_{ij}^2 - \frac{1}{N}\sum_{i=1}^{N} d_{ij}^2 - \frac{1}{N}\sum_{j=1}^{N} d_{ij}^2 + \frac{1}{N^2}\sum_{i=1}^{N}\sum_{j=1}^{N} d_{ij}^2\right) \tag{11}$$

Let the matrix $H = \mathbf{I} - \frac{1}{N}\mathbf{1}\mathbf{1}^\top$ and $\bar{D} = (d_{ij}^2)$, we can represent B as

$$B = -\frac{1}{2} H \bar{D} H \tag{12}$$

Then our optimization problem becomes clear

$$\min_{X\in\mathbb{R}^{d\times N}} \|B - X^\top X\|_F^2 \tag{13}$$

This problem has a closed-form solution

$$X = \Lambda_m^{\frac{1}{2}} E_m \tag{14}$$

where Λ_m is the diagonal matrix composed of d largest eigenvalues of B, E_m is the matrix composed of the corresponding d eigenvectors.

I. Auto-Encoder (AE)

An autoencoder is a type of artificial neural network used to learn efficient data codings in an unsupervised manner. [9] The aim of an autoencoder is to learn a representation (encoding) for a set of data, typically for dimensionality reduction. It is a neural network that learns to copy its input to its output. It has a hidden layer that describes a code used to represent the input, and it is constituted by two main parts: an encoder that maps the input into the code, and a decoder that maps the code to a reconstruction of the original input.

The overall architecture of an autoencoder is shown in Fig. 3.To specify the algorithm of autoencoder in mathematics, it takes the input $\mathbf{x} \in \mathcal{X} \subset \mathbb{R}^m$ and maps it to $\mathbf{h} \in \mathcal{F} \subset \mathbb{R}^d$.

An autoencoder consists of two parts, the encoder and the decoder, which can be defined as transitions ϕ and ψ, such that:

$$\begin{aligned} &\phi : \mathcal{X} \to \mathcal{F} \\ &\psi : \mathcal{F} \to \mathcal{X} \\ &\phi, \psi = \arg\min_{\phi,\psi} \|X - (\phi \circ \psi)(X)\|^2 \end{aligned} \tag{15}$$

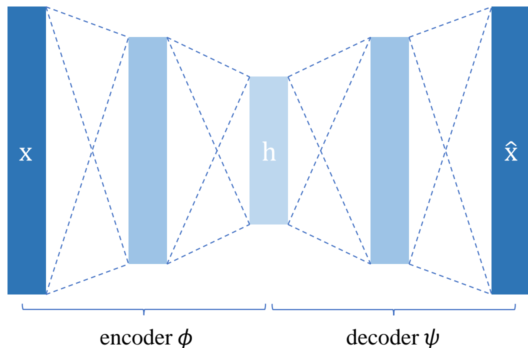


Fig. 3. Illustration of a typical autoencoder.

Since the feature space $\mathcal{F}$ have lower dimensionality than the input space $\mathcal{X}$, the feature vector $\mathbf{h} = \phi(\mathbf{x})$ can be regarded as a compressed representation of the input $\mathbf{x}$.

III. Experiment

A. Dataset

Download Animals with Attributes (AwA2) dataset [8]. This dataset consists of 37322 images of 50 animal classes with 2048-dimensional pre-extracted deep learning features for each image. We randomly split the images in each category into 60% for training and 40% for testing, using stratified sampling. We also use 5-fold cross-validation within the training set to select optimal dimensionality.

B. Support Vector Classifier

In all following experiments, we use Support Vector Classifier (SVC) to classify processed data. The implementation is *sklearn.svm.SVC*, with linear kernel and one-vs-rest decision function. The regularization parameter C is set as 1.0. As a baseline, we also perform grid search on C, using original data features.

TABLE I
Accuracy of SVC with different values of C

C	Accuracy
0.125	92.45%
0.25	92.45%
0.5	92.44%
1	92.39%
2	92.30%
4	92.24%
8	92.20%

C. Feature Selection by Variance

Using variance as the criterion, we select top 2–2000 features with largest variance and evaluate them with SVM. Results are given in Table I. Selection-by-variance works surprisingly well given its simplicity. Trained with approximately 1/8 of original features (250), SVM gives an accuracy within 97.5% of the accuracy when all features are used.

TABLE II
Accuracy with top k features, selected by variance

k	Accuracy
2	9.85%
5	24.80%
10	46.97%
50	81.70%
100	87.92%
250	90.13%
500	91.60%
1000	92.17%
2000	92.87%

We also observe that, in our case, the effect of doubling the number of features, from 1 000 to 2 000, is only a 0.7% increase in accuracy, suggesting existence of redundant information in the additional 1 000 features. We hypothesize that, if the top 1 000 features give good performance on SVM, the last 1 000, with redudant information, should also give comparable performance. We test our hypothesis by training the SVM with top k features with *least* variance, giving the results in Table III. Our results confirm our hypothesis.

TABLE III
Accuracy with top k features, selected by least variance

k	Accuracy
2	6.46%
5	10.36%
10	18.84%
50	55.22%
100	71.65%
250	82.74%
500	87.32%
1000	90.19%
2000	92.83%

D. PCA and Kernel PCA

We use *sklearn.decomposition.KernelPCA* module to implement PCA, with kernel coefficient $\gamma = 1/n_{features}$, polynomial degree $d = 3$, bias term $c_0 = 1$. To accelerate computation, we set parallel jobs $n_jobs = 4$. Please note that large number of parallel jobs may lead to memory error. Table IV and Figure 4 show the result, from which we know that RBF kernel performs worst and cosine kernel performs best. All kernels improve accuracy when dimension increase, however, when dimension is greater than 32, then accuracy gain is much more slower. Therefore $dimension = 32, kernel = cosine$ may be a reasonable choice of parameters.

Also notice that when dimension is less than 16, the cosine kernel outperforms all other kernels, when dimension is greater than 16, cosine and polynomial have almost same accuracy. The performance gap between RBF kernel and other kernels increase with higher dimension.

To further illustrate effect of PCA, we also visualize 2-dimension linear PCA results as in Figure 5. The visualization shows that PCA indeed maximizes data variance, since most

TABLE IV
ACCURACY OF PCA WITH DIFFERENT KERNELS AND DIMENSIONS

Dimension	Linear	RBF	Poly	Sigmoid	Cosine
2	18.47%	17.31%	19.98%	17.14%	19.56%
4	44.85%	43.85%	44.44%	42.62%	47.63%
8	69.47%	65.58%	68.54%	66.48%	71.18%
16	82.60%	77.20%	82.80%	81.63%	83.04%
32	87.84%	83.64%	88.82%	88.30%	89.21%
64	89.22%	86.32%	90.87%	90.63%	91.10%
128	89.95%	88.00%	91.77%	91.54%	92.03%
256	90.73%	89.02%	92.17%	92.07%	92.39%
512	91.62%	89.80%	92.40%	92.23%	92.66%
1024	92.06%	90.18%	92.47%	92.33%	92.75%
2048	92.30%	90.84%	92.67%	92.34%	92.85%

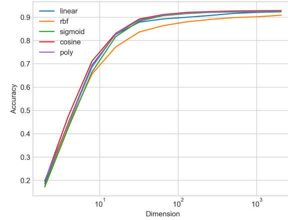


Fig. 4. Accuracy-Dimension curve of PCA with different kernels. The x-axis is log-scale axis, since the dimension is increasing exponentially.

of data are distributed in a large range. However, PCA is unsupervised model, therefore data with different labels are not separated.

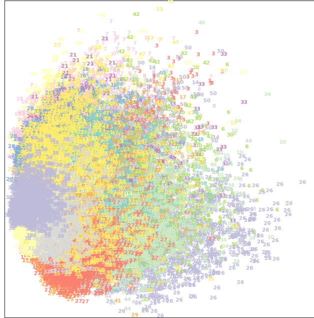


Fig. 5. Visualization of 2-dimensional linear PCA result, where numbers denotes class labels. We also use different colors for different class labels.

E. LDA

LDA is implemented by *sklearn.discriminant_analysis.LinearDiscriminantAnalysis*, with default parameters. We use default solver *svd*, which does not rely on the calculation of the covariance matrix. This can be an advantage in our experiment since the number of features is large. Note that LDA requires $n_component \leq \min(n_classes - 1, n_features)$, therefore the grid search range is limited to [1,50). Table V shows part of result, and Figure 6 show the whole results. The figure indicates that accuracy grows with dimension, however when dimension is greater than 30, accuracy grows much slower. Then we can assume 30 is a reasonable choice of dimension for LDA. Also, such choice is consistent with results in PCA in previous section.

TABLE V
ACCURACY OF LDA WITH DIFFERENT DIMENSIONS

Dimension	Accuracy
1	19.67%
2	29.60%
4	51.03%
8	57.33%
16	79.19%
32	88.85%
49	91.18%

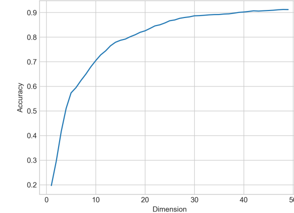


Fig. 6. Accuracy-Dimension curve of LDA

To compare effect of LDA and PCA, we visualize 2-dimension LDA results as in Figure 7. Since LDA is supervised model, it separates different classes. However, 2 dimensions are not enough to separate 50 classes in this dataset, therefore some classes are overlapped.

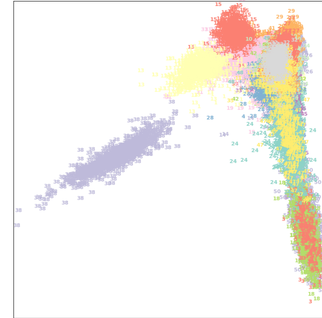


Fig. 7. Visualization of 2-dimensional LDA result, where numbers denotes class labels. We also use different colors for different class labels.

F. FA

Result of FA is illustrated in Table VI and Figure 8. Here we compare FA with linear PCA, and find that them perform similar when $D < 10$, FA outperforms linear PCA when $10 < D < 100$,m and FA performs worse than linear PCA when $400 < D$. This might result from the heteroscedastic property of FA, i.e. when dimension is low, it is helpful to consider independent variance of each dimension, but when dimension is high, such consideration might be too complex and it might lead to over-fitting.

We also visualize 2-dimensional FA as Figure 9. The result is very similar with linear PCA, which further validates theoretical similarity between FA and PCA.

TABLE VI
ACCURACY OF FA WITH DIFFERENT DIMENSIONS

Dimension	Accuracy
2	18.18%
4	44.15%
8	69.90%
16	83.57%
32	88.58%
64	90.04%
128	90.06%
256	90.63%
512	91.23%
1024	91.62%
2048	91.62%

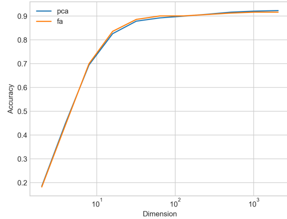


Fig. 8. Accuracy-Dimension curve of FA compared with linear PCA. The x-axis is log-scale axis, since the dimension is increasing exponentially.

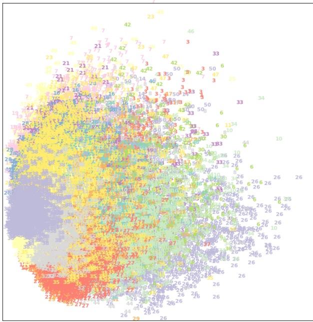


Fig. 9. Visualization of 2-dimensional FA result, where numbers denotes class labels. We also use different colors for different class labels.

G. t-Distributed Stochastic Neighbor Embedding (t-SNE)

We use *sklearn.manifold.TSNE* module to implement t-SNE. There several hyper-parameters we need to choose. Similar to mentioned above, *n_components* is the reduced dimension number. The parameter *init* is to determine how the low dimensional data are initialized. There are two ways – *pca* and *random*, which initialized the output in the form of PCA representation and in a random manner respectively. In our experiment, we only use pca for initialization by default. The parameter *method* indicates how the gradient will be calculated. If it is *exact*, then t-SNE will compute the real gradient on each data point, which is $O(N^2)$. However, *barnes_hut* leads to an approximation of the real gradient based on Barnes-Hut tree, which is $O(N \log N)$. But this approximation can only be used when *n_components=2* or *n_components=3*. In this experiment, we focus on the classification and visualization of t-SNE, and we also analyze the drawbacks of t-SNE.

We first compare the results of using Barnes-Hut approximation for gradients and using the exact gradients during optimization. Tab. VII shows the result. We find that the performance of these two methods are similar to each other. However, during experiment, we find that using Barnes-Hut approximation will accelerate the algorithm by more than 100x, *i.e.* the computation time and resource are dramatically saved.

TABLE VII
ACCURACY OF T-SNE USING DIFFERENT GRADIENT CALCULATION METHODS

Dimension	Accuracy	
	Barnes Hut	exact
2	86.90%	87.50%
3	87.80%	86.70%

Since t-SNE depends heavily on computational resources, in the official documentation of scikit-learn [10], it suggests that *It is highly recommended to use another dimensionality reduction method (e.g. PCA for dense data or TruncatedSVD for sparse data) to reduce the number of dimensions to a reasonable amount (e.g. 50) if the number of features is very high.* Same as above, we conduct this experiment on 2 and 3 dimensions with Barnes-Hut approximation. We tried 3 methods:

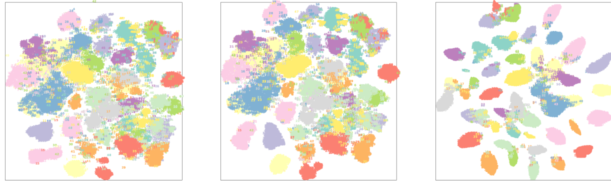
1) use raw data as input of t-SNE
2) pre-reduce to 64-dim using PCA
3) pre-reduce to 49-dim using LDA

Tab. VIII shows the result. During experiment we found that preprocessing the raw data will increase time efficiency, which is the same as our expectation. From tab. VIII we also find that it is clear that preprocessing data using LDA will dramatically boost the performance!

TABLE VIII
ACCURACY OF T-SNE USING DIFFERENT PREPROCESSING METHODS

Dimension	Accuracy		
	raw	PCA to 64	LDA to 49
2	86.90%	86.90%	**95.40%**
3	87.80%	87.50%	**95.50%**

This result is reasonable if we think over the motivation of LDA – to maximize the distances of clusters and the tightness of each cluster, *i.e.* maximize $\sigma_{between}$ and minimize σ_{within}. We can get a more intuitive understanding if we visualize the data, as is shown in Fig. 10. We find it obvious that the "clusters" in Fig. 10(c) (using LDA as preprocessing) become farther and each "cluster" becomes tighter.



(a) t-SNE 2D Visualization without preprocessing (b) t-SNE 2D Visualization with PCA preprocessing (c) t-SNE 2D Visualization with LDA preprocessing

Fig. 10. Illustration of 2D t-SNE with different preprocessing tricks

It seems that "LDA then t-SNE" is the best configuration – even 2 dimension leads to accuracy of 95+% on validation data. However, we still conducted experiments with the number of dimensions as the variant. This time, we all use the exact gradient for fair comparison. Due to the computational limitation, we only did experiments of dimensions $\{2,3,4,8,16\}$. However, it is enough to show the problem. Tab. IX shows that higher dimension leads to worse performance. Maybe it is because higher dimension makes it harder to converge. In short, it is not only slow but with performance not as good as "LDA then t-SNE".

TABLE IX
ACCURACY OF T-SNE WITH DIFFERENT DIMENSIONS

Dimension	Accuracy
2	87.50%
3	86.70%
4	87.00%
8	78.00%
16	77.80%

Therefore, "LDA then t-SNE" with 2 or 3 dimensions is not only a good method for dimensionality reduction, but also a good algorithm for data visualization.

H. Locally Linear Embedding (LLE)

We use *sklearn.manifold.LocallyLinearEmbedding* module to implement LLE. There are 2 hyper-parameters we need to choose – the reduced data dimension and the number of neighbors to reconstruct each data point. All the other parameters are chosen as default. Due to the computational limitation, we only conduct experiment on $\{2,8,32,128\}$ dimensions and $\{4,16,64\}$ nearest neighbors. Tab. X shows the validation accuracy of the reduced data using linear SVC. Unfortunately, we found the performance is dramatically lowered – only an accuracy of no more than or near 10%. Though the performance is better with higher dimensions and more neighbors, the performance is still not acceptable.

However, if we change the linear kernel in SVC to an rbf kernel, we see a dramatic increase in performance, as is shown in Tab. XI. In Sec. IV-A, we visualize the reduced data and give a possible explanation. As we can see, reduced data with 128 dimensions reconstructed by 64 nearest neighbors will lead to the best performance, which is good enough and comparable to PCA then SVC with rbf kernel.

TABLE X
ACCURACY OF LLE WITH DIFFERENT DIMENSIONS AND NEIGHBORS (CLASSIFIED WITH LINEAR KERNEL)

	#neighbor=4	#neighbor=16	#neighbor=64
#dim=2	4.6%	4.5%	4.3%
#dim=8	4.3%	4.4%	4.2%
#dim=32	4.5%	4.6%	5.9%
#dim=128	**11.2%**	9%	8.4%

TABLE XI
ACCURACY OF LLE WITH DIFFERENT DIMENSIONS AND NEIGHBORS (CLASSIFIED WITH RBF KERNEL)

	#neighbor=4	#neighbor=16	#neighbor=64
#dim=2	10.9%	34.9%	38.4%
#dim=8	43.4%	65.5%	74.7%
#dim=32	84.2%	84.9%	87.4%
#dim=128	88.7%	89.2%	**90.8%**

I. Multi-Dimensional Scaling (MDS)

We use *sklearn.manifold.MDS* module to implement MDS. We conduct experiments of metric MDS by setting the parameter *metric* to be true by default. The distance metric in this algorithm is Euclidean distance by default. We explore the performance of MDS with different reduced dimensions. Unfortunately, due to the computational limitation, we only do experiments of $\{2,4,8,16\}$ dimensions. Tab. XII shows the result. We find the result close to that in PCA. But due to the low efficiency, this algorithm may not be acceptable.

TABLE XII
ACCURACY OF MDS WITH DIFFERENT DIMENSIONSS (CLASSIFIED WITH RBF KERNEL)

Dimension	Accuracy
2	18.80%
4	39.30%
8	51.70%
16	**73.40%**

Actually, we think MDS has a very close relationship with PCA, mathematically. In Sec. IV-A, we visualize the reduced data and find it similar to the visualization of PCA, which confirms our thoughts. However, due to the low efficiency and high computational demands of this algorithm, we do not think MDS is a good way for dimensionality reduction. But may be it can work well in some data analysis tasks. For example, it may work well in analyzing similarity or dissimilarity data.

J. Auto-Encoder (AE)

In this section, we implemented a simple autoencoder using PyTorch [11]. The structure of the encoder ϕ and the decoder ψ are simple, both consisting 4 linear layers and 3 ReLUs between each linear layer. To expatiate, the dimensions of input and hidden layers in ϕ are 2048, 1024, 512, 512, d,

and the dimensions of hidden layers in ψ and the output are d, 512, 512, 1024, 2048. In our experiment, we choose the encoded dimension in $\{2, 4, 8, 16, 32, 64, 128, 256, 512\}$.

As for training, we simply use the stochastic gradient descent (SGD) method with batch size 128. The learning rate shrinks from 10^{-3} to 10^{-4} exponentially for 200 epochs. We trained each autoencoder to convergence on a single NVIDIA RTX2080Ti and used the encoded $d-$dimensional data as new data representation. Tab. XIII shows the performance. We used both linear kernel and rbf kernel for classification. We find the performance is similar. As we can see, the optimal dimension is 128 or 256. Lower dimension my suffer from insufficient data representation, while higher dimension may lead the network harder to train. During experiment, we also find that this straight-forward method guarantees much more efficiency compared to MDS, LLE, etc..

TABLE XIII
ACCURACY OF AUTOENCODER WITH DIFFERENT DIMENSIONS

Dimension	Accuracy	
	linear	rbf
2	70.30%	62.60%
4	85.30%	87%
8	87.20%	89.90%
16	89%	90.40%
32	89.30%	91.70%
64	90.80%	92.80%
128	**91%**	92.80%
256	90.90%	**93.10%**
512	90.40%	92.70%

Now we propose a possible explanation of the performance of using autoencoder for dimensionality reduction. Fig. 11 shows the reconstruction loss. As we can see, the reconstruction loss achieved the lowest point on 128 and 256 (they are very close). However, the reconstruction loss corresponding to $d = 512$ is higher than these two. So may be the reconstruction loss somehow reflect the performance.

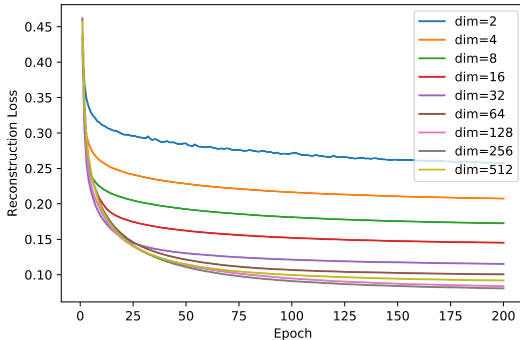


Fig. 11. Reconstruction loss of different dimension number.

IV. Further Discussion

A. Feature Learning: What are they learning?

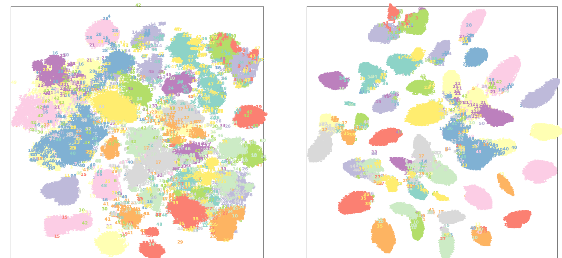
During the implementation of different feature learning methods, we found hard to understand the difference of the behaviors of different methods, though their motivations are clear enough. So this section provides a brief discussion to illustrate some of our thoughts about what on earth are these algorithms learning, *i.e.* the properties of the reduced data tending to have, and also their possible reasons. These algorithms do have similarities in that they all aims to preserve something in data representation, be it the pairwise distances or local properties. However, if we visualize them they are totally different, we summarize the behavior of each feature learning algorithm as follows.

- t-SNE: learn the clustering of the raw data implicitly
- LLE: place most data points on a very small number of axises
- MDS: treat the data globally, very similar behavior to that of PCA
- AutoEncoder: transform the raw data to a radical-pattern representation

We first begin with t-SNE. In the formulation of t-SNE, there is not anything concerning "clustering". It even seems similar to the formulation of MDS, which aims to preserve the distances between each pair of data points. However, t-SNE aims to preserve the **relative** distance instead. This is achieved by assigning a conditional distribution to each data point, as is shown in Eq. 1. So what leads to the clustering effect of t-SNE?

We think 2 main factors contribute to this. First is in the loss function, as is shown in Eq. 4. Since the non-symmetric property of Kullback–Leibler divergence, the loss tends to penalize more on large $p_{j|i}$ modeled by small $q_{j|i}$. Hence, t-SNE preserves local similarity structure of the data. The second reason is in the choice of t-distribution when modeling the reduced data. It will lead similar data points closer and set dissimilar data points apart (see Fig. 1).

As a result, t-SNE actually learns a clustered representation of the raw data. As is shown in Fig. 12, t-SNE on LDA-preprocessed data leads to a better clustering, same as the motivation of LDA.

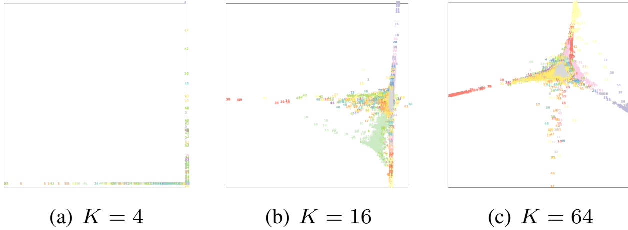


(a) t-SNE 2D Visualization without preprocessing (b) t-SNE 2D Visualization with LDA preprocessing

Fig. 12. Illustration of 2D t-SNE

Then we analyze the behavior of LLE. As we can see in Fig. 13, we find that LLE tends to place most data points on a very small number of "axises". Also, we find that more neighbors to reconstruct leads to thicker axises. This can be

explained by the pattern of the loss function in LLE. To simplify, that is to minimize $\|y_i - \sum_{j\in\mathcal{N}(y_i)} w_j y_j\|$ where w is the pre-calculated reconstruction coefficient. Therefore, when the number of neighbors chosen is small, the reduced data tends to form straight lines because only this will lead to the smallest loss. This distribution is clearly not friendly to linear SVC, leading to the results in Tab. X. However, this algorithm do preserve local relationship in a very compact way. That is the reason why its performance is not bad using kernel SVC (see Tab. XI).



(a) $K = 4$ (b) $K = 16$ (c) $K = 64$

Fig. 13. Illustration of 2D LLE with different K

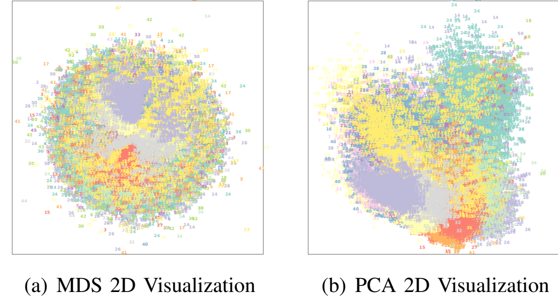
Now we show the connections between MDS and PCA. As we can see in Fig. 14, both MDS and PCA do not set data points in different classes apart but let them remain "combining" to each other. The motivation of MDS is clear – to preserve the pairwise distance in reduced data. But MDS does not minimize the loss *w.r.t.* the distance matrix explicitly. Instead, it uses a trick to represent the inner product matrix *w.r.t.* the distance matrix, and then optimizes on this inner product matrix. So MDS is suitable in problems like this: we have known the distance matrix of several data points, then how to reconstruct the coordinates of these data points. We now focus on the last step of both algorithms. Let the inner product matrix be B.

In PCA, the last step is $Bv = \lambda v$. Then we choose the eigenvectors of top-d eigenvalues to compose the matrix V. In this way, $V^\top X$ is the reduced data. In MDS, the last step is $\min_Y \|B - Y^\top Y\|$. Then we choose the eigenvectors of top-d eigenvalues to compose the matrix V, and let Λ be a diagonal matrix composed ot the d eigenvalues. In this way $\Lambda^{1/2}V$ is the reduced data.

In short, MDS and PCA are similar in that (i) both of them considers the global relationship of the raw data, and (ii) both of them are related to eigenvalues and eigenvectors of the inner product matrix $B = X^\top X$.

The behavior of autoencoder is also worth mentioning. As is shown in Fig. 15, autoencoder tends to transform the raw data to a radical-pattern representation. Maybe it is because data with this pattern is the easiest to reconstruct the raw data. *I.e.* each direction corresponds to one pattern of reconstruction. Also, we think that directions of the rays simply corresponds to the label of the data point.

In this section, we analyze some interesting phenomenon in feature learning algorithms, which may also provide another way to boost our understanding of these seemingly abstruse methods.



(a) MDS 2D Visualization (b) PCA 2D Visualization

Fig. 14. Illustration of 2D MDS, with PCA for comparison

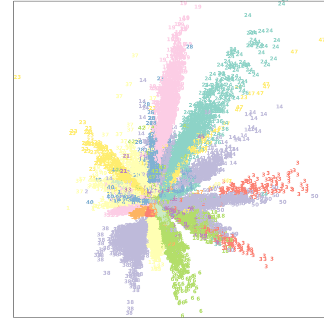


Fig. 15. Illustration of 2D AutoEncoder

B. Overall Comparison

In this section, we summarize the optimal configuration of each dimensionality reduction method. Note that "optimal" not only refers to accuracy, but also refers to computational efficiency concerning time and resource demands. Tab. XIV shows the result.

TABLE XIV
SUMMARY OF THE BEST CONFIGURATION OF ALL METHODS USED IN THIS REPORT

Method	Accuracy	Dimension	Other Configurations
Selection-by-Variance	92.17%	1000	
PCA	91.62%	512	
Kernel PCA	92.39%	256	use cosine kernel
LDA	88.85%	32	
FA	91.23%	512	
t-SNE	**95.40%**	**2**	preprocess using LDA
LLE	90.80%	128	use 64-NN
MDS	73.40%	16	
AutoEncoder	93.10%	256	

There are also many experimental conclusions in this project. We summarize them below.

- In the total 2000+ dimension features, 1000 are redundant. We test our hypothesis by training the SVM with top k features with least variance.
- In PCA, all kernels improve accuracy when dimension increase, however, when dimension is greater than 32, then accuracy gain is much more slower.

- In LDA, accuracy grows with dimension, however when dimension is greater than 30, accuracy grows much slower.
- The results of FA shows its heteroscedastic property, *i.e.* when dimension is low, it is helpful to consider independent variance of each dimension, but when dimension is high, such consideration might be too complex and it might lead to over-fitting.
- "LDA then t-SNE" with 2 or 3 dimensions is not only a good method for dimensionality reduction, but also a good algorithm for data visualization.
- The accuracy of LLE increases with more dimensions and more neighbors, but it is still not an ideal way because of the high computational cost.
- MDS is not an efficient method for dimensionality reduction due to its computational demands, but it is a suitable algorithm for data reconstruction tasks based on pairwise similarity
- AutoEncoder is a straight-forward way for dimensionality reduction of great efficiency and good performance. However, high dimensions may lead to difficulty of training the network.
- Patterns of the data derived from different feature learning methods are analyzed, which are interesting and provide us with some intuitive understanding.

V. Conclusion

In this project report, we evaluated the performance of some popular dimensionality reduction algorithms on Animals with Attributes (AwA2) dataset, including variance feature selection, PCA, kernel PCA, FA, LDA, t-SNE, LLE, MDS and AE. Our experiment shows that t-SNE with LDA preprocessing gives the best performance, achieving 95.40% accuracy with only two dimensions.

References

[1] K. Pearson, "Liii. on lines and planes of closest fit to systems of points in space," *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science*, vol. 2, no. 11, pp. 559–572, 1901.

[2] B. Schölkopf, A. Smola, and K.-R. Müller, "Nonlinear component analysis as a kernel eigenvalue problem," *Neural computation*, vol. 10, no. 5, pp. 1299–1319, 1998.

[3] R. A. Fisher, "The use of multiple measurements in taxonomic problems," *Annals of eugenics*, vol. 7, no. 2, pp. 179–188, 1936.

[4] L. L. Thurstone, "Multiple factor analysis.," *Psychological review*, vol. 38, no. 5, p. 406, 1931.

[5] V. D. M. Laurens and G. Hinton, "Visualizing data using t-sne," *Journal of Machine Learning Research*, vol. 9, no. 2605, pp. 2579–2605, 2008.

[6] S. T. Roweis and L. K. Saul, "Nonlinear dimensionality reduction by locally linear embedding," *science*, vol. 290, no. 5500, pp. 2323–2326, 2000.

[7] A. Mead, "Review of the development of multidimensional scaling methods," *Journal of the Royal Statistical Society: Series D (The Statistician)*, vol. 41, no. 1, pp. 27–39, 1992.

[8] Y. Xian, C. H. Lampert, B. Schiele, and Z. Akata, "Zero-shot learning—a comprehensive evaluation of the good, the bad and the ugly," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 41, no. 9, pp. 2251–2265, 2019.

[9] M. A. Kramer, "Nonlinear principal component analysis using autoassociative neural networks," *AIChE journal*, vol. 37, no. 2, pp. 233–243, 1991.

[10] S. learn 0.22.2 Documentation, "sklearn.manifold.tsne." https://scikit-learn.org/stable/modules/generated/sklearn.manifold.TSNE.html.

[11] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer, "Automatic differentiation in pytorch," 2017.