Project 1: Dimensionality Reduction

Sijia Li 518030910294 qdhlsj@sjtu.edu.cn

1 Experiment Requirements

1. Download Animals with Attributes (AwA2) dataset. This dataset consists of 37322 images of 50 animal classes with pre-extracted deep learning features for each image. Split the images in each category into 60% for training and 40testing. You can use K-fold cross-validation within the training set to determine hyper-parameters, such as C in SVM.
2. Use linear SVM for image classification based on the deep learning features.
3. Reduce the dimensionality of deep learning features using three methods (one feature selection method, one feature projection method, one feature learning method) and perform image classification again based on the obtained low-dimensional features. Record the performance variance w.r.t. different feature dimensionality.
4. Explore the optimal dimensionality reduction method and the optimal dimensionality.
5. Summarize your experimental results and write a project report in English. The project report should contain experimental setting (i.e., dataset, feature, training/testing split), the dimensionality reduction methods you tried, the experimental results you obtained, and the experimental observations based on your experimental results.

2 Method Theory

2.1 Feature Selection

2.1.1 Forward Selection

Forward selection begins with a model that contains no variables. Then it starts adding the most significant variables one after the other until a pre-specified stopping rule is reached or all the variables under consideration are included. For the code detail, we use the package in sklearn: *sklearn.feature_selection* as follows:

```
knn = KNeighborsClassifier(n_neighbors=n_comp)
selector = SequentialFeatureSelector(knn, n_features_to_select=n_comp,direction="forward")
selector.fit(X_train, y_train)
```

Algorithm 1 Sequential forward feature set generation - SFG.

function SFG(F - full set, U - measure)
  initialize: $S = \{\}$ ▷ S stores the selected features
  repeat
    $f = \text{FINDNEXT}(F)$
    $S = S \cup \{f\}$
    $F = F - \{f\}$
  until S satisfies U *or* $F = \{\}$
  return S
end function

图 1: Forward Selection

2.1.2 Backward Selection

Backward selection begins with a model that contains all variables. Then it starts removing the least significant variables one after the other until a pre-specified stopping rule is reached or no variable is left in the model.

Algorithm 2 Sequential backward feature set generation - SBG.

function SBG(F - full set, U - measure)
  initialize: $S = \{\}$ ▷ S holds the removed features
  repeat
    $f = \text{GETNEXT}(F)$
    $F = F - \{f\}$
    $S = S \cup \{f\}$
  until S does not satisfy U *or* $F = \{\}$
  return $F \cup \{f\}$
end function

图 2: Backward Selection

For the code detail, we use the package in sklearn: *sklearn.feature_selection* as follows:

```
knn = KNeighborsClassifier(n_neighbors=n_comp)
selector = SequentialFeatureSelector(knn, n_features_to_select=n_comp,direction="backward")
selector.fit(X_train, y_train)
```

2.2 Feature Projection

2.2.1 PCA (Principle Component Analysis)

PCA transformation is linear transformation that transforms a set of correlated variables into a smaller number of uncorrelated variables called principal components. One interpretations of PCA is

maximum variance direction: when X is decentralized,

$$\frac{1}{n}\sum_{i=1}^{n}(v^T x_i)^2 = \frac{1}{n}v^T XX^T v$$

$$max_v v^T XX^T v$$

$$s.t. v^T v = 1$$

The Lagrangian form is $L_v = v^T XX^T v + \lambda(1 - v^T v)$a, resulting in $XX^T v = \lambda v$, which is the eigen decompostion.

For the code detail, we use *KernelPCA* in the package *sklearn.decomposition* as follows:

```
transformer = KernelPCA(n_components=n_comp, kernel=Kernel, copy_X=True, n_jobs=8)
transformer.fit(X_train)
X_train_proj = transformer.transform(X_train)
X_test_proj = transformer.transform(X_test)
acc = SVMmodel.runSVM(X_train_proj, X_test_proj, y_train, y_test, C, Kernel)
```

2.2.2 LDA (Linear Discriminative Analysis)

LDA is typically used for multi-class classification. LDA best discriminates training instances by their classes. The major difference betwee LDA and PCA is that LDA finds a linear combination of input features that optimizes class separability while PCA attempts to find a set of uncorrelated components of maximum variance in a dataset. Another key difference betwee the two is that PCA is an unsupervised algorithm whereas LDA is a supervised algorithm where it takes class labels into account.

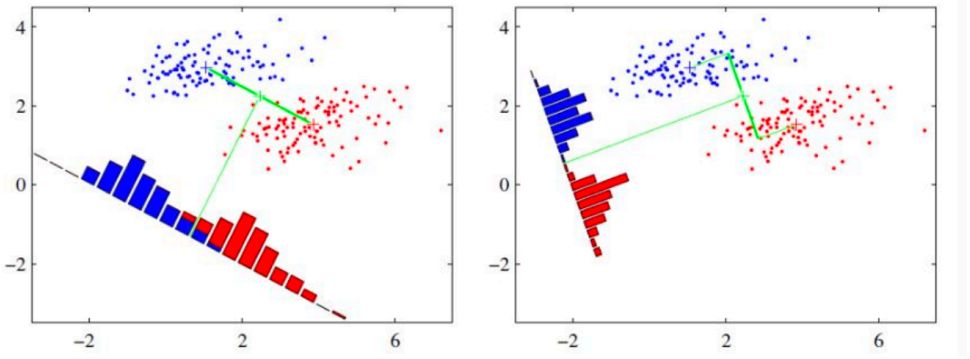


图 3: LDA

The task is to learn a linear function that projects X_i to $z = X_i^T \beta$, so that the projection maximizes the inter-class variance and minimizes the intra-class variance.

The objective function is

$$J(v) = \frac{v^T S_B v}{v^T S_W v}$$

We can use Lagrange multipliers for this constrained maximization and get the result

$$v \propto S_W^{-1}(\mu_1 - \mu_2)$$

For the code detail, we use *LinearDiscriminantAnalysis* in the package *sklearn.discriminant_analysis* as follows:

```
transformer = LinearDiscriminantAnalysis(solver='svd', n_components=n_comp)
transformer.fit(X_train, y_train)
X_train_proj = transformer.transform(X_train)
X_test_proj = transformer.transform(X_test)
acc = SVMmodel.runSVM(X_train_proj, X_test_proj, y_train, y_test, C, 'linear')
```

2.2.3 Auto-Encoder

Auto-Encoders is a type of artificial neural network that aims to copy their inputs to their outputs. They compress the input into a latent-space representation, and then reconstructs the output from this representation. An auto-encoder is composed of two parts

1. Encoder: compresses the input into a latent-space representation.
2. Decoder: reconstruct the input from the latent space representation.

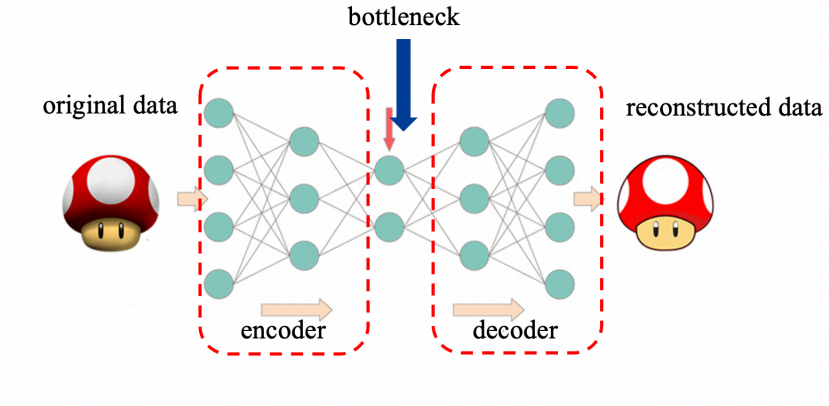


图 4: AutoEncoder

2.3 Feature Learning

2.3.1 SNE (Stochastic Neighborhood Embedding)

Stochastic Neighbor Embedding computes the probability that pairs of data points in the high-dimensional space are related and then chooses a low-dimensional embedding which produce a similar distribution.

First evaluate the similarity of data point x_i and data point x_j according to

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

Then evaluate the similarity of data point $\hat{x_i}$ and data point $\hat{x_j}$ according to

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

Since from x to $\hat{x}$ maintains the transition probability from i-th sample to j-th sample, so $q(j|i)$ should be close to $p(j|i)$, that is to minimize

$$L = \sum_i KL(P_i||Q_i) = \sum_i \sum_j p(j|i) log \frac{p(j|i)}{q(j|i)}$$

For the t-SNE, just replace Gaussian distribution in SNE with t-distribution.

$$p(j|i) = \frac{(1 + ||x_i - x_j||^2)^{-1}}{\sum_{k \neq i} (1 + ||x_i - x_k||^2)^{-1}}$$

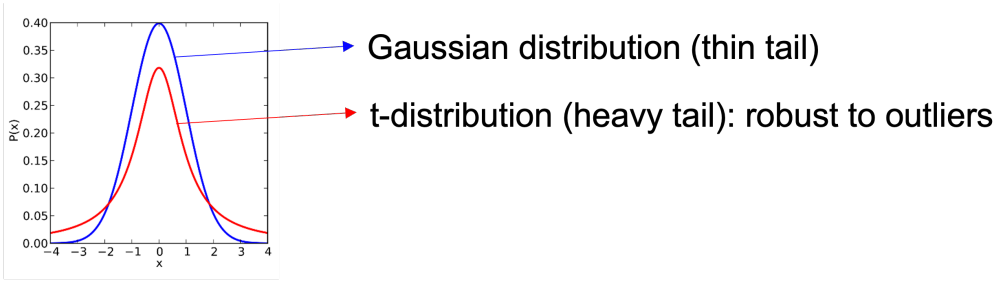


图 5: t-SNE

For the code detail, we use *TSNE* in the package *sklearn.manifold* as follows:

```
transformer = TSNE(n_components=n_comp, perplexity=number_perp, method=m, n_jobs=number_jobs)
transformer.fit(X_train)
X_train_proj = transformer.fit_transform(X_train)
X_test_proj = transformer.fit_transform(X_test)
acc = SVMmodel.runSVM(X_train_proj, X_test_proj, y_train, y_test, C, 'linear')
```

2.3.2 LLE (Local Linear Embedding)

Local Linear Embedding Recovers global non-linear structure from linear fits.Each local patch of the manifold can be written as a linear, weighted sum of its neighbours given enough data.

The objective function is

For the code detail, we use *LLE* in the package *sklearn.manifold* as follows: $min_{\mathbf{W}} ||\mathbf{x}_i - \sum_{j \in N(i)} w_{ij} \mathbf{x}_j||^2$ s.t. $\sum_j w_{ij} = 1$.

We can transfer $\mathbf{W}$ to $\mathbf{Y}$ to maintain local relationship: $min_{\mathbf{Y}} ||\mathbf{y}_i - \sum_{j \in N(i)} w_{ij} \mathbf{y}_j||^2$ s.t. $\mathbf{Y}^T \mathbf{Y} = \mathbf{I}$

```
transformer = LocallyLinearEmbedding(n_neighbors=n_neigh, n_components=n_comp, eigen_solver='dense',
                                                    n_jobs=number_jobs)
transformer.fit(X_train)
X_train_proj = transformer.fit_transform(X_train)
X_test_proj = transformer.fit_transform(X_test)
acc = SVMmodel.runSVM(X_train_proj, X_test_proj, y_train, y_test, C, 'linear')
```

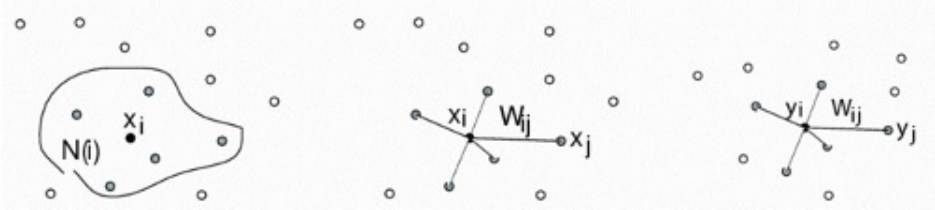


图 6: t-SNE

2.3.3 MDS (Multi Dimensional Scaling)

Multi-dimensional scaling (MDS): A technique used for analyzing similarity or dissimilarity of data as distances in a geometric spaces. Projects data to a lower dimension such that data points that are close to each other (in terms if Euclidean distance) in the higher dimension are close in the lower dimension as well.

2.3.4 ISOMap

ISOMap first constructs K-nearest neighbour graph G. For each pair of points in G, compute shortest path: geodesic distance. Then use classical MDS with geodesic distance matrix $\bar{\mathbf{D}}$.

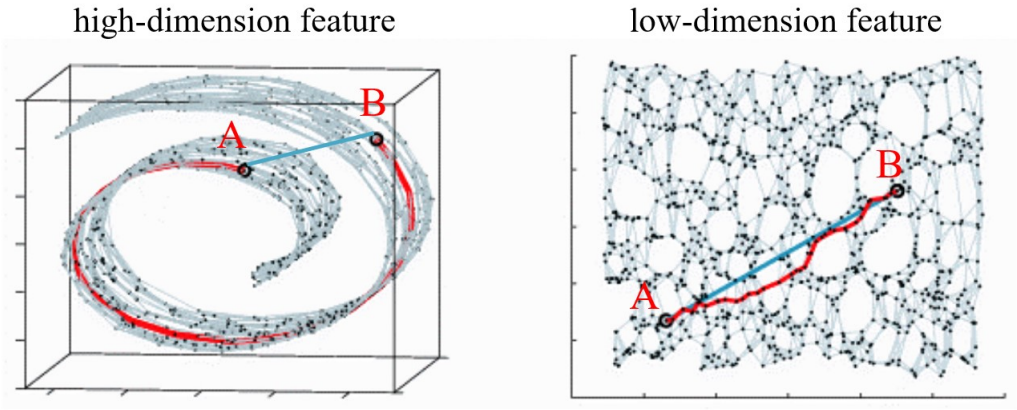


图 7: ISOMap

3 Experiment

The datastset we use is Attributes (AwA2) dataset. This dataset consists of 37322 images of 50 animal classes with pre-extracted deep learning features for each image. We split the images in each category into 60% for training and 40% for testing.

We use the linear SVM for the baseline. To determine the optimal C in SVM, we use K-fold cross-validation. The results show that when C is 0.002, the performance is best

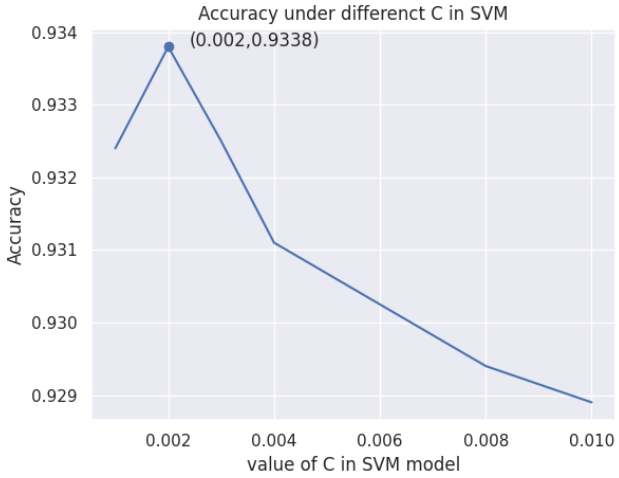


图 8: Accuracy under different C in SVM

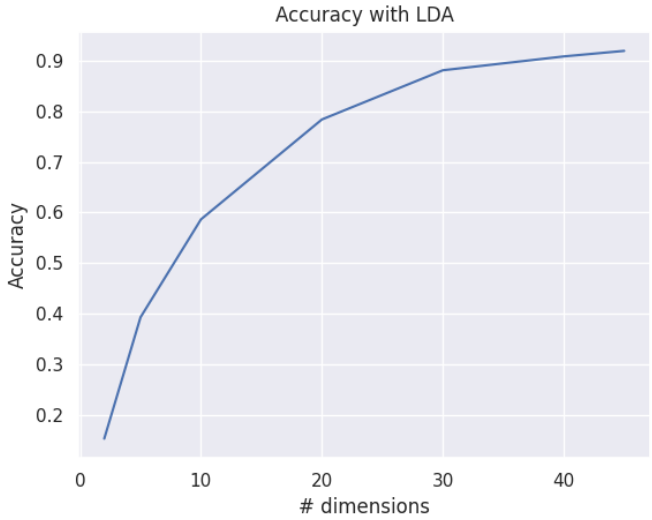
3.1 Feature Projection

3.1.1 PCA

# dimensions	Accuracy
2	0.142742
5	0.438341
10	0.705071
20	0.840110
50	0.898721
100	0.916672
200	0.926318
500	**0.928662**
750	0.926050
1000	0.923371

PCA Result

3.1.2 LDA



# dimensions	Accuracy
2	0.152790
5	,0.392793
10	0.586442
20	0.783977
30	0.881238
40	0.908634
45	0.919620

LDA Result

3.1.3 Auto-Encoder

3.2 Feature Learning

3.2.1 SNE

3.2.2 LLE

3.2.3 MDS