SDM274 Final Project

Ziyuan Li 12211225

This project comprehensively covers AI/ML knowledge covering PCA, SVM, MLP, Adaboost etc...

December 27, 2024

Contents

1. K-Means++ Algorithm

K-means is a *clustering* algorithm, which is one of the unsupervised learning algorithms. One of the biggest advantage of unsupervised learning is that they do not require explicit feedback whether outputs of systems are correct i.e. tagging.

- Note: The biggest trend of AI nowadays, LLMs, are trained primarily using self-supervised learning, which is a subset of unsupervised learning.

The primary goals of unsupervised learning are: reducing data dimensionality (like PCA), finding natural groupings or clusters (like K-means), modeling data density distributions, and uncovering hidden causal factors. They have utilities in compressing data, detecting outliers,facilitating other types of learning by providing preliminary insights or pre-processing and so on.

1.1. Principles of K means ++

- Basic K means

Training in K means essentially contains **three steps**:

1. Initialization:
   - cluster centers are randomly initialized
2. Assignment (E-step, E for Expectation):
   - each data point is assigned to the nearest cluster
2. Assignment (M step , M for Maximization):
   - The centroids are re-calculated by taking means of all points assigned to it

E step and M step loops until the algorithm converges.

- Convergence

The final goal is

$$\begin{aligned} \min_{\{m\},\{r\}} J(\{m\},\{r\}) = \min_{\{m\},\{r\}} \sum_{n=1}^{N}\sum_{k=1}^{K} r_k^{(n)} \parallel m_k - x^{(n)}\parallel^2 \\ s.t. \sum_k r_k^{(n)} = 1, \forall n \quad \text{where} \quad r_k^{(n)} \in \{0,1\}, \forall k,n \end{aligned} \tag{1}$$

That is, minimizing the total distance of all points to their nearest centroid.

When the assignments do not change in the assignment step, the algorithm converges (at least to a local minimum). However the objective is *non-convex*, that is, there is no guarantee to converge at global minimum.

As for the prediction part, a new data point is simply assigned to the nearest centroid.

- K means ++

Standard K-means randomly initializes centroids, which can lead to poor clustering. Because the non-convex property of J, poor initialization can get stuck in local optima.

A basic assumption to solve this problem is that, *points that are farther from existing centroids are more likely to be good candidates for new centroids*. This assumption helps spread out the initial centroids, preventing poor assignment.

The probability of selecting a point as a new centroid should be proportional to $D(x)^2$, where $D(x)$ is the distance to the nearest existing centroid.

An optimized initialization called *K means ++* is therefore designed to implement such idea.

1. First a center point m_1 is selected randomly from existing data points. Then, for each data point the distance to a nearest centroid is calculated.
2. The probability

$$p(x^{(n)}) = \frac{D(x^{(n)})^2}{\sum_j D(x^{(j)})^2} \tag{2}$$

is then calculated. The denominator is a sum that helps the overall summation sum to 1. The critical term here, the numerator, is the square of the distance mentioned before. New centroid is then chosen based on this probability, with the data point being farthest to the closese centroid most likely to be chosen.

3. Steps 1 and 2 loops until a specific number of centroids are chosen based on data points. Later optimizations are the same as standard K-means.

1.2. Implementation

1. Initialization

```python
def _init_centroids(self, X: npt.NDArray) -> npt.NDArray:
        """
        K-means++ 的初始化步骤

        Args:
            X: 输入数据，形状为 (n_samples, n_features)

        Returns:
            初始化的质心，形状为 (n_clusters, n_features)
        """
        n_samples, n_features = X.shape
        centroids = np.zeros((self.n_clusters, n_features))

        # 随机选择第一个质心
        centroids[0] = X[np.random.randint(n_samples)]

        for k in range(1, self.n_clusters):
            distances = []
```

```
            # 计算每个点到质心的距离
            for centroid in centroids[:k]:
                diff = X - centroid
                norms = np.linalg.norm(diff, axis=1)
                distances.append(norms)
            distances = np.array(distances).T
            min_distances = np.min(distances, axis=1) # 选取最小距离
            # 根据距离的平方作为概率选择新的质心
            prob = min_distances ** 2 / np.sum(min_distances ** 2)
            centroids[k] = X[np.random.choice(n_samples, p=prob)]

        return centroids
```

In this function I calculated the distance matrix of every point to every centroid, then selects the smallest distance. The new centroid is calculated using the probability in equation 2.

2. Expectation

```python
def _compute_distances(self, X: npt.NDArray, centroids: npt.NDArray) -> npt.NDArray:
        """
        计算每个样本到所有质心的距离

        Args:
            X: 输入数据
            centroids: 当前质心

        Returns:
            距离矩阵，形状为 (n_samples, n_clusters)
        """
        distances = np.zeros((X.shape[0], self.n_clusters))
        for k in range(self.n_clusters): # 对距离矩阵的每一列进行计算，一列存储所有数据点对这个 cluster 中心的距离。
            distances[:, k] = np.linalg.norm(X - centroids[k], axis=1)
            # X - centroids[k]用了广播机制，每一行是一个差值向量，之后直接用 norm 对每一行进行计算
        return distances

    def _assign_clusters(self, distances: npt.NDArray) -> npt.NDArray:
        """
        根据距离分配聚类标签

        Args:
            distances: 距离矩阵

        Returns:
            聚类标签
        """
        return np.argmin(distances, axis=1)
```

It calculates the distance matrix and updates the assignment. The labels are stored in a matrix (samples, cluster).

3. Maximization

```python
def _update_centroids(self, X: npt.NDArray, labels: npt.NDArray) -> npt.NDArray:
        """
        更新质心位置

        Args:
            X: 输入数据
            labels: 当前聚类标签

        Returns:
            更新后的质心
        """
        centroids = np.zeros((self.n_clusters, X.shape[1]))
        for k in range(self.n_clusters):
            mask = labels == k
            if np.any(mask):
                centroids[k] = X[mask].mean(axis=0)
        return centroids
```

It updates the centroids bases on means.

Note:

1. The centroids are updated based on data points, but after M step they (often) do not overlap with data points any more.
2. For convenience, we directly use the indexes as labels (starts from 0). We define a property for the correct converted labels to be visited from the outside

```python
@property
    def labels(self) -> np.ndarray:
        """返回从 1 开始的聚类标签"""
        if self.membership is None:
            return None
        return np.argmax(self.membership, axis=1) + 1
```

1.3. Results

```
质心: [[18.72180328 16.29737705  0.88508689  6.20893443  3.72267213  3.60359016
   6.06609836]
 [11.96441558 13.27480519  0.8522      5.22928571  2.87292208  4.75974026
   5.08851948]
 [14.64847222 14.46041667  0.87916667  5.56377778  3.27790278  2.64893333
```

```
    5.19231944]]

调整后的标签:
[1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 3 1 1 3]
参考标签：
[1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1]
准确率: 0.8952
```

The overall accuracy is 89.52%.

2. Soft K means

This is another optimization to the Standard K means algorithm.

2.1. Basic Principles of Soft K means

1. Difference in E step: data points are given probabilities to each clusters, instead of hard assignments. The responsibilities are calculated by:

$$r_k^{(n)} = \frac{\exp\left[-\beta d\left(m_k, x^{(n)}\right)\right]}{\sum_j \exp\left[-\beta d\left(m_j, x^{(n)}\right)\right]} \tag{3}$$

2. Difference in M step: For each centroid, the weighted mean of all data points (weighted by membership probability) is calculated to update new centroid.

$$m_k = \frac{\sum_n r_k^{(n)} x^{(n)}}{\sum_n r_k^{(n)}} \tag{4}$$

2.2. Implementation

1. E step

```python
def _update_membership(self, X: npt.NDArray, update_internal: bool = True) -> npt.NDArray:

        distances = self._compute_distances(X)
        exp_distances = np.exp(-self.beta * distances)
        membership = exp_distances / np.sum(exp_distances, axis=1, keepdims=True)

        if update_internal:
            self.membership = membership

        return membership
```

2. M step

```python
def _compute_centroids(self, X: npt.NDArray) -> npt.NDArray:
        """
        更新簇中心：m_k = Σ_n r_k^(n)x^(n) / Σ_n r_k^(n)

        Args:
            X: 输入数据，形状为 (n_samples, n_features)

        Returns:
            更新后的簇中心，形状为 (n_clusters, n_features)
        """
        # 初始化质心数组
        self.centroids = np.zeros((self.n_clusters, X.shape[1]))

```

```
        for k in range(self.n_clusters):
            # 分子：所有点的加权和。归属度和 X 做点乘
            # 例如：当一个点对 k 的归属度为 0.3，会用 0.3 和这个点的坐标相乘。这里计算所有点的这个乘积
            numerator = np.sum(self.membership[:, k][:, np.newaxis] * X, axis=0) # 这里让 membership 的第 k 行和 X 做点乘，然后求和。

            # 分母：权重和
            denominator = np.sum(self.membership[:, k])
            self.centroids[k] = numerator / denominator

        return self.centroids # 这样可以
```

3. At the output, the membership probs are converted to hard labels.

```python
def predict(self, X: npt.NDArray) -> npt.NDArray:
        """
        预测新数据的聚类标签（硬分配）

        Args:
            X: 输入数据

        Returns:
            聚类标签（从 1 开始）
        """

        membership = self._update_membership(X, update_internal=False)
        return np.argmax(membership, axis=1) + 1
```

2.3. Results

```
质心: [[14.63500455 14.45317958  0.87927954  5.56228177  3.27610281  2.65818239
   5.18934573]
 [11.96116899 13.27337974  0.85215326  5.22816181  2.87295787  4.74007699
   5.08574684]
 [18.69091238 16.28472842  0.88495127  6.20403039  3.7192383   3.60449166
   6.06196339]]
调整后的标签: [1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1]
真实标签: [1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1]
准确率: 0.8952380952380953
```

The overall accuracy is high, although there is little improvement from K means ++ in our case.

3. PCA

PCA is a commonly used, linear algebra based method to reduce the dimensionality of data.

It represents original data points by *components*. The number of components can be manually designated, and is often far less than the number of *features*.

3.1. Basic Principles

PCA works by identifying the directions (principal components) in the data where there is the most variation. It is like finding the most important "viewing angles" of the data. Each subsequent principal component is perpendicular to the previous ones and captures the next most important direction of variation.

Mathematically, these principal components are the eigenvectors of the data's covariance matrix, ordered by their corresponding eigenvalues (which represent the amount of variance explained by each component).

When we want to reduce dimensions, simply keep the top N components that capture most of the variance and project the data onto these components.

- Steps

1. Normalize the data points, and calculate the covariance matrix

$$C = \frac{1}{N}\sum_{n=1}^{N}(x^{(n)} - \bar{x})(x^{(n)} - \bar{x})^T \tag{5}$$

2. Select top M eigenvectors

3. Project each input vector into subspace

$$z_j = u_j^T x; z = U_{1:M}^T x \tag{6}$$

3.2. Implementation

1. Calculating eigen vecs and eigenvalues

```python
def fit(self, X: npt.NDArray) -> 'PCA':
        """
        训练 PCA 模型

        Args:
            X: 形状为 (n_samples, n_features) 的训练数据

        Returns:
            self: 训练后的 PCA 实例
        """
        if self.random_state is not None:
            np.random.seed(self.random_state)

        self.n_samples_, self.n_features_ = X.shape
```

```
15
16          # 计算并减去均值
17          self.mean_ = np.mean(X, axis=0)
18          X_centered = X - self.mean_
19
20          # 计算协方差矩阵
21          cov_matrix = np.cov(X_centered.T)
22
23          # 计算特征值和特征向量
24          eigenvals, eigenvecs = np.linalg.eig(cov_matrix)
25
26          # 按特征值大小排序
27          idx = eigenvals.argsort()[::-1]
28          eigenvals = eigenvals[idx]
29          eigenvecs = eigenvecs[:, idx]
30
31          # 要保留的主成分数量
32          if self.n_components is None:
33              self.n_components = self.n_features_
34
35          self.components_ = eigenvecs[:, :self.n_components] # 取前 n_components 列。这里会发生信息损失
36          self.explained_variance_ = eigenvals[:self.n_components]
37          total_var = eigenvals.sum()
38          self.explained_variance_ratio_ = eigenvals[:self.n_components] / total_var
39
40          return self
```

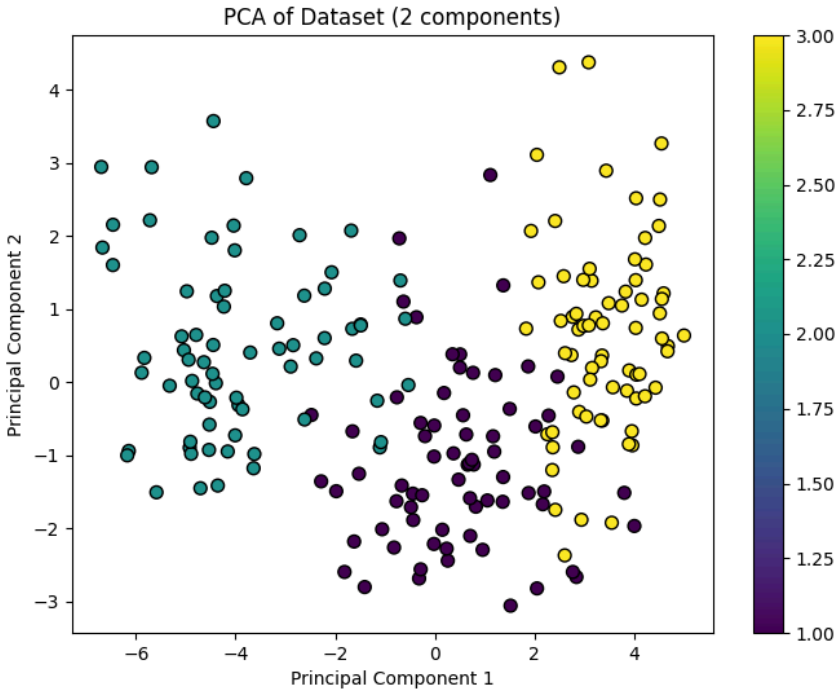
2. Mapping

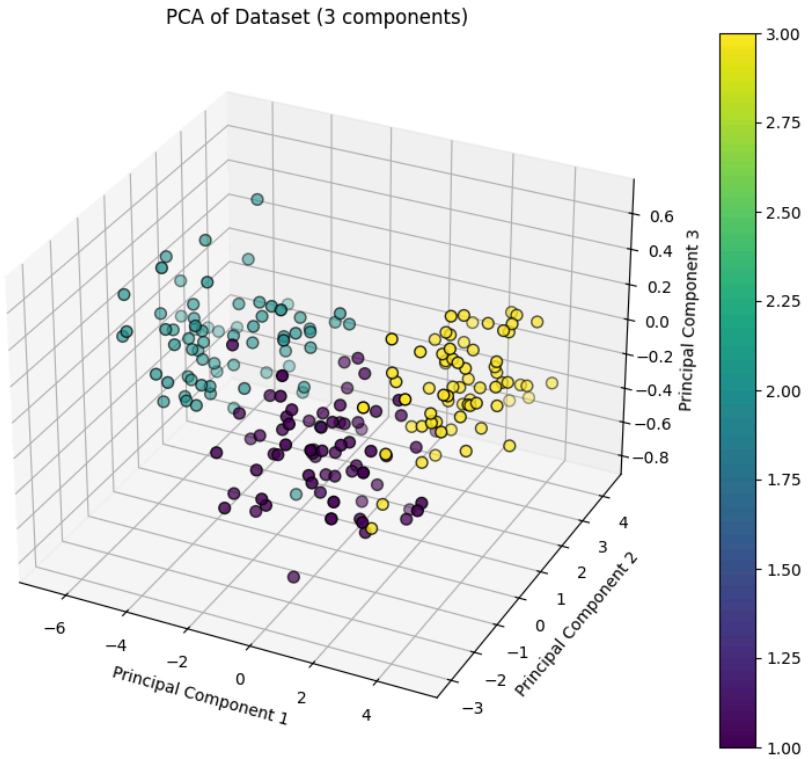
```python
1   def transform(self, X: npt.NDArray) -> npt.NDArray:
2           """
3           使用训练好的 PCA 模型将数据转换到新的特征空间
4
5           Args:
6               X: 形状为 (n_samples, n_features) 的输入数据
7
8           Returns:
9               形状为 (n_samples, n_components) 的转换后的数据
10          """
11          # 检查特征数量是否匹配
12          if X.shape[1] != self.n_features_:
13              raise ValueError(f"期望的特征数量为 {self.n_features_}，但输入数据的特征数量为 {X.shape[1]}")
14
15          # 中心化数据
16          X_centered = X - self.mean_
17
```

```
        # 投影到主成分空间
        X_transformed = np.dot(X_centered, self.components_)

        return X_transformed
```

3.3. Results

```
Reconstruction error: 0.012919023579588521
Reconstruction error (3 components): 0.002450535895729073
```




PCA of Dataset (3 components)
Principal Component 1
Principal Component 2
Principal Component 3

4. Nonlinear Autoencoder

Autoencoder is a special type of neural network that has a bottleneck layer in the middle of its structure. It is often used for two purposes: dimensionality reducements and anomaly detection.

An autoencoder works by trying to reconstruct its own input data through a constrained architecture. It consists of two main parts: an encoder that compresses the input data into a lower-dimensional representation (the bottleneck layer), and a decoder that attempts to reconstruct the original data from this compressed representation.

The key insight is that by forcing the network to reconstruct the input through a bottleneck, it must learn the most important features of the data to achieve good reconstruction.

Unlike PCA which finds linear relationships, autoencoders can capture nonlinear relationships in the data due to their use of activation functions and multiple layers.

4.1. Implementation

We build the audoencoder based on MLP code we written before.

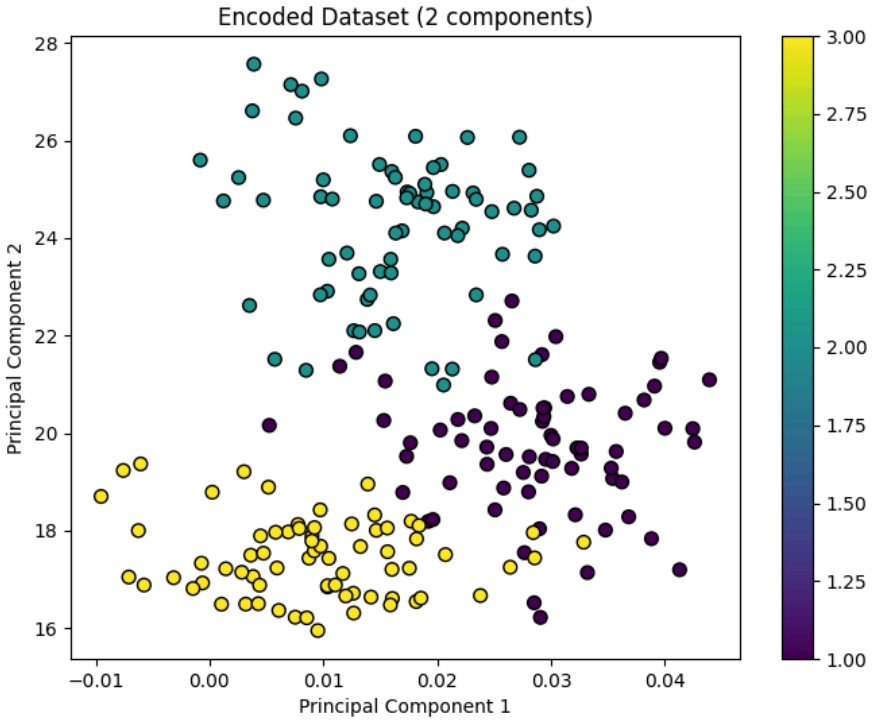
Key changes:

```python
def get_loss(self, x_pred, x_true):
        x_pred = x_pred.reshape(-1, self.output_shape)
        x_true = x_true.reshape(-1, self.output_shape)

        # MSE loss for reconstruction
        return np.mean((x_pred - x_true) ** 2)


    def get_loss_grad(self, x_pred, x_true):
        # The loss here is similar to MSE, and the grad is alike.
        x_pred = x_pred.reshape(-1, self.output_shape)
        x_true = x_true.reshape(-1, self.output_shape)

        return (x_pred - x_true) / self.output_shape


nonlinear_layers = [
        Linear(input_size=7, output_size=2),
        ReLU(),
        Linear(2,2),
        Linear(input_size=2, output_size=7)
    ]
```

4.2. Results

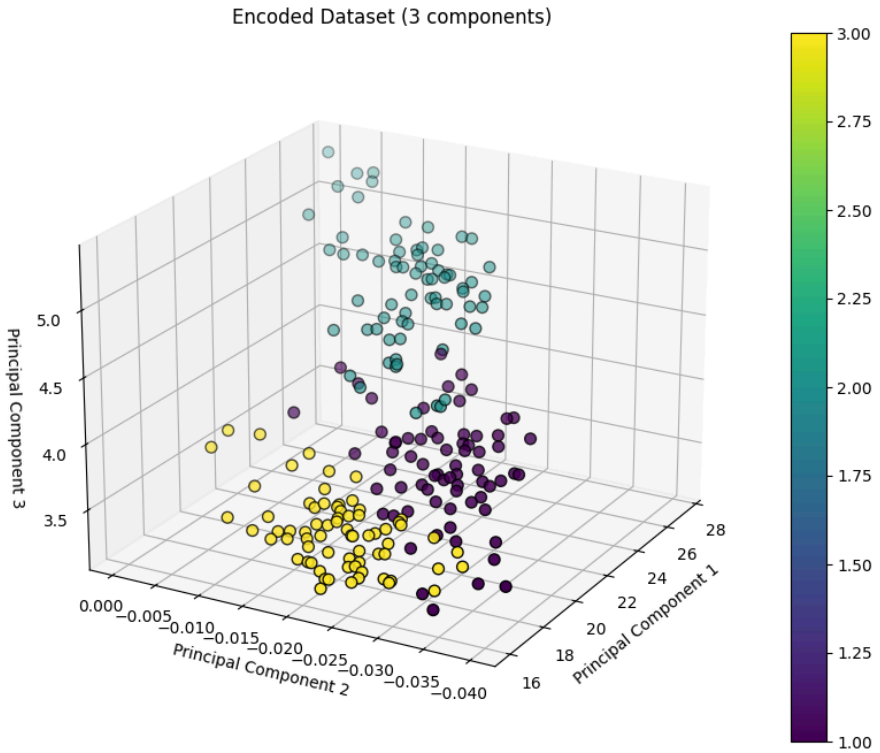
1. 2 dimensions

```
[[15.13676411 14.6227186   0.8656738   5.64196229  3.28308037  3.63448522
   5.42482115]]
[[15.26  14.84   0.871  5.763  3.312  2.221  5.22 ]]
Reconstruction Error: 0.535941549349706
```



2. 3 dimensions

```
[[15.15961006 14.65627171  0.86811233  5.6555102   3.28999074  3.64412463
   5.4375927 ]
 [14.70293282 14.29592383  0.85017506  5.52054024  3.20529841  3.58524176
   5.30643438]
 [14.27744504 13.96019647  0.83346448  5.39479536  3.12639238  3.53038517
   5.18424113]
 [13.92279769 13.68035916  0.81953511  5.28998181  3.06062234  3.48465893
   5.0823877 ]
 [15.68352055 15.06966054  0.88868866  5.8103434   3.3871495   3.71167142
   5.58805278]]
[[15.26   14.84    0.871   5.763   3.312   2.221   5.22  ]
 [14.88   14.57    0.8811  5.554   3.333   1.018   4.956 ]
 [14.29   14.09    0.905   5.291   3.337   2.699   4.825 ]
 [13.84   13.94    0.8955  5.324   3.379   2.259   4.805 ]
 [16.14   14.99    0.9034  5.658   3.562   1.355   5.175 ]]
Reconstruction Error with 3D middle layer: 0.48490035703294065
```



Slightly better than 2D.

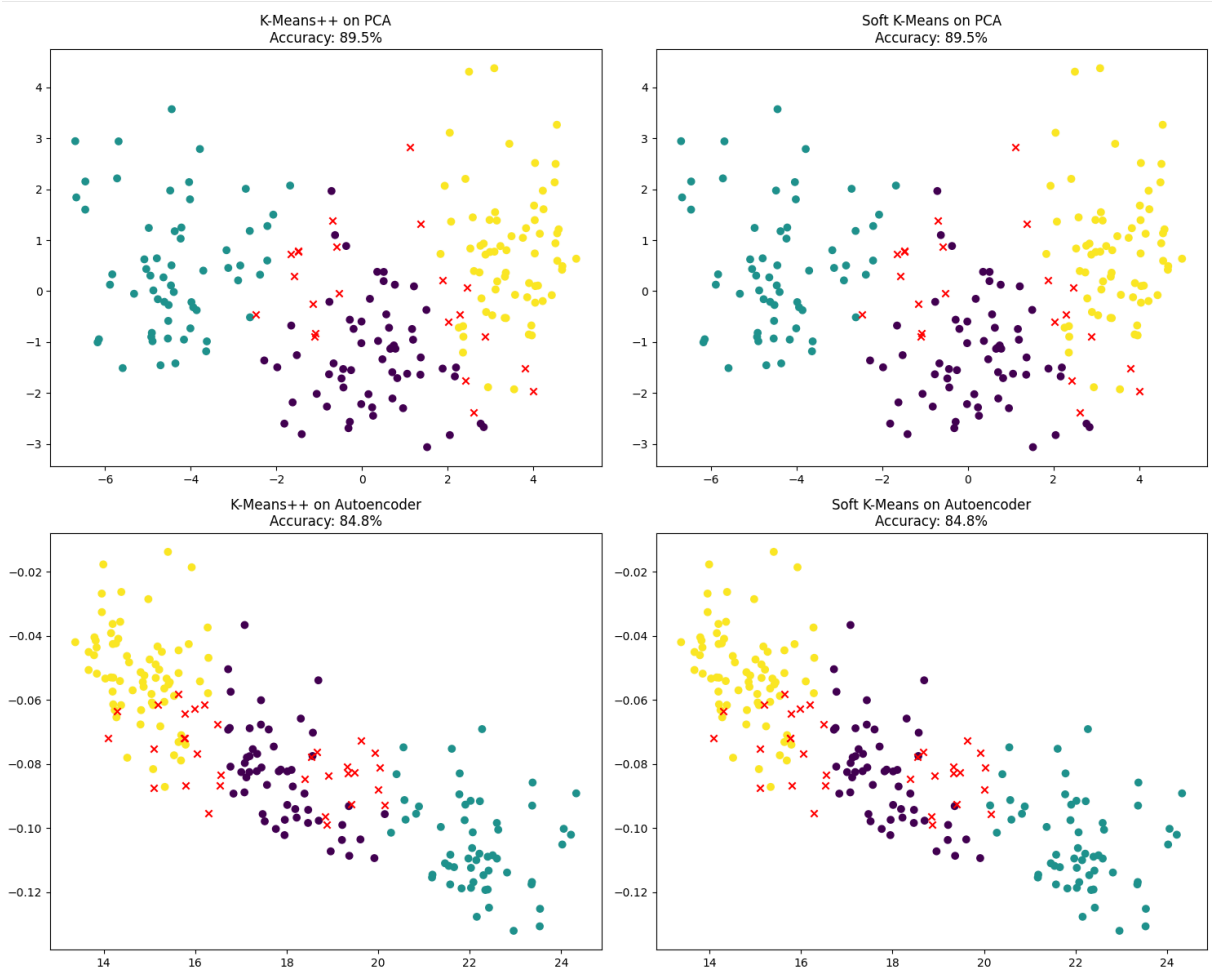
5. Clustering with Reduced Dimensions

This section focuses on the concrete results of clustring and dimensionality reducing.

Corresponding script is stored in `experiments/clustering_reduced.py`

For clear visualization, we choose the dimensions to be **2**.

5.1. Results



1. The difference between two clustering methods are little.
2. The difference between PCA and autoencoder is significant. In this case, autoencoder is worse than PCA in accuracy.

5.2. Conclusion

In simple task like this, autoencoder is outperformed by PCA in terms of both computing time and final accuracy in clustering and reconstruction.

> Neural networks are not solution to *ALL*.

This is also due to the optimum of autoencoder is hard to find, while the optimum of PCA is mathematically guaranteed. This exposes another **drawback of neural networks**: they are **hard to train** and **often do NOT guarantee to give best results**.

6. MLP for Multi-Class Classification

Here we reused much of the code in *Assignment 08 Multiclass MLP*. Thanks to the few-weeks-ago me.

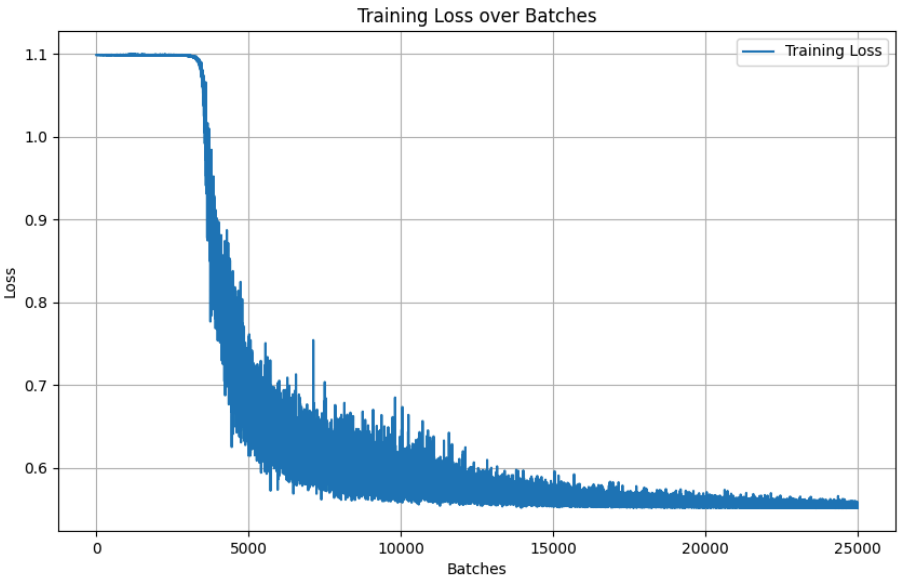
We created a new class that inherits from the MLP class.

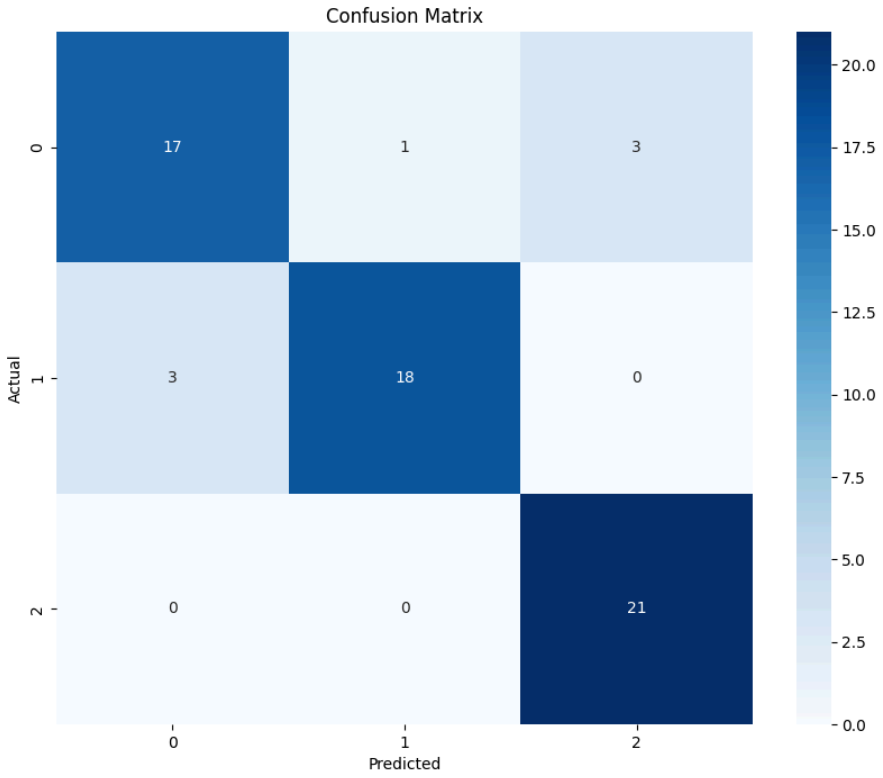
The key changes here are the predict method and the softmax function.

- The model will output 3 values. They should be converted into probabilities (that add up to 1) by the softmax function.
- correspondingly, the predict function chooses the index with the greatest possibility and choose it as the final prediction.
- The gradient also expands to multi-dimensional. It subtracts the logits with the true values to feedback the results.

6.1. Implementation

```python
layers = [
        Linear(input_size=7, output_size=16),
        ReLU(),
        Linear(input_size=16, output_size=16),
        ReLU(),
        Linear(input_size=16, output_size=3),
        Sigmoid()
    ]

    epochs = 5000
    lr = 5e-2

    mlp = MLPMultiClass(layers, epochs=epochs, lr=lr,
    input_shape=X_train.shape[1], output_shape=3)
    mlp.train_MBGD(X_train_std, y_train)
```




Confusion Matrix
17
1
3
3
18
0
0
0
21
Actual
Predicted
0
1
2
20.0
17.5
15.0
12.5
10.0
7.5
5.0
2.5
0.0

7. SVM and SVM with Gaussian Kernel

Support Vector Machine is an optimization-based algorithm for separating different data points.

It is a **supervised learning** algorithm.

7.1. Principles

In logistic regression we drew a line to separate two classes. Now, we are thinking about how to optimize the location of margins. Intuitively, we want to make the "gaps" as wide as possible. This is the intuition behind SVM.

The 'vector' in support vector machine stands for a data point. It acts a "support" for the margin to maximize the width of it.

SVM uses two classes, −1 and +1. If any data point goes beyond the boundary where $w^T x+b$ =1, it is classified as +1, and the same also applies for −1 values. Samples lie between are undefined.

The gap can be computed by

$$\lambda = \frac{2}{w^T w} \tag{7}$$

Therefore we want to minimize the norm of w, with the constraint that the true label aligns with the $w^T x^{(i)} + b$ term.

$$\begin{aligned} &\min_{w,b} \frac{1}{2} \parallel w \parallel^2 \\ &\text{s.t.} \quad \big(w^T x^{(i)} + b\big) t^{(i)} \geq 1, \quad \forall i = 1, ..., N \end{aligned} \tag{8}$$

- Lagrange Multipliers

Direct optimization is difficult because of the constraints. Lagrange multipliers help us convert this constrained problem into an unconstrained one.

We can convert formula (8) into an unconstrained problem using lagrange multipliers and use KKT condition to solve it.

$$L(w, b, \alpha) = \frac{1}{2} \parallel w \parallel^2 - \sum_{i=1}^{N} \alpha_i [(w^T x^{(i)} + b) t^{(i)} - 1] \tag{9}$$

KKT condition tells us to take derivatives to w and b, and make them zero. Also there is a constraint that

$$\alpha_i^* [1 - (w^{*T} x^{(i)} + b^*) t^{(i)}] = 0 \tag{10}$$

Finally, after taking derivative of J to w we have

$$w = \sum_{i=1}^{N} \alpha_i t^{(i)} x^{(i)} \tag{11}$$

7.2. Improvements

- Kernels

Standard SVM only deals with linearly separable binary classification.

We can use a kernel to make it separate non linearily-separable data points.

The idea behind is that a product of original features is produced using a kernel function.

Originally, a new point is calculated by

$$y = \text{sign}\left[b + x \cdot \left(\sum_{i \in S} \alpha_i t^{(i)} x^{(i)}\right)\right] \tag{12}$$

We use a kernel function to do the dot product implicitly.

$$y = \text{sign}\left[b + \sum_{i \in S} \alpha_i t^{(i)} K(x^{(i)}, x)\right] \tag{13}$$

The calculation of α_i is the same with standard SVM.

$$\frac{\partial L}{\partial \alpha_i} = 1 - \sum_{j=1}^{n} \alpha_j y_i y_j K(x_i, x_j) = 0 \tag{14}$$

In practice, SMO algorithm is usually used to calculate the α values. I used the algorithm provided on Blackboard.

- Multi class

In my code I used a OvO classifier (training on 1 vs 2, 2 vs 3, and 3 vs 1) as mentioned in lecture 8. This is not the main point of this section.

7.3. Implementation

1. Optimizing for α (that is the central idea of the whole algorithm)

```python
def _train_binary_svm(self, X: npt.NDArray, y: npt.NDArray) -> Tuple:
        """
        训练二分类 SVM（使用 SMO 算法）

        Args:
            X: 训练数据
            y: 训练标签 (+1/-1)

        Returns:
            (alpha, b, support_vectors, sv_y)
        """
        n_samples = X.shape[0]

        # 计算核矩阵
        K = self._compute_kernel(X, X)
```

```
16
17          # 初始化 SMO 求解器
18          p = -np.ones(n_samples)
19          solver = Solver(Q=K,
20                          p=p,
21                          y=y,
22                          C=self.C,
23                          tol=self.tol)
24
25          # SMO 迭代优化
26          for _ in range(self.max_iter):
27              i, j = solver.working_set_select()
28              if i < 0:  # 收敛条件
29                  break
30              solver.update(i, j)
31
32          # 获取支持向量
33          support_mask = solver.alpha > self.tol
34          return (solver.alpha[support_mask],
35                  solver.calculate_rho(),
36                  X[support_mask],
37                  y[support_mask])
```

2. Kernel

```python
def _compute_kernel(self, X1: npt.NDArray, X2: npt.NDArray) ->
npt.NDArray:
        """计算核矩阵"""
        if self.kernel == 'linear':
            return np.dot(X1, X2.T)
        elif self.kernel == 'rbf':
            # 计算 RBF 核 K(x,y) = exp(-gamma||x-y||^2)
            squared_norm = np.sum(X1**2, axis=1).reshape(-1, 1) + \
                           np.sum(X2**2, axis=1) - \
                           2 * np.dot(X1, X2.T)
            return np.exp(-self.gamma * squared_norm)
        else:
            raise ValueError(f"Unsupported kernel type: {self.kernel}")
```

3. Prediction

```python
def _predict_binary(self, x: npt.NDArray, classifier_params: Tuple) -
> int:
        """
        使用二分类器进行预测

        Args:
            x: 输入样本
            classifier_params: (alpha, b, support_vectors, sv_y)
```

```
8
9           Returns:
10              预测类别 (+1/-1)
11          """
12          alpha, b, support_vectors, sv_y = classifier_params
13
14          # 计算决策函数
15          K = self._compute_kernel(support_vectors, x.reshape(1, -1))
16          decision = np.sum(alpha * sv_y * K.reshape(-1)) - b
17          return 1 if decision >= 0 else -1
```

7.4. Results

```
1   Training Linear SVM...
2   Linear SVM Training Accuracy: 0.8333
3   Linear SVM Test Accuracy: 0.8571
4
5   Linear SVM Classification Report:
6                 precision    recall  f1-score   support
7
8               1       1.00      0.45      0.62        11
9               2       0.88      1.00      0.93        14
10              3       0.81      1.00      0.89        17
11
12       accuracy                           0.86        42
13      macro avg       0.89      0.82      0.82        42
14  weighted avg       0.88      0.86      0.84        42
15
16  Training RBF SVM...
17  RBF SVM Training Accuracy: 0.9167
18  RBF SVM Test Accuracy: 0.8571
19
20  RBF SVM Classification Report:
21                precision    recall  f1-score   support
22
23              1       0.73      0.73      0.73        11
24              2       0.93      1.00      0.97        14
25              3       0.88      0.82      0.85        17
26
27       accuracy                           0.86        42
28      macro avg       0.85      0.85      0.85        42
29  weighted avg       0.86      0.86      0.86        42
30
31
32  Single Sample Prediction Example:
33  True label: 1
34  Predicted label: 1
```

8. AdaBoost Algorithm

We can use an ensemble of Experts to make decision, not only use one decision method at at time.

The intuition behind is that *if every inspector is better than guessing, than the ensemble will be powerful.*

8.1. Adaboost Principles

It stands for Adaptive Boosting.

1. Adaptive: focuses on hard-to-classify samples by increasing their weights
2. Boosting: Combines many weak classifiers into a strong one

That is, it will assign weights both to inspectors *and* samples.

- For inspectors, it is called *Importance* (α)
  - primarily used for predictions
- For samples, it is called *weights*
  - harder to classify samples will have higher weights
  - used for training

For our case, we will use decision stumps to stress on the *simplicity* of the inspectors.

- decision stumps are naive classifiers that looks at only one feature and gives prediction based on one scalar boundary.
- we need to add the support for weights for our adaboost algorithm.

8.2. Implementation

1. Stumps

It calculates the information gain of features and then selects the feature with greatest information gain.

The entropy is weighted.

```python
def _weighted_entropy(self, y, sample_weight):
    # 不是简单计数每个类别有多少样本
    # 而是计算每个类别的样本权重之和
    weighted_counts = np.zeros(len(classes))
    for i, c in enumerate(classes):
        weighted_counts[i] = np.sum(sample_weight[y == c])

    # 计算概率时用权重和代替样本数
    probs = weighted_counts / np.sum(sample_weight)
```

2. Adaboost

```python
 def fit(self, X: npt.NDArray, y: npt.NDArray) -> 'AdaBoost':
        """
        训练 AdaBoost 分类器
```

```
4
5           Args:
6               X: 训练数据
7               y: 训练标签 (-1 或 1)
8
9           Returns:
10              self
11          """
12          n_samples = X.shape[0]
13          # 初始化样本权重
14          sample_weight = np.ones(n_samples) / n_samples
15
16          for i in range(self.n_estimators):
17              # 训练决策树桩
18              estimator = DecisionStump()
19              estimator.fit(X, y, sample_weight)
20
21              # 获取预测结果
22              predictions = estimator.predict(X)
23
24              # 计算加权错误率
25              incorrect = predictions != y
26              error = np.sum(sample_weight * incorrect) / np.sum(sample_weight)
27
28              # 如果错误率为 0 或 1，停止训练
29              if error == 0 or error >= 1.0:
30                  break
31
32              # 计算这个分类器的权重
33              estimator_weight = self.learning_rate * np.log((1 - error) /
                error)
34
35              # 保存分类器和权重
36              self.estimators.append(estimator)
37              self.estimator_weights[i] = estimator_weight
38
39              # 更新样本权重
40              sample_weight *= np.exp(estimator_weight * incorrect)
41              sample_weight /= np.sum(sample_weight)  # 归一化
42
43          return self
44
45      def predict(self, X: npt.NDArray) -> npt.NDArray:
46          """
47          预测新样本的标签
48
49          Args:
50              X: 测试数据
```

```
51
52          Returns:
53              predictions: 预测标签 (-1 或 1)
54          """
55          n_samples = X.shape[0]
56          predictions = np.zeros(n_samples)
57
58          # 累积所有分类器的加权预测
59          for estimator, weight in zip(self.estimators, self.estimator_weights):
60              predictions += weight * estimator.predict(X)
61
62          # 返回符号
63          return np.sign(predictions)
```

8.3. Results

```python
# 分割数据
    X_train, X_test, y_train, y_test = train_test_split(
        X, y, test_size=0.2, random_state=42
    )

    # 训练模型
    model = AdaBoost(n_estimators=50)
    model.fit(X_train, y_train)

    eval = Evaluation()

    # 预测并评估
    y_pred = model.predict(X_test)
    # accuracy = accuracy_score(y_test, y_pred)
    accuracy = eval.calculate_accuracy(y_test, y_pred)
    print(f"测试集准确率: {accuracy:.4f}")
```

```
测试集准确率: 0.7500
```

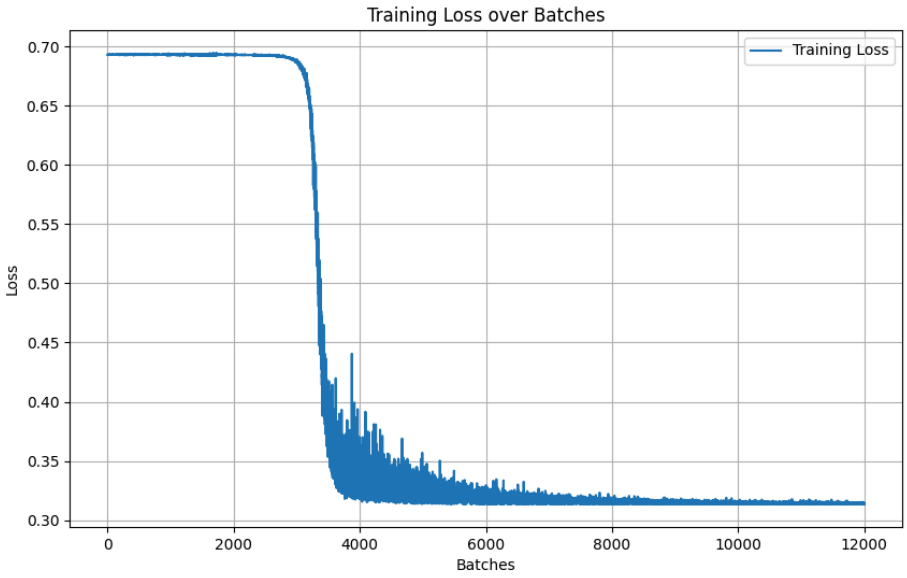
9. Binary Classification

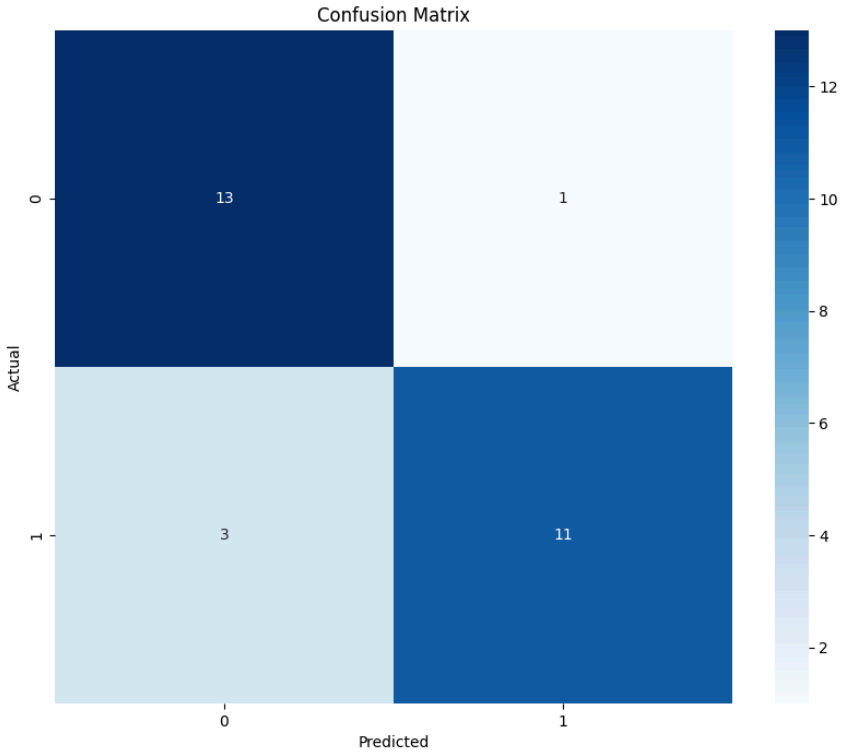
We will focus on the implementations in this section.

1. MLP

```python
def MLP_analysis():

    layers = [
        Linear(input_size=7, output_size=16),
        ReLU(),
        Linear(input_size=16, output_size=16),
        ReLU(),
        Linear(input_size=16, output_size=2),
        Sigmoid()
    ]

    epochs = 3000
    lr = 5e-2

    mlp = MLPMultiClass(layers, epochs=epochs, lr=lr,
    input_shape=X_train.shape[1], output_shape=2)
    mlp.train_MBGD(X_train_std, y_train_onehot)

    loss_history = mlp.loss
    length = len(loss_history)

    # Plot the loss history
    plt.figure(figsize=(10, 6))
    plt.plot(range(length), loss_history, label='Training Loss')
    plt.xlabel('Batches')
    plt.ylabel('Loss')
    plt.title('Training Loss over Batches')
    plt.legend()
    plt.grid(True)
    plt.show()

    y_pred = mlp.predict(X_test_std)

    y_test_numerical = np.argmax(y_test_onehot, axis=1)

    from sklearn.metrics import confusion_matrix
    import seaborn as sns

    cf_matrix = confusion_matrix(y_test_numerical, y_pred)

    plt.figure(figsize=(10, 8))
    sns.heatmap(cf_matrix, annot=True, fmt='d', cmap='Blues')
    plt.xlabel('Predicted')
    plt.ylabel('Actual')
```

```
44      plt.title('Confusion Matrix')
45      plt.show()
```






```
1  测试集准确率: 0.8571428571428571
2                precision    recall  f1-score   support
3
4             0       0.81      0.93      0.87        14
5             1       0.92      0.79      0.85        14
6
7      accuracy                           0.86        28
```

```
    macro avg       0.86      0.86      0.86        28
weighted avg       0.86      0.86      0.86        28
```

2. SVM

```python
def SVM_analysis():
    classifiers = {
        'Linear SVM': OVOSVM(C=1.0, kernel='linear'),
        'RBF SVM': OVOSVM(C=1.0, kernel='rbf', gamma=0.1)
    }

    eval = Evaluation()

    for name, clf in classifiers.items():
        print(f"\nTraining {name}...")

        # 训练模型
        clf.fit(X_train, y_train)

        # 在训练集上评估
        train_pred = clf.predict(X_train)
        # train_acc = accuracy_score(y_train, train_pred)
        train_acc = eval.calculate_accuracy(y_train, train_pred)
        print(f"{name} Training Accuracy: {train_acc:.4f}")

        # 在测试集上评估
        test_pred = clf.predict(X_test)
        # test_acc = accuracy_score(y_test, test_pred)
        test_acc = eval.calculate_accuracy(y_test, test_pred)
        print(f"{name} Test Accuracy: {test_acc:.4f}")

        # 输出详细的分类报告
        print(f"\n{name} Classification Report:")
        print(classification_report(y_test, test_pred))
```

```
Training RBF SVM...
RBF SVM Training Accuracy: 0.9554
RBF SVM Test Accuracy: 0.8929

RBF SVM Classification Report:
              precision    recall  f1-score   support

           1       1.00      0.79      0.88        14
           3       0.82      1.00      0.90        14

    accuracy                           0.89        28
   macro avg       0.91      0.89      0.89        28
weighted avg       0.91      0.89      0.89        28
```

3. Adaboost

```python
def Adaboost_analysis():
    ada = AdaBoost(n_estimators=50)
    ada.fit(X_train, y_train_ada)
    y_pred_ada = ada.predict(X_test)
    accuracy = eval.calculate_accuracy(y_test_ada, y_pred_ada)
    print(f"测试集准确率: {accuracy:.4f}")
```

```
测试集准确率: 0.6786
```

9.1. Summary

On this dataset, when doing binary classification, the accuracy is ranked by

SVM / SVM w. Gaussian > MLP > Adaboost Decision Stumps

Training conveniency:

SVM / SVM w. Gaussian ≈ Adaboost Decision Stumps > MLP

10. Summary

- K means ++ and Soft K means both shows good accuracy (~89%) on this dataset.
- PCA and nonlin autoencoder both reduces dimensions.
  - PCA is faster and more mathematically solid
  - On this dataset, autoencoder is outperformed, considering final precision and training convenience.
- MLP and autoencoders - neural models: they are intrinsically harder to train and to explain. On simple dataset like this, they are often beaten by data-driven models, both in accuracy and training costs. They might perform better in larger tasks like classifying numbers like in assignment 8.
- Adaboost can achieve much better than guessing results (~75%) even if we are using simplest inspectors. However it still lags behind SVM and MLP. Maybe it will be better if we use more complex inspectors.

10.1. Advantages and Disadvantages

K-means++ & Soft K-means: Advantages:
- High accuracy (89%)
- Simple and interpretable
- Fast computation
- No labels needed

Disadvantages:
- Only spherical clusters
- Sensitive to initialization

PCA: Advantages:
- Fast and deterministic
- Mathematically solid
- Easy to interpret

Disadvantages:
- Only linear relationships
- Hard to choose components
- May lose important info

Autoencoder: Advantages:
- Can capture nonlinear patterns
- Flexible representation

Disadvantages:
- Complex to train
- Outperformed by PCA here
- Computationally expensive

MLP: Advantages:
- Handles complex patterns

- Flexible architecture

Disadvantages:
- Overkill for simple data
- Hard to interpret
- **Complex training process**

AdaBoost: Advantages:
- Improves weak classifiers
- Simple implementation

Disadvantages:
- Lower accuracy (75%)

SVM: Advantages:
- Good in high dimensions
- Strong generalization
- Versatile kernels

Disadvantages:
- Kernel selection crucial

10.2. General Observations

- Simpler models (K-means++, PCA) worked best on this dataset
- Neural models were unnecessarily complex
- Consider data complexity when choosing between linear/nonlinear methods