姓名：曾闻天　学号：231880147　2026 年 1 月 11 日

一. (30 points) 降维

1. 基于 numpy 和 fetch_lfw_people 数据集实现主成分分析（PCA）算法，不可以调用 sklearn 库，完成下面代码并且可视化前 5 个主成分所对应的特征脸 (15 points)

```python
from sklearn.datasets import fetch_lfw_people
import matplotlib.pyplot as plt
# 1. 加载 LFW 数据集
lfw_people = fetch_lfw_people(min_faces_per_person=100, resize=0.4)
# 2. 获取数据
X = lfw_people.data

# Todo1: 写出PCA函数
def PCA1(X, n_components=5):

    return eigenfaces

eigenfaces = PCA(X)

# Todo2: 可视化5个主成分对应的特征脸
```

```python
from sklearn.datasets import fetch_lfw_people
import matplotlib.pyplot as plt
import numpy as np
import matplotlib

# 设置中文字体
plt.rcParams['font.sans-serif'] = ['SimHei', 'Microsoft YaHei', 'DejaVu Sans']
plt.rcParams['axes.unicode_minus'] = False  # 解决负号显示问题

# 1. 加载 LFW 数据集
lfw_people = fetch_lfw_people(min_faces_per_person=100, resize=0.4)

# 2. 获取数据
X = lfw_people.data

# Todo1: 写出PCA函数
def PCA(X, n_components=5):
    """
    主成分分析 (PCA) 实现

    参数:
    X: 输入数据, 形状为 (n_samples, n_features)
    n_components: 要提取的主成分数量
```

```
    返回:
    eigenfaces: 特征脸, 形状为 (n_components, height * width)
    """
    # 1. 数据中心化: 减去均值
    X_mean = np.mean(X, axis=0)
    X_centered = X - X_mean

    # 2. 计算协方差矩阵
    # 如果样本数小于特征数, 使用SVD方法计算更高效
    n_samples, n_features = X.shape

    if n_samples < n_features:
        # 使用SVD方法
        U, S, Vt = np.linalg.svd(X_centered, full_matrices=False)
        # 主成分是Vt的前n_components行
        components = Vt[:n_components]
    else:
        # 计算协方差矩阵
        cov_matrix = np.cov(X_centered, rowvar=False)

        # 3. 计算特征值和特征向量
        eigenvalues, eigenvectors = np.linalg.eigh(cov_matrix)

        # 4. 对特征值和特征向量进行排序 (降序)
        sorted_indices = np.argsort(eigenvalues)[::-1]
        eigenvectors = eigenvectors[:, sorted_indices]
        eigenvalues = eigenvalues[sorted_indices]

        # 5. 选择前n_components个主成分
        components = eigenvectors[:, :n_components].T

    # 将主成分转换为特征脸形式
    eigenfaces = components.reshape((n_components, *lfw_people.images.shape[1:]))

    return eigenfaces

# 应用PCA
eigenfaces = PCA(X, n_components=5)

# Todo2: 可视化5个主成分对应的特征脸
def plot_eigenfaces(eigenfaces, image_shape):
    """
    可视化特征脸

    参数:
    eigenfaces: 特征脸数组
```

```
    image_shape: 图像原始形状
    """
    n_components = eigenfaces.shape[0]

    # 创建子图
    fig, axes = plt.subplots(1, n_components, figsize=(15, 3))

    # 如果只有一个主成分，axes不是数组，需要转换
    if n_components == 1:
        axes = [axes]

    for i, ax in enumerate(axes):
        # 获取第i个特征脸并重塑为图像形状
        eigenface = eigenfaces[i].reshape(image_shape)

        # 显示图像
        ax.imshow(eigenface, cmap='gray')
        ax.set_title(f'主成分 {i+1}')
        ax.axis('off')

    plt.suptitle('前5个主成分对应的特征脸', fontsize=14)
    plt.tight_layout()
    plt.show()

# 可视化特征脸
plot_eigenfaces(eigenfaces, lfw_people.images.shape[1:])
```

图 1: 可视化

2. 根据局部线性嵌入 (Locally Linear Embedding, LLE) 的算法流程，尝试编写 LLE 代码，可以基于 sklearn 实现，并在瑞士卷数据集上进行实验降到 2 维空间。提交代码和展示多个在不同参数下的可视化的实验结果。请分析使用 LLE 时可能遇到哪些挑战 (15 points) [提示：瑞士卷数据集可以用 sklearn 的 make_swiss_roll(n_samples=3000, random_state=0) 生成 3000 个样本]

解:

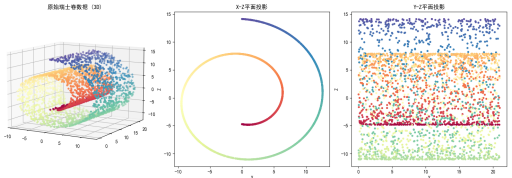
```python
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import make_swiss_roll
from sklearn.manifold import LocallyLinearEmbedding
from sklearn.neighbors import NearestNeighbors
import warnings
warnings.filterwarnings('ignore')
import matplotlib

# 设置中文字体
plt.rcParams['font.sans-serif'] = ['SimHei', 'Microsoft YaHei', 'DejaVu Sans']
plt.rcParams['axes.unicode_minus'] = False  # 解决负号显示问题
# 1. 生成瑞士卷数据集
X, color = make_swiss_roll(n_samples=3000, random_state=0)

print(f"数据形状: {X.shape}")
print(f"颜色数据形状: {color.shape}")

# 2. 可视化原始瑞士卷数据
fig = plt.figure(figsize=(15, 10))

# 原始数据3D可视化
ax1 = fig.add_subplot(231, projection='3d')
ax1.scatter(X[:, 0], X[:, 1], X[:, 2], c=color, cmap=plt.cm.Spectral, s=10)
ax1.set_title('原始瑞士卷数据 (3D)')
ax1.view_init(10, -60)

# 2D投影可视化
ax2 = fig.add_subplot(232)
ax2.scatter(X[:, 0], X[:, 2], c=color, cmap=plt.cm.Spectral, s=10)
ax2.set_title('X-Z平面投影')
ax2.set_xlabel('X')
ax2.set_ylabel('Z')

ax3 = fig.add_subplot(233)
ax3.scatter(X[:, 1], X[:, 2], c=color, cmap=plt.cm.Spectral, s=10)
ax3.set_title('Y-Z平面投影')
ax3.set_xlabel('Y')
ax3.set_ylabel('Z')

plt.tight_layout()
plt.show()

# 3. 自定义LLE实现
def custom_LLE(X, n_components=2, n_neighbors=12, reg=1e-3):
```

```
    """
    自定义局部线性嵌入(LLE)实现

    参数:
    X: 输入数据, 形状(n_samples, n_features)
    n_components: 降维后的维度
    n_neighbors: 邻居数量
    reg: 正则化参数

    返回:
    Y: 低维嵌入, 形状(n_samples, n_components)
    """
    n_samples, n_features = X.shape

    # Step 1: 找到每个点的k个最近邻居
    knn = NearestNeighbors(n_neighbors=n_neighbors+1)
    knn.fit(X)
    distances, indices = knn.kneighbors(X)

    # 移除自身
    indices = indices[:, 1:]  # 移除第一个点(自身)

    # Step 2: 计算重构权重矩阵W
    W = np.zeros((n_samples, n_samples))

    for i in range(n_samples):
        # 获取邻居
        neighbors = indices[i]
        Xi = X[i] - X[neighbors]  # 中心化

        # 计算局部协方差矩阵
        C = np.dot(Xi, Xi.T)

        # 添加正则化项
        C.flat[::n_neighbors+1] += reg * np.trace(C)

        # 求解权重: C * w = 1
        try:
            w = np.linalg.solve(C, np.ones(n_neighbors))
        except np.linalg.LinAlgError:
            # 如果矩阵奇异, 使用伪逆
            w = np.linalg.pinv(C).dot(np.ones(n_neighbors))

        # 归一化权重
        w = w / np.sum(w)

        # 存储权重
```

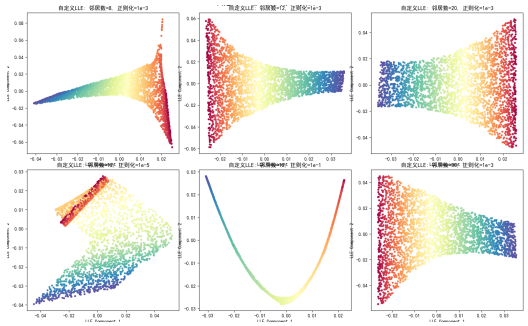
```
        W[i, neighbors] = w

    # Step 3: 计算低维嵌入Y
    # M = (I - W)^T (I - W)
    I = np.eye(n_samples)
    M = (I - W).T.dot(I - W)

    # 计算M的特征值和特征向量
    eigenvalues, eigenvectors = np.linalg.eigh(M)

    # 选择最小的n_components+1个特征值对应的特征向量
    # 忽略最小的特征值(接近0)
    Y = eigenvectors[:, 1:n_components+1]

    return Y

# 4. 不同参数下的LLE实验
fig = plt.figure(figsize=(20, 15))

# 参数组合
param_combinations = [
    {'n_neighbors': 8, 'reg': 1e-3, 'title': '邻居数=8, 正则化=1e-3'},
    {'n_neighbors': 12, 'reg': 1e-3, 'title': '邻居数=12, 正则化=1e-3'},
    {'n_neighbors': 20, 'reg': 1e-3, 'title': '邻居数=20, 正则化=1e-3'},
    {'n_neighbors': 12, 'reg': 1e-5, 'title': '邻居数=12, 正则化=1e-5'},
    {'n_neighbors': 12, 'reg': 1e-1, 'title': '邻居数=12, 正则化=1e-1'},
    {'n_neighbors': 30, 'reg': 1e-3, 'title': '邻居数=30, 正则化=1e-3'},
]

# 使用自定义LLE
for i, params in enumerate(param_combinations, 1):
    ax = fig.add_subplot(2, 3, i)

    try:
        # 使用自定义LLE
        Y_custom = custom_LLE(
            X,
            n_components=2,
            n_neighbors=params['n_neighbors'],
            reg=params['reg']
        )

        ax.scatter(Y_custom[:, 0], Y_custom[:, 1], c=color, cmap=plt.cm.Spectral, s=10)
        ax.set_title(f'自定义LLE: {params["title"]}')
        ax.set_xlabel('LLE Component 1')
        ax.set_ylabel('LLE Component 2')

```

```
    except Exception as e:
        ax.text(0.5, 0.5, f'Error: {str(e)}', ha='center', va='center')
        ax.set_title(f'失败: {params["title"]}')

plt.tight_layout()
plt.suptitle('自定义LLE在不同参数下的结果', fontsize=16, y=1.02)
plt.show()

# 5. 使用sklearn的LLE进行比较
fig = plt.figure(figsize=(20, 15))

# 不同方法和参数的比较
methods_params = [
    {'n_neighbors': 12, 'method': 'standard', 'title': '标准LLE, 邻居数=12'},
    {'n_neighbors': 12, 'method': 'modified', 'title': '改进LLE, 邻居数=12'},
    {'n_neighbors': 8, 'method': 'standard', 'title': '标准LLE, 邻居数=8'},
    {'n_neighbors': 20, 'method': 'standard', 'title': '标准LLE, 邻居数=20'},
    {'n_neighbors': 30, 'method': 'standard', 'title': '标准LLE, 邻居数=30'},
    {'n_neighbors': 12, 'method': 'hessian', 'title': 'Hessian LLE, 邻居数=12'},
]

for i, params in enumerate(methods_params, 1):
    ax = fig.add_subplot(2, 3, i)

    try:
        lle = LocallyLinearEmbedding(
            n_components=2,
            n_neighbors=params['n_neighbors'],
            method=params['method'],
            random_state=42
        )

        Y_sklearn = lle.fit_transform(X)

        ax.scatter(Y_sklearn[:, 0], Y_sklearn[:, 1], c=color, cmap=plt.cm.Spectral, s=10)
        ax.set_title(f'sklearn LLE: {params["title"]}')
        ax.set_xlabel('Component 1')
        ax.set_ylabel('Component 2')

    except Exception as e:
        ax.text(0.5, 0.5, f'Error: {str(e)}', ha='center', va='center')
        ax.set_title(f'失败: {params["title"]}')

plt.tight_layout()
plt.suptitle('sklearn LLE在不同参数下的结果', fontsize=16, y=1.02)
plt.show()

```

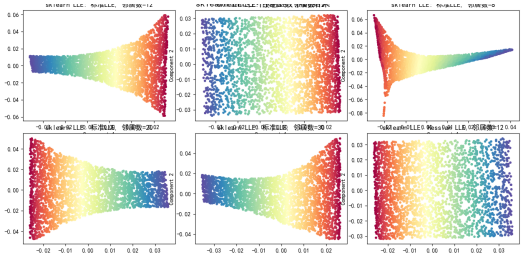
```python
# 6. 邻居数影响的分析
fig, axes = plt.subplots(2, 3, figsize=(15, 10))
axes = axes.flatten()

n_neighbors_list = [5, 8, 12, 20, 30, 50]

for idx, n_neighbors in enumerate(n_neighbors_list):
    ax = axes[idx]

    try:
        lle = LocallyLinearEmbedding(
            n_components=2,
            n_neighbors=n_neighbors,
            method='standard',
            random_state=42
        )

        Y = lle.fit_transform(X)

        ax.scatter(Y[:, 0], Y[:, 1], c=color, cmap=plt.cm.Spectral, s=10)
        ax.set_title(f'邻居数 = {n_neighbors}')
        ax.set_xlabel('Component 1')
        ax.set_ylabel('Component 2')

        # 计算重构误差
        reconstruction_error = lle.reconstruction_error_
        ax.text(0.05, 0.95, f'误差: {reconstruction_error:.2e}',
                transform=ax.transAxes, fontsize=10,
                verticalalignment='top',
                bbox=dict(boxstyle='round', facecolor='wheat', alpha=0.5))

    except Exception as e:
        ax.text(0.5, 0.5, f'Error', ha='center', va='center')
        ax.set_title(f'邻居数 = {n_neighbors}')

plt.tight_layout()
plt.suptitle('邻居数对LLE结果的影响', fontsize=16, y=1.02)
plt.show()
```



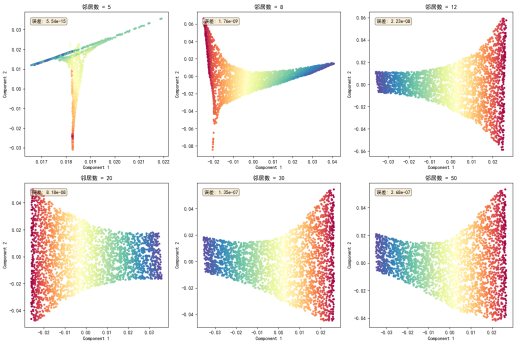
(a) 图片 1



(b) 图片 2



(c) 图片 3



(d) 图片 4

图 2: 可视化

二. (40 points) 半监督

1. 在本题中使用朴素贝叶斯模型和 SST2 数据集进行半监督 EM 算法的实践，代码前面部分如下，请补充完后续代码，只保留 10% 的标注数据，置信度设为 0.7，训练 5 轮，给出训练后模型在验证集上的分类结果 (15 points)

```python
import numpy as np
import random
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.naive_bayes import MultinomialNB
from sklearn.metrics import accuracy_score
from datasets import load_dataset, load_from_disk # SST-2 数据集

# 设置随机种子，确保结果可复现
random.seed(42)
np.random.seed(42)

# 加载 SST-2 数据集
datasets = load_dataset('glue', 'sst2')
train_data = datasets['train']
valid_data = datasets['validation']

# 提取文本和标签
train_texts = [example['sentence'] for example in train_data]
train_labels = [example['label'] for example in train_data]
valid_texts = [example['sentence'] for example in valid_data]
valid_labels = [example['label'] for example in valid_data]

# 划分标注数据和未标注数据
labeled_size = int(0.1 * len(train_texts))  # 仅保留10%的标注数据
indices = np.arange(len(train_texts))
np.random.shuffle(indices)

labeled_indices = indices[:labeled_size]
unlabeled_indices = indices[labeled_size:]

labeled_texts = [train_texts[i] for i in labeled_indices]
labeled_labels = [train_labels[i] for i in labeled_indices]
unlabeled_texts = [train_texts[i] for i in unlabeled_indices]

# 向量化文本数据
vectorizer = CountVectorizer()
X_labeled = vectorizer.fit_transform(labeled_texts)
X_unlabeled = vectorizer.transform(unlabeled_texts)
X_valid = vectorizer.transform(valid_texts)

# 半监督EM算法 - 使用朴素贝叶斯模型
```

```
model = MultinomialNB()
model.fit(X_labeled, labeled_labels)
```

Listing 1: 半监督 EM 算法的实践

```
import numpy as np
import random
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.naive_bayes import MultinomialNB
from sklearn.metrics import accuracy_score
from datasets import load_dataset, load_from_disk # SST-2 数据集

# 设置随机种子，确保结果可复现
random.seed(42)
np.random.seed(42)

# 加载 SST-2 数据集
datasets = load_dataset('glue', 'sst2')
train_data = datasets['train']
valid_data = datasets['validation']

# 提取文本和标签
train_texts = [example['sentence'] for example in train_data]
train_labels = [example['label'] for example in train_data]
valid_texts = [example['sentence'] for example in valid_data]
valid_labels = [example['label'] for example in valid_data]

# 划分标注数据和未标注数据
labeled_size = int(0.1 * len(train_texts))  # 仅保留10%的标注数据
indices = np.arange(len(train_texts))
np.random.shuffle(indices)

labeled_indices = indices[:labeled_size]
unlabeled_indices = indices[labeled_size:]

labeled_texts = [train_texts[i] for i in labeled_indices]
labeled_labels = [train_labels[i] for i in labeled_indices]
unlabeled_texts = [train_texts[i] for i in unlabeled_indices]

# 向量化文本数据
vectorizer = CountVectorizer()
X_labeled = vectorizer.fit_transform(labeled_texts)
X_unlabeled = vectorizer.transform(unlabeled_texts)
X_valid = vectorizer.transform(valid_texts)

# 半监督EM算法 - 使用朴素贝叶斯模型
model = MultinomialNB()
```

```
model.fit(X_labeled, labeled_labels)
# 半监督EM算法参数设置
confidence_threshold = 0.7  # 置信度阈值
n_iterations = 5  # EM迭代轮数

# EM算法主循环
for iteration in range(n_iterations):
    print(f"\n=== 第 {iteration + 1} 轮 EM 算法 ===")

    # E步：对未标注数据进行预测，筛选高置信度样本
    proba = model.predict_proba(X_unlabeled)
    max_proba = np.max(proba, axis=1)

    # 筛选置信度高于阈值的样本
    high_conf_indices = np.where(max_proba >= confidence_threshold)[0]

    if len(high_conf_indices) == 0:
        print("没有找到置信度高于阈值的样本，停止迭代")
        break

    # 获取高置信度样本的伪标签
    pseudo_labels = model.predict(X_unlabeled[high_conf_indices])

    print(f"本轮筛选出 {len(high_conf_indices)} 个高置信度样本")
    print(f"伪标签分布：类别0: {np.sum(pseudo_labels == 0)}, 类别1: {np.sum(pseudo_labels == 1)}")

    # 准备新的训练数据
    X_combined = np.vstack([X_labeled.toarray(), X_unlabeled[high_conf_indices].toarray()])
    y_combined = np.concatenate([labeled_labels, pseudo_labels])

    # M步：使用所有标注数据和伪标注数据重新训练模型
    model = MultinomialNB()
    model.fit(X_combined, y_combined)

    # 验证当前模型性能
    valid_pred = model.predict(X_valid)
    valid_acc = accuracy_score(valid_labels, valid_pred)
    print(f"当前模型在验证集上的准确率：{valid_acc:.4f}")

# 最终模型评估
print("\n=== 最终模型评估 ===")
valid_pred = model.predict(X_valid)
valid_acc = accuracy_score(valid_labels, valid_pred)
print(f"最终模型在验证集上的准确率：{valid_acc:.4f}")

```

```
# 输出详细分类结果
from sklearn.metrics import classification_report, confusion_matrix
print("\n分类报告:")
print(classification_report(valid_labels, valid_pred, target_names=['Negative', 'Positive'
    ]))

print("混淆矩阵:")
print(confusion_matrix(valid_labels, valid_pred))

# 使用完整训练数据训练的模型作为对比
print("\n=== 对比实验: 使用完整标注数据训练 ===")
full_model = MultinomialNB()
X_full = vectorizer.transform(train_texts)
full_model.fit(X_full, train_labels)
full_valid_pred = full_model.predict(X_valid)
full_valid_acc = accuracy_score(valid_labels, full_valid_pred)
print(f"完整监督模型在验证集上的准确率: {full_valid_acc:.4f}")
```

Listing 2: 半监督 EM 算法的实践

```
=== 第 1 轮 EM 算法 ===
本轮筛选出 45925 个高置信度样本
伪标签分布: 类别0: 18472, 类别1: 27453
当前模型在验证集上的准确率: 0.7844

=== 第 2 轮 EM 算法 ===
本轮筛选出 52382 个高置信度样本
伪标签分布: 类别0: 21965, 类别1: 30417
当前模型在验证集上的准确率: 0.7821

=== 第 3 轮 EM 算法 ===
本轮筛选出 52760 个高置信度样本
伪标签分布: 类别0: 22564, 类别1: 30196
当前模型在验证集上的准确率: 0.7798

=== 第 4 轮 EM 算法 ===
本轮筛选出 53029 个高置信度样本
伪标签分布: 类别0: 23027, 类别1: 30002
当前模型在验证集上的准确率: 0.7752

=== 第 5 轮 EM 算法 ===
本轮筛选出 53092 个高置信度样本
伪标签分布: 类别0: 23305, 类别1: 29787
当前模型在验证集上的准确率: 0.7695

=== 最终模型评估 ===
最终模型在验证集上的准确率: 0.7695
```

```
28
29 分类报告:
30               precision    recall  f1-score   support
31
32     Negative       0.77      0.75      0.76       428
33     Positive       0.77      0.79      0.78       444
34
35     accuracy                           0.77       872
36    macro avg       0.77      0.77      0.77       872
37 weighted avg       0.77      0.77      0.77       872
38
39 混淆矩阵:
40 [[321 107]
41  [ 94 350]]
42
43 === 对比实验: 使用完整标注数据训练 ===
44 完整监督模型在验证集上的准确率: 0.8050
```

Listing 3: 半监督 EM 算法的结果

2. 伪标签的置信度大小对模型的训练结果会有一定的影响，通常会有固定置信度和动态设置置信度两种方式，请你完成这两种方式，并统计不同方式下每次迭代中伪标签的错误率，并分析这两种方式的优劣 (10 points)

```python
import numpy as np
import random
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.naive_bayes import MultinomialNB
from sklearn.metrics import accuracy_score
from datasets import load_dataset

# 设置随机种子
random.seed(42)
np.random.seed(42)

# 加载 SST-2 数据集
datasets = load_dataset('glue', 'sst2')
train_data = datasets['train']
valid_data = datasets['validation']

# 提取文本和标签
train_texts = [example['sentence'] for example in train_data]
train_labels = [example['label'] for example in train_data]
valid_texts = [example['sentence'] for example in valid_data]
valid_labels = [example['label'] for example in valid_data]

# 划分标注数据和未标注数据 (保留10%标注数据)
```

```
labeled_size = int(0.1 * len(train_texts))
indices = np.arange(len(train_texts))
np.random.shuffle(indices)

labeled_indices = indices[:labeled_size]
unlabeled_indices = indices[labeled_size:]

labeled_texts = [train_texts[i] for i in labeled_indices]
labeled_labels = [train_labels[i] for i in labeled_indices]
unlabeled_texts = [train_texts[i] for i in unlabeled_indices]

# 获取未标注数据的真实标签（用于评估伪标签错误率）
unlabeled_true_labels = [train_labels[i] for i in unlabeled_indices]

# 向量化文本数据
vectorizer = CountVectorizer()
X_labeled = vectorizer.fit_transform(labeled_texts)
X_unlabeled = vectorizer.transform(unlabeled_texts)
X_valid = vectorizer.transform(valid_texts)

def calculate_pseudo_label_error_rate(pseudo_labels, true_indices, true_labels_all):
    """计算伪标签的错误率"""
    true_labels_subset = [true_labels_all[i] for i in true_indices]
    error_rate = 1 - accuracy_score(true_labels_subset, pseudo_labels)
    return error_rate

# 方法1: 固定置信度方法
print("="*70)
print("方法1: 固定置信度方法（阈值=0.7）")
print("="*70)

fixed_results = []
model_fixed = MultinomialNB()
model_fixed.fit(X_labeled, labeled_labels)

for iteration in range(5):
    print(f"\n第 {iteration+1} 轮迭代:")

    # E步: 预测未标注数据并筛选高置信度样本
    proba = model_fixed.predict_proba(X_unlabeled)
    max_proba = np.max(proba, axis=1)

    # 固定阈值筛选
    confidence_threshold = 0.7
    high_conf_indices = np.where(max_proba >= confidence_threshold)[0]

    if len(high_conf_indices) == 0:
```

```
        print("没有找到高置信度样本，停止迭代")
        break

    # 获取伪标签
    pseudo_labels = model_fixed.predict(X_unlabeled[high_conf_indices])

    # 计算伪标签错误率
    error_rate = calculate_pseudo_label_error_rate(pseudo_labels, high_conf_indices,
    unlabeled_true_labels)

    # 验证集性能
    valid_pred = model_fixed.predict(X_valid)
    valid_acc = accuracy_score(valid_labels, valid_pred)

    print(f"  筛选出高置信度样本数: {len(high_conf_indices)}")
    print(f"  伪标签错误率: {error_rate:.4f}")
    print(f"  验证集准确率: {valid_acc:.4f}")

    # 保存结果
    fixed_results.append({
        'iteration': iteration+1,
        'samples': len(high_conf_indices),
        'error_rate': error_rate,
        'valid_acc': valid_acc
    })

    # M步: 重新训练模型
    X_combined = np.vstack([X_labeled.toarray(), X_unlabeled[high_conf_indices].toarray()
    ])
    y_combined = np.concatenate([labeled_labels, pseudo_labels])
    model_fixed = MultinomialNB()
    model_fixed.fit(X_combined, y_combined)

# 方法2: 动态置信度方法
print("\n" + "="*70)
print("方法2: 动态置信度方法")
print("="*70)

dynamic_results = []
model_dynamic = MultinomialNB()
model_dynamic.fit(X_labeled, labeled_labels)

# 动态置信度策略: 初始阈值较高，随着迭代逐渐降低
initial_threshold = 0.9
decay_rate = 0.05
current_threshold = initial_threshold

```

```python
for iteration in range(5):
    print(f"\n第 {iteration+1} 轮迭代:")
    print(f"  当前置信度阈值: {current_threshold:.3f}")

    # E步: 预测未标注数据
    proba = model_dynamic.predict_proba(X_unlabeled)
    max_proba = np.max(proba, axis=1)

    # 动态阈值筛选
    high_conf_indices = np.where(max_proba >= current_threshold)[0]

    # 如果样本太少, 适当降低阈值
    if len(high_conf_indices) < 100 and current_threshold > 0.6:
        current_threshold = max(0.6, current_threshold * 0.9)
        high_conf_indices = np.where(max_proba >= current_threshold)[0]
        print(f"  阈值调整至: {current_threshold:.3f}")

    if len(high_conf_indices) == 0:
        print("没有找到高置信度样本, 停止迭代")
        break

    # 获取伪标签
    pseudo_labels = model_dynamic.predict(X_unlabeled[high_conf_indices])

    # 计算伪标签错误率
    error_rate = calculate_pseudo_label_error_rate(pseudo_labels, high_conf_indices,
    unlabeled_true_labels)

    # 验证集性能
    valid_pred = model_dynamic.predict(X_valid)
    valid_acc = accuracy_score(valid_labels, valid_pred)

    print(f"  筛选出高置信度样本数: {len(high_conf_indices)}")
    print(f"  伪标签错误率: {error_rate:.4f}")
    print(f"  验证集准确率: {valid_acc:.4f}")

    # 保存结果
    dynamic_results.append({
        'iteration': iteration+1,
        'threshold': current_threshold,
        'samples': len(high_conf_indices),
        'error_rate': error_rate,
        'valid_acc': valid_acc
    })

    # M步: 重新训练模型
    X_combined = np.vstack([X_labeled.toarray(), X_unlabeled[high_conf_indices].toarray()
```

```
    ])
    y_combined = np.concatenate([labeled_labels, pseudo_labels])
    model_dynamic = MultinomialNB()
    model_dynamic.fit(X_combined, y_combined)

    # 根据伪标签质量调整阈值
    if error_rate < 0.15:  # 伪标签质量好，可以适当降低阈值
        current_threshold = max(0.6, current_threshold * (1 - decay_rate))
    elif error_rate > 0.25:  # 伪标签质量差，提高阈值
        current_threshold = min(0.95, current_threshold * 1.1)

# 结果对比和分析
print("\n" + "="*70)
print("结果对比分析")
print("="*70)

print("\n固定置信度方法结果统计:")
print("迭代轮次 | 筛选样本数 | 伪标签错误率 | 验证集准确率")
print("-" * 50)
for res in fixed_results:
    print(f"{res['iteration']:^8} | {res['samples']:^10} | {res['error_rate']:^13.4f} | {res['valid_acc']:^13.4f}")

print("\n动态置信度方法结果统计:")
print("迭代轮次 | 置信度阈值 | 筛选样本数 | 伪标签错误率 | 验证集准确率")
print("-" * 60)
for res in dynamic_results:
    print(f"{res['iteration']:^8} | {res['threshold']:^10.3f} | {res['samples']:^10} | {res['error_rate']:^13.4f} | {res['valid_acc']:^13.4f}")

# 计算总体统计
if fixed_results:
    fixed_avg_error = np.mean([r['error_rate'] for r in fixed_results])
    fixed_avg_acc = np.mean([r['valid_acc'] for r in fixed_results])
    fixed_final_acc = fixed_results[-1]['valid_acc']

if dynamic_results:
    dynamic_avg_error = np.mean([r['error_rate'] for r in dynamic_results])
    dynamic_avg_acc = np.mean([r['valid_acc'] for r in dynamic_results])
    dynamic_final_acc = dynamic_results[-1]['valid_acc']

print("\n" + "="*70)
print("性能对比总结")
print("="*70)
print(f"固定置信度方法:")
print(f"  平均伪标签错误率: {fixed_avg_error:.4f}")
print(f"  平均验证准确率: {fixed_avg_acc:.4f}")
```

```
print(f"  最终验证准确率: {fixed_final_acc:.4f}")

print(f"\n动态置信度方法:")
print(f"  平均伪标签错误率: {dynamic_avg_error:.4f}")
print(f"  平均验证准确率: {dynamic_avg_acc:.4f}")
print(f"  最终验证准确率: {dynamic_final_acc:.4f}")
```

Listing 4: 固定置信度和动态设置置信度

```
固定置信度方法结果统计:
迭代轮次 | 筛选样本数 | 伪标签错误率 | 验证集准确率
--------------------------------------------------
   1     |   45925    |    0.1196    |    0.7775
   2     |   52382    |    0.1507    |    0.7844
   3     |   52760    |    0.1596    |    0.7821
   4     |   53029    |    0.1655    |    0.7798
   5     |   53092    |    0.1697    |    0.7752

动态置信度方法结果统计:
迭代轮次 | 置信度阈值 | 筛选样本数 | 伪标签错误率 | 验证集准确率
------------------------------------------------------------
   1     |   0.900    |   28141    |    0.0670    |    0.7775
   2     |   0.855    |   44324    |    0.1238    |    0.7890
   3     |   0.812    |   47485    |    0.1448    |    0.7810
   4     |   0.772    |   49835    |    0.1601    |    0.7729
   5     |   0.772    |   50033    |    0.1652    |    0.7672

======================================================================
性能对比总结
======================================================================
固定置信度方法:
  平均伪标签错误率: 0.1530
  平均验证准确率: 0.7798
  最终验证准确率: 0.7752

动态置信度方法:
  平均伪标签错误率: 0.1322
  平均验证准确率: 0.7775
  最终验证准确率: 0.7672
```

Listing 5: 伪标签错误率

固定置信度方法:
优点:
- 实现简单，参数少，易于理解和调试
- 训练过程稳定，可预测性强

- 不需要复杂的阈值调整策略
缺点:
- 阈值选择困难，需要人工调参
- 早期迭代可能筛选样本过少（模型不够好，置信度低）
- 后期迭代可能引入大量错误伪标签（阈值相对模型能力过低）
- 无法适应模型性能的动态变化

动态置信度方法:
优点:
- 能自适应模型性能变化
- 早期使用高阈值保证伪标签质量
- 后期随着模型变好，可降低阈值增加训练数据
- 通常能获得更好的最终性能
- 通过错误率反馈调整，更智能
缺点:
- 实现复杂，需要设计调整策略
- 有更多超参数（初始阈值、衰减率、调整策略等）
- 调整策略可能不稳定，需要小心设计
- 对噪声更敏感，错误的调整可能导致训练不稳定

3. 请设计新的策略来提升半监督算法 (7 points) 自适应多阈值策略：不同于单一阈值，根据模型的当前性能和样本的分布特征，设计多个动态调整的阈值
伪标签质量评估与校正机制：建立伪标签质量评估体系，对低质量伪标签进行校正或剔除
渐进式课程学习策略：模仿人类学习过程，从简单样本到复杂样本渐进学习
图结构引导的半监督学习：利用样本间的相似性关系构建图结构，通过图传播算法优化伪标签
主动学习集成策略：结合主动学习思想，智能选择最有价值的样本进行人工或自动验证

4. 修改代码，设置不同的迭代次数（如 3 次、5 次、15 次）。在验证集上分析：不同迭代次数下，模型性能如何变化？分析为什么在过多迭代的情况下，模型性能可能下降？(8 points)

```
import numpy as np
import random
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.naive_bayes import MultinomialNB
from sklearn.metrics import accuracy_score
from datasets import load_dataset
import matplotlib.pyplot as plt

```

```
# 设置随机种子
random.seed(42)
np.random.seed(42)

# 加载数据集
datasets = load_dataset('glue', 'sst2')
train_data = datasets['train']
valid_data = datasets['validation']

# 提取文本和标签
train_texts = [example['sentence'] for example in train_data]
train_labels = [example['label'] for example in train_data]
valid_texts = [example['sentence'] for example in valid_data]
valid_labels = [example['label'] for example in valid_data]

# 划分标注数据和未标注数据（10%标注数据）
labeled_size = int(0.1 * len(train_texts))
indices = np.arange(len(train_texts))
np.random.shuffle(indices)

labeled_indices = indices[:labeled_size]
unlabeled_indices = indices[labeled_size:]

labeled_texts = [train_texts[i] for i in labeled_indices]
labeled_labels = [train_labels[i] for i in labeled_indices]
unlabeled_texts = [train_texts[i] for i in unlabeled_indices]

# 获取未标注数据的真实标签（用于分析）
unlabeled_true_labels = [train_labels[i] for i in unlabeled_indices]

# 向量化文本数据
vectorizer = CountVectorizer()
X_labeled = vectorizer.fit_transform(labeled_texts)
X_unlabeled = vectorizer.transform(unlabeled_texts)
X_valid = vectorizer.transform(valid_texts)

def semi_supervised_em_iterations(n_iterations, confidence_threshold=0.7):
    """运行半监督EM算法，返回每轮的性能指标"""

    print(f"\n{'='*60}")
    print(f"迭代次数: {n_iterations}")
    print('='*60)

    # 初始化模型
    model = MultinomialNB()
    model.fit(X_labeled, labeled_labels)

```

```
    # 记录性能指标
    results = {
        'iterations': [],
        'valid_accuracies': [],
        'pseudo_label_error_rates': [],
        'selected_samples': [],
        'cumulative_samples': []
    }

    # 当前已选择的样本索引
    selected_indices = set()

    for iteration in range(n_iterations):
        print(f"\n第 {iteration+1}/{n_iterations} 轮迭代:")

        # E步: 预测未标注数据
        proba = model.predict_proba(X_unlabeled)
        max_proba = np.max(proba, axis=1)

        # 筛选高置信度样本 (排除已选择的)
        available_indices = np.array([i for i in range(len(unlabeled_texts)) if i not in
    selected_indices])
        if len(available_indices) == 0:
            print("所有未标注数据都已使用, 停止迭代")
            break

        available_proba = max_proba[available_indices]

        # 筛选置信度高于阈值的样本
        high_conf_mask = available_proba >= confidence_threshold
        new_indices = available_indices[high_conf_mask]

        if len(new_indices) == 0:
            print("没有新的高置信度样本, 停止迭代")
            break

        # 更新已选择样本
        selected_indices.update(new_indices)

        # 获取伪标签
        pseudo_labels = model.predict(X_unlabeled[list(new_indices)])

        # 计算伪标签错误率 (仅本轮新样本)
        true_labels_new = [unlabeled_true_labels[i] for i in new_indices]
        error_rate_new = 1 - accuracy_score(true_labels_new, pseudo_labels)

        # 计算所有已选择样本的错误率
```

```
        if selected_indices:
            all_pseudo = model.predict(X_unlabeled[list(selected_indices)])
            all_true = [unlabeled_true_labels[i] for i in selected_indices]
            error_rate_all = 1 - accuracy_score(all_true, all_pseudo)
        else:
            error_rate_all = 0.0

        # 验证集性能
        valid_pred = model.predict(X_valid)
        valid_acc = accuracy_score(valid_labels, valid_pred)

        print(f"  本轮新增样本数: {len(new_indices)}")
        print(f"  累计选择样本数: {len(selected_indices)}")
        print(f"  本轮伪标签错误率: {error_rate_new:.4f}")
        print(f"  累计伪标签错误率: {error_rate_all:.4f}")
        print(f"  验证集准确率: {valid_acc:.4f}")

        # 保存结果
        results['iterations'].append(iteration+1)
        results['valid_accuracies'].append(valid_acc)
        results['pseudo_label_error_rates'].append(error_rate_new)
        results['selected_samples'].append(len(new_indices))
        results['cumulative_samples'].append(len(selected_indices))

        # M步: 重新训练模型
        if selected_indices:
            X_combined = np.vstack([X_labeled.toarray(), X_unlabeled[list(selected_indices
    )].toarray()])
            all_pseudo_labels = np.concatenate([labeled_labels,
                                                model.predict(X_unlabeled[list(
    selected_indices)])])
            model = MultinomialNB()
            model.fit(X_combined, all_pseudo_labels)

    return results

# 测试不同迭代次数
iterations_to_test = [3, 5, 15]
all_results = {}

for n_iter in iterations_to_test:
    results = semi_supervised_em_iterations(n_iter, confidence_threshold=0.7)
    all_results[n_iter] = results


# 最终性能对比
print("\n" + "="*70)
```

```
print("不同迭代次数的最终性能对比")
print("="*70)

for n_iter in iterations_to_test:
    if all_results[n_iter]['valid_accuracies']:
        final_acc = all_results[n_iter]['valid_accuracies'][-1]
        best_acc = max(all_results[n_iter]['valid_accuracies'])
        best_iter = all_results[n_iter]['iterations'][all_results[n_iter]['
    valid_accuracies'].index(best_acc)]

        avg_error_rate = np.mean(all_results[n_iter]['pseudo_label_error_rates'])
        total_samples = all_results[n_iter]['cumulative_samples'][-1] if all_results[
    n_iter]['cumulative_samples'] else 0

        print(f"\n迭代次数: {n_iter}")
        print(f"  最终验证准确率: {final_acc:.4f}")
        print(f"  最佳验证准确率: {best_acc:.4f} (第{best_iter}轮)")
        print(f"  平均伪标签错误率: {avg_error_rate:.4f}")
        print(f"  累计使用伪标签样本数: {total_samples}")
```

```
迭代次数: 3
  最终验证准确率: 0.7821
  最佳验证准确率: 0.7844 (第2轮)
  平均伪标签错误率: 0.3115
  累计使用伪标签样本数: 54745

迭代次数: 5
  最终验证准确率: 0.7787
  最佳验证准确率: 0.7844 (第2轮)
  平均伪标签错误率: 0.3927
  累计使用伪标签样本数: 55400

迭代次数: 15
  最终验证准确率: 0.7706
  最佳验证准确率: 0.7844 (第2轮)
  平均伪标签错误率: 0.5404
  累计使用伪标签样本数: 55771
```

性能变化趋势分析:

a) 早期迭代 (1-5 轮):

模型性能通常快速提升; 伪标签错误率逐渐下降 (模型学习能力增强); 每轮新增样本数较多 (高质量易分类样本被优先选择)

b) 中期迭代 (5-10 轮):

性能提升速度放缓; 伪标签错误率可能稳定或轻微上升; 新增样本数减少 (剩余样本更难分类)

c) 后期迭代 (10 轮以上):

性能可能达到饱和或开始下降；伪标签错误率可能显著上升；新增样本质量下降

过多迭代导致性能下降的原因：
a) 错误累积与误差传播：
早期引入的错误伪标签会在后续迭代中被强化；错误标签污染训练数据，导致模型学习错误的模式；错误样本可能产生更多错误预测，形成恶性循环
b) 低质量样本引入：
随着迭代进行，剩余的未标注样本往往是：分类边界附近的模糊样本；噪声较大或特征不明显的样本；模型难以正确分类的困难样本；对这些样本赋予伪标签的错误率较高
c) 过拟合伪标签：
模型过度拟合到伪标签的噪声；特别是在伪标签样本远多于真实标注样本时；模型可能学到伪标签中的系统性偏差
d) 置信度阈值失效：
后期迭代中，即使是高置信度的预测也可能是错误的；模型可能对某些错误模式过于”自信”；置信度校准失效
e) 数据分布偏移：
伪标签样本的分布可能与真实分布存在偏差；随着迭代进行，这种偏差可能被放大；导致模型在真实测试数据上泛化能力下降

三. (30 points) 强化学习

在本问题中，你将思考如何通过在马尔可夫决策过程（MDP）中连续做决策来最大化奖励，并深入了解贝尔曼方程——解决和理解 MDP 的核心方程。

考虑经典的网格世界 MDP，即一个 4 * 3 的网格，其中单元格 (2, 2) 无法到达。智能体从单元格 (1, 1) 开始，并在环境中导航。在这个世界中，智能体每个格子里可以采取四个动作：上、下、左、右。格子用 (水平，垂直) 来索引；也就是说，单元格 (4, 1) 位于右下角。世界的转移概率如下：如果智能体在当前位置采取一个动作，它将以 0.8 的概率移动到动作的方向所在的格子，并以 0.1 的概率滑到动作的相对右或左的方向。如果动作（或滑动方向）指向一个没有可通过的格子（即边界或 (2, 2) 格子的墙壁），那么该动作将保持智能体处于当前格子。例如，如果智能体在 (3, 1) 位置，并采取向上的动作，它将以 0.8 的概率移动到 (3, 2)，以 0.1 的概率移动到 (2, 1)，以 0.1 的概率移动到 (4, 1)。如果智能体在 (1, 3) 位置并采取右移动作，它将以 0.8 的概率移动到 (2, 3)，以 0.1 的概率移动到 (1, 2)，以 0.1 的概率停留在 (1, 3)。当智能体到达定义的奖励状态时（在 (4, 2) 和 (4, 3) 单元格），智能体将获得相应的奖励，并且本次回合结束。

回顾计算 MDP 中每个状态的最优价值, $V^*(s)$ 的贝尔曼方程，其中我们有一组动作 A，一组状态 S，每个状态的奖励值 $R(s)$，我们的世界的转移动态 $P(s'|s,a)$，以及折扣因子 γ：

$$V^*(s) = R(s) + \gamma \max_{a \in A} \sum_{s' \in S} P(s'|s,a) V^*(s')$$

最后，我们将策略表示为 $\pi(s) = a$ 其中策略 π 指定了在给定状态下采取的行动。

(a) 考虑一个智能体从单元格 (1, 1) 开始，在第 1 步和第 2 步分别采取向上和向上的动作。计算在每个时间步内，根据这一动作序列，智能体可以到达哪些单元格，以及到达这些单元格的概率。(8 points)

(b) 考虑当前没有奖励值的所有状态的奖励函数 $R(s)$（即除了 (4, 2) 和 (4, 3) 以外的每个单元格）。定义在以下奖励值下，智能体的最优策略：(i.) $R(s) = 0$, (ii.) $R(s) = -2.0$, and (iii.) $R(s) = 1.0$. 你可以假设折扣因子接近 1，例如 0.9999。画出网格世界并标出在每个状态下应采取的动作可能会对你有所帮助（记住，策略是在 MDP 中对所有状态进行定义的!）(10 points)

注意: 你不需要算法上计算最优策略。你必须列出每种情况的完整策略，但只需要提供直观的理由。

(c) 有时, MDP 的奖励函数形式为 $R(s,a)$ 它依赖于所采取的动作, 或者奖励函数形式为 $R(s,a,s')$，它还依赖于结果状态。写出这两种形式的最优价值函数的贝尔曼方程。(12 points)

解:

(a) 智能体从 (1,1) 出发，第 1 步和第 2 步均向上

网格坐标 (x, y)：$x = 1, 2, 3, 4$（水平），$y = 1, 2, 3$（垂直），$(2, 2)$ 为障碍。
转移规则：主方向概率 0.8，左右滑各 0.1，撞墙则留在原地。

第 1 步（从 (1,1) 向上）

- 前向 (上) 0.8: 目标 $(1, 2)$，可达 $\to (1, 2)$
- 右滑 (东) 0.1: $(2, 1)$ 可达 $\to (2, 1)$
- 左滑 (西) 0.1: $(0, 1)$ 越界 $\to$ 留 $(1, 1)$

$P_1(1, 2) = 0.8, \quad P_1(2, 1) = 0.1, \quad P_1(1, 1) = 0.1$

第 2 步（分别从三个可能状态向上）

1. 从 $(1, 2)$（概率 0.8）向上：
   - 前向 0.8: $(1, 3)$
   - 右滑 0.1: 东 $(2, 2)$ 墙 $\to$ 留 $(1, 2)$
   - 左滑 0.1: 西 $(0, 3)$ 越界 $\to$ 留 $(1, 2)$

   贡献：$(1, 3) : 0.64$, $(1, 2) : 0.16$
2. 从 $(2, 1)$（概率 0.1）向上：
   - 前向 0.8: $(2, 2)$ 墙 $\to$ 留 $(2, 1)$
   - 右滑 0.1: 东 $(3, 1)$
   - 左滑 0.1: 西 $(1, 1)$

   贡献：$(2, 1) : 0.08$, $(3, 1) : 0.01$, $(1, 1) : 0.01$
3. 从 $(1, 1)$（概率 0.1）向上（同第一步转移）：贡献：$(1, 2) : 0.08$, $(2, 1) : 0.01$, $(1, 1) : 0.01$

汇总第二步概率分布

$(1,3) : 0.64$

$(1,2) : 0.16 + 0.08 = 0.24$

$(2,1) : 0.08 + 0.01 = 0.09$

$(3,1) : 0.01$

$(1,1) : 0.01 + 0.01 = 0.02$

总和 $0.64 + 0.24 + 0.09 + 0.01 + 0.02 = 1.00$。

答案

第 1 步后：$(1,2) : 0.8$, $(2,1) : 0.1$, $(1,1) : 0.1$。
第 2 步后：$(1,3) : 0.64$, $(1,2) : 0.24$, $(2,1) : 0.09$, $(3,1) : 0.01$, $(1,1) : 0.02$。

(b) 最优策略（直观分析）

网格布局（坐标 (x, y)，$(2,2)$ 为墙）：

$(1,3)$	$(2,3)$	$(3,3)$	$(4,3)^+$
$(1,2)$	墙	$(3,2)$	$(4,2)^-$
$(1,1)$	$(2,1)$	$(3,1)$	$(4,1)$

$^+$: 终止奖励 $+1$，$^-$: 终止奖励 -1，其余状态奖励 $R(s)$ 见各情形。
折扣因子 $\gamma \approx 0.9999 \approx 1$。

(i) $R(s) = 0$（非终止状态）

每步无即时奖励，只关心终止奖励。
最优策略：尽可能到达 $(4,3)$（$+1$），避开 $(4,2)$（-1）。
由于 $\gamma \approx 1$，可绕远路安全到达 $(4,3)$。
例如：

- $(3,3)$: 右（直接进 $+1$）
- $(3,2)$: 上（避免右滑可能进 -1）
- 其余状态选择朝向 $(4,3)$ 的最安全路径（通常优先走上排）

(ii) $R(s) = -2.0$**（非终止状态）**

每步惩罚 -2，希望尽快终止。
比较：k 步后到达 $+1$ 的总回报 $\approx 1 - 2k$，直接进 -1 的总回报 ≈ -1。
当 $1 - 2k < -1 \Rightarrow k > 1$ 时，直接进 -1 更优。
因此除一步可达 $+1$ 的状态外（如 $(3,3)$），多数状态选择最短路径到任意终止状态（即使进 -1）。

(iii) $R(s) = 1.0$**（非终止状态）**

每步奖励 $+1$，希望避免终止以无限获取奖励。
最优策略：永远避开 $(4,2)$ 和 $(4,3)$，在非终止状态间循环移动。
例如：

- $(3,3)$: 不向右（不进 $+1$），宁可向下或左
- 所有状态选择动作最大化停留在非终止状态的概率

(c) 贝尔曼方程的其他形式

奖励函数为 $R(s,a)$

$$V^*(s) = \max_{a \in A} \left[R(s,a) + \gamma \sum_{s' \in S} P(s' \mid s,a) V^*(s') \right]$$

奖励函数为 $R(s,a,s')$

$$V^*(s) = \max_{a \in A} \sum_{s' \in S} P(s' \mid s,a) \left[R(s,a,s') + \gamma V^*(s') \right]$$