清华大学
数据挖掘：原理与算法
2024 年秋季学期

大作业 1-动作识别聚类分析

杨天行　郑楚阳　李欣然　2025 年 1 月 3 日

1　实验任务

本次作业选用的数据集包含 30 名受试者在日常生活中的 10299 条动作记录，每条数据包含 1 个 561 维向量以及动作标签，分别用基于网格的方法、基于密度的方法实现了数据集的聚类分析任务，并分析结果将结果可视化。[1][2]

2　数据集分析及预处理

首先我们对数据集中的信息检查了缺失值和重复值，并可视化观察该数据集是否平衡，如下图所示，可以观察得到不同被试的动作数量没有显著的差异，数据基本是平衡的。

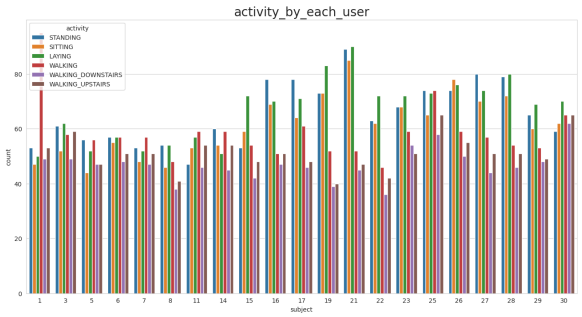

图 1: 不同受试者不同被试动作数量的对比柱状图

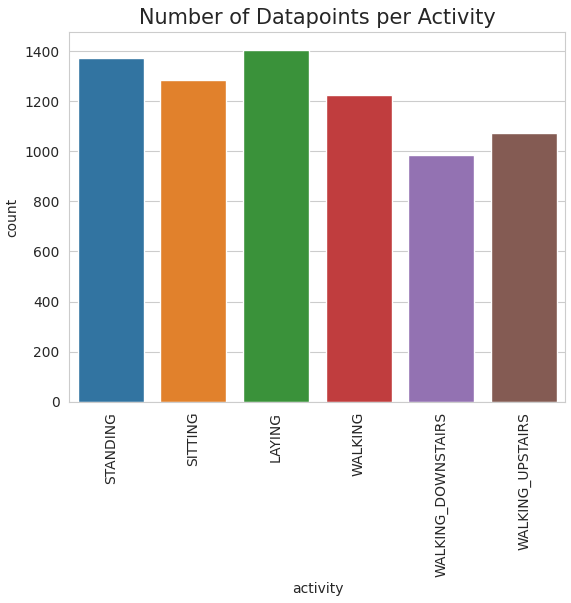

图 2: 不同测试动作的数量

我们对数据进行了可视化，以此决定数据集预处理方法。下面是该数据集的数值变量的相关性热图，可知在静态动作中 (sit,stand, lie down) 动态特征用处不大，在动态动作中 (walking, walkingupstairs,walkingdownstairs) 动态特征很重要

可知，以动态特征 tBodyAccMagmean 为例，可以看出静态动作更集中，而动态特征较为分散，都在-1 附近。

如下图，对于加速度的尺度来说，如果 tAccMean 小于-0.8，则动作是静态的如果 tAccMean 大于-0.6，则动作是动态的，如果 tAccMean 大于 0.0，则动作是 walkingdownstairs，基本上可以根据这一准则分类四分之三的动作。对于 angleXgravityMean 这一指标，如果 angleXgravityMean 大于 0，则动作是 laying。

[1] 大作业代码已开源在`https://github.com/yangtx7/DataMining-Project1`

[2] 任务分工：杨天行、郑楚阳：数据分析与实验；李欣然：撰写报告

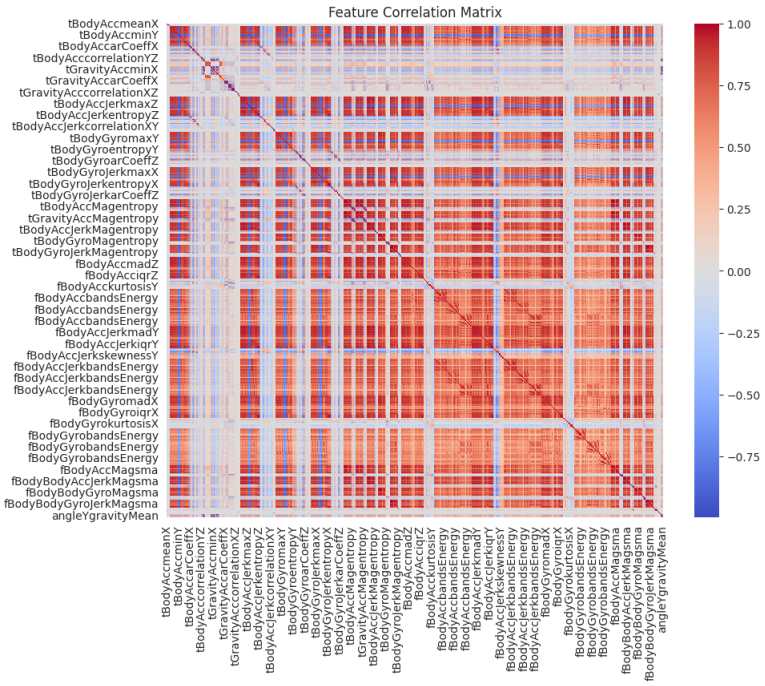

图 3: 各个特征之间相关性的 heatmap

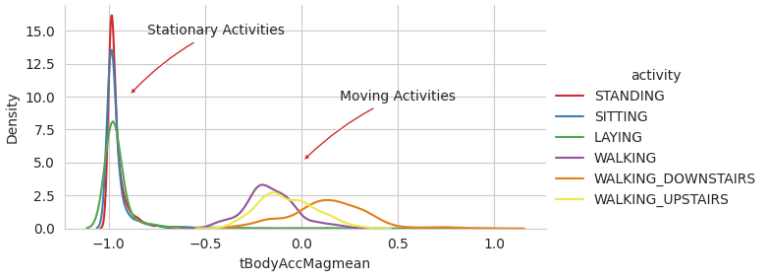

图 4: 不同动作的密度图

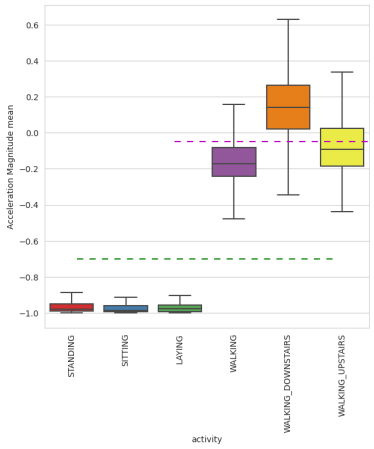

图 5: 不同动作对加速度尺度的影响

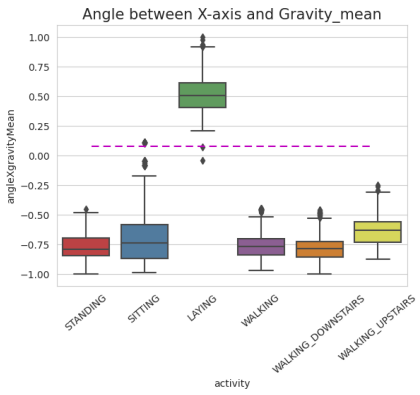

图 6: 不同动作对 x 轴和水平面夹角的影响

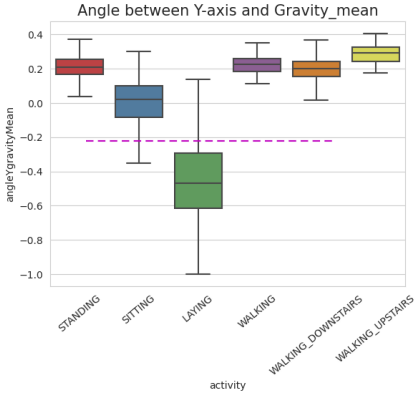

图 7: 不同动作对 y 轴和水平面夹角的影响

利用 t-sne 对数据进行降维，可以得出除了 standing 和 sitting，其它的动作都比较好区分。模型可能在这两个变量上表现欠佳。

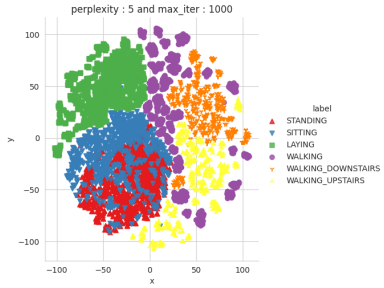

图 8: 困惑度为 5 的 t-sne 数据可视化

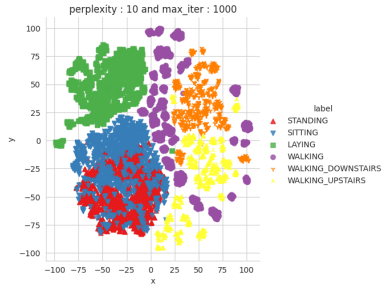

图 9: 困惑度为 10 的 t-sne 数据可视化

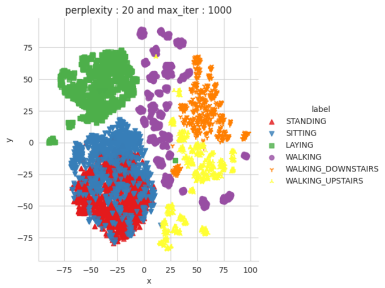

图 10: 困惑度为 20 的 t-sne 数据可视化

应用 PCA 方法对训练集和测试集进行可视化：

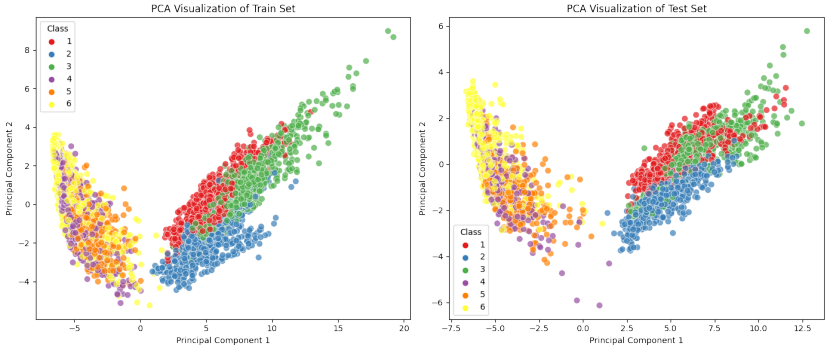

图 11: PCA 可视化

3 聚类实验

在本次实验中我们对比了凝聚层次聚类、BIRCH、KMeans、DBSCAN、STING 共 5 种聚类方法，并手动实现了 KMeans 进行对比。

3.1 层次聚类方法

层次聚类（Hierarchical Clustering）通过逐步合并或分离数据点来构建数据的层次结构，实现聚类的效果。本次实验中，我们选取了凝聚层次聚类和 BIRCH 两种层次聚类算法。

3.1.1 凝聚层次聚类

凝聚层次聚类（Agglomerative Hierarchical Clustering）的原理是从每个数据点作为单独的簇开始，不断合并最近的簇，直到所有数据点合并为一个簇，形成层次结构。它通常简单直观，适用于小数据集，但是难以处理大规模数据集，计算复杂度高。

本实验中，我们使用 sklearn 库中的 AgglomerativeClustering 类实现凝聚层次聚类。由于该模型并非参数化模型，因此我们将训练集和测试集合并起来进行聚类分析。通过对 n_component, affinity, linkage 等

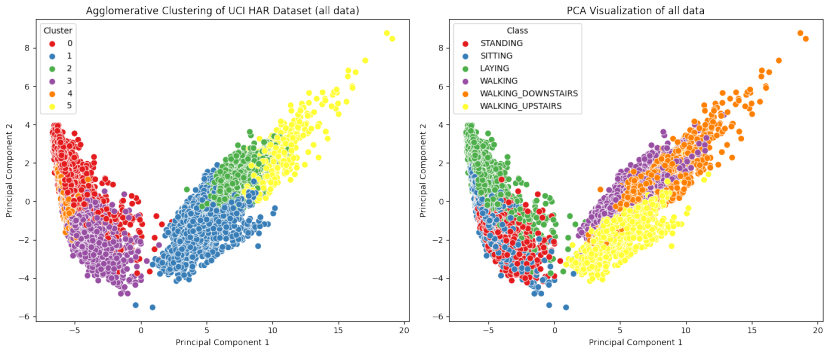

图 12: Agglomerative Hierarchical Clustering 聚类结果可视化

参数进行网格搜索，实现的最佳模型效果如 12所示。从和 ground truth 的对比中可以看出，模型可以较好地实现动态动作和静态动作的区分上，但在两类动作内部的区分上效果欠佳。在动态动作中，模型将一部分的 WALKING 划分到了 WALKING UPSTAIRS 中。

```
from sklearn.cluster import AgglomerativeClustering
from sklearn.decomposition import PCA
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler

# 定义参数网格
param_grid = {
    'standard_scaler': [True, False],
    'pca__n_components': [0.8, 0.9, 0.99, False],
    'agglomerative__affinity': ['euclidean', 'manhattan', 'cosine'],
    'agglomerative__linkage': ['ward', 'complete', 'average', 'single']
}

best_score = -1
best_params = {}

# 循环遍历参数网格
for standard_scaler in param_grid['standard_scaler']:
    for n_components in param_grid['pca__n_components']:
        for affinity in param_grid['agglomerative__affinity']:
            for linkage in param_grid['agglomerative__linkage']:
                # 跳过不兼容的参数组合
                if linkage == 'ward' and affinity != 'euclidean':
                    continue

                if standard_scaler and n_components:
                    pp = Pipeline([
                        ('sc', StandardScaler()),
                        ('pca', PCA(n_components=n_components)),
                    ])
                elif not standard_scaler and n_components:
```

```
                    pp = PCA(n_components=n_components)
                elif not n_components:
                    pp = None
                if pp:
                    X_pca = pp.fit_transform(X_all)
                else:
                    X_pca = X_all
                model = AgglomerativeClustering(n_clusters=6, affinity=affinity,
                    linkage=linkage)

                # 拟合模型并预测
                labels = model.fit_predict(X_pca)
                # 计算 adjusted rand score
                score = adjusted_rand_score(y_all, labels)

                # 更新最佳参数和评分
                if score > best_score:
                    best_score = score
                    best_params = {
                        'standard_scaler': standard_scaler,
                        'pca__n_components': n_components,
                        'agglomerative__affinity': affinity,
                        'agglomerative__linkage': linkage
                    }

# 输出最佳参数和评分
print("Best parameters found: ", best_params)
print("Best adjusted rand score: ", best_score)
```

3.1.2 BIRCH 聚类

BIRCH 的原理是通过构建一个树状数据结构 (称为 CF 树，Clustering Feature 来逐步聚类数据。CF 树由多个节点组成，每个节点包含一个簇的摘要信 ree) 息。BIRCH 通过一次扫描数据集即可找到一个初步的聚类结果，并通过少量额外扫描进一步优化，具有较高的时间效率。但是 BIRCH 仅适用于度量数值数据，不适用于非数值序可能导致不同的聚类结果。对数据点的插入顺序较为敏感，且性能和聚类结果依赖于参数的选择，如值和簇的数量。

本实验中，我们使用 sklearn 库中的 Birch 类实现 birch 聚类。由于该模型属于参数化模型，因此我们使用训练集进行模型拟合，用测试集进行预测分析。通过对 n_component, affinity, linkage 等参数进行网格搜索，实现的最佳模型效果如 12所示。从和 ground truth 的对比中可以看出，和凝聚层次聚类算法类似，模型同样可以较好地实现动态动作和静态动作的区分上，但在两类动作内部的区分上效果欠佳。

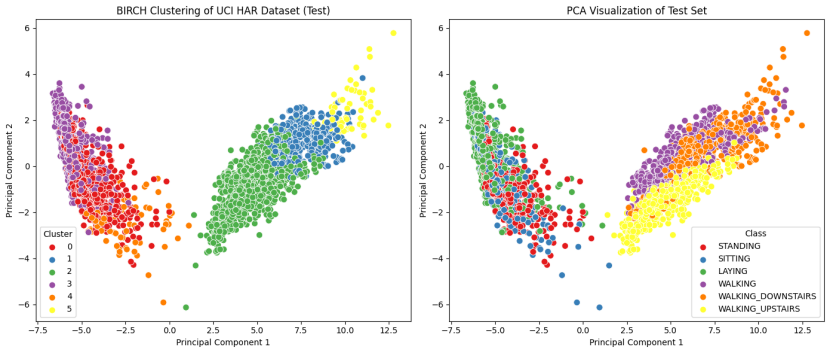

图 13: BIRCH 聚类结果可视化

3.2 密度聚类方法

3.2.1 DBSCAN

该算法的原理是定义一个点的邻域范围 e 和最小点数 min samples，如果一个点的邻域范围内点的数量大于等于 minsamples，则该点被认为是核心点，并与其邻域范围内的点形成一个簇。非核心点若在某核心点的邻域范围内，则归属于该簇; 否则标记为噪声点。该方法能发现任意形状的簇，对噪声点具有鲁棒性。但是需要调节超参数，超参数在不同数据集上较难优化。

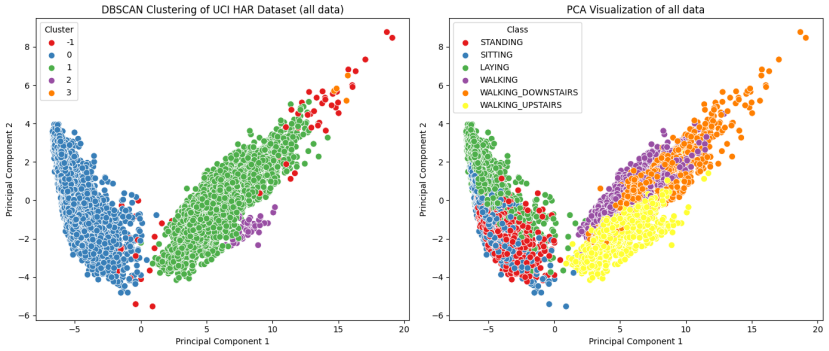

图 14: DBSCAN 聚类结果可视化

本实验中，我们使用 pyclustering 库中的 dbscan 类实现 DBSCAN 聚类。由于该模型属于非参数化模型，因此我们使用所有数据进行模型聚类分析。通过对 n_component, eps, neighbors 等参数进行网格搜索，实现的最佳模型效果如 12所示。由于 DBSCAN 模型无法指定聚类的簇数，因此聚类结果很难和实际样本对应。从和 ground truth 的对比中可以看出，DBSCAN 模型只能粗略地区分动态动作和静态动作。对于离群点，模型都识别成了噪声。

```
# 测算最佳参数模型的运行时间
from sklearn import neighbors
from sklearn.pipeline import Pipeline
from pyclustering.cluster.dbscan import dbscan
from sklearn.preprocessing import StandardScaler
from sklearn.decomposition import PCA

'''
```

```
Best parameters:
{
'standard_scaler': False,
'pca__n_components': 0.8,
'dbscan_eps': 2.0,
'dbscan_neighbors': 3
}
'''

pp = Pipeline([
    ('pca', PCA(n_components=0.8))
])
X_pca = pp.fit_transform(X_all)

n_repetition = 10
run_times = []
for _ in range(n_repetition):
    best_model = dbscan(X_pca.tolist(), eps = 2.0, neighbors = 3)

    start_time = time.time()
    best_model.process()
    run_times.append(time.time() - start_time)

    clusters = best_model.get_clusters()
    y_pred = np.full(X_pca.shape[0], -1)
    for cluster_id, cluster in enumerate(clusters):
        y_pred[cluster] = cluster_id

print(f"Average run time: {np.mean(run_times):.2f} ± {np.std(run_times):.2f} seconds")
compactness, separation, SC, acc, ARI = print_metrics(X_pca, y_pred, y_all)
```

3.3 划分方法

3.3.1 KMeans

K-Means 算法目标是最小化簇内误差平方和（Within-Cluster Sum of Squares, WCSS）。K-Means 算法的具体步骤如下：

.1. 选择初始质心：随机选择 K 个数据点作为初始的簇中心。

.2. 分配数据点：对于每个数据点，根据其到各簇中心的距离，将其分配到最近的簇。

.3. 更新质心：计算每个簇中所有点的均值，将该均值作为新的簇中心。

.4. 重复：重复步骤 2 和 3，直到簇中心不再发生变化或变化非常小，或者达到预设的迭代次数。

手动实现的 KMeans 类代码如下：

```
class KMeansManual:
```

```
    def __init__(self, n_clusters, max_iter=300, tol=1e-4, init='k-means++',
        random_state=None):
        self.n_clusters = n_clusters
        self.max_iter = max_iter
        self.tol = tol
        self.init = init
        self.random_state = random_state
        self.cluster_centers_ = None
        self.labels_ = None

    def _init_centroids(self, data):
        np.random.seed(self.random_state)
        if self.init == 'random':
            indices = np.random.choice(len(data), self.n_clusters, replace=False)
            return data[indices]
        elif self.init == 'k-means++':
            centroids = [data[np.random.choice(len(data))]]
            for _ in range(1, self.n_clusters):
                distances = np.min(pairwise_distances(data, centroids), axis=1)
                probabilities = distances / distances.sum()
                next_centroid = data[np.random.choice(len(data), p=probabilities)]
                centroids.append(next_centroid)
            return np.array(centroids)
        else:
            raise ValueError(f"Unknown init method: {self.init}")

    def fit(self, data):
        self.cluster_centers_ = self._init_centroids(data)
        for i in range(self.max_iter):
            distances = pairwise_distances(data, self.cluster_centers_)
            labels = np.argmin(distances, axis=1)
            new_centroids = np.array([data[labels == j].mean(axis=0) for j in
                range(self.n_clusters)])
            if np.linalg.norm(new_centroids - self.cluster_centers_) < self.tol:
                break
            self.cluster_centers_ = new_centroids
        self.labels_ = labels
        return self

    def predict(self, data):
        distances = pairwise_distances(data, self.cluster_centers_)
        return np.argmin(distances, axis=1)
```

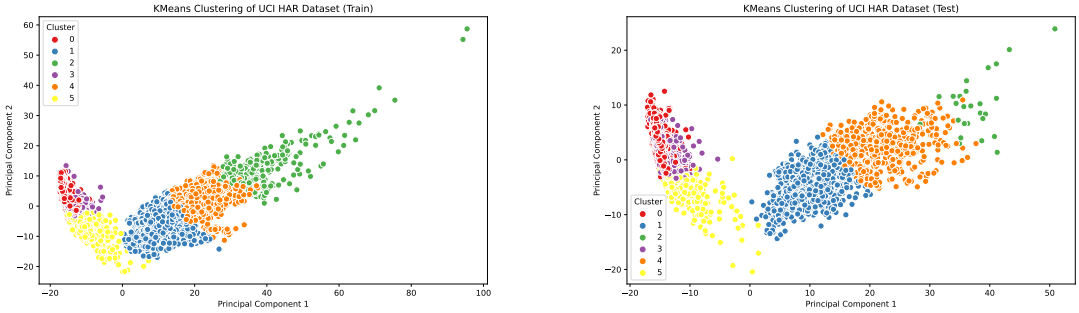

图 15: KMeans 聚类结果可视化

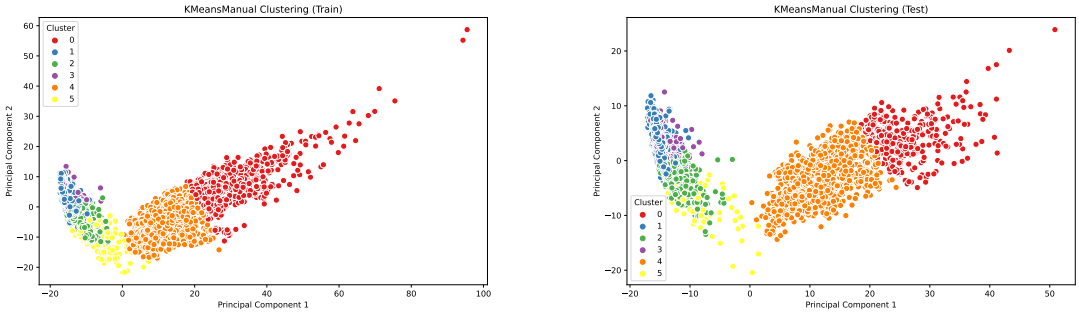

图 16: KMeans（手动实现）聚类结果可视化

3.4 基于网格的聚类方法

3.4.1 STING

STING（Statistical Information Grid）聚类算法是一种基于网格的空间数据聚类方法，通过将数据空间划分为多层次的网格结构，逐步细化地分析数据分布特性。算法首先对数据进行分层次划分，每一层的网格单元包含对应区域内的数据统计信息，如均值、标准差和数据点数量。在聚类过程中，STING 通过自顶向下或自底向上的方式递归处理网格，利用统计信息对网格进行过滤或合并，从而提高算法的效率和可扩展性。由于 STING 无需扫描所有数据点，且充分利用了网格统计特性，其计算效率高，适合处理大规模数据，但对聚类结果的精度可能受网格划分粒度影响。

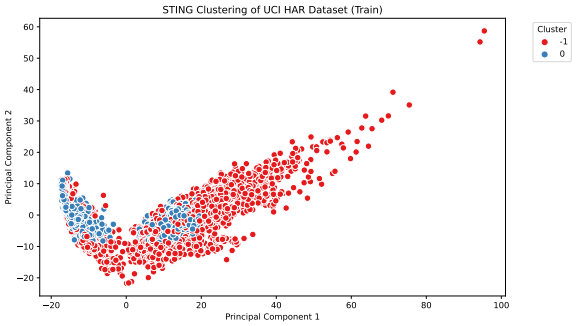

图 17: 训练集上 STING 可视化

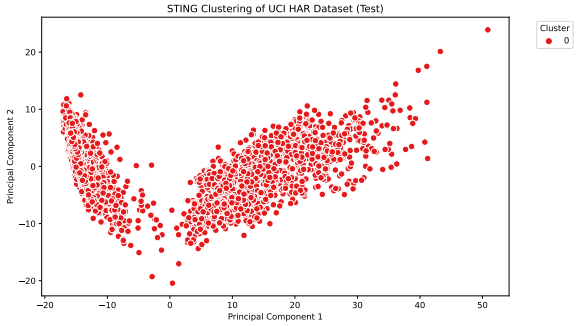

图 18: 测试集上 STING 可视化

3.5 实验结果

下面的表格汇总了所有方法的结果，除了使用如 accuracy, adjusted_rand_index 等外在指标对聚类结果进行评估外，我们还使用了 compactness, separation, silhouette_score 等内在指标对聚类结果进行评估。对于层次聚类与密度聚类方法，我们对方法的超参数进行网格搜索；对于划分方法以及网格聚类方法，我们使用 Optuna 进行调参。

其中 Accuracy 指标的计算方式是将预测的类别和实际类别利用匈牙利算法找到最佳指派，然后将标签对应后计算准确率。DBSCAN 由于无法生成 6 个簇，因此无法计算准确率。

			内在指标			外在指标		
	算法	split	compactness	separation	silhouette_score	accuracy	adjusted_rand_score	运行时间（s）
层次聚类	Agglo	all data	2.631	9.574	0.210	0.612	0.514	1.85 ± 0.11
	BIRCH	test	2.442	9.005	0.279	0.598	0.522	0.87 ± 0.04
密度聚类	DBSCAN	all data	3.349	11.153	0.497	NA	0.330	2.72 ± 0.04
划分	KMeans	train	8.913	9.065	0.245	0.548	0.429	2.4577 ± 0.4965
		test	8.403	9.065	0.265	0.565	0.447	
	KMeans-Manual	train	11.871	9.150	0.185	0.546	0.410	1.4348 ± 0.0359
		test	11.292	9.150	0.185	0.576	0.422	
网格	STING	all data	12.413	9.795	0.200	0.270	0.077	1.795 ± 0.1532

4 总结

实验对比了凝聚层次聚类、BIRCH、KMeans、DBSCAN 和 STING 五种聚类方法，结果表明不同方法在动态动作和静态动作的区分上整体表现良好，但在动作内部类别的细分上差异明显。层次聚类方法中，凝

聚层次聚类能够较好地区分动态与静态动作，但在细分类别上存在一定混淆；BIRCH 则在时间效率上优于凝聚层次聚类，但分类效果与前者相当。划分方法中，KMeans 在性能稳定性和分类结果上表现优异，但对初始质心的选择较为敏感；手动实现的 KMeans 进一步提高了运行效率，尽管分类精度略低于库函数实现。密度聚类方法 DBSCAN 能够发现任意形状的簇，对噪声数据处理具有优势，但其无法生成固定数量的类别，限制了与实际样本的对比分析。网格方法 STING 则依赖网格划分的粒度，在运行效率上表现突出，但聚类精度较低。总体来看，凝聚层次聚类和 KMeans 在类别区分上的表现较优，而 DBSCAN 和 STING 在特殊应用场景下具备一定的实用价值。