电子科技大学

实验报告

课程名称：数据分析综合课程设计

学　　院：计算机科学与工程学院

专　　业：计算机科学与技术（互联网+）

指导教师：陈端兵、蔡世民

学生姓名：蔡与望

学　　号：2020010801024

电 子 科 技 大 学

实 验 报 告

实验一

一、实验项目名称：决策树与随机森林

二、实验学时：4

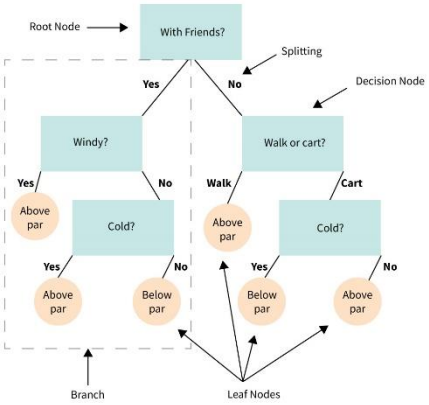
三、实验目的

掌握决策树与随机森林的原理和代码实现。

四、实验原理

4.1 决策树

决策树是一类常见的机器学习方法，核心思路是从给定训练数据集中，学得一个树形模型，用以对新示例进行分类。



每次划分的期望都是，节点的纯度越来越高。也因此，不同的纯度衡量指标，引出了不同的决策树构造算法，例如 ID3 算法以信息增益为基础，CART 算法以基尼系数为基础。本次实验中将采用后者，即 CART 算法和基尼系数，来构造决策树。

基尼系数代表了一个数据集的“混乱程度”。基尼系数越高，说明对应的数据集越混乱；反之则越整齐。它的公式如下：

$$Gini(S) = 1 - \sum_{i=1}^{m} \left(\frac{|S_i|}{|S|} \right)^2$$

$$Gini(S, A) = \sum_{v=1}^{V} \frac{|S_v|}{|S|} Gini(S_v)$$

例如，计数[3,3,4]的基尼系数为 1-0.3² -0.3² -0.4² = 0.66，说明这组计数背后的数据集很混乱。而计数[0,0,10]的基尼系数为 1-0² -0² -1² =0，说明这组计数背后的数据集很整齐。而[3,3,4]和[0,0,10]这两组计数的加权平均基尼系数为 0.5*0.66+0.5*0=0.33。

在划分离散型特征时，该特征的所有可取的类别中，存在一个“最佳类别”。与基于其它“非最佳类别”的划分相比，基于“最佳类别”的划分可以使得两个子集的加权平均基尼系数最小。

在划分连续性特征时，为了将连续的特征离散化，我们取该特征下所有相邻值的二分点，作为若干个可能的阈值。而该特征的所有可能的阈值中，存在一个“最佳阈值”。与基于其它“非最佳阈值”的划分相比，基于“最佳阈值”的划分可以使得两个子集的加权平均基尼系数最小。

综合起来看，在划分数据集时，该数据集的所有特征中，存在一个“最佳特征”。与基于其它“非最佳特征”的划分相比，基于“最佳特征”的划分可以使得两个子集的加权平均基尼系数最小。决策树就是这样，不断地依照最佳特征来划分数据集，直到划分出纯数据集，或达到指定停止条件。

4.2 随机森林

随机森林的思想是：对 N 个样本有放回地抽样，抽出 N 个样本作为新的训练集，从 M 个特征中无放回地取出 m 个特征，训练出一棵决策树。如此重复 k 次，得到 k 个数据和特征均不相同的训练集。从而，有 k 棵不同的决策树，最后对这 k 棵决策树预测结果取均值。

随机森林在计算量没有显著提高的前提下，提高了预测精度。它对多元线性不敏感，对缺失数据和非平衡数据更稳健，并且可预测多达几千个解释变量。

五、实验内容与要求

1. 使用代码实现决策树和随机森林。
2. 在给定的数据集上验证决策树和随机森林的准确率。

六、实验器材（设备、元器件）：

笔记本电脑。

七、实验步骤

calculate_gini 函数计算一组计数的基尼系数。

```
def calculate_gini(counts: list[int]):
    total = sum(counts)

    if total == 0:
        return 0
```

```
    return 1 - sum((count / total) ** 2 for count in counts)
```

calculate_weighted_gini 函数计算若干组计数的加权平均基尼系数，其中每组计数的权重为：该组计数的总数 / 所有计数的总数。

```
def calculate_weighted_gini(counts_list: list[list[int]]):
    total = sum(sum(counts) for counts in counts_list)
    return sum((sum(counts) / total) * calculate_gini(counts) for counts in counts_list)
```

divide_by_discrete_feature 函数基于指定的离散型特征，将数据集划分为两个子集。以成人数据集为例：基于“学历是否为本科”，可以将数据集划分为两个子集。

```
def divide_by_discrete_feature(dataset: pd.DataFrame, feature: str):
    # 统计出所有可取的类别。
    choices = dataset[feature].unique()

    # 对于每个类别，都计算一遍划分后的加权平均基尼系数，然后找到使得加权平均基尼系数最小的最佳类别。
    best_choice = None
    best_yes_subset = None
    best_no_subset = None
    min_gini = 1.0
    for choice in choices:
        # 根据“特征是否取该类别”划分出两个子集。
        yes_subset: pd.DataFrame = dataset[dataset[feature] == choice]
        no_subset: pd.DataFrame = dataset[dataset[feature] != choice]

        # 分别统计两个子集中，各标签的数量。
        yes_counts = yes_subset.iloc[:, -1].value_counts()
        no_counts = no_subset.iloc[:, -1].value_counts()

        # 计算当前划分下的加权平均基尼系数。
        gini = calculate_weighted_gini([yes_counts, no_counts])

        # 更新最佳类别和最小系数。
        if gini < min_gini:
            best_choice = choice
            best_yes_subset = yes_subset
            best_no_subset = no_subset
            min_gini = gini

    return best_choice, best_yes_subset, best_no_subset, min_gini
```

divide_by_continuous_feature 函数基于指定的连续型特征，将数据集划分为两个子集。以成人数据集为例：基于“年龄是否小于 30.5 岁”，可以将数据集划分为两个子集。

```
def divide_by_continuous_feature(dataset: pd.DataFrame, feature: str):
    # 计算出所有可能的阈值。
    unique_values = dataset[feature].unique()
    thresholds = (unique_values[:-1] + unique_values[1:]) / 2

    # 对于每个阈值，都计算一遍划分后的加权平均基尼系数，然后找到使得加权平均基尼系数最小的最佳阈值。
    best_threshold = None
    best_less_subset = None
    best_greater_subset = None
    min_gini = 1.0
    for threshold in thresholds:
        # 根据“特征是否小于该阈值”划分出两个子集。
        less_subset: pd.DataFrame = dataset[dataset[feature] < threshold]
        greater_subset: pd.DataFrame = dataset[dataset[feature] > threshold]

        # 分别统计两个子集中，各标签的数量。
        less_counts = less_subset.iloc[:, -1].value_counts()
        greater_counts = greater_subset.iloc[:, -1].value_counts()

        # 计算当前划分下的加权平均基尼系数。
        gini = calculate_weighted_gini([less_counts, greater_counts])

        # 更新最佳阈值和最小系数。
        if gini < min_gini:
            best_threshold = threshold
```

```
            best_less_subset = less_subset
            best_greater_subset = greater_subset
            min_gini = gini

    return best_threshold, best_less_subset, best_greater_subset, min_gini
```

divide 函数将数据集划分为两个子集。

```
def divide(dataset: pd.DataFrame):
    # 统计出所有可能的特征。
    features = dataset.columns[:-1]

    # 对于每个特征，都计算一遍划分后的加权平均基尼系数，然后找到使得加权平均基尼系数最小的最佳特征。
    best_feature = None
    best_choice = None
    best_threshold = None
    best_left_subset = None
    best_right_subset = None
    min_gini = 1.0
    for feature in features:
        # 在标签相同的情况下，如果该特征下只有一个类别或取值，
        # 说明该特征无法继续划分，直接跳过。
        if len(dataset[feature].unique()) == 1:
            continue

        # 如果是离散型特征，则找到最佳类别。
        if dataset[feature].dtype == object:
            choice, left_subset, right_subset, gini = divide_by_discrete_feature(
                dataset, feature
            )
            threshold = None

        # 如果是连续型特征，则找到最佳阈值。
        else:
            (
                threshold,
                left_subset,
                right_subset,
                gini,
            ) = divide_by_continuous_feature(dataset, feature)
            choice = None

        # 更新最佳特征、最佳类别、最佳阈值和最小系数。
        if gini < min_gini:
            best_feature = feature
            best_choice = choice
            best_threshold = threshold
            best_left_subset = left_subset
            best_right_subset = right_subset
            min_gini = gini

    return (
        best_feature,
        best_choice,
        best_threshold,
        best_left_subset,
        best_right_subset,
    )
```

build_decision_tree 函数在指定的数据集上，构建决策树。决策树每个节点的数据结构如下：

- feature：划分数据集的特征。

- choice：如果 feature 是离散型的，那么 choice 是最佳类别；否则为 None。

- threshold：如果 feature 是连续型的，那么 threshold 是最佳阈值；否则为 None。

- left：该节点的左子树。

- right：该节点的右子树。

```
def build_decision_tree(dataset: pd.DataFrame, current_depth: int = 1):
    # 如果数据集中只有一种标签，那么直接返回这个类别。
    if len(dataset.iloc[:, -1].unique()) == 1:
```

```
        return dataset.iloc[:, -1].unique()[0]

    # 找到该数据集的最佳特征与最佳类别（或阈值）。
    feature, choice, threshold, left_subset, right_subset = divide(dataset)

    # 如果找不到最佳特征，说明所有特征都只有一种类别或取值。
    # 用投票法决定该节点的类别。
    if feature is None:
        return dataset.iloc[:, -1].value_counts().index[0]

    # 递归地构建左右子树。
    left_tree = build_decision_tree(left_subset, current_depth + 1)
    right_tree = build_decision_tree(right_subset, current_depth + 1)

    # 返回当前节点。
    return {
        "feature": feature,
        "choice": choice,
        "threshold": threshold,
        "left": left_tree,
        "right": right_tree,
    }
```

sample_dataset 函数使用 Bootstrap 算法，从数据集中有放回地抽取同样多行数据，并随机选取一小部分属性，构成一个新的数据集。

```
def sample_dataset(dataset: pd.DataFrame, num_samples: int):
    num_features = int(np.sqrt(dataset.shape[1]))
    samples: list[pd.DataFrame] = []

    for _ in range(num_samples):
        sample = dataset.sample(dataset.shape[0], replace=True)
        features = np.random.choice(sample.columns[:-1], num_features, replace=False)
        sample = pd.concat([sample[features], sample.iloc[:, -1]], axis=1)
        samples.append(sample)

    return samples
```

build_random_forest 函数在指定的数据集上，构建随机森林。

```
def build_random_forest(dataset: pd.DataFrame, num_trees: int = 30):
    samples = sample_dataset(dataset, num_trees)
    decision_trees = [build_decision_tree(sample) for sample in samples]
    return decision_trees
```

在 adult 数据集上测试决策树与随机森林，分别取得了 83%和 87%的准确率。这说明编写的代码成功实现了这两种算法，并且能够取得不错的精度。其中，随机森林的准确率一般会优于决策树。

```
Decision tree accuracy: 0.83
Random forest accuracy: 0.87
```

八、总结和讨论

通过本次实验，我掌握了基于 CART 算法的决策树，以及将其作为弱学习器的随机森林的实现。

随机森林的核心思想是避免一棵决策树的“一言堂”，让若干棵各异的决策树来共同决策。它能在不显著提高计算量的前提下，提高预测的准确率；所以在日后遇到类似问题时，如果能用决策树解决，那么不妨也同时试一试随机森林。当然，如果需要解释的因子数量很少，比如只有 5 个甚至更少，那么随机森林的优势也并不明显。

电 子 科 技 大 学

实 验 报 告

实验二

一、实验项目名称：节点重要性排序与评估

二、实验学时：4

三、实验目的

掌握节点重要性排序的评估方法和算法实现。

四、实验原理

4.1 SIR 传播过程

SIR 传播模型常用于描绘疾病在人群中传播的过程，或是进行对类似过程的仿真。其基本模型为：

$$S \xrightarrow{\mu} I \xrightarrow{\beta} R$$

在一个网络中，存在一些源感染者。每一轮传播中，每个感染者以概率μ感染其未感染过的邻居；同时，也以概率β康复。当网络中的所有节点都康复时，传播过程结束。

由于 SIR 传播过程是对现实过程的仿真，其通常能作为衡量一种节点重要性排序算法的真实性的标准。也即，以一个人为感染源，当传播结束时，被感染后又康复的人越多，说明此人在网络中越重要。

4.2 相关性计算

两个序列的相关性可以通过下式计算。

$$\tau = \frac{2(n_+ - n_-)}{n(n-1)}$$

相关性系数越高，说明两个序列越相关；反之，则越不相关。

4.3 k-shell 分解法

k-shell 分解法通过不断剥去一个网络中最外围的节点，给所有节点分层。其步骤为：找到度数最小的节点，将它们的度数作为其重要性分数，然后将这些节点从网络中移除。如此重复，直至所有节点都被移除。

4.4 概率模型

概率模型利用传播的快速衰减特性，从一个节点出发，将信息以一定的概率传播到邻居，得到邻居感染的可能性，再从邻居出发向外以一定的概率传播，3 层后终止。最后，将所有 3 层邻居的感染分数相加得到此节点的重要性分数。

$$uninf(p,i) = \prod_{q \in \Gamma_{i-1}(u)} [1 - score(q, i-1) * \beta]$$

$$score(p,i) = 1 - uninf(p,i)$$

$$Rank(u) = \sum_{i=1}^{3} \sum_{v \in \Gamma_i(u)} score(v,i)$$

五、实验内容与要求

1. 使用代码实现 SIR 传播过程及相关性系数计算过程，并给出两个小规模的示例。
2. 使用代码实现节点重要性排序算法，至少包括 k-shell 和概率模型。
3. 给出 5 个网络节点排序结果，用表格列出每个网络前 10 的节点及分数，并计算和 SIR 仿真结果的相关性，并给出相关性列表。

六、实验器材（设备、元器件）：

笔记本电脑。

七、实验步骤

simulate_SIR 函数模拟 SIR 传播过程，并通过 visualize_SIR 函数进行可视化。

```
def simulate_SIR(
    network: Network, source: int = 0, beta: float = 0.3, gamma: float = 0.1
):
    # 初始化各节点的状态。
    susceptible_nodes = set(range(network.shape[0])) - {source}
    infected_nodes = {source}
    recovered_nodes = set()

    # 记录各状态在各时刻上的节点数量。
    susceptible_counts = [len(susceptible_nodes)]
    infected_counts = [len(infected_nodes)]
    recovered_counts = [len(recovered_nodes)]

    # 如果还有感染者，SIR 传播过程就会继续。
    while infected_nodes:
        # 记录本轮新感染和新康复的节点。
        new_infected_nodes = set()
        new_recovered_nodes = set()

        # 对于每个感染者来说：
        for infected_node in infected_nodes:
            # 他会以概率 β 感染其邻居。
            neighbors = find_neighbors(network, infected_node)
            susceptible_neighbors = set(neighbors) & susceptible_nodes
            new_infected_neighbors = {
```

```
                susceptible_neighbor
                for susceptible_neighbor in susceptible_neighbors
                if np.random.rand() < beta
            }
            new_infected_nodes |= new_infected_neighbors

            # 他会以概率 γ 康复。
            if np.random.rand() < gamma:
                new_recovered_nodes.add(infected_node)

        # 更新各节点状态。
        susceptible_nodes -= new_infected_nodes
        infected_nodes = (infected_nodes | new_infected_nodes) - new_recovered_nodes
        recovered_nodes |= new_recovered_nodes

        # 更新各状态在当前时刻上的节点数量。
        susceptible_counts.append(len(susceptible_nodes))
        infected_counts.append(len(infected_nodes))
        recovered_counts.append(len(recovered_nodes))

    # 打包计数结果。
    counts = {
        "susceptible": susceptible_counts,
        "infected": infected_counts,
        "recovered": recovered_counts,
    }

return counts

     def visualize_SIR(counts: dict[str, list[int]]):
    plt.stackplot(
        range(len(counts["infected"])),
        counts.values(),
        labels=counts.keys(),
        colors=["blue", "red", "green"],
        alpha=0.6,
    )
    plt.title("SIR Propagation")
    plt.xlabel("Days")
    plt.ylabel("Number")
    plt.margins(x=0, y=0)
    plt.legend()
plt.show()
```

calculate_correlation 函数计算两个序列的相关性系数。

```
def calculate_correlation(sequence1: list[int], sequence2: list[int]):
    # 计算两个长度为 1 的序列的相关性系数没有意义。
    assert len(sequence1) > 1 and len(sequence2) > 1

    # 逐一比较每一个变化对的正相关性和负相关性。
    pairs = list(zip(sequence1, sequence2))
    num_positive = 0
    num_negative = 0

    for index, (x1, y1) in enumerate(pairs[:-1]):
        for x2, y2 in pairs[index + 1 :]:
            if (x1 < x2 and y1 < y2) or (x1 > x2 and y1 > y2):
                num_positive += 1
            elif (x1 < x2 and y1 > y2) or (x1 > x2 and y1 < y2):
                num_negative += 1

    # 导出相关性系数。
    total = np.sum(range(len(sequence1)))
    correlation = np.round((num_positive - num_negative) / total, 3)

    return correlation
```

k_shell_sort 函数使用 k-shell 分解法，对节点进行排序。

```
def k_shell_sort(network: Network):
    # 转换为浮点型，以便在后面用 `inf` 表示已移除的节点。
    network = network.astype(float)
```

```
    # 初始化每个节点的度数。
    degrees = np.asarray([np.sum(row) for row in network])

    # 存储排序后的节点。
    order = []

    # 如果还有节点没被排序，排序过程就会继续。
    while len(order) < network.shape[0]:
        # 找到度数最小的节点。
        min_degree = np.min(degrees)
        min_degree_nodes = np.nonzero(degrees == min_degree)[0]

        # 将这些节点从网络中移除。
        network[:, min_degree_nodes] = 0
        network[min_degree_nodes, :] = np.inf

        # 更新每个节点的度数。
        degrees = np.asarray([np.sum(row) for row in network])

        # 将这些节点添加到排序后的节点中。
        order.extend((node, int(min_degree)) for node in min_degree_nodes)

    return order
```

group_nodes_by_distance 函数将节点按照距离分组。例如，分组结果[[0],[1,2],[3,4]]代表：节点 0 和节点 0 的距离为 0，节点 1、2 和节点 0 的距离为 1，节点 3、4 和节点 0 的距离为 2。

```
def group_nodes_by_distance(network: Network, source: int):
    # 记录各层内的节点。
    layers = defaultdict(list)

    # 从源节点开始 BFS。
    queue = [(source, 0)]
    visited_nodes = {source}

    while queue:
        # 弹出队头节点，并记录其和源节点的距离。
        node, distance = queue.pop(0)
        layers[distance].append(node)

        # 访问该节点的未被访问的邻居，并将它们加入待记录的队列。
        neighbors = find_neighbors(network, node)
        unvisited_neighbors = set(neighbors) - visited_nodes
        queue.extend((neighbor, distance + 1) for neighbor in unvisited_neighbors)
        visited_nodes |= unvisited_neighbors

    # 根据字典键大小顺序，转换为嵌套列表。
    layers = [layers[distance] for distance in sorted(layers.keys())]

    return layers
```

calculate_probabilistic_importance 函数计算概率模型下，一个节点的重要性分数，也即各节点被其感染的感染分数之和。

```
def calculate_probabilistic_importance(network: Network, source: int, beta: float):
    # 记录各节点的感染分数。
    scores = np.zeros(network.shape[0])
    scores[source] = 1

    # 按照到源节点的距离分层。
    layers = group_nodes_by_distance(network, source)

    # 在 1-3 层上，从近到远，计算每个节点的感染分数。
    for layer_index, layer in enumerate(layers[1:4]):
        for node in layer:
            # 找到该节点处于前一层的邻居。
            neighbors = find_neighbors(network, node)
            influencers = np.intersect1d(neighbors, layers[layer_index])

            # 根据这些邻居的感染分数，计算该节点的感染分数。
```

```
            scores[node] = 1 - np.prod(1 - scores[influencers] * beta)

    # 导出重要性分数。
    importance = np.round(np.sum(scores) - 1, 3)

    return importance
```

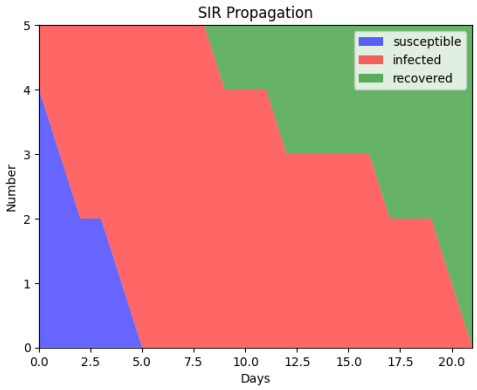
probabilistic_sort 函数使用概率模型，对节点进行排序。

```
def probabilistic_sort(network: Network, beta: float = 0.3):
    with_importance_nodes = [
        (node, calculate_probabilistic_importance(network, node, beta))
        for node in range(network.shape[0])
    ]
    order = sorted(with_importance_nodes, key=lambda pair: pair[1])
    return order
```

SIR_sort 函数使用 SIR 模型，对节点进行排序。每个节点的分数是：以它为感染源时，最终康复的节点数。

```
def SIR_sort(network: Network, beta: float = 0.3, gamma=0.1):
    with_importance_nodes = [
        (node, simulate_SIR(network, node, beta, gamma)["recovered"][-1])
        for node in range(network.shape[0])
    ]
    order = sorted(with_importance_nodes, key=lambda pair: pair[1])
    return order
```

SIR 传播过程的可视化如下图所示。横轴是时间，纵轴是节点数量，蓝色部分代表易感节点，红色部分代表感染节点，绿色部分代表康复节点。



对于自行构造的 5 个网络分别进行 SIR、k-shell 和概率模型排序，并计算后两者与 SIR 排序的相关性，如下各图所示。

```
Network #1:

  SIR K-shell Probabilistic
0   5       2         0.518
1   5       2         0.678
2   5       2           0.8
3   5       2         0.678
4   5       2         0.518

SIR ~ K-shell correlation: 0.4
SIR ~ Probabilistic correlation: 0.2
```

```
Network #2:

  SIR K-shell Probabilistic
0   6       2         0.542
1   6       2         0.718
2   6       1         0.878
3   6       1         0.878
4   6       2         0.718
5   6       2         0.542

SIR ~ K-shell correlation: 0.2
SIR ~ Probabilistic correlation: 0.2
```

```
Network #3:

  SIR K-shell Probabilistic
0   7       3         0.688
1   7       3         0.688
2   6       3          1.04
3   5       3         0.688
4   7       2          0.72
5   7       2          0.56
6   7       1         0.304

SIR ~ K-shell correlation: -0.048
SIR ~ Probabilistic correlation: 0.238
```

```
Network #4:

  SIR K-shell Probabilistic
0   8       3         0.766
1   8       3         0.766
2   8       3          1.24
3   8       3         0.888
4   8       2          0.76
5   8       2           0.6
6   8       1         0.312
7   7       2         0.645

SIR ~ K-shell correlation: 0.071
SIR ~ Probabilistic correlation: -0.143
```

```
Network #5:

  SIR K-shell Probabilistic
0   9       2         1.306
1   9       3         1.417
2   9       2         1.306
3   9       3         1.452
4   1       3         1.414
5   9       4          1.26
6   9       2         1.234
7   9       4         0.877
8   9       3         1.414

SIR ~ K-shell correlation: 0.056
SIR ~ Probabilistic correlation: -0.056
```

可以看出，k-shell 和概率模型对节点重要性的排序，与 SIR 传播中的重要性排序大致相关。对于某些相关性不明显的，猜测是因为网络规模过小，导致 SIR 排序对于各节点都给出了相同的重要性。

八、总结和讨论

通过本次实验，我掌握了 SIR 传播过程的概念、模型和代码实现，同时掌握了使用 k-shell 和概率模型对节点进行排序的方法。由于 SIR 是对于现实世界的抽象简化仿真，其可以作为重要性的“标准答案”，以此衡量各种节点排序算法的准确性和真实性。

遗憾的是，本次实验中的各个网络，在经过数十次的反复试验后，效果均不甚令人满意。可以考虑扩大网络规模，可以预见这样能够提高相关性。

同时，概率模型相比 k-shell 来说，有着更高的准确性和信服度，因为 k-shell 经常会对处于不同环境下的节点给出相同的排序重要性，而实际上这些节点在整个网络中的重要性不尽相同。

电 子 科 技 大 学

实 验 报 告

实验三

一、实验项目名称：聚类

二、实验学时：4

三、实验目的

掌握几种基本的聚类算法的原理和代码实现。

四、实验原理

4.1 K-means

K-means 是一种经典而简单的聚类算法，基于质心和欧氏距离的概念。

它的算法步骤如下：

1. 随机选择 K 个数据点作为初始质心；
2. 对于每个数据点，计算其与每个质心之间的欧氏距离，将其分配到最近的质心所代表的簇中；
3. 更新每个簇的质心为该簇内所有数据点的均值；
4. 重复步骤 2 和 3，直到质心的位置不再发生显著变化或达到最大迭代次数。

这种算法简单且易于实现，计算效率高，对于大型数据集也能够有效处理，尤其适用于数据点具有明显的球状分布的情况。

然而，K-means 也有一些缺点：需要预先指定簇的数量 K，该参数的选择可能会影响聚类结果；对于非球状分布、不同大小和密度的簇，效果可能不佳；对于具有噪声点和异常值的数据集，K-means 可能会将其错误地分配到某个簇中。

4.2 DBSCAN

DBSCAN 是一种密度聚类算法，基于密度可达性和密度连接的概念。

给定一个数据集，DBSCAN 算法将每个数据点分为三类：核心点、边界点和噪声点。核心点是在给定半径ε内具有至少 MinPts 个邻居的点，它们位于一个簇的中心。边界点是在给定半径ε内具有少于 MinPts 个邻居的点，但是它们位于核心点的邻域中。噪声点是既不是核心点也不是边界点的点。

它的算法步骤如下：

1. 随机选择一个未访问的数据点；
2. 如果该点是一个核心点，则创建一个新簇，并将该点及其密度可达的所有点添加到簇中；
3. 重复步骤 2，直到无法找到更多密度可达的点；
4. 如果当前点是一个边界点而不是核心点，则将其标记为噪声点；
5. 重复步骤 1-4，直到所有的数据点都被访问。

与 K-Means 相比，DBSCAN 算法不需要预先指定簇的数量，能够自动发现任意形状的簇；同时，还能够识别噪声点，并将其标记为噪声簇，从而过滤掉异常值；对于具有不同形状、大小和密度的簇具有较好的健壮性。

然而，DBSCAN 也有一些缺点：对于具有不同密度的簇，参数选择可能变得困难；对于高维数据，由于所谓的“维度灾难”，DBSCAN 的性能可能下降；对于具有不同密度的簇，可能需要调整参数ε和 MinPts 的值来获得最佳结果。

五、实验内容与要求

1. 在金融数据集上，进行数据预处理和聚类建模。
2. 分析客户画像，形成分析结果。

六、实验器材（设备、元器件）：

笔记本电脑。

七、实验步骤

k_means 函数使用 K-Means 算法进行聚类。

```
def k_means(dataset: pd.DataFrame, num_clusters: int = 3):
    # 随机挑选若干个点，作为聚类中心。
    centers = dataset.sample(num_clusters)
    centers.index = range(num_clusters)

    # 存储聚类结果。
    cluster_assignments = pd.Series(index=dataset.index)

    # 不断更新聚类中心，直至收敛。
    while True:
        # 将每个点分配到最近的聚类中心。
        new_cluster_assignments = dataset.apply(
            lambda point: (centers - point).pow(2).sum(axis=1).pow(0.5).idxmin(),
            axis=1,
        )

        # 如果与上一次的聚类结果相同，说明聚类已经收敛。
        if new_cluster_assignments.equals(cluster_assignments):
            break

        # 否则，更新聚类结果。
        cluster_assignments = new_cluster_assignments

        # 根据新的聚类结果，计算新的聚类中心。
        centers = dataset.groupby(cluster_assignments).mean()
```

```
return cluster_assignments
```

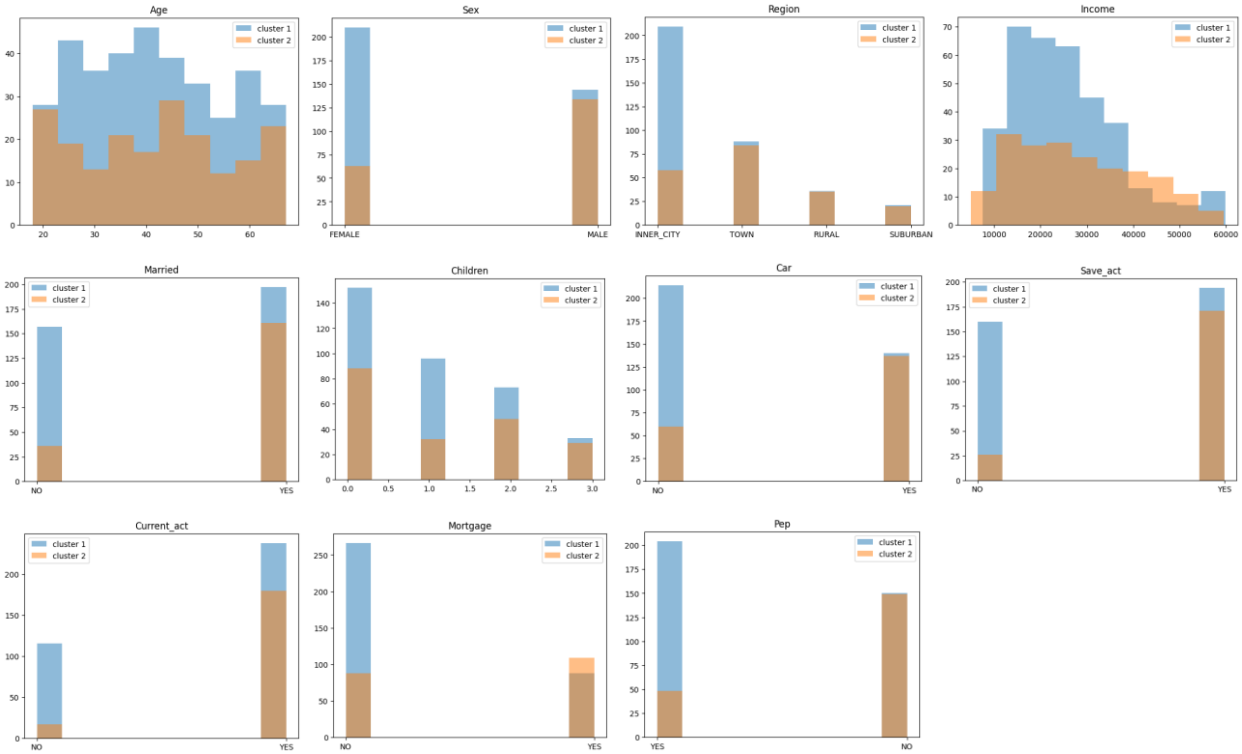
dbscan 函数使用 DBSCAN 算法进行聚类。

```
def dbscan(dataset: pd.DataFrame, eps: float = 10, min_samples: int = 3):
    # 将所有点标记为未访问。
    visited_points = pd.Series(False, index=dataset.index)

    # 存储聚类结果。
    cluster_assignments = pd.Series(0, index=dataset.index, dtype=int)

    # 记录当前聚类的编号。
    cluster_id = 0

    # 计算每个点的邻域。
    neighborhoods = dataset.apply(
        lambda point: (dataset - point).pow(2).sum(axis=1).pow(0.5) <= eps,
        axis=1,
    )

    # 从第一个点开始，依次访问每个点。
    for point_id in dataset.index:
        # 如果该点已经被访问过，则跳过。
        if visited_points[point_id]:
            continue

        # 将该点标记为已访问。
        visited_points[point_id] = True

        # 计算该点的邻域。
        neighborhood = neighborhoods[point_id]

        # 如果该点的邻域中的点的数量小于 min_samples，将该点标记为噪声。
        if neighborhood.sum() < min_samples:
            cluster_assignments[point_id] = -1
            continue

        # 否则，将该点加入一个新的聚类。
        cluster_id += 1
        cluster_assignments[point_id] = cluster_id

        # 依次访问该点的邻域中的点。
        for neighbor_id in neighborhood[neighborhood].index:
            # 如果该点已经被访问过，则跳过。
            if visited_points[neighbor_id]:
                continue

            # 将该点标记为已访问。
            visited_points[neighbor_id] = True

            # 计算该点的邻域。
            neighbor_neighborhood = neighborhoods[neighbor_id]

            # 如果该点的邻域中的点的数量大于等于 min_samples，将该点加入当前聚类。
            if neighbor_neighborhood.sum() >= min_samples:
                cluster_assignments[neighbor_id] = cluster_id

            # 否则，将该点标记为噪声。
            else:
                cluster_assignments[neighbor_id] = -1

return cluster_assignments
```

使用 DBSCAN 算法进行聚类后，发现可以大致将人群分为两大类。

```
1  result.value_counts()
✓ 0.0s
1    354
2    197
3     34
4      8
5      5
6      2
Name: count, dtype: int64
```

可视化对比各个维度，分析人群画像。



经过分析可以发现，第一类的人群中，大多数：是女性，来自城市，收入集中在 10000-30000 之间，没有车，有购房按揭，购买了个人股权计划。第二类的人群中，大多数：是男性，已婚，有车，愿意存款，愿意投资，未购买个人股权计划。

八、总结和讨论

通过本次实验，我掌握了 K-means 和 DBSCAN 聚类算法的原理和代码实现，以及它们各自的优缺点。聚类常常用来挖掘数据中的商业信息，刻画用户画像。对此，关联规则挖掘也是一种有效的手段。

电 子 科 技 大 学

实 验 报 告

实验四

一、实验项目名称：推荐系统

二、实验学时：4

三、实验目的

掌握基于矩阵分解的推荐系统算法的原理和代码实现。

四、实验原理

4.1 矩阵分解算法

评分预测是推荐系统中十分重要的一环。目前，主流的评分预测实现方法包括平均值算法、基于用户的邻域算法、基于物品的邻域算法、矩阵分解算法等。其中，基于 SVD 的一系列矩阵分解模型是运用最广泛的模型。

矩阵分解，顾名思义，就是把用户-物品的评分矩阵，用若干个矩阵的乘积进行拟合，从而得到空缺评分的预测值。然而，传统的 SVD 分解有着计算复杂度高、需要补全稀疏矩阵、存储空间大、数据失真等问题。因此，后续研究提出了一系列基于 SVD 而改进的矩阵分解算法。

4.2 Funk-SVD

Funk-SVD 的基本思想是，将评分矩阵$R_{m\times n}$分解为用户特征矩阵$P_{m\times k}$和物品特征矩阵$Q_{n\times k}$的乘积，即$R = PQ^T$。用户特征矩阵可以理解为，m 位用户在 k 个特征方向上的喜好倾向；物品特征矩阵可以理解为，n 个物品在同样 k 个特征方向上的表现程度。

为了得到这样的 P 和 Q，Funk-SVD 可以采用损失函数，配合随机梯度下降法，逐步学习。具体来说，记(u,i)表示有记录的用户-物品评分对，那么要优化的目标函数是

$$\min_{p,q} \sum_{(u,i)} \left(r_{ui} - p_u q_i^T\right)^2 + \lambda(\|p_u\|^2 + \|q_i\|^2)$$

随机梯度下降的更新规则是

$$p_u^* = p_u + \alpha(2e_{ui}q_i - 2\lambda p_u)$$
$$q_i^* = q_i + \alpha(2e_{ui}p_u - 2\lambda q_i)$$

$$e_{ui} = r_{ui} - p_u q_i^T$$

经过迭代，最终就可以得到能够较好地拟合已知评分的 P 和 Q。而它们的乘积，就可以用来预测缺失的评分。

2.3 BiasSVD

BiasSVD 在 Funk-SVD 的基础上，引入了偏置项的概念。

由于每个用户评分高低的习惯不同，所以每个用户在评分时，都有一些与物品无关的因素，也就是用户偏置；又由于每个物品本身质量对评分的影响也不同，所以每个物品收到的评分，也有一些与用户无关的因素，也就是物品偏置。而这两种偏置，是基于一个全局偏置来起作用的，也就是所有评分的平均值。所以，预测的评分可以用下式来表示。

$$r_{ui} = u + b_u + b_i + p_u q_i^T$$

其中，u是评分平均值，b_u是用户偏置，b_i是物品偏置。从而，优化的目标函数变为

$$\min_{p,q,b} \sum_{(u,i)} \left(r_{ui} - u - b_u - b_i - p_u q_i^T\right)^2 + \lambda(\|p_u\|^2 + \|q_i\|^2 + \|b_u\|^2 + \|b_i\|^2)$$

更新规则随之变为

$$p_u^* = p_u + \alpha(2e_{ui}q_i - 2\lambda p_u)$$
$$q_i^* = q_i + \alpha(2e_{ui}p_u - 2\lambda q_i)$$
$$b_u^* = b_u + \alpha(2e_{ui} - 2\lambda b_u)$$
$$b_i^* = b_i + \alpha(2e_{ui} - 2\lambda b_i)$$
$$e_{ui} = r_{ui} - u - b_u - b_i - p_u q_i^T$$

五、实验内容与要求

1. 使用代码实现基于矩阵分解的推荐系统算法。
2. 在 MovieLens 数据集上验证算法，并且 MSE 不能高于 1.5。

六、实验器材（设备、元器件）：

笔记本电脑。

七、实验步骤

首先，读取 MovieLens 的评分数据集和电影数据集。

```
ratings_data_set = pd.read_csv("data/ratings.csv")
movies_data_set = pd.read_csv("data/movies.csv")
```

将评分数据集划分为训练集、验证集和测试集，比例为 8:1:1。训练集用于计算评分预测矩阵，验证集用于调整超参数，测试集用于评估模型准确性。

```
train_set = ratings_data_set.sample(frac=0.8, random_state=42)
validation_and_test_set = ratings_data_set.drop(train_set.index)
validation_set = validation_and_test_set.sample(frac=0.5, random_state=42)
test_set = validation_and_test_set.drop(validation_set.index)
```

确定评分矩阵的维度，即用户总数 m 和电影总数 n。

```
users_num = ratings_data_set.userId.unique().shape[0]
movies_num = movies_data_set.movieId.unique().shape[0]
```

build_R 函数根据给定的评分数据，创建评分矩阵；未评分用 NaN 来表示。

```
def build_R(ratings: pd.DataFrame):
    R = np.full((users_num, movies_num), np.nan)

    for rating in ratings.itertuples():
        row = rating.userId - 1
        column = movies_data_set[movies_data_set.movieId == rating.movieId].index[0]
        R[row, column] = rating.rating

    return R
```

matrix_factorization 函数使用 BiasSVD 算法来分解评分矩阵。

```
def matrix_factorization(R: np.ndarray, k=3, steps=3000, lr=0.0002, reg=0.01):
    m, n = R.shape

    P = np.random.rand(m, k)
    Q = np.random.rand(n, k)
    B = np.nanmean(R)
    BP = np.random.rand(m)
    BQ = np.random.rand(n)

    for step in range(steps):
        for i, row in enumerate(R):
            for j, value in enumerate(row):
                if np.isnan(value):
                    continue

                diff = value - B - BP[i] - BQ[j] - np.dot(P[i], Q[j])
                P[i] += lr * (2 * diff * Q[j] - 2 * reg * P[i])
                Q[j] += lr * (2 * diff * P[i] - 2 * reg * Q[j])
                BP[i] += lr * (2 * diff - 2 * reg * BP[i])
                BQ[j] += lr * (2 * diff - 2 * reg * BQ[j])

    return P, Q, B, BP, BQ
```

build_R_hat 函数构建评分预测矩阵。

```
def build_R_hat(P: np.ndarray, Q: np.ndarray, B: float, BP: np.ndarray, BQ: np.ndarray):
    m = P.shape[0]
    n = Q.shape[0]

    return (
        P @ Q.T
        + B
        + BP.reshape((-1, 1)).repeat(n, axis=1)
        + BQ.reshape((1, -1)).repeat(m, axis=0)
    )
```

evaluate 函数在真实的评分数据集上，使用 MSE 来评估预测的准确性。

```
def evaluate(R_hat: np.ndarray, R_real: np.ndarray):
    m, n = R_real.shape

    values_num = np.count_nonzero(~np.isnan(R_real))
    square_error = 0.0

    for i in range(m):
        for j in range(n):
            if np.isnan(R_real[i, j]):
                continue

            square_error += (R_real[i, j] - R_hat[i, j]) ** 2

    return square_error / values_num
```

使用训练集计算评分预测矩阵。

```
R_train = build_R(train_set)
P, Q, B, BP, BQ = matrix_factorization(R_train)
R_hat = build_R_hat(P, Q, B, BP, BQ)
```

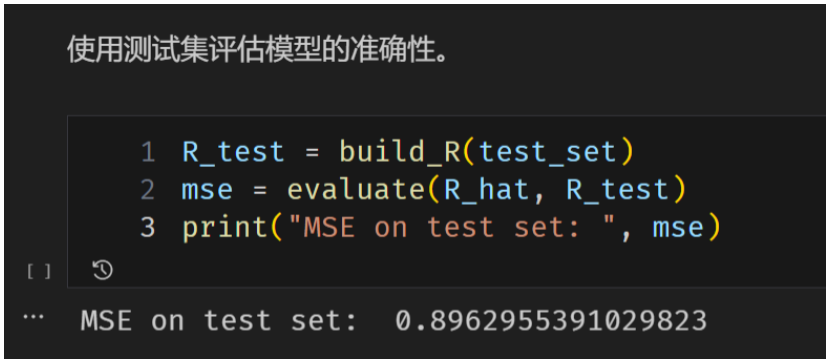
使用验证集调整超参数。

```
R_validation = build_R(validation_set)
mse = evaluate(R_hat, R_validation)
print("MSE on validation set: ", mse)
```

使用测试集评估模型的准确性。

```
R_test = build_R(test_set)
mse = evaluate(R_hat, R_test)
print("MSE on test set: ", mse)
```

实验中编写的 BiasSVD 模型，可以在测试集上取得 0.896 的 MSE 值，满足实验要求。



八、总结和讨论

通过本次实验，我对推荐系统中 SVD 系列的矩阵分解算法有了深入的了解，包括传统 SVD 的缺点、Funk-SVD、BiasSVD 等；并且通过编写 BiasSVD 的代码，熟悉了它的原理和细节。

然而，该实验并没有考虑到用户喜好随时间的变化。如果能够结合评分数据集中的 timestamp 值，引入时间维度，模型应该会更准确而健壮。