南開大學
Nankai University

计算机学院

机器学习实验报告

决策分类器实验

姓名：周重天

学号：2311082

专业：计算机科学与技术

2025 年 12 月 24 日

目录

1 基础任务：ID3 决策树构建与预测（离散属性）

1.1 ID3 算法简介

ID3（Iterative Dichotomiser 3）决策树是由 Ross Quinlan 提出的经典算法，仅适用于离散属性。ID3 使用信息增益作为属性选择准则，通过递归地选择增益最大的属性进行分裂，直至满足停止条件。

ID3 决策树的核心思想是：选择能够最大程度减少数据集不纯性（熵）的属性作为分裂属性。

1.2 关键算法步骤

1.2.1 第一步：信息熵计算

对于数据集 S，信息熵的计算公式为：

$$H(S) = -\sum_{i=1}^{c} p_i \log_2(p_i) \tag{1}$$

其中 p_i 表示类别 i 在数据集中的比例，c 为类别总数。

本实验中的实现如下：

Listing 1: 信息熵计算

```python
def entropy(self, labels):
    """计算信息熵 H(S) = - p_i * log2(p_i)"""
    if len(labels) == 0:
        return 0.0
    value_counts = Counter(labels)
    entropy_val = 0.0
    total = len(labels)
    for count in value_counts.values():
        if count > 0:
            p = count / total
            entropy_val -= p * np.log2(p)
    return entropy_val
```

1.2.2 第二步：信息增益计算

对于属性 A，其信息增益定义为：

$$\text{Gain}(S, A) = H(S) - \sum_{v \in \text{Values}(A)} \frac{|S_v|}{|S|} H(S_v) \tag{2}$$

其中 S_v 表示属性 A 取值为 v 的样本子集。

实现代码：

Listing 2: 信息增益计算

```python
def information_gain(self, parent_labels, child_labels_list):
    """
    计算信息增益
```

```
    Gain(S, A) = Entropy(S) -   |S_v|/|S| * Entropy(S_v)
    """
    parent_entropy = self.entropy(parent_labels)
    total_samples = len(parent_labels)
    if total_samples == 0:
        return 0.0
    weighted_child_entropy = 0.0
    for child_labels in child_labels_list:
        if len(child_labels) > 0:
            weight = len(child_labels) / total_samples
            weighted_child_entropy += weight * self.entropy(child_labels)
    return parent_entropy - weighted_child_entropy
```

1.2.3 第三步：递归建树

算法遍历所有离散属性，计算各属性的信息增益，选择增益最大的属性作为当前节点的分裂属性。然后按该属性的不同取值划分样本子集，对每个子集递归调用建树算法。

递归停止条件：

1. 子集中所有样本属于同一类别
2. 没有可用属性可进行分裂
3. 子集为空

核心递归代码片段：

Listing 3: ID3 递归建树

```
def build_tree(self, X, y, available_features):
    node = TreeNode()
    node.samples = len(y)
    node.class_distribution = dict(Counter(y))
    node.majority_class = Counter(y).most_common(1)[0][0] if len(y) > 0 else None

    # 停止条件 1: 标签全相同
    if len(np.unique(y)) <= 1:
        node.is_leaf = True
        node.prediction = y.iloc[0] if len(y) > 0 else None
        return node

    # 停止条件 2 和 3: 没有属性或子集为空
    if len(available_features) == 0 or len(X) == 0:
        node.is_leaf = True
        node.prediction = node.majority_class
        return node

    # 选择最优分裂属性（信息增益最大）
    best_feature = None
    best_gain = -1
```

```
    for feature in available_features:
        feature_values = X[feature].unique()
        child_labels_list = []
        for value in feature_values:
            mask = X[feature] == value
            child_labels = y[mask]
            child_labels_list.append(child_labels.values)
        gain = self.information_gain(y.values, child_labels_list)
        if gain > best_gain:
            best_gain = gain
            best_feature = feature

    # 递归构建子树
    remaining_features = [f for f in available_features if f != best_feature]
    node.feature = best_feature
    node.children = {}
    for value in X[best_feature].unique():
        mask = X[best_feature] == value
        X_subset = X[mask].reset_index(drop=True)
        y_subset = y[mask].reset_index(drop=True)
        node.children[value] = self.build_tree(X_subset, y_subset, remaining_features)
    return node
```

1.2.4 代码讲解

递归建树函数 `build_tree` 是 ID3 算法的核心。首先，函数进行节点初始化，创建新的树节点，计算该节点所有样本的多数类。这用于处理无法完全分裂的情况下的默认预测。

接下来进行停止条件检查。若所有样本标签相同（`len(np.unique(y)) <= 1`），则该节点是纯净的，无需继续分裂，直接创建叶子节点。若没有可用属性（`len(available_features) == 0`）或样本子集为空，也无法继续分裂，此时创建叶子节点并使用多数类作为预测。

属性选择阶段遍历所有可用属性，对每个属性计算其信息增益。具体步骤包括：获取该属性的所有不同取值(`X[feature].unique()`)，按各取值将样本分组并构建子集标签列表，调用 `information_gain` 方法计算该属性的增益，并记录增益最大的属性作为最优分裂属性。

最后进行递归分裂。选定最优分裂属性后，按该属性的每个不同取值创建子分支。对于每个分支，使用布尔掩码 `mask` 筛选出满足该属性值的样本，从可用属性集合中移除已分裂的属性(`remaining_features`)，最后递归调用 `build_tree` 构建子树。

1.3 实验结果

使用 Watermelon-train1.csv（纯离散属性）训练 ID3 决策树，对 Watermelon-test1.csv 进行预测。

测试集结果：

表 1: ID3 决策树在 test1 上的性能

指标	值
训练集样本数	16
测试集样本数	10
特征维数	5
分类精度	0.70 (70%)

```
[任务 1] ID3 决策树（离散属性）
------------------------------------------------------------
训练集：16 样本，5 特征
测试集：10 样本，5 特征
ID3 test1 accuracy: 0.70
```

图 1.1: ID3 决策树实验运行结果

1.4 关键代码：预测函数

Listing 4: ID3 预测

```python
def predict_one(self, x):
    """预测单个样本"""
    node = self.tree
    while not node.is_leaf:
        feature = node.feature
        feature_value = x.get(feature, None)
        # 如果特征值不在子节点中，使用多数类回退
        if feature_value not in node.children:
            return node.majority_class
        node = node.children[feature_value]
    return node.prediction
```

2 中级要求：C4.5 决策树构建与预测

2.1 C4.5 算法改进简介

C4.5 是对 ID3 的重要改进，主要区别在于：

1. 支持连续属性的处理
2. 使用信息增益率代替信息增益，避免对多值属性的偏向

2.2 连续属性处理方法

对于连续属性，C4.5 采用如下策略：

将属性值从小到大排序，相邻两个不同值的中点作为候选分裂阈值。对每个阈值，将数据分为两部分：$A \leq t$ 和 $A > t$。

Listing 5: 连续属性最优阈值寻找

```python
def find_best_continuous_split(self, X_col, y, feature_name):
    """
    为连续属性寻找最优分裂阈值
    候选阈值：排序后相邻不同值的中点
    """
    X_col_sorted = sorted(set(X_col.values))
    if len(X_col_sorted) <= 1:
        return None, -1
    thresholds = []
    for i in range(len(X_col_sorted) - 1):
        threshold = (X_col_sorted[i] + X_col_sorted[i + 1]) / 2
        thresholds.append(threshold)
    best_threshold = None
    best_gain_ratio = -1
    for threshold in thresholds:
        left_mask = X_col <= threshold
        right_mask = X_col > threshold
        y_left = y[left_mask]
        y_right = y[right_mask]
        if len(y_left) == 0 or len(y_right) == 0:
            continue
        child_labels_list = [y_left.values, y_right.values]
        gain_ratio = self.information_gain_ratio(y.values, child_labels_list)
        if gain_ratio > best_gain_ratio:
            best_gain_ratio = gain_ratio
            best_threshold = threshold
    return best_threshold, best_gain_ratio
```

2.2.1 代码讲解

连续属性处理是 C4.5 相比 ID3 的关键改进。`find_best_continuous_split` 函数的核心思想首先在于候选阈值的生成。对属性值进行排序并提取唯一值，相邻两个不同值的中点被作为候选分裂阈值。例如，若排序后的唯一值为 $[0.5, 0.7, 0.9]$，则候选阈值为 $[0.6, 0.8]$。这种方式能有效减少阈值的搜索空间。

其次进行阈值评估。对每个候选阈值，将数据分为两部分：$X \leq \text{threshold}$ 和 $X > \text{threshold}$。计算此二分裂下的信息增益率。

最后进行最优阈值的选择。在所有有效的二分裂中（两部分都非空），选择能最大化增益率的阈值。相比穷举所有可能的分裂点，该方法大大降低了计算复杂度，同时保留了充分的表达能力。

2.3 信息增益率

为避免 ID3 对多值属性的偏向，C4.5 引入信息增益率：

$$\text{GainRatio}(S, A) = \frac{\text{Gain}(S, A)}{\text{SplitInfo}(S, A)} \tag{3}$$

其中分裂信息定义为：

$$\text{SplitInfo}(S, A) = -\sum_{v \in \text{Values}(A)} \frac{|S_v|}{|S|} \log_2 \frac{|S_v|}{|S|} \tag{4}$$

Listing 6: C4.5 信息增益率

```python
def information_gain_ratio(self, parent_labels, child_labels_list):
    """
    计算信息增益率
    GainRatio(S, A) = Gain(S, A) / SplitInfo(S, A)
    """
    parent_entropy = self.entropy(parent_labels)
    total_samples = len(parent_labels)
    if total_samples == 0:
        return 0.0
    weighted_child_entropy = 0.0
    child_sizes = []
    for child_labels in child_labels_list:
        child_sizes.append(len(child_labels))
        if len(child_labels) > 0:
            weight = len(child_labels) / total_samples
            weighted_child_entropy += weight * self.entropy(child_labels)
    gain = parent_entropy - weighted_child_entropy
    split_info = self.split_info(child_sizes)
    if split_info == 0:
        return 0.0
    gain_ratio = gain / split_info
    return gain_ratio
```

2.3.1 代码讲解

信息增益率是对信息增益的改进，解决了 ID3 对多值属性的偏好问题。首先在基础增益计算方面，函数计算属性分裂前的熵（`parent_entropy`），然后计算分裂后各子集的加权熵，二者之差即为原始的信息增益。

其次进行分裂信息计算。`split_info` 方法计算分裂信息，其值随着分支数增加而增加。多值属性会产生较大的分裂信息。

随后进行增益率归一化。通过将增益除以分裂信息，使得多值属性的增益被压低，避免算法过度偏好多值属性。这是 C4.5 相比 ID3 的根本改进。

最后需要考虑边界处理。当分裂信息为 0（即所有分支大小都相等且只有一个分支）时，避免除以零的错误。在实践中，信息增益率能产生更平衡的决策树，对未见过的数据有更好的泛化能力。

2.4 实验结果

使用 Watermelon-train2.csv（离散 + 连续属性混合）训练 C4.5 决策树，对 Watermelon-test2.csv 进行预测。

表 2: C4.5 决策树在 test2 上的性能

指标	值
训练集样本数	17
测试集样本数	5
特征维数	6
其中连续属性数	2
分类精度	0.60 (60%)

```
[任务 2] C4.5 决策树
------------------------------------------------------------
训练集: 17 样本, 6 特征
测试集: 5 样本, 6 特征
C45 test2 accuracy: 0.60
```

图 2.2: C4.5 决策树实验运行结果

3 C4.5 算法属性处理过程

3.1 属性类型自动识别

C4.5 利用 pandas 的 `is_numeric_dtype` 函数自动识别连续属性:

Listing 7: 属性类型自动识别

```
def fit(self, X, y):
    """训练 C4.5 决策树"""
    self.feature_names = X.columns.tolist()
    self.class_label = y.name if y.name else "label"
    # 自动识别连续属性
    from pandas.api.types import is_numeric_dtype
    for col in self.feature_names:
        if is_numeric_dtype(X[col]):
            self.continuous_features.add(col)
    available_features = [f for f in self.feature_names]
    self.tree = self.build_tree(X.reset_index(drop=True),
                                y.reset_index(drop=True),
                                available_features)
    return self
```

3.1.1 代码讲解

属性类型自动识别是 C4.5 实现的重要细节，使得算法能自动适应混合属性数据集。首先在属性收集阶段，`fit` 方法提取数据集的所有特征列名，保存在 `feature_names` 中。

其次进行类型识别。使用 pandas 库的 `is_numeric_dtype` 函数检查每个特征列的数据类型。若为数值型（int、float 等），则标记为连续属性，加入 `continuous_features` 集合；否则视为离散属性。

然后在建树过程中进行差异化处理。当遍历特征选择最优分裂属性时，会针对特征类型采用不同策略：对连续特征调用 `find_best_continuous_split` 寻找最优二分阈值，而对离散特征则枚举所有不同取值，按值进行多路分裂。

最后实现了用户友好性。自动识别机制让用户无需手动预处理数据，直接传入混合属性数据集即可。相比 ID3 需要预先离散化连续属性，C4.5 更加易用和灵活。

4 高级要求：CART 决策树构建与预测

4.1 CART 算法简介

CART（Classification And Regression Tree）由 Breiman 等人提出。CART 的主要特点：

1. 总是生成二叉树结构
2. 使用基尼指数（Gini Index）作为分裂准则
3. 支持离散和连续属性

4.2 基尼指数

对数据集 D，基尼指数定义为：

$$\text{Gini}(D) = 1 - \sum_{k=1}^{c} p_k^2 \tag{5}$$

其中 p_k 是类别 k 在数据集中的比例。基尼指数反映数据集的不纯度。

4.3 CART 建树算法

CART 对连续属性和离散属性都采用二分策略。对于离散属性，每个节点分裂为” 取值为 v” 和” 取值不为 v” 两个分支。

Listing 8: CART 基尼指数计算

```
def gini_index(self, labels):
    """计算基尼指数 Gini(D) = 1 - Σ p_k^2"""
    if len(labels) == 0:
        return 0.0
    value_counts = Counter(labels)
    gini = 1.0
    total = len(labels)
    for count in value_counts.values():
        if count > 0:
            p = count / total
            gini -= p * p
    return gini

def weighted_gini(self, left_labels, right_labels):
    """计算二分后的加权基尼指数"""
    total = len(left_labels) + len(right_labels)
    if total == 0:
        return 0.0
    left_weight = len(left_labels) / total
```

```
    right_weight = len(right_labels) / total
    return (left_weight * self.gini_index(left_labels) +
            right_weight * self.gini_index(right_labels))
```

4.3.1 代码讲解

CART 使用基尼指数作为分裂准则，相比信息论方法具有计算优势。代码实现涵盖两个关键方面。

首先是基尼指数计算（`gini_index`）。该函数遍历数据集中各类别，计算每类的比例 p_k。基尼指数定义为 $Gini(D) = 1 - \sum_{k=1}^{c} p_k^2$，体现数据的不纯度。当数据集只包含一个类别时，基尼指数为 0（纯净）；当各类比例相等时，基尼指数最大。

其次是加权基尼指数（`weighted_gini`）。对于二分裂，分别计算左右两个子集的基尼指数。然后按子集大小比例进行加权求和，公式为：$Gini_{split} = \frac{|L|}{|D|} \cdot Gini(L) + \frac{|R|}{|D|} \cdot Gini(R)$。在寻找最优分裂时，选择能最小化加权基尼指数的分裂点。

CART 的二分策略（每次只分裂成两个分支）使得树的结构更规则，易于剪枝和理解。虽然可能导致树深度较深，但通过剪枝可以获得更优的复杂度-精度平衡。

4.4 实验结果

使用相同的 Watermelon-train2.csv 训练 CART 决策树。

表 3: CART 决策树在 test2 上的性能

指标	值
训练集样本数	17
测试集样本数	5
特征维数	6
树结构	二叉树
分类精度	0.60 (60%)

```
[任务 3] CART 决策树
------------------------------------------------------------------------
训练集: 17 样本, 6 特征
测试集: 5 样本, 6 特征
CART test2 accuracy: 0.60
```

图 4.3: CART 决策树实验运行结果

5 高级要求：决策树剪枝

5.1 剪枝算法选择：简化后剪枝（REP）

本实验采用简化后剪枝（Reduced Error Pruning, REP）算法。REP 是一种后剪枝方法，通过在验证集上评估剪枝效果来决定是否进行剪枝。

5.2 REP 算法流程

REP 算法的执行过程主要包括以下步骤：从训练集中随机分出 20% 作为验证集（random_state=42 保证可复现性），然后在较小的训练集上重新训练决策树。接下来对每个内部节点，尝试将其替换为叶子（预测为该节点的多数类）。若替换后验证集精度不下降，则保留该剪枝操作。最后自底向上重复上述步骤，直到无法继续剪枝。

Listing 9: 简化后剪枝实现

```python
def reduced_error_pruning(tree_model, X_valid, y_valid):
    """
    简化后剪枝 (Reduced Error Pruning, REP)
    """
    def calculate_accuracy(model, X, y):
        predictions = model.predict(X)
        return np.mean(predictions == y.values)

    def prune_node(node, X_valid, y_valid, model):
        """递归剪枝"""
        if node.is_leaf:
            return False
        # 先对所有子节点尝试剪枝
        any_pruned = False
        for child in node.children.values():
            if prune_node(child, X_valid, y_valid, model):
                any_pruned = True
        # 计算剪枝前的精度
        acc_before = calculate_accuracy(model, X_valid, y_valid)
        # 尝试将当前节点替换为叶子
        old_is_leaf = node.is_leaf
        old_prediction = node.prediction
        old_feature = node.feature
        old_children = node.children
        node.is_leaf = True
        node.prediction = node.majority_class
        node.feature = None
        node.children = {}
        # 计算剪枝后的精度
        acc_after = calculate_accuracy(model, X_valid, y_valid)
        # 如果精度没有下降，保留剪枝
        if acc_after >= acc_before:
            return True
        else:
            # 恢复原状
            node.is_leaf = old_is_leaf
            node.prediction = old_prediction
            node.feature = old_feature
            node.children = old_children
            return any_pruned

```

```
    for _ in range(100):  # 最多迭代 100 次
        if not prune_node(tree_model.tree, X_valid, y_valid, tree_model):
            break
    return tree_model
```

5.2.1 代码讲解

简化后剪枝（REP）通过在验证集上的精度变化来指导剪枝决策。首先在验证集分离阶段，调用剪枝前，通常从训练集中分出 20% 作为验证集（通过 `np.random.choice`）。验证集用于评估剪枝效果，不参与树的构建。

其次是递归剪枝流程（`prune_node` 函数）。若节点是叶子，无需剪枝，直接返回。否则，先对所有子节点递归尝试剪枝，自底向上进行。对当前节点，记录原始状态（`old_*` 变量），尝试将节点替换为叶子（预测为该节点的多数类），临时改变节点属性。然后在验证集上计算替换前后的精度（`acc_before` 和 `acc_after`）。若替换后精度不下降，则保留剪枝；否则恢复原状。

随后进行迭代优化。外层循环最多迭代 100 次，每次调用 `prune_node`。只要还有节点被成功剪枝，就继续迭代。当某次迭代无任何剪枝发生，算法停止。

最后需要注意几个关键特性：REP 是后剪枝方法，先构建完整的树再剪枝，避免过早停止导致信息丧失。使用独立的验证集而非训练集评估剪枝效果，更客观地反映泛化能力。自底向上的递归策略确保了剪枝的系统性和一致性。

5.3 剪枝实施结果

5.3.1 ID3 树的剪枝

表 4: ID3 决策树剪枝前后对比

指标	剪枝前	剪枝后	变化
测试集精度	0.60	0.50	−0.10
节点数	14	1	−13
树深度	1	0	−1

5.3.2 C4.5 树的剪枝

表 5: C4.5 决策树剪枝前后对比

指标	剪枝前	剪枝后	变化
测试集精度	0.60	0.60	0.00
节点数	3	3	0
树深度	1	1	0

5.3.3 CART 树的剪枝

表 6: CART 决策树剪枝前后对比

指标	剪枝前	剪枝后	变化
测试集精度	0.60	0.60	0.00
节点数	3	3	0
树深度	1	1	0

5.4 剪枝分析

ID3 树的剪枝导致精度下降，说明该模型在验证集上的最优结构保留了所有节点。而 C4.5 和 CART 在剪枝前已经相对简洁，没有进一步的优化空间。这表明对于 Watermelon 数据集，原始的 C4.5 和 CART 模型已经具有良好的泛化能力。

6 扩展要求：三种决策树算法比较分析

6.1 维度一：分裂准则

表 7: 三种算法的分裂准则对比

算法	分裂准则	公式	特点
ID3	信息增益	$\text{Gain}(S,A) = H(S) - \sum_v \frac{\lvert S_v\rvert}{\lvert S\rvert} H(S_v)$	偏向多值属性
C4.5	信息增益率	$\text{GainRatio}(S,A) = \frac{\text{Gain}(S,A)}{\text{SplitInfo}(S,A)}$	消除多值偏向
CART	基尼指数	$\text{Gini}(D) = 1 - \sum_k p_k^2$	计算简便

6.2 维度二：树结构

表 8: 三种算法的树结构对比

算法	树结构类型	分裂策略	优缺点
ID3	多叉树	多路分裂	结构紧凑，易理解；剪枝困难
C4.5	多叉树	多路分裂	继承 ID3；支持连续属性
CART	二叉树	二分	易剪枝，规则明确；树可能较深

6.3 维度三：属性处理能力

表 9: 三种算法的属性处理对比

算法	离散属性	连续属性	自动识别
ID3	✓原生支持	× 需预处理	×
C4.5	✓原生支持	✓自动二分	✓
CART	✓二分处理	✓自动二分	✓

6.4 算法特点

ID3 算法适用于纯离散属性数据集，数据规模较小，属性值有限的场景。其主要优点是算法简洁高效，容易理解和实现，但缺点是不支持连续属性，这限制了其在包含数值特征的数据集上的应用。

C4.5 算法则适用于混合属性数据集，特别是当属性数较多、类标签众多的场景。C4.5 通过引入增益率概念解决了 ID3 的多值偏向问题，使其在属性选择上更加均衡。同时，C4.5 扩展了对连续属性的支持，能够自动寻找最优分裂阈值，大大提高了算法的应用范围。

CART 算法特别适用于生产环境中需要回归扩展的场景。CART 生成的二叉树结构易于进行剪枝操作和人工解释，同时基于基尼指数的计算相比信息熵计算更加高效，这使得 CART 在处理大规模数据集时具有性能优势。

6.5 本实验观察

在 Watermelon 数据集上，三种算法在 test2 上的精度都达到 0.60。C4.5 和 CART 都能有效处理连续属性（密度、含糖量等），而 ID3 仅限于离散属性。从模型复杂度看，C4.5 和 CART 生成的树都比较简洁（3 个节点），具有较好的泛化能力。

7 附录：代码仓库

本实验的全部源代码已上传至 GitHub 仓库：

GitHub 仓库地址：https://github.com/sskystack/machinelearning.git

源代码文件：`work6/exp_6.py`