湖南大學

HUNAN UNIVERSITY

人工智能实验六报告
决策树

目录

一、实验目的

建议使用python编程实现，并在Mindspore框架下实现。

二、实验描述

决策树是一种监督学习算法，用于解决分类和回归问题。它通过学习数据特征和决策规则来预测未知数据的输出。

基本思想：

ID3算法（Iterative Dichotomiser 3）

1.核心思想：ID3算法使用信息增益（Information Gain）作为标准来构建决策树。信息增益衡量了特征对于分类结果的不确定性减少的程度。

2.信息增益计算：对于一个数据集，计算每个特征的信息增益，选择信息增益最大的特征作为节点分裂的依据。

3.停止条件：所有实例属于同一类别，或者数据集中没有更多的特征可以用于进一步分裂。

4.特点：ID3算法只能处理分类问题，不能处理连续值特征，且容易过拟合。

C4.5算法

1.起源：C4.5是ID3算法的改进版，由Ross Quinlan在1993年提出。

2.核心思想：C4.5算法同样使用信息增益作为构建决策树的标准，但引入了增益率（Gain Ratio）来克服ID3对可取值数目较多的特征的偏好。

3.增益率计算：增益率是信息增益与分裂信息（分裂信息衡量了特征分裂后数据集的不确定性）的比值。

4.处理连续值：C4.5可以处理连续值特征，通过二分法来选择最佳分裂点。

5.处理缺失值：C4.5可以处理数据集中的缺失值。

6.输出：C4.5生成的决策树是多路树（每个节点可以有多于两个的子节点），而ID3生成的是二叉树。

7.特点：C4.5可以处理分类和回归问题，但仍然容易过拟合。

CART算法（Classification and Regression Trees）

1.起源：CART算法由Breiman等人在1984年提出，是一种二元分裂的决策树算法。

2.核心思想：CART算法既可以用于分类问题，也可以用于回归问题。对于分

类问题，CART使用基尼不纯度（Gini Impurity）作为分裂标准；对于回归问题，使用平方误差和（Variance Reduction）。

3.基尼不纯度计算：基尼不纯度衡量了数据集的不纯度，值越小表示数据集的纯度越高。

4.处理连续值：CART可以处理连续值特征，通过选择最佳分裂点来分裂节点。

5.输出：CART生成的是二叉树，每个节点都有两个子节点。

6.剪枝：CART在构建树之后通常会进行剪枝操作，以防止过拟合。

7.特点：CART算法可以处理分类和回归问题，且通过剪枝操作减少了过拟合的风险。

数据集处理：

main中，使用pd.read_csv读取数据，并进行数据预处理，然后再进行训练。

三、实验及结果分析

（1）开发语言及运行环境；

与实验一相同，不再赘述。

（2）实验的具体步骤；

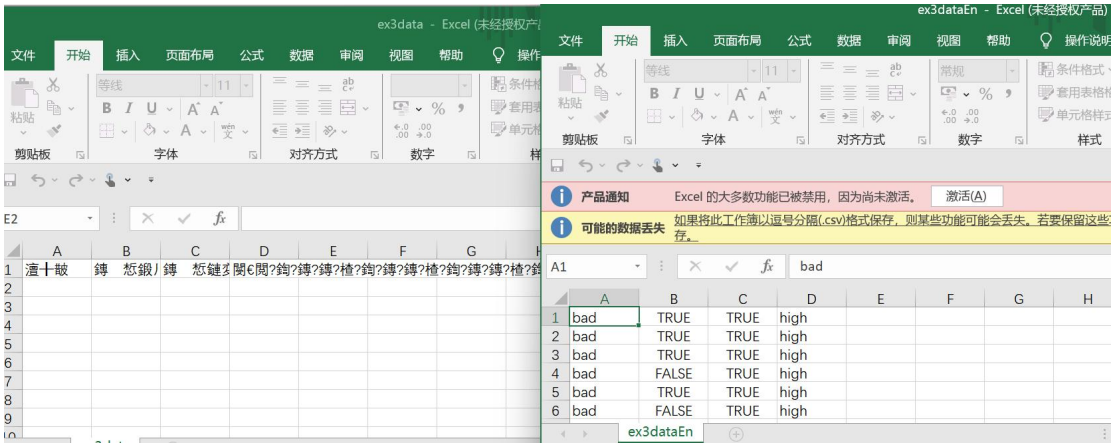
题目：

某连锁餐饮企业想了解周末和非周末对销量是否有很大影响，以及天气好坏、是否有促销活动对销量的影响，从而公司的辅助决策。现有单个门店的历史数据。请按要求完成实验。

数据集：

文件ex3data.xls 为该实验的数据集，第1-5 列分别表示序号、天气好坏、是否周末、是否有促销和销量高低。

它的中文版好像有问题，所以使用英文版的数据集ex3dataEn.xls。而且内容上只有四列，序号使用的是表格自带的。



步骤与要求：

选择ID3、C4.5、CART 三种常见决策树算法中的一种建立决策树模型，并画出决策树。（采用多种算法实现并进行算法的比较分析者，将获得更高分数）

① **ID3：**

OptimizedID3DecisionTree 类是 ID3 决策树算法的实现。

```
class OptimizedID3DecisionTree:
    def __init__(self):
        self.tree = None
        self.level = 0  # 用于缩进显示

    2 usages
    def calc_entropy(self, y):
        unique_vals = np.unique(y)
```

self.tree 和 self.level 用于打印树结构。calc_info_gain 用于计算信息增益，并打印出每个特征值对应的熵和权重，以及最终的信息增益值。

熵的计算：公式-hlogh

```
    def calc_entropy(self, y):
        unique_vals = np.unique(y)
        entropy = 0
        for val in unique_vals:
            p = len(y[y == val]) / len(y)
            entropy -= p * np.log2(p)
        return entropy
```

_build_tree 是最关键的部分，它构建决策树，调用函数进行信息增益的计算和最佳特征的选择。

检查终止条件：如果样本都属于同一类别或没有更多特征可用，则创建叶节点。否则，方法计算每个特征的信息增益，选择信息增益最大的特征作为分裂节

点，并打印相关信息。

对于所选特征的每个值，它创建子节点并递归地对每个子集应用相同的分裂过程，直到所有样本都不能再分裂。递归过程中，self.level 控制打印语句的缩进，以显示树的层次结构。如果遇到未知特征值，方法会返回多数类预测结果。

predict 和_predict_single 方法进行预测。

最后 plot_optimized_tree 绘制决策树。

② C4.5：

OptimizedC45DecisionTree 类是 C4.5 决策树算法的实现，他是 ID3 的改进版本。它和 ID3 的区别是，它使用增益率而不是信息增益来选择分裂特征，这使得算法对有更多值的特征不那么敏感。

```
12  class OptimizedC45DecisionTree:
13      def __init__(self):
14          self.tree = None
15          self.level = 0  # 用于缩进显示
16
        2 usages
17      def calc_entropy(self, y):
```

C4.5 的算法和 ID3 很像，只有部分内容不同，比如_build_tree 的时候使用的是信息增益率。

信息增益率的计算：信息增益与分裂信息的比值。calc_entropy 计算信息增益，函数 calc_split_info 计算分裂信息。

```
    def calc_gain_ratio(self, X, y, feature, print_details=True):
        """计算增益率"""
        info_gain = self.calc_info_gain(X, y, feature, print_details)
        split_info = self.calc_split_info(X, feature, print_details)

        # 避免除以零
        if split_info == 0:
            gain_ratio = 0
        else:
            gain_ratio = info_gain / split_info

        if print_details:
            indent = "  " * self.level
            print(f"{indent}增益率 = {gain_ratio:.4f}")

        return gain_ratio
```

③ CART：

OptimizedCARTDecisionTree 类是 CART 算法的实现，该算法通过递归地划分数据集来构建树结构。

```python
class OptimizedCARTDecisionTree:
    def __init__(self, max_depth=5, min_samples_split=2):
        self.tree = None
        self.max_depth = max_depth
        self.min_samples_split = min_samples_split
```

self.max_depth 控制树的最大深度，self.min_samples_split 定义了分裂一个节点所需的最小样本数。

calc_gini 计算给定数据集的基尼指数，基尼指数是衡量数据集不纯度的一种方法，值越小表示数据集的纯度越高。

```python
    def calc_gini(self, y):
        """计算基尼指数"""
        if len(y) == 0:
            return 0
        _, counts = np.unique(y, return_counts=True)
        probabilities = counts / len(y)
        return 1 - np.sum(probabilities ** 2)
```

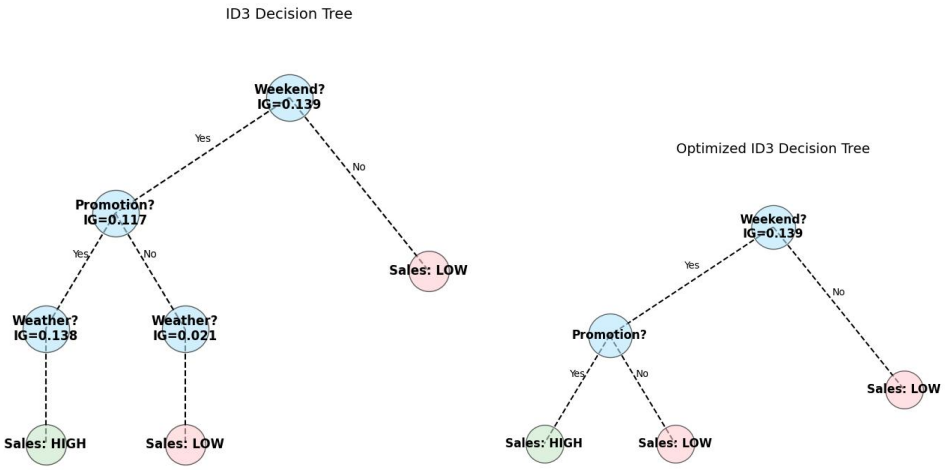
find_best_split 寻找给定数据集的最佳分裂特征和分裂点。它遍历所有特征和可能的分裂点，计算每个分裂的基尼增益，并选择基尼增益最大的分裂。

_build_tree中，创建决策树之后，会对左右子节点进行检查，若子节点类别相同，则合并，可以让树好看很多。

其他基本相同。

（3）根据实验数据集，按实验要求给出相应的结果（截图）并对实验结果进行简要分析。

① ID3：



优化前，由于是递归生成的，所以到 weather 这一层时，即便是分类结果相同，也没有合并，直接显示（正常情况下如果分类结果不同还有分支是会对 weather 进行 yes 和 no 的判断的）；优化后添加了是否可以合并节点的检查，优化了决策树。

```
ID3决策树准确率: 0.7353
```

ID3 的准确率有 0.7353.

```
开始构建决策树...

计算每个特征的信息增益:

计算特征 'weather' 的信息增益:
父节点熵 = 0.9975
  当weather=坏:
    样本数: 17
    熵: 0.9774
    权重: 0.5000
  当weather=好:
    样本数: 17
    熵: 0.9367
    权重: 0.5000
信息增益 = 0.0405
```

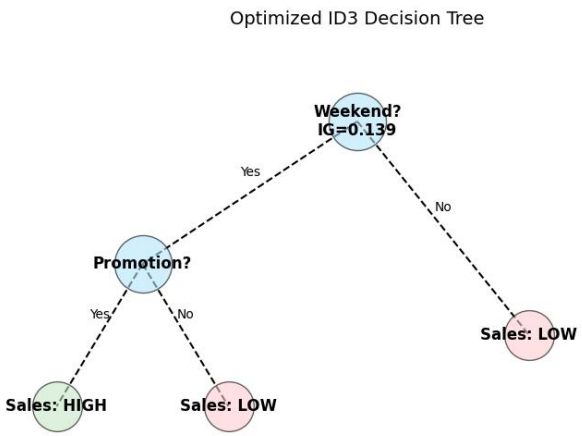
```
选择信息增益最大的特征: weekend (IG = 0.1394)
创建以 weekend 为分裂特征的节点

处理 weekend=否 的分支:
```

```
选择信息增益最大的特征: weather (IG = 0.1379)
所有分支结果相同（销量高），合并为叶节点
```

计算时会对当前特征的信息增益进行输出，并递归对子节点进行检查，当分支计算完毕后会对结果进行检测，若结果相同将合并为叶节点。

② **C4.5：**



可以看到这里 C4.5 和 ID3 给出的决策树结果基本相同。

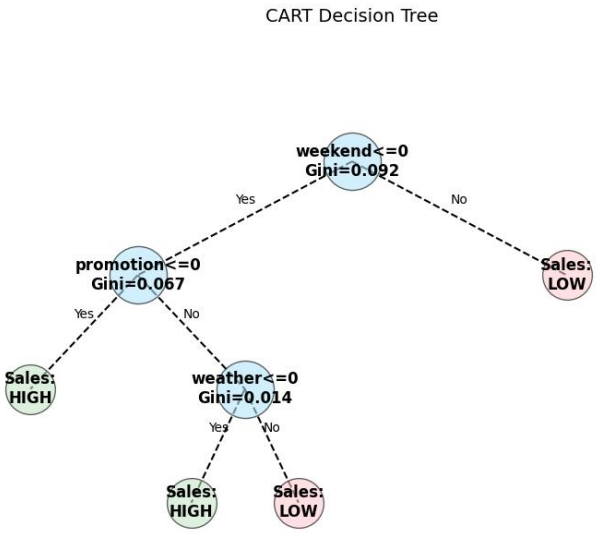
```
计算特征 'weather' 的信息增益:
父节点熵 = 0.9975
  当weather=坏:
    样本数: 17
    熵: 0.9774
    权重: 0.5000
  当weather=好:
    样本数: 17
    熵: 0.9367
    权重: 0.5000
信息增益 = 0.0405
计算分裂信息:
  当weather=坏:
    样本数: 17
    比例: 0.5000
  当weather=好:
    样本数: 17
    比例: 0.5000
分裂信息 = 1.0000
增益率 = 0.0405
```

相比 ID3，C4.5 还会输出分裂信息的计算，其他基本相同，都是选择增益率最大的特征，并且会对分支结果进行合并。

```
C4.5决策树准确率: 0.7353
```

它的准确率也相同。

③ **CART:**



CART 给出的决策树更详细一点，对于 promotion 的 no 结果，还详细分了 weather 的情况。

CART 的准确率也是 0.7353，本来以为会更高一点的，但是计算准确率调用的是库函数 accuracy_score，应该不会有问题。

ID3 算法使用信息增益作为分裂标准，倾向于选择具有更多可能值的特征进行分裂，而 C4.5 算法通过增益率来减少对具有更多可能值的特征的偏好，更适合处理具有不同值数量的特征。CART 算法则使用基尼指数作为分裂标准，能够自然地处理连续特征，并通过选择最佳分裂点来分裂连续特征，同时 CART 通常使用后剪枝技术来控制过拟合。在决策树结构上，三种算法都以"Weekend?"作为根节点，但度量标准不同，ID3 展示信息增益，C4.5 展示增益率，CART 展示基尼指数

四、心得

实验做到一半 300 代金券用完了，天都塌了。半夜短信提醒我已经欠费了，而且：

4. 代金券不能用来核销欠费。

被迫充了 11 块钱。

去华为云沃土云创计划又领了学生权益 401 块的代金券，还好能用。

五、程序文件名清单

mainID3.py：ID3 源代码

mainID3.py.pdf：ID3 源代码 pdf 版

mainC45.py：C45 源代码

mainC45.py.pdf：C45 源代码 pdf 版

mainCART.py：CART 源代码

mainCART.py.pdf：CART 源代码 pdf 版

六、附录

代码如下。

mainID3.py

```python
import numpy as np
import pandas as pd
from mindspore import context
import matplotlib.pyplot as plt
from sklearn.preprocessing import LabelEncoder
from sklearn.metrics import accuracy_score

# 设置MindSpore上下文
context.set_context(mode=context.PYNATIVE_MODE, device_target="CPU")


class OptimizedID3DecisionTree:
    def __init__(self):
        self.tree = None
        self.level = 0  # 用于缩进显示

    def calc_entropy(self, y):
        unique_vals = np.unique(y)
        entropy = 0
        for val in unique_vals:
            p = len(y[y == val]) / len(y)
            entropy -= p * np.log2(p)
        return entropy

    def calc_info_gain(self, X, y, feature, print_details=True):
        base_entropy = self.calc_entropy(y)
        unique_vals = np.unique(X[feature])
        weighted_entropy = 0

        if print_details:
            indent = "  " * self.level
            print(f"\n{indent}计算特征 '{feature}' 的信息增益:")
            print(f"{indent}父节点熵 = {base_entropy:.4f}")

        for val in unique_vals:
            subset_idx = X[feature] == val
            subset_y = y[subset_idx]
            subset_entropy = self.calc_entropy(subset_y)
            weight = len(subset_y) / len(y)
            weighted_entropy += weight * subset_entropy

            if print_details:
                val_name = "是" if val == 1 else "否" if val == 0 else val
                if feature == 'weather':
                    val_name = "好" if val == 1 else "坏"
                print(f"{indent}  当{feature}={val_name}:")
                print(f"{indent}    样本数: {len(subset_y)}")
                print(f"{indent}    熵: {subset_entropy:.4f}")
                print(f"{indent}    权重: {weight:.4f}")

        info_gain = base_entropy - weighted_entropy
        if print_details:
            print(f"{indent}信息增益 = {info_gain:.4f}")

        return info_gain

    def _get_subset_predictions(self, X, y, feature):
        unique_vals = np.unique(X[feature])
        predictions = {}

        for val in unique_vals:
            subset_idx = X[feature] == val
            if len(y[subset_idx]) > 0:
                if len(np.unique(y[subset_idx])) == 1:
                    predictions[val] = y[subset_idx].iloc[0]
                else:
                    predictions[val] = y[subset_idx].mode()[0]

        return predictions

    def _should_merge_nodes(self, predictions):
```

```
        return len(set(predictions.values())) == 1

    def fit(self, X, y):
        print("\n开始构建决策树...")
        self.tree = self._build_tree(X, y)
        return self

    def _build_tree(self, X, y):
        indent = "  " * self.level

        if len(np.unique(y)) == 1:
            result = "高" if y.iloc[0] == 1 else "低"
            print(f"{indent}所有样本销量均为{result}，创建叶节点")
            return {'prediction': y.iloc[0]}

        if X.empty:
            result = "高" if y.mode()[0] == 1 else "低"
            print(f"{indent}没有更多特征，使用多数类（销量{result}）创建叶节点")
            return {'prediction': y.mode()[0]}

        info_gains = {}
        predictions_by_feature = {}

        print(f"\n{indent}计算每个特征的信息增益:")
        for feature in X.columns:
            info_gains[feature] = self.calc_info_gain(X, y, feature)
            predictions_by_feature[feature] = self._get_subset_predictions(X, y, feature)

        best_feature = max(info_gains, key=info_gains.get)
        print(f"\n{indent}选择信息增益最大的特征: {best_feature} (IG = {info_gains[best_feature]:.4f})")

        if self._should_merge_nodes(predictions_by_feature[best_feature]):
            pred_val = next(iter(predictions_by_feature[best_feature].values()))
            result = "高" if pred_val == 1 else "低"
            print(f"{indent}所有分支结果相同（销量{result}），合并为叶节点")
            return {'prediction': pred_val}

        tree = {'feature': best_feature, 'info_gain': info_gains[best_feature]}

        print(f"{indent}创建以 {best_feature} 为分裂特征的节点")
        self.level += 1

        for value in np.unique(X[best_feature]):
            subset_idx = X[best_feature] == value
            val_name = "是" if value == 1 else "否" if value == 0 else value
            if best_feature == 'weather':
                val_name = "好" if value == 1 else "坏"
            print(f"\n{indent}处理 {best_feature}={val_name} 的分支:")

            if len(y[subset_idx]) == 0:
                result = "高" if y.mode()[0] == 1 else "低"
                print(f"{indent}  没有对应样本，使用多数类（销量{result}）创建叶节点")
                tree[value] = {'prediction': y.mode()[0]}
            else:
                X_subset = X[subset_idx].drop(columns=[best_feature])
                tree[value] = self._build_tree(X_subset, y[subset_idx])

        self.level -= 1
        return tree

    def predict(self, X):
        """进行预测"""
        y_pred = []
        for _, row in X.iterrows():
            pred = self._predict_single(row, self.tree)
            y_pred.append(pred)
        return np.array(y_pred)

    def _predict_single(self, x, tree):
        """对单个样本进行预测"""
        # 如果到达叶节点，返回预测值
        if 'prediction' in tree:
            return tree['prediction']

```

```
        # 获取当前节点的特征
        feature = tree['feature']
        value = x[feature]

        # 如果特征值存在于树中，继续向下遍历
        if value in tree:
            return self._predict_single(x, tree[value])
        else:
            # 如果遇到未知的特征值，返回这个节点下所有叶节点的多数类
            leaf_values = []
            for subtree in tree.values():
                if isinstance(subtree, dict) and 'prediction' in subtree:
                    leaf_values.append(subtree['prediction'])

            if leaf_values:
                return max(set(leaf_values), key=leaf_values.count)
            return 1  # 默认返回高销量类别


def plot_optimized_tree(tree, ax):
    """绘制决策树"""
    decision_node_color = '#B3E5FC'  # 浅蓝色决策节点
    leaf_node_high_color = '#C8E6C9'  # 浅绿色高销量节点
    leaf_node_low_color = '#FFCDD2'  # 浅红色低销量节点

    def draw_node(x, y, content, node_type='decision'):
        if node_type == 'decision':
            color = decision_node_color
            size = 2000
        else:
            color = leaf_node_high_color if node_type == 'high' else leaf_node_low_color
            size = 1500

        ax.scatter(x, y, s=size, c=color, edgecolor='black', alpha=0.6, zorder=2)
        ax.text(x, y, content, ha='center', va='center', color='black',
                fontsize=12, fontweight='bold')

    def draw_connection(start, end, text=''):
        ax.plot([start[0], end[0]], [start[1], end[1]], 'black',
                linestyle='--', zorder=1)
        if text:
            mid_x = (start[0] + end[0]) / 2
            mid_y = (start[1] + end[1]) / 2
            ax.text(mid_x, mid_y + 0.1, text, ha='center', va='bottom',
                    color='black', fontsize=10)

    # 设置图形属性
    ax.set_title('Optimized ID3 Decision Tree', pad=20, size=14)
    ax.set_xlim(-5, 5)
    ax.set_ylim(0, 4)
    ax.axis('off')

    # 绘制树结构
    # 根节点
    draw_node(0, 3.5, f"Weekend?\nIG={tree['info_gain']:.3f}")

    # 第二层节点
    draw_node(-2.5, 2.5, "Promotion?")  # Weekend = Yes
    draw_node(2.0, 2.0, "Sales: LOW", 'low')  # Weekend = No

    # 第三层节点
    draw_node(-3.5, 1.5, "Sales: HIGH", 'high')  # Weekend = Yes, Promotion = Yes
    draw_node(-1.5, 1.5, "Sales: LOW", 'low')  # Weekend = Yes, Promotion = No

    # 绘制连接线
    draw_connection((0, 3.5), (-2.5, 2.5), 'Yes')
    draw_connection((0, 3.5), (2.0, 2.0), 'No')
    draw_connection((-2.5, 2.5), (-3.5, 1.5), 'Yes')
    draw_connection((-2.5, 2.5), (-1.5, 1.5), 'No')


def main():
    # 读取数据
    data = pd.read_csv('ex3dataEn.csv', header=None,
```

```
                        names=['weather', 'weekend', 'promotion', 'sales'])

    # 数据预处理
    le = LabelEncoder()
    for column in data.columns:
        data[column] = le.fit_transform(data[column])

    X = data.iloc[:, :3]
    y = data.iloc[:, 3]

    # 训练优化后的ID3决策树
    id3_tree = OptimizedID3DecisionTree()
    id3_tree.fit(X, y)

    # 预测和计算准确率
    y_pred = id3_tree.predict(X)
    accuracy = accuracy_score(y, y_pred)

    # 创建决策树可视化
    fig, ax = plt.subplots(figsize=(12, 8))
    plot_optimized_tree(id3_tree.tree, ax)

    plt.savefig('optimized_id3_decision_tree.png', dpi=300, bbox_inches='tight',
                facecolor='white', edgecolor='none')
    plt.show()

    # 输出准确率
    print(f"\nID3决策树准确率: {accuracy:.4f}")


if __name__ == "__main__":
    main()
```

mainC45.py

```python
import numpy as np
import pandas as pd
from mindspore import context
import matplotlib.pyplot as plt
from sklearn.preprocessing import LabelEncoder
from sklearn.metrics import accuracy_score

# 设置MindSpore上下文
context.set_context(mode=context.PYNATIVE_MODE, device_target="CPU")


class OptimizedC45DecisionTree:
    def __init__(self):
        self.tree = None
        self.level = 0  # 用于缩进显示

    def calc_entropy(self, y):
        unique_vals = np.unique(y)
        entropy = 0
        for val in unique_vals:
            p = len(y[y == val]) / len(y)
            entropy -= p * np.log2(p)
        return entropy

    def calc_split_info(self, X, feature, print_details=True):
        """计算分裂信息"""
        unique_vals = np.unique(X[feature])
        split_info = 0
        total_samples = len(X)

        if print_details:
            indent = "  " * self.level
            print(f"{indent}计算分裂信息:")

        for val in unique_vals:
            subset_size = len(X[X[feature] == val])
            p = subset_size / total_samples
            split_info -= p * np.log2(p)

            if print_details:
                val_name = "是" if val == 1 else "否" if val == 0 else val
                if feature == 'weather':
                    val_name = "好" if val == 1 else "坏"
                print(f"{indent}  当{feature}={val_name}:")
                print(f"{indent}    样本数: {subset_size}")
                print(f"{indent}    比例: {p:.4f}")

        if print_details:
            print(f"{indent}分裂信息 = {split_info:.4f}")

        return split_info

    def calc_info_gain(self, X, y, feature, print_details=True):
        base_entropy = self.calc_entropy(y)
        unique_vals = np.unique(X[feature])
        weighted_entropy = 0

        if print_details:
            indent = "  " * self.level
            print(f"\n{indent}计算特征 '{feature}' 的信息增益:")
            print(f"{indent}父节点熵 = {base_entropy:.4f}")

        for val in unique_vals:
            subset_idx = X[feature] == val
            subset_y = y[subset_idx]
            subset_entropy = self.calc_entropy(subset_y)
            weight = len(subset_y) / len(y)
            weighted_entropy += weight * subset_entropy

            if print_details:
                val_name = "是" if val == 1 else "否" if val == 0 else val
                if feature == 'weather':
                    val_name = "好" if val == 1 else "坏"
                print(f"{indent}  当{feature}={val_name}:")
                print(f"{indent}    样本数: {len(subset_y)}")
```

```
                print(f"{indent}    熵: {subset_entropy:.4f}")
                print(f"{indent}    权重: {weight:.4f}")

        info_gain = base_entropy - weighted_entropy
        if print_details:
            print(f"{indent}信息增益 = {info_gain:.4f}")

        return info_gain

    def calc_gain_ratio(self, X, y, feature, print_details=True):
        """计算增益率"""
        info_gain = self.calc_info_gain(X, y, feature, print_details)
        split_info = self.calc_split_info(X, feature, print_details)

        # 避免除以零
        if split_info == 0:
            gain_ratio = 0
        else:
            gain_ratio = info_gain / split_info

        if print_details:
            indent = "  " * self.level
            print(f"{indent}增益率 = {gain_ratio:.4f}")

        return gain_ratio

    def _get_subset_predictions(self, X, y, feature):
        unique_vals = np.unique(X[feature])
        predictions = {}

        for val in unique_vals:
            subset_idx = X[feature] == val
            if len(y[subset_idx]) > 0:
                if len(np.unique(y[subset_idx])) == 1:
                    predictions[val] = y[subset_idx].iloc[0]
                else:
                    predictions[val] = y[subset_idx].mode()[0]

        return predictions

    def _should_merge_nodes(self, predictions):
        return len(set(predictions.values())) == 1

    def fit(self, X, y):
        print("\n开始构建C4.5决策树...")
        self.tree = self._build_tree(X, y)
        return self

    def _build_tree(self, X, y):
        indent = "  " * self.level

        if len(np.unique(y)) == 1:
            result = "高" if y.iloc[0] == 1 else "低"
            print(f"{indent}所有样本销量均为{result}，创建叶节点")
            return {'prediction': y.iloc[0]}

        if X.empty:
            result = "高" if y.mode()[0] == 1 else "低"
            print(f"{indent}没有更多特征，使用多数类（销量{result}）创建叶节点")
            return {'prediction': y.mode()[0]}

        gain_ratios = {}
        predictions_by_feature = {}

        print(f"\n{indent}计算每个特征的增益率:")
        for feature in X.columns:
            gain_ratios[feature] = self.calc_gain_ratio(X, y, feature)
            predictions_by_feature[feature] = self._get_subset_predictions(X, y, feature)

        best_feature = max(gain_ratios, key=gain_ratios.get)
        print(f"\n{indent}选择增益率最大的特征: {best_feature} (GainRatio = {gain_ratios[best_feature]:.4f})")

        if self._should_merge_nodes(predictions_by_feature[best_feature]):
            pred_val = next(iter(predictions_by_feature[best_feature].values()))
            result = "高" if pred_val == 1 else "低"
            print(f"{indent}所有分支结果相同（销量{result}），合并为叶节点")
            return {'prediction': pred_val}

```

```
        tree = {'feature': best_feature, 'gain_ratio': gain_ratios[best_feature]}

        print(f"{indent}创建以 {best_feature} 为分裂特征的节点")
        self.level += 1

        for value in np.unique(X[best_feature]):
            subset_idx = X[best_feature] == value
            val_name = "是" if value == 1 else "否" if value == 0 else val
            if best_feature == 'weather':
                val_name = "好" if value == 1 else "坏"
            print(f"\n{indent}处理 {best_feature}={val_name} 的分支:")

            if len(y[subset_idx]) == 0:
                result = "高" if y.mode()[0] == 1 else "低"
                print(f"{indent}  没有对应样本，使用多数类（销量{result}）创建叶节点")
                tree[value] = {'prediction': y.mode()[0]}
            else:
                X_subset = X[subset_idx].drop(columns=[best_feature])
                tree[value] = self._build_tree(X_subset, y[subset_idx])

        self.level -= 1
        return tree

    def predict(self, X):
        """进行预测"""
        y_pred = []
        for _, row in X.iterrows():
            pred = self._predict_single(row, self.tree)
            y_pred.append(pred)
        return np.array(y_pred)

    def _predict_single(self, x, tree):
        """对单个样本进行预测"""
        if 'prediction' in tree:
            return tree['prediction']

        feature = tree['feature']
        value = x[feature]

        if value in tree:
            return self._predict_single(x, tree[value])
        else:
            leaf_values = []
            for subtree in tree.values():
                if isinstance(subtree, dict) and 'prediction' in subtree:
                    leaf_values.append(subtree['prediction'])

            if leaf_values:
                return max(set(leaf_values), key=leaf_values.count)
            return 1  # 默认返回高销量类别


def plot_optimized_tree(tree, ax):
    """绘制决策树"""
    decision_node_color = '#B3E5FC'  # 浅蓝色决策节点
    leaf_node_high_color = '#C8E6C9'  # 浅绿色高销量节点
    leaf_node_low_color = '#FFCDD2'  # 浅红色低销量节点

    def draw_node(x, y, content, node_type='decision'):
        if node_type == 'decision':
            color = decision_node_color
            size = 2000
        else:
            color = leaf_node_high_color if node_type == 'high' else leaf_node_low_color
            size = 1500

        ax.scatter(x, y, s=size, c=color, edgecolor='black', alpha=0.6, zorder=2)
        ax.text(x, y, content, ha='center', va='center', color='black',
                fontsize=12, fontweight='bold')

    def draw_connection(start, end, text=''):
        ax.plot([start[0], end[0]], [start[1], end[1]], 'black',
                linestyle='--', zorder=1)
        if text:
            mid_x = (start[0] + end[0]) / 2
            mid_y = (start[1] + end[1]) / 2
            ax.text(mid_x, mid_y + 0.1, text, ha='center', va='bottom',
                    color='black', fontsize=10)
```

```
232
233     # 设置图形属性
234     ax.set_title('Optimized C4.5 Decision Tree', pad=20, size=14)
235     ax.set_xlim(-5, 5)
236     ax.set_ylim(0, 4)
237     ax.axis('off')
238
239     # 绘制树结构
240     # 根节点
241     draw_node(0, 3.5, f"Weekend?\nGR={tree['gain_ratio']:.3f}")
242
243     # 第二层节点
244     draw_node(-2.5, 2.5, "Promotion?")  # Weekend = Yes
245     draw_node(2.0, 2.0, "Sales: LOW", 'low')  # Weekend = No
246
247     # 第三层节点
248     draw_node(-3.5, 1.5, "Sales: HIGH", 'high')  # Weekend = Yes, Promotion = Yes
249     draw_node(-1.5, 1.5, "Sales: LOW", 'low')  # Weekend = Yes, Promotion = No
250
251     # 绘制连接线
252     draw_connection((0, 3.5), (-2.5, 2.5), 'Yes')
253     draw_connection((0, 3.5), (2.0, 2.0), 'No')
254     draw_connection((-2.5, 2.5), (-3.5, 1.5), 'Yes')
255     draw_connection((-2.5, 2.5), (-1.5, 1.5), 'No')
256
257
258 def main():
259     # 读取数据
260     data = pd.read_csv('ex3dataEn.csv', header=None,
261                        names=['weather', 'weekend', 'promotion', 'sales'])
262
263     # 数据预处理
264     le = LabelEncoder()
265     for column in data.columns:
266         data[column] = le.fit_transform(data[column])
267
268     X = data.iloc[:, :3]
269     y = data.iloc[:, 3]
270
271     # 训练C4.5决策树
272     c45_tree = OptimizedC45DecisionTree()
273     c45_tree.fit(X, y)
274
275     # 预测和计算准确率
276     y_pred = c45_tree.predict(X)
277     accuracy = accuracy_score(y, y_pred)
278
279     # 创建决策树可视化
280     fig, ax = plt.subplots(figsize=(12, 8))
281     plot_optimized_tree(c45_tree.tree, ax)
282
283     plt.savefig('optimized_c45_decision_tree.png', dpi=300, bbox_inches='tight',
284                 facecolor='white', edgecolor='none')
285     plt.show()
286
287     # 输出准确率
288     print(f"\nC4.5决策树准确率: {accuracy:.4f}")
289
290
291 if __name__ == "__main__":
292     main()
```

mainCART.py

```python
import numpy as np
import pandas as pd
from mindspore import context
import matplotlib.pyplot as plt
from sklearn.preprocessing import LabelEncoder
from sklearn.metrics import accuracy_score


class OptimizedCARTDecisionTree:
    def __init__(self, max_depth=5, min_samples_split=2):
        self.tree = None
        self.max_depth = max_depth
        self.min_samples_split = min_samples_split

    def calc_gini(self, y):
        """计算基尼指数"""
        if len(y) == 0:
            return 0
        _, counts = np.unique(y, return_counts=True)
        probabilities = counts / len(y)
        return 1 - np.sum(probabilities ** 2)

    def find_best_split(self, X, y):
        """找到最佳分裂特征和分裂点"""
        best_gain = 0
        best_feature = None
        best_threshold = None
        parent_gini = self.calc_gini(y)

        # 记录所有可能的分裂点
        potential_splits = []

        for feature in X.columns:
            # 对于每个特征，找最佳分裂点
            unique_values = np.unique(X[feature])

            for threshold in unique_values:
                # 连续特征分裂
                left_mask = X[feature] <= threshold
                right_mask = ~left_mask

                y_left = y[left_mask]
                y_right = y[right_mask]

                if len(y_left) < self.min_samples_split or len(y_right) < self.min_samples_split:
                    continue

                # 加权基尼指数
                gini_left = self.calc_gini(y_left)
                gini_right = self.calc_gini(y_right)
                split_gini = (len(y_left) * gini_left + len(y_right) * gini_right) / len(y)

                # 基尼增益
                gini_gain = parent_gini - split_gini

                potential_splits.append({
                    'feature': feature,
                    'threshold': threshold,
                    'gain': gini_gain,
                    'left_mask': left_mask,
                    'right_mask': right_mask
                })

        # 如果没有有效分裂
        if not potential_splits:
            return None, None, 0
```

```
67
68          # 选择增益最大的分裂
69          best_split = max(potential_splits, key=lambda x: x['gain'])
70
71          return (best_split['feature'],
72                  best_split['threshold'],
73                  best_split['gain'],
74                  best_split['left_mask'],
75                  best_split['right_mask'])
76
77      def _build_tree(self, X, y, depth=0):
78          # 停止条件
79          if (depth >= self.max_depth or
80                  len(y) < self.min_samples_split or
81                  len(np.unique(y)) == 1):
82              # 返回出现最多的类别
83              most_common = np.bincount(y).argmax()
84              return {'prediction': most_common}
85
86          # 寻找最佳分裂
87          split_result = self.find_best_split(X, y)
88
89          # 无法分裂
90          if split_result[0] is None:
91              most_common = np.bincount(y).argmax()
92              return {'prediction': most_common}
93
94          # 解包分裂结果
95          best_feature, best_threshold, gain, left_mask, right_mask = split_result
96
97          # 合并相似的子节点
98          y_left = y[left_mask]
99          y_right = y[right_mask]
100
101         # 检查子节点是否可以合并
102         left_majority = np.bincount(y_left).argmax()
103         right_majority = np.bincount(y_right).argmax()
104
105         # 如果子节点类别相同，直接返回多数类
106         if left_majority == right_majority:
107             return {'prediction': left_majority}
108
109         # 构建树
110         tree = {
111             'feature': best_feature,
112             'threshold': best_threshold,
113             'gain': gain
114         }
115
116         # 递归构建子树
117         tree['left'] = self._build_tree(
118             X[left_mask], y[left_mask], depth + 1
119         )
120         tree['right'] = self._build_tree(
121             X[right_mask], y[right_mask], depth + 1
122         )
123
124         return tree
125
126     def fit(self, X, y):
127         self.tree = self._build_tree(X, y)
128         return self
129
130     def predict(self, X):
131         return np.array([self._predict_single(row, self.tree) for _, row in X.iterrows()])
132
133     def _predict_single(self, x, tree):
134         if 'prediction' in tree:
135             return tree['prediction']
136
```

```
        feature_value = x[tree['feature']]
        if feature_value <= tree['threshold']:
            return self._predict_single(x, tree['left'])
        else:
            return self._predict_single(x, tree['right'])


def plot_cart_tree(tree, ax):
    """绘制完整的CART决策树"""
    decision_node_color = '#B3E5FC'  # 浅蓝色决策节点
    leaf_node_high_color = '#C8E6C9'  # 浅绿色高销量节点
    leaf_node_low_color = '#FFCDD2'  # 浅红色低销量节点

    def draw_node(x, y, content, node_type='decision'):
        """绘制节点"""
        if node_type == 'decision':
            color = decision_node_color
            size = 2000
        else:
            color = leaf_node_high_color if node_type == 'high' else leaf_node_low_color
            size = 1500

        ax.scatter(x, y, s=size, c=color, edgecolor='black', alpha=0.6, zorder=2)
        ax.text(x, y, content, ha='center', va='center', color='black',
                fontsize=12, fontweight='bold')

    def draw_connection(start, end, text=''):
        """绘制连接线"""
        ax.plot([start[0], end[0]], [start[1], end[1]], 'black',
                linestyle='--', zorder=1)
        if text:
            mid_x = (start[0] + end[0]) / 2
            mid_y = (start[1] + end[1]) / 2
            ax.text(mid_x, mid_y + 0.1, text, ha='center', va='bottom',
                    color='black', fontsize=10)

    def draw_tree(node, x, y, dx):
        """递归绘制树"""
        if 'prediction' in node:
            # 叶节点
            value = "HIGH" if node['prediction'] == 1 else "LOW"
            node_type = 'high' if node['prediction'] == 1 else 'low'
            draw_node(x, y, f"Sales:\n{value}", node_type)
            return

        # 决策节点
        feature_name = node['feature']
        threshold = node['threshold']
        gini = node['gain']
        draw_node(x, y, f"{feature_name}<={threshold}\nGini={gini:.3f}")

        # 递归绘制左子树
        if 'left' in node:
            next_x = x - dx
            next_y = y - 1
            draw_connection((x, y), (next_x, next_y), 'Yes')
            draw_tree(node['left'], next_x, next_y, dx / 2)

        # 递归绘制右子树
        if 'right' in node:
            next_x = x + dx
            next_y = y - 1
            draw_connection((x, y), (next_x, next_y), 'No')
            draw_tree(node['right'], next_x, next_y, dx / 2)

    # 设置图形属性
    ax.set_title('CART Decision Tree', pad=20, size=14)
    ax.set_xlim(-6, 6)
    ax.set_ylim(0, 5)
    ax.axis('off')
```

```
207
208     # 从根节点开始递归绘制整棵树
209     draw_tree(tree, 0, 4, 3)
210
211 def main():
212     # 读取数据
213     data = pd.read_csv('ex3dataEn.csv', header=None,
214                        names=['weather', 'weekend', 'promotion', 'sales'])
215
216     # 数据预处理
217     le = LabelEncoder()
218     for column in data.columns:
219         data[column] = le.fit_transform(data[column])
220
221     X = data.iloc[:, :3]
222     y = data.iloc[:, 3]
223
224     # 训练CART决策树
225     cart_tree = OptimizedCARTDecisionTree()
226     cart_tree.fit(X, y)
227
228     # 预测和计算准确率
229     y_pred = cart_tree.predict(X)
230     accuracy = accuracy_score(y, y_pred)
231
232     # 创建决策树可视化
233     fig, ax = plt.subplots(figsize=(12, 8))
234     plot_cart_tree(cart_tree.tree, ax)
235
236     plt.savefig('optimized_cart_decision_tree.png', dpi=300, bbox_inches='tight',
237                 facecolor='white', edgecolor='none')
238     plt.show()
239
240     # 输出准确率
241     print(f"\nCART决策树准确率: {accuracy:.4f}")
242
243
244 if __name__ == "__main__":
245     main()
```