1. 实验目的

1.学会理解数据并对数据进行预处理；
2.理解决策树的原理并掌握其构建方法。

2. 实验内容

1．熟悉 Pandas 的安装和使用，并对数据进行预处理和相关分析；
2. 使用 Python 语言编码实现一种决策树算法，解决个人年收入的分类问题。

3. 实验环境

- ✓ Jupyter
- ✓ PyCharm 2023.1.4

4. 提交文件说明

- ✓ lab2.ipynb：未对决策树进行剪枝优化或交叉验证的代码（已包含运行结果）
- ✓ lab2_upper.py：对决策树进行剪枝优化和交叉验证的代码
- ✓ test_adult.csv：进行数据预处理后的测试集数据
- ✓ train_adult.csv：进行数据预处理后的训练集数据
- ✓ test_predictions.csv：测试集预测结果文件

5. 实验过程

1. 数据预处理的过程及结果
（1）删除无用或冗余的特征

```
In [38]: df_train_set.drop(['fnlwgt', 'educationNum'], axis=1, inplace=True)
         df_test_set.drop(['fnlwgt', 'educationNum'], axis=1, inplace=True)
```

（2）去除重复的样本

```
In [40]: df_train_set.drop_duplicates(inplace=True)
         df_test_set.drop_duplicates(inplace=True)
```

重复的样本可能会导致模型偏向某些特定的数据模式，影响模型的泛化能力；也会增加计算成本，影响训练效率。

通过删除重复样本，确保每个样本都是独立的，数据集更加干净，模型训练更有效率。

（3）处理缺失值和异常值

In [41]:
```
### 2.3 缺失值处理
```

In [42]:
```
df_train_set.dropna(inplace=True)
df_test_set.dropna(inplace=True)
```

In [43]:
```
### 2.4 异常值处理
```

In [44]:
```
new_columns = ['workclass', 'education', 'maritalStatus', 'occupation', 'relationship', 'race', 'sex',
               'nativeCountry', 'income']
for col in new_columns:
    df_train_set = df_train_set[~df_train_set[col].str.contains(r'\?', regex=True)]
    df_test_set = df_test_set[~df_test_set[col].str.contains(r'\?', regex=True)]

df_train_set.reset_index(drop=True, inplace=True)
df_test_set.reset_index(drop=True, inplace=True)
```

缺失值会导致模型无法处理，或者引入不确定性。直接删除含有缺失值的行，确保模型输入的数据完整。尽管删除后数据量会减少，但剩余的数据质量更高。

在数据集中，某些离散型变量可能以'?'表示未知或缺失的值，这些值不能直接用于模型训练。将包含'?'的行视为异常值，进行删除处理。删除异常值可以提高模型的准确性，减少数据噪声。

（4）处理连续型变量

In [45]:
```
### 2.5 连续型变量处理
```

In [46]:
```
bins = [0, 25, 50, 75, 100]
df_train_set['age'] = pd.cut(df_train_set['age'], bins, labels=False)
df_test_set['age'] = pd.cut(df_test_set['age'], bins, labels=False)
```

对 age（年龄）分箱处理进行（决策树模型对离散型变量处理较为直接，将连续变量离散化可以简化模型的复杂度。分箱处理可以捕获年龄与收入之间的非线性关系）。将年龄划分为四个区间：[0,25)、[25,50)、[50,75)、[75,100)，使用 labels=False 将区间映射为整数标签（0、1、2、3）。

处理后降低了模型对数据的精细程度要求，减少过拟合的风险。

（5）编码离散型变量

In [48]:
```
# 定义一个通用的映射函数，处理未知类别
```

In [49]:
```
def create_mapping(column_values):
    unique_values = column_values.unique()
    mapping = {label: idx for idx, label in enumerate(unique_values)}
    mapping['unknown'] = len(mapping)  # 为未知类别添加一个索引
    return mapping
```

In [50]:
```
# 处理训练集
mappings = {}  # 保存所有映射，以便在测试集中使用相同的映射

for col in new_columns:
    if col == 'income':
        continue  # income列单独处理
    mapping = create_mapping(df_train_set[col])
    mappings[col] = mapping
    df_train_set[col] = df_train_set[col].map(mapping)
```

```
# 处理测试集，使用与训练集相同的映射
for col in new_columns:
    if col == 'income':
        continue
    mapping = mappings[col]
    if 'unknown' not in mapping:
        mapping['unknown'] = len(mapping)
    df_test_set[col] = df_test_set[col].map(lambda x: mapping.get(x, mapping['unknown']))
```

使用整数编码方式，将每个类别映射为一个唯一的整数。在测试集和训练集中，可能出现训练集中未见过的类别。通过在映射中添加'unknown'类别，确保模型能够处理未知类别。

编码后保留了类别信息，模型可以利用这些信息进行学习。

（6）处理目标变量 income

```
# income编码
income_mapping = {'<=50K': 0, '>50K': 1}
df_train_set['income'] = df_train_set['income'].str.strip()
df_train_set['income'] = df_train_set['income'].map(income_mapping)
mappings['income'] = income_mapping
```

income 是模型的目标变量，需要将其转换为数值形式进行二分类任务。去除空格和特殊字符，确保数据的一致性。将'<=50K'映射为 0，表示收入低于或等于 50K；将'>50K'映射为 1，表示收入高于 50K。

处理后明确了分类标准，便于模型训练和评估。同时，也确保了目标变量的数据质量。

（7）填充剩余的缺失值

```
# 如果仍有缺失值，可以选择填充或删除
df_train_set.fillna(-1, inplace=True)
df_test_set.fillna(-1, inplace=True)
```

在经过前面的处理后，可能仍存在少量的缺失值。而直接删除可能会丢失过多的数据，选择填充特殊值代替：使用-1 作为填充值，表示该特征未知或缺失。

填充后保留了更多的数据，增加了样本量。

（8）数据集的最终检查

```
# 检查数据集长度
print("训练集样本数：", len(df_train_set))
print("测试集样本数：", len(df_test_set))

# 检查是否存在缺失值
print("训练集缺失值情况：\n", df_train_set.isnull().sum())
print("测试集缺失值情况：\n", df_test_set.isnull().sum())
```

确认数据集的样本数量是否合理，避免因处理步骤导致的数据过少。同时，检查是否仍存在缺失值，确保数据完整性。

2. 构建决策树的基本过程

（a）训练集和测试集的划分方法

数据集已经被预先划分为训练集和测试集，分别保存在 'adult.data' 和 'adult.test' 文件中。

训练集：①仅使用训练集 df_train_set 来构建决策树模型；②模型的参数和结构完全由训练集的数据决定。

测试集：①测试集 df_test_set 仅用于评估模型的性能，不参与模型的训练过程；②在预测时，使用训练集中的映射和编码方式对测试集进行相同的预处理，确保特征的含义和取值一致。

在整个过程中，训练集和测试集始终是分开的，没有将测试集的数据用于模型的训练，也没有从训练集中抽取部分数据作为测试集。

（b）特征选择的具体过程

在每个节点上重复进行特征选择：①计算当前数据集的基尼指数；②遍历所有特征和特征值，尝试不同的划分方式；③对每个划分，计算划分后的加权基尼指数；④选择使加权基尼指数最小的特征和特征值进行划分。

递归地构建决策树：①使用最佳特征和特征值，将数据集划分为左、右子集；②对左、右子集分别递归地进行特征选择和划分，构建子树；③当满足停止条件（如节点纯净或没有特征可用）时，停止递归，生成叶节点。

总的来说，代码实现了一个基于 CART 算法的决策树模型（通过计算基尼指数，衡量数据集的纯度，指导特征选择；选择最佳特征和划分点，使得数据集划分后，子集的纯度最大化；递归地构建决策树，直到满足停止条件，最终形成一棵完整的决策树模型），用于对成人收入数据集进行分类预测。

构建决策树具体过程如下：

（1）计算基尼系数

In [52]:
```
def calc_gini(df):
    labels = df['income']
    label_counts = labels.value_counts()
    total = len(labels)
    gini = 1.0 - sum((count / total) ** 2 for count in label_counts)
    return gini
```

使用基尼系数公式计算：

$$\text{Gini} = 1 - \sum_{k=1}^{K} \left(\frac{\text{count}_k}{\text{total}}\right)^2$$

（$count_k$是第 k 类的样本数）

返回计算得到的基尼系数 gini。

（2）按照特定的属性、属性值划分数据集

```
def split_dataset(df, index, value):
    feature = df.columns[index]
    left_df = df[df[feature] == value]
    right_df = df[df[feature] != value]
    return left_df, right_df
```

根据指定特征和特征值，将数据集 df 划分为左右两个子集，并返回左、右子集。

（3）选择最佳特征进行划分

```
def choose_best_feature_to_split(df):
    base_gini = calc_gini(df)
    best_gini = float('inf')
    best_feature_index = -1
    best_value = None
    best_splits = None

    num_features = len(df.columns) - 1  # Exclude the label column 'income'
    for i in range(num_features):
        feature = df.columns[i]
        unique_values = df[feature].unique()
        for value in unique_values:
            left_df, right_df = split_dataset(df, i, value)
            if len(left_df) == 0 or len(right_df) == 0:
                continue  # Skip invalid splits

            total_instances = len(df)
            weight_left = len(left_df) / total_instances
            weight_right = len(right_df) / total_instances
            gini_left = calc_gini(left_df)
            gini_right = calc_gini(right_df)
            gini_split = weight_left * gini_left + weight_right * gini_right

            if gini_split < best_gini:
                best_gini = gini_split
                best_feature_index = i
                best_value = value
                best_splits = (left_df, right_df)

    if best_feature_index == -1:
        # No valid split found
        return None, None, None
    else:
        return (best_feature_index, best_value), best_splits, best_gini
```

在当前数据集 df 中，选择一个最佳的特征和相应的特征值进行划分，使得划分后的加权基尼指数最小。

（4）递归构建决策树

```
def build_decision_tree(df, columns, flags):
    labels = df['income']
    # Base case 1: If all labels are the same, return the label
    if len(labels.unique()) == 1:
        return {'label': labels.iloc[0]}

    # Base case 2: If no features left to split on, return the majority label
    if len(df.columns) == 1:  # Only the label column is left
        majority_label = labels.value_counts().idxmax()
        return {'label': majority_label}

    # Choose the best feature to split
    best_feature, best_splits, best_gini = choose_best_feature_to_split(df)

    if best_feature is None:
        # No valid split found, return majority label
        majority_label = labels.value_counts().idxmax()
        return {'label': majority_label}

    feature_index, feature_value = best_feature
    feature_name = df.columns[feature_index]

    # Build subtrees
    left_df, right_df = best_splits

    left_subtree = build_decision_tree(left_df.drop(columns=[feature_name]), columns, flags)
    right_subtree = build_decision_tree(right_df.drop(columns=[feature_name]), columns, flags)

    # Return the tree
    return {'feature_index': feature_index,
            'feature_name': feature_name,
            'value': feature_value,
            'left': left_subtree,
            'right': right_subtree}
```

通过递归地选择最佳特征进行划分，构建一棵决策树，使得每个叶节点尽可能纯净。递归结束条件确保了树的构建在适当的时候停止，避免过拟合。

（5）构建决策树并保存模型

```
def save_decision_tree(cart):
    np.save('cart.npy', cart)

def load_decision_tree():
    cart = np.load('cart.npy', allow_pickle=True)
    return cart.item()
```

将构建好的决策树模型 cart 保存到文件 cart.npy 中。

3. 对决策树进行剪枝优化和交叉验证的设计

通过预剪枝和交叉验证，增强了模型的泛化能力。

预剪枝策略：①最大深度(max_depth)：当树的深度达到设定的最大深度时，停止划分，防止过拟合；②最小样本数(min_samples_split)：当节点的样本数小于设定值时，停止划分，防止过拟合。

```
def build_decision_tree(df, max_depth=None, min_samples_split=2, depth=0):
    print(f"构建深度为 {depth} 的决策树，样本数：{len(df)}")
    labels = df['income']
    # 基础情况1：如果所有标签都相同，返回该标签
    if len(labels.unique()) == 1:
        return {'label': labels.iloc[0]}
    # 预剪枝条件
    if max_depth is not None and depth >= max_depth:
        majority_label = labels.value_counts().idxmax()
        return {'label': majority_label}
    if len(df) < min_samples_split:
        majority_label = labels.value_counts().idxmax()
        return {'label': majority_label}
    # 选择最好的划分特征
    best_feature, best_splits, best_gini = choose_best_feature_to_split(df)
    if best_feature is None:
        majority_label = labels.value_counts().idxmax()
        return {'label': majority_label}
    feature_name = best_feature
    # 构建子树
    left_df, right_df = best_splits

    # 调试信息
    print(f"在深度 {depth} 处划分特征 '{feature_name}'，基尼指数 {best_gini}")

    left_subtree = build_decision_tree(left_df, max_depth, min_samples_split, depth + 1)
    right_subtree = build_decision_tree(right_df, max_depth, min_samples_split, depth + 1)

    # 返回树
    return {'feature_name': feature_name,
            'left': left_subtree,
            'right': right_subtree}
```

交叉验证：①将数据集随机打乱，分成 k 个等大小的折；②在每个折中，使用 k-1 个折作为训练集，1 个折作为验证集；③训练模型并在验证集上评估，记

录准确率；④最终返回所有折的平均准确率。

```
1 usage
def cross_validate(df, k=5, max_depth=None, min_samples_split=2):
    indices = np.arange(len(df))
    np.random.shuffle(indices)
    fold_size = len(df) // k
    acc_list = []
    for i in range(k):
        val_indices = indices[i * fold_size:(i + 1) * fold_size]
        train_indices = np.setdiff1d(indices, val_indices)
        train_df = df.iloc[train_indices]
        val_df = df.iloc[val_indices]
        cart = build_decision_tree(train_df, max_depth=max_depth, min_samples_split=min_samples_split)
        pred_list = predict(cart, val_df)
        acc = calc_acc(pred_list, val_df['income'].to_numpy())
        acc_list.append(acc)
    mean_acc = np.mean(acc_list)
    return mean_acc
```

进行交叉验证，选择最佳参数

```
best_acc = 0
best_params = None
for param in params:
    acc = cross_validate(df_train_processed, k=5, max_depth=param['max_depth'],
                         min_samples_split=param['min_samples_split'])
    print(f"参数 {param} 下的交叉验证准确率为: {acc}")
    if acc > best_acc:
        best_acc = acc
        best_params = param

print(f"最佳参数为 {best_params}，交叉验证准确率为 {best_acc}")
```

4. 测试结果和预测准确率

（1）未对决策树进行剪枝优化或交叉验证的结果：

In [29]:
```
# 开始预测
columns = df_train.columns.to_list()
cart = load_decision_tree()  # 加载模型
test_list = df_test_set['income'].to_numpy()
pred_list = predict(cart, df_test_set, columns)
acc = calc_acc(pred_list, test_list)
print("测试集上的准确率为:", acc)
```

测试集上的准确率为: 0.8167020523708421

In [30]:
```
# 将预测结果输出到新的.csv文件中
df_test_set['prediction'] = pred_list
df_test_set.to_csv('test_predictions.csv', index=False)
print("预测结果已保存到 test_predictions.csv 文件中。")
```

预测结果已保存到 test_predictions.csv 文件中。

（2）已对决策树进行剪枝优化和交叉验证的结果：

```
训练集样本数： (30162, 105)
测试集样本数： (15060, 105)
参数 {'max_depth': None, 'min_samples_split': 2} 下的交叉验证准确率为: 0.847
```

```
参数 {'max_depth': 5, 'min_samples_split': 5} 下的交叉验证准确率为: 0.843
参数 {'max_depth': 8, 'min_samples_split': 10} 下的交叉验证准确率为: 0.849
最佳参数为 {'max_depth': 10, 'min_samples_split': 5}，交叉验证准确率为 0.852
构建深度为 0 的决策树，样本数：30162
在深度 0 处划分特征 'capitalGain'，基尼指数 0.459
构建深度为 1 的决策树，样本数：28400
在深度 1 处划分特征 'age'，基尼指数 0.450
构建深度为 2 的决策树，样本数：26000
在深度 2 处划分特征 'education_Bachelors'，基尼指数 0.430
```

（中间省略）

```
构建深度为 10 的决策树，样本数：200
所有样本均为类别 1，停止划分
构建深度为 10 的决策树，样本数：150
所有样本均为类别 0，停止划分
在测试集上的准确率为: 0.8529681744325046
预测结果已保存到 test_predictions.csv 文件中。
```

可以发现，经过剪枝优化和交叉验证后的决策树，在测试集上的准确率有了提升，从原先的 81.67%提高了 85.30%左右。

个人签名 ： 柯育淳

2024 年 10 月 20 日