1. 实验目的

构建一棵基于 ID3 算法的决策树,并用该决策树对小数据集进行预测,以此熟悉掌握决策树的 ID3 算法,然后训练学习一些大规模数据并且完成决策,并期望利用该决策树对于一些样例做出正确的预测。

2. 算法核心

决策树 ID3 算法的核心是基于信息增益选择最优属性进行分类。它主要包括以下几个核心步骤：

计算信息熵（Entropy）：

信息熵是衡量数据混乱程度的指标。数据越纯（同一类别占比越大），熵越低；数据越混乱（类别分布越均匀），熵越高。

熵公式：

$$H(D) = -\sum_{i=1}^{n} p_i \log_2 p_i$$

计算信息增益（Information Gain）：

信息增益是通过某个属性划分数据后，熵的减少量，表示属性对分类的贡献。信息增益公式：

$$Gain(D, A) = H(D) - \sum_{v \in Values(A)} \frac{|D_v|}{|D|} H(D_v)$$

选择最优属性：

对所有候选属性，计算其信息增益，选择信息增益最大的属性作为当前节点的划分属性。

递归构建决策树：

根据最优属性划分数据，生成子节点；

对每个子节点重复上述步骤，直到满足停止条件（如数据纯度达到一定水平或没有更多属性可用）。

停止条件：数据集中所有样本属于同一类别；没有剩余的候选属性；达到预设的树深度限制。

核心思想：

通过逐步减少数据的不确定性，递归地将数据划分为更纯的子集，最终形成

一个以最优属性为分裂点的树形结构，用于分类或决策。

3. 实验过程

3.1 数据结构定义

TreeNode 表示决策树的节点，包含属性 attribute（当前节点的划分属性）、branches（分支字典）和 classification（叶子节点的分类结果）。

提供方法 is_leaf() 判断节点是否为叶子节点。

```
class TreeNode:
    tj-messi
    def __init__(self, attribute=None, classification=None):
        self.attribute = attribute  # 节点的属性
        self.branches = {}  # 子节点（键为分裂的值）
        self.classification = classification  # 叶子节点的分类结果

    2 usages (2 dynamic)  tj-messi
    def is_leaf(self):
        return self.classification is not None
```

Sample 表示一个样本数据，包括 attributes（属性值列表）和 classification（分类结果）。

```
class Sample:
    tj-messi
    def __init__(self, attributes, classification):
        self.attributes = attributes  # 属性集合
        self.classification = classification  # 分类结果
```

3.2 功能函数定义

calculate_entropy(samples)：计算给定样本集的熵，用于衡量数据的不确定性。

```
2 usages  tj-messi
def calculate_entropy(samples):
    counts = Counter(sample.classification for sample in samples)
    total = len(samples)
    entropy = -sum((count / total) * math.log2(count / total) for count in counts.values())
    return entropy
```

calculate_gain(samples, attribute_index)：计算某个属性对样本集的信息增益。

```
1 usage  tj-messi
def calculate_gain(samples, attribute_index):
    total_entropy = calculate_entropy(samples)
    split_samples = defaultdict(list)

    for sample in samples:
        split_samples[sample.attributes[attribute_index]].append(sample)

    split_entropy = sum(
        (len(subset) / len(samples)) * calculate_entropy(subset)
        for subset in split_samples.values()
    )
    return total_entropy - split_entropy
```

get_majority_class(samples)：返回样本集中数量最多的类别，作为当前节点的分类结果。此时要注意 samples 中没有分类好的先别输出。

```
2 usages  tj-messi
def get_majority_class(samples):
    if not samples:
        return "未知"  # 或者返回一个合理的默认值
    counts = Counter(sample.classification for sample in samples)
    return counts.most_common(1)[0][0]
```

choose_best_attribute(samples, attributes)：遍历所有属性，计算每个属性的信息增益。 返回信息增益最大的属性的索引，作为当前节点的划分属性。

```
def choose_best_attribute(samples, attributes):
    best_gain = -1
    best_index = -1
    for i, attribute in enumerate(attributes):
        gain = calculate_gain(samples, i)
        if gain > best_gain:
            best_gain = gain
            best_index = i
    return best_index
```

构建决策树：

stop_criteria(samples, attributes)：

判断是否满足停止条件：

1. 属性集合为空。

2. 所有样本的分类相同。
3. 达到最大深度限制。

```
1 usage  tj-messi
def stop_criteria(samples, attributes):
    global depth
    return len(attributes) == 0 or all(
        sample.classification == samples[0].classification for sample in samples
    ) or depth >= MAX_DEPTH
```

`build_decision_tree(samples, attributes)`：递归构建决策树的主函数：

1.判断是否满足停止条件，若满足则创建叶子节点。

2.选择最优划分属性，基于该属性将样本划分为子集。

3.递归构建每个子节点，并将子节点挂载到当前节点上。

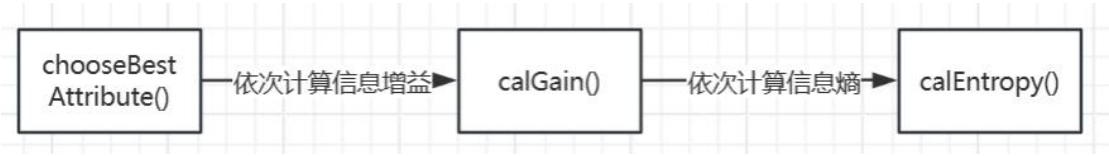
```
def build_decision_tree(samples, attributes):
    global depth
    if stop_criteria(samples, attributes):
        return TreeNode(classification=get_majority_class(samples))

    best_index = choose_best_attribute(samples, attributes)
    best_attribute = attributes[best_index]
    node = TreeNode(attribute=best_attribute)

    split_samples = defaultdict(list)
    for sample in samples:
        split_samples[sample.attributes[best_index]].append(sample)

    remaining_attributes = attributes[:best_index] + attributes[best_index + 1:]
    depth += 1

    for value, subset in split_samples.items():
        node.branches[value] = build_decision_tree(subset, remaining_attributes)

    depth -= 1
    return node
```

predict(tree, attribute_values, attribute_names)： 输入测试样本的属性值，通过递归从根节点到叶子节点进行预测。如果遇到测试样本的属性值不在训练集中，返回当前节点所有子节点的多数类。

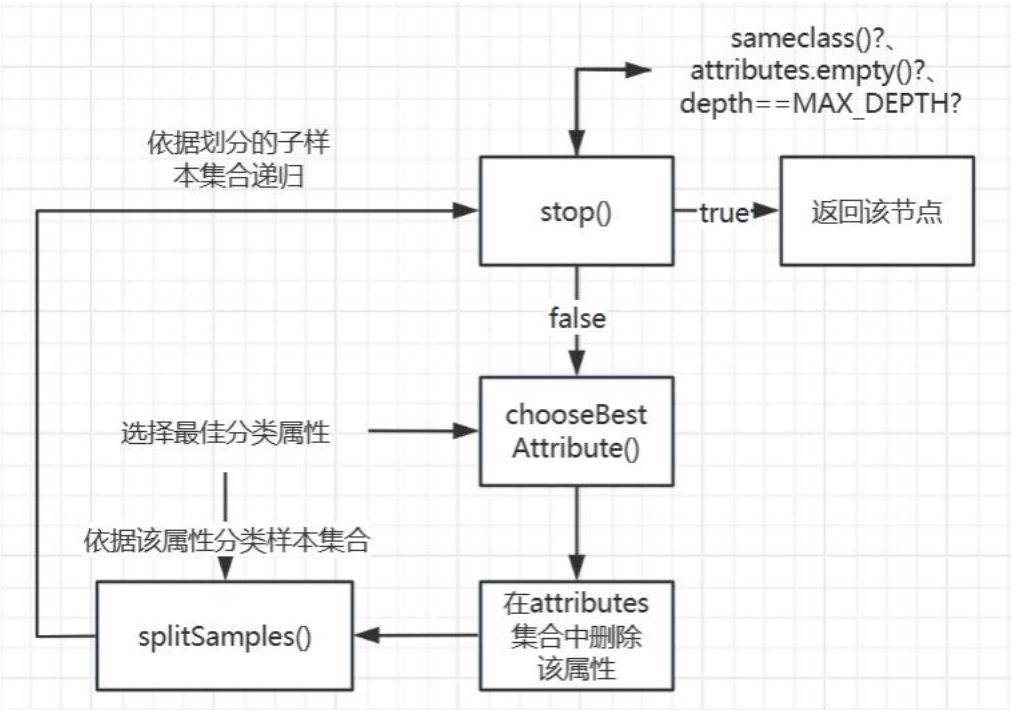
```
def predict(tree, attribute_values, attribute_names):
    if tree.is_leaf():
        return tree.classification

    # 找到当前属性在列表中的索引
    attribute_index = attribute_names.index(tree.attribute)
    attribute_value = attribute_values[attribute_index]

    # 如果在分支中找不到该值，则返回树中多数类作为默认预测
    if attribute_value not in tree.branches:
        # 返回节点下所有子节点的多数分类
        child_classes = [
            child.classification
            for child in tree.branches.values()
            if child.is_leaf()
        ]
        return get_majority_class([Sample( attributes: [], cls) for cls in child_classes])

    # 递归预测
    return predict(tree.branches[attribute_value], attribute_values, attribute_names)
```

calculate_accuracy(tree, samples, attribute_names)：遍历样本集，利用决策树预测每个样本的分类结果，与实际分类对比，计算分类准确率。

3.3 程序总流程



先计算信息熵关系挑选出最佳分类属性



选择了分类属性之后用这个最佳属性来分类样本。

4. 选择数据

选择了一个小样本数据和一个大数据。

```
def main1():
    training_data = [
        Sample( attributes: ["青绿", "蜷缩", "浊响", "清晰", "凹陷", "硬滑"], classification: "是"),
        Sample( attributes: ["乌黑", "蜷缩", "沉闷", "清晰", "凹陷", "硬滑"], classification: "是"),
        Sample( attributes: ["乌黑", "蜷缩", "浊响", "清晰", "凹陷", "硬滑"], classification: "是"),
        Sample( attributes: ["青绿", "稍蜷", "浊响", "清晰", "稍凹", "软粘"], classification: "是"),
        Sample( attributes: ["乌黑", "稍蜷", "浊响", "稍糊", "稍凹", "软粘"], classification: "是"),
        Sample( attributes: ["青绿", "硬挺", "清脆", "清晰", "平坦", "软粘"], classification: "否"),
        Sample( attributes: ["浅白", "稍蜷", "沉闷", "稍糊", "凹陷", "硬滑"], classification: "否"),
        Sample( attributes: ["乌黑", "稍蜷", "浊响", "清晰", "稍凹", "软粘"], classification: "否"),
        Sample( attributes: ["浅白", "蜷缩", "浊响", "模糊", "平坦", "硬滑"], classification: "否"),
        Sample( attributes: ["青绿", "蜷缩", "沉闷", "稍糊", "稍凹", "硬滑"], classification: "否"),
    ]
    test_data = [
        Sample( attributes: ["青绿", "蜷缩", "沉闷", "清晰", "凹陷", "硬滑"], classification: "是"),
        Sample( attributes: ["浅白", "蜷缩", "浊响", "清晰", "凹陷", "硬滑"], classification: "是"),
        Sample( attributes: ["乌黑", "稍蜷", "浊响", "清晰", "稍凹", "硬滑"], classification: "是"),
        Sample( attributes: ["乌黑", "稍蜷", "沉闷", "稍糊", "稍凹", "硬滑"], classification: "是"),
        Sample( attributes: ["浅白", "硬挺", "清脆", "模糊", "平坦", "硬滑"], classification: "否"),
        Sample( attributes: ["浅白", "蜷缩", "浊响", "模糊", "平坦", "软粘"], classification: "否"),
        Sample( attributes: ["青绿", "稍蜷", "浊响", "稍糊", "凹陷", "硬滑"], classification: "否"),
    ]
    attributes = ["色泽", "根蒂", "敲声", "纹理", "脐部", "触感"]

    decision_tree = build_decision_tree(training_data, attributes)
    train_accuracy = calculate_accuracy(decision_tree, training_data, attributes)
    test_accuracy = calculate_accuracy(decision_tree, test_data, attributes)

    print(f"训练集准确度: {train_accuracy}")
    print(f"测试集准确度: {test_accuracy}")
```

之后选择了一个大数据 nursery trainingdata

```
great_pret very_crit foster more critical convenient slightly_prob recommended spec_prior
great_pret very_crit foster more critical convenient slightly_prob priority spec_prior
great_pret very_crit foster more critical convenient slightly_prob not_recom not_recom
great_pret very_crit foster more critical convenient problematic recommended spec_prior
great_pret very_crit foster more critical convenient problematic priority spec_prior
great_pret very_crit foster more critical convenient problematic not_recom not_recom
great_pret very_crit foster more critical inconv nonprob recommended spec_prior
great_pret very_crit foster more critical inconv nonprob priority spec_prior
great_pret very_crit foster more critical inconv nonprob not_recom not_recom
great_pret very_crit foster more critical inconv slightly_prob recommended spec_prior
great_pret very_crit foster more critical inconv slightly_prob priority spec_prior
great_pret very_crit foster more critical inconv slightly_prob not_recom not_recom
great_pret very_crit foster more critical inconv problematic recommended spec_prior
great_pret very_crit foster more critical inconv problematic priority spec_prior
great_pret very_crit foster more critical inconv problematic not_recom not_recom
```

```python
def main2():
    # 读取数据文件
    filename = "nursery_trainingdata.txt"
    training_data = []
    test_data = []
    attributes = ["parents", "has_nurs", "form", "children", "housing", "finance", "social", "hea

    with open(filename, "r") as file:
        lines = file.readlines()
        for i, line in enumerate(lines):
            # 跳过空行和无效行
            parts = line.strip().split()
            if len(parts) < len(attributes) + 1:  # 属性值 + 类别至少应有 9 列
                continue
            attributes_values = parts[:-1]  # 属性值
            classification = parts[-1]  # 类别
            sample = Sample(attributes_values, classification)
            if i % 5 == 0:  # 20% 数据作为测试集
                test_data.append(sample)
            else:  # 80% 数据作为训练集
                training_data.append(sample)

    # 检查是否成功加载数据
    if not training_data:
        print("训练集为空，请检查数据文件内容。")
        return
    if not test_data:
        print("测试集为空，请检查数据文件内容。")
        return
```

来训练。

5. 结果分析

。训练结果如下：

小数据：

```
D:\anaconda3\envs\yolo\python.exe D:\cv\cv-computer-visual\4.cv中等\机器学习\决策树ID3算法分类\ID3-classification.py
训练集准确度: 0.7
测试集准确度: 0.7142857142857143

Process finished with exit code 0
```

大数据：

```
D:\anaconda3\envs\yolo\python.exe D:\cv\cv-computer-visual\4.cv中等\机器学习\决策树ID3算法分类\ID3-classification.py
训练集准确度: 0.8551130678407044
测试集准确度: 0.8531412565026011

Process finished with exit code 0
```

综合以上两个案例:case 1:西瓜的小样例(共 10 个训练样例);case 2:nursery 的大样例(共 12000 个训练样例),发现: .当样例较小时,生成的决策树对于训练集的准确度很高,但此时的决策树的泛化能力显然较弱;当样例较大时,生成的决策树对于训练集的准确度并不是很高,但此时该决策树的泛化能力较强;

上述训练样例符合决策树的特点：容易过拟合。

6. 总结

本实验根据 ID3 算法构建了一棵基本的决策树;构建决策树包括构建了相关的数据类型(结构体),构建了生成决策树的 buildDecisionTree()等函数,同时构建了一些利用生成的决策树生成数据的函数,如 predictData()(计算准确率);

待改进: 1.采用了 ID3 算法,故比较难以处理连续的特征,如:工资、长度等。同时 ID3 算法决策的核心是信息增益(贪心),故其会倾向于选择取值较多的属性,故可能并不是整体上的最优解;2.本程序并未考虑到缺失值的问题,留待后续学习了 ID4.5 后自行改进;