实验三

实验要求

实验目的: **1、理解决策树算法的原理;** **2、能够使用决策树算法处理具体问题;** **3、能够使用Python语言实现决策树算法。**
实验内容: 1、 读取数据文件iris.data，使用随机种子将原始数据集打乱，将数据集划分为训练集和测试集，比例为8：2; 2、 使用ID3决策树在训练集上进行建模（注意：ID3决策树只能处理离散型数据，如果输入属性为连续型需要进行属性离散化），并将决策树通过图片的形式进行展现（使用Graphviz工具); 3、 使用测试集对2中构建好的决策树进行检验，并计算其准确率; 4、 使用C4.5算法在训练集上进行建模，并将决策树通过图片的形式进行展现（使用Graphviz工具); 5、 使用测试集对4中构建好的决策树进行检验，并计算其准确率 6、 读取给定的数据集watermelon.txt，构建其决策树，并通过图片的形式展现（使用Graphviz工具); 7、 将上述实验内容的核心代码及实验结果截图放到“实验过程及分析”中。

实验过程以及分析

1. 数据准备

实验描述

读取数据文件iris.data，使用随机种子将原始数据集打乱，将数据集划分为训练集和测试集，比例为8：2;

代码实现

首先，我们实现了加载和处理Iris数据集的函数：

- `load_iris_data()`：读取数据，设置随机种子打乱数据，并按8:2的比例划分训练集和测试集
- `discretize_features()`：由于ID3算法不能直接处理连续型数据，此函数将连续特征离散化为几个区间

```python
# 第1步：读取数据文件iris.data，打乱并划分数据集
def load_iris_data():
    # 读取数据
    column_names = ['sepal_length', 'sepal_width', 'petal_length',
'petal_width', 'class']
    iris_data = pd.read_csv('iris.data', header=None, names=column_names)

    # 打乱数据（设置随机种子确保结果可复现）
    iris_data = iris_data.sample(frac=1,
random_state=42).reset_index(drop=True)

    # 分离特征和标签
    X = iris_data.iloc[:, :-1]
    y = iris_data.iloc[:, -1]

    # 划分训练集和测试集 (8:2)    X_train, X_test, y_train, y_test =
train_test_split(X, y, test_size=0.2, random_state=42)

    return X_train, X_test, y_train, y_test, iris_data


# 属性离散化函数（用于ID3算法）
def discretize_features(X_train, X_test, n_bins=3):
    X_train_discrete = X_train.copy()
    X_test_discrete = X_test.copy()

    for column in X_train.columns:
        # 使用训练集的数据来确定分箱边界
        bins = pd.qcut(X_train[column], n_bins, duplicates='drop',
retbins=True)[1]

        # 对训练集进行离散化
        X_train_discrete[column] = pd.cut(X_train[column], bins=bins,
labels=False, include_lowest=True)

        # 对测试集进行离散化（使用与训练集相同的分箱边界）
        X_test_discrete[column] = pd.cut(X_test[column], bins=bins,
labels=False, include_lowest=True)

    return X_train_discrete, X_test_discrete
```

2. ID3决策树算法实现

实验描述

使用ID3决策树在训练集上进行建模（注意：ID3决策树只能处理离散型数据，如果输入属性为连续型需要进行属性离散化），并将决策树通过图片的形式进行展现（使用Graphviz工具）;

代码实现

ID3决策树类(`ID3DecisionTree`)实现了以下核心功能:

- `_entropy()`: 计算熵
- `_information_gain()`: 计算信息增益
- `_find_best_feature()`: 找到最佳分裂特征
- `_build_tree()`: 递归构建决策树
- `predict()`: 使用决策树进行预测
- `visualize()`: 使用Graphviz可视化决策树
  ID3算法基于信息增益选择最佳分裂特征，对于每个节点，都选择信息增益最大的特征进行分裂。

```
# ID3决策树算法实现
class ID3DecisionTree:
    def __init__(self, max_depth=None):
        self.max_depth = max_depth
        self.tree = None

    def fit(self, X, y):
        feature_names = X.columns
        self.tree = self._build_tree(X, y, feature_names, depth=0)
        return self

    def _entropy(self, y):
        """计算熵"""
        counts = Counter(y)
        entropy = 0
        for label in counts:
            p = counts[label] / len(y)
            entropy -= p * math.log2(p)
        return entropy

    def _information_gain(self, X, y, feature_name):
        """计算信息增益"""
        entropy_parent = self._entropy(y)

        # 计算条件熵
        values = X[feature_name].unique()
```

```
        weighted_entropy = 0

        for value in values:
            subset_indices = X[feature_name] == value
            subset_y = y[subset_indices]
            weight = len(subset_y) / len(y)
            weighted_entropy += weight * self._entropy(subset_y)

        # 信息增益 = 父节点熵 - 条件熵
        information_gain = entropy_parent - weighted_entropy
        return information_gain

    def _find_best_feature(self, X, y, feature_names):
        """找到最佳分裂特征"""
        best_gain = -1
        best_feature = None

        for feature in feature_names:
            gain = self._information_gain(X, y, feature)
            if gain > best_gain:
                best_gain = gain
                best_feature = feature

        return best_feature

    def _build_tree(self, X, y, feature_names, depth):
        # 基本情况：所有样本属于同一类别
        if len(np.unique(y)) == 1:
            return {'type': 'leaf', 'label': y.iloc[0]}

        # 没有特征可分裂或达到最大深度
        if len(feature_names) == 0 or (self.max_depth is not None and
depth >= self.max_depth):
            most_common_label = y.value_counts().idxmax()
            return {'type': 'leaf', 'label': most_common_label}

        # 找到最佳分裂特征
        best_feature = self._find_best_feature(X, y, feature_names)

        # 如果无法找到有效的特征进行分裂
        if best_feature is None:
            most_common_label = y.value_counts().idxmax()
            return {'type': 'leaf', 'label': most_common_label}

        # 创建当前节点
        tree = {'type': 'node', 'feature': best_feature, 'children': {}}

        # 根据特征值分裂
        for value in X[best_feature].unique():
            mask = X[best_feature] == value
```

```
            subset_X = X[mask].drop(best_feature, axis=1)
            subset_y = y[mask]
            remaining_features = [f for f in feature_names if f != best_feature]

            # 如果子集为空
            if len(subset_X) == 0:
                most_common_label = y.value_counts().idxmax()
                tree['children'][value] = {'type': 'leaf', 'label': most_common_label}
            else:
                tree['children'][value] = self._build_tree(subset_X, subset_y, remaining_features, depth + 1)

        return tree

    def predict(self, X):
        if self.tree is None:
            raise Exception("Model not trained yet!")

        predictions = []
        for _, sample in X.iterrows():
            predictions.append(self._predict_sample(sample, self.tree))

        return predictions

    def _predict_sample(self, sample, tree):
        if tree['type'] == 'leaf':
            return tree['label']

        feature = tree['feature']
        value = sample[feature]

        # 处理测试集中出现训练集中没有的值的情况
        if value not in tree['children']:
            # 返回所有子节点中多数类
            labels = [self._predict_sample(sample, child) for child in tree['children'].values()]
            return max(set(labels), key=labels.count)

        return self._predict_sample(sample, tree['children'][value])

    def visualize(self, feature_names, class_names=None):
        """可视化决策树"""
        dot = graphviz.Digraph(comment='Decision Tree')

        # 递归添加节点
        self._add_nodes(dot, self.tree, "0", feature_names, class_names)

        return dot
```

```
    def _add_nodes(self, dot, tree, node_id, feature_names, class_names,
parent_id=None, edge_label=None):
        if tree['type'] == 'leaf':
            label = str(tree['label']) if class_names is None else
class_names[tree['label']]
            dot.node(node_id, label=f"Class: {label}", shape='box')
        else:
            dot.node(node_id, label=f"{tree['feature']}")

            # 添加子节点
            for i, (value, child) in enumerate(tree['children'].items()):
                child_id = f"{node_id}_{i}"
                self._add_nodes(dot, child, child_id, feature_names,
class_names, node_id, str(value))

        # 连接父节点和当前节点
        if parent_id is not None:
            dot.edge(parent_id, node_id, label=edge_label)

        return dot
```

3. C4.5决策树算法实现

实验描述

使用C4.5算法在训练集上进行建模，并将决策树通过图片的形式进行展现（使用Graphviz工具）;

代码实现

C4.5决策树类(`C45DecisionTree`)是ID3的改进版，主要区别在于:

- 使用信息增益率(`_information_gain_ratio()`)而非信息增益来选择分裂特征
- 能够处理连续特征(`_handle_continuous_feature()`)，找到最佳分割阈值

```
class C45DecisionTree:
    def __init__(self, max_depth=None):
        self.max_depth = max_depth
        self.tree = None

    def fit(self, X, y):
        feature_names = X.columns
        self.tree = self._build_tree(X, y, feature_names, depth=0)
        return self

    def _entropy(self, y):
        """计算熵"""
```

```
        counts = Counter(y)
        entropy = 0
        for label in counts:
            p = counts[label] / len(y)
            entropy -= p * math.log2(p)
        return entropy

    def _information_gain_ratio(self, X, y, feature_name):
        """计算信息增益率"""
        # 计算信息增益
        entropy_parent = self._entropy(y)

        values = X[feature_name].unique()
        weighted_entropy = 0

        # 特征的固有信息
        intrinsic_info = 0

        for value in values:
            subset_indices = X[feature_name] == value
            subset_y = y[subset_indices]
            weight = len(subset_y) / len(y)
            weighted_entropy += weight * self._entropy(subset_y)

            # 计算特征的固有信息
            intrinsic_info -= weight * math.log2(weight)

        # 信息增益和增益率
        info_gain = entropy_parent - weighted_entropy

        # 避免除以零
        if intrinsic_info == 0:
            return 0

        gain_ratio = info_gain / intrinsic_info
        return gain_ratio

    def _handle_continuous_feature(self, X, y, feature_name):
        """处理连续特征，找到最佳分割点"""
        # 获取排序后的唯一值
        unique_values = sorted(X[feature_name].unique())

        best_gain_ratio = -1
        best_threshold = None

        # 尝试每对相邻值的中点作为阈值
        for i in range(len(unique_values) - 1):
            threshold = (unique_values[i] + unique_values[i + 1]) / 2

            # 创建二值特征
```

```
            X_temp = X.copy()
            X_temp[f"{feature_name}_binary"] = X[feature_name] <= threshold

            # 计算这个二值特征的信息增益率
            gain_ratio = self._information_gain_ratio(X_temp, y, f"{feature_name}_binary")

            if gain_ratio > best_gain_ratio:
                best_gain_ratio = gain_ratio
                best_threshold = threshold

        return best_threshold, best_gain_ratio

    def _find_best_feature(self, X, y, feature_names):
        """找到最佳分裂特征"""
        best_gain_ratio = -1
        best_feature = None
        best_threshold = None
        is_continuous = False

        for feature in feature_names:
            # 检查是否为连续特征（值的数量大于某个阈值）
            if len(X[feature].unique()) > 10:  # 假设连续特征有很多唯一值
                threshold, gain_ratio = self._handle_continuous_feature(X, y, feature)
                if gain_ratio > best_gain_ratio:
                    best_gain_ratio = gain_ratio
                    best_feature = feature
                    best_threshold = threshold
                    is_continuous = True
            else:
                # 离散特征
                gain_ratio = self._information_gain_ratio(X, y, feature)
                if gain_ratio > best_gain_ratio:
                    best_gain_ratio = gain_ratio
                    best_feature = feature
                    best_threshold = None
                    is_continuous = False

        return best_feature, best_threshold, is_continuous

    def _build_tree(self, X, y, feature_names, depth):
        # 基本情况：所有样本属于同一类别
        if len(np.unique(y)) == 1:
            return {'type': 'leaf', 'label': y.iloc[0]}

        # 没有特征可分裂或达到最大深度
        if len(feature_names) == 0 or (self.max_depth is not None and depth >= self.max_depth):
```

```
            most_common_label = y.value_counts().idxmax()
            return {'type': 'leaf', 'label': most_common_label}

        # 找到最佳分裂特征
        best_feature, threshold, is_continuous =
self._find_best_feature(X, y, feature_names)

        # 如果无法找到有效的特征进行分裂
        if best_feature is None:
            most_common_label = y.value_counts().idxmax()
            return {'type': 'leaf', 'label': most_common_label}

        # 创建当前节点
        if is_continuous:
            tree = {
                'type': 'node',
                'feature': best_feature,
                'is_continuous': True,
                'threshold': threshold,
                'children': {}
            }

            # 根据阈值分裂
            # 小于等于阈值的子集
            left_mask = X[best_feature] <= threshold
            left_X = X[left_mask]
            left_y = y[left_mask]

            # 大于阈值的子集
            right_mask = ~left_mask
            right_X = X[right_mask]
            right_y = y[right_mask]

            # 为左子树构建子树
            if len(left_X) == 0:
                most_common_label = y.value_counts().idxmax()
                tree['children']['<='] = {'type': 'leaf', 'label':
most_common_label}
            else:
                tree['children']['<='] = self._build_tree(left_X, left_y,
feature_names, depth + 1)

            # 为右子树构建子树
            if len(right_X) == 0:
                most_common_label = y.value_counts().idxmax()
                tree['children']['>'] = {'type': 'leaf', 'label':
most_common_label}
            else:
                tree['children']['>'] = self._build_tree(right_X, right_y,
feature_names, depth + 1)
```

```
        else:
            # 离散特征
            tree = {
                'type': 'node',
                'feature': best_feature,
                'is_continuous': False,
                'children': {}
            }

            # 根据特征值分裂
            for value in X[best_feature].unique():
                mask = X[best_feature] == value
                subset_X = X[mask]
                subset_y = y[mask]

                # 如果子集为空
                if len(subset_X) == 0:
                    most_common_label = y.value_counts().idxmax()
                    tree['children'][value] = {'type': 'leaf', 'label': most_common_label}
                else:
                    tree['children'][value] = self._build_tree(subset_X, subset_y, feature_names, depth + 1)

        return tree

    def predict(self, X):
        if self.tree is None:
            raise Exception("Model not trained yet!")

        predictions = []
        for _, sample in X.iterrows():
            predictions.append(self._predict_sample(sample, self.tree))

        return predictions

    def _predict_sample(self, sample, tree):
        if tree['type'] == 'leaf':
            return tree['label']

        feature = tree['feature']

        if tree.get('is_continuous', False):
            value = sample[feature] <= tree['threshold']
            key = '<=' if value else '>'
        else:
            value = sample[feature]
            key = value

        # 处理测试集中出现训练集中没有的值的情况
```

```python
        if key not in tree['children']:
            # 返回所有子节点中多数类
            labels = [self._predict_sample(sample, child) for child in tree['children'].values()]
            return max(set(labels), key=labels.count)

        return self._predict_sample(sample, tree['children'][key])

    def visualize(self, feature_names, class_names=None):
        """可视化决策树"""
        dot = graphviz.Digraph(comment='Decision Tree')

        # 递归添加节点
        self._add_nodes(dot, self.tree, "0", feature_names, class_names)

        return dot

    def _add_nodes(self, dot, tree, node_id, feature_names, class_names, parent_id=None, edge_label=None):
        if tree['type'] == 'leaf':
            label = str(tree['label']) if class_names is None else class_names[tree['label']]
            dot.node(node_id, label=f"Class: {label}", shape='box')
        else:
            if tree.get('is_continuous', False):
                dot.node(node_id, label=f"{tree['feature']} <= {tree['threshold']:.2f}")
            else:
                dot.node(node_id, label=f"{tree['feature']}")

            # 添加子节点
            for i, (value, child) in enumerate(tree['children'].items()):
                child_id = f"{node_id}_{i}"
                self._add_nodes(dot, child, child_id, feature_names, class_names, node_id, str(value))

        # 连接父节点和当前节点
        if parent_id is not None:
            dot.edge(parent_id, node_id, label=edge_label)

        return dot
```

4. 评估和可视化

实验描述

1. 使用测试集对2中构建好的决策树进行检验，并计算其准确率
2. 使用测试集对4中构建好的决策树进行检验，并计算其准确率

代码实现

- `evaluate_model()`：计算模型的预测准确率
- 每个决策树类都有`visualize()`方法，使用Graphviz库将决策树结构可视化为图像

```
# 评估函数
def evaluate_model(y_true, y_pred):
    """计算准确率"""
    correct = sum(y_true == y_pred)
    return correct / len(y_true)
```

5. 西瓜数据集处理

实验描述

读取给定的数据集watermelon.txt，构建其决策树，并通过图片的形式展现（使用Graphviz工具）；

代码实现

- `load_watermelon_data()`：加载西瓜数据集
- 使用与Iris数据集相同的算法在西瓜数据集上构建决策树

```
# 西瓜数据集处理
def load_watermelon_data():
    """加载西瓜数据集"""
    try:
        # 使用中文逗号作为分隔符
        watermelon_data = pd.read_csv('./data/watermelon.txt', sep='，',
encoding='utf-8')

        print(f"成功加载西瓜数据集：{watermelon_data.shape[0]}行 x
{watermelon_data.shape[1]}列")
        print("数据列名:", list(watermelon_data.columns))

        # 检查目标变量列
        if '质量' in watermelon_data.columns:
            target_column = '质量'
        elif '好瓜' in watermelon_data.columns:
            target_column = '好瓜'
        else:
            target_column = watermelon_data.columns[-1]  # 默认使用最后一列

        print(f"目标变量列名为：{target_column}")
        print("目标变量分布:")
        print(watermelon_data[target_column].value_counts())
```

```
        # 分离特征和标签
        X = watermelon_data.drop(target_column, axis=1)
        y = watermelon_data[target_column]

        return X, y, watermelon_data

    except Exception as e:
        print(f"加载西瓜数据集时出错: {str(e)}")
        print("错误详情:", e)
        try:
            # 尝试读取文件内容进行调试
            with open('watermelon.txt', 'r', encoding='utf-8') as f:
                content = f.readlines()[:5]  # 读取前5行
            print("文件内容前5行:")
            for line in content:
                print(line.strip())
        except Exception as read_error:
            print(f"读取文件失败: {read_error}")

        return None, None, None
```

主函数（实验流程）

描述

主函数将以上封装的代码组装好，并构建了整个完整的实验流程。

代码实现

```
# 主函数
def main():
    # 加载Iris数据集
    print("加载Iris数据集...")
    X_train, X_test, y_train, y_test, iris_data = load_iris_data()
    print(f"训练集大小: {len(X_train)}, 测试集大小: {len(X_test)}")

    # 为ID3算法离散化数据
    print("\n离散化特征(用于ID3算法)...")
    X_train_discrete, X_test_discrete = discretize_features(X_train,
X_test)

    # ID3决策树
    print("\n使用ID3算法构建决策树...")
    id3_tree = ID3DecisionTree(max_depth=5)
    id3_tree.fit(X_train_discrete, y_train)

    # 可视化ID3决策树
    print("可视化ID3决策树...")
```

```
    dot_id3 = id3_tree.visualize(X_train.columns)
    dot_id3.render('id3_tree', format='png', cleanup=True)
    print("ID3决策树已保存为'id3_tree.png'")

    # 评估ID3决策树
    print("\n评估ID3决策树...")
    id3_predictions = id3_tree.predict(X_test_discrete)
    id3_accuracy = evaluate_model(y_test.values, id3_predictions)
    print(f"ID3决策树在测试集上的准确率: {id3_accuracy:.4f}")

    # C4.5决策树
    print("\n使用C4.5算法构建决策树...")
    c45_tree = C45DecisionTree(max_depth=5)
    c45_tree.fit(X_train, y_train)

    # 可视化C4.5决策树
    print("可视化C4.5决策树...")
    dot_c45 = c45_tree.visualize(X_train.columns)
    dot_c45.render('c45_tree', format='png', cleanup=True)
    print("C4.5决策树已保存为'c45_tree.png'")

    # 评估C4.5决策树
    print("\n评估C4.5决策树...")
    c45_predictions = c45_tree.predict(X_test)
    c45_accuracy = evaluate_model(y_test.values, c45_predictions)
    print(f"C4.5决策树在测试集上的准确率: {c45_accuracy:.4f}")

    # 西瓜数据集
    print("\n处理西瓜数据集...")
    try:
        X_watermelon, y_watermelon, watermelon_data = 
load_watermelon_data()

        # 构建决策树（使用ID3，因为西瓜数据集主要是离散特征）
        print("为西瓜数据集构建决策树...")
        watermelon_tree = ID3DecisionTree(max_depth=5)
        watermelon_tree.fit(X_watermelon, y_watermelon)

        # 可视化西瓜数据集决策树
        print("可视化西瓜数据集决策树...")
        dot_watermelon = watermelon_tree.visualize(X_watermelon.columns)
        dot_watermelon.render('watermelon_tree', format='png', 
cleanup=True)
        print("西瓜数据集决策树已保存为'watermelon_tree.png'")
    except Exception as e:
        print(f"处理西瓜数据集时出错: {e}")
        print("请确保'watermelon.txt'文件存在且格式正确")
```

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

实验结果

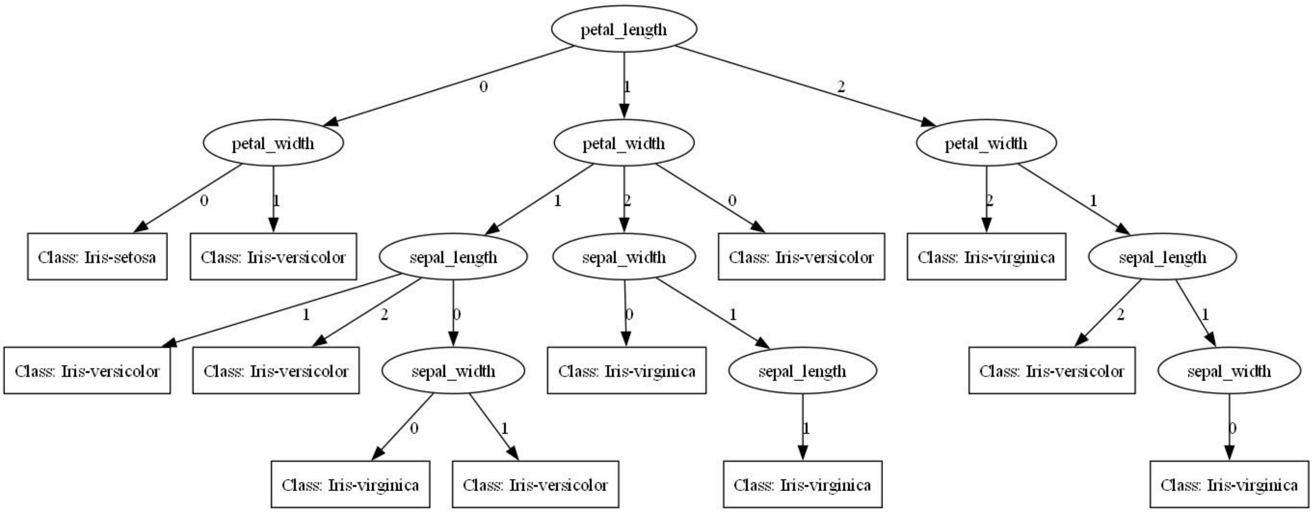
终端输出

```
C:\Users\19065\miniconda3\python.exe D:\coding\简简单单挖掘个数据
\exp03\main.py
加载Iris数据集...
训练集大小: 120, 测试集大小: 30

离散化特征(用于ID3算法)...

使用ID3算法构建决策树...
可视化ID3决策树...
ID3决策树已保存为'id3_tree.png'

评估ID3决策树...
ID3决策树在测试集上的准确率: 0.9333

使用C4.5算法构建决策树...
可视化C4.5决策树...
D:\coding\简简单单挖掘个数据\exp03\main.py:440: ParserWarning: Falling back
to the 'python' engine because the separator encoded in utf-8 is > 1 char
long, and the 'c' engine does not support such separators; you can avoid
this warning by specifying engine='python'.
  watermelon_data = pd.read_csv('./data/watermelon.txt', sep=', ',
encoding='utf-8')
C4.5决策树已保存为'c45_tree.png'

评估C4.5决策树...
C4.5决策树在测试集上的准确率: 1.0000

处理西瓜数据集...
成功加载西瓜数据集: 17行 x 7列
数据列名: ['色泽', '根蒂', '敲声', '纹理', '脐部', '触感', '质量']
目标变量列名为: 质量
目标变量分布:
质量
否    9
是    8
Name: count, dtype: int64
```

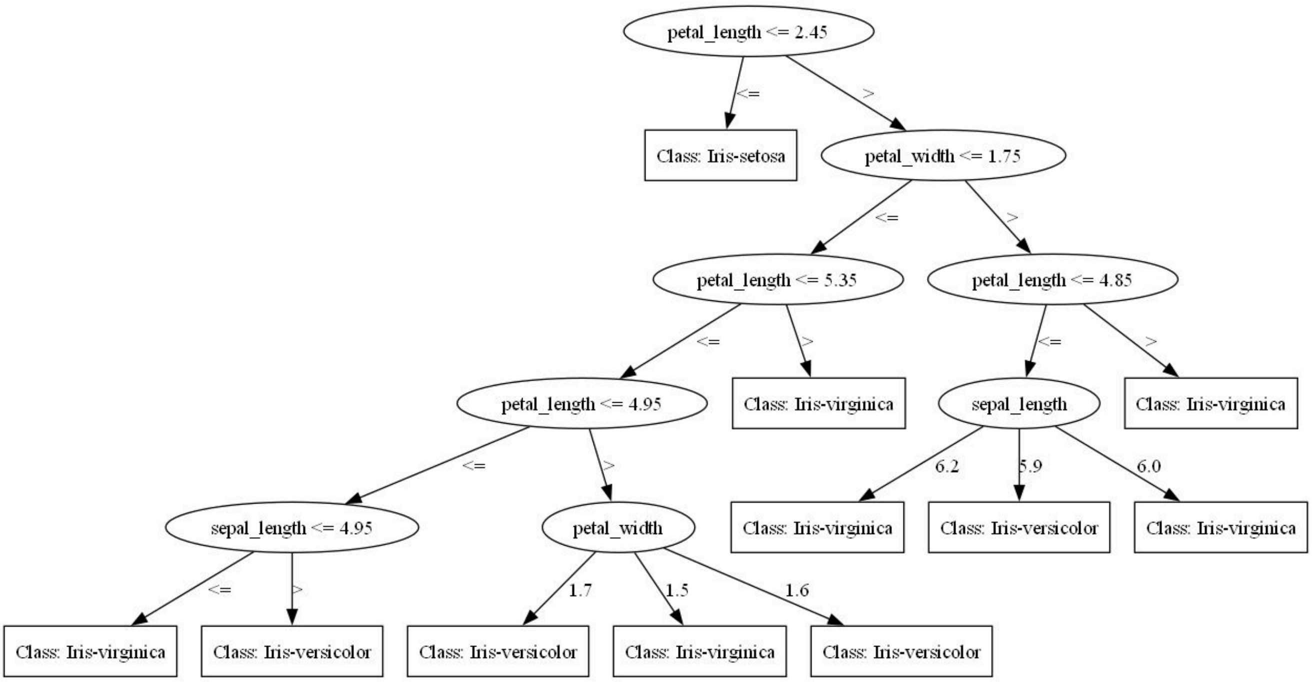
```
为西瓜数据集构建决策树...
可视化西瓜数据集决策树...
西瓜数据集决策树已保存为'watermelon_tree.png'

进程已结束，退出代码为 0
```

图片输出

id3_tree

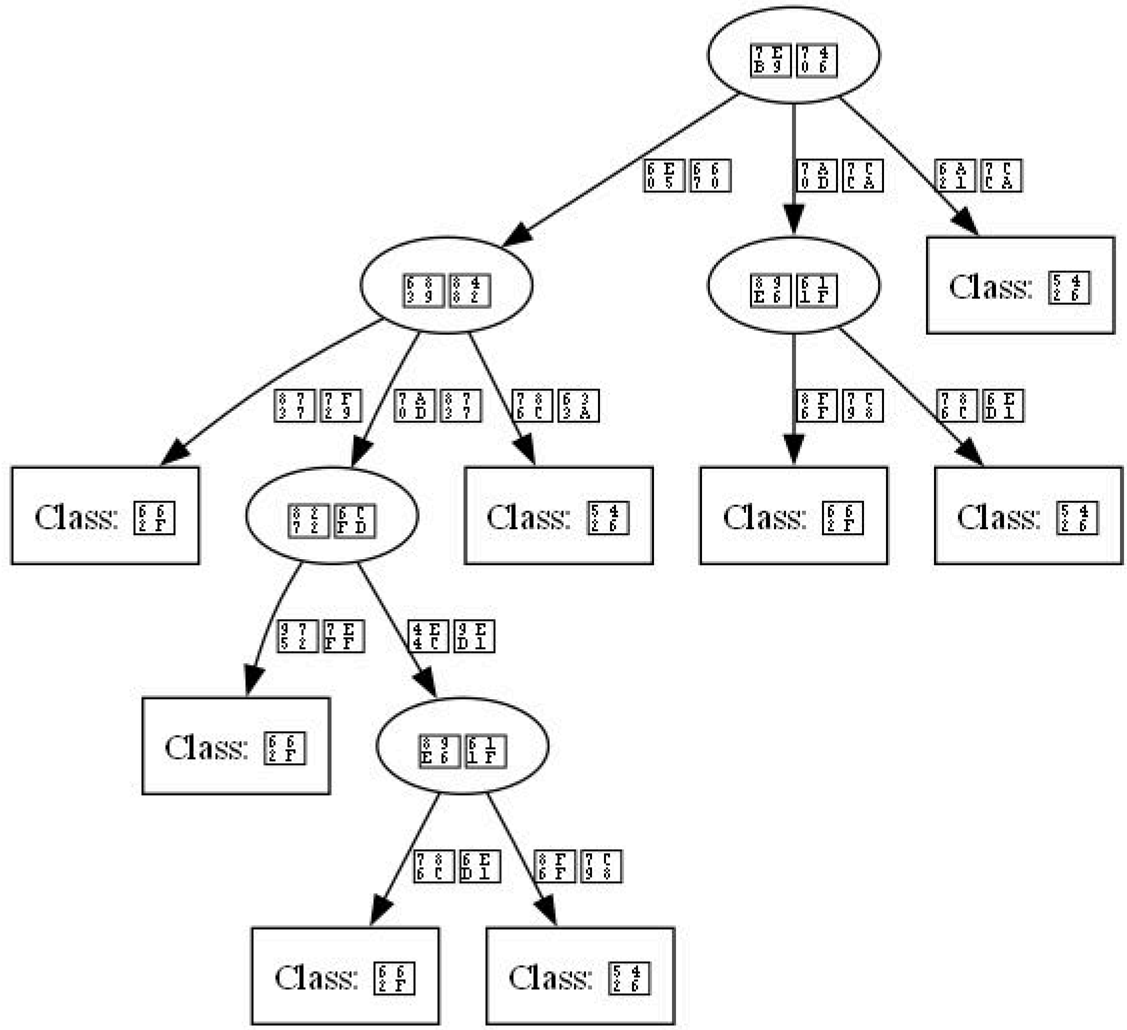


c45_tree



西瓜数据集

> 不支持中文我是真没办法，真懒得改了 O。o



实验体会

通过本次实验，我深入理解了决策树算法，掌握了ID3和C4.5的原理与实现。使用Iris和西瓜数据集，我学会了数据预处理，包括数据集划分和特征离散化，尤其为ID3准备离散数据。模型构建中，我实现了两种算法，并通过Graphviz可视化决策树，直观理解其工作原理。评估时，C4.5在处理连续特征时表现更优。实验中，面对特征离散化和数据问题的挑战，我通过调试解决，提升了问题解决能力。未来，我计划探索剪枝技术和缺失值处理，以优化模型性能。这次实验不仅让我掌握决策树算法，还增强了编程和分析能力，为后续学习和工作奠定了基础。