1、 下列说法正确的是？

- [x] A、训练决策树的过程就是构建决策树的过程
- [x] B、ID3算法是根据信息增益来构建决策树
- [ ] C、C4.5算法是根据基尼系数来构建决策树
- [ ] D、决策树模型的可理解性不高

2、 下列说法错误的是？

- ○ A、从树的根节点开始，根据特征的值一步一步走到叶子节点的过程是决策树做决策的过程
- ◉ B、决策树只能是一棵二叉树
- ○ C、根节点所代表的特征是最优特征

```python
import numpy as np
def calcInfoEntropy(feature, label):
    '''
    计算信息熵
    :param feature:数据集中的特征，类型为ndarray
    :param label:数据集中的标签，类型为ndarray
    :return:信息熵，类型float
    '''
    label_set = set(label)
    result=0
    for l in label_set:
        count = 0
        for j in range(len(label)):
            if label[j] == l:
                count += 1
        p = count/len(label)
        result+=-p*np.log2(p)

    return result

def calcHDA(feature, label, index, value):
    '''
    计算信息熵
    :param feature:数据集中的特征，类型为ndarray
    :param label:数据集中的标签，类型为ndarray
    :param index:需要使用的特征列索引，类型为int
    :param value:index所表示的特征列中需要考察的特征值，类型为int
```

```
    :return:信息熵，类型float
    '''
    count = 0
    sub_feature = []#list
    sub_label = []
    for i in range(len(feature)):
        if feature[i][index] == value:
            count += 1
            sub_feature.append(feature[i])
            sub_label.append(label[i])
    pHA = count/len(feature)
    e = calcInfoEntropy(sub_feature,sub_label)
    return pHA*e

def calcInfoGain(feature, label, index):
    '''
    计算信息增益
    :param feature:测试用例中字典里的feature
    :param label:测试用例中字典里的label
    :param index:测试用例中字典里的index，即feature部分特征列的索引
    :return:信息增益，类型float
    '''
    base_e = calcInfoEntropy(feature,label)
    f = np.array(feature)
    f_set = set(f[:, index])
    sum_HDA = 0
    for a in f_set:
        sum_HDA += calcHDA(feature, label, index, a)
    return base_e-sum_HDA
```

ID3

```
import numpy as np

class DecisionTree(object):
    def __init__(self):
        self.tree = {}
    def calcInfoGain(self, feature, label, index):
        '''
        计算信息增益
```

```
        :param feature:测试用例中字典里的feature，类型为ndarray
        :param label:测试用例中字典里的label，类型为ndarray
        :param index:测试用例中字典里的index，即feature部分特征列的索引。该
索引指的是feature中第几个特征，如index:0表示使用第一个特征来计算信息增益。
        :return:信息增益，类型float
        '''
        def calcInfoEntropy(label):
            '''
            计算信息熵
            :param label:数据集中的标签，类型为ndarray
            :return:信息熵，类型float
            '''
            label_set = set(label)
            result = 0
            for l in label_set:
                count = 0
                for j in range(len(label)):
                    if label[j] == l:
                        count += 1
                p = count / len(label)
                result -= p * np.log2(p)
            return result
        def calcHDA(feature, label, index, value):
            '''
            计算信息熵
            :param feature:数据集中的特征，类型为ndarray
            :param label:数据集中的标签，类型为ndarray
            :param index:需要使用的特征列索引，类型为int
            :param value:index所表示的特征列中需要考察的特征值，类型为int
            :return:信息熵，类型float
            '''
            count = 0
            sub_feature = []
            sub_label = []
            for i in range(len(feature)):
                if feature[i][index] == value:
                    count += 1
                    sub_feature.append(feature[i])
                    sub_label.append(label[i])
            pHA = count / len(feature)
            e = calcInfoEntropy(sub_label)
            return pHA * e
```

```python
        base_e = calcInfoEntropy(label)
        f = np.array(feature)
        f_set = set(f[:, index])
        sum_HDA = 0
        for value in f_set:
            sum_HDA += calcHDA(feature, label, index, value)
        return base_e - sum_HDA

    def getBestFeature(self, feature, label):
        max_infogain = 0
        best_feature = 0
        for i in range(len(feature[0])):
            infogain = self.calcInfoGain(feature, label, i)
            if infogain > max_infogain:
                max_infogain = infogain
                best_feature = i
        return best_feature

    def createTree(self, feature, label):
        if len(set(label)) == 1:
            return label[0]
        if len(feature[0]) == 1 or len(np.unique(feature, axis=0)) == 1:
            vote = {}
            for l in label:
                if l in vote.keys():
                    vote[l] += 1
                else:
                    vote[l] = 1
            max_count = 0
            vote_label = None
            for k, v in vote.items():
                if v > max_count:
                    max_count = v
                    vote_label = k
            return vote_label

        best_feature = self.getBestFeature(feature, label)
        tree = {best_feature: {}}
        f = np.array(feature)
        f_set = set(f[:, best_feature])
```

```
        for v in f_set:
            sub_feature = []
            sub_label = []
            for i in range(len(feature)):
                if feature[i][best_feature] == v:
                    sub_feature.append(feature[i])
                    sub_label.append(label[i])
            tree[best_feature][v] = self.createTree(sub_feature,
sub_label)
        return tree

    def fit(self, feature, label):
        '''
        :param feature: 训练集数据，类型为ndarray
        :param label:训练集标签，类型为ndarray
        :return: None
        '''
        self.tree = self.createTree(feature, label)

    def predict(self, feature):
        '''
        :param feature:测试集数据，类型为ndarray
        :return:预测结果，如np.array([0, 1, 2, 2, 1, 0])
        '''
        result = []

        def classify(tree, feature):
            if not isinstance(tree, dict):
                return tree
            t_index, t_value = list(tree.items())[0]
            f_value = feature[t_index]
            if isinstance(t_value, dict):
                classLabel = classify(tree[t_index][f_value],
feature)
                return classLabel
            else:
                return t_value

        for f in feature:
            result.append(classify(self.tree, f))

        return np.array(result)
```

信息增益率

```
import numpy as np

def calcInfoGain(feature, label, index):
    '''
    计算信息增益
    :param feature:测试用例中字典里的feature，类型为ndarray
    :param label:测试用例中字典里的label，类型为ndarray
    :param index:测试用例中字典里的index，即feature部分特征列的索引。该索引指的是feature中第几个特征，如index:0表示使用第一个特征来计算信息增益。
    :return:信息增益，类型float
    '''

    def calcInfoEntropy(label):
        '''
        计算信息熵
        :param label:数据集中的标签，类型为ndarray
        :return:信息熵，类型float
        '''

        label_set = set(label)
        result = 0
        for l in label_set:
            count = 0
            for j in range(len(label)):
                if label[j] == l:
                    count += 1
            p = count / len(label)
            result -= p * np.log2(p)
        return result

    def calcHDA(feature, label, index, value):
        '''
        计算信息熵
        :param feature:数据集中的特征，类型为ndarray
        :param label:数据集中的标签，类型为ndarray
        :param index:需要使用的特征列索引，类型为int
        :param value:index所表示的特征列中需要考察的特征值，类型为int
        :return:信息熵，类型float
```

```
        '''
        count = 0
        sub_feature = []
        sub_label = []
        for i in range(len(feature)):
            if feature[i][index] == value:
                count += 1
                sub_feature.append(feature[i])
                sub_label.append(label[i])
        pHA = count / len(feature)
        e = calcInfoEntropy(sub_label)
        return pHA * e

    base_e = calcInfoEntropy(label)
    f = np.array(feature)
    f_set = set(f[:, index])
    sum_HDA = 0
    for value in f_set:
        sum_HDA += calcHDA(feature, label, index, value)
    return base_e - sum_HDA

def calcInfoGainRatio(feature, label, index):
    '''
    计算信息增益率
    :param feature:测试用例中字典里的feature，类型为ndarray
    :param label:测试用例中字典里的label，类型为ndarray
    :param index:测试用例中字典里的index，即feature部分特征列的索引。该索引指的是feature中第几个特征，如index:0表示使用第一个特征来计算信息增益。
    :return:信息增益率，类型float
    '''

    info_gain = calcInfoGain(feature, label, index)
    unique_value = list(set(feature[:, index]))
    IV = 0
    for value in unique_value:
        len_v = np.sum(feature[:, index] == value)
        IV -= (len_v/len(feature))*np.log2((len_v/len(feature)))
    return info_gain/IV
```

基尼系数

```
import numpy as np

def calcGini(feature, label, index):
    '''
    计算基尼系数
    :param feature:测试用例中字典里的feature，类型为ndarray
    :param label:测试用例中字典里的label，类型为ndarray
    :param index:测试用例中字典里的index，即feature部分特征列的索引。该索引指的是feature中第几个特征，如index:0表示使用第一个特征来计算信息增益。
    :return:基尼系数，类型float
    '''

    #********* Begin *********#
    def _gini(label):
        unique_label = list(set(label))
        gini = 1
        for l in unique_label:
            p = np.sum(label == l)/len(label)
            gini -= p**2
        return gini
    unique_value = list(set(feature[:, index]))
    gini = 0
    for value in unique_value:
        len_v = np.sum(feature[:, index] == value)
        gini += (len_v/len(feature))*_gini(label[feature[:, index] == value])
    return gini
    #********* End *********#
```

剪枝

```
import numpy as np
from copy import deepcopy

class DecisionTree(object):
    def __init__(self):
        #决策树模型
        self.tree = {}

    def calcInfoGain(self, feature, label, index):
```

```
        '''
        计算信息增益
        :param feature:测试用例中字典里的feature，类型为ndarray
        :param label:测试用例中字典里的label，类型为ndarray
        :param index:测试用例中字典里的index，即feature部分特征列的索引。该索引指的是feature中第几个特征，如index:0表示使用第一个特征来计算信息增益。
        :return:信息增益，类型float
        '''

        # 计算熵
        def calcInfoEntropy(feature, label):
            '''
            计算信息熵
            :param feature:数据集中的特征，类型为ndarray
            :param label:数据集中的标签，类型为ndarray
            :return:信息熵，类型float
            '''

            label_set = set(label)
            result = 0
            for l in label_set:
                count = 0
                for j in range(len(label)):
                    if label[j] == l:
                        count += 1
                # 计算标签在数据集中出现的概率
                p = count / len(label)
                # 计算熵
                result -= p * np.log2(p)
            return result

        # 计算条件熵
        def calcHDA(feature, label, index, value):
            '''
            计算信息熵
            :param feature:数据集中的特征，类型为ndarray
            :param label:数据集中的标签，类型为ndarray
            :param index:需要使用的特征列索引，类型为int
            :param value:index所表示的特征列中需要考察的特征值，类型为int
            :return:信息熵，类型float
            '''
            count = 0
```

```
            sub_feature = []
            sub_label = []
            for i in range(len(feature)):
                if feature[i][index] == value:
                    count += 1
                    sub_feature.append(feature[i])
                    sub_label.append(label[i])
            pHA = count / len(feature)
            e = calcInfoEntropy(sub_feature, sub_label)
            return pHA * e

        base_e = calcInfoEntropy(feature, label)
        f = np.array(feature)
        f_set = set(f[:, index])
        sum_HDA = 0
        for value in f_set:
            sum_HDA += calcHDA(feature, label, index, value)
        return base_e - sum_HDA

    def getBestFeature(self, feature, label):
        max_infogain = 0
        best_feature = 0
        for i in range(len(feature[0])):
            infogain = self.calcInfoGain(feature, label, i)
            if infogain > max_infogain:
                max_infogain = infogain
                best_feature = i
        return best_feature

    def calc_acc_val(self, the_tree, val_feature, val_label):
        result = []

        def classify(tree, feature):
            if not isinstance(tree, dict):
                return tree
            t_index, t_value = list(tree.items())[0]
            f_value = feature[t_index]
            if isinstance(t_value, dict):
                classLabel = classify(tree[t_index][f_value],
feature)
                return classLabel
            else:
```

```
                return t_value

        for f in val_feature:
            result.append(classify(the_tree, f))

        result = np.array(result)
        return np.mean(result == val_label)

    def createTree(self, train_feature, train_label):
        if len(set(train_label)) == 1:
            return train_label[0]
        if len(train_feature[0]) == 1 or
len(np.unique(train_feature, axis=0)) == 1:
            vote = {}
            for l in train_label:
                if l in vote.keys():
                    vote[l] += 1
                else:
                    vote[l] = 1
            max_count = 0
            vote_label = None
            for k, v in vote.items():
                if v > max_count:
                    max_count = v
                    vote_label = k
            return vote_label

        best_feature = self.getBestFeature(train_feature,
train_label)
        tree = {best_feature: {}}
        f = np.array(train_feature)
        f_set = set(f[:, best_feature])
        for v in f_set:
            sub_feature = []
            sub_label = []
            for i in range(len(train_feature)):
                if train_feature[i][best_feature] == v:
                    sub_feature.append(train_feature[i])
                    sub_label.append(train_label[i])

            tree[best_feature][v] = self.createTree(sub_feature,
sub_label)
```

```python
        return tree

    def post_cut(self, val_feature, val_label):
        def get_non_leaf_node_count(tree):
            non_leaf_node_path = []

            def dfs(tree, path, all_path):
                for k in tree.keys():
                    if isinstance(tree[k], dict):
                        path.append(k)
                        dfs(tree[k], path, all_path)
                        if len(path) > 0:
                            path.pop()
                    else:
                        all_path.append(path[:])

            dfs(tree, [], non_leaf_node_path)

            unique_non_leaf_node = []
            for path in non_leaf_node_path:
                isFind = False
                for p in unique_non_leaf_node:
                    if path == p:
                        isFind = True
                        break
                if not isFind:
                    unique_non_leaf_node.append(path)
            return len(unique_non_leaf_node)

        def get_the_most_deep_path(tree):
            non_leaf_node_path = []

            def dfs(tree, path, all_path):
                for k in tree.keys():
                    if isinstance(tree[k], dict):
                        path.append(k)
                        dfs(tree[k], path, all_path)
                        if len(path) > 0:
                            path.pop()
                    else:
                        all_path.append(path[:])
```

```
            dfs(tree, [], non_leaf_node_path)

            max_depth = 0
            result = None
            for path in non_leaf_node_path:
                if len(path) > max_depth:
                    max_depth = len(path)
                    result = path
            return result

        def set_vote_label(tree, path, label):
            for i in range(len(path)-1):
                tree = tree[path[i]]
            tree[path[len(path)-1]] = vote_label

        acc_before_cut = self.calc_acc_val(self.tree, val_feature,
val_label)
        for _ in range(get_non_leaf_node_count(self.tree)):
            path = get_the_most_deep_path(self.tree)

            tree = deepcopy(self.tree)
            step = deepcopy(tree)

            for k in path:
                step = step[k]

            vote_label = sorted(step.items(), key=lambda item:
item[1], reverse=True)[0][0]

            set_vote_label(tree, path, vote_label)

            acc_after_cut = self.calc_acc_val(tree, val_feature,
val_label)

            if acc_after_cut > acc_before_cut:
                set_vote_label(self.tree, path, vote_label)
                acc_before_cut = acc_after_cut


    def fit(self, train_feature, train_label, val_feature,
val_label):
```

```
        '''
        :param train_feature:训练集数据，类型为ndarray
        :param train_label:训练集标签，类型为ndarray
        :param val_feature:验证集数据，类型为ndarray
        :param val_label:验证集标签，类型为ndarray
        :return: None
        '''

        #************* Begin ************#
        self.tree = self.createTree(train_feature, train_label)
        # 后剪枝
        self.post_cut(val_feature, val_label)
        #************* End **************#

    def predict(self, feature):
        '''
        :param feature:测试集数据，类型为ndarray
        :return:预测结果，如np.array([0, 1, 2, 2, 1, 0])
        '''

        #************* Begin ************#
        result = []

        def classify(tree, feature):
            if not isinstance(tree, dict):
                return tree
            t_index, t_value = list(tree.items())[0]
            f_value = feature[t_index]
            if isinstance(t_value, dict):
                classLabel = classify(tree[t_index][f_value],
feature)
                return classLabel
            else:
                return t_value

        for f in feature:
            result.append(classify(self.tree, f))

        return np.array(result)
        #************* End **************#
```

鸢尾花

```python
import pandas as pd
from sklearn.tree import DecisionTreeClassifier

train_df = pd.read_csv('./step7/train_data.csv').as_matrix()
train_label = pd.read_csv('./step7/train_label.csv').as_matrix()
test_df = pd.read_csv('./step7/test_data.csv').as_matrix()

dt = DecisionTreeClassifier()
dt.fit(train_df, train_label)
result = dt.predict(test_df)

result = pd.DataFrame({'target':result})
result.to_csv('./step7/predict.csv', index=False)
```