1. （必填）自己提出的问题的理解（罗列全部）：
2. 提出的问题1：如何选择分类的特征

讨论后的理解：常用的特征选择有信息增益，信息增益比，基尼系数等。

提出的问题2：如何避免过拟合

讨论后的理解：产生过拟合有两个原因

原因1：样本问题

（1）样本里的噪音数据干扰过大，大到模型过分记住了噪音特征，反而忽略了真实的输入输出间的关系；（什么是噪音数据？）

（2）样本抽取错误，包括（但不限于）样本数量太少，抽样方法错误，抽样时没有足够正确考虑业务场景或业务特点，等等导致抽出的样本数据不能有效足够代表业务逻辑或业务场景；

（3）建模时使用了样本中太多无关的输入变量。

原因2：构建决策树的方法问题

在决策树模型搭建中，我们使用的算法对于决策树的生长没有合理的限制和修剪的话，决策树的自由生长有可能每片叶子里只包含单纯的事件数据或非事件数据，可以想象，这种决策树当然可以完美匹配（拟合）训练数据，但是一旦应用到新的业务真实数据时，效果是一塌糊涂。

针对原因1的解决方法：

合理、有效地抽样，用相对能够反映业务逻辑的训练集去产生决策树；

针对原因2的解决方法（主要）：

剪枝：提前停止树的增长或者对已经生成的树按照一定的规则进行后剪枝。

1. （必填）别人提出的问题的理解（选择几个问题罗列，并给出理解）：
2. 问题3：cart算法对比C4.5算法有什么优势，为什么要选择cart算法？

自己的理解：ID3 和 C4.5 虽然在对训练样本集的学习中可以尽可能多地挖掘信息，但是其生成的决策树分支、规模都比较大，CART 算法的二分法可以简化决策树的规模，提高生成决策树的效率。C4.5 用的是多叉树，用二叉树效率更高；

问题4：为什么说熵只和分布有关，而和值无关。

自己的理解：从公式可以看出，熵是各项自信息的累加值，由于每一项都是整正数，故而随机变量取值个数越多，状态数也就越多，累加次数就越多，信息熵就越大，混乱程度就越大，纯度越小。

问题5：以信息增益作为划分训练数据集的特征，为什么会存在偏向于选取值较多的特征？

自己的理解：随机变量取值个数越多，状态数也就越多，累加次数就越多，信息熵就越大，混乱程度就越大，纯度越小。选择这样的特征进行分类信息增益最大。

三、（必填）读书计划

1、本周完成的内容章节：李航书第五章

2、下周计划：李航书第六章

四、（选做）读书摘要及理解或伪代码的具体实现（读书摘要、伪代码的具体实现代码等可以写到这个部分）

1、读书摘要及理解（选做）

**三种方法的差异**

划分标准的差异：ID3 使用信息增益偏向特征值多的特征，C4.5 使用信息增益率克服信息增益的缺点，偏向于特征值小的特征，CART 使用基尼指数克服 C4.5 需要求 log 的巨大计算量，偏向于特征值较多的特征。

使用场景的差异：ID3 和 C4.5 都只能用于分类问题，CART 可以用于分类和回归问题；ID3 和 C4.5 是多叉树，速度较慢，CART 是二叉树，计算速度很快；

样本数据的差异：ID3 只能处理离散数据且缺失值敏感，C4.5 和 CART 可以处理连续性数据且有多种方式处理缺失值；从样本量考虑的话，小样本建议 C4.5、大样本建议 CART。C4.5 处理过程中需对数据集进行多次扫描排序，处理成本耗时较高，而 CART 本身是一种大样本的统计方法，小样本处理下泛化误差较大 ；

样本特征的差异：ID3 和 C4.5 层级之间只使用一次特征，CART 可多次重复使用特征；

剪枝策略的差异：ID3 没有剪枝策略，C4.5 是通过悲观剪枝策略来修正树的准确性，而 CART 是通过代价复杂度剪枝。

2.代码实现

import numpy as np

import pandas as pd

import matplotlib.pyplot as plt

from sklearn.datasets import load\_iris

from sklearn.model\_selection import train\_test\_split

from collections import Counter

import math

from math import log

import pprint

# 书上题目5.1

def create\_data():

datasets = [['青年', '否', '否', '一般', '否'],

['青年', '否', '否', '好', '否'],

['青年', '是', '否', '好', '是'],

['青年', '是', '是', '一般', '是'],

['青年', '否', '否', '一般', '否'],

['中年', '否', '否', '一般', '否'],

['中年', '否', '否', '好', '否'],

['中年', '是', '是', '好', '是'],

['中年', '否', '是', '非常好', '是'],

['中年', '否', '是', '非常好', '是'],

['老年', '否', '是', '非常好', '是'],

['老年', '否', '是', '好', '是'],

['老年', '是', '否', '好', '是'],

['老年', '是', '否', '非常好', '是'],

['老年', '否', '否', '一般', '否'],

]

labels = [u'年龄', u'有工作', u'有自己的房子', u'信贷情况', u'类别']

# 返回数据集和每个维度的名称

return datasets, labels

#ID算法生成决策树

# 定义节点类 二叉树

class Node:

def \_\_init\_\_(self, root=True, label=None, feature\_name=None, feature=None):

self.root = root

self.label = label

self.feature\_name = feature\_name

self.feature = feature

self.tree = {}

self.result = {'label:': self.label, 'feature': self.feature, 'tree': self.tree}

def \_\_repr\_\_(self):

return '{}'.format(self.result)

def add\_node(self, val, node):

self.tree[val] = node

def predict(self, features):

if self.root is True:

return self.label

return self.tree[features[self.feature]].predict(features)

class DTree:

def \_\_init\_\_(self, epsilon=0.1):

self.epsilon = epsilon

self.\_tree = {}

# 熵

@staticmethod

def calc\_ent(datasets):

data\_length = len(datasets)

label\_count = {}

for i in range(data\_length):

label = datasets[i][-1]

if label not in label\_count:

label\_count[label] = 0

label\_count[label] += 1

ent = -sum([(p / data\_length) \* log(p / data\_length, 2) for p in label\_count.values()])

return ent

# 经验条件熵

def cond\_ent(self, datasets, axis=0):

data\_length = len(datasets)

feature\_sets = {}

for i in range(data\_length):

feature = datasets[i][axis]

if feature not in feature\_sets:

feature\_sets[feature] = []

feature\_sets[feature].append(datasets[i])

cond\_ent = sum([(len(p) / data\_length) \* self.calc\_ent(p) for p in feature\_sets.values()])

return cond\_ent

# 信息增益

@staticmethod

def info\_gain(ent, cond\_ent):

return ent - cond\_ent

def info\_gain\_train(self, datasets):

count = len(datasets[0]) - 1

ent = self.calc\_ent(datasets)

best\_feature = []

for c in range(count):

c\_info\_gain = self.info\_gain(ent, self.cond\_ent(datasets, axis=c))

best\_feature.append((c, c\_info\_gain))

# 比较大小

best\_ = max(best\_feature, key=lambda x: x[-1])

return best\_

def train(self, train\_data):

"""

input:数据集D(DataFrame格式)，特征集A，阈值eta

output:决策树T

"""

\_, y\_train, features = train\_data.iloc[:, :-1], train\_data.iloc[:, -1], train\_data.columns[:-1]

# 1,若D中实例属于同一类Ck，则T为单节点树，并将类Ck作为结点的类标记，返回T

if len(y\_train.value\_counts()) == 1:

return Node(root=True,

label=y\_train.iloc[0])

# 2, 若A为空，则T为单节点树，将D中实例树最大的类Ck作为该节点的类标记，返回T

if len(features) == 0:

return Node(root=True, label=y\_train.value\_counts().sort\_values(ascending=False).index[0])

# 3,计算最大信息增益 同5.1,Ag为信息增益最大的特征

max\_feature, max\_info\_gain = self.info\_gain\_train(np.array(train\_data))

max\_feature\_name = features[max\_feature]

# 4,Ag的信息增益小于阈值eta,则置T为单节点树，并将D中是实例数最大的类Ck作为该节点的类标记，返回T

if max\_info\_gain < self.epsilon:

return Node(root=True, label=y\_train.value\_counts().sort\_values(ascending=False).index[0])

# 5,构建Ag子集

node\_tree = Node(root=False, feature\_name=max\_feature\_name, feature=max\_feature)

feature\_list = train\_data[max\_feature\_name].value\_counts().index

for f in feature\_list:

sub\_train\_df = train\_data.loc[train\_data[max\_feature\_name] == f].drop([max\_feature\_name], axis=1)

# 6, 递归生成树

sub\_tree = self.train(sub\_train\_df)

node\_tree.add\_node(f, sub\_tree)

# pprint.pprint(node\_tree.tree)

return node\_tree

def fit(self, train\_data):

self.\_tree = self.train(train\_data)

return self.\_tree

def predict(self, X\_test):

return self.\_tree.predict(X\_test)

datasets, labels = create\_data()

data\_df = pd.DataFrame(datasets, columns=labels)

dt = DTree()

tree = dt.fit(data\_df)

print(dt.predict(['老年', '否', '否', '一般']))