**《机器学习基础》实验报告**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **年级、专业、班级** | | **2021级计算机科学与技术（卓越）1班** | | | **姓名** | **韩昊辰** |
| **实验题目** | **决策树算法实践** | | | | | |
| **实验时间** | **2023.5.10** | | **实验地点** | **图书馆** | | |
| **实验成绩** |  | | **实验性质** | **□验证性 □设计性 □综合性** | | |
| 教师评价：  **□**算法/实验过程正确； **□**源程序/实验内容提交 **□**程序结构/实验步骤合理；  **□**实验结果正确； **□**语法、语义正确； **□**报告规范；  其他：  评价教师签名： | | | | | | |
| 一、实验目的  掌握决策树回归算法原理。 | | | | | | |
| 二、实验项目内容  1.理解并**描述**决策树分类、回归算法原理。  2.**编程**实践，将决策树分类、回归算法分别应用于合适的数据集(如鸢尾花、波士顿房价预测、UCI数据集、Kaggle数据集)，要求算法至少用于两个数据集(分类2个，回归2个)。 | | | | | | |
| 三、实验过程或算法（源程序）  1，决策树分类、回归算法原理  决策树（decision tree）：是一种基本的分类与回归方法。从本质上说，分类问题的决策树实际是基于特征对实例进行分类的过程，可以认为是if-then的集合，也可以认为是定义在特征空间与类空间上的条件概率分布。  决策树的创建通常有三个步骤：特征选择、决策树的生成、决策树的修剪。用决策树分类的过程可以概括为从根节点开始，对实例的某一特征进行测试，根据测试结果将实例分配到其子节点，此时每个子节点对应着该特征的一个取值，如此递归的对实例进行测试并分配，直到到达叶节点，最后将实例分到叶节点的类中。  对于决策树的相关特征可以概括如下：   * 决策树学习的目标：根据给定的训练数据集构建一个决策树模型，使它能够对实例进行正确的分类。 * 决策树学习的本质：从训练集中归纳出一组分类规则，或者说是由训练数据集估计条件概率模型。 * 决策树学习的损失函数：正则化的极大似然函数 * 决策树学习的测试：最小化损失函数 * 决策树学习的目标：在损失函数的意义下，选择最优决策树的问题。 * 决策树原理和问答猜测结果游戏相似，根据一系列数据，然后给出游戏的答案。   决策树的生成遵循以下的步骤：  首先创建根节点，将所有训练数据放置于根节点，挑选最优特征，按照该特征的不同取值将训练数据分割成子集。如果这些子集已经被基本分类，那么构建叶节点，将所有子集放在叶节点中。如果子集没有被基本分类，那么继续选择最优特征进行划分，直到所有训练数据子集被基本正确的分类，或者没有合适的特征为止。  选择最优特征的方式有很多，不同的最优特征选择方式将决策树分成了不同的类型，包括ID3，ID4.5，CART决策树。ID3 使用信息增益作为选择特征的准则；C4.5 使用信息增益比作为选择特征的准则；CART 使用 Gini 指数作为选择特征的准则。Gini指数定义如下：    它表示数据集D的纯度高低，Gini指数越小，D纯度越高。对于属性a的Gini指数定义如下：    选择那个使得划分后基尼指数最小的属性作为最优划分属性。  在本实验中，我们选择CART决策树。CART 与 ID3，C4.5 不同之处在于 CART 生成的树必须是二叉树。也就是说，无论是回归还是分类问题，无论特征是离散的还是连续的，无论属性取值有多个还是两个，内部节点只能根据属性值进行二分。CART 的全称是分类与回归树。从这个名字中就应该知道，CART 既可以用于分类问题，也可以用于回归问题。  决策树刚刚生成后，往往对训练数据的分类很准确，但对未知测试数据的分类缺没有那么精确，即会出现过拟合现象，可以用决策树剪枝的方法提高决策树的泛化能力。决策树剪枝分预剪枝和后剪枝。  预剪枝是指在决策树的生成过程中，对每个节点在划分前先进行评估，若当前的划分不能带来泛化性能的提升，则停止划分，并将当前节点标记为叶节点。  后剪枝是指先从训练集生成一颗完整的决策树，然后自底向上对非叶节点进行考察，若将该节点对应的子树替换为叶节点，能带来泛化性能的提升，则将该子树替换为叶节点。  本实验采用后剪枝的方法。将数据集预留一部分数据作为验证集来进行剪枝。  对于回归树，由于特征是连续值，不能再根据连续属性的可取值对节点进行划分，而是应该使用取值范围进行划分。最简单的策略是二分法，即先将所有样本的取值排序，然后去两个样本取值的中点作为一个切分点，如有n个样本，则共计生成n-1个切分点。任选一个切分点，将样本按照小于切分点值和大于切分点值分为两部分去，计算划分后的信息增益，选取让信息增益最大的划分点作为最优切分点。  信息增益计算公式为：    对于决策树回归问题，目标最小化损失函数进行分裂样本，CART回归树的损失函数是MSE损失函数，选择切分点s将数据集分为两部分，计算两部分MSE损失函数之和，得到最小损失函数对应切分点s\*，讲切分的两部分分别递归切分，直到不满足切分条件。停止切分的条件很多，在本实验中我们使用当切分后任意一个部分的样本数小于4（或用户任意指定，作为回归树参数传入），得到叶节点，叶节点预测值可以等于包含样本标记值的算术平均。 | | | | | | |
| 四、实验结果及分析 4.1 决策树分类4.1.1 离散型决策树构建及于西瓜数据集上的应用 数据集描述如下：  用于训练的数据data\_word.csv使用的西瓜书上的例子：    Cash决策树构建代码：  #计算基尼指数  def gini(data):  data\_label = data.iloc[:, -1]  label\_num = data\_label.value\_counts() #有几类，每一类的数量  res = 0  for k in label\_num.keys():  p\_k = label\_num[k]/len(data\_label)  res += p\_k \*\* 2  return 1 - res  # 计算每个特征取值的基尼指数，找出最优切分点  def gini\_index(data,a):  feature\_class = data[a].value\_counts()  res = []  for feature in feature\_class.keys():  weight = feature\_class[feature]/len(data)  gini\_value = gini(data.loc[data[a] == feature])  res.append([feature, weight \* gini\_value])  res = sorted(res, key = lambda x: x[-1])  return res[0]  #获取标签最多的那一类  def get\_most\_label(data):  data\_label = data.iloc[:,-1]  label\_sort = data\_label.value\_counts(sort=True)  return label\_sort.keys()[0]  #挑选最优特征，即基尼指数最小的特征  def get\_best\_feature(data):  features = data.columns[:-1]  res = {}  for a in features:  temp = gini\_index(data, a) #temp是列表，【feature\_value, gini】  res[a] = temp  res = sorted(res.items(),key=lambda x:x[1][1])  return res[0][0], res[0][1][0] # 根蒂，英挺  def drop\_exist\_feature(data, best\_feature, value, type):  attr = pd.unique(data[best\_feature]) #表示特征所有取值的数组  if type == 1: #使用特征==value的值进行划分  new\_data = [[value], data.loc[data[best\_feature] == value]]  else:  new\_data = [attr, data.loc[data[best\_feature] != value]]  new\_data[1] = new\_data[1].drop([best\_feature], axis=1) #删除该特征  return new\_data  #创建决策树  def create\_tree\_lisan(data):  data\_label = data.iloc[:,-1]  if len(data\_label.value\_counts()) == 1: #只有一类  return data\_label.values[0]  if all(len(data[i].value\_counts()) == 1 for i in data.iloc[:,:-1].columns): #所有数据的特征值一样，选样本最多的类作为分类结果  return get\_most\_label(data)  best\_feature, best\_feature\_value = get\_best\_feature(data) #根据信息增益得到的最优划分特征  Tree = {best\_feature:{}} #用字典形式存储决策树  Tree[best\_feature][best\_feature\_value] = create\_tree\_lisan(drop\_exist\_feature(data, best\_feature, best\_feature\_value, 1)[1])  Tree[best\_feature]['Others'] = create\_tree\_lisan(drop\_exist\_feature(data, best\_feature, best\_feature\_value, 2)[1])  return Tree  def predict(Tree , test\_data):  first\_feature = list(Tree.keys())[0] #第一个特征  second\_dict = Tree[first\_feature] #第一个特征后面的字典  input\_first = test\_data.get(first\_feature) #预测输入的第一个特征值是多少  input\_value = second\_dict[input\_first] if input\_first == first\_feature else second\_dict['Others'] #预测输入对应的字典  if isinstance(input\_value , dict): #判断分支还是不是字典  class\_label = predict(input\_value, test\_data)  else:  class\_label = input\_value  return class\_label  将其用在西瓜数据集上，其中，西瓜数据的特征全部为离散属性  if \_\_name\_\_ == '\_\_main\_\_':  #读取数据  data = pd.read\_csv('data\_word.csv')  #创建决策树  dicision\_Tree = create\_tree\_lisan(data)  print(dicision\_Tree)  #测试数据  test\_data\_1 = {'色泽':'青绿','根蒂':'蜷缩','敲声':'浊响','纹理':'稍糊','脐部':'凹陷','触感':'硬滑'}  test\_data\_2 = {'色泽': '乌黑', '根蒂': '稍蜷', '敲声': '浊响', '纹理': '清晰', '脐部': '凹陷', '触感': '硬滑'}  result = predict(dicision\_Tree,test\_data\_2)  print('分类结果为'+'好瓜'if result == 1 else '坏瓜')  运行结果：  {'根蒂': {'硬挺': 0, 'Others': {'纹理': {'模糊': 0, 'Others': {'色泽': {'浅白': {'敲声': {'浊响': 1, 'Others': 0}}, 'Others': {'触感': {'软粘': 1, 'Others': {'脐部': {'稍凹': {'敲声': {'沉闷': 0, 'Others': 1}}, 'Others': {'敲声': {'沉闷': 1, 'Others': 1}}}}}}}}}}}}  分类结果为好瓜 4.1.2 离散型决策树后剪枝处理及在西瓜数据集上的运用 数据集描述：  用于训练的train\_data：    用于测试的test\_data：    对以上决策树代码进行改进，增加了以下代码：  #测试很多案例，返回准确率  def predict\_more(Tree, test\_data, test\_label):  cnt = 0  #计算如果该节点不剪枝的准确率  for i in range(len(test\_data)):  after\_data = test\_data.reset\_index().loc[i].to\_dict()  pred = predict(Tree, after\_data)  if pred == test\_label[i]:  cnt += 1  return cnt / len(test\_label)  #用于预测节点剪枝后的预测正确数  def equalNums(label, featPreLabel):  res = 0  for l in label:  if l == featPreLabel:  res += 1  return res  # 后剪枝  def post\_prunning(tree , test\_data , test\_label , names):  newTree = tree.copy() #copy是浅拷贝  names = np.asarray(names)  # 取决策节点的名称 即特征的名称  featName = list(tree.keys())[0]  # 取特征的列  featCol = np.argwhere(names == featName)[0][0]  names = np.delete(names, [featCol]) #删掉使用过的特征  newTree[featName] = tree[featName].copy() #取值  featValueDict = newTree[featName] #当前特征下面的取值情况  featPreLabel = featValueDict.pop("prun\_label") #如果当前节点剪枝的话是什么标签，并删除\_vpdl  # 分割测试数据 如果有数据 则进行测试或递归调用:  split\_data = drop\_exist\_feature(test\_data,featName) #删除该特征，按照该特征的取值重新划分数据  split\_data = dict(split\_data)  for featValue in featValueDict.keys(): #每个特征的值  if type(featValueDict[featValue]) == dict: #如果下一层还是字典，说明还是子树  split\_data\_feature = split\_data[featValue] #特征某个取值的数据，如“脐部”特征值为“凹陷”的数据  split\_data\_lable = split\_data[featValue].iloc[:, -1].values  # 递归到下一个节点  newTree[featName][featValue] = post\_prunning(featValueDict[featValue],split\_data\_feature,split\_data\_lable,split\_data\_feature.columns)  # 根据准确率判断是否剪枝，注意这里的准确率是到达该节点数据预测正确的准确率，而不是整体数据集的准确率  # 因为在修改当前节点时，走到其他节点的数据的预测结果是不变的，所以只需要计算走到当前节点的数据预测对了没有即可  ratioPreDivision = equalNums(test\_label, featPreLabel) / test\_label.size #判断测试集的数据如果剪枝的准确率  #计算如果该节点不剪枝的准确率  ratioAfterDivision = predict\_more(newTree, test\_data, test\_label)  if ratioAfterDivision < ratioPreDivision:  newTree = featPreLabel # 返回剪枝结果，其实也就是走到当前节点的数据最多的那一类  return newTree  为了保持和西瓜书上的决策树一致，我们直接输入书上的决策树进行剪枝。  注意，字典传入函数后在函数内进行修改，即使没有返回值原本的字典也会改变；在比较剪枝前后的准确率时，我们并不需要判断整棵树的预测情况，而是只需要判断当前节点为根节点的子树对于走到当前节点的部分测试集的预测情况；西瓜书对于后剪枝的定义为判断所有非叶子结点的剪枝预测情况，所以即使一个节点的子树已经判断剪枝或是没有剪枝，该节点都要再次判断。  # 读取数据  train\_data = pd.read\_csv('./train\_data.csv')  test\_data = pd.read\_csv('./test\_data.csv')  test\_data\_label = test\_data.iloc[:, -1].values  names = test\_data.columns  column\_count = dict([(ds, list(pd.unique(train\_data[ds]))) for ds in train\_data.iloc[:, :-1].columns])  continuous\_features = ['密度', '含糖率'] # 先标注连续值  dicision\_Tree = {"脐部": {"prun\_label": 1  , '凹陷': {'色泽': {"prun\_label": 1, '青绿': 1, '乌黑': 1, '浅白': 0}}  , '稍凹': {'根蒂': {"prun\_label": 1  , '稍蜷': {'色泽': {"prun\_label": 1  , '青绿': 1  , '乌黑': {'纹理': {"prun\_label": 1  , '稍糊': 1, '清晰': 0, '模糊': 1}}  , '浅白': 1}}  , '蜷缩': 0  , '硬挺': 1}}  , '平坦': 0}}  print('剪枝前的决策树:')  print(dicision\_Tree)  print('剪枝前的测试集准确率: {}'.format(predict\_more(dicision\_Tree, test\_data, test\_data\_label)))  print('-' \* 20 + '剪枝' + '-' \* 20)  new\_tree = post\_prunning(dicision\_Tree, test\_data, test\_data\_label, names)  print('剪枝后的决策树:')  print(new\_tree)  print('剪枝后的测试集准确率: {}'.format(predict\_more(new\_tree, test\_data, test\_data\_label)))  运行结果：  剪枝前的决策树:  {'脐部': {'prun\_label': 1, '凹陷': {'色泽': {'prun\_label': 1, '青绿': 1, '乌黑': 1, '浅白': 0}}, '稍凹': {'根蒂': {'prun\_label': 1, '稍蜷': {'色泽': {'prun\_label': 1, '青绿': 1, '乌黑': {'纹理': {'prun\_label': 1, '稍糊': 1, '清晰': 0, '模糊': 1}}, '浅白': 1}}, '蜷缩': 0, '硬挺': 1}}, '平坦': 0}}  剪枝前的测试集准确率: 0.42857142857142855  --------------------剪枝--------------------  剪枝后的决策树:  {'脐部': {'凹陷': 1, '稍凹': {'根蒂': {'稍蜷': {'色泽': {'青绿': 1, '乌黑': 1, '浅白': 1}}, '蜷缩': 0, '硬挺': 1}}, '平坦': 0}}  剪枝后的测试集准确率: 0.7142857142857143 4.1.3 连续型决策树构建及在西瓜数据集上的运用 数据集描述：  用于训练的train\_data:    用于测试的test\_data：    连续性决策树在以上的代码基础上增加对连续性特征的处理，代码如下：  #计算信息熵  def cal\_information\_entropy(data):  data\_label = data.iloc[:,-1]  label\_class =data\_label.value\_counts() #总共有多少类  Ent = 0  for k in label\_class.keys():  p\_k = label\_class[k]/len(data\_label)  Ent += -p\_k\*np.log2(p\_k)  return Ent  #对于离散特征a，计算给定数据属性a的信息增益  def cal\_information\_gain(data, a):  Ent = cal\_information\_entropy(data)  feature\_class = data[a].value\_counts() #特征有多少种可能  gain = 0  for v in feature\_class.keys():  weight = feature\_class[v]/data.shape[0]  Ent\_v = cal\_information\_entropy(data.loc[data[a] == v])  gain += weight\*Ent\_v  return Ent - gain  #对于连续特征b，计算给定数据属性b的信息增益  def cal\_information\_gain\_continuous(data, a):  n = len(data) #总共有n条数据，会产生n-1个划分点，选择信息增益最大的作为最优划分点  data\_a\_value = sorted(data[a].values) #从小到大排序  Ent = cal\_information\_entropy(data) #原始数据集的信息熵Ent(D)  select\_points = []  for i in range(n-1):  val = (data\_a\_value[i] + data\_a\_value[i+1]) / 2 #两个值中间取值为划分点  data\_left = data.loc[data[a]<val]  data\_right = data.loc[data[a]>val]  ent\_left = cal\_information\_entropy(data\_left)  ent\_right = cal\_information\_entropy(data\_right)  result = Ent - len(data\_left)/n \* ent\_left - len(data\_right)/n \* ent\_right  select\_points.append([val, result])  select\_points.sort(key = lambda x : x[1], reverse= True) #按照信息增益排序  return select\_points[0][0], select\_points[0][1] #返回信息增益最大的点, 以及对应的信息增益  #获取标签最多的那一类  def get\_most\_label(data):  data\_label = data.iloc[:,-1]  label\_sort = data\_label.value\_counts(sort=True)  return label\_sort.keys()[0]  #获取最佳划分特征  def get\_best\_feature(data):  features = data.columns[:-1]  res = {}  for a in features:  if a in continuous\_features:  temp\_val, temp = cal\_information\_gain\_continuous(data, a)  res[a] = [temp\_val, temp]  else:  temp = cal\_information\_gain(data, a)  res[a] = [-1, temp] #离散值没有划分点，用-1代替  res = sorted(res.items(),key=lambda x:x[1][1],reverse=True)  return res[0][0],res[0][1][0]  #将数据转化为（属性值：数据）的元组形式返回，并删除之前的特征列，只针对离散数据  def drop\_exist\_feature(data, best\_feature):  attr = pd.unique(data[best\_feature])  new\_data = [(nd, data[data[best\_feature] == nd]) for nd in attr]  new\_data = [(n[0], n[1].drop([best\_feature], axis=1)) for n in new\_data]  return new\_data  #创建决策树  def create\_tree(data):  data\_label = data.iloc[:,-1]  if len(data\_label.value\_counts()) == 1: #只有一类  return data\_label.values[0]  if all(len(data[i].value\_counts()) == 1 for i in data.iloc[:,:-1].columns): #所有数据的特征值一样，选样本最多的类作为分类结果  return get\_most\_label(data)  best\_feature, best\_feature\_val = get\_best\_feature(data) #根据信息增益得到的最优划分特征  if best\_feature in continuous\_features: #连续值  node\_name = best\_feature + '<' + str(best\_feature\_val)  Tree = {node\_name:{}} #用字典形式存储决策树  Tree[node\_name]['是'] = create\_tree(data.loc[data[best\_feature] < best\_feature\_val])  Tree[node\_name]['否'] = create\_tree(data.loc[data[best\_feature] > best\_feature\_val])  else:  Tree = {best\_feature:{}}  exist\_vals = pd.unique(data[best\_feature]) # 当前数据下最佳特征的取值  if len(exist\_vals) != len(column\_count[best\_feature]): # 如果特征的取值相比于原来的少了  no\_exist\_attr = set(column\_count[best\_feature]) - set(exist\_vals) # 少的那些特征  for no\_feat in no\_exist\_attr:  Tree[best\_feature][no\_feat] = get\_most\_label(data) # 缺失的特征分类为当前类别最多的  for item in drop\_exist\_feature(data, best\_feature): # 根据特征值的不同递归创建决策树  Tree[best\_feature][item[0]] = create\_tree(item[1])  return Tree  #根据创建的决策树进行分类  def predict(Tree , test\_data):  first\_feature = list(Tree.keys())[0]  if (feature\_name:= first\_feature.split('<')[0]) in continuous\_features:  second\_dict = Tree[first\_feature]  val = float(first\_feature.split('<')[-1])  input\_first = test\_data.get(feature\_name)  if input\_first < val:  input\_value = second\_dict['是']  else:  input\_value = second\_dict['否']  else:  second\_dict = Tree[first\_feature]  input\_first = test\_data.get(first\_feature)  input\_value = second\_dict[input\_first]  if isinstance(input\_value , dict): #判断分支还是不是字典  class\_label = predict(input\_value, test\_data)  else:  class\_label = input\_value  return class\_label  if \_\_name\_\_ == '\_\_main\_\_':  data = pd.read\_csv('西瓜数据集3.0.csv')  # 统计每个特征的取值情况作为全局变量  column\_count = dict([(ds, list(pd.unique(data[ds]))) for ds in data.iloc[:, :-1].columns])  test = cal\_information\_gain\_continuous(data, '密度')  continuous\_features = ['密度', '含糖率'] #先标注连续值  dicision\_tree = create\_tree(data)  print(dicision\_tree)  test\_data = {'色泽':'青绿','根蒂':'蜷缩','敲声':'浊响','纹理':'清晰','脐部':'凹陷','触感':'硬滑','密度':0.51,'含糖率':0.3}  result = predict(dicision\_tree, test\_data)  print('预测结果为:{}'.format('好瓜' if result == 1 else '坏瓜'))  运行结果：  {'纹理': {'清晰': {'密度<0.3815': {'是': 0, '否': 1}}, '稍糊': {'触感': {'软粘': 1, '硬滑': 0}}, '模糊': 0}}  预测结果为:好瓜 4.1.4 连续性决策分类树在Iris数据集上的运行效果 数据集描述：  用于训练的数据iris\_data.csv：    共234组数据  用于测试的数据iris\_data\_test.csv：    共67组数据  # 预测iris数据  data = pd.read\_csv('iris\_data.csv')  column\_count = dict([(ds, list(pd.unique(data[ds]))) for ds in data.iloc[:, :-1].columns])  continuous\_features = ['花萼长度','花萼宽度','花瓣长度','花瓣宽度']  dicision\_tree = create\_tree(data)  print(dicision\_tree)  test\_data = pd.read\_csv('iris\_data\_test.csv')  test\_data\_val = test\_data.iloc[:,:-1]  test\_data\_label = test\_data.iloc[:,-1]  print('测试集准确率: {}'.format(predict\_more(dicision\_tree, test\_data\_val, test\_data\_label)))  运行结果：  {'花瓣宽度<1.0': {'是': 0.0, '否': {'花瓣宽度<1.8': {'是': {'花瓣长度<5.0': {'是': {'花萼长度<5.0': {'是': 2.0, '否': 1.0}}, '否': {'花萼长度<6.0': {'是': 0.0, '否': 2.0}}}}, '否': 2.0}}}}  测试集准确率: 0.9850746268656716 在以上代码中，我犯了一个错误，认为决策树回归就是对特征值为连续值的样本的分类，但这本质上是分类问题，而决策树回归实际上处理的是标签值为连续型的样本的处理，这本质是回归问题，下面是重新实现决策树回归算法：4.2 决策树回归4.2.1 决策树构建 **决策树构建代码：**  # 计算平均值，负责生成叶子节点，叶子节点根据公式就是求所在子区域的平均值  def regLeaf(dataset):  return mean(dataset[:, -1])  # 计算损失函数中的总方差=方差x样本个数=每个点与估计的差（叶节点的均值）的平方和  def regErr(dataset):  return var(dataset[:, -1]) \* shape(dataset)[0]  # 根据特征索引和分裂点划分数据集为左右两部分  def splitDataset(dataset, index\_of\_feature, value):  mat0 = dataset[nonzero(dataset[:, index\_of\_feature] <= value)[0], :]  mat1 = dataset[nonzero(dataset[:, index\_of\_feature] > value)[0], :]  return mat0, mat1  # 选择最佳特征和相应的划分点  def choose\_best\_feature(dataset, leafType=regLeaf, errType=regErr, ops=(1, 4)):  min\_error = ops[0] # 最小下降损失  min\_samples = ops[1] # 最小样本数  if len(set(dataset[:, -1])) == 1: # 数据集中都是同一个目标值  return None, leafType(dataset) # 返回最佳划分特征为None,切分点为 均值  m, n = shape(dataset) # 统计数据集的行和列  Loss = errType(dataset) # 损失函数  bestLoss = inf # 最优切分点的误差  bestIndex = 0 # 最优特征索引  bestValue = 0 # 最优切分点的值  for featureIndex in range(n - 1): # 遍历特征  for value in set(dataset[:, featureIndex]):  mat0, mat1 = splitDataset(dataset, featureIndex, value)  if (shape(mat0)[0] < min\_samples) or (shape(mat1)[0] < min\_samples): # 若两个子区域的叶子节点小于4个样本，则跳过此划分点  continue  curLoss = errType(mat0) + errType(mat1)  if curLoss < bestLoss:  bestLoss = curLoss  bestIndex = featureIndex  bestValue = value  if Loss - bestLoss < min\_error: # 若切分之前的损失-切分之后的损失< min\_error，那么就是损失减少不明显，停止切分  return None, leafType(dataset)  mat0, mat1 = splitDataset(dataset, featureIndex, value)  if (shape(mat0)[0] < min\_samples) or (shape(mat1)[0] < min\_samples):  return None, leafType(dataset)  return bestIndex, bestValue  # 生成回归树  def generate\_tree(dataset, leafType=regLeaf, errType=regErr, ops=(1, 4)):  feature\_index, value = choose\_best\_feature(dataset, leafType, errType, ops) # 最优切分点划分  if feature\_index == None:  return value  reTree = {}  reTree['spInd'] = feature\_index # 最优切分特征的索引  reTree['spVal'] = value # 最优切分点的值  lSet, rSet = splitDataset(dataset, feature\_index, value)  reTree['left'] = generate\_tree(lSet, leafType, errType, ops)  reTree['right'] = generate\_tree(rSet, leafType, errType, ops)  return reTree  **剪枝处理：**  # 判断是叶子还是树  def isTree(obj):  return (type(obj).\_\_name\_\_ == 'dict')  # 获取左右叶子的平均值  def getMean(reTree):  if isTree(reTree['left']):  reTree['left'] = getMean(reTree['left'])  if isTree(reTree['right']):  reTree['right'] = getMean(reTree['right'])  return (reTree['left'] + reTree['right']) / 2.0  # 后剪枝  def postpruning(reTree, test\_data):  if shape(test\_data)[0] == 0:  print("判断测试集为空，执行过吗？")  return getMean(reTree)  if (isTree(reTree['left']) or isTree(reTree['right'])):  lSet, rSet = splitDataset(test\_data, reTree['spInd'], reTree['spVal'])  if isTree(reTree['left']):  reTree['left'] = postpruning(reTree['left'], lSet)  if isTree(reTree['right']):  reTree['right'] = postpruning(reTree['right'], rSet)  if not isTree(reTree['left']) and not isTree(reTree['right']):  lSet, rSet = splitDataset(test\_data, reTree['spInd'], reTree['spVal'])  errorNoMerge = sum(power(lSet[:, -1] - reTree['left'], 2)) + sum(  power(rSet[:, -1] - reTree['right'], 2)) # 剪枝前的误差  treeMean = (reTree['left'] + reTree['right']) / 2.0  errorMerge = sum(power(test\_data[:, -1] - treeMean, 2)) # 剪枝后的误差  if errorMerge < errorNoMerge:  print("剪枝")  return treeMean # 剪枝后的节点值为原来左右叶子结点的均值  else:  return reTree # 不剪枝  else:  return reTree  **预测样本：**  # 回归树预测一个样本  def reTree\_predict\_one\_test(reTree, one\_test\_example):  first\_feature\_index = reTree[list(reTree.keys())[0]] # 获取第一个结点的特征索引  feature\_spVal = reTree[list(reTree.keys())[1]] # 获取第一个结点的分裂点  predict\_val = 0.0  if one\_test\_example[first\_feature\_index] <= feature\_spVal:  if type(reTree['left']).\_\_name\_\_ == 'dict':  predict\_val = reTree\_predict\_one\_test(reTree['left'], one\_test\_example)  else:  predict\_val = reTree['left']  else:  if type(reTree['right']).\_\_name\_\_ == 'dict':  predict\_val = reTree\_predict\_one\_test(reTree['right'], one\_test\_example)  else:  predict\_val = reTree['right']  return predict\_val  # 回归树预测所有测试数据结果  def reTree\_predict(reTree, test\_data):  classLabel = []  for one\_test in test\_data:  classLabel.append(reTree\_predict\_one\_test(reTree, one\_test))  return classLabel 4.2.2 在自创数据集上的应用 数据集描述：  训练数据：    测试数据：    代码：  myData = load\_data('re\_data.txt')  reTree = generate\_tree(myData)  reTree\_str = str(reTree)  print("未剪枝之前的决策树：")  print(reTree\_str)  test\_example = [0.8935465]  predict\_val = reTree\_predict\_one\_test(reTree, test\_example)  print(predict\_val)  print("剪枝之后的决策树：")  testData = load\_data('re\_data\_test.txt')  postPrunTree = postpruning(reTree, testData)  print(postPrunTree)  predict\_val = reTree\_predict\_one\_test(postPrunTree, test\_example)  print(predict\_val)  运行结果：   4.2.3 在糖尿病数据集上的应用 # 处理数据集  def handle\_input\_data(data,target):  dataset = []  for i in range(len(data)):  tmp = list(append(data[i], target[i]))  dataset.append(tmp)  return array(dataset)  # 计算MSE误差  def cul\_r(predict\_y,y):  res = 0  for i in range(len(y)):  r\_i = (y[i]-predict\_y[i])\*\*2  res += r\_i  return res/(len(y)\*\*2)  def diabetes\_retree():  diabetes = load\_diabetes()  data = diabetes.data  target = diabetes.target  x\_train, x\_test, y\_train, y\_test = train\_test\_split(data, target, test\_size=0.2, random\_state=666)  myData = handle\_input\_data(x\_train, y\_train)  testData = handle\_input\_data(x\_test, y\_test)  reTree = generate\_tree(myData)  reTree\_str = str(reTree)  print("未剪枝之前的决策树：")  print(reTree\_str)  classlist = reTree\_predict(reTree,testData)  print("平均误差为：", cul\_r(classlist,y\_test))  print("剪枝之后的决策树：")  postPrunTree = postpruning(reTree, testData)  print(postPrunTree)  classlist = reTree\_predict(postPrunTree, testData)  print("平均误差为：", cul\_r(classlist,y\_test))  运行结果：   对本实验的思考 由于本人能力所限，在实验过程中走了很多歪路，浪费了很多时间，但还是有不小的收获。首先，决策树的分类非常类似于人的决策过程，对问题的求解实际就是if-then的过程。其次，回归决策树和分类决策树构建有很多相似之处。首先选取最优划分属性时都会对每一个可能划分点进行划分后的子集合求loss函数，分类树的loss采用Gini函数，反映该子集的纯度；回归树使用MSE函数，反映子集中预测值与真实值的偏差程度。而分类树的叶节点大概率会包含多个样本，它们的类别标签相同，而回归树的叶节点通过提前停止切分的方式保证包含多个样本，但其标签值可能互不相同，取标签平均值作为叶节点的预测值。 | | | | | | |