山东大学 计算机科学与技术 学院

机器学习与模式识别 课程实验报告

|  |  |  |  |
| --- | --- | --- | --- |
| 学号：202300130183 | 姓名： 宋浩宇 | | 班级：23级人工智能班 |
| 实验题目：Experiment 9: Decision Tree | | | |
| 实验学时：2 | | 实验日期：2025/4/8 | |
| 实验环境：  软件环境：  系统：Windows 11 家庭中文版23H2 22631.4317  计算软件：MATLAB 版本: 9.8.0.1323502 (R2020a)  Java 版本: Java 1.8.0\_202-b08 with Oracle Corporation Java HotSpot(TM) 64-Bit Server VM mixed mode  硬件环境：  CPU：13th Gen Intel(R) Core(TM) i9-13980HX 2.20 GHz  内存：32.0 GB (31.6 GB 可用)  磁盘驱动器：NVMe WD\_BLACKSN850X2000GB  显示适配器：NVIDIA GeForce RTX 4080 Laptop GPU | | | |
| 1. 实验内容   In this exercise, you need to implement Decision Tree.   1. 实验步骤 2. 加载数据 3. 数据预处理和测试集/训练集划分 4. 实现算法 5. 训练模型 6. 测试结果正确率 7. 模型可视化 8. 测试结果   正确率统计为：    决策树可视化为（顺序为ID3算法生成的决策树、C45算法生成的决策树、CART算法生成的决策树）：        附录：实现源代码   |  | | --- | | '''超参数设置'''  train\_set\_rate = 0.7  '''训练集占比'''  max\_depth = 10  '''决策树最大深度'''  def load\_data(filename):      import csv      return csv.reader(open(filename, 'r'))    class DataSet:      """      # DataSet      这个类用于存储数据集，包括特征和标签，以及特征和标签的名称。      ## method      ### from\_csv(filename)      从csv文件中读取数据集，并返回一个DataSet对象。        ### from\_list(features, labels, features\_names, labels\_names)      从列表中读取数据集，并返回一个DataSet对象。        ### shuffle()      随机打乱数据集。      """      data\_count : int      '''数据个数'''      feature\_count : int      '''特征个数'''      features : list      '''特征列表'''      features\_names : dict      '''特征名称字典，用于从名称索引到列号'''      labels : list      '''标签列表'''      labels\_names : dict      '''标签名称字典，用于从名称索引到列号'''      labels\_count : dict      '''标签种类和对应个数'''      def \_\_init\_\_(self, data\_set = None, features:list = None, labels:list = None, features\_names:dict = None, labels\_names:dict = None):            self.labels\_names = {}          self.features\_names = {}          self.labels\_count = {}          if data\_set is not None:              features\_table = []              labels\_table = []                for row in data\_set:                  features\_table.append(list(row[:len(row) - 1]))                  labels\_table.append(list(row[-1]))              labels\_table[0] = [''.join(labels\_table[0])]              for i in range(len(labels\_table[0])):                  self.labels\_names[labels\_table[0][i]] = i              for i in range(len(features\_table[0])):                  self.features\_names[features\_table[0][i]] = i              self.labels = [[float(value) for value in row] for row in labels\_table[1:]]              self.features = [[float(value) for value in row] for row in features\_table[1:]]              self.data\_count = len(self.features) - 1              self.feature\_count = len(self.features[0])              for i in self.labels:                  if i[0] not in self.labels\_count:                      self.labels\_count[i[0]] = 1                  else:                      self.labels\_count[i[0]] += 1              return          if data\_set is None and features is not None and labels is not None and features\_names is not None and labels\_names is not None:              self.features = features              self.labels = labels              self.features\_names = features\_names              self.labels\_names = labels\_names              if len(self.features) >= 1:                    self.data\_count = len(self.features)                  self.feature\_count = len(self.features[0])              else:                  self.data\_count = 0                  self.feature\_count = 0              for i in self.labels:                  if i[0] not in self.labels\_count:                      self.labels\_count[i[0]] = 1                  else:                      self.labels\_count[i[0]] += 1              return          if data\_set is None and features is None and labels is None and features\_names is None and labels\_names is None:              self.data\_count = 0              self.feature\_count = 0              self.features = []              self.labels = []              self.features\_names = {}              self.labels\_names = {}              return      @classmethod      def from\_csv(cls, filename):          data\_reader = load\_data(filename)          return cls(data\_reader)      @classmethod      def from\_list(cls, features:list, labels:list, features\_names:dict, labels\_names:dict):          return cls(None, features, labels, features\_names, labels\_names)      def shuffle(self):          result = []          for i in range(self.data\_count):              result.append(self.features[i] + self.labels[i])          import random          random.shuffle(result)            self.features = [[float(value) for value in row[:len(row) - 1]] for row in result]          self.labels = [[float(value) for value in row[len(row) - 1:]] for row in result]      def min\_max\_normalize(self):          '''          最大最小值归一化处理          '''          min\_values = [float('inf') for i in range(self.feature\_count)]          max\_values = [float('-inf') for i in range(self.feature\_count)]          for i in range(self.data\_count):              for j in range(self.feature\_count):                  if self.features[i][j] < min\_values[j]:                      min\_values[j] = self.features[i][j]                  if self.features[i][j] > max\_values[j]:                      max\_values[j] = self.features[i][j]          for i in range(self.data\_count):              for j in range(self.feature\_count):                  self.features[i][j] = (self.features[i][j] - min\_values[j]) / (max\_values[j] - min\_values[j])        def \_\_repr\_\_(self):          return "DataSet(data\_count={}, feature\_count={}, features\_names={}, labels\_names={})".format(self.data\_count, self.feature\_count, self.features\_names, self.labels\_names) + "\n" + str(str(self.features[:10]) + "...") + "\n" + str(str(self.labels[:10]) + "...") + "\n" + str(self.labels\_count)      def information\_entropy(self) -> float:          '''          计算信息熵          '''          entropy = 0          for label in self.labels\_count:              import math              p = self.labels\_count[label] / self.data\_count              entropy -= p \* math.log2(p)          return entropy        def gini\_index(self) -> float:          '''          计算基尼指数          '''          gini = 1          for label in self.labels\_count:              p = self.labels\_count[label] / self.data\_count              gini -= p\*\*2          return gini  class DecisionNode:      '''      # DecisionNode      决策树的节点类，包括特征名称、特征索引、阈值、子树、待分类样本集合。      '''      selectable\_features : list      '''可选择的特征'''      feature\_name : str      feature\_index : int      threshold : float      children : list      divided\_set : DataSet      \_\_leaf:bool      \_\_class:str        def \_\_init\_\_(self, feature\_name:str = None, feature\_index:int = None, threshold:float = None, children:list = None, divided\_set:DataSet = None, selectable\_features:list = None, leaf:bool = False, class\_label:str = None):          self.feature\_name = feature\_name          self.feature\_index = feature\_index          self.threshold = threshold          self.children = children          self.divided\_set = divided\_set          self.selectable\_features = selectable\_features          self.\_\_leaf = False          if class\_label is not None:              self.\_\_class = class\_label          if self.divided\_set is None:              self.\_\_leaf = True          if leaf:              self.\_\_leaf = True          if self.divided\_set is not None:              if self.divided\_set.data\_count == 0:                  self.\_\_leaf = True              if len([i for i in self.divided\_set.labels\_count.keys()]) == 1:                  self.\_\_leaf = True          else:              self.\_\_leaf = True          if self.\_\_leaf is not None:              if self.\_\_leaf:                  if len([i for i in self.divided\_set.labels\_count.keys()]) == 1:                      self.\_\_class = list(self.divided\_set.labels\_count.keys())[0]          else:              self.\_\_class = None          if leaf:              self.\_\_leaf = True      def is\_leaf(self):          return self.\_\_leaf      def plot\_text(self):          if self.\_\_leaf:              return str(self.\_\_class)          else:              return str(self.feature\_name) + " <= " + str(self.threshold)      def get\_class(self):          return self.\_\_class      def \_\_divide(self, feature\_index:int, threshold:float, feature\_name:str = None):          '''          划分子节点          '''          negative\_features\_list = []          positive\_features\_list = []          negative\_labels\_list = []          positive\_labels\_list = []            for i in range(self.divided\_set.data\_count):              if self.divided\_set.features[i][feature\_index] <= threshold:                  negative\_features\_list.append(self.divided\_set.features[i])                  negative\_labels\_list.append(self.divided\_set.labels[i])              else:                  positive\_features\_list.append(self.divided\_set.features[i])                  positive\_labels\_list.append(self.divided\_set.labels[i])          positive\_class = None          negative\_class = None          positive\_leaf = None          negative\_leaf = None            if len(positive\_labels\_list) == 0:              positive\_class = list(self.divided\_set.labels\_count.keys())[0]              positive\_leaf = True          if len(negative\_labels\_list) == 0:              negative\_class = list(self.divided\_set.labels\_count.keys())[1]              negative\_leaf = True            positive\_set = DataSet.from\_list(positive\_features\_list, positive\_labels\_list, self.divided\_set.features\_names, self.divided\_set.labels\_names)          negative\_set = DataSet.from\_list(negative\_features\_list, negative\_labels\_list, self.divided\_set.features\_names, self.divided\_set.labels\_names)          selectable\_features = [i for i in self.selectable\_features]          selectable\_features.remove(feature\_name)          if positive\_set.data\_count == 0:              if list(self.divided\_set.labels\_count.values())[0] > list(self.divided\_set.labels\_count.values())[1]:                  positive\_class = list(self.divided\_set.labels\_count.keys())[0]              else:                  positive\_class = list(self.divided\_set.labels\_count.keys())[1]              positive\_leaf = True          if negative\_set.data\_count == 0:              if list(self.divided\_set.labels\_count.values())[0] > list(self.divided\_set.labels\_count.values())[1]:                  negative\_class = list(self.divided\_set.labels\_count.keys())[0]              else:                  negative\_class = list(self.divided\_set.labels\_count.keys())[1]              negative\_leaf = True          if selectable\_features == []:              if list(self.divided\_set.labels\_count.values())[0] > list(self.divided\_set.labels\_count.values())[1]:                  negative\_class = list(self.divided\_set.labels\_count.keys())[0]              else:                  negative\_class = list(self.divided\_set.labels\_count.keys())[1]              negative\_leaf = True              if list(self.divided\_set.labels\_count.values())[0] > list(self.divided\_set.labels\_count.values())[1]:                  positive\_class = list(self.divided\_set.labels\_count.keys())[0]              else:                  positive\_class = list(self.divided\_set.labels\_count.keys())[1]              positive\_leaf = True          self.children = [DecisionNode(None, None, None,None, positive\_set, selectable\_features, positive\_leaf, positive\_class),DecisionNode(None, None, None,None, negative\_set, selectable\_features, negative\_leaf, negative\_class)]          self.feature\_name = feature\_name          self.feature\_index = feature\_index          self.threshold = threshold        def divide\_ID3(self):          '''          ID3算法划分子节点          '''          if self.\_\_leaf:              return          # 先找出每一种特征的信息增益最大值和对应的阈值          global\_max\_gain = {}          global\_max\_threshold = {}          from tqdm import tqdm          for i in tqdm(self.selectable\_features):              max\_gain = []              max\_threshold = []              feature\_index = self.divided\_set.features\_names[i]              feature\_values = [row[feature\_index] for row in self.divided\_set.features]              feature\_values.sort()              for j in tqdm(range(len(feature\_values) - 1)):                  threshold = (feature\_values[j] + feature\_values[j + 1]) / 2                  self.\_\_divide(feature\_index, threshold, i)                  gain = self.information\_gain()                  max\_gain.append(gain)                  max\_threshold.append(threshold)                global\_max\_gain[i] = max(max\_gain)              for j in range(len(max\_gain)):                  if max\_gain[j] == global\_max\_gain[i]:                      global\_max\_threshold[i] = max\_threshold[j]          # print(global\_max\_gain)          # print(global\_max\_threshold)          # 选出信息增益最大的特征和阈值          max\_gain\_feature = max(global\_max\_gain, key=global\_max\_gain.get)          max\_gain\_threshold = global\_max\_threshold[max\_gain\_feature]          # print("选出特征：{}，阈值：{}".format(max\_gain\_feature, max\_gain\_threshold))          self.\_\_divide(train\_set.features\_names[max\_gain\_feature], max\_gain\_threshold, max\_gain\_feature)          self.children[0].divide\_ID3()          self.children[1].divide\_ID3()      def divide\_C45(self):          '''          C4.5算法划分子节点          '''          if self.\_\_leaf:              return          # 先找出每一种特征的信息增益最大值和对应的阈值          global\_max\_gain = {}          global\_max\_threshold = {}          from tqdm import tqdm          for i in tqdm(self.selectable\_features):              max\_gain = []              max\_threshold = []              feature\_index = self.divided\_set.features\_names[i]              feature\_values = [row[feature\_index] for row in self.divided\_set.features]              feature\_values.sort()              for j in tqdm(range(len(feature\_values) - 1)):                  threshold = (feature\_values[j] + feature\_values[j + 1]) / 2                  self.\_\_divide(feature\_index, threshold, i)                  gain = self.information\_gain()                  max\_gain.append(gain)                  max\_threshold.append(threshold)                global\_max\_gain[i] = max(max\_gain)              for j in range(len(max\_gain)):                  if max\_gain[j] == global\_max\_gain[i]:                      global\_max\_threshold[i] = max\_threshold[j]            avg\_gain = sum(global\_max\_gain.values()) / len(global\_max\_gain) - 1e-10          to\_select\_feature = [i for i in global\_max\_gain if global\_max\_gain[i] >= avg\_gain]          max\_ratio\_gain = {}            for i in tqdm(to\_select\_feature):              self.\_\_divide(self.divided\_set.features\_names[i], global\_max\_threshold[i], i)              gain\_ration = self.gain\_ratio()              max\_ratio\_gain[i] = gain\_ration          # print(global\_max\_gain)          # print(global\_max\_threshold)          # 选出信息增益最大的特征和阈值          try:              max\_gain\_feature = max(max\_ratio\_gain, key=max\_ratio\_gain.get)          except Exception as e:              print(e)              print(global\_max\_gain)              print(avg\_gain)              exit()          max\_gain\_threshold = global\_max\_threshold[max\_gain\_feature]          # print("选出特征：{}，阈值：{}".format(max\_gain\_feature, max\_gain\_threshold))          self.\_\_divide(train\_set.features\_names[max\_gain\_feature], max\_gain\_threshold, max\_gain\_feature)          self.children[0].divide\_C45()          self.children[1].divide\_C45()        def divide\_CART(self):          '''          CART算法划分子节点          '''          if self.\_\_leaf:              return          # 先找出每一种特征的基尼系数最小值和对应的阈值          global\_min\_gini = {}          global\_min\_threshold = {}          from tqdm import tqdm          for i in tqdm(self.selectable\_features):              min\_gini = []              min\_threshold = []              feature\_index = self.divided\_set.features\_names[i]              feature\_values = [row[feature\_index] for row in self.divided\_set.features]              feature\_values.sort()              for j in tqdm(range(len(feature\_values) - 1)):                  threshold = (feature\_values[j] + feature\_values[j + 1]) / 2                  self.\_\_divide(feature\_index, threshold, i)                  gini = self.gini\_index()                  min\_gini.append(gini)                  min\_threshold.append(threshold)                global\_min\_gini[i] = min(min\_gini)              for j in range(len(min\_gini)):                  if min\_gini[j] == global\_min\_gini[i]:                      global\_min\_threshold[i] = min\_threshold[j]            # print(global\_min\_gini)          # print(global\_min\_threshold)          # 选出基尼系数最小的特征和阈值          min\_gini\_feature = min(global\_min\_gini, key=global\_min\_gini.get)          min\_gini\_threshold = global\_min\_threshold[min\_gini\_feature]          # print("选出特征：{}，阈值：{}".format(min\_gini\_feature, min\_gini\_threshold))          self.\_\_divide(train\_set.features\_names[min\_gini\_feature], min\_gini\_threshold, min\_gini\_feature)          self.children[0].divide\_CART()          self.children[1].divide\_CART()        def pruning(self):          self.\_\_leaf = True          labels\_count = self.divided\_set.labels\_count          max\_possible = max(labels\_count.values())          for i in labels\_count:              if labels\_count[i] == max\_possible:                  self.\_\_class = i                  self.\_\_leaf = True        def unpruning(self):          self.\_\_leaf = False          self.\_\_class = None        def decision(self, data\_dict:dict = None, data\_list:list = None) -> str:          '''          进行决策获得结果          '''          if data\_dict is not None:              if self.\_\_leaf:                  return self.\_\_class              if data\_dict[self.feature\_name] <= self.threshold:                  return self.children[0].decision(data\_dict)              else:                  return self.children[1].decision(data\_dict)          if data\_list is not None:              if self.\_\_leaf:                  return self.\_\_class              if data\_list[self.feature\_index] <= self.threshold:                  return self.children[0].decision(None, data\_list)              else:                  return self.children[1].decision(None, data\_list)          if self.\_\_leaf:              return self.\_\_class      def information\_gain(self) -> float:          '''          计算信息增益          '''          self\_information\_entropy = self.divided\_set.information\_entropy()          children\_information\_entropy = 0          for child in self.children:              coefficient = abs(child.divided\_set.data\_count / self.divided\_set.data\_count)              if coefficient == 0:                  continue              children\_information\_entropy += coefficient \* child.divided\_set.information\_entropy()          return self\_information\_entropy - children\_information\_entropy      def gini\_index(self) -> float:          '''          计算基尼指数          '''          gini\_index = 0          for child in self.children:              coefficient = abs(child.divided\_set.data\_count / self.divided\_set.data\_count)+1e-10              gini\_index+=coefficient \* child.divided\_set.gini\_index()          return gini\_index        def gain\_ratio(self) -> float:          '''          计算信息增益比          '''          self\_information\_entropy = self.divided\_set.information\_entropy()          children\_information\_entropy = 0          for child in self.children:              import math              coefficient = abs(child.divided\_set.data\_count / self.divided\_set.data\_count)              coefficient+=1e-10              if coefficient == 0:                  continue              children\_information\_entropy += coefficient \* child.divided\_set.information\_entropy() \* math.log2(coefficient) + 1e-10          children\_information\_entropy = -children\_information\_entropy          return self\_information\_entropy / children\_information\_entropy      def \_\_repr\_\_(self):          return "DecisionNode(feature\_name={}, feature\_index={}, threshold={}, children={}, divided\_set={})".format(self.feature\_name, self.feature\_index, self.threshold, self.children, self.divided\_set)  import matplotlib.pyplot as plt  class PlotTreeNode:      def \_\_init\_\_(self, value):          self.val = value          self.left = None          self.right = None  from collections import deque  def array\_to\_bst(array):      if not array:          return None      iter\_array = iter(array)      root = PlotTreeNode(next(iter\_array))      queue = deque([root])      while queue:          current\_node = queue.popleft()          try:              left\_value = next(iter\_array)              if left\_value is not None:                  current\_node.left = PlotTreeNode(left\_value)                  queue.append(current\_node.left)              right\_value = next(iter\_array)              if right\_value is not None:                  current\_node.right = PlotTreeNode(right\_value)                  queue.append(current\_node.right)          except StopIteration:              break      return root  import matplotlib.pyplot as plt  def plot\_tree(node, parent\_name, node\_name, edge\_label, pos=None, x=0, y=0, layer=1):      if pos is None:          pos = {}      pos[node\_name] = (x, y)      plt.text(x, y, str(node.val), fontsize=12, ha='center')      plt.scatter(x, y, s=200, color='gray')      if parent\_name is not None:          plt.plot([x, pos[parent\_name][0]], [y, pos[parent\_name][1]], 'k-')          plt.scatter(x, y, s=200, color='gray')          plt.text((x+pos[parent\_name][0])/2, (y+pos[parent\_name][1])/2, edge\_label, fontsize=8, ha='center')      if node.left:          plot\_tree(node.left, node\_name, node\_name+"≤", '≤', pos, x-1/2\*\*layer, y-1, layer+1)      if node.right:          plot\_tree(node.right, node\_name, node\_name+"≥", '≥', pos, x+1/2\*\*layer, y-1, layer+1)      return pos  def draw\_bst(root):      fig, ax = plt.subplots()      ax.axis('off')      plot\_tree(root, None, 'Root', None)      plt.show()    class DecisionTree:      '''      # DecisionTree      决策树类，包括根节点、最大深度。      '''      root : DecisionNode      max\_depth : int      feature\_names : dict      label\_names : dict      def \_\_init\_\_(self, max\_depth:int = None, root:DecisionNode = None , feature\_names:dict = None, label\_names:dict = None):          if max\_depth is not None:              self.max\_depth = max\_depth          if root is not None:              self.root = root          if feature\_names is not None:              self.feature\_names = feature\_names          if label\_names is not None:              self.label\_names = label\_names          pass        @classmethod      def train\_ID3(cls, train\_set:DataSet, max\_depth:int):          '''          获取一个决策树对象          '''          features\_names = [i for i in train\_set.features\_names.keys()]          root = DecisionNode(None, None, None, None, train\_set, features\_names)          root.divide\_ID3()          return cls(max\_depth, root, train\_set.features\_names, train\_set.labels\_names)      @classmethod      def train\_C45(cls, train\_set:DataSet, max\_depth:int):          '''          获取一个决策树对象          '''          features\_names = [i for i in train\_set.features\_names.keys()]          root = DecisionNode(None, None, None, None, train\_set, features\_names)          root.divide\_C45()          return cls(max\_depth, root, train\_set.features\_names, train\_set.labels\_names)      @classmethod      def train\_CART(cls, train\_set:DataSet, max\_depth:int):          '''          获取一个决策树对象          '''          features\_names = [i for i in train\_set.features\_names.keys()]          root = DecisionNode(None, None, None, None, train\_set, features\_names)          root.divide\_CART()          return cls(max\_depth, root, train\_set.features\_names, train\_set.labels\_names)      def \_\_decision(self,data\_list:list = None, data\_dict:dict = None):          '''          进行决策获得结果          '''          return self.root.decision(data\_dict, data\_list)        def decision\_from\_data\_list(self, data\_list:list):          return self.\_\_decision(data\_list)      def decision\_from\_data\_dict(self, data\_dict:dict):          return self.\_\_decision(None, data\_dict)        def test(self, test\_set:DataSet, plot = None):          '''          测试模型          '''          correct\_count = 0          error\_count = 0          from tqdm import tqdm          if plot is not None:              for i in tqdm(range(test\_set.data\_count)):                  test\_sample = {}                  for j in range(len(test\_set.features[i])):                      test\_sample[list(test\_set.features\_names.keys())[j]] = test\_set.features[i][j]                  result = self.decision\_from\_data\_dict(test\_sample)                  # print("预测结果：{}，真实结果：{}".format(result, test\_set.labels[i]))                  if result == test\_set.labels[i][0]:                      correct\_count += 1                  else:                      error\_count += 1              print("正确率：{}%".format(correct\_count / (correct\_count + error\_count) \* 100))          else:              for i in tqdm(range(test\_set.data\_count)):                  test\_sample = {}                  for j in range(len(test\_set.features[i])):                      test\_sample[list(test\_set.features\_names.keys())[j]] = test\_set.features[i][j]                  result = self.decision\_from\_data\_dict(test\_sample)                  # print("预测结果：{}，真实结果：{}".format(result, test\_set.labels[i]))                  if result == test\_set.labels[i][0]:                      correct\_count += 1                  else:                      error\_count += 1          return correct\_count / (correct\_count + error\_count) \* 100      def \_\_repr\_\_(self):          return "DecisionTree(max\_depth={})".format(self.max\_depth) + "\n" + str(self.root)        def plot(self):          '''          绘制决策树          '''          node\_array = [self.root.plot\_text()]          tem = self.root          queue = deque([tem])          while queue:              node = queue.popleft()              if node is None:                  continue              if node.is\_leaf():                  node\_array.append(None)                  node\_array.append(None)                  continue              if node.children:                  if len(node.children) == 2:                      node\_array.append(node.children[0].plot\_text())                      node\_array.append(node.children[1].plot\_text())                      queue.append(node.children[0])                      queue.append(node.children[1])                      # print("{} => {}".format(node.plot\_text(), node.children[0].plot\_text()))              else:                  node\_array.append(None)                  node\_array.append(None)          print(node\_array)          # 示例使用          root = array\_to\_bst(node\_array)          draw\_bst(root)        def post\_pruning(self, test\_set:DataSet):          '''          后剪枝          '''          tem = self.root          queue = deque([tem])          to\_pruning = []          while queue:              node = queue.popleft()              if node is None:                  continue              to\_pruning.append(node)              if node.children:                  if len(node.children) == 2:                      queue.append(node.children[0])                      queue.append(node.children[1])              else:                  continue          to\_pruning.reverse()          for i in to\_pruning:              if i.is\_leaf():                  continue              old\_correct\_rate = self.test(test\_set)              i.pruning()              new\_correct\_rate = self.test(test\_set)              if new\_correct\_rate < old\_correct\_rate:                  i.unpruning()      def random\_split\_data\_set(dataset:DataSet, rate:float) -> tuple[DataSet,DataSet]:      '''      训练集和测试机划分      '''      data\_set\_size = dataset.data\_count      data\_set.shuffle()      train\_set = DataSet.from\_list(dataset.features[:int(data\_set\_size \* rate)], dataset.labels[:int(data\_set\_size \* rate)], dataset.features\_names, dataset.labels\_names)      test\_set = DataSet.from\_list(dataset.features[int(data\_set\_size \* rate):], dataset.labels[int(data\_set\_size \* rate):], dataset.features\_names, dataset.labels\_names)      return train\_set, test\_set      if \_\_name\_\_ == '\_\_main\_\_':      import time      # 数据加载和预处理      data\_set = DataSet.from\_csv('f:/Homework/机器学习作业/实验/实验7/ex7Data/ex7Data.csv')      # data\_set = DataSet.from\_csv('f:/Homework/机器学习作业/实验/实验7/ex7Data/ex7DataSubset.csv')      data\_set.min\_max\_normalize()      train\_set, test\_set = random\_split\_data\_set(data\_set, 0.7)        # 训练      start\_time = time.time()      ID3\_tree = DecisionTree.train\_ID3(train\_set, 5)      end\_time = time.time()      print("ID3决策树训练时间：{}s".format(end\_time - start\_time))        start\_time = time.time()      C45\_tree = DecisionTree.train\_C45(train\_set, 5)      end\_time = time.time()      print("C45决策树训练时间：{}s".format(end\_time - start\_time))        start\_time = time.time()      CART\_tree = DecisionTree.train\_CART(train\_set, 5)      end\_time = time.time()      print("CART决策树训练时间：{}s".format(end\_time - start\_time))      # 后剪枝      start\_time = time.time()      ID3\_tree.post\_pruning(test\_set)      end\_time = time.time()      print("ID3决策树后剪枝时间：{}s".format(end\_time - start\_time))        start\_time = time.time()      C45\_tree.post\_pruning(test\_set)      end\_time = time.time()      print("C45决策树后剪枝时间：{}s".format(end\_time - start\_time))        start\_time = time.time()      CART\_tree.post\_pruning(test\_set)      end\_time = time.time()      print("CART决策树后剪枝时间：{}s".format(end\_time - start\_time))      # 测试      start\_time = time.time()      ID3\_tree.test(test\_set,1)      end\_time = time.time()      print("ID3决策树测试时间：{}s".format(end\_time - start\_time))        start\_time = time.time()      C45\_tree.test(test\_set,1)      end\_time = time.time()      print("C45决策树测试时间：{}s".format(end\_time - start\_time))        start\_time = time.time()      CART\_tree.test(test\_set,1)      end\_time = time.time()      print("CART决策树测试时间：{}s".format(end\_time - start\_time))        # 可视化      ID3\_tree.plot()      C45\_tree.plot()      CART\_tree.plot() | | | | |