**数据挖掘实验报告**

学 院： 计算机科学与工程学院

班 级： 软件工程2016-2班

姓 名： 魏仲华

学 号： 201601060928

**山 东 科 技 大 学**

2019年05月

数据挖掘实验报告

实验序号：实验1　 实验项目名称：编程实现Apriori算法

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 学　号 | 201601060928 | 姓　名 | 魏仲华 | 专业班级 | 软件2016-2 |
| 实验地点 | J13-132 | 指导教师 | 刘彤 | 实验时间 | 2019/4/18 |
| 一、运行环境  1. 操作系统： Windows 10 1803 64bit  2. 编程语言： Python3.6.4 64bit  3. 使用的类库：无  二、算法思想  Aprior算法大致如下所述：   1. 由筛选满足条件的初始频繁一项集，并对其进行计数。 2. 然后对计数后的频繁项集进行连接运算构成频繁二项集。 3. 筛选满足最小支持度的频繁二项集，并进行计数。 4. 重复类似于2,3步，生成频繁三四五六项集等     三、核心代码  程序核心代码如下：  aprior\_gen：由频繁n项集得到频繁n+1项集候选集  def **apriori\_gen**(L):  Ck = []  for l1 in L:  for l2 in L:  *# print(l1, l2)*  if l1[: -1] == l2[: -1] and l1[-1] < l2[-1]:  c = l1 + [l2[-1]]  *# print(c)*  *# connent*  if has\_infrequent\_subset(c, L):  pass  *# delete c*  else:  Ck.append(c)  return Ck  def **find\_frequent\_1\_itemsets**(D, min\_sup):  """  find frequent 1 itemsets  """  count = Counter()  for t in D:  for item in t:  count.increase(item)  **print**(count)  res = []  for item in count:  if count[item] >= min\_sup:  res.append([item])  return res  aprior: 算法框架  def **aprior**(D, min\_sup):  L = [0]  *# find out frequent 1 itemsets*  L[0] = find\_frequent\_1\_itemsets(D, min\_sup)  *# print(L[0])*  *# k = 2*  while L[-1]:  Ck = apriori\_gen(L[-1])  *# print(Ck)*  count = Counter()  for t in D:  Ct = subset(Ck, t)  *# print(Ct)*  *# print("#")*  for c in Ct:  count.increase(tuple(c))  *# print(count)*  Lk = list(**filter**(lambda x: count[tuple(x)] >= min\_sup, Ck))  *# print(Lk)*  L.append(Lk)  return [item for item in L[:-1]]  四、运行结果  运行python apriori.py    data = [  [1,2,5],  [2,4],  [2,3],  [1,2,4],  [1,3],  [2,3],  [1,3],  [1,2,3,5],  [1,2,3]  ]  输入数据如上所示，简便起见固定在自己代码里。  最下边一行即为输出的按频繁项集长度分类的全部频繁项集  五、问题及解决方案  本实验比较简单，编写的过程中没有遇到太多的问题 | | | | | |

数据挖掘实验报告

实验序号：实验2　 实验项目名称：编程实现FP-Tree算法

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 学　号 | 201601060928 | 姓　名 | 魏仲华 | 专业班级 | 软件2016-2 |
| 实验地点 | J13-132 | 指导教师 | 刘彤 | 实验时间 | 2019/4/25 |
| 一、运行环境  1. 操作系统： Windows 10 1803 64bit  2. 编程语言： Python3.6.4 64bit  3. 使用的类库：无  二、算法思想  FP树算法大致如下所述：  1. 生成按支持数递减的的频繁项集合L  2. 构建FP树：首先创建树根节点，然后对于每个事务，按L的次序创建分支， 并更新节点的计数。同时，使用一个节点链表将相同的节点串联起来并进行计数。  3.构建条件模式基，并根据条件FP树。若树只有一个分支，通过子集遍历分支输出所有的频繁项集及其支持度，若不只一个分支，递归进行这一步。  三、核心代码  代码主要数据结构包括**ConditionalFPTree（条件FP树），FPTree（FP树），FPTreeBase（条件FP树与FP树的基类），Node（树节点数据结构）, NodeLinkListItem（FP树中节点链数据结构）, FrequentPattern（频繁模式数据结构）等类**。  通过FPGrowth方法进行调用生成频繁项集。  def **fpGrowth**(tree, alpha):  if tree.has\_unique\_branch():  nodes = tree.get\_all\_node\_exclude\_root()  *# print([str(r) for r in nodes])*  for node\_list in get\_subset(nodes):  beta = FrequentPattern(node\_list)  if beta.support\_count >= tree.min\_sup:  beta = beta | alpha  **print**(beta)  else:  **print**(tree.get\_head\_table())  for item in tree.get\_head\_table():  *# 产生模式beta*  beta = FrequentPattern([item])  beta = beta | alpha  **print**(beta)  conditional\_pattern\_base = []  def **get\_prefix\_path**(node):  path = []  cur\_node = node.get\_parent()  while cur\_node and not cur\_node.is\_root():  path.insert(0, cur\_node)  cur\_node = cur\_node.get\_parent()  return path    for node in item.nodes:  prefix\_path = get\_prefix\_path(node)  if prefix\_path:  conditional\_pattern\_base.append(FrequentPattern(prefix\_path, support\_count=node.cnt))  if conditional\_pattern\_base:  tree\_beta = ConditionalFPTree(min\_sup=tree.min\_sup)  tree\_beta.set\_conditional\_pattern\_base(conditional\_pattern\_base)  tree\_beta.build()  fpGrowth(tree\_beta, beta)  class **FPTreeBase**():  def **\_\_init\_\_**(self, \*, min\_sup):  self.\_rank = {}  self.min\_sup = min\_sup  self.nodes = [] *# 记录树的节点*  self.edges = {} *# 记录树的边*  self.parents = {} *# 记录父节点，方便回溯*  self.node\_link\_list = collections.OrderedDict()  self.init() *# 初始化null根节点*  def **create\_node**(self, value, cnt):  nodes\_length = **len**(self.nodes)  node = Node(nodes\_length, value, cnt, self)  self.nodes.append(node)  return node  def **get\_all\_node\_exclude\_root**(self):  return self.nodes[1:]    def **init**(self):  self.create\_node('null', 0)  self.parents[0] = None    def **insert**(self, record):  pass  def **get\_head\_table**(self):  return list(self.node\_link\_list.values())[::-1]  *# def init\_node\_list(self, itemset):*  *# # 初始化节点链*  *# for item in itemset:*  *# self.node\_link\_list[item] =*  def **update\_node\_list**(self, node):  *# 更新节点链及其支持度计数*  *# print("@@@@@", str(node))*  if node.v not in self.node\_link\_list:  self.node\_link\_list[node.v] = NodeLinkListItem(node.v)  self.node\_link\_list[node.v].nodes.append(node)  def **get\_sorted\_record**(self, record):  return **sorted**(record, key=lambda x: self.\_rank[x])  def **get\_root**(self):  return self.nodes[0]  def **insert\_node**(self, item):  pass    def **insert\_affair**(self, affair):  items = affair  root = self.get\_root()  cur\_node = root  common\_prefix = True  for (item, support\_count) in items:  if common\_prefix:  *# print(str(cur\_node),[str(item) for item in cur\_node.get\_child()])*  if **len**(cur\_node.get\_child()) == 0:  common\_prefix = False  update\_flag = False  for node in cur\_node.get\_child():  if str(node.v) == str(item):  node.cnt += support\_count  *# print("Update %s" % (str(node),))*  cur\_node = node  update\_flag = True  break  if not update\_flag:  common\_prefix = False  if not common\_prefix:  node = self.create\_node(item, support\_count)  *# print("CREATE %d" % (item, ))*  cur\_node.append\_child(node)  cur\_node = node  self.update\_node\_list(node)  def **has\_unique\_branch**(self):  """  判断是否只有一个分支  """  *# 根节点度数为一，即为只有一条边*  return **len**(self.edges[0]) == 1  class **FPTree**(**FPTreeBase**):  """  FP树的数据结构，0代表根节点  """  def **\_\_init\_\_**(self, \*, min\_sup):  super().**\_\_init\_\_**(min\_sup=min\_sup)  self.D = []  def **loads**(self, D):  self.D = D  def **get\_inital\_frequent\_itemset**(self, D, min\_sup):  count = Counter()  for t in D:  for item in t:  count.increase(item)  *# print(count)*  res = {}  for item in count:  if count[item] >= min\_sup:  res[item] = count[item]  sorted\_items = **sorted**(res.items(), key=lambda x: -x[1])  for i in **range**(**len**(sorted\_items)):  self.\_rank[sorted\_items[i][0]] = i  return {key: value for key, value in sorted\_items}  *# return itemset*  def **build**(self):  self.get\_inital\_frequent\_itemset(self.D, self.min\_sup)  *# self.init\_node\_list(itemset)*  *# print(itemset)*  *# print(self.node\_link\_list)*  for record in self.D:  sorted\_record = self.get\_sorted\_record(record)  *## 以下是条件FP树与FP树的共同之处*  *#print(list(map(lambda x: (x, 1), sorted\_record)))*  self.insert\_affair(list(**map**(lambda x: (x, 1), sorted\_record)))  class **ConditionalFPTree**(**FPTreeBase**):  def **\_\_init\_\_**(self, \*, min\_sup, D = []):  super().**\_\_init\_\_**(min\_sup=min\_sup)  self.D = D  def **set\_conditional\_pattern\_base**(self, D):  self.D = D  def **is\_empty**(self):  return not self.D  def **build**(self):  for record in self.D:  self.insert\_affair(list(**map**(lambda x: (x, record.support\_count), record.pattern)))  class **FrequentPattern**:  def **\_\_init\_\_**(self, u=[], \*\*kwargs):  self.support\_count = kwargs.get("support\_count", None)  self.pattern = []  self.join(u)  def **\_\_str\_\_**(self):  return "%d %s" % (self.support\_count or 0, str(self.pattern))    def **\_add\_frequent\_pattern**(self, other):  if other.support\_count:  self.pattern += other.pattern  self.support\_count = **min**(self.support\_count, other.support\_count) if self.support\_count else other.support\_count  def **\_add\_nodelink\_item**(self, other):  self.pattern.append(other.item)  self.support\_count = **min**(self.support\_count, other.count) if self.support\_count else other.count    def **\_add\_node\_item**(self, other):  self.pattern.append(other.v)  self.support\_count = **min**(self.support\_count, other.cnt) if self.support\_count else other.cnt  def **join**(self, u):  if **isinstance**(u, list):  for item in u:  if **isinstance**(item, Node):  self.\_add\_node\_item(item)  elif **isinstance**(item, NodeLinkListItem):  self.\_add\_nodelink\_item(item)  elif **isinstance**(u, FrequentPattern):  self.\_add\_frequent\_pattern(u)  def **\_\_or\_\_**(self, other):  self.join(other)  return self  class **Node**():  def **\_\_init\_\_**(self, identify, v, cnt, tree):  self.\_tree = tree  self.v = v  self.cnt = cnt  self.identify = identify  def **is\_root**(self):  return not self.identify  def **get\_child**(self):  if self.identify not in self.\_tree.edges:  return []  return self.\_tree.edges[self.identify]  def **get\_parent**(self):  return self.\_tree.parents[self.identify]  def **append\_child**(self, node):  if self.identify not in self.\_tree.edges:  self.\_tree.edges[self.identify] = []  self.\_tree.edges[self.identify].append(node)  self.\_tree.parents[node.identify] = self  def **\_\_str\_\_**(self):  return "%s:%d" % (self.v, self.cnt)  五、完成结果    最后输出的结果即频繁项集及其支持度  六、问题及解决方案  FP树数据结构比较复杂，写的时候出现了许多笔误，但因代码过于复杂排错过程也十分艰难。后来改进了调试流程；在实现的过程中发现FP与条件FP树之间有很多重复代码，因此后边改进的过程中提取出来了一个基类来实现他们的公共部分。 | | | | | |

数据挖掘实验报告

实验序号：实验3，4　 实验项目名称：编程实现ID3、C4.5、CART算法

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 学　号 | 201601060928 | 姓　名 | 魏仲华 | 专业班级 | 软件2016-2 |
| 实验地点 | J13-132 | 指导教师 | 刘彤 | 实验时间 | 2019/4/30 |
| 一、运行环境  1. 操作系统： Windows 10 1803 64bit  2. 编程语言： Python3.6.4 64bit  3. 使用的类库：numpy, pandas  二、算法思想  决策树算法大致如下所述：   1. 根据属性选择算法（如ID3采用信息增益，C4.5采用信息增益率，CART采用基尼指数），选择最好的属性 2. 根据该属性的值分区产生子树，针对子树递归进行建树过程     对于ID3算法，采用信息增益算法：        对于C4.5算法，在ID3的基础上改进为信息增益率：      对于CART算法，则使用基尼指数      另外关于CART剪枝，其目的是用于消除过拟合。剪枝主要思想是：比较所有节点剪枝前后的预测误差，对于预测误差最大的分支进行剪枝。其评价指标是表面误差增益率。  表面误差增益率公式如下：    三、核心代码  代码主要包括在四个文件中：  **base.py 包括所有决策树的基类，主要包含DecisionTree（决策树基类）, Tree（代表决策树的树结构），Node（代表决策树结点）, Edge（代表决策树的边）四个类**  **C45.py**主要包括决策树类C45**Tree，继承自DecisionTree**  ID3.py 主要包括决策树类**ID3Tree，继承自DecisionTree**  CART.py主要包括决策树类**CARTTree，继承自DecisionTree**  utils.py 包含一些辅助的工具函数  test\_id3.py 是对ID3的单元测试文件  test\_cart.py 是对CART的单元测试文件  test\_c45.py 是对C4.5的单元测试文件  另外还编写了相应的图形界面用于展示建树结果，限于篇幅相关代码在此不再展示。  class **Tree**:  def **\_\_init\_\_**(self):  super().**\_\_init\_\_**()  self.edges = {  }  self.root\_id = "id"  self.nodes = {  }  def **get\_root**(self):  return self.nodes[self.root\_id]    def **\_get\_child\_node\_json\_data**(self, node\_id):  if node\_id not in self.edges:  return {  "id": node\_id,  "value": self.nodes[node\_id].value,  "is\_leaf": True,  }  else:  return {  "id": node\_id,  "value": self.nodes[node\_id].value,  "children": [self.\_get\_child\_node\_json\_data(edge.target.id) for edge in self.edges[node\_id]]  }  def **get\_node\_json\_data**(self):  return self.\_get\_child\_node\_json\_data(self.root\_id)  def **get\_edge\_json\_data**(self):  return [edge.get\_json\_data() for node\_edges in self.edges.values() for edge in node\_edges]  def **get\_json\_data**(self):  return {  "nodes": self.get\_node\_json\_data(),  "edges": self.get\_edge\_json\_data(),  }  def **create\_node**(self):  return Node(tree=self)  def **add\_node**(self, node):  if not self.nodes:  self.root\_id = node.id  self.nodes[node.id] = node  def **add\_edge**(self, par\_node, node, edge\_value):  par\_node\_id = par\_node.id  if par\_node\_id not in self.edges:  self.edges[par\_node\_id] = []  edge = Edge()  edge.source = par\_node  edge.target = node  edge.value = edge\_value  self.edges[par\_node\_id].append(edge)  class **DecisionTree**:  def **\_\_init\_\_**(self):  self.df = None  self.tree = Tree()  def **load\_csv\_data\_set**(self, filename):  self.df = pd.read\_csv(filename)  def **load\_data\_set**(self, data):  self.df = data  def **set\_class\_attribute**(self, index):  """  设置用于分类的属性字段  : index 字段序号  """  self.\_class\_attribute = index  def **get\_class\_attribute**(self):  """  获取用于分类属性字段  """  if **hasattr**(self, '\_class\_attribute'):  return self.df.columns[self.\_class\_attribute]  else:  return self.df.columns[-1]  def **get\_json\_result**(self):  return self.tree.get\_json\_data()  def **attribute\_selection\_method**(self, D, attribute\_list):  raise Exception("unimplemented method")  def **is\_both\_same\_class**(self, D):  group\_size = D.groupby(self.get\_class\_attribute()).size()  return (True, group\_size.idxmax()) if group\_size.size == 1 else (False, None)  def **is\_discrete**(self, t):  return True  def **get\_part**(self, D, splitting\_criterion):  attribute = splitting\_criterion[0]  parts = splitting\_criterion[1]  return [(part, D[D[attribute].isin(part)]) for part in parts]  def **get\_majority\_class**(self, D):  return str(D.groupby(self.get\_class\_attribute()).size().idxmax())  def **\_generate\_decision\_tree**(self, D, attribute\_list):  tree = self.tree  N = tree.create\_node()  N.D = D  tree.add\_node(N)  is\_same\_class, C = self.is\_both\_same\_class(D)  **print**(attribute\_list)  **print**(D)  if is\_same\_class:  N.is\_leaf = True  N.value = str(C)  **print**(type(N.value), N.value)  return N *# 返回N作为叶节点，类C标记*  if **len**(attribute\_list) == 0:  N.is\_leaf = True  N.value = str(self.get\_majority\_class(D))  **print**(type(N.value), N.value)  return N *# 返回叶节点，多数类*  splitting\_criterion = self.attribute\_selection\_method(D, attribute\_list)  N.value = str(splitting\_criterion[0])  *# print(splitting\_criterion)*  if self.is\_discrete(splitting\_criterion):  *#print("Hello World")*  attribute\_list.drop(attribute\_list[attribute\_list == splitting\_criterion[0]].index, inplace=True)  *#attribute\_list.index = range(len(attribute\_list))*  *# print(attribute\_list)*  for (label, Dj) in self.get\_part(D, splitting\_criterion):  if Dj.empty:  **print**("Empty")  value = self.get\_majority\_class(D) *# 为啥会是空？？？？*  else:  child\_N = self.\_generate\_decision\_tree(Dj, attribute\_list)  child\_N.label = label  tree.add\_edge(N, child\_N, label)  return N  def **build**(self):  class\_attribute = self.get\_class\_attribute()  attrs = []  for attr in self.df.columns:  if attr != class\_attribute:  attrs.append(attr)  self.\_generate\_decision\_tree(self.df, pd.Series(attrs))  def **test\_one\_record**(self, record):  tree = self.tree  node = tree.get\_root()  while node.has\_child():  value = record[node.value]  node = node.get\_child\_by\_value(value)  if node is None:  *# 没有找到*  return "null"  return node.value  def **test**(self, test\_records):  res = []  *# precision = 0*  *# recall = 0*  for record in test\_records:  series = pd.Series(record, index=self.df.columns)  value = self.test\_one\_record(series)  if value == series[self.get\_class\_attribute()]:  res.append((1, value))  else:  res.append((0, value))  return res  class **C45Tree**(**DecisionTree**):  def **\_\_init\_\_**(self):  super().**\_\_init\_\_**()  def **attribute\_selection\_method**(self, D, attribute\_list):  def **getInfo**(D):  """  获取元组分类的熵  """  group = D.groupby(self.get\_class\_attribute())  total = **len**(D)  *# print([len(x[1]) for x in group1])*  InfoD = -**sum**([**len**(x[1]) / total \* math.log2(**len**(x[1]) / total) for x in group])  return InfoD  def **getInfoSubD**(D, attribute):  """  获取对属性划分后的熵  """  info = 0  group1 = D.groupby(attribute)  for x in group1:  *#column\_name = x[0]*  Dj = x[1]  property\_size = **len**(Dj)  info += property\_size / (**len**(group1)) \* getInfo(Dj)  return info  def **getSplitInfoA**(D, attribute):  """  获取对属性划分后的熵  """  split\_info = 0  group1 = D.groupby(attribute)  D\_size = **len**(group1)  for x in group1:  *#column\_name = x[0]*  Dj = x[1]  Dj\_size = **len**(Dj)  split\_info += (Dj\_size / D\_size) \* math.log2( Dj\_size / D\_size)  return split\_info  candidate\_splitting\_criterion = []  InfoD = getInfo(D)  total = **len**(D.groupby(self.get\_class\_attribute()))  for attribute in attribute\_list:  InfoSubD = getInfoSubD(D, attribute)  gain = InfoD - InfoSubD  split\_info = getSplitInfoA(D, attribute)  gain\_rate = gain / split\_info  candidate\_splitting\_criterion.append((gain\_rate, attribute))  **print**(gain\_rate, attribute)  attribute = **max**(candidate\_splitting\_criterion, key=lambda item: item[0])[1]  splitting\_criterion = (attribute, list([item] for item in D[attribute].drop\_duplicates()))  **print**("chose attribute", splitting\_criterion)  return splitting\_criterion  class **CARTree**(**DecisionTree**):  def **\_\_init\_\_**(self):  super().**\_\_init\_\_**()  def **attribute\_selection\_method**(self, D, attribute\_list):  def **getGini**(D):  """  获取元组分类的熵  """  group = D.groupby(self.get\_class\_attribute())  total = **len**(D)  *# print([len(x[1]) for x in group1])*  InfoD = 1 - **sum**([math.pow(**len**(x[1]) / total, 2) for x in group])  return InfoD  def **getGiniA**(D, attribute):  """  获取对属性划分后的熵  """  min\_gini = (None, tuple())  values = list(D[attribute].drop\_duplicates())  **print**(values)  choices = get\_split\_choice(values)  D\_size = **len**(choices)  for choice in choices:  gini = 0  D1\_size = **len**(choice[0])  D2\_size = **len**(choice[1])  D1 = D[D[attribute].isin(choice[0])]  D2 = D[D[attribute].isin(choice[1])]  gini = (D1\_size / D\_size) \* getGini(D1) + (D2\_size / D\_size) \* getGini(D2)  if min\_gini[0] == None or gini < min\_gini[0]:  min\_gini = (gini, choice)  return min\_gini  candidate\_splitting\_criterion = []  gini\_d = getGini(D)  total = **len**(D.groupby(self.get\_class\_attribute()))  for attribute in attribute\_list:  (gini\_a, choice) = getGiniA(D, attribute)  gain = gini\_d - gini\_a  candidate\_splitting\_criterion.append((gain, attribute, choice))  *# print(gain, attribute)*  splitting\_criterion = **max**(candidate\_splitting\_criterion, key=lambda item: item[0])[1:]  **print**("chose attribute", splitting\_criterion)  return splitting\_criterion    def **get\_surface\_err\_gains**(self, class\_field, N, node, queue):  childs = node.get\_all\_child()  child\_cnt = **len**(childs)  node.\_child\_num = 0  node.\_Rt = 0  if child\_cnt:  RT = 0  child\_num = 0  for child\_node in childs:  self.get\_surface\_err\_gains(class\_field, N, child\_node, queue)  RT += node.\_Rt  node.\_child\_num += child\_node.\_child\_num  class\_desc = child\_node.get\_class\_description(class\_field)  Rt = **min**([item[1] for item in class\_desc]) / N  *#Rt = min(class\_desc[0][1], class\_desc[1][1]) / (class\_desc[0][1] + class\_desc[1][1]) \* ( class\_desc[0][1] + class\_desc[1][1] / N)*  err\_gain = (Rt - RT) / (child\_num - 1)  queue.append((node, err\_gain))  node.\_child\_num = child\_num  node.\_Rt = RT  else:  class\_desc = node.get\_class\_description(class\_field)  Rt = **min**([item[1] for item in class\_desc]) / N  node.\_Rt = Rt  node.\_child\_num = 0  def **pruning\_tree**(self):  “”” CART剪枝实现  queue = []  self.get\_surface\_err\_gains(self.get\_class\_attribute(), **len**(self.df), self.tree.get\_root(), queue)  pruned\_node = **max**(queue, key=lambda x: x[1])[0]  pruned\_node.cut\_up\_childs()  pruned\_node.value = self.get\_majority\_class(pruned\_node.D)  def **main**():  tree = CARTree()  tree.load\_csv\_data\_set("data.csv")  tree.build()  *#print(decision\_tree.get\_json\_result())*  class **ID3Tree**(**DecisionTree**):  def **\_\_init\_\_**(self):  super().**\_\_init\_\_**()  def **attribute\_selection\_method**(self, D, attribute\_list):  def **getInfo**(D):  group = D.groupby(self.get\_class\_attribute())  total = **len**(D)  *# print([len(x[1]) for x in group1])*  InfoD = -**sum**([**len**(x[1]) / total \* math.log2(**len**(x[1]) / total) for x in group])  return InfoD  candidate\_splitting\_criterion = []  InfoD = getInfo(D)  total = **len**(D.groupby(self.get\_class\_attribute()))  for attribute in attribute\_list:  group1 = D.groupby(attribute)  InfoageD = 0  for x in group1:  column\_name = x[0]  cdf = x[1]  property\_size = **len**(cdf)  *# print(property\_size)*  *# print(column\_name)*  *# print(cdf)*  InfoageD += property\_size / (**len**(group1)) \* getInfo(cdf)  *# print(InfoageD)*  gain = InfoD - InfoageD  candidate\_splitting\_criterion.append((gain, attribute))  **print**(gain, attribute)  attribute = **max**(candidate\_splitting\_criterion, key=lambda item: item[0])[1]  splitting\_criterion = (attribute, list([item] for item in D[attribute].drop\_duplicates()))  **print**("chose attribute", splitting\_criterion)  return splitting\_criterion  class **Edge**:  def **\_\_init\_\_**(self):  super().**\_\_init\_\_**()  self.source = None,  self.target = None  self.value = None  def **get\_json\_data**(self):  return {  "id": self.source.id + ":" + self.target.id,  "source": self.source.id,  "target": self.target.id,  "value": self.value  }  class **Node**:  def **\_\_init\_\_**(self, tree):  super().**\_\_init\_\_**()  self.value = None  self.\_tree = tree  self.id = str(uuid.uuid1())  *# 设置关联的子数据集*  self.D = None  def **get\_tree**(self):  return self.\_tree  def **cut\_up\_childs**(self):  del self.\_tree.edges[self.id]  def **get\_class\_description**(self, class\_field):  data = self.D.groupby(class\_field)  res = [(item[0], **len**(item[1])) for item in data]  return res  def **has\_child**(self):  return self.id in self.\_tree.edges  def **\_\_str\_\_**(self):  u = (self.label, self.value)  return str(u)  def **get\_child\_by\_value**(self, value):  """  返回具有指定边值的子节点  """  edges = self.\_tree.edges[self.id]  for edge in edges:  if str(value) in [str(s) for s in edge.value]:  return edge.target  return None  def **get\_all\_child**(self):  """  返回该节点的所有子节点  """  if self.id not in self.\_tree.edges:  return []  edges = self.\_tree.edges[self.id]  return [ edge.target for edge in edges]  四、结果分析  数据集选用dataset目录下的car.csv，数据来源：  <https://archive.ics.uci.edu/ml/datasets/Car+Evaluation>  使用ID3算法对测试数据（抽取了训练集一部分数据作为测试数据）进行测试：    使用C4.5算法对测试数据（抽取了训练集一部分数据作为测试数据）进行测试：    准确率下方是决策树的结构序列化形式，用于图形界面中构建树  使用CART算法对测试数据（抽取了训练集一部分数据作为测试数据）进行测试：    两个准确率分别是剪枝前后的准确率，剪枝后准确率降低了，比较遗憾的是因时间原因未能改进。  七、问题及解决方案  在写决策树的过程中，引入了科学计算库numpy, pandas极大的简化了计算。与编写FP树时相同，代码结构也是边写边改，抽象出了一些公共的部分作为基类。在CART剪枝的算法中，对算法理解未能很到位，导致执行的结果与预想有一定的偏差。 | | | | | |

数据挖掘实验报告

实验序号：实验5　 实验项目名称：隐式马尔科夫

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 学　号 | 201601060928 | 姓　名 | 魏仲华 | 专业班级 | 软件2016-2 |
| 实验地点 | J13-132 | 指导教师 | 刘彤 | 实验时间 | 2019/5/14 |
| 一、运行环境  1. 操作系统： Windows 10 1803 64bit  2. 编程语言： Python3.6.4 64bit  3. 使用的类库：numpy  二、算法思想  本实验包含隐式马尔科夫问题（HMM）主要的两类问题，即评估问题和解码问题。  HMM定义：    对于评估问题，采用前向算法求解。    对于解码问题，采用维特比算法求解。    四、核心代码  import numpy as np  class **HMM**:  def **\_\_init\_\_**(self, pi, A, B, S, V):  *# 初始概率*  self.pi = pi  *# 隐状态概率转移矩阵*  self.A = A  *# 观测状态概率转移矩阵*  self.B = B  *# 隐藏状态数量*  self.N = **len**(S)  *# 可观测状态数量*  self.M = **len**(V)  self.S = S  self.V = V  self.S\_map = {self.S[i]: i for i in **range**(**len**(self.S))}  self.V\_map = {self.V[i]: i for i in **range**(**len**(self.V))}  def **get\_S\_index**(self, value):  return self.S\_map[value]  def **get\_V\_index**(self, value):  return self.V\_map[value]  def **\_forward\_algorithm**(self, observation):  """  前向算法  """  o = list(**map**(lambda x: self.get\_V\_index(x), observation))  alpha = np.empty((**len**(observation), self.N))  *# print(alpha)*  T = **len**(observation)  *# 1.初始化*  for i in **range**(self.N):  alpha[0][i] = self.pi[i] \* self.B[i, o[0]]  *# 2. 递归处理*  for t in **range**(1, T):  for j in **range**(self.N):  alpha[t][j] = **sum**([alpha[t-1][i] \* self.A[i][j] for i in **range**(self.N)]) \* self.B[j][o[t]]  **print**(alpha)  *# 3. 终结*  lambda\_ = **sum**([alpha[T-1][i] for i in **range**(self.N)])  return lambda\_  def **evaluation**(self, observation):  """  HMM第一类问题：评估问题求解  Parameters  ----------  observation : numpy.array  观测序列值  """  return self.\_forward\_algorithm(observation)    def **\_viterbi**(self, observation):  """  VITERBI 算法实现  """  o = list(**map**(lambda x: self.get\_V\_index(x), observation))  delta = np.empty((**len**(observation), self.N))  phi = np.empty((**len**(observation), self.N), dtype=int)  *# print(alpha)*  T = **len**(observation)  *# 1.初始化*  for i in **range**(self.N):  delta[0][i] = self.pi[i] \* self.B[i, o[0]]  phi[0][i] = 0  *# 2. 递归处理*  for t in **range**(1, T):  for j in **range**(self.N):  delta[t][j] = np.array([delta[t-1][i]\*self.A[i][j] for i in **range**(self.N)]).max() \* self.B[j][o[t]]  phi[t][j] = np.array([delta[t-1][i]\*self.A[i][j] for i in **range**(self.N)]).argmax()  **print**(phi)  **print**(delta)  *# 3. 终结*  p\_star = delta[T-1].max()  q\_t\_star = np.argmax(delta[T-1])  q\_star = q\_t\_star  res = []  res.append(self.S[q\_star])  for t in **range**(self.N)[::-1]:  q\_star = phi[t+1][q\_star]  res.append(self.S[q\_star])  return res[::-1]  def **decode**(self, observation):  """  HMM第二类问题：解码问题求解  Parameters:  ----------  observation : numpy.array  观测序列值  """  return self.\_viterbi(observation)    四、结果分析  评估问题测试：  def **test\_evalution\_assignment**(self):  pi = np.array([0.3,0.7])  **print**(pi)  A = np.array([[0.1,0.9],[0.8,0.2]])  B = np.array([[0.7,0.1,0.2],[0.3,0.5,0.2]])  S = ['吃', '睡']  V=["哭", "没精神", "找妈妈"]  hmm = HMM(pi, A, B, S, V)  observation = np.array(['哭', '没精神', '找妈妈'])  res = hmm.evaluation(observation)  **print**(res)  self.assertAlmostEqual(res, 0.026880000000000005)  解码问题测试：  def **test\_decode\_assignment**(self):  pi = np.array([0.3,0.7])  **print**(pi)  A = np.array([[0.1,0.9],[0.8,0.2]])  B = np.array([[0.7,0.1,0.2],[0.3,0.5,0.2]])  S = ['吃', '睡']  V=["哭", "没精神", "找妈妈"]  hmm = HMM(pi, A, B, S, V)  observation = np.array(['哭', '没精神', '找妈妈'])  res = hmm.decode(observation)  **print**(res)  self.assertEqual(res, ['吃', '睡', '吃'])  注：self.assertEqual中是运行结果  七、问题及解决方案  在编写的过程中，又一次感受到了Python科学计算库numpy的简洁之处。本实验相对简单，主要公式写正确就没问题了。 | | | | | |