《数据仓库与数据挖掘》实验报告

姓名：	朱志儒	学号：	SA20225085	日期：	2020/12/14
上机题目：	决策树 ID3 算法				

操作环境：
OS：Window 10
CPU：AMD Ryzen 5 3600X 6-Core Processor 4.25GHz
GPU：GeForce RTX 2070 super

一、基础知识：

决策树

决策树是一种树形结构，可以是二叉树或非二叉树，树中每个非叶节点表示一个特征属性上的测试，每个分支代表这个特征属性在某个值域上的输出，每个叶节点存放一个类别。使用决策树进行分类的过程就是从根节点开始，测试待分类项中相应的特征属性，按照其值选择输出分支，直到到达叶子节点，将叶子节点存放的类别作为分类结果。

ID3

ID3 模型是以信息熵和信息增益作为衡量标准的分类模型。

熵是指信息的混乱程度，熵值越大，变量的不确定性也就越大，计算信息熵的公式：

$$\mathrm{Entropy(S)} = -\sum_{i=1}^{m} p(u_i) log_2(p(u_i))$$

其中，$\mathrm{p}(u_i) = \frac{|u_i|}{|S|}$，$\mathrm{p}(u_i)$为类别$u_i$在样本$S$中出现的概率。

条件熵是指在已知第二个随机变量 X 的值的前提下，随机变量 Y 的信息熵。计算特征 A 对数据集 S 的条件熵的公式：

$$\mathrm{H(S|A)} = \sum_{V \in Value(A)} \frac{|S_V|}{|S|} Entropy(S_V)$$

其中，A 表示样本特征，Value(A)是特征 A 所有的取值集合，V 是 A 中一个特征值，S_V是 S 中 A 的值为 V 的样例集合。

信息增益是指在某个条件下，信息复杂度，即不确定性，减少的程度。计算信息增益的公式：

$$\mathrm{infoGain(S,A) = Entropy(S) - H(S|A)}$$

其中，A 表示样本特征。

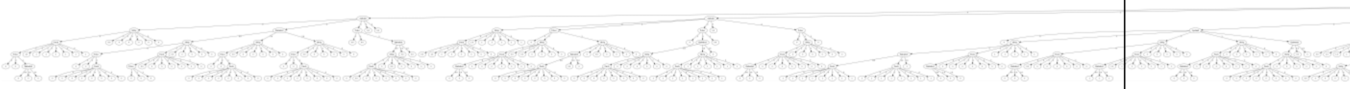
在构建决策树时，选择信息增益最大的特征作为决策点。

二、实验过程：

根据上次实验的方法处理训练集数据，依据这些数据构建决策树，由于整个数据的特征项并不是很多，所以构建好决策树后没有进行剪枝操作。最后，根据构建好的决策树对测试集数据进行分类，分类的过程就是从根节点进行搜索直到叶节点，叶节点的标签就是该数据的分类标签。

三、结果分析：

构建决策树，如图所示（原图见附件）



对测试集数据进行分类，准确率如下：

ID3准确率： 0.7583732057416268

附录

算法源代码（C/C++/JAVA 描述）：

```
def read_data_set():
    ''''''处理数据，提取特征'''
    trainData = pd.read_csv("train.csv")
    testData = pd.read_csv("test.csv")
    # 将训练集和测试集整合
    data = pd.concat([trainData, testData], axis=0).reset_index(drop=True)
    # male: 0, female: 1
    data['Sex'].replace(['male', 'female'], [0, 1], inplace=True)
    # S: 0, C: 1, Q: 2
    data['Embarked'].replace(['S', 'C', 'Q'], [0, 1, 2], inplace=True)

    # print(data[data['Fare'].isnull()])
    # Pclass: 3, Embarked: 0, Sex: 0
    # 填补 Fare 为 NaN 的数据
    data['Fare'] = data['Fare'].fillna(
        np.mean(data[((data['Pclass'] == 3) & (data['Embarked'] == 0) & (data['Sex'] == 0))]['Fare']))
    # print(data[data['Fare'].isnull()])
    # Empty DataFrame

    # data['FareLimit'] = pd.qcut(data['Fare'], 4)
    # print(data.groupby(['FareLimit'])['Survived'].mean())
    # 使用 FareLimit 替代 Fare
```

```python
    data['FareLimit'] = 0
    data.loc[data['Fare'] <= 8.662, 'FareLimit'] = 0
    data.loc[(data['Fare'] > 8.662) & (data['Fare'] <= 14.454), 'FareLimit'] = 1
    data.loc[(data['Fare'] > 14.454) & (data['Fare'] <= 53.1), 'FareLimit'] = 2
    data.loc[data['Fare'] > 53.1, 'FareLimit'] = 3

    # print(data[data['Embarked'].isnull()])
    # Pclass: 1, Sex: 1
    # 填补 Embarked 为 NaN 的数据
    data['Embarked'] = data['Embarked'].fillna(
        stats.mode(data[((data['Pclass'] == 1) & (data['Sex'] == 1))]['Embarked'])[0][0])
    # print(data[data['Embarked'].isnull()])
    # Empty DataFrame

    # 添加新特征：家人 Family
    data['Family'] = data['SibSp'] + data['Parch']

    # 填补 Age 为 NaN 的数据
    dataAgeNanIndex = data[data['Age'].isnull()].index
    for i in dataAgeNanIndex:
        # 取 Pclass、Family 相同的数据的平均值
        meanAge = data['Age'][
            (data['Pclass'] == data.iloc[i]['Pclass']) & (data['Family'] == data.iloc[i]['Family'])].mean()
        data['Age'].iloc[i] = meanAge

    # data['AgeLimit'] = pd.cut(data['Age'], 5)
    # print(data.groupby(['AgeLimit'])['Survived'].mean())

    # 使用 AgeLimit 替代 Age
    data['AgeLimit'] = 0
    data.loc[data['Age'] <= 16, 'AgeLimit'] = 0
    data.loc[(data['Age'] > 16) & (data['Age'] <= 32), 'AgeLimit'] = 1
    data.loc[(data['Age'] > 32) & (data['Age'] <= 48), 'AgeLimit'] = 2
    data.loc[(data['Age'] > 48) & (data['Age'] <= 60), 'AgeLimit'] = 3
    data.loc[data['Age'] > 60, 'AgeLimit'] = 4

    # 删除无用列
```

```
    data.drop(labels=["Age", "Fare", "Ticket", "Cabin", "Name", "PassengerId", 'Family'], axis=1, inplace=True)

    data = data[['Pclass', 'Sex', 'SibSp', 'Parch', 'Embarked', 'FareLimit', 'AgeLimit', 'Survived']]

    trainX = data[:len(trainData)].values
    testY = data[len(trainData):]['Survived'].values.tolist()
    testX = data[len(trainData):].drop(labels='Survived', axis=1).values

    return trainX, testX, testY


def empirical_entropy(train_set, D):
    ''''根据数据集train_set计算类别D的信息熵'''
    dict_of_kinds = {}
    for i in range(len(train_set)):
        label = train_set[i][D]
        if label not in dict_of_kinds.keys():
            dict_of_kinds[label] = 1
        else:
            dict_of_kinds[label] += 1
    summ = len(train_set)
    for key in dict_of_kinds.keys():
        pd = dict_of_kinds[key] / summ
        dict_of_kinds[key] = pd * math.log(pd)
    return -sum(list(dict_of_kinds.values()))


def condition_entropy(train_set, A, D):
    ''''根据数据集train_set，在已知条件A的前提下，计算D的条件熵'''
    dict_of_kinds = {}
    for i in range(len(train_set)):
        label = train_set[i][A]
        if label not in dict_of_kinds.keys():
            dict_of_kinds[label] = [i]
        else:
            dict_of_kinds[label].append(i)
    for key in dict_of_kinds.keys():
        dict_of_acct = {}
        for j in dict_of_kinds[key]:
```

```python
            label = train_set[j][D]
            if label not in dict_of_acct.keys():
                dict_of_acct[label] = 1
            else:
                dict_of_acct[label] += 1
        summ = len(dict_of_kinds[key])
        for keyy in dict_of_acct.keys():
            pd = dict_of_acct[keyy] / summ
            dict_of_acct[keyy] = pd * math.log(pd)
        dict_of_kinds[key] = len(dict_of_kinds[key]) / len(train_set) * (-sum(list(dict_of_acct.values())))
    return sum(list(dict_of_kinds.values()))


def informatin_gain(train_set, D, A):
    '''''计算信息增益'''
    return empirical_entropy(train_set, D) - condition_entropy(train_set, A, D)


def get_child_set(items):
    '''''求 items 的非空真子集'''
    result = [[]]
    for x in items:
        result.extend([subset + [x] for subset in result])
    return result[1:len(result) - 1]


class decision_node:
    '''''定义节点类'''

    def __init__(self, col, value=None, child_node=None):

        self.col = col
        self.value = value
        self.child_node = child_node


class Decision_tree:
    '''''定义决策树类'''

    def __init__(self, train_set, D, dict_of_labels, function, dict_of_col):
```

```
        self.labels = [i for i in range(len(dict_of_labels.keys()))]
        self.train_set = train_set
        self.D = D
        self.dict_of_labels = dict_of_labels
        self.function = function
        self.dict_of_col = dict_of_col
        self.decision_tree = self.build_tree(train_set, [], function)

    def is_same(self, remain_set):
        ''''数据集 D 中的样本属于同一类别 C，则将当前结点标记为 C 类叶结点'''
        acct = remain_set[0][self.D]
        for row in remain_set:
            if acct != row[self.D]:
                return False
        return True

    def classify(self, test_set):
        ''''根据已生成的决策树将 test_set 中的数据分类'''
        results = []
        for row in test_set:
            head = self.decision_tree
            while head.col != -1:
                for key in head.child_node.keys():
                    if key == row[head.col]:
                        head = head.child_node[key]
                        break
            results.append(head.value)
        return results

    def find_mode(self, remain_set):
        ''''找出 remain_set 中出现次数最多的类别'''
        result = [0, 0]
        for line in remain_set:
            result[int(line[self.D])] += 1
        if result[0] > result[1]:
            return 0
        else:
            return 1

    def build_tree(self, remain_set, used_col, function):
```

```
        '''''构建决策树'''
        if self.is_same(remain_set):
            '''''数据集D中的样本属于同一类别C，则将当前结点标记为C类叶结点'''
            return decision_node(-1, value=remain_set[0][self.D])
        if len(used_col) == len(self.dict_of_labels.keys()):
            '''''特征集A为空集，或数据集D中所有样本在A中所有特征上取值相同，此时无法划分。将当前结点标记为叶结点，类别为D中出现最多的类'''
            return decision_node(-1, value=self.find_mode(remain_set))

        '''''构建ID3模型决策树'''
        entropies = []
        for i in self.labels:
            if i not in used_col:
                entropies.append(function(remain_set, self.D, i))
            else:
                entropies.append(-1)
        '''''选择信息增益最大的特征作为决策点'''
        choose = entropies.index(max(entropies))
        new_used_col = used_col + [choose]
        child_node = {}
        for label in self.dict_of_labels[choose]:
            new_remain_set = []
            for row in remain_set:
                if row[choose] == label:
                    new_remain_set.append(row)
            if len(new_remain_set) == 0:
                '''''数据集D为空集，则将当前结点标记为叶结点，类别为父结点中出现最多的类'''
                child_node[label] = decision_node(-1, value=self.find_mode(remain_set))
            else:
                child_node[label] = self.build_tree(new_remain_set, new_used_col, function)
        return decision_node(choose, child_node=child_node)

    def visit_node(self, graph, father_col, node, strings):
```

```
        # 绘制决策树时访问节点函数
        if node.col != -1:
            new_col = father_col + self.dict_of_col[node.col] + str(strings)
            graph.node(new_col, self.dict_of_col[node.col])
            graph.edge(father_col, new_col, str(strings))
            for key in node.child_node.keys():
                self.visit_node(graph, new_col, node.child_node[key], key)
        else:
            new_col = father_col + str(node.value) + str(strings)
            graph.node(new_col, str(node.value))
            graph.edge(father_col, new_col, str(strings))

    def draw_tree(self):
        # 绘制决策树的图像
        tree_name = 'ID3_Decision_Tree.gv'
        graph = Digraph(tree_name, format='png')
        node = self.decision_tree
        graph.node(str(node.col), self.dict_of_col[node.col])
        for key in node.child_node.keys():
            self.visit_node(graph, str(node.col), node.child_node[key], key)
        graph.render(tree_name, view=True)

def validation(testY, result, function_name):
    count = 0
    for i in range(len(result)):
        if int(result[i]) == int(testY[i]):
            count += 1
    print(function_name + '准确率:', count / len(result))


if __name__ == '__main__':
    dict_of_col = {0: 'Pclass', 1: 'Sex', 2: 'SibSp', 3: 'Parch', 4: 'Embarked', 5: 'FareLimit', 6: 'AgeLimit'}
    dict_of_train_labels = {0: [1, 2, 3], 1: [0, 1], 2: [0, 1, 2, 3, 4, 5, 8], 3: [0, 1, 2, 3, 4, 5, 6, 9],
```

	`` 4: [0.0, 1.0, 2.0], 5: [0, 1, 2, 3], 6: [0, 1, 2, 3, 4]} train_set, test_set, testY = read_data_set() dt_ig = Decision_tree(train_set, 7, dict_of_train_labels, informatin_gain, dict_of_col) dt_ig.draw_tree() re1 = dt_ig.classify(test_set) validation(testY, re1, 'ID3') ```