1. 主要问题描述

生活中我们经常会遇到各种各样的预测分类问题，比如垃圾分类的时候，要预测他是干垃圾、湿垃圾、厨余垃圾还是有害垃圾，又比如根据今天晚上有没有晚霞或者乌云多不多来预测明天是晴天、阴天还是雨天。垃圾分类是比较简单的分类，但很多分类问题更加复杂，比如给出一个客户一年的消费记录，如何预测他是核心客户，还是普通客户，是值得信任的客户，还是信用不好的客户，这些是人工难以计算的。而分类会带来的好处，也很多，比如抓准了核心客户，就可以有针对性地拉拢，就不必广撒网，就能节省人力物力同时还获得不错的收益，如果预测准了明天的天气，就能提前做好计划，不至于刚刚准备出门郊游就下起雷阵雨。如果能根据通话记录来筛选出可疑人员，就能让警方特意留意，说不定就能捣毁一个电话诈骗窝点或者是传销团伙。所以这就促使了决策树和随机森林分类算法的诞生。

2. 方法描述

2.1. 决策树

决策树（decision tree）是一类常见的机器学习方法，核心思路是从给定训练数据集学得一个类似于流程图的树型模型用以对新示例进行分类，每个节点表示在一个属性上的测试，每个分支代表该测试的输出，每个树叶节点代表类或类分布。决策树是基于树结构来进行决策的，符合人类在面临决策性问题时一种很自然的处理机制，所以他自然而然地具有很强的可解释性。根据分裂方式的不同，决策树也有不同的分类，比较常见的方式是 ID3 决策树和 CART 决策树。

ID3 决策树：

使用信息增益来评判属性的分类效果，首先介绍信息熵:令 p_I 为 D 中的任一元组属于类C_i概率,估计为$\frac{|C_I,D|}{|D|}$。D 中元组分类需要的信息熵(entropy) :$info(D) = -\Sigma_{i=1}^{m} p_i \log_2 p_i$。然后(利用 A 分裂 D 为 v 个部分后)分类 D 需要的信息

为:$info_{\mathrm{A}}(D=\frac{\Sigma_{j=1}^{v}|D_j|}{|D|}*info(D_j)$。以属性 A 分枝得到的信息增益：$Gain(A)=info(D)-info_A(D)$。

CART 决策树：

信息增益倾向于有大量不同取值的属性（划分更细，更纯），在极端情况下每个划分子集只有一个样本，即一个类，此时 Info(d)=0，按照 ID3 的评判方式就是最好的，但是并不一定实际使用中就是最好的。CART 决策树使用 Gini 指数来避免这一问题。

Gini 指数度量数据元组的不纯度。数据 D 包含 n 类别的样本, Gini(D) 定义为:$\mathrm{gini(D)}=1-\Sigma_{\mathrm{j}=1}^{n}p_j^2$，$p_j$为类别 j 在数据 D 中的频率，也就是所占百分比。数据集 D 基于属性 A 分裂为子集 $\mathrm{D_1}$ 和 $\mathrm{D_2}$, gini 指标定义为：$\mathrm{gini_A(D)}=\frac{|\mathrm{D_1}|}{|\mathrm{D}|}gini(D_1)+\frac{|D_2|}{|D|}gini(D_2),\mathrm{gini(A)}=\mathrm{gini(D)}-\mathrm{gini_A(D)}$。具有最小 gini 值的属性将用于分裂节点。

2.2. 随机森林

上面介绍的决策树在实际运用过程中也存在一些问题：泛化性能差，过拟合，分类精度差等，催生了基于决策树改进的随机森林算法。

随机森林就是通过集成学习的思想将多棵树集成的一种算法，它的基本单元是决策树，而它的本质属于机器学习的一大分支——集成学习（Ensemble Learning）方法。集成学习通过建立几个模型组合的来解决单一预测问题。它的工作原理是生成多个分类器/模型，各自独立地学习和作出预测。这些预测最后结合成单预测，因此优于任何一个单分类的做出预测。

随机森林的名称中有两个关键词，一个是“随机”，一个就是“森林”。“森林”我们很好理解，一棵叫做树，那么成百上千棵就可以叫做森林了。

从直观角度来解释，每棵决策树都是一个分类器（假设现在针对的是分类问题），那么对于一个输入样本，N 棵树会有 N 个分类结果。而随机森林集成了所有的分类投票结果，将投票次数最多的类别指定为最终的输出。

前面提到，随机森林中有许多的分类树。我们要将一个输入样本进行分类，我们

需要将输入样本输入到每棵树中进行分类。打个形象的比喻：森林中召开会议，讨论某个动物到底是老鼠还是松鼠，每棵树都要独立地发表自己对这个问题的看法，也就是每棵树都要投票。该动物到底是老鼠还是松鼠，要依据投票情况来确定，获得票数最多的类别就是森林的分类结果。森林中的每棵树都是独立的，99.9%不相关的树做出的预测结果涵盖所有的情况，这些预测结果将会彼此抵消。少数优秀的树的预测结果将会超脱于芸芸“噪音”，做出一个好的预测。

将若干个弱分类器的分类结果进行投票选择，从而组成一个强分类器，这就是随机森林 bagging 的思想（关于 bagging 的一个有必要提及的问题：bagging 的代价是不用单棵决策树来做预测，具体哪个变量起到重要作用变得未知，所以 bagging 改进了预测准确率但损失了解释性。）。

3. 算法流程

3.1. 决策树算法流程

总体流程：使用自顶向下的分治方式构造决策树，使用分类属性（如果是量化属性，则需先进行离散化）递归的通过选择相应的测试属性，来划分样本。测试属性是根据某种启发信息或者是统计信息来进行选择（如：信息增益），ID3 和 CART 的算法流程的主要区别也就在这里。同时 ID3 是有多少个属性值分裂出多少个儿子节点，CART 每次只会分裂成两个。

具体流程：

- 树以代表训练样本的单个结点开始。
- 如果样本都在同一个类. 则该结点成为树叶，并用该类标记。
- 否则，算法选择最有分类能力的属性作为决策树的当前结点.
- 根据当前决策结点属性取值的不同，将训练样本数据集分为若干子集，每个取值形成一个分枝，有几个取值形成几个分枝。针对上一步得到的一个子集，重复进行先前步骤，递归形成每个划分样本上的决策树。一旦一个属性出现在一个结点上，就不必在该结点的任何后代考虑它。
- 递归划分步骤仅当下列条件之一成立时停止：

  ①给定结点的所有样本属于同一类。

②没有剩余属性可以用来进一步划分样本。

③如果某一分枝，没有满足该分支中已有分类的样本，则以样本的多数类创建一个树叶。

算法伪代码：

```
算法：Generate_decision_tree。由数据分区D中的训练元组产生决策树。
输入：
  • 数据分区D，训练元组和它们对应类标号的集合。
  • attribute_list，候选属性的集合。
  • Attribute_selection_method，一个确定“最好地”划分数据元组为个体类的分裂准则的过程。
    这个准则由分裂属性（splitting_attribute）和分裂点或划分子集组成。
输出：一棵决策树。
方法：
(1)  创建一个结点N;
(2)  if D中的元组都在同一类C中then
(3)    返回N作为叶结点,以类C标记;
(4) if attribut_list为空then
(5)    返回N作为叶结点，标记为D中的多数类;                    //多数表决
(6) 使用Attribute_selection_method(D, attribute_list),找出“最好的”splitting_criterion;
(7) 用splitting_criterion标记结点N;
(8)  if splitting_attribute是离散值的,并且允许多路划分then      //不限于二叉树
(9)    attribute_list←attribute_list-splitting_attribute; // 删除分裂属性
(10) for splitting_criterion的每个输出j
     //划分元组并对每个分区产生子树
(11)   设Dj是D中满足输出j的数据元组的集合;                    // 一个分区
(12)   if Dj为空then
(13)     加一个树叶到结点N,标记为D中的多数类;
(14)   else加一个由Generate_decision_tree(Dj,attribute_list)返回的结点到N;
     endfor
(15) 返回N;
```

3.2. 随机森林算法流程

算法流程：

- 原始训练集大小为 N，应用 bootstrap 法有放回地每次随机抽取 N 个样本，抽取 k 次，由此构建 k 棵分类树（CART），未被抽到的样本组成了 k 个袋外数据；
- 假设有 M 个变量，在每一棵树的每个节点处随机抽取 m 个变量（m 远小于 M），然后在 m 中选择一个最具分类能力的变量；
- 自顶向下递归分裂，按照基尼指数最小原则，直到满足分支停止规则而停止生长，不做任何剪枝；
- 将生成的多棵分类树组成随机森林，用随机森林分类器对新的数据进行判别与分类，分类结果按树分类器的投票多少决定。

伪代码：

Algorithm 15.1 *Random Forest for Regression or Classification.*

1. For $b = 1$ to B:
   (a) Draw a bootstrap sample $\mathbf{Z}^*$ of size N from the training data.
   (b) Grow a random-forest tree T_b to the bootstrapped data, by recursively repeating the following steps for each terminal node of the tree, until the minimum node size n_{min} is reached.
      i. Select m variables at random from the p variables.
      ii. Pick the best variable/split-point among the m.
      iii. Split the node into two daughter nodes.
2. Output the ensemble of trees $\{T_b\}_1^B$.

To make a prediction at a new point x:

Regression: $\hat{f}_{\text{rf}}^B(x) = \frac{1}{B}\sum_{b=1}^{B} T_b(x)$.

Classification: Let $\hat{C}_b(x)$ be the class prediction of the bth random-forest tree. Then $\hat{C}_{\text{rf}}^B(x) = \textit{majority vote}\ \{\hat{C}_b(x)\}_1^B$.

4. 核心算法代码

4.1. 决策树代码

ID3 建树代码：

```
private TreeNode rootTree_ID3(Set<Record> records,Set<String> attrN
ames){
    TreeNode root=new TreeNode();
    root.records=records;
    //同属一个类别，标记后返回
    if(haveSameLabel(records)) {
        for (Record record : records) {
            root.label = record.label;
            return root;
        }
    }

    //没有可用来划分的属性或者所有记录在可用属性都有相同的值
    if(attrNames.size() == 0 || haveSameValue(records,attrNames)){
        root.label=findTheMostLabel(records);//找出最多的类别
        return root;
    }
```

```
    String name = findMaxPartition(records,attrNames);//找到信息增益
最大的特征

    if(name.equals("")){
        name = attrNames.stream().findFirst().orElse(name);
    }
    root.attrName=name;

    //构建新的可用属性集合（去除本节点使用的）
    Set<String> newAttrNames = new HashSet<>(attrNames);
    newAttrNames.remove(name);
    //按照信息增益最大的属性划分集合，并且每一个集合产生一个儿子结点
    Map<String,Set<Record>> divSets=divByAttr(records,name);
    for(Map.Entry<String,Set<Record>> entry: divSets.entrySet()){
        AttributeField attributeField = new AttributeField();
        attributeField.name=name;
        if(!isContinuous(name)) {
            attributeField.attrValueDisc = entry.getKey();
        }
        else{
            attributeField.attrValueCont = entry.getKey();
        }
        root.children.put(attributeField,rootTree_ID3(entry.getValu
e(),newAttrNames));
    }
    return root;
}
private String findMaxPartition(Set<Record> records, Set<String> at
trNames){
    double maxGain=0;//记录最大信息增益和对应属性名
    String maxGainName="";
    //枚举可用属性名称，寻找信息增益最大的那一个
    for(String attrName : attrNames){
        double gain= gainPartition(records,attrName);
        if(gain > maxGain){
            maxGain=gain;
            maxGainName=attrName;
        }
    }
    return maxGainName;
}
//按照某个属性划分之后，信息增益的大小
private double gainPartition(Set<Record> records, String attributeN
ame){
```

```
    double gain=entropy(records);
    Map<String,Set<Record>> count = divByAttr(records,attributeName
);

    //累加每种值的贡献
    for(Map.Entry<String,Set<Record>> entry: count.entrySet()){
        gain-
=(double)entry.getValue().size()/records.size()*entropy(entry.getVa
lue());
    }
    return gain;
}
```

CART 建树代码：

```
private TreeNode rootTree_CART(Set<Record> records,Set<String> attr
s,Map<String,Set<String>> values){
    TreeNode root=new TreeNode();
    root.records=records;
    //同属一个类别，标记后返回
    if(haveSameLabel(records)) {
        for (Record record : records) {
            root.label = record.label;
            return root;
        }
    }

    //所有记录在可用属性都有相同的值
    if(haveSameValue(records,attrs)){
        root.label=findTheMostLabel(records);
        return root;
    }

    //Key 为属性名，value 为值
    Pair<String,String> pair = findMinGini(records,attrs,values);

    if(pair.getKey().equals("")){
        System.out.println("pair.getKey 为空");
//            pair.getKey() = attributesSet.stream().findFirst().or
Else(name);
    }
    root.attrName=pair.getKey();

    //构建新的可用属性集合（去除本节点使用的）
    Set<String> newAttrNames = new HashSet<>(attrs);
    Map<String,Set<String>> newValues = new HashMap<>(values);
```

```
    if(values.get(root.attrName).size() == 2){//使用的属性只有两个值
了，应该将这个属性删除
        newAttrNames.remove(root.attrName);
        //此处偷懒，不删 values 中的键值对，因为后面也不会用到
    }
    else{
        //属性剩余值多于两个，删除属性值，不删除属性
        newValues.get(root.attrName).remove(pair.getValue());
    }

    //按照信息增益最大的属性划分集合，按照等于值和不等于，划分为两个儿子节点
    Map<String,Set<Record>> divSets = divByAttr_CART(records,pair.g
etKey(),pair.getValue());

    for(Map.Entry<String,Set<Record>> entry: divSets.entrySet()){
        AttributeField attributeField = new AttributeField();
        attributeField.name=pair.getKey();
        if(!isContinuous(pair.getKey())) {
            attributeField.attrValueDisc = entry.getKey();
        }
        else{
            attributeField.attrValueCont = entry.getKey();
        }
        root.children.put(attributeField,rootTree_ID3(entry.getValu
e(),newAttrNames));
    }
    return root;
}
//按照某种属性划分的 Gini 系数
private Pair<Double,String> Gini(Set<Record> records, String attrNa
me, Set<String> v){
    double result = Double.MAX_VALUE;
    String value="";
    for(String s : v){//遍历每个属性值
        //划分为两部分
        Set<Record> tmp = new HashSet<>();//等于这个属性值的事务
        Set<Record> left = new HashSet<>();//不等于这个属性值的事务
        for(Record record : records){//把每个事务按照是否等于该属性值划分
            String curv = getValue(attrName,record);
            if(curv.equals(s)) tmp.add(record);
            else left.add(record);
        }
        double ans=(double)tmp.size()/records.size()*Gini(tmp)+(dou
ble)left.size()/records.size()*Gini(left);
```

```
        if(ans < result){
            result = ans;
            value = s;
        }
    }
    return new Pair<>(result,value);
}

//给定记录、属性、和属性的可选值，计算最小的基尼指数对应的属性名称，返回属性名和对应的值
private Pair<String,String> findMinGini(Set<Record> records, Set<String> attrs, Map<String,Set<String>> values){
    double result = Double.MAX_VALUE;
    String value = "";
    String answer = "";
    for(String attr : attrs){//枚举属性
        Pair<Double,String> cur = Gini(records, attr, values.get(attr));
        if(cur.getKey() < result){
            result = cur.getKey();
            value = cur.getValue();
            answer = attr;
        }
    }
    return new Pair<>(answer,value);
}
```

随机森林建树代码：单颗树的构造代码，只需要反复调用并保存树根就可建立随机森林

```
//构建随机森林
private TreeNode rootForest(Set<Record> records,int m,int deep){
    TreeNode root=new TreeNode();
    root.records=records;
    //同属一个类别，标记后返回
    if(haveSameLabel(records)) {
        for (Record record : records) {
            root.label = record.label;
            return root;
        }
    }
    Set<String> chosen = randomChoose(attributes,m);//随机选择 m 个

    //所有记录在随机选择的属性都有相同的值
    if(haveSameValue(records,chosen)){
```

```
        root.label=findTheMostLabel(records);
        return root;
    }

    Map<String,Set<String>> values = new HashMap<>();
    for(String attr : chosen){
        values.put(attr,ValueOfAttrs.get(attr));
    }

    Pair<String,String> pair = findMinGini(records,chosen,values);
    root.attrName=pair.getKey();
    Map<String,Set<Record>> divSets=divByAttr_CART(records,pair.get
Key(),pair.getValue());

    //和普通建树一样的划分过程
    for(Map.Entry<String,Set<Record>> entry: divSets.entrySet()){
        AttributeField attributeField = new AttributeField();
        attributeField.name=root.attrName;
        if(!isContinuous(root.attrName)) {
            attributeField.attrValueDisc = entry.getKey();
        }
        else{
            attributeField.attrValueCont = entry.getKey();
        }
        root.children.put(attributeField,rootForest(entry.getValue(
),m,deep+1));
    }
    return root;
}
```

分类代码：所有建树方法在分类的时候使用的都是同一个分类函数

```
//对一条新记录预测其分类
private String classify(TreeNode root,Record record){
    if(root.isLeaf()){
        return root.label;
    }
    //连续的属性需要划分为离散值
    String value = getValue(root.attrName,record);//该条新记录中该节点
分支属性的值

    //value 该分支到哪个儿子
    int pass=0;//记录分支中是否有"@else"字符串，为下一步做准备
    for(Map.Entry<AttributeField, TreeNode> entry : root.children.e
ntrySet()){
```

```
        if(!isContinuous(root.attrName) && entry.getKey().attrValue
Disc.equals(value)){//判断该分支属性值是否和该记录相同
            return classify(entry.getValue(), record);
        }
        else if(isContinuous(root.attrName) && entry.getKey().attrV
alueCont.equals(value)){
            return classify(entry.getValue(), record);
        }
        String cv=entry.getKey().attrValueCont;
        String dv=entry.getKey().attrValueDisc;
        if(cv!=null && cv.equals("@else")) ++pass;
        else if(dv!=null && dv.equals("@else")) ++pass;
    }

    //如果到达这里，说明没有匹配上，而有 else 的情况下（CART 建树），就一定是往
else 的分支走
    if(pass==1){
        for(Map.Entry<AttributeField, TreeNode> entry : root.childr
en.entrySet()){
            String cv=entry.getKey().attrValueCont;
            String dv=entry.getKey().attrValueDisc;
            if((cv!=null && cv.equals("@else")) || (dv!=null && dv.
equals("@else"))){
                return classify(entry.getValue(), record);
            }
        }
    }

    //仍然没有匹配上，就随机划分(划分给第一个)，因为这说明训练集中没有对应的值
//        System.out.printf("分类出现错误，出现了训练时没有出现的属性值，属
性名称:%s,属性值:%s\n",root.attrName,value);
    for(Map.Entry<AttributeField, TreeNode> entry : root.children.e
ntrySet()){
        return classify(entry.getValue(), record);
    }
    return "";
}
```

5. 实验结果截图及分析

使用 iris 文件训练，iris 文件测试的结果：

ID3 决策树：耗时 60ms，测试集共 150 个数据，共正确 149 个，错误 1 个，

正确率 99.333%。总的来说正确率非常不错。混淆矩阵如下：

实际 \ 预测	Iris-setosa	Iris-virginica	Iris-versicolor	总计
Iris-setosa	50	0	0	50
Iris-virginica	0	49	1	50
Iris-versicolor	0	0	50	50
总计	50	49	51	150

CART 决策树：耗时 64ms，测试集共 150 个数据，共正确 149 个，错误 1 个，**正确率 99.333%**。总的来说正确率非常不错。混淆矩阵如下：

实际 \ 预测	Iris-setosa	Iris-virginica	Iris-versicolor	总计
Iris-setosa	50	0	0	50
Iris-virginica	0	49	1	50
Iris-versicolor	0	0	50	50
总计	50	49	51	150

随机森林：建树 100 棵，取 m=2（每次随机两个属性），耗时 335ms，测试集共 150 个数据，共正确 149 个，错误 1 个，正确率 99.333%。正确率非常不错，混淆矩阵如下：

实际 \ 预测	Iris-setosa	Iris-virginica	Iris-versicolor	总计
Iris-setosa	50	0	0	50
Iris-virginica	0	49	1	50
Iris-versicolor	0	0	50	50
总计	50	49	51	150

使用 adult.data 文件训练，adult.test 文件测试的结果：

ID3 决策树：耗时 2001ms。测试集共 16281 个数据，共正确 13036 个，错误 3245 个，正确率 **80.069%**。正确率还算高，混淆矩阵如下：

实际 \ 预测	<=50K	>50K	总计

<=50K	11035	1400	12435
>50K	1845	2001	3846
总计	12880	3401	16281

CART 决策树：耗时 3655ms。测试集共 16281 个数据，共正确 13070 个，错误 3211 个，正确率 **80.278%**。正确率还算高，混淆矩阵如下：

实际 \ 预测	<=50K	>50K	总计
<=50K	11049	1386	12435
>50K	1825	2021	3846
总计	12874	3407	16281

随机森林：建立 100 棵决策树，m=3，耗时 239860ms。测试集共 16281 个数据，共正确 13778 个，错误 2503 个，正确率 **84.626%**，正确率比单独用决策树高，混淆矩阵如下：

实际 \ 预测	<=50K	>50K	总计
<=50K	11595	840	12435
>50K	1663	2183	3846
总计	13258	3023	16281

分析：可以发现，在运行时间上，随机森林因为要建很多棵树，所以时间是单独的决策树的好几倍，而 ID3 决策树和 CART 决策树时间差不多。在正确率上，某些噪声较小的数据，随机森林和决策树正确率都很高，然而在一些有噪声的数据里，随机森林的正确率比决策树高，总的来说是用时间换了正确率，在需要高一点正确率又不需要很高的效率的时候，随机森林是一个不错的选择，否则的话，决策树会是一个比较好的选择。

6. 总结

本次实验，主要的难点在代码编写上，算法并不是很难，数学计算也不复杂，决策树和随机森林加一起的代码有 700 多行，文件大小有 28K，对编码能力是一个

挑战，经常写着写着感觉脑容量不够了，忘记要干什么，可能这就是工程中会出现的问题，为了解决这种情况，我写了一个工作记录，把打算要做的工作在计划的时候全部写下来，这样在一部分做完之后就可以看文件，就可以知道下一步要做什么了。最开始写 CART 决策树的时候把流程弄错了，除了找属性，其他和 ID3 都是一样的（实际上 CART 每次只分裂出两个儿子节点），导致后面也改了不少代码，经验就是写之前一定要弄清楚，改代码又慢又不想改。

本次实验由于时间不是很充裕，在时间和正确率上并没有做多少优化，最后 adult 的 85%正确率也感觉不是非常满意，以后可能会再进行优化。