基于决策树的英雄联盟游戏胜负预测

实验报告分享

com.con.

Apr 2, 2022

任务流程和实验框架

两条主线：

- 数据处理：知道该怎么处理比知道怎么实现数据处理更重要.
  - 数据的容器：csv file $\longrightarrow$ pd.DataFrame $\longrightarrow$ np.ndarray $\longrightarrow$ list
  - 数据的解释：sample $\longrightarrow$ feature $\longrightarrow$ interval $\longrightarrow$ *feature_type
- 算法实现：根据算法的需求决定数据结构的设计.
  - 递归调用：进度的更新 vs 回溯
  - 数据结构：信息的传递 vs 保存

数据描述与处理

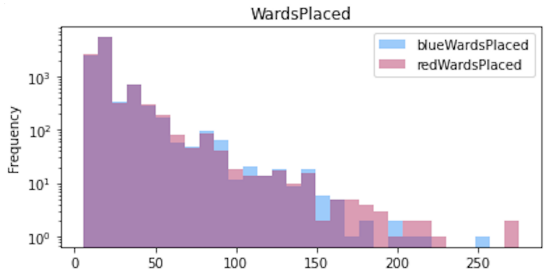
数据描述

要知道怎么处理数据，首先要了解数据的形式. 以 *WardsPlaced 为例:

```
for c in df:
    print(df[c].value_counts())
```

```
16     1255
15     1217
17      988
14      974
18      831
       ...
165       1
120       1
148       1
111       1
137       1
Name: blueWardsPlaced, Length: 147, dtype: int64
```

```
cols = ['blueWardsPlaced', 'redWardsPlaced']
df[cols].plot.hist(ax=axs[0,0], bins=30,
                   color=['#239CFF', '#CF366C'],
                   log=True, alpha=0.5)
```



数据描述与处理

数据描述

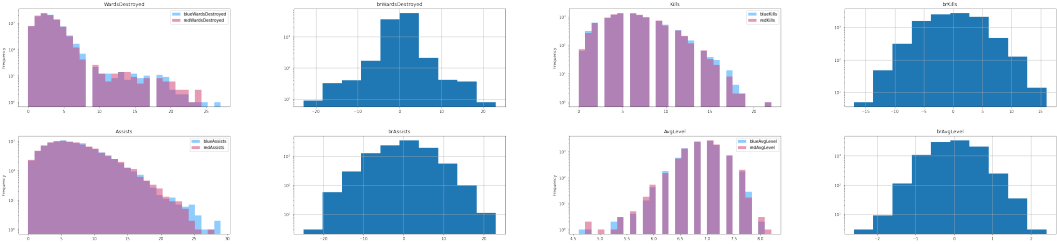
通过上述对取值范围的观察，可以考虑把特征分成两类：

```
# 选择各方法所处理的特征
discrete_features = ['blueWins',  # {0, 1}
                     'brFirstBlood',  # {-1, 1}
                     'blueEliteMonsters', 'redEliteMonsters', 'brEliteMonsters',  # {0, 1, 2}
                     'blueDragons', 'redDragons', 'brDragons',  # {0, 1}
                     'blueHeralds', 'redHeralds', 'brHeralds',  # {0, 1}
                     'blueTowersDestroyed', 'redTowersDestroyed', 'brTowersDestroyed',  # {0, 1, 2, 3, 4}
                     ]  # 具有少数几个可能取值的特征，可以直接使用
q_features = ['blueWardsPlaced', 'redWardsPlaced', 'brWardsPlaced',
              'blueWardsDestroyed', 'redWardsDestroyed', 'brWardsDestroyed',
              'blueTotalMinionsKilled', 'redTotalMinionsKilled', 'brTotalMinionsKilled',
              'blueTotalJungleMinionsKilled', 'redTotalJungleMinionsKilled', 'brTotalJungleMinionsKilled',
              'blueKills', 'redKills', 'brKills',
              'blueDeaths', 'redDeaths', 'brDeaths',
              'blueAssists', 'redAssists', 'brAssists',
              'blueTotalGold', 'redTotalGold', 'brTotalGold',
              'blueAvgLevel', 'redAvgLevel', 'brAvgLevel',
              'blueTotalExperience', 'redTotalExperience', 'brTotalExperience',
              ]  # 连续分布的特征，考虑采用分位数离散化
```

数据描述与处理

数据描述

对于取值数目众多的数据类型，考虑采用分位数对其进行离散化. 这需要观察其分布:



观察分布特点，我们采用以下分位点进行离散化:

```
QUANTILES = [.05, .25, .5, .75, .95]  # 分位点
```

数据描述与处理

数据描述

之前的图像中的红蓝色只为了少画两张图，实际上没有可比性……对数据进一步处理：

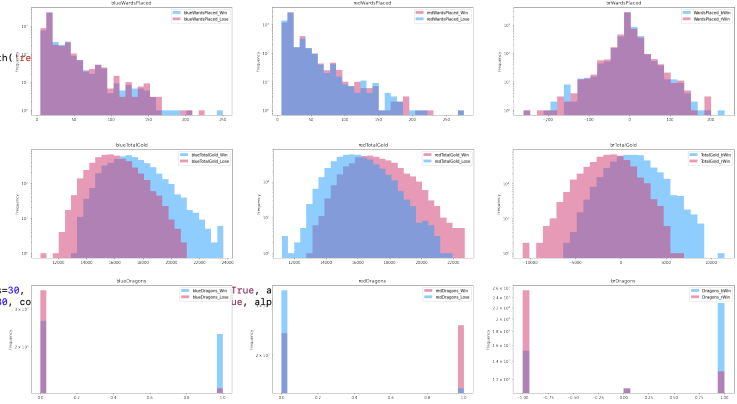
```
info_names = [c[3:] for c in df.columns if c.startswith('re
for info in info_names: # 对于每个特征名字
    blue = 'blue' + info
    red = 'red' + info
    blue_win = 'blue' + info + '_Win'
    blue_lse = 'blue' + info + '_Lose'
    red_win = 'red' + info + '_Win'
    red_lse = 'red' + info + '_Lose'

    df[red_win] = df[red]
    df[red_lse] = df[red]
    df[blue_win] = df[blue]
    df[blue_lse] = df[blue]

    df.loc[(df.blueWins == 1), blue_win] = None
    df.loc[(df.blueWins == 1), red_lse] = None
    df.loc[(df.blueWins == 0), blue_lse] = None
    df.loc[(df.blueWins == 0), red_win] = None

    fig, axs = plt.subplots(1, 2, figsize=(20,5))
    df[[blue_win, blue_lse]].plot.hist(ax=axs[0], bins=30,
    df[[red_win, red_lse]].plot.hist(ax=axs[1], bins=30, co
    axs[0].set_title(blue)
    axs[1].set_title(red)

    plt.show()
```



数据描述与处理

离散化

采用以下代码对数据进行离散化:

```
for c in df.columns[1:]:  # 遍历每一列特征，跳过标签列
    if c in discrete_features:
        continue
    if c in q_features:
        col = df[c]
        BIN_BOUND = [np.NINF]  # 区间起点 (防止分 bin 后出现 NaN)
        for qnt in QUANTILES:  # 读取 config 中选定的分位数
            BIN_BOUND.append(col.quantile(qnt))
        BIN_BOUND.append(np.inf)  # 区间终点
        BIN = pd.IntervalIndex.from_tuples([(BIN_BOUND[i], BIN_BOUND[i+1]) for i in range(0, 6)])
        discrete_df[c] = pd.cut(col, BIN)  # 相比 qcut 这样做的好处是分位点比较整，且不会出现分位点不唯一的情况.
```

注 1. 分位点的起点和终点分别设置为 $-\infty$ 和 ∞, 是为了防止划分区间后出现 NaN.

注 2. 分位完成后, 应当检查下分类结果是否正确, 特别地, 检查数据中是否有 NaN.

```
discrete_df.isnull().sum()
```

注 3. pd.cut vs pd.qcut: 产生 NaN, 可解释性, 唯一性.

注 4. 单独拎出来做判断的方法使得后面舍去部分特征时不需要改参数.

数据描述与处理

二值化

可以进一步对数据做二值化:

```
_types = discrete_df[c].value_counts().index.tolist()  # 取值的数目
for _type in _types:  # 创造一系列数据, 每个都是 {0, 1} 取值的
    discrete_df[c + str(_type)] = [1 if _sample == _type else 0 for _sample in discrete_df[c]]
discrete_df = discrete_df.drop(columns=c)
```

算法实现

输入分析

按代码框架形成的训练集，可以分析每个模型实例需要获知的信息：

- 数据及其结构，包括
  - 类别的名称 (其实在训练集中也可以获得)，我们约定用一个列表传入
  - 特征的名称 (训练集中已经 strip 掉了)
  - 训练集的特征和类别

  这些东西在生成训练集的代码框架中都已经生成好了，原样接收即可.
- 模型的参数，包括
  - 用于预剪枝的参数，包括最大深度、分裂样本阈值、信息增益阈值等.
  - 信息熵的计算方法.

  这些东西的类型也非常直观，写清楚文档即可.

算法实现

成员设计

除了输入信息之外，为了减少内部函数互相调用时麻烦的传参，故除上述输入外，一些不用回溯的数据也被设计成了类的成员．最终的设计如下：

```
def __init__(self, classes: list, features: list,
             max_depth=10, min_samples_split=10, min_info_gain=0.05,
             impurity_t='entropy'):

    self.classes = classes  # 模型的分类，如 [0, 1]
    self.feature_name = features  # 每个特征的名字
    self.data = None  # 缓存数据句柄
    self.label = None  # 缓存数据句柄

    self.max_depth = min(len(features), max_depth)  # 预剪枝：决策树时的最大深度
    self.min_samples_split = min_samples_split  # 预剪枝：到达该节点的样本数小于该值则不再分裂
    self.min_info_gain = min_info_gain
    self.impurity_t = impurity_t  # 计算混杂度（不纯度）的计算方式，例如 entropy 或 gini

    self.root = None  # 定义根节点，未训练时为空
    self.feature_unique = []  # 每个属性的取值空间，与 features 相对应
    self.info_split = []  # 每个属性的 SplitInformation，与 features 相对应
```

算法实现

模型训练

决策树算法基于树，其核心部分是递归进行. 每次调用中，需要以下步骤:

(1) 判断是否到达递归基. 到达递归基后，需要给当前节点作出分类.

- 无需分裂: 随意取一个样本的类型作为分类即可.
- 数目不够或达最大深度: 有样本则少数服从多数; 无样本则跟随父节点类型. $\longrightarrow$ 需要传参
- 没有多余特征: 可以让最大深度不大于特征数解决.

(2) 确定到达当前节点的样本，以及截至该节点还未被选择的特征. $\longrightarrow$ 需要回溯，作参数

(3) 求解该节点上选择每种特征的信息增益，具体来说，对每个特征，

- 计算各子节点上的混杂度. $\longrightarrow$ 因为到达每个节点的样本不同，需要单独统计
- 计算惩罚项 Split in Information. $\longrightarrow$ 对每个节点都一样，可以提前计算
- 计算信息增益. $\longrightarrow$ 利用上两步结果，以及本节点的混杂度 (上一步已计算) 获得，选择传参

(4) 选出最佳特征，更新状态，递归调用.

算法实现

模型训练

根据上述分析，将递归调用部分函数接口设计如下：

```
def expand_node(self, mask, sample, impur, p_mix=1, depth=0, branch=('root', '')):
```

其中，前四个参数分别对应分析中需要传参的可选特征、可选样本、本节点混杂度和父节点的多数特征，二后两个参数分别用于控制深度和输出信息，不需参与计算.

算法实现

模型训练

```
def fit(self, feature: np.ndarray, label: np.ndarray):

    # 1. 用 self.data 和 self.label 接收数据

    # 2. 计算 SplitInformation
    # 先计算特征的取值数目: 用 self.feature_unique 存储
    # 再统计每个特征的正负例个数
    # 最后调用 entropy 计算 SI

    # 3. 计算初始节点混杂度, 存入 impur

    # 4. 递归调用, 开始建树
    self.root = self.expand_node([True] * len(self.feature_name),  # 初始节点可选择所有的特征
                                 set(range(0, self.data.shape[0]))  # 同时保有所有的样本
                                 impur)

def expand_node(self, mask, sample, impur, p_mix=1, depth=0, branch=('root', '')):

    # 1. 递归基
    # 处理情况 1: 纯洁, 无需分裂
    # 处理情况 2: 层数达到分裂阈值 或 到达该节点的数据太少

    # 2. 找到最佳分裂特征, 递归调用 expand_node
    # (1) 计算各特征的信息增益, 保存为 fea_sieve
    fea_sieve = {}  # 用于存储 可选特征(index): [gain, [每 _type 的起始 impurity(减少可能的计算)], [sample_i]]
    # (2) 筛选最佳特征
    # 隐含递归基: 初值即为 IG 都一样时的选择

    # 3. 递归调用
    # 更新 mask
    # 为可能的空孩子节点统计本节点的类型
    for _type in range(0, len(self.feature_unique[k_min])):  # 为特征的每一取值依次建立子树
        self.expand_node(new_mask,
                         fea_sieve[k_min][2][_type].copy(),
                         branch=(self.feature_name[k_min], self.feature_unique[k_min][_type]),
                         depth=depth+1,
                         p_mix=self.classes[np.argmax(mix_parent)],
                         impur=fea_sieve[k_min][1][_type])  # 读取所保存的计算结果
```

算法实现

模型训练

```
fea_sieve = {}  # 用于存储 可选特征(index): [gain, [每 _type 的起始 impurity(减少可能的计算)], [sample_i]]
for _fea in range(0, len(self.feature_name)):  # 每个特征
    if mask[_fea]:  # 若可供本节点筛选
        # a. 初始化容器
        mix_i = []  # 保存混杂度计算数据
        for i in range(0, len(self.feature_unique[_fea])):  # 循环赋值避免浅拷贝问题
            mix_i.append([0] * len(self.classes))  # mix_i[_type][_class] 为该特征分类下的某类实例数目
        sample_i = []  # 保存每个节点留下的样本编号
        for i in range(0, len(self.feature_unique[_fea])):  # 循环赋值避免浅拷贝问题
            sample_i.append(set())  # sample_i[_type] 为分流到该特征分类下子节点的所有样本
        # b. 查找, 分流
        for index in range(0, self.data.shape[0]):
            if index in sample:  # 该样本之前没被筛掉
                _class = self.classes.index(self.label[index])  # 该样本类别在 mix_i 中的 offset
                _type = self.feature_unique[_fea].index(self.data[index][_fea])  # 该样本特征类型在 mix_i 中的 offset
                mix_i[_type][_class] += 1  # 统计相应类别数目
                sample_i[_type].add(index)  # 将该样本加入节点留下的样本中
        # c. 计算混杂度及 IG
        ig = impur
        s_all = len(sample)  # 该节点下样本总数
        child_imp = []
        for _set, _type in zip(sample_i, mix_i):  # 取出每个类型的混杂度数据 _type[_class]
            imp = self.impurity(_type)  # _type 的混杂度
            child_imp.append(imp)
            ig -= imp * len(_set) / s_all
        ig /= self.info_split[_fea]  # GainRatio
        # d. 将信息加入字典
        fea_sieve[_fea] = [ig, child_imp, sample_i]
```

注 1. 初始化注意浅拷贝问题.

注 2. 对于类别、特征和特征的取值, 都用类成员映射成下标的 offset, 便于应对各种输入.

算法实现

回溯输出

为了保存树的信息和方便回溯输出结果，定义以下辅助类：

```
class DTNode:

    def __init__(self, mode, branch, label=0, var=0, varname='', level=0):
        if mode == 'leaf':
            self.end = True  # Leaf Node
            self.label, self.branch, self._level = label, branch, level
        else:
            self.end = False
            self.variable, self.varname, self.branch, self._level = var, varname, branch, level
            self.children = []  # 孩子节点的列表 [DTNode]

    def __str__(self):
        indent = '  ->' * self._level
        if self.end:
            return " ".join([indent, 'WHEN', self.branch[0], "EQUALS", str(self.branch[1]),
                             'REACHES LEAF, CLASS=', str(self.label), '\n'])
        this_layer = " ".join([indent, 'WHEN', self.branch[0], "EQUALS", str(self.branch[1]),
                               "SELECT", self.varname, '\n'])
        for child in self.children:
            this_layer += str(child)
        return this_layer
```

这里既要记录辅助输出的信息，又要记录帮助模型判断的信息.

算法实现

模型预测

模型预测函数在框架中已经给出，以下是递归调用的实现：

```
def traverse_node(self, current: DTNode, feature):
    # 递归基
    if current.end:  # 到达叶子节点
        return current.label
    # 深入子树
    label = feature[current.variable]  # 取出本节点用于分类的属性
    for index in range(0, len(self.feature_unique[current.variable])):  # 以次找子区间
        child = self.feature_unique[current.variable][index]
        if label == child:
            return self.traverse_node(current.children[index], feature)
    raise ValueError("Not Found!")
```

事实上如果没有做二值化，额外加个类型判断还可以对没有离散化的样例进行预测.

模型调优

参数调整

- 最初的准确率是 0.55. 消除若干处逻辑错误后, 准确率提升到了 0.63.
- 通过调整最大深度和最小分裂阈值, 准确率提升到了 0.68 (深度 8, 阈值 27). 准确率随二者的变化都是单峰——有一个极大值, 增大或缩小参数都会显著地降低准确率.
- 调整分位点 QUANTILES:
  - 从 QUANTILES 中删除部分分位点数目, 模型准确率大幅下降 (0.59-0.62).
  - 调整分位点具体位置而不改变分位点数目, 模型准确率略微下降 (0.67-0.68).
- 特征处理方法:
  - 只使用特征 blue$*$+ red $*$, 准确率略微提升 (0.68).
  - 只使用特征 br$*$, 准确率略微下降 (0.67). PS: 一起使用时, 树中也没有选择任何 br$*$.
  - 对特征进行二值化后, 准确率有小幅提升 (0.70), 但是执行时间显著增加了, 且所需的树的深度增加 (12). 考虑到这样做之后对原始数据进行分类会变得麻烦, 因此是否值得还是一个问题.
- gini 混杂度略优于 entropy (0.67).

模型调优

训练结果

- 可选参数: 深度 8, 分裂阈值 27, gini 混杂度, 全体特征, 未二值化, 结果准确率为 0.6867.
- 最终参数: 深度 12, 分裂阈值 29, gini 混杂度, 全体特征, 二值化, 结果准确率为 0.7014.

```
accuracy: 0.6867
 WHEN root EQUALS  SELECT blueTotalGold
  -> WHEN blueTotalGold EQUALS (-inf, 14194.0] SELECT redTotalGold
  ->  -> WHEN redTotalGold EQUALS (-inf, 14238.8] REACHES LEAF, CLASS= 1
  ->  -> WHEN redTotalGold EQUALS (14238.8, 15427.5] REACHES LEAF, CLASS= 0
  ->  -> WHEN redTotalGold EQUALS (15427.5, 16378.0] SELECT redEliteMonsters
  ->  ->  -> WHEN redEliteMonsters EQUALS 0 REACHES LEAF, CLASS= 0
  ->  ->  -> WHEN redEliteMonsters EQUALS 1 SELECT redWardsPlaced
  ->  ->  ->  -> WHEN redWardsPlaced EQUALS (-inf, 12.0] REACHES LEAF, CLASS= 0
  ->  ->  ->  -> WHEN redWardsPlaced EQUALS (12.0, 14.0] REACHES LEAF, CLASS= 0
  ->  ->  ->  -> WHEN redWardsPlaced EQUALS (14.0, 16.0] REACHES LEAF, CLASS= 0
  ->  ->  ->  -> WHEN redWardsPlaced EQUALS (16.0, 20.0] REACHES LEAF, CLASS= 0
  ->  ->  ->  -> WHEN redWardsPlaced EQUALS (20.0, 53.0] REACHES LEAF, CLASS= 0
```

```
accuracy: 0.7014
 WHEN root EQUALS  SELECT redTotalGold(-inf, 14238.8]
  -> WHEN redTotalGold(-inf, 14238.8] EQUALS 0 SELECT redTotalGold(14238.8, 15427.5]
  ->  -> WHEN redTotalGold(14238.8, 15427.5] EQUALS 0 SELECT blueKills(12.0, inf]
  ->  ->  -> WHEN blueKills(12.0, inf] EQUALS 0 SELECT blueTotalGold(-inf, 14194.0]
  ->  ->  ->  -> WHEN blueTotalGold(-inf, 14194.0] EQUALS 0 SELECT blueTotalGold(17459.0, 19190.5]
  ->  ->  ->  ->  -> WHEN blueTotalGold(17459.0, 19190.5] EQUALS 0 SELECT blueTotalGold(19190.5, inf]
  ->  ->  ->  ->  ->  -> WHEN blueTotalGold(19190.5, inf] EQUALS 0 SELECT redTotalGold(19137.0, inf]
  ->  ->  ->  ->  ->  ->  -> WHEN redTotalGold(19137.0, inf] EQUALS 0 SELECT redTotalGold(15427.5, 16378.0]
  ->  ->  ->  ->  ->  ->  ->  -> WHEN redTotalGold(15427.5, 16378.0] EQUALS 0 SELECT redTowersDestroyed2
  ->  ->  ->  ->  ->  ->  ->  ->  -> WHEN redTowersDestroyed2 EQUALS 0 SELECT blueTotalGold(14194.0, 15415.5]
  ->  ->  ->  ->  ->  ->  ->  ->  ->  -> WHEN blueTotalGold(14194.0, 15415.5] EQUALS 0 SELECT redTotalExperience(19879.0, inf]
  ->  ->  ->  ->  ->  ->  ->  ->  ->  ->  -> WHEN redTotalExperience(19879.0, inf] EQUALS 0 SELECT blueAvgLevel(7.4, inf]
```

- 未来优化方向:
  - 实现随机划分 k-fold 验证集.
  - 使用粒子群算法搜索最优的 QUANTILES 设置 (需要验证集).
  - 尝试更多特征组合和设置 (需要验证集).
  - 对决策树实现图形输出.