基于决策树的英雄联盟游戏胜负预测

实验报告

班级：学堂在线 1期机器学习-训练营

姓名：王雨静

时间：2020年11月7日

写在前面

本实验报告仅供参考，具体代码实现请看文件夹中的hw1.ipynb。

目录

一、实验目的

本次以英雄联盟对局胜负预测任务为基础，要求实现决策树算法相关细节，加深对算法的理解，并了解做机器学习任务的大致流程。
原始数据共40列，38个特征，1个标签，和一个对局标号。
要求通过特征预测标签。

二、特征提取

2.1 特征概览和分析

将各个特征按标签0和1分类，画出累计直方图(cumulative histogram)。

代码实现：

In [6]:

```
#plot hists to discover the relecance of feature to label

import matplotlib.pyplot as plt

figure, axes = plt.subplots(14, 3, figsize = (18, 84), dpi = 100)
ax = axes.flatten()
for i, c in enumerate(df.columns[1:]):
    ax[i].hist(df.loc[df['blueWins'] == 1][c], bins = 100, alpha = 0.5, histtype='step', cumulat
ive = True,density=True)
    ax[i].hist(df.loc[df['blueWins'] == 0][c], bins = 100, alpha = 0.5, histtype='step', cumulat
ive = True,density=True)
    ax[i].set_title(c)

#plt.savefig("beforeDiscrete.png") #保存作的图
plt.show()
```

详细图见文件夹中beforeDiscrete.png

注释：蓝线表示蓝方获胜，橙线表示红方获胜。

分析：红线和橙线之间的距离越大，表示该特征与标签的关联度越高。

2.2 增删特征

redKills, redDeaths分别和blueDeaths, blueKills重复，删除。

有2.1分析得到一些特征和标签的关联性不强，可以舍去。

舍去的特征：'blueWardsPlaced'， 'blueWardsDestroyed', 'blueTotalJungleMinionsKilled'， 'redWardsPlaced', 'redWardsDestroyed', 'redTowersDestroyed'， 'redTotalJungleMinionsKilled'，'brWardsPlaced'。

2.3 特征离散化

离散方法一：去掉两端1%的极值后，等区间划分。

离散方法二：去掉两端1%的极值后，等比划分。

代码实现：

```
for c in df.columns[1:]: # 遍历每一列特征，跳过标签列

    if c == 'brFirstBlood':continue
    if c == 'blueEliteMonsters' or c == 'blueDragons' or c == 'blueHeralds' or c == 'blueTowersDestroyed'
    if c == 'redEliteMonsters' or c == 'redDragons' or c == 'redHeralds' or c == 'redTowersDestroyed': co
    if c == 'brEliteMonsters' or c == 'brDragons' or c == 'brHeralds' or c == 'brTowersDestroyed': conti

    #离散方法1：去极值后按x值等区间划分
    win_1 = df.loc[df['blueWins'] == 1][c].tolist()
    lose_1 = df.loc[df['blueWins'] == 0][c].tolist()
    win_1.sort()
    lose_1.sort()
    lowerbound = win_1[int(0.01 * len(win_1))]
    upperbound = lose_1[int(0.99 * len(lose_1))]
    step = int((upperbound - lowerbound) * 1000 /num) #可调参数：分成几个区间
    bins = [ -np.inf] + [ i/1000 for i in list(range(int(lowerbound*1000), int(upperbound*1000), step))]
    discrete_df[c] = pd.cut(discrete_df[c],bins, labels = False)

    #离散方法2：等比划分
    list1 = df[c].tolist()
    list1.sort()
    lower = list1[int(0.01 * len(list1))]
    upper = list1[int(0.99 * len(list1))]
    bins = [-np.inf, lower, upper, np.inf]
    for i in range(1,num):
        new = list1[int((i/num)*len(list1))]
        bins.append(new)
    bins = set(bins)
    bins = list(bins)
    bins.sort()
    discrete_df2[c] = pd.cut(discrete_df2[c],bins, labels = False)
```

2.4 离散效果

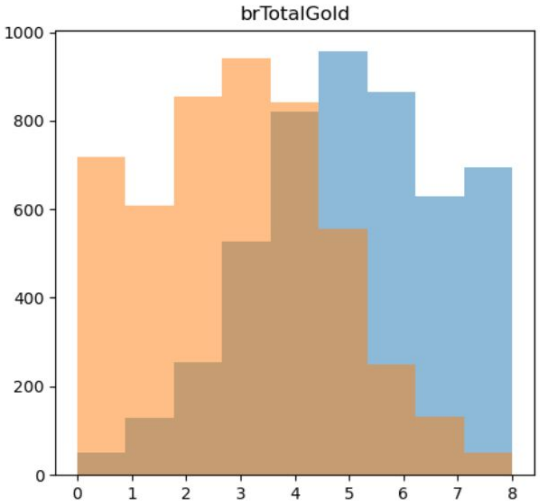
将离散后的特征按标签0和1分类，画出直方图。

详细图见文件夹中:

9区间：afterDiscrete1_2.png（最终使用的划分方法），afterDiscrete2_2.png

12区间：afterDiscrete1.png, afterDiscrete2.png

分析：可以发现如‘brTotalGold’等特征的离散效果不错。



三、建决策树

3.1 数据集准备

1. 分开标签和特征。
2. 划分训练集、验证集和测试集。比例：7112 : 791 : 1976，约为7 : 1 : 2。

3.2 准备计算用函数

新增的函数包括：众数、1的概率、根据概率返回1或0、为信息熵重新定义的log2、信息熵、信息增益、信息增益率、基尼值、基尼指数。

3.3 决策树模型

3.3.1 init：

```
class DecisionTree(object):

    def __init__(self,  features, criterion = "bestgainratio", splitter = 'best',
                 max_depth = 20, min_samples_split = 10):
        self.criterion = criterion
        self.splitter = splitter
        self.max_depth = max_depth
        self.min_samples_split = min_samples_split
        self.features = features
```

3.3.2 根据splitter和criterion返回合适的特征：

最大信息增益：

```
    def bestgain(self, X, y):
        best_Index = -1
        best_gain = -np.inf
        ent = Ent(y)
        for i in range(X.shape[1]):
            gain = Gain(X[:,i], y, ent)
            if gain > best_gain:
                best_gain = gain
                best_Index = i #信息增益最大的特征
        return best_Index
```

最大信息增益比：

```
    def bestgainratio(self, X, y):
        n = X.shape[1]
        gain_arr = np.zeros(n)
        ent = Ent(y)
        for i in range(n):
            gain_arr[i] = Gain(X[:,i], y, ent)
        m_gain = np.mean(gain_arr) #平均增益
        best_Index = -1
        best_gain_ratio = -np.inf
        for i in range(n):
            if gain_arr[i] > m_gain:
                gain_ratio = Gain_Ratio(X[:, i], y, ent)
                if gain_ratio > best_gain_ratio:
                    best_gain_ratio = gain_ratio
                    best_Index = i #信息增益比最大的特征
        return best_Index
```

最小的基尼指数：

```
    def bestgini(self, X, y):
        best_Index = -1
        best_gini_index = np.inf
        for i in range(X.shape[1]):
            gini_index = Gini_index(X[:,i], y)
            if gini_index < best_gini_index:
                best_gini_index = gini_index
                best_Index = i #基尼指数最小的特征
        return best_Index

    def rand_(self, X, y):
        return np.random.choice(X.shape[1])
```

随机返回一个标签：

```
    def rand_(self, X, y):
        return np.random.choice(X.shape[1])
```

3.3.3 建树函数：

返回值为node，是标签值（0或1），或者一个字典，供查询。

字典结构：‘#’键的值为这个结点的名字（标签序号）；其他键为这个标签的取值，值是标签值或者下一个结点的字典。

*预剪枝

样本数量小于等于min_samples_split，或者特征数量小于等于总特征数-树的最大深度时，返回该节点的众数；当1或0的概率大于0.85时，返回该节点的众数。

思考：为什么用众数，不用1的概率？

在此次实验中，输赢的概率各占一半
假设这个节点1的概率为0.8：(用概率的准确率，众数的准确率)
实际概率为0.8：0.8*0.8 + 0.2*0.2 = 0.68 < 0.8；
实际概率为0.5：0.8*0.5 + 0.2*0.5 = 0.5 = 0.5；
实际概率为0.2：0.8*0.2 + 0.2*0.8 = 0.32 > 0.2。

由此可见当实际概率大于0.5时，即接近模型预测的概率时，使用众数效果较好。
基于对自己建的模型的自信（不是），决定用众数。

```
    #build the tree
    def build_(self, X, y, feat_lst, criterion):
        m, n = X.shape #样本，特征数量

        if len(set(y)) == 1: return y[0] #当y中只有一种label时，返回改标签

        #考虑pre pruning返回情况
        if prob1(y) > 0.85 : return 1 #当1的概率大于90%时，该节点为1
        if prob1(y) < 0.15: return 0 #当1的概率小于10%时，该节点为0

        #当样本数量小于等于min_samples_split， 或者特征数量小于等于总特征数-树的最大深度时，返回该节点的众数
        if m <= self.min_samples_split or n <= len(self.features) - self.max_depth : return cal_mode(y)


        if n == 1: #当特征数量为1时，该节点的值为y中的众数
            node = {'#': feat_lst[0]} #结点，存储特征的索引
            x = X[:, 0]
            for val in set(x):
                node[val] = cal_mode(y[x==val])                                    #prob修改的地方
        else:
            best_Index = criterion(X, y)
            splitVal = set(X[:,best_Index]) #该特征的所有取值
            if len(splitVal)==1:
                return cal_mode(y) #特征值都一样，返回频数最大的类别                          #prob修改的地方
            else:
                node = {'#':feat_lst[best_Index]} #结点，存储特征的索引
                index = list(range(n))
                index.pop(best_Index) #需要划分的特征index
                feat_1 = feat_lst[:] #避免影响，前面的
                feat_1.pop(best_Index)
                for val in splitVal:
                    i_sample = X[:, best_Index] == val #子数据集
                    node[val] = self.build_(X[i_sample][:,index], y[i_sample], feat_1, criterion)
        return node
```

3.3.4 fit

选取合适的criterion函数，调用build_建树。

```
    def fit(self, X, y):
        assert len(self.features) == len(X[0])# 输入数据的特征数目应该和模型定义时的特征数目相同
        #建树
        if self.splitter == 'best':
            if self.criterion == 'bestgain':
                self.tree = self.build_(X, y, list(range(X.shape[1])), self.bestgain)
            elif self.criterion == 'bestgainratio':
                self.tree = self.build_(X, y, list(range(X.shape[1])), self.bestgainratio)
            elif self.criterion == 'bestgini':
                self.tree = self.build_(X, y, list(range(X.shape[1])), self.bestgini)
            else:
                raise('gini/gain/gainratio')
        else:
            self.tree = self.self.build_(X, y, list(range(X.shape[1])), self.rand_) #随便建一颗树
        return self
```

3.3.5 predict 和 predict_

```
def predict(self, X):
    #assert len(X.shape) == 1 or len(X.shape) == 2 # 只能是1维或2维
    if len(X.shape) > 1: #二维数组
        rst = np.zeros(X.shape[0])
        for i, x in enumerate(X):
            rst[i] = self.predict_(x)
            #rst[i] = gety(rst[i])                                    #prob修改的地方
    elif len(X) == 0:
        rst = -1
    else:
        rst = self.predict_(X)
        #rst = gety(rst)                                              #prob修改的地方
    return rst
```

```
def predict_(self, x):
    tree = self.tree
    while True:
        if isinstance(tree, dict):
            key = tree['#'] #树的名字
        else:
            return tree
        try:
            tree = tree[x[key]] #根据取值进入下一级
        except:
            return -1
    '''
```

3.4 训练模型

在训练集上训练一个模型，并查看在验证集上的准确率

```
#为后剪枝训练一个模型
DT = DecisionTree(criterion = "bestgini", splitter = 'best', features=feature_names, max_depth = 2, min_sample
DT.fit(x_training, y_training) #在训练集上训练
p_val = DT.predict(x_val) #在测试集上预测，获得预测值
#print(p_val)
val_acc = accuracy_score(p_val, y_val)# 将测试预测值与测试集标签对比获得准确率
print('accuracy:{:.4f}'.format(val_acc))# 输出准确率
```

```
accuracy:0.7332
```

树的样子：

```
Out[17]: {'#': 28,
          0: 0,
          1: {'#': 7, 0: 0, 1: 0, 2: 0, 3: 0, 4: 0, 5: 0, 6: 0},
          2: {'#': 25, 0: 0, 1: 0, -1: 0},
          3: {'#': 24, 0: 0, 1: 0, 2: 0, -2: 0, -1: 0},
          4: {'#': 24, 0: 1, 1: 1, 2: 1, -1: 0, -2: 0},
          5: {'#': 30, 2: 0, 3: 1, 4: 1, 5: 1, 6: 1, 7: 1, 9: 1},
          6: {'#': 9, 0: 1, 1: 1, 2: 1, 3: 1, 4: 1, 5: 1, 6: 1, 7: 1, 9: 1},
          7: {'#': 24, 0: 1, 1: 1, 2: 1, -2: 1, -1: 1},
          8: 1}
```

3.5 后剪枝

3.5.1 错误降低剪枝

遍历所有的字典节点，将其替换为0或1后，如果其在验证集上的准确率是否上升或不变，就更新模型。

计算在验证集上的准确率：

```
#后剪枝函数
#先准备一棵用来修改和预测的树
DTbusy = copy.deepcopy(DT)
#先定义一些函数
#返回在val上的准确率
def acc_val(dtree):
    DTbusy.tree = dtree #借用DTbusy进行测试
    p_val = DTbusy.predict(x_val) #在val集上预测，获得预测值
    val_acc = accuracy_score(p_val, y_val)
    return val_acc
```

返回下一个修改的节点：

```
#return type: list of keys to locate the node
def nextnode(keys, tree): #keys: list, tree: dict
    newkeys = copy.deepcopy(keys)
    count = 0
    for value in tree.values():
        count += isinstance(value, int or float)
    if count == len(tree):
        return newkeys
    else:
        for newkey, value in tree.items():
            if isinstance(value, int or float): pass
            if isinstance(value, dict):
                newkeys.append(newkey)
                return nextnode(newkeys, value)
```

替换节点为1或0：

```
def replacenode(keys, tree, k): #replace node with k
    depth = len(keys)
    for i in range(depth - 1):
        tree = tree[keys[i]]
    tree[keys[depth - 1]] = k
```

剪枝函数：

返回的是修剪好的树。

修剪过程出打印出已经遍历的节点，和修改的节点。

最后打印出剪完枝的老树，表示剪枝完成。

```
def postpruning(tree): #返回修剪好的树，类型为dict
    oldTree = copy.deepcopy(tree) #用来砍到没有子节点的树
    savedTree = copy.deepcopy(tree) #最优树
    saved_acc = acc_val(savedTree) #最优树的正确率

    keys = []
    keys = nextnode(keys, oldTree)
    counts = 0
    print(keys)

    while(len(keys) != 0):
        newTree = copy.deepcopy(savedTree) #复制一份最优树
        replacenode(keys, oldTree, 0)
        replacenode(keys, newTree, 0) #将节点换成0
        acc0 = acc_val(newTree)
        if  acc0 >= saved_acc:
            saved_acc = acc0
            savedTree = copy.deepcopy(newTree)
            counts += 1
            print("replace key:", keys, "   value: 0")
            print("accuracy:", saved_acc)
        replacenode(keys, newTree, 1) #将节点换成1
        acc1 = acc_val(newTree)
        if acc1 >= saved_acc:
            saved_acc = acc1
            savedTree = copy.deepcopy(newTree)
            counts += 1
            print("repalce keys:", keys, "   value: 1")
            print("accuracy:", saved_acc)

        keys = []
        keys = nextnode(keys, oldTree)
        print(keys)

    print("oldTree: ", oldTree)
    print("counts:", counts)
    return savedTree
```

复制一份生成的树：

```
In [19]: DT2 = copy.deepcopy(DT)
         p_val = DT2.predict(x_val) #在val集上预测，获得预测值
         #print(p_val)
         val_acc = accuracy_score(p_val, y_val)# 将val预测值与val集标签对比获得准确率
         print('accuracy:{:.4f}'.format(val_acc))# 输出准确率
         print(DT2.tree)

         accuracy:0.7332
         {'#': 28, 0: 0, 1: {'#': 7, 0: 0, 1: 0, 2: 0, 3: 0, 4: 0, 5: 0, 6: 0}, 2: {'#': 25,
         0: 0, 1: 0, -1: 0}, 3: {'#': 24, 0: 0, 1: 0, 2: 0, -2: 0, -1: 0}, 4: {'#': 24, 0: 1,
         1: 1, 2: 1, -1: 0, -2: 0}, 5: {'#': 30, 2: 0, 3: 1, 4: 1, 5: 1, 6: 1, 7: 1, 9: 1}, 6:
         {'#': 9, 0: 1, 1: 1, 2: 1, 3: 1, 4: 1, 5: 1, 6: 1, 7: 1, 9: 1}, 7: {'#': 24, 0: 1, 1:
         1, 2: 1, -2: 1, -1: 1}, 8: 1}
```

进行后剪枝：

```
In [20]: #！！将上面打印出来的树手动复制下来 #我也不知道为什么不复制不能跑
DT2.tree = {'#': 28, 0: 0, 1: {'#': 7, 0: 0, 1: 0, 2: 0, 3: 0, 4: 0, 5: 0, 6: 0}, 2: {'
#后剪枝
DT2.tree = postpruning(DT2.tree)
p_val2 = DT2.predict(x_val) #在val集上预测，获得预测值
val_acc = accuracy_score(p_val2, y_val)# 将val预测值与val集标签对比获得准确率
print('accuracy:{:.4f}'.format(val_acc))# 输出准确率
```

```
[1]
replace key: [1]    value: 0
accuracy: 0.7332490518331226
[2]
replace key: [2]    value: 0
accuracy: 0.7332490518331226
[3]
replace key: [3]    value: 0
accuracy: 0.7332490518331226
[4]
[5]
repalce keys: [5]    value: 1
accuracy: 0.7357774968394437
[6]
repalce keys: [6]    value: 1
accuracy: 0.7357774968394437
[7]
repalce keys: [7]    value: 1
accuracy: 0.7357774968394437
[]
oldTree:  {'#': 28, 0: 0, 1: 0, 2: 0, 3: 0, 4: 0, 5: 0, 6: 0, 7: 0, 8: 1}
counts: 6
accuracy:0.7358
```

在测试集上的准确率：

```
p_test = DT2.predict(x_test) #在测试集上预测，获得预测值
print(p_test)
test_acc = accuracy_score(p_test, y_test)# 将测试预测值与测试集标签对比获得准确率
print('accuracy:{:.4f}'.format(test_acc))# 输出准确率
```

```
[0. 1. 0. ... 0. 1. 1.]
accuracy:0.7267
```

结论：在验证集上的准确率提升了，后剪枝可能有效。但也存在过拟合的可能性

3.5.2 规则后剪枝

想要尝试一下规则后剪枝，但是，对于怎么写代码毫无头绪，于是决定，手动剪枝。

准备：离散的时候，划分区间数减少为一共4个区间，最大深度为2。

生成的树：{'#': 28,

0: 0,

1: {'#': 24, 0: 0, 1: 0, 2: 1, -2: 0, -1: 0},

2: {'#': 30, 1: 1, 2: 1, 3: 1},

3: 1}

化简后：{'#': 28,

0: 0,

1: {'#': 24, 0: 0, 1: 0, 2: 1, -2: 0, -1: 0},

2: 1

3: 1}

存在的规则：f[28] f[24]

若先判断 f[28]， 最优树已经生成了，accuracy 在val上0.7320 （test: 0.7120)。

如果先判断 f[24],

DT2.tree = {'#': 24,

2: 1,

0:{'#': 28, 0: 0, 1: 0, 2: 0, 3: 1},

1:{'#': 28, 0: 0, 1: 0, 2: 0, 3: 1},

-1:{'#': 28, 0: 0, 1: 0, 2: 0, 3: 1},

-2:{'#': 28, 0: 0, 1: 0, 2: 0, 3: 1}}

然后借用postpruning剪枝，得到：{'#': 24,

2: 1,

0: {'#': 28, 0: 0, 1: 0, 2: 0, 3: 1},

1: 1,

-1: {'#': 28, 0: 0, 1: 0, 2: 0, 3: 1},

-2: 0}

Accuracy在val上为0.6283 （test: 0.6215）

所以选择原本的树。

最终accuracy 0.7120。

1976个测试样本中，有570个被误分类

标准差 S = sqrt(1 * 570/1975) = 0.5372

标准误差 SEM = 0.537222/sqrt(1976) = 0.01208

σ (error) = sqrt((15/52*(1-15/52)/1976) = 0.01019

实际accuracy为0.7115 +-1.96 * σ = 0.7115 +-0.0098的置信度为95%。

四、模型调优

4.1 寻找最佳深度

In [22]:
```
#寻找最佳深度 这个模块可以跳过
depth = list(range(1, len(feature_names)))#用这个比较可靠，但根据经验，可以略减小树的深
depth = list(range(1, 6)) #7112 / 7^6 = 0.06045个样本，已经非常可能过拟合了
acc1 = []
acc2 = []

for dep in depth:
    DT = DecisionTree(criterion = "bestgini", splitter = 'best',features=feature_names,
    DT.fit(x_training, y_training) #在训练集上训练
    p_val = DT.predict(x_val) #在val集上预测，获得预测值
    val_acc1 = accuracy_score(p_val, y_val)# 将预测值与验证集标签对比获得准确率
    print('depth:', dep, ' accuracy:{:.4f}'.format(val_acc1))# 输出准确率
    acc1.append(val_acc1)
    countinvalid = 0
    p_valnew = []
    for i in p_val:
        if i == 1:
            p_valnew.append(1)
        elif i == 0:
            p_valnew.append(0)
        else:
            countinvalid += 1
            p_valnew.append(0)
    val_acc2 = accuracy_score(p_valnew, y_val)# 将预测值与验证集标签对比获得准确率
    acc2.append(val_acc2)
    print('invalid', countinvalid, 'illusion accuracy:{:.4f}'.format(val_acc2))# 输出准
```

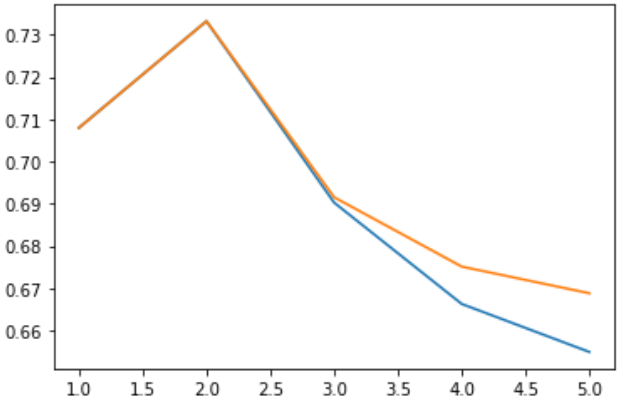
下图的注释：y轴 - 准确率，x轴-最大深度，蓝线-准确率，橙线-将invalid数据换成0后的准确率。

分析：当深度大于2时，随着深度加深，无法判断的值（invalid）增多，accuracy降低。

但由于在验证集（791个样本）95%置信度的区间约为Accuracy+-0.03，accuracy降低得可能并不显著。

结论：

最优深度为2层（左右）。



4.2 寻找最佳离散组数

In [24]:
```
#寻找最佳离散组数 这个模块可以跳过
i_s = []
val1 = []
val2 = []
for i in range(1, 25):
    midclass = i
    i_s.append(i)

    print('midclass:', midclass)
    #按要求离散特征
    discrete_df = discretedf(df, num = midclass, choice = 1)
    discrete_df2 = discretedf(df, num = midclass, choice = 2)

    #划分训练集、验证集和测试集
    all_y = discrete_df['blueWins'].values # 所有标签数据
    feature_names = discrete_df.columns[1:] # 所有特征的名称
    all_x = discrete_df[feature_names].values # 所有原始特征值，pandas的DataFrame.value
    x_train, x_test, y_train, y_test = train_test_split(all_x, all_y, test_size=0.2, ra
    x_training, x_val, y_training, y_val = train_test_split(x_train, y_train, test_size
    all_y.shape, all_x.shape, x_train.shape, x_training.shape, x_val.shape, x_test.shap

    DT = DecisionTree(criterion = "bestgini", splitter = 'best', features=feature_name
    DT.fit(x_training, y_training) #在训练集上训练
    p_val = DT.predict(x_val) #在测试集上预测，获得预测值
    val_acc = accuracy_score(p_val, y_val)# 将测试预测值与测试集标签对比获得准确率
    val1.append(val_acc)
    print('method: 1 accuracy:{:.4f}'.format(val_acc))# 输出准确率
```

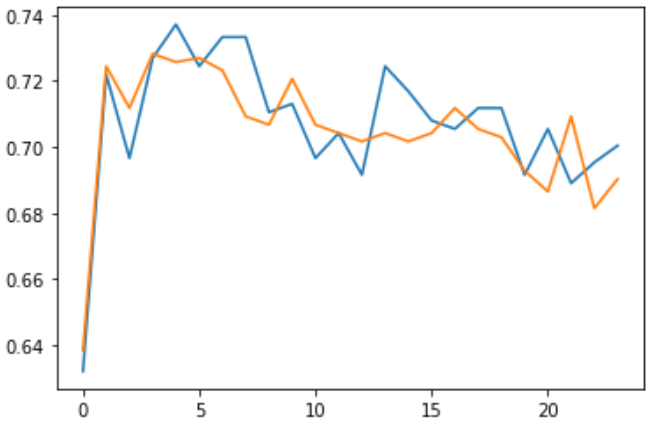
```
#等比划分
discrete_df = discrete_df2
#划分训练集、验证集和测试集
all_y = discrete_df['blueWins'].values # 所有标签数据
feature_names = discrete_df.columns[1:] # 所有特征的名称
all_x = discrete_df[feature_names].values # 所有原始特征值，pandas的DataFrame.value
x_train, x_test, y_train, y_test = train_test_split(all_x, all_y, test_size=0.2, ra
x_training, x_val, y_training, y_val = train_test_split(x_train, y_train, test_size
all_y.shape, all_x.shape, x_train.shape, x_training.shape, x_val.shape, x_test.shap

DT = DecisionTree(criterion = "bestgini", splitter = 'best', features=feature_name
DT.fit(x_training, y_training) #在训练集上训练
p_val = DT.predict(x_val) #在测试集上预测，获得预测值
val_acc = accuracy_score(p_val, y_val)# 将测试预测值与测试集标签对比获得准确率
val2.append(val_acc)
print('method 2 accuracy:{:.4f}'.format(val_acc))# 输出准确率
```

x轴表示除了两个极值外的分界线数量k，实际组数为其+3）k = midclass -1

y轴表示accuracy。

蓝线等区间划分，橙线等比划分。



分析：

在验证集（791个样本）95%置信度的区间约为Accuracy+-0.03。

k = 0时，accuracy较低。

k > 1时，对着区间划分数量增多，accuracy呈下降趋势，但是并不显著。

4.3 其他调优可能

1. 使用信息增益、信息增益比建树。由于树的层数较少，且各特征的离散组数相似，使用不同的条件建出来的树的准确率是非常近似甚至完全相同的。
2. 预剪枝优化：调整Min_sample split等预剪枝参数。考虑到两层树，9个区间，每个子叶的平均样本数量约为训练集的1.23%，88个。过拟合可能性较小。故没有修改这个值。
3. 交叉考量最大深度、后剪枝、离散组数等，找出最优树。更深的树，在训练时更可能过拟合训练集，（导致在验证集上准确率较低）在后剪枝过程中过拟合验证集（使得实际准确率远低于在验证集上的准确率）。由于时间精力以及设备能力有限，没有深入探索。

五、总结

5.1 实验结果：

置信度为95%的置信区间为0.7267±0.0196。

In [26]:
```
#最终测试
p_test = DT2.predict(x_test) #在测试集上预测，获得预测值
print(p_test)
test_acc = accuracy_score(p_test, y_test)# 将测试预测值与测试集标签对比获得准确率
print('accuracy:{:.4f}'.format(test_acc))# 输出准确率
```

```
[0. 1. 0. ... 0. 1. 1.]
accuracy:0.7267
```

In [27]:
```
#最终树的样子
DT2.tree
```

Out[27]:
```
{'#': 28,
 0: 0,
 1: 0,
 2: 0,
 3: 0,
 4: {'#': 24, 0: 1, 1: 1, 2: 1, -1: 0, -2: 0},
 5: 1,
 6: 1,
 7: 1,
 8: 1}
```

In [28]:
```
feature_names[28], feature_names[24]
```

Out[28]:
```
('brTotalGold', 'brEliteMonsters')
```

结论：置信度为95%的置信区间为0.7267±0.0196。

5.2 收获

在英雄联盟游戏胜负预测任务这个案例中，学习实践了机器学习的各个阶段的任务，包括：确定任务、数据分析、特征工程、数据集划分、模型设计、模型训练和效果测试、结果分析和调优等。收获颇丰。