实验一

基于决策树的英雄联盟游戏胜负预测

2022年6月17日

特征离散化

如果数据的取值小于5种类型则跳过，
否则将数据划分为5个区间，使用了pandas的cut方法，按照等间距划分。

In [5]:
```
discrete_df = df.copy() # 先复制一份数据
for c in df.columns[1:]: # 遍历每一列特征，跳过标签列
    '''
    请离散化每一列特征，即discrete_df[c] = ...

    提示:
    对于有些特征本身取值就很少，可以跳过即 if ... : continue
    对于其他特征，可以使用等区间离散化、等密度离散化或一些其他离散化方法
    可参考使用pandas.cut或qcut
    '''
    if len(discrete_df[c].unique()) < 5:
        continue
    discrete_df[c] = pd.cut(discrete_df[c], 5, labels=False)
discrete_df.head()
```

Out[5]:

	blueWins	blueWardsPlaced	blueWardsDestroyed	blueKills	blueDeaths	blueAssists	blueEliteMonsters	blueDragons	blueHeralds	blueTowersDestroyed	...	t
0	0	0	0	2	1	1	0	0	0	0	...	
1	0	0	0	1	1	0	0	0	0	0	...	
2	0	0	0	1	2	0	1	1	0	0	...	
3	0	0	0	0	1	0	1	0	1	0	...	
4	0	1	0	1	1	1	0	0	0	0	...	

5 rows × 44 columns

决策树模型的实现

1. 混杂度计算

这里提供两种计算混杂度的方法，分别是gini系数和熵。
第一步先计算每个种类取值的概率，用pandas的value_counts(normalize)实现。
第二部则根据不同的方法计算混杂度。

```
def impurity(self, label):
    val_data = pd.Series(label).value_counts(normalize=True)
    impur = 0
    if self.impurity_t == 'gini':
        for val in val_data:
            impur += 1 - val * val
    elif self.impurity_t == 'entropy':
        for val in val_data:
            impur -= val * np.log(val)
    return impur
```

2. 信息增益计算

第一步根据特征的取值，划分数据。

第二步计算各个数据的混杂度，并加权平均，作为用该特征划分后的混杂度。

第三步用原来的混杂度减去当前计算的，则为信息增益

```
def gain(self, data, label, old):
    cnt_map = {}
    n = len(label)
    for i in range(n):
        if data[i] in cnt_map.keys():
            cnt_map[data[i]].append(label[i])
        else:
            cnt_map[data[i]] = [label[i]]
    new = 0
    for k, v in cnt_map.items():
        new += len(v) / n * self.impurity(v)
    return old - new
```

3. 构建决策树

采用预剪枝策略：

如果样本数小于min_samples_split 或者 高度大于 max_depth则停止

构建树的步骤是：

1）计算当前的混杂度

2）遍历所有特征，找到最大信息增益，则可得最佳特征

3）将最佳特征从数据中移除，并将数据按照该特征的取值划分

4）递归该操作，直到到达终止条件

```
def expand_node(self, columns, data, label, depth):
    if len(set(label)) == 1: # label取值相同
        return label[0]
    elif len(label) <= self.min_samples_split or depth == self.max_depth or len(data) == 0 or (len(data) == 1 and l
        if len(data) == 1 and len(set(data[0])) == 1: # 特征相同而label不同，可能忽略了重要特征
            print("there are some import features ignore!")
        return np.argmax(np.bincount(label)) # 返回label中最多的
    n = len(data[0]); m = len(data[:,0])
    impur = self.impurity(label) # 计算信息混杂度
    gain_ls = [self.gain(data[:,i], label, impur) for i in range(n)] # 遍历特征，计算信息增益
    idx = np.argmax(gain_ls)
    best_feature = columns[idx] # 找到最好的特征
    tree = {columns[idx]: {}}
    featValues = set(data[:,idx])
    data_map = {}; label_map = {}
    for i in range(m):  # 根据特征的取值，将数据划分
        k = data[i,idx]
        if k in data_map.keys():
            data_map[k].append(data[i])
            label_map[k].append(label[i])
        else:
            data_map[k] = [data[i]]
            label_map[k] = [label[i]]

    columns.pop(idx)
    for k in data_map.keys():  # 按照各部分数据继续递归
        tree[best_feature][k] = self.expand_node(columns[:], np.delete(data_map[k], idx, axis=1), label_map[k], dep
    return tree
```

4. 分类

根据搭建的决策树进行分类。

```
def traverse_node(self, node, feature):
    best_feature = list(node.keys())[0] # 树节点的特征
    next_node = node[best_feature]
    idx = list(self.features).index(best_feature)
    if feature[idx] not in next_node.keys():  # 取值不再树分支中，返回数量最多的种类
        return self.max_label
    if type(next_node[feature[idx]]).__name__ == 'dict':  # 下一个节点不是叶子结点，递归下去
        label = self.traverse_node(next_node[feature[idx]], feature)
    else:
        label = next_node[feature[idx]]   # 下一节点是叶子结点，则返回label取值
    return label
```

5. fit 和 predit

```
def fit(self, feature, label):
    assert len(self.features) == len(feature[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
    '''
    训练模型
    feature为二维numpy（n*m）数组，每行表示一个样本，有m个特征
    label为一维numpy（n）数组，表示每个样本的分类标签

    提示：一种可能的实现方式为
    self.root = self.expand_node(feature, label, depth=1) # 从根节点开始分裂，模型记录根节点
    '''
    self.max_label = np.argmax(np.bincount(label))
    self.root = self.expand_node(list(self.features[:]), feature, label, depth=1)


def predict(self, feature):
    assert len(feature.shape) == 1 or len(feature.shape) == 2 # 只能是1维或2维
    '''
    预测
    输入feature可以是一个一维numpy数组也可以是一个二维numpy数组
    如果是一维numpy（m）数组则是一个样本，包含m个特征，返回一个类别值
    如果是二维numpy（n*m）数组则表示n个样本，每个样本包含m个特征，返回一个numpy一维数组

    提示：一种可能的实现方式为
    if len(feature.shape) == 1: # 如果是一个样本
        return self.traverse_node(self.root, feature) # 从根节点开始路由
    return np.array([self.traverse_node(self.root, f) for f in feature]) # 如果是很多个样本
    '''
    if len(feature.shape) == 1:
        return self.traverse_node(self.root, feature)
    return np.array([self.traverse_node(self.root, f) for f in feature])
```

执行后的准确率为0.7141

```
# 定义决策树模型，传入算法参数
DT = DecisionTree(classes=[0,1], features=feature_names, max_depth=5, min_samples_split=10, impurity_t='gini')

DT.fit(x_train, y_train) # 在训练集上训练
p_test = DT.predict(x_test) # 在测试集上预测，获得预测值
print(p_test) # 输出预测值
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 0]
accuracy: 0.7141
```

构建的树为：

```
In [8]: DT.root

Out[8]: {'brTotalGold': {3: {'blueTotalGold': {2: {'redWardsPlaced': {0: {'redWardsDestroyed': {0: 1,
                1: 1,
                2: 1,
                3: 0,
                4: 1}},
              1: 1,
              2: 1,
              3: 0}},
            3: {'redTotalGold': {1: {'blueKills': {2: 1, 3: 1, 1: 1}},
              2: {'blueWardsPlaced': {0: 1, 4: 1, 1: 1, 3: 1, 2: 1}},
              0: 1}},
            4: 1,
            1: 1}},
          1: {'blueAvgLevel': {2: {'blueWardsPlaced': {0: {'brTotalExperience': {1: 0,
                2: 0,
                0: 0}},
              1: {'blueAssists': {0: 0, 1: 0}},
              2: 0}},
            3: {'brAssists': {2: {'redWardsPlaced': {0: 0, 1: 0, 4: 0, 2: 0}},
              0: 0,
              1: {'blueDeaths': {2: 0, 1: 0, 4: 0, 3: 0, 0: 0}},
              3: 0}},
            1: 0,
            0: 0,
            4: 0}},
          2: {'brTotalExperience': {2: {'brKills': {2: {'brEliteMonsters': {4: 1,
                2: 0,
                3: 1,
                0: 0,
                1: 0}},
              3: {'redTotalExperience': {3: 1, 2: 1, 4: 0, 1: 1}},
              1: {'redTotalExperience': {3: 0, 2: 0, 4: 0}},
              4: 1}},
            3: {'brWardsDestroyed': {2: {'brDragons': {-1: 1, 1: 1, 0: 1}},
              1: {'blueKills': {2: 0, 0: 1, 1: 1, 3: 1}},
              0: 1,
              3: {'brKills': {2: 1, 3: 1}},
              4: 1}},
            1: {'brTotalJungleMinionsKilled': {2: {'brEliteMonsters': {2: 0,
                1: 0,
                3: 0,
                4: 1,
                0: 0}},
              3: 0,
              1: {'brEliteMonsters': {3: 0, 0: 0, 1: 0, 2: 0, 4: 1}},
              0: 0}}}},
          4: 1,
          0: 0}}
```