实验报告

背景

基于决策树算法预测英雄联盟红蓝双方胜负。

数据分析

数据源：[Kaggle eague-of-legends-diamond-ranked-games-10-min]()

数据预览

In [5]:
```
data_df.head(3)
```

Out[5]:

	gameId	blueWins	blueWardsPlaced	blueWardsDestroyed	blueFirstBlood	blueKills	blueDeaths	blueAssists	blueEliteMonsters	blueDragons	blueHeralds
0	4519157822	0	28	2	1	9	6	11	0	0	0
1	4523371949	0	12	1	0	5	5	5	0	0	0
2	4521474530	0	15	0	0	7	11	4	1	1	0

查看数据结构

In [6]:
```
data_df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 9879 entries, 0 to 9878
Data columns (total 40 columns):
 #   Column                        Non-Null Count  Dtype
---  ------                        --------------  -----
 0   gameId                        9879 non-null   int64
 1   blueWins                      9879 non-null   int64
 2   blueWardsPlaced               9879 non-null   int64
 3   blueWardsDestroyed            9879 non-null   int64
 4   blueFirstBlood                9879 non-null   int64
 5   blueKills                     9879 non-null   int64
 6   blueDeaths                    9879 non-null   int64
 7   blueAssists                   9879 non-null   int64
 8   blueEliteMonsters             9879 non-null   int64
 9   blueDragons                   9879 non-null   int64
 10  blueHeralds                   9879 non-null   int64
 11  blueTowersDestroyed           9879 non-null   int64
 12  blueTotalGold                 9879 non-null   int64
 13  blueAvgLevel                  9879 non-null   float64
 14  blueTotalExperience           9879 non-null   int64
 15  blueTotalMinionsKilled        9879 non-null   int64
 16  blueTotalJungleMinionsKilled  9879 non-null   int64
 17  blueGoldDiff                  9879 non-null   int64
 18  blueExperienceDiff            9879 non-null   int64
 19  blueCSPerMin                  9879 non-null   float64
 20  blueGoldPerMin                9879 non-null   float64
 21  redWardsPlaced                9879 non-null   int64
 22  redWardsDestroyed             9879 non-null   int64
 23  redFirstBlood                 9879 non-null   int64
 24  redKills                      9879 non-null   int64
 25  redDeaths                     9879 non-null   int64
 26  redAssists                    9879 non-null   int64
 27  redEliteMonsters              9879 non-null   int64
 28  redDragons                    9879 non-null   int64
 29  redHeralds                    9879 non-null   int64
 30  redTowersDestroyed            9879 non-null   int64
 31  redTotalGold                  9879 non-null   int64
 32  redAvgLevel                   9879 non-null   float64
 33  redTotalExperience            9879 non-null   int64
 34  redTotalMinionsKilled         9879 non-null   int64
 35  redTotalJungleMinionsKilled   9879 non-null   int64
 36  redGoldDiff                   9879 non-null   int64
 37  redExperienceDiff             9879 non-null   int64
 38  redCSPerMin                   9879 non-null   float64
 39  redGoldPerMin                 9879 non-null   float64
dtypes: float64(6), int64(34)
memory usage: 3.0 MB
```

- 共有9879条记录

- 红蓝双方各有19个特征，`blueWins`为label
- 特征数据类型全部为numerical
- 没有缺失数据

特征工程

决策数据对数据的scale和异常值不敏感，因此没必要做数据的标准化和异常值的检测。

差值特征

任务目标是预测红蓝双方的胜负，根据经验，数据的差值更能凸现优势的一方，因此基于原始数据的特征构建差值特征。

In [14]:
```python
drop_features = ['blueGoldDiff', 'redGoldDiff', 'blueExperienceDiff', 'redExperienceDiff']
data_df.drop(columns=drop_features, inplace=True)
```

构建所有原始特征的差值特征：

In [15]:
```python
feat_names = list(map(lambda feat: feat[3:], filter(lambda feat: feat.startswith('red'), data_df.columns)))
diff_features = []

for feat in feat_names:
    diff_feat_name = feat + 'BRDiff'
    blue_feat_name = 'blue' + feat
    red_feat_name = 'red' + feat
    data_df[diff_feat_name] = data_df[blue_feat_name] - data_df[red_feat_name]
    diff_features.append(diff_feat_name)
```

特征离散化

决策树模型适合处理类别特征的数据，不适合处理连续数据值性的特征，因此需要将数据值性特征进行离散化。

In [18]:
```python
BINS = 20
```

In [19]:
```python
discrete_df = data_df.copy()
for col in data_df.columns[1:]:
    if len(data_df[col].unique()) > BINS:
        discrete_df[col] = pd.cut(discrete_df[col], bins=BINS, labels=False)
```

模型算法

本实验使用决策树模型预测红蓝双方的胜负。

决策树算法描述

决策树是一颗二叉树，非叶子节点存储的特征和取值，叶子节点存储数据实例。

自顶向下递归的构建决策树：

- 选择**最佳**的决策属性A
- 将A作为当前节点的决策属性
- 根据属性的值分裂数据集，分别存储在左子树和右子树
- 当**训练样本被完美分类**，则退出循环，否则继续递归分裂

选择分裂特征和特征值

使用信息熵Entropy(或基尼系数Gini)衡量数据集的混杂度impurity，基于信息增益Information Gain选择最好的分裂属性。

信息增益：以信息熵为例，`IG = 分裂前的信息熵 - 分裂后的加权信息熵`

停止条件

1. 当训练样本的标签值全部相同，停止分裂
2. 当训练样本的特征值全部相同，停止分裂

过拟合

决策树模型很容易过拟合数据（训练集performance很好，测试集performance很差），因此需要进行正则化。

防止过拟合：

- Pre-pruning
    - 当节点的训练样本过少时，停止分裂
    - 当信息增益小于某个阈值，停止分裂
- Post-pruning
    - 将数据分成训练集和验证集，自底向上剪枝，提升验证集的performance
    - 规则后剪枝

决策树算法实现

- 使用信息熵`entropy()`衡量的数据impurity
- 实现`pre-pruning`防止过拟合
    - `min_sample_split` 当节点的数据数量少于设定的阈值时，不再继续分裂
    - `max_depth` 当树的深度大于设定的阈值时，不再继续分裂
- `build_tree()` 自顶向下、递归的构建决策树
- `fit(X, y)` 训练函数
- `predict(X)` 预测函数

决策树算法实现Python代码示例：

```
import collections
import numpy as np


class Node(object):
    """Tree node"""
    def __init__(self, column=None, value=None, left=None, right=None,
data=None):
        self.column = column
        self.value = value
```

```python
        self.left = left
        self.right = right
        self.data = data

    @property
    def is_leaf(self):
        return self.data is not None

    def __str__(self):
        return 'Tree node column index: %s value:%s' % (self.column,
self.value)


# sentinel node
empty = Node()


class DecisionTree(object):
    def __init__(self, classes, features, max_depth=10,
                 min_samples_split=10, impurity_t='entropy'):
        """
        :param classes: label classes
        :param features: feature names
        :param max_depth: max depth of decision tree
        :param min_samples_split: min samples of split
        :param impurity_t: impurity.
        """
        self.classes = classes
        self.features = features
        self.max_depth = max_depth
        self.min_samples_split = min_samples_split
        self.impurity_t = impurity_t
        self.root = empty

    @staticmethod
    def entropy(labels: np.ndarray):
        """Calculate entropy."""
        assert isinstance(labels, np.ndarray)

        n_labels = len(labels)
        counter_labels = list(collections.Counter(labels).values())
        probs = np.array(counter_labels) / n_labels # NOQA
        return -np.sum([p*np.log2(p) for p in probs])

    def gain(self, set1, set2):
        """Calculate split sets information gain."""
        assert isinstance(set1, np.ndarray)
        assert isinstance(set2, np.ndarray)
```

```
        total_set = np.concatenate((set1, set2))
        before_split_entropy = self.entropy(total_set)
        after_split_entropy = np.sum([self.entropy(s) * len(s) / len(total_set)
for s in (set1, set2)])
        return before_split_entropy - after_split_entropy

    @staticmethod
    def _split_set(xs, column, value):
        """
        Split set.
        :param xs: split set
        :param column: column index
        :param value: compare value
        :return split set row index
        """

        set1_idx = []
        set2_idx = []

        for row in range(len(xs)):
            if xs[row, column] <= value:
                set1_idx.append(row)
            else:
                set2_idx.append(row)

        return set1_idx, set2_idx

    def build_tree(self, xs, ys, depth=1):
        """
        Build decision tree recursively.

        :param xs: features
        :param ys: labels
        :param depth: tree depth start from 1
        :return tree node.
        """
        max_gain = 0.0
        best_column = None
        best_value = None
        best_split_set1 = None
        best_split_set2 = None

        # stop split
        # case1 all the labels are same
        if len(np.unique(ys)) == 1:
            return Node(data=ys)

        # case2 all the input are same
        for col in range(xs.shape[1]):
```

```
            if len(np.unique(xs[:, col])) > 1:
                break
        else:
            return Node(data=ys)

        # pre-pruning
        # min_samples_split
        if len(ys) < self.min_samples_split:
            # print('pre-pruning min_samples_split')
            return Node(data=ys)

        # max_depth
        if depth > self.max_depth:
            # print('pre-pruning max_depth')
            return Node(data=ys)

        # find best split feature and value
        for col in range(len(self.features)):
            for val in np.unique(xs[:, col]):
                set1_idx, set2_idx = self._split_set(xs, col, val)

                gain = self.gain(ys[set1_idx], ys[set2_idx])
                if gain > max_gain:
                    max_gain = gain
                    best_column = col
                    best_value = val
                    best_split_set1 = set1_idx
                    best_split_set2 = set2_idx

        node = Node(best_column, best_value)
        node.left = self.build_tree(xs[best_split_set1, :],
ys[best_split_set1], depth+1)
        node.right = self.build_tree(xs[best_split_set2, :],
ys[best_split_set2], depth+1)

        return node

    def traverse_tree(self, x):
        """Traverse decision tree."""
        assert self.root != empty

        root = self.root
        while True:
            if root.is_leaf:
                # leaf node
                return collections.Counter(root.data).most_common(1)[0][0]

            if x[root.column] < root.value:
                root = root.left
```

```python
            else:
                root = root.right

    def fit(self, xs, ys):
        """
        Train decision tree.
        :param xs: train features
        :pram ys: train labels
        :return None
        """
        assert len(self.features) == len(xs[0])
        self.root = self.build_tree(xs, ys)

    def predict(self, xs):
        """
        Predict.
        :param xs: predict features
        :return predict labels
        """
        assert len(xs.shape) in (1, 2)

        if len(xs.shape) == 1:
            return np.array([self.traverse_tree(xs)])

        return np.array([self.traverse_tree(x) for x in xs])
```

实验结果

In [22]:

```python
tree_clf = DecisionTree(classes=[0,1], features=discrete_features, max_depth=10, min_samples_split=30)

%time tree_clf.fit(x_train, y_train)
print('Accuracy: %.4f' % accuracy_score(y_test, tree_clf.predict(x_test)))
```

```
CPU times: user 27.2 s, sys: 57.6 ms, total: 27.3 s
Wall time: 27.3 s
Accuracy: 0.6690
```