ATEC 任务报告

学号：20307110315
姓名：周训哲

1.任务描述

利用 lightgbm 模型实现互联网交易欺诈识别。具体而言，作者尝试了多种实现方法，如使用 gbdt、rf、goss、dart 的 boost 类型，采用 is_unbalance 参数设置是否为不平衡数据，采用 f1_score 和 roc_auc 的评分函数以及对应损失函数，采用过采样、欠采样、SMOTE 等数据处理方法，等等一系列处理不平衡数据的方法。最终实验下来发现，采用 dart 的 boost 类型的不平衡参数训练方法在实训的模型中的得分最高。

2.数据集描述

采用何种数据集开展实验？

本次任务数据来源于用户转账到卡的交易数据，每行数据代表用户发起的一笔支付。此次建模任务需要 将那些因用户被电信诈骗而发起的转账交易识别出来，从而减少资金损失，保障用户资金安全。

数据集有什么特点？

有极强的不平衡性，由于互联网欺诈行为和普通交易相比数量实在过少，所以在数据处理的时候要着重考虑数据的不平衡性。除此以外有很多特征值并非是易于拟合的数字，而是字符串序列类型，所以在拟合数据时还要关注数据的类型。

数据集的特征值有什么？

本次每个数据集的数据源有 3 份：

driver_data，建模样本表，每行数据是一笔交易，数据可以分为几部分：交易主键，交易主体，交易属性，数值型变量，行为序列编码，是否欺诈的标签（只有训练数据集才有）；

event_data_card，driver 数据中收益卡近 15 天(从 driver 表中的交易时间算起)的历史收款数据，包含交易主键，交易主体，交易属性；

event_data_user，driver 数据中用户近 15 天的历史付款数据，包含交易主键，交易主体，交易属性

交易主键

字段	字段含义	字段类型
e_id	交易主键	STRING

交易主体

字段	字段含义	字段类型
a_k_id	用户主键	STRING
c_k_id	收款卡主键；收款方为卡时才可能有该属性	STRING

字段	字段含义	字段类型
time	发生时间	STRING
a_id_1	用户相关 id_1	STRING
b_id_1	收款方相关 id_1;当收款方为体系内账号时才可能有值，卡为体系外	STRING
c_id_1	收款卡相关 id_1;当收款方为卡时才可能有该属性	STRING
a_info_1	用户信息_1;与用户基本属性有关	STRING
b_info_1	收款方信息_1;当收款方为体系内账号时才可能有值，与收款方基本属性有关	STRING
c_info_1	收款卡信息_1;当收款方为卡时才可能有该属性，与收款卡的基本属性有关	STRING
a_info_2	用户信息_2;与用户基本属性有关	STRJNG
b_info_2	收款方信息_2;当收款方为体系内账号时才可能有值，与收款方基本属性有关	STRING
c_info_2	收款卡信息_2;当收款方为卡时才可能有该属性，与收款卡的基本属性有关	STRING
_info_2	交易信息_2;与设备位置信息相关	STRING
e_info_3	交易信思_3;与设备位置信思相关	STRING
a_info_3	用户信息_3;与用户基本属性相关	STRING
_info_4	交易信息_4;与设备位置信息相关	STRING
e_info_5	交易信息_5;与设备位置信息相关	STRING
a_info_4	用户信息_4;与用户基本属性相关	STRING
a_info_5	用户信息_5;与用户基本属性相关	DOUBLE
hh	发生时间-小时	BIGINT
week_day	发生时间-周几	BIGINT
c_id_2	收款卡相关 id;衍生于收款卡主键	STRJNG
c_info_3	收款卡信息_3;当收款方为卡时才可能有该属性，与收款卡的基本属性有关	STRING
e_info_6	交易信息_6;反映交易类型	STRING
a_info_6	用户信思_6;与用户基本属性相关	STRING
a_info_7	用户信息_7;与用户基本属性相关	STRING
c_info_4	收款卡信息_4;当收款方为卡时才可能有该属性，与收款卡的基本属性有关	STRING
c_info_5	收款卡信息_5;当收款方为卡时才可能有该属性，与收款卡的基本属性有关	STRING
if_succ	交易是否完成，即交易可能被中止；收款方为卡时才有意义	BIGINT
e_info_13	交易信息_13;与金额相关	BIGINT
dt	日 期	STRING

3.方法介绍

3.1 数据预处理（如有）

从原始数据到输入模型的数据之间，经过了哪些处理？这些处理有何作用？

有多种处理方法。

首先介绍权重法，即处理少量的样本时，赋予更高的权重，使得训练样本尽量均衡：

```
train_weight = [1 if i == 0 else 3 for i in dtrain.get_label().tolist()]
valid_weight = [1 if i == 0 else 3 for i in dvalid.get_label().tolist()]
dtrain = lgb.Dataset(trade_var_label[trade_var_label['dt']<=20210710][model_var],
                     label=trade_var_label[trade_var_label['dt']<=20210710]['is_fraud'],
                     weight=train_weight, silent=True)
dvalid = lgb.Dataset(trade_var_label[trade_var_label['dt']>=20210711][model_var],
                     label=trade_var_label[trade_var_label['dt']>=20210711]['is_fraud'],
                     weight=valid_weight, silent=True)
```

然后是 SMOTE 方法：

```
class Smote:
    def __init__(self, samples, N=10, k=5):
        self.n_samples, self.n_attrs = samples.shape
        self.N = N
        self.k = k
        self.samples = samples
        self.newindex = 0
    def over_sampling(self):
        N = int(self.N / 100)
        self.synthetic = np.zeros((self.n_samples * N, self.n_attrs))
        neighbors = NearestNeighbors(n_neighbors=self.k).fit(self.samples)
        print('neighbors', neighbors)
        for i in range(len(self.samples)):
            print('samples', self.samples[i])
            nnarray = neighbors.kneighbors(self.samples[i].reshape((1, -1)), return_distance=False)[
                0]  # Finds the K-neighbors of a point.
            print('nna', nnarray)
            self._populate(N, i, nnarray)
        return self.synthetic
    def _populate(self, N, i, nnarray):
        for j in range(N):
            print('j', j)
            nn = random.randint(0, self.k - 1)  # 包括end
            dif = self.samples[nnarray[nn]] - self.samples[i]
            gap = random.random()
            self.synthetic[self.newindex] = self.samples[i] + gap * dif
            self.newindex += 1
            print(self.newindex)
```

还有对缺失值的填充（初始文档中填充为-1）：

```python
def median_fillna(df, features):
    for feature in features:
        df[feature + '_meadian'] = df[feature].median()
    return df
# 连续特征的处理方式
def continues_fillna(df, features):
    df = median_fillna(df, features)
    # df=interpolate_fillna(df,features)
    return df
# 使用-1填充离散型特征，并生成是否为空的特征列
def category_fillna(df, category_cols):
    for category_col in category_cols:
        df[category_col].fillna(-1, inplace=True)
    return df
# 为train和test数据标记是否是离散值，当数据unquie唯一值个数超过threld的时候，则被认为是连续特征，否则是离散特征。
# 并且填充缺失值
def set_cate_type_and_fillna(train_x, threld=80):
    # 给离散列转换类型,得到离散值
    data_clo_unique_num = pd.DataFrame(train_x.apply(lambda x: len(x.unique())), columns=['unique'])
    for i in data_clo_unique_num.index:
        print(i, " ", data_clo_unique_num.loc[i, 'unique'])
    cate_cols = set(data_clo_unique_num[data_clo_unique_num['unique'] < threld].index)
    cont_cols = set(train_x.columns) - cate_cols
    for i in cate_cols:
        train_x = train_x.astype('category')
    train_x = category_fillna(train_x, cate_cols)
    train_x = continues_fillna(train_x, cont_cols)
```

除了这几种基本的填充方式，还有利用随机森林的回归填充：

```python
rfr = RandomForestRegressor(random_state=0, n_estimators=200, n_jobs=-1)

known_data = trade_var_label[trade_var_label.card_e_info_2_rank.notnull()]
unknown_data = trade_var_label[trade_var_label.card_e_info_2_rank.isnull()]
card_e_info_2_rank = known_data.iloc[:, 28]
card_e_info_2_rank = np.array(card_e_info_2_rank)
X = known_data.iloc[:, :28]
X = np.array(X)
rfr.fit(X, card_e_info_2_rank)
trade_var_label.loc[(trade_var_label.card_e_info_2_rank.isnull()), 'card_e_info_2_rank'] = rfr.predict(unknown_data.iloc[:, :28])
known_data = trade_var_label[trade_var_label.user__trade_cnt_nofail_15d_1d.notnull()]
unknown_data = trade_var_label[trade_var_label.user__trade_cnt_nofail_15d_1d.isnull()]
user__trade_cnt_nofail_15d_1d = known_data.iloc[:, 29]
user__trade_cnt_nofail_15d_1d = np.array(user__trade_cnt_nofail_15d_1d)
X = known_data.iloc[:, :28]
X = np.array(X)
rfr.fit(X, user__trade_cnt_nofail_15d_1d)
trade_var_label.loc[(trade_var_label.user__trade_cnt_nofail_15d_1d.isnull()), 'user__trade_cnt_nofail_15d_1d'] = rfr.predict(
    unknown_data.iloc[:, :28])
known_data = trade_var_label[trade_var_label.user_amt_sum_nofail_15d_1d.notnull()]
unknown_data = trade_var_label[trade_var_label.user_amt_sum_nofail_15d_1d.isnull()]
user_amt_sum_nofail_15d_1d = known_data.iloc[:, 30]
user_amt_sum_nofail_15d_1d = np.array(user_amt_sum_nofail_15d_1d)
X = known_data.iloc[:, :28]
X = np.array(X)
rfr.fit(X, user_amt_sum_nofail_15d_1d)
trade_var_label.loc[(trade_var_label.user_amt_sum_nofail_15d_1d.isnull()), 'user_amt_sum_nofail_15d_1d'] = rfr.predict(
    unknown_data.iloc[:, :28])
known_data = trade_var_label[trade_var_label.user__trade_cnt_fail_15d_1d.notnull()]
unknown_data = trade_var_label[trade_var_label.user__trade_cnt_fail_15d_1d.isnull()]
user__trade_cnt_fail_15d_1d = known_data.iloc[:, 31]
user__trade_cnt_fail_15d_1d = np.array(user__trade_cnt_fail_15d_1d)
X = known_data.iloc[:, :28]
```

最终由于特征值有很多都是字符串类型，所以不能采用随机森林回归进行填充，然后分别对其他填充方式进行测试，发现效果都不如直接填充-1 的效果好，故最终还是采用原始方式——填充缺失值为-1。在测试中发现，对于 lightgbm 中我选择的模型，使用权重后的

auc 评分会增加，但是官网的 score 却会降低，故没有采用权重处理。最后对于 SMOTE 方法，发现其效果不如默认参数“un_balance”，故最终没有使用 SMOTE 过采样处理。

3.2 算法描述

LightGBM 简介：

GBDT (Gradient Boosting Decision Tree) 是机器学习中一个长盛不衰的模型，其主要思想是利用弱分类器（决策树）迭代训练以得到最优模型，该模型具有训练效果好、不易过拟合等优点。GBDT 不仅在工业界应用广泛，通常被用于多分类、点击率预测、搜索排序等任务；在各种数据挖掘竞赛中也是致命武器，据统计 Kaggle 上的比赛有一半以上的冠军方案都是基于 GBDT。而 LightGBM（Light Gradient Boosting Machine）是一个实现 GBDT 算法的框架，支持高效率的并行训练，并且具有更快的训练速度、更低的内存消耗、更好的准确率、支持分布式可以快速处理海量数据等优点。

LightGBM 优化：

为了能够在不损害准确率的条件下加快 GBDT 模型的训练速度，lightGBM 在传统的 GBDT 算法上进行了如下优化：

- 基于 Histogram 的决策树算法。
- 单边梯度采样 Gradient-based One-Side Sampling(GOSS)：使用 GOSS 可以减少大量只具有小梯度的数据实例，这样在计算信息增益的时候只利用剩下的具有高梯度的数据就可以了，相比 XGBoost 遍历所有特征值节省了不少时间和空间上的开销。
- 互斥特征捆绑 Exclusive Feature Bundling(EFB)：使用 EFB 可以将许多互斥的特征绑定为一个特征，这样达到了降维的目的。
- 带深度限制的 Leaf-wise 的叶子生长策略：大多数 GBDT 工具使用低效的按层生长 (level-wise) 的决策树生长策略，因为它不加区分的对待同一层的叶子，带来了很多没必要的开销。实际上很多叶子的分裂增益较低，没必要进行搜索和分裂。LightGBM 使用了带有深度限制的按叶子生长 (leaf-wise) 算法。
- 直接支持类别特征(Categorical Feature)
- 支持高效并行
- Cache 命中率优化

基本原理：

基于 Histogram 的决策树算法：

（1）直方图算法

Histogram algorithm 应该翻译为直方图算法，直方图算法的基本思想是：先把连续的浮点特征值离散化成 个整数，同时构造一个宽度为 的直方图。在遍历数据的时候，根据离散化后的值作为索引在直方图中累积统计量，当遍历一次数据后，直方图累积了需要的统计量，然后根据直方图的离散值，遍历寻找最优的分割点。

（2）直方图做差加速

LightGBM 另一个优化是 Histogram（直方图）做差加速。一个叶子的直方图可以由它的父亲节点的直方图与它兄弟的直方图做差得到，在速度上可以提升一倍。

带深度限制的 Leaf-wise 算法：

LightGBM 采用 Leaf-wise 的增长策略，该策略每次从当前所有叶子中，找到分裂增益

最大的一个叶子，然后分裂，如此循环。因此同 Level-wise 相比，Leaf-wise 的优点是：在分裂次数相同的情况下，Leaf-wise 可以降低更多的误差，得到更好的精度；Leaf-wise 的缺点是：可能会长出比较深的决策树，产生过拟合。因此 LightGBM 会在 Leaf-wise 之上增加了一个最大深度的限制，在保证高效率的同时防止过拟合。

单边梯度采样算法：

GOSS 是一个样本的采样算法，目的是丢弃一些对计算信息增益没有帮助的样本留下有帮助的。根据计算信息增益的定义，梯度大的样本对信息增益有更大的影响。因此，GOSS 在进行数据采样的时候只保留了梯度较大的数据，但是如果直接将所有梯度较小的数据都丢弃掉势必会影响数据的总体分布。所以，GOSS 首先将要进行分裂的特征的所有取值按照绝对值大小降序排序（XGBoost 一样也进行了排序，但是 LightGBM 不用保存排序后的结果），选取绝对值最大的 个数据。然后在剩下的较小梯度数据中随机选择 个数据。接着将这 个数据乘以一个常数 ，这样算法就会更关注训练不足的样本，而不会过多改变原数据集的分布。最后使用这 个数据来计算信息增益。

互斥特征捆绑算法：

将相互独立的特征进行绑定是一个 NP-Hard 问题，LightGBM 的 EFB 算法将这个问题转化为图着色的问题来求解，将所有的特征视为图的各个顶点，将不是相互独立的特征用一条边连接起来，边的权重就是两个相连接的特征的总冲突值，这样需要绑定的特征就是在图着色问题中要涂上同一种颜色的那些点（特征）。此外，我们注意到通常有很多特征，尽管不是%相互排斥，但也很少同时取非零值。如果我们的算法可以允许一小部分的冲突，我们可以得到更少的特征包，进一步提高计算效率。经过简单的计算，随机污染小部分特征值将影响精度最多 ， 是每个绑定中的最大冲突比率，当其相对较小时，能够完成精度和效率之间的平衡。具体步骤可以总结如下：

1. 构造一个加权无向图，顶点是特征，边有权重，其权重与两个特征间冲突相关；
2. 根据节点的度进行降序排序，度越大，与其它特征的冲突越大；
3. 遍历每个特征，将它分配给现有特征包，或者新建一个特征包，使得总体冲突最小。

算法允许两两特征并不完全互斥来增加特征捆绑的数量，通过设置最大冲突比率 来平衡算法的精度和效率。

特征合并算法，其关键在于原始特征能从合并的特征中分离出来。绑定几个特征在同一个 bundle 里需要保证绑定前的原始特征的值可以在 bundle 中识别，考虑到 histogram-based 算法将连续的值保存为离散的 bins，我们可以使得不同特征的值分到 bundle 中的不同 bin（箱子）中，这可以通过在特征值中加一个偏置常量来解决。比如，我们在 bundle 中绑定了两个特征 A 和 B，A 特征的原始取值为区间 ，B 特征的原始取值为区间，我们可以在 B 特征的取值上加一个偏置常量，将其取值范围变为，绑定后的特征取值范围为 ，这样就可以放心的融合特征 A 和 B 了。

整体流程：

①准备数据，对数据进行预处理。

②对数据集加上参数进行训练并保存模型

③使用已经训练好的模型进行排序，并将排序好的数据传输

④对排序好的预测数据进行打分：提交结果为每个测试样本是 1 的概率，也就是 is_fraud 为 1 的概率。评价方法为 3 个打扰率下欺诈金额的召回比例的加权（越大越好），打扰率为 0.005，0.01，0.05，加权系数分别为 0.4，0.3，0.3。

LightGBM 的优缺点

优点

（1）速度更快

- LightGBM 采用了直方图算法将遍历样本转变为遍历直方图，极大的降低了时间复杂度；
- LightGBM 在训练过程中采用单边梯度算法过滤掉梯度小的样本，减少了大量的计算；
- LightGBM 采用了基于 Leaf-wise 算法的增长策略构建树，减少了很多不必要的计算量；
- LightGBM 采用优化后的特征并行、数据并行方法加速计算，当数据量非常大的时候还可以采用投票并行的策略；
- LightGBM 对缓存也进行了优化，增加了缓存命中率；

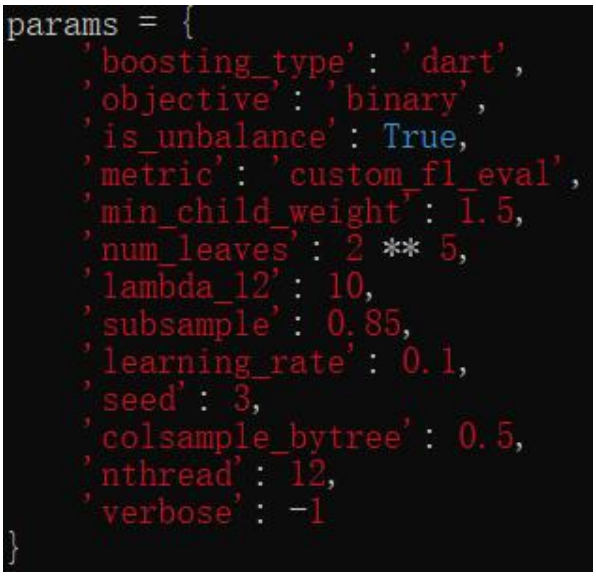
（2）内存更小

- XGBoost 使用预排序后需要记录特征值及其对应样本的统计值的索引，而 LightGBM 使用了直方图算法将特征值转变为 bin 值，且不需要记录特征到样本的索引，将空间复杂度从 降低为 ，极大的减少了内存消耗；
- LightGBM 采用了直方图算法将存储特征值转变为存储 bin 值，降低了内存消耗；
- LightGBM 在训练过程中采用互斥特征捆绑算法减少了特征数量，降低了内存消耗。

缺点

- 可能会长出比较深的决策树，产生过拟合。因此 LightGBM 在 Leaf-wise 之上增加了一个最大深度限制，在保证高效率的同时防止过拟合；
- Boosting 族是迭代算法，每一次迭代都根据上一次迭代的预测结果对样本进行权重调整，所以随着迭代不断进行，误差会越来越小，模型的偏差（bias）会不断降低。由于 LightGBM 是基于偏差的算法，所以会对噪点较为敏感；
- 在寻找最优解时，依据的是最优切分变量，没有将最优解是全部特征的综合这一理念考虑进去；

具体参数



4.实验结果分析

4.1 评价指标

提交结果为每个测试样本是 1 的概率，也就是 is_fraud 为 1 的概率。评价方法为 3 个打扰率下欺诈金额的召回比例的加权（越大越好），打扰率为 0.005，0.01，0.05，加权系数分别为 0.4，0.3，0.3。

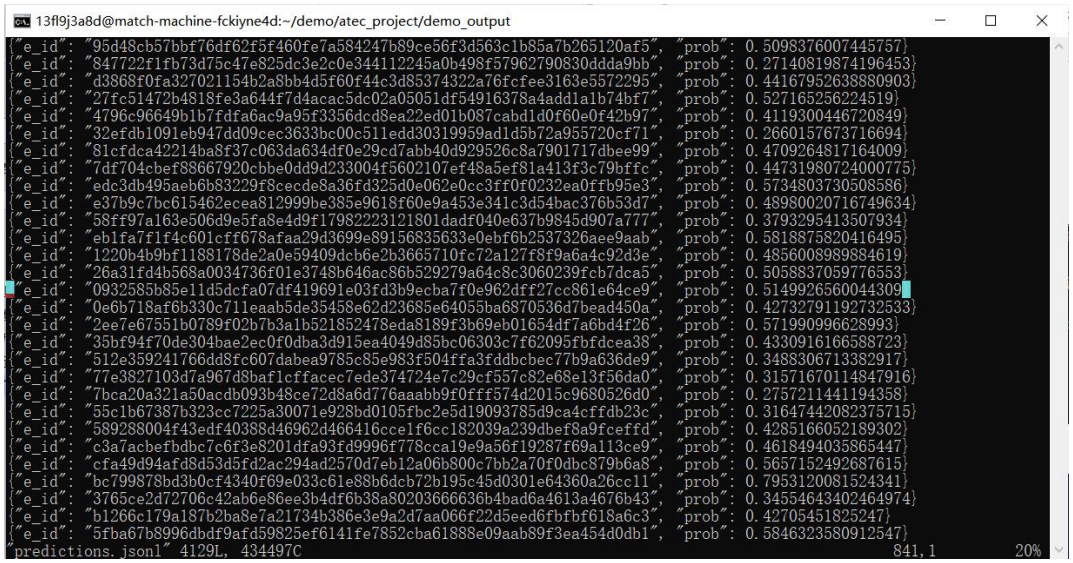
4.2 定量评价结果

```
[13f19j3a8d@match-machine-fckiyne4d atec_project]$ [13f19j3a8d@match-machine-fckiyne4d atec_project]$ LOCAL=1 sh run.sh
start python code
Loading Time = 6.843639850616455
/home/13f19j3a8d/demo/atec_project/your_name_env/lib/python3.7/site-packages/lightgbm/basic.py:1491: UserWarning: 'silen
t' argument is deprecated and will be removed in a future release of LightGBM. Pass 'verbose' parameter via 'params' ins
tead.
  _log_warning("'silent' argument is deprecated and will be removed in a future release of LightGBM. "
auc: 0.8405862970117032
start read data
start produce var
start predict
start write  data
run success
```

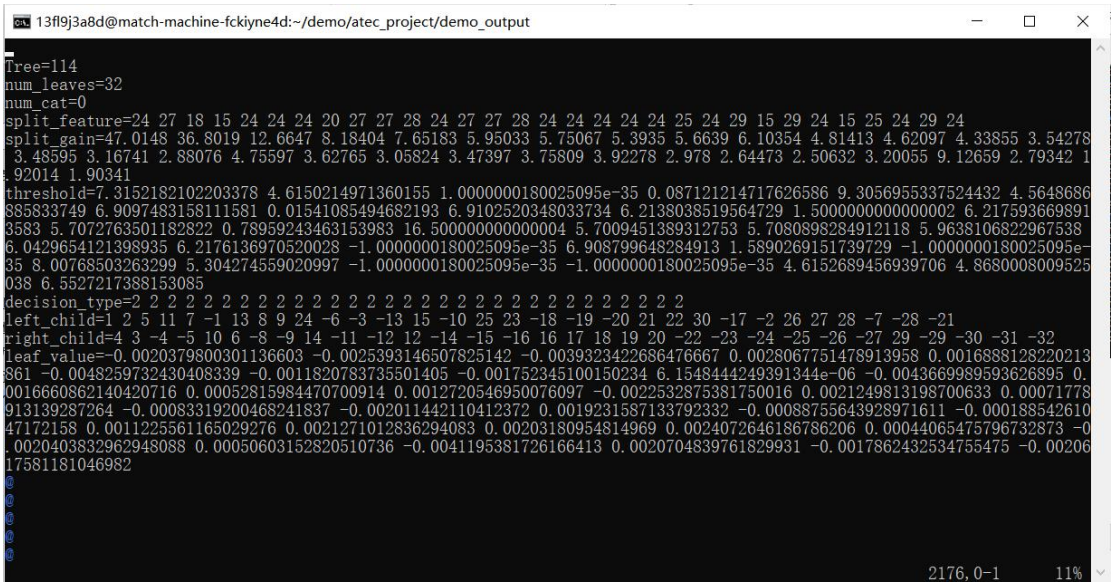
在本地运行的 auc 评分为 8.40 分

云端测试评分为 0.227945 分

部分预测数据输出：



模型中某一棵训练树:



由于是在命令行运行，并且是提交到云端，所以无法实现作图操作。

5. 总结

本次实训是切实结合实际数据进行的机器学习模型测试，需要自己寻找特定模型，自己调整参数，对机器学习的了解有极高的要求，也是对我们一个极好的锻炼。通过本次实训，我充分了解了处理不平衡数据的一系列方法，了解了 lightgbm 模型的工作原理及实现。由于进行了一次完整的模型选择以及调参、数据处理等操作。我也对机器学习的实际应用有了更深的了解。