Final Project: 某闯关类手游用户流失预测

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1月7日

目录

一、案例简介

手游在当下的日常娱乐中占据着主导性地位，成为人们生活中放松身心的一种有效途径。近年来，各种类型的手游，尤其是闯关类的休闲手游，由于其对碎片化时间的利用取得了非常广泛的市场。然而在此类手游中，新用户流失是一个非常严峻的问题，有相当多的新用户在短暂尝试后会选择放弃，而如果能在用户还没有完全卸载游戏的时候针对流失可能性较大的用户施以干预（例如奖励道具、暖心短信），就可能挽回用户从而提升游戏的活跃度和公司的潜在收益，因此用户的流失预测成为一个重要且挑战性的问题。在毕业项目中我们将从真实游戏中非结构化的日志数据出发，构建用户流失预测模型，综合已有知识设计适合的算法解决实际问题。

二、作业说明

- 根据给出的实际数据（包括用户游玩历史，关卡特征等），预测测试集中的用户是否为流失用户（二分类）；
- 方法不限，使用百度云进行评测，评价指标使用 AUC；
- 提交代码与实验报告，报告展示对数据的观察、分析、最后的解决方案以及不同尝试的对比等；
- 最终评分会参考达到的效果以及对所尝试方法的分析。

三、数据概览

本次使用的是一个休闲类闯关手游的数据，用户在游戏中不断闯关，每一关的基本任务是在限定步数内达到某个目标。每次闯关可能成功也可能失败，一般情况下用户只在完成一关后进入下一关，闯关过程中可以使用道具或提示等帮助。

对大多数手游来说，用户流失往往发生在早期，因此次周的留存情况是公司关注的一个重点。本次数据选取了 2020.2.1 注册的所有用户在 2.1-2.4 的交互数据，数据经过筛选保证这些注册用户在前四日至少有两日登录。流失的定义则参照次周（2.7-2.13）的登录情况，如果没有登录为流失。

本次的数据和以往结构化的形式不同，展现的是更原始的数据记录，更接近公司实际日志的形式，共包含 5 个文件：

train.csv

训练集用户，包括用户 id（从 1 开始）以及对应是否为流失用户的 label（1：流失，0：留存）。

训练集共 8158 个用户，其中流失用户大约占 1/3，需要注意的是为了匿名化，这里数据都经过一定的非均匀抽样处理，流失率并不反映实际游戏的情况，用户与关卡的 id 同样经过了重编号，但对于流失预测任务来说并没有影响。

dev.csv

验证集格式和训练集相同，主要为了方便离线测试与模型选择。

test.csv

测试集只包含用户 id，任务就是要预测这些用户的流失概率。

level_seq.csv

这个是核心的数据文件，包含用户游玩每个关卡的记录，每一条记录是对某个关卡的一次尝试，具体每列的含义如下：

- user_id：用户 id，和训练、验证、测试集中的可以匹配；
- level_id：关卡 id；
- f_success：是否通关（1：通关，0：失败）；
- f_duration：此次尝试所用的时间（单位 s）；
- f_reststep：剩余步数与限定步数之比（失败为 0）；
- f_help：是否使用了道具、提示等额外帮助（1：使用，0：未使用）；
- time：时间戳。

level_meta.csv

每个关卡的一些统计特征，可用于表示关卡，具体每列的含义如下：

- f_avg_duration：平均每次尝试花费的时间（单位 s，包含成功与失败的尝试）；
- f_avg_passrate：平均通关率；
- f_avg_win_duration：平均每次通关花费的时间（单位 s，只包含通关的尝试）；
- f_avg_retrytimes：平均重试次数（第二次玩同一关算第 1 次重试）；
- level_id：关卡 id，可以和 level_seq.csv 中的关卡匹配。

四、Tips

- 一个基本的思路可以是：根据游玩关卡的记录为每个用户提取特征 → 结合 label 构建表格式的数据集 → 使用不同模型训练与测试；
- 还可以借助其他模型（如循环神经网络）直接对用户历史序列建模；
- 数据量太大运行时间过长的话，可以先在一个采样的小训练集上调参；
- 集成多种模型往往能达到更优的效果；
- 可以使用各种开源工具。

五、特征工程

5.1 特征提取

5.1.1 总体特征

- ‘last_day’: 最后登录的日期（日）
- ‘rounds’: 游戏总轮数
- ‘num_days’: 登录的天数
- ‘max_level’: 最大关卡

5.1.2 单天特征

- ‘last_level’: 当天最后停留关卡
- ‘last_trytimes’: 最后关卡尝试次数
- ‘max_retrytimes’: 每一关最多尝试次数
- ‘num_help’: 使用help的次数
- ‘num_levels’: 当天尝试的关卡数
- ‘num_rounds’: 当天玩了几轮游戏
- ‘total_duration’: 当天玩的时间总数
- ‘passrate_ratio’: 玩家通过率和平均通过率的比值的均值
    - 求均值（每一关 玩家passrate / f_avg_passrate）
- ‘retrytimes_ratio’: 尝试次数的加权和
    - 权重：`weight = math.exp(meta_df[meta_df['level_id'] == level]['f_avg_retrytimes'].values[0])`
- ‘last_time’: 前一天是否登录

5.1.3 代码实现

```
import math


feature_names = ['last_day', 'rounds', 'num_days', 'max_level',

                 '1last_level', '1last_trytimes', '1max_retrytimes',
```

```
                  '1num_help','1passrate_ratio',
                  '1num_levels', '1num_rounds',
                  '1total_duration', '1win_duration_ratio',
'1retrytimes_ratio', '1last_time',
```

```
def extract_day_features(seqDf, featureDf, meta_df):
  zeros = np.zeros(featureDf.shape[0])

  # for c in feature_names: #将每一列初始化为0
  #   featureDf[c] = zeros

  # for i in range(5): #跑5个id测试一下

  for j, i in enumerate(featureDf.index): #遍历featureDf每一个Id
    if j % 500 == 0: print(j)

    id = featureDf['user_id'][i] #取出id
    # print(id)
    id_df = seqDf.loc[seqDf['user_id'] == id] #取出这个id的seq数据

    # 提取总体的特征

    days = np.array(id_df['day']) #所有玩的day
    featureDf.loc[i, 'last_day'] = days.max() #最后一天玩的日子
    featureDf.loc[i, 'rounds'] = id_df.shape[0] #一共玩了多少轮
    days = set(days)
    featureDf.loc[i, 'num_days'] = len(days) #有几天玩了
    featureDf.loc[i, 'max_level'] = id_df['level_id'].max()

    for day in days:
      day_df = id_df.loc[id_df['day'] == day] #取出这一天的数据

      #当天玩的最大关卡
      last_level = str(day) +'last_level'
      lastlevel = day_df['level_id'].max()
      featureDf.loc[i, last_level] = lastlevel
```

```
        #最后关卡的尝试次数
        last_trytimes = str(day) + 'last_trytimes'
        t_df = day_df.loc[day_df['level_id'] == lastlevel]
        lasttrytimes = t_df.shape[0]
        featureDf.loc[i, last_trytimes] = lasttrytimes

        #max_retrytimes：每一关最多retry了几次
        #num_levels: 当天通关数（玩了多少个不同的关卡？ 存在问题：assume玩的都通关了）
        #passrate_ratio
        #retrytimes_ratio

        max_retrytimes = str(day) + 'max_retrytimes'
        num_levels = str(day) + 'num_levels'
        passrate_ratio = str(day) +'passrate_ratio'
        retrytimes_ratio = str(day) +'retrytimes_ratio'

        passsum = 0
        levels = set(day_df['level_id'])
        maxretrytimes = 0
        ratiosum = 0
        weights = 0

        for level in levels:
          t_df = day_df.loc[day_df['level_id'] == level]
          retrytimes = t_df.shape[0] - t_df['f_success'].sum() #所有尝试次数，减去成功的
          if retrytimes > maxretrytimes:
            maxretrytimes = retrytimes

          passrate = t_df['f_success'].sum() / t_df.shape[0]
          avg = meta_df[meta_df['level_id'] == level]['f_avg_passrate'].values[0]
          if avg == 0 : passratio = 1
          else: passratio = passrate / avg
          passsum += passratio
```

```
        weight = math.exp(meta_df[meta_df['level_id'] ==
level]['f_avg_retrytimes'].values[0])
        reratio = retrytimes * weight
        ratiosum += reratio
        weights += weight


    passrateratio = passsum / len(levels)
    retrytimesratio = ratiosum / weights

    featureDf.loc[i, max_retrytimes] = maxretrytimes
    featureDf.loc[i, num_levels] = len(levels)
    featureDf.loc[i, passrate_ratio] = passrateratio
    featureDf.loc[i, retrytimes_ratio] = retrytimesratio


    #当天玩了几轮
    num_rounds = str(day) + 'num_rounds'
    featureDf.loc[i, num_rounds] =  day_df.shape[0]

    #当天玩的时间总数
    #当天使用help的总数
    total_duration = str(day) + 'total_duration'
    totalduration = 0
    num_help = str(day) +'num_help'
    numhelp = 0
    for round in day_df.index:
        totalduration += day_df['f_duration'][round]
        numhelp += day_df['f_help'][round]

    featureDf.loc[i, total_duration] = totalduration
    featureDf.loc[i, num_help] = numhelp

    #win_duration_ratio
    win_duration_ratio = str(day) + 'win_duration_ratio'
    winsum = 0
    t_df = day_df.loc[day_df['f_success'] == 1]
    for round in t_df.index:
        level = t_df['level_id'][round]
```

```
        win_ratio = t_df['f_duration'][round] /
meta_df[meta_df['level_id'] == level]['f_avg_win_duration'].values[0]
        winsum += win_ratio
      if t_df.shape[0] == 0: windurationratio = 0
      else: windurationratio = winsum / t_df.shape[0]
      featureDf.loc[i, win_duration_ratio] = windurationratio

      #前一天玩了吗
      last_time = str(day) + 'last_time'
      lastday = day - 1
      if lastday in days:
        featureDf.loc[i, last_time] = 1
```

5.2 特征处理

5.2.0 特征预览

从下图可以发现，有些特征中较大的极值比较多。如果直接进行标准化或归一化，特征值会聚集在较小的区间。可以考虑去掉极值后再等分or等比划分。

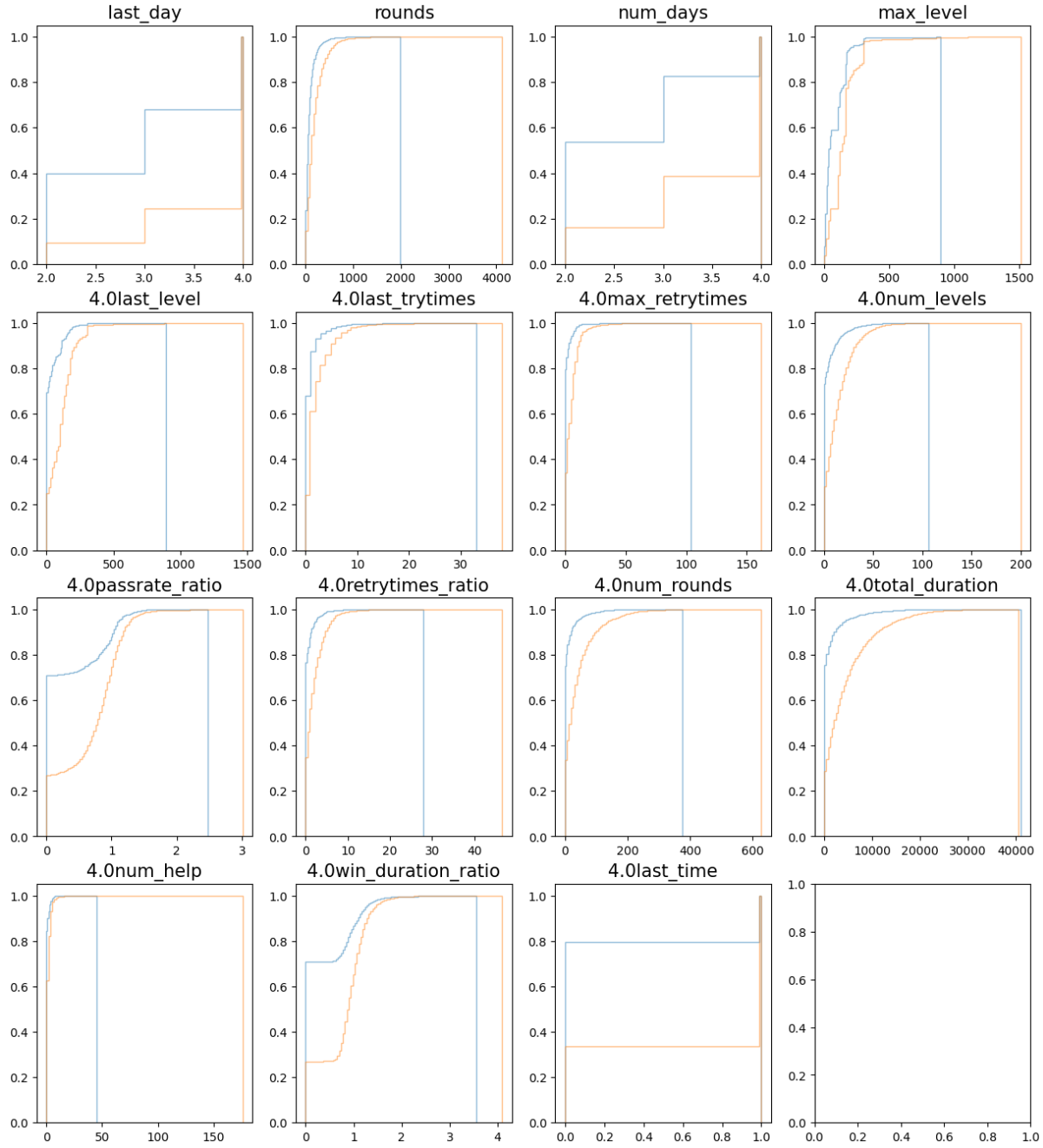


Figure 1. 特征预览

5.2.1 去极值离散化（等区间）

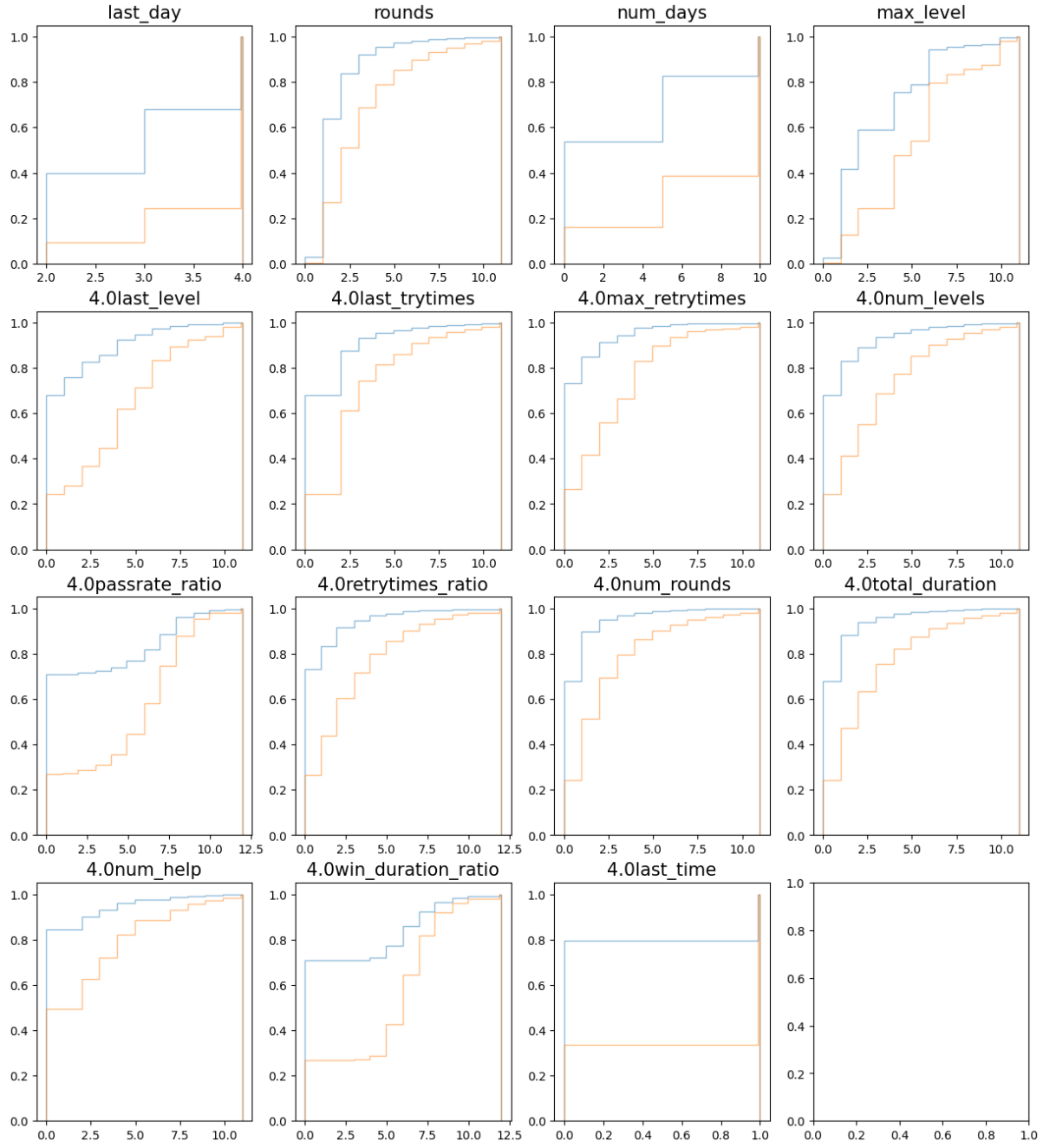


Figure 2. 等区间划分

5.2.2 去极值离散化（等比）

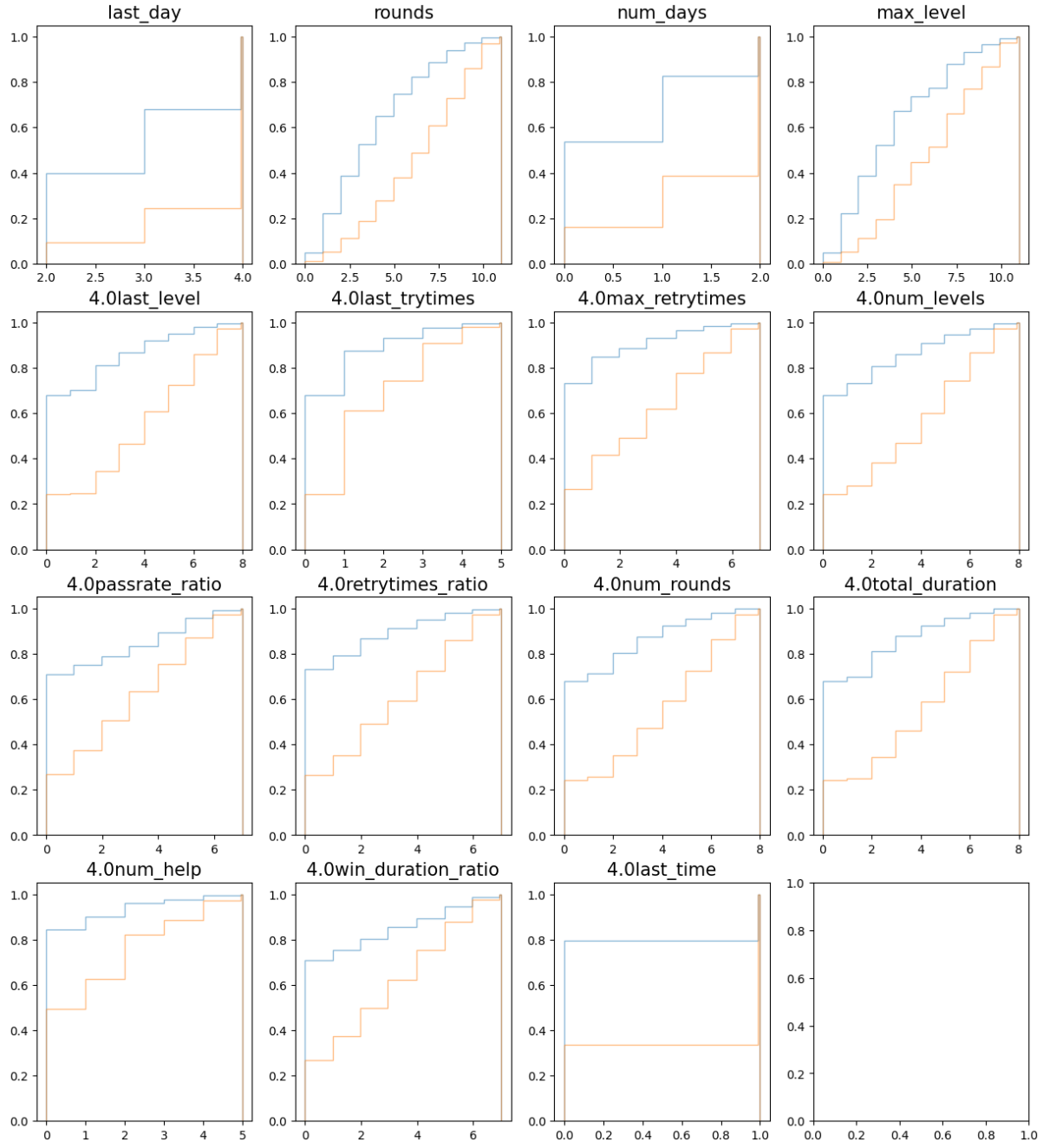


Figure 3. 等比划分

5.2.3 划分效果对比

对于含有较大极值，分布呈现log状的特征，经过去极值等区间划分后，仍保留其分布特征，但每个区间含有的样本量更平均一些。而经过等比划分后，每个区间含有的样本量更加平均，且有更多分组含有更多的'label'为1即流失的样本。

因此，如果做回归，可能使用方法1离散效果更好，但如果做分类，可能使用方法2离散效果更好。（这只是猜想）

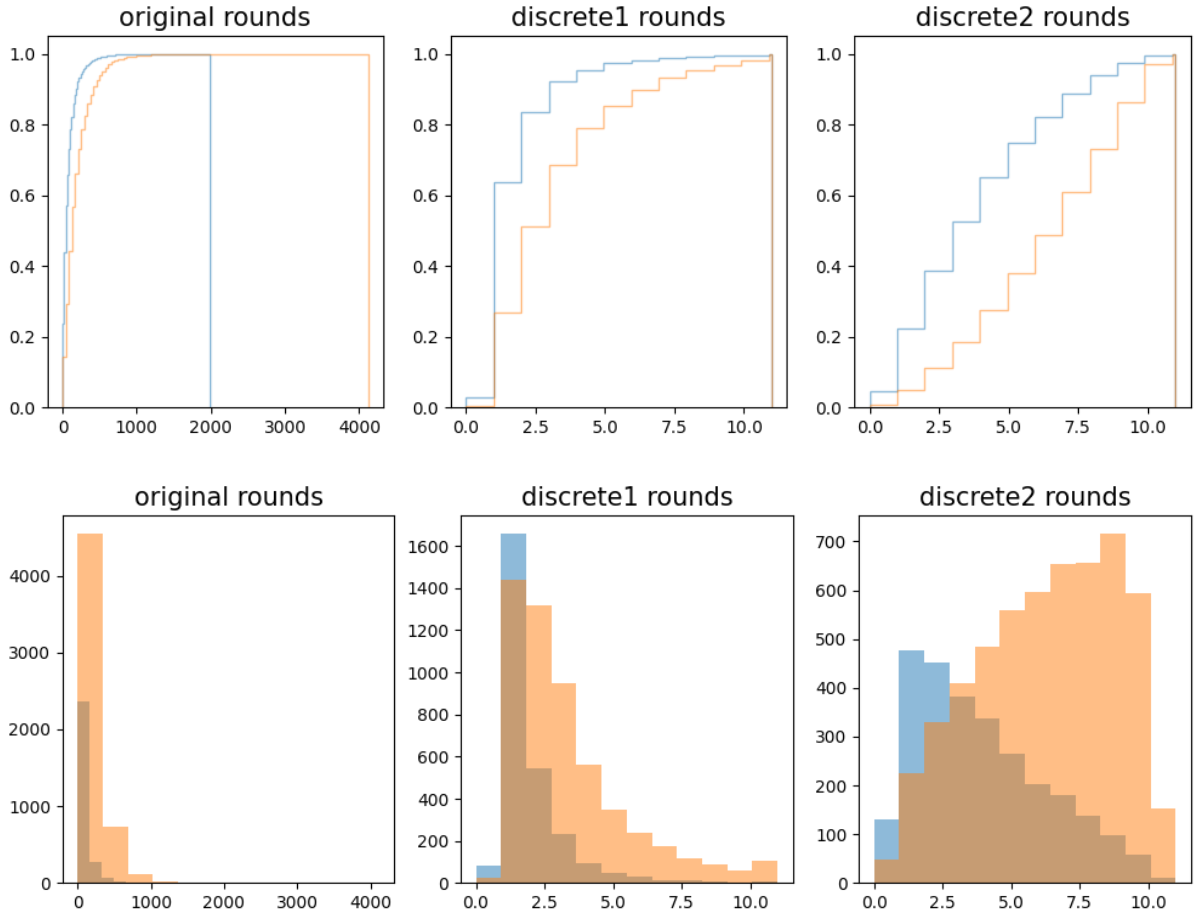


Figure 4. 划分效果对比（原图，等区间，等比）

六、模型选择

6.1 简单模型性能对比

模型是直接调库，或者进行小幅度调优。

	验证集AUC值			
模型	原始数据	MInMax归一化	等区间离散化	等比离散化
DecisionTreeClassifier (max_depth = 4)	0.7284	0.7284	0.7829	0.7887
DecisionTreeRegressor (max_depth = 4)	0.7907	0.7907	0.7829	0.7887
BernoulliNB	0.7248	0.7255	0.7264	0.7264
MultinomialNB	0.6658	0.6470	0.6913	0.6901
ComplementNB	0.6658	0.7128	0.7163	0.7125
LinearSVR	0.2920	0.7164	0.7702	0.7228
MLPClassifier	0.7670	0.7908	0.7676	0.7567

特征归一化能够大幅度提升LinearSVR模型的性能（和向量空间的距离息息相关），小幅度提升ComplementNB的性能。
特征去极值离散化对DecisionTreeClassifier、MultinomialNB、LinearSVR的性能有提升效果，可能是因为去极值离散化弱化了极值对模型的影响。其中等区间离散化在linearSVR中的表现优于等比离散化。

6.2 集成模型调优

6.2.1 XBGClassifier

6.2.1.1 n_estimators

迭代次数，生成树的个数。

```
Fix:
      max_depth = 3,
      min_child_weight = 1,
      gamma = 0,
      subsample = 1,
      colsample_bytree = 1,
      reg_alpha = 0,
      reg_lambda = 1,
      learning_rate = 0.1,

n_estimators :  [1, 2, 16, 64, 128, 256, 512, 1024, 2048]
```

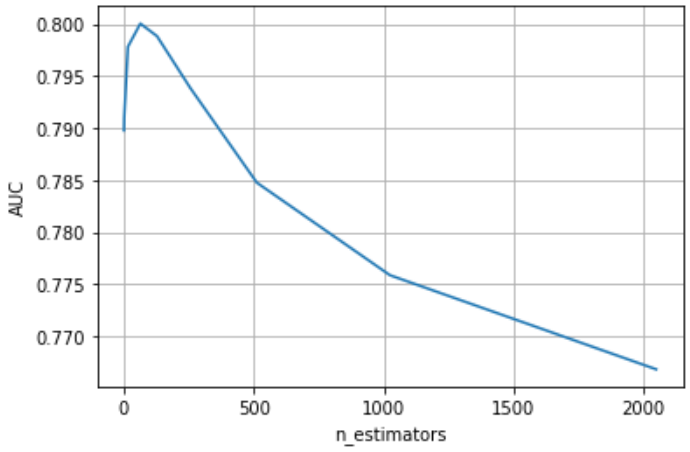


Figure 5. XGB调参n_estimators（大范围）

从Figure 5中可以发现，模型在n_estimators = 64附近达到最优。

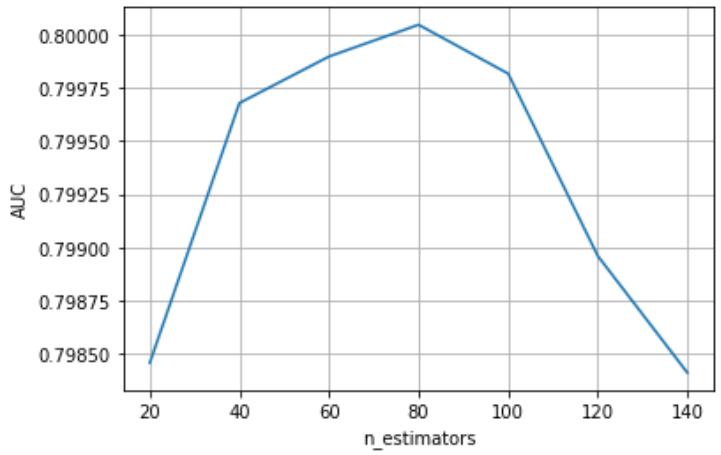


Figure 6. XGB调参n_estimators（小范围）

从Figure 6中可以发现，模型在n_estimators = 80附近达到最优，但是并不显著。没有必要再继续细化

6.2.1.2 max_depth

Maximum depth of a tree. Increasing this value will make the model more complex and more likely to overfit.

Fix:

```
      n_estimators = 80,
      min_child_weight = 1,
      gamma = 0,
      subsample = 1,
      colsample_bytree = 1,
      reg_alpha = 0,
      reg_lambda = 1,
      learning_rate = 0.1,
max_depth :  [1, 2, 3, 4, 5, 6, 7, 10, 15]
```

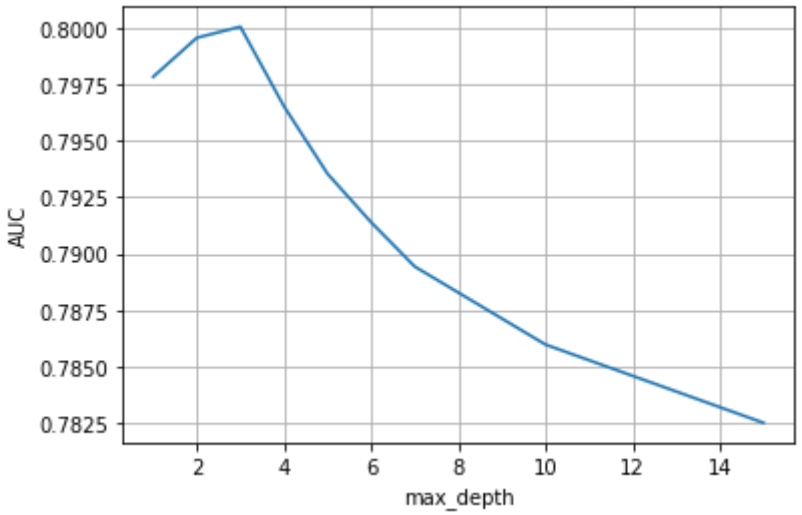


Figure 7. XGB调参max_depth

从Figure 7中可以发现，模型在max_depth = 3时达到最佳。

6.2.1.3 min_child_weight

Minimum sum of instance weight (hessian) needed in a child. If the tree partition step results in a leaf node with the sum of instance weight less than min_child_weight, then the building process will give up further partitioning. In linear regression task, this simply corresponds to minimum number of instances needed to be in each node. The larger min_child_weight is, the more conservative the algorithm will be.

Fix:

```
      n_estimators = 80,
      Max_depth = 3,
      gamma = 0,
      subsample = 1,
      colsample_bytree = 1,
      reg_alpha = 0,
      reg_lambda = 1,
      learning_rate = 0.1,
min_child_weight :  [0.1, 0.5, 1, 2, 3, 4, 5, 6, 10]
```

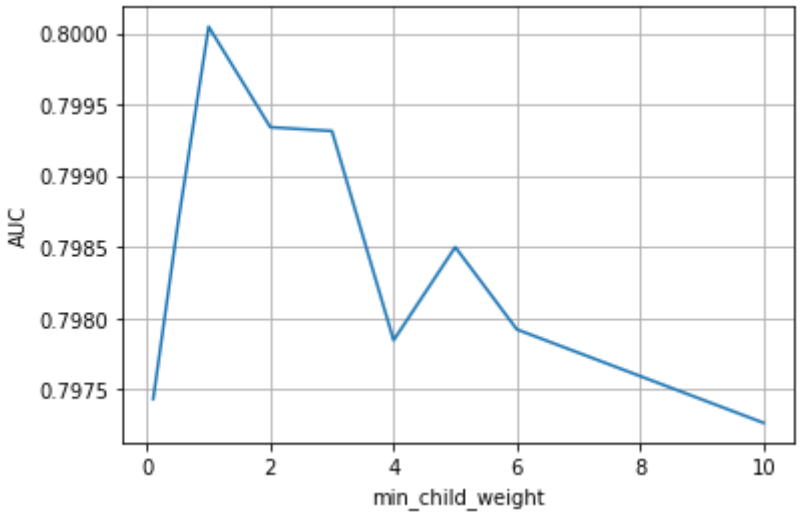


Figure 8. XGB调参min_child_weight

从Figure 8中可以发现，模型在min_child_weight = 1时达到最佳，但结果并不显著，应该是因为max_depth固定为3后，min_child_weight对子叶分化结果的影响已经不大。

6.2.1.4 gamma

Minimum loss reduction required to make a further partition on a leaf node of the tree. The larger gamma is, the more conservative the algorithm will be.

Fix:

```
      n_estimators = 80,
      Max_depth = 3,
      min_child_weight = 1,
      subsample = 1,
      colsample_bytree = 1,
      reg_alpha = 0,
      reg_lambda = 1,
      learning_rate = 0.1
gamma :  [0, 0.1, 0.2, 0.3, 0.4, 1, 2, 3, 4]
```

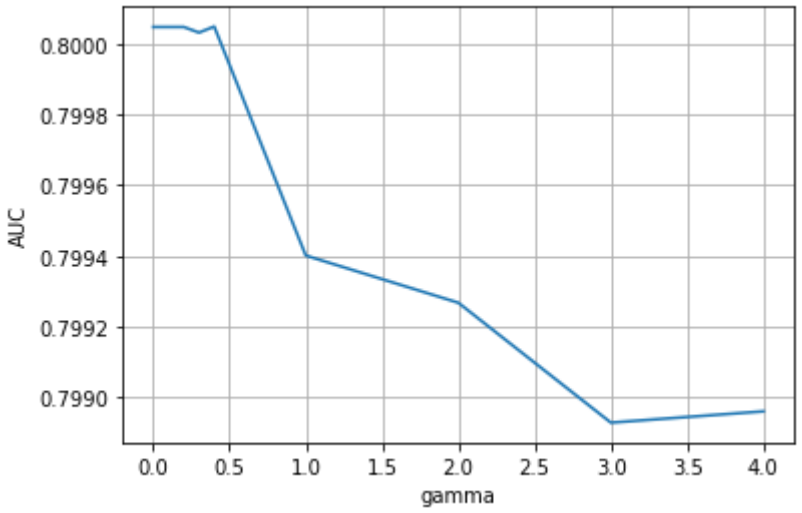


Figure 9. XGB调参gamma

从Figure 9中可以发现，gamma取值在0-0.4区间模型最优。

6.2.1.5 subsample

Subsample ratio of the training instances. Setting it to 0.5 means that XGBoost would randomly sample half of the training data prior to growing trees. and this will prevent overfitting. Subsampling will occur once in every boosting iteration.

每次Boosting的取样大小。

Fix:

```
      n_estimators = 80,
      Max_depth = 3,
      min_child_weight = 1,
      gamma = 0,
      colsample_bytree = 1,
      reg_alpha = 0,
      reg_lambda = 1,
      learning_rate = 0.1,
subsample :  [0.01, 0.2, 0.4, 0.6, 0.7, 0.8, 0.9, 1]
```

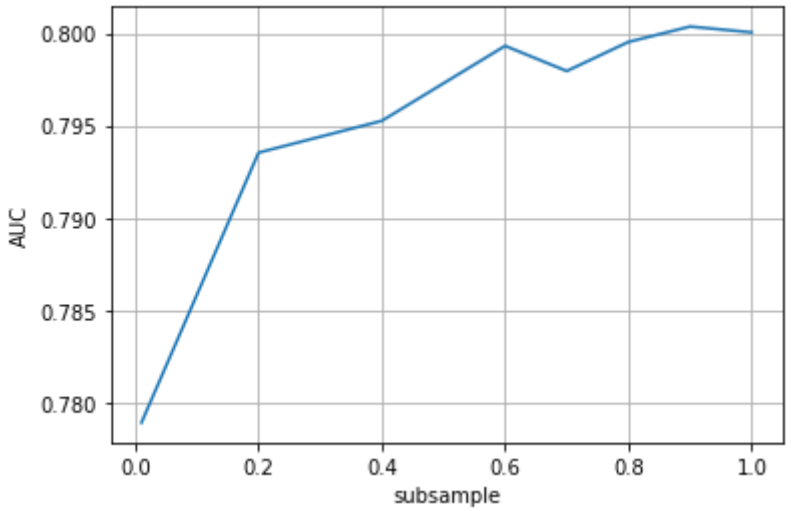


Figure 10. XGB调参subsample

从Figure 10中可以发现，在0到1的区间上，subsample越大，模型越优。可以选用所有sample作为boosting的样本。

6.2.1.6 colsample_bytree

colsample_bytree is the subsample ratio of columns when constructing each tree. Subsampling occurs once for every tree constructed.

每棵树生成的时候，对列的取样。

Fix:

```
n_estimators = 80,
      max_depth = 3,
      min_child_weight = 1,
      gamma = 0,
      subsample = 1,
      reg_alpha = 0,
      reg_lambda = 1,
      learning_rate = 0.1,
colsample_bytree :  [0.6, 0.7, 0.8, 0.9, 1]
```

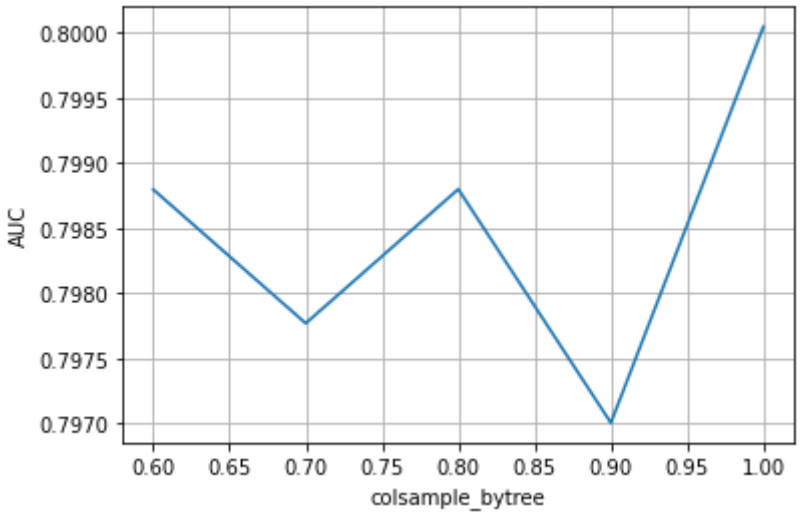


Figure 11. XGB调参colsampel_bytree

从Figure 11可以发现，colsample_bytree对模型效果影响不大。

6.2.1.7 reg_alpha

L1 regularization term on weights. Increasing this value will make model more conservative.

Fix:

```
        n_estimators = 80,
        max_depth = 3,
        min_child_weight = 1,
        gamma = 0,
        subsample = 1,
        colsample_bytree = 1,
        reg_lambda = 1,
        learning_rate = 0.1,
reg_alpha :  [0, 0.05, 0.1, 1, 2, 3, 10, 50, 100, 200]
```

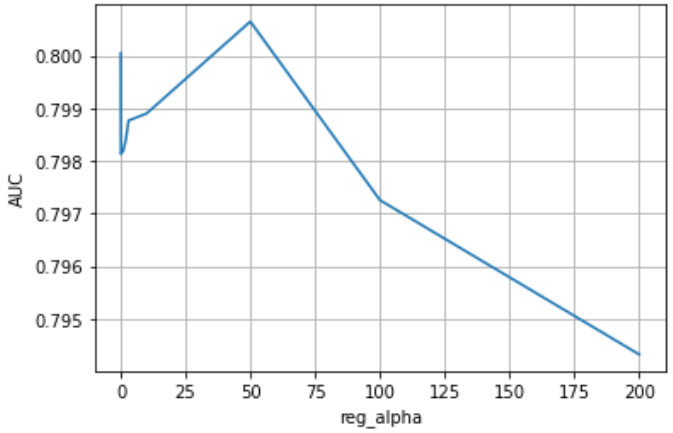


Figure 12. XGB调参reg_alpha

从Figure 12中可以发现随着reg_alpha的增大，model更加保守，但是性能呈现先下降，后提升，再下降的趋势。

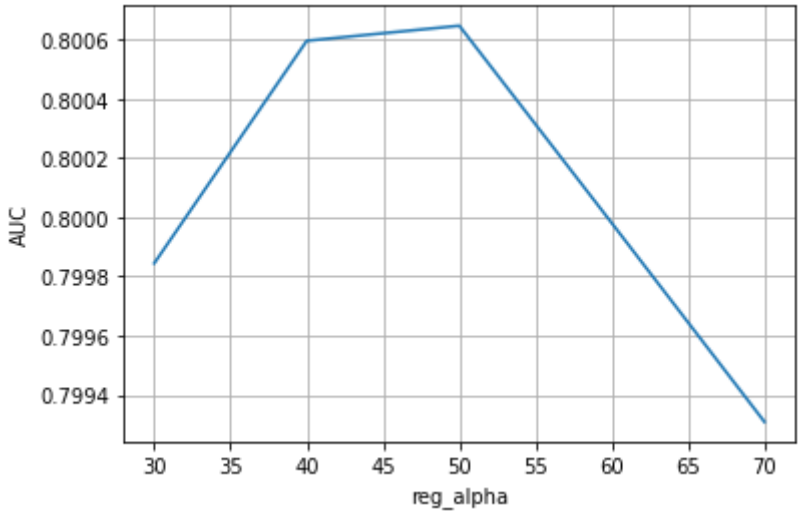


Figure 13. XGB调参reg_alpha（2）

在reg_alpha = 50附近，性能呈现先提升，后下降的趋势，但是并不显著。

6.2.1.8 reg_lambda

L2 regularization term on weights. Increasing this value will make model more conservative.

Fix：

```
      n_estimators = 80,
      max_depth = 3,
      min_child_weight = 1,
      gamma = 0,
      subsample = 1,
      colsample_bytree = 1,
      reg_alpha = 0,
      learning_rate = 0.1,
reg_lambda :  [0.05, 0.1, 1, 2, 3]
```

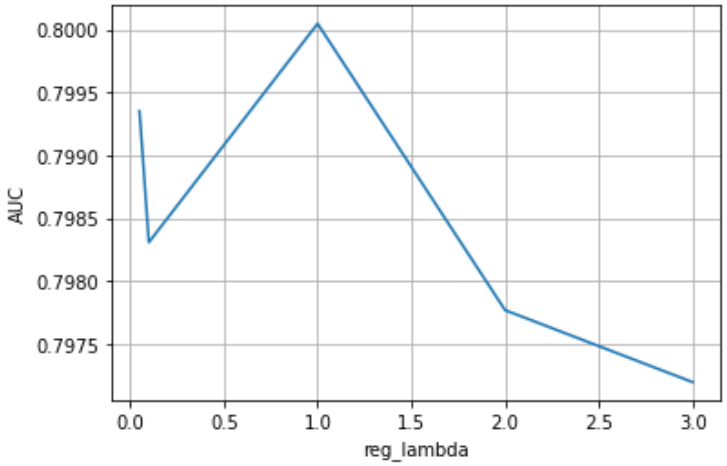


Figure 14. XGB调参reg_lambda

调参效果不显著。

6.2.1.9 eta

即learning rate。

Fix:

```
        n_estimators = 80,
        max_depth = 3,
        min_child_weight = 1,
        gamma = 0,
        subsample = 1,
        colsample_bytree = 1,
        reg_alpha = 0,
        reg_lambda = 1 ,
eta :  [0.001, 0.01, 0.05, 0.1, 0.2, 0.3, 0.5, 1]
```

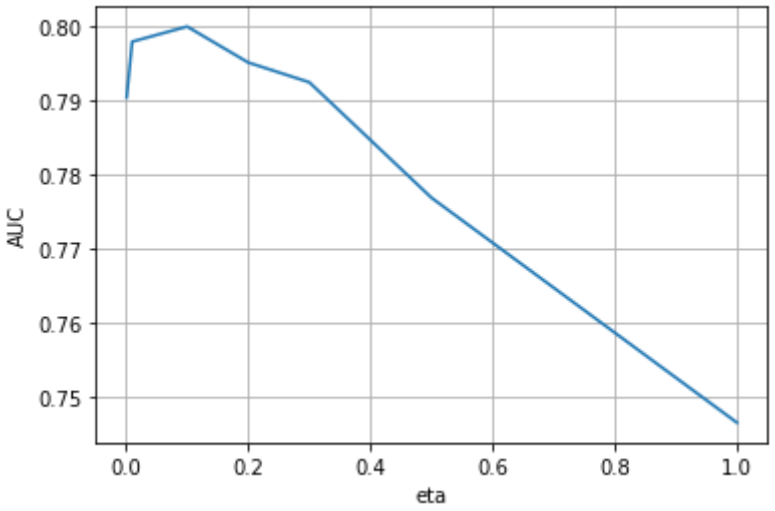


Figure 15. XGB调参eta

随着eta的增长，auc整体呈现下降趋势，步长太大了，可能直接跨过了最优的情况。

6.2.1.10 确定参数

模型参数：

```
n_estimators = 80,
max_depth = 3,
min_child_weight = 1,
gamma = 0,
subsample = 1,
colsample_bytree = 1,
reg_alpha = 0,
reg_lambda = 1 ,
learning_rate = 0.1,
```

由于没有使用grid_search等方法对参数进行多维的分析调整，这里找到的只是在验证集上局部最优的情况，且存在过拟合验证集的可能性。

6.2.1.11 生成模型

使用原始特征值生成的树如下图所示。

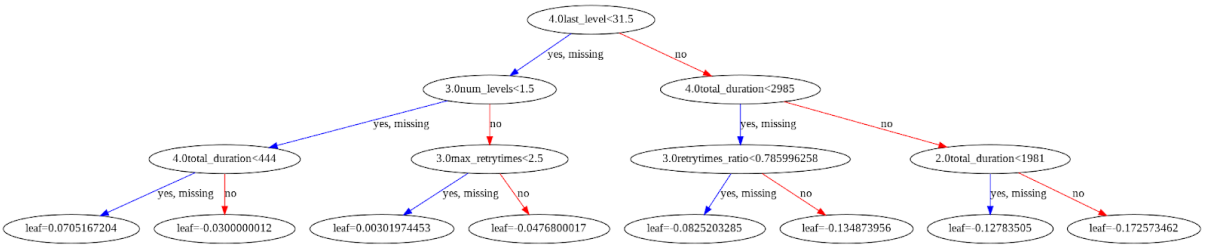


Figure 16. XGBoost生成树

这个模型使用不同方法处理的特征作为输入的效果差异并不大。

	验证集AUC值			
模型	原始数据	MInMax归一化	等区间离散化	等比离散化
XGBClassifier	0.8000	0.8000	0.7943	0.7960

6.2.2 LGBM

具体调参过程就不赘述了。（图片参考Appendix1）最后的参数：

```
boosting_type='gbdt',
                        objective = 'binary',
                        metric = 'auc',
                        verbose = 0,
                        max_depth = 4,
                        learning_rate = 0.02,
                        num_leaves = 35,
                        feature_fraction=0.5,
                        bagging_fraction= 0.7,
                        bagging_freq= 10,
                        lambda_l1= 0.4,
                        lambda_l2= 10,
                        cat_smooth = 10,
                        class_weight = 'balanced'
```

	验证集AUC值			
模型	原始数据	MInMax归一化	等区间离散化	等比离散化
LGBMClassifier	0.8016	0.8016	0.7979	0.8000

其实很多参数调整的效果都并不显著，导致的结果就是过拟合验证集了，所以效果不是很理想。

6.3 RNN模型

使用pytorch搭建RNN模型。

6.3.1 DataLoaders

以训练集为例：

读取dataframe中的数据，构成矩阵

```
Day1_X = np.hstack((train_feature_df[Day1_feature_names].values,
np.zeros((8158, 1)))) #补上一列 #第一天没有‘last_time’列
Day2_X = train_feature_df[Day2_feature_names].values
Day3_X = train_feature_df[Day3_feature_names].values
Day4_X = train_feature_df[Day4_feature_names].values
X_train = np.array([Day1_X, Day2_X, Day3_X, Day4_X])
y_train = np.array(train_feature_df['label'])
Day1_X.shape, X_train.shape
```

MinMax归一化

```
from sklearn.preprocessing import MinMaxScaler
scaler = MinMaxScaler()
for i in range(4):
  X_train[i] = scaler.fit_transform(X_train[i])
  X_dev[i] = scaler.transform(X_dev[i])
  X_test[i] = scaler.transform(X_test[i])
```

Dataset

```
class DayFeatureDataset(Dataset): #train and dev sets
  def __init__(self, is_train_set = True):
    self.X = X_train if is_train_set else X_dev
    self.X = torch.tensor(self.X).type(torch.float32)
    self.y = y_train if is_train_set else y_dev
    self.y = torch.LongTensor(self.y)
    # self.one_hot = torch.nn.functional.one_hot(self.y, 2)
    self.len = self.X.shape[1]

  def __getitem__(self, index):
    return self.X[:, index, :], self.y[index]

  def __len__(self):
```

```
    return self.len

class DayFeatureDataset_test(Dataset): #test
  def __init__(self):
    self.X = X_test
    self.X = torch.tensor(self.X).type(torch.float32)
    self.len = self.X.shape[1]

  def __getitem__(self, index):
    return self.X[:, index, :]

  def __len__(self):
    return self.len

trainset = DayFeatureDataset(is_train_set = True)
trainloader = DataLoader(trainset, batch_size = BATCH_SIZE, shuffle =
True)
```

6.3.2 RNN分类器

详细见PytorchRNN.ipynb。

同时考虑模型的效果和速度，选用的是GRU层，将hidden layer通过线性层转化为输出层。

output是0和1的概率。

这里列出主要代码：

```
class RNNClassifier(torch.nn.Module):
  def __init__(self, input_size, hidden_size, output_size, n_layers=1,
bidirectional=False):
    super(RNNClassifier, self).__init__()
    self.hidden_size = hidden_size
    self.n_layers = n_layers
    self.n_directions = 2 if bidirectional else 1
    self.gru = torch.nn.GRU(input_size, hidden_size, n_layers,
                            bidirectional=bidirectional)
    self.fc = torch.nn.Linear(hidden_size * self.n_directions,
output_size)

  def _init_hidden(self, batch_size):
    hidden = torch.zeros(self.n_layers * self.n_directions,
```

```
                            batch_size, self.hidden_size)
    return create_tensor(hidden)

  def forward(self, input, seq_lengths):
    # input shape : B x S -> S x B
    input = input.transpose(0, 1)
    batch_size = input.shape[1]
    # print(batch_size)

    hidden = self._init_hidden(batch_size)

    output, hidden = self.gru(input, hidden)

    if self.n_directions == 2:
      hidden_cat = torch.cat([hidden[-1], hidden[-2]], dim=1)
    else:
      hidden_cat = hidden[-1]
    fc_output = self.fc(hidden_cat)
    return fc_output

def trainModel():
  total_loss = 0
  for i, (features, labels) in enumerate(trainloader, 1):
    inputs, seq_lengths, target = make_tensors(features, labels)
    output = classifier(inputs, seq_lengths)
    loss = criterion(output, target)
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()

    total_loss += loss.item()
    if i % 10 == 0:
      print(f'[{time_since(start)}] Epoch {epoch} ', end='')
      print(f'[{i * len(inputs)}/{len(trainset)}] ', end='')
      print(f'loss={total_loss / (i * len(inputs))}')
  return total_loss

def devModel():
  correct = 0
```

```
  total = len(devset)
  print("evaluating trained model ...")
  with torch.no_grad():
    for i, (features, labels) in enumerate(devloader, 1):
      inputs, seq_lengths, target = make_tensors(features, labels)
      output = classifier(inputs, seq_lengths)
      pred = output.max(dim=1, keepdim=True)[1]
      correct += pred.eq(target.view_as(pred)).sum().item()

  percent = '%.2f' % (100 * correct / total)
  print(f'Dev set: Accuracy {correct}/{total} {percent}%')

  return correct / total

def devModel_auc():
  correct = 0
  proba = []
  total = len(devset)
  print("evaluating trained model ...")
  with torch.no_grad():
    for i, (features, labels) in enumerate(devloader, 1):
      inputs, seq_lengths, target = make_tensors(features, labels)
      output = classifier(inputs, seq_lengths)
      pred = output.max(dim=1, keepdim=True)[1]
      proba_i = proba1(output)
      proba.append(proba_i)
      correct += pred.eq(target.view_as(pred)).sum().item()

  percent = '%.2f' % (100 * correct / total)
  print(f'Dev set: Accuracy {correct}/{total} {percent}%')

  probas = proba[0]
  for i in range(1, len(proba)):
    probas = torch.cat((probas, proba[i]))
  probas = np.array(probas).reshape(-1, 1)
  AUC = roc_auc_score(y_dev, probas)
  print("Dec set: AUC ", AUC )

  return correct / total, AUC
```

6.3.3 RNN调参

参数的选择以稳定的高AUC为标准，同时兼顾训练效率。

主要参数：

```
HIDDEN_SIZE
BATCH_SIZE
N_LAYER
N_EPOCHS
bidirectional
lr
```

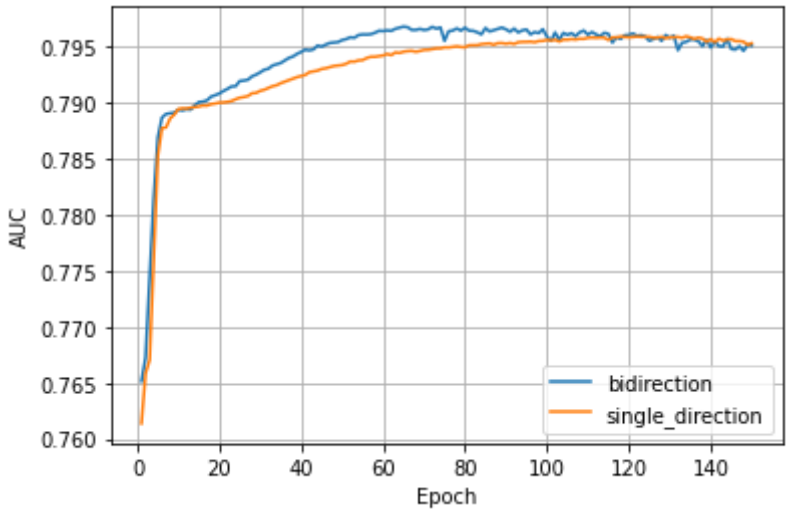


Figure 17. RNN_bidirectrional

Birectional在epoch小于100时，模型效果比单向略好一些。

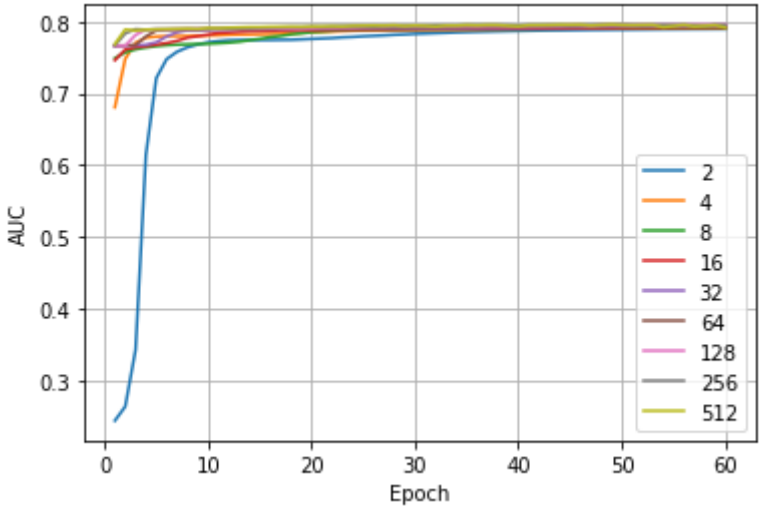


Figure 18. RNN HIDDEN_SIZE

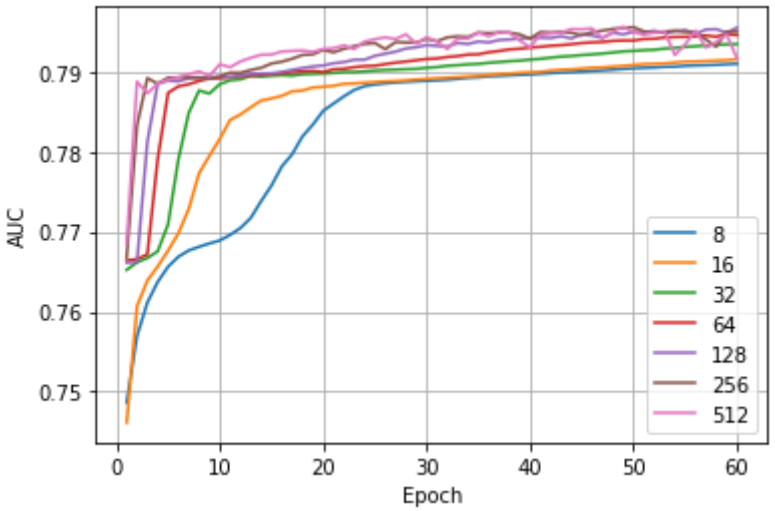


Figure 19. RNN HIDDEN_SIZE(2)

想要学得快一点，可以选用更多的hidden_size，但是后期抖动会比较大。

选择64, 震荡比较少，比较稳定。

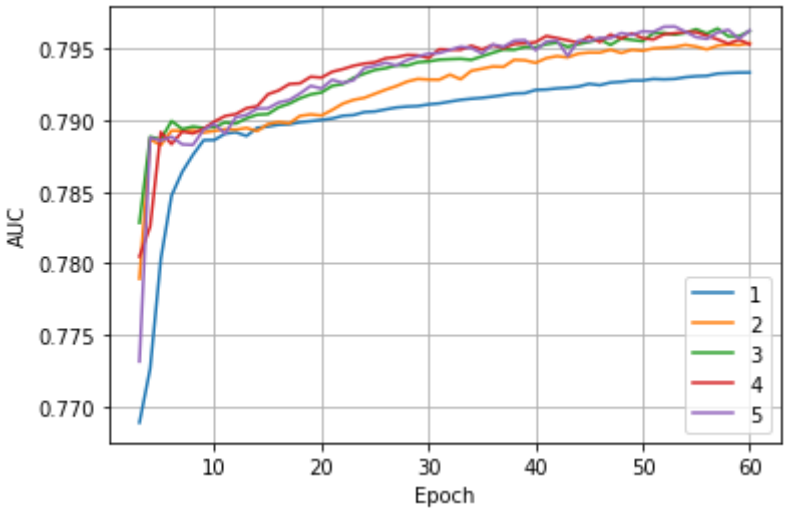


Figure 20. N_LAYERS

选择3层的模型。

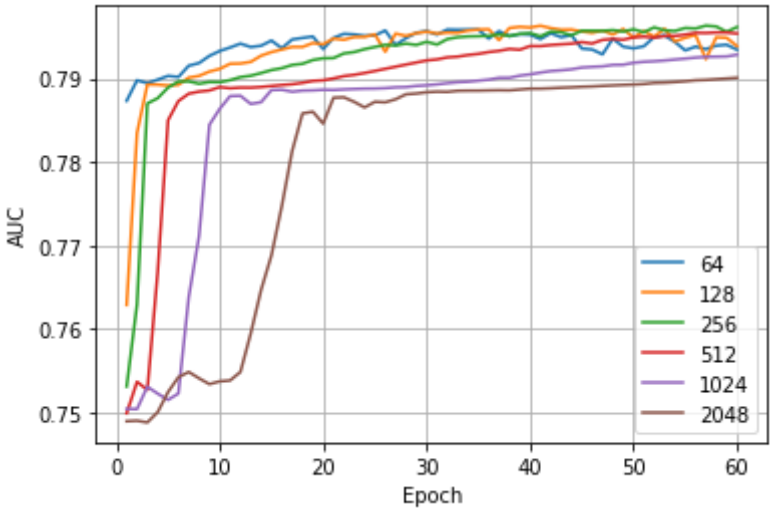


Figure 21. RNN BATCH_SIZE

BATCH_SIZE 选择256。

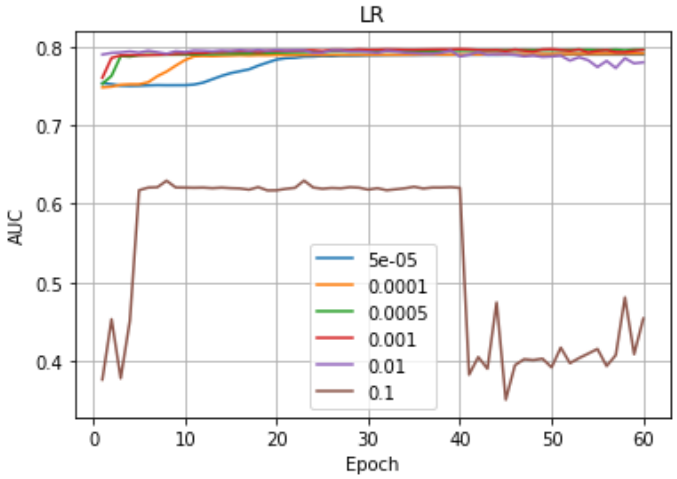


Figure 22. RNN LR

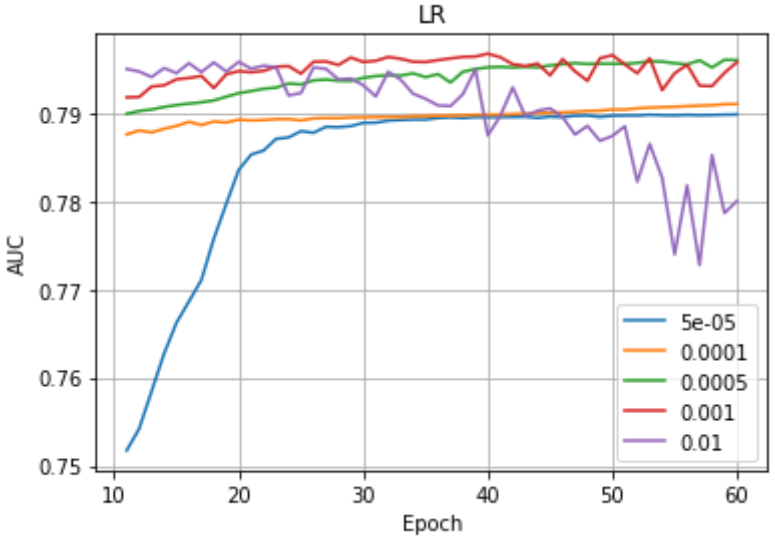


Figure 23. RNN LR(2)

Learning Rate (LR）选择0.0005.

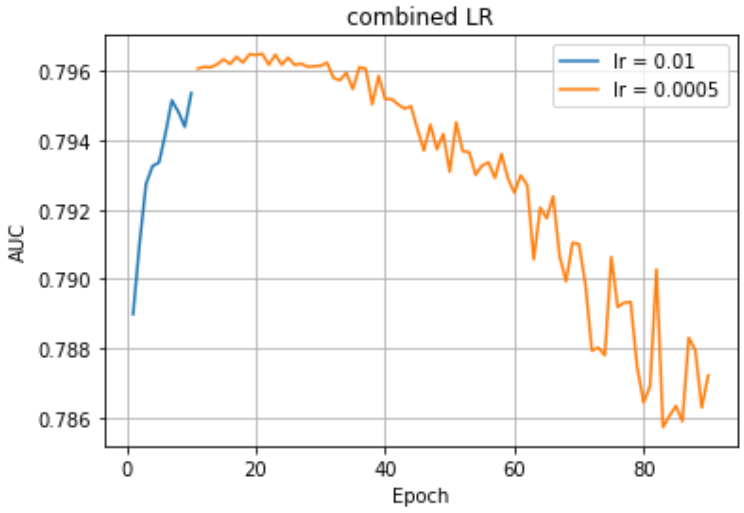


Figure 24. RNN combined LR

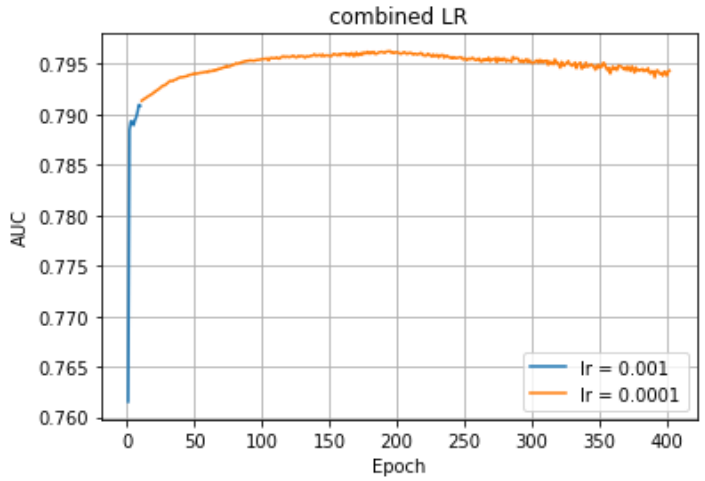


Figure 25. RNN combined LR (2)

先使用0.01的LR，再使用0.0005的LR，在EPOCH > 30后，AUC也有和LR = 0.01时一样的下降趋势。

先使用0.001的LR，再使用0.0001的LR，模型效果较稳定，在EPOCH达到200左右时效果最好。

6.3.4 RNN 模型效果

在验证集上的AUC约为0.79579，测试集上的AUC为0.80038。

由于RNN只使用了每日的特征，没有使用整体特征，所以就没有和其他模型作对比。

七、特征对比与分析

7.1 增加特征*

和吴序同学交流后，又加入了一些特征。其中包括：

整体特征：

'succ_max_level'：最大成功关卡

'total_duration'：总共用时

'succ_total_duration'：成功关卡用时

'mean_reststep'：剩余步数的均值

'succ_mean_reststep'：成功关卡剩余步数的均值

'all_help'：使用help的次数

'succ_all_help'：成功关卡使用help的次数

'passrate'：成功率

'time_ratio'：使用时间和平均时间的比值

'suc_time_ratio'：成功关卡使用时间和平均时间的比值

单日特征：

'sep_time'：当天第一次登录和最后一次登录的时间差

'num_succ'：当天成功过关的关卡数

7.2 单个特征的得分

代码：

```
aucs = []
for i, feature_name in enumerate(feature_names):
  x_train_0 = train_feature_df[feature_name].values.reshape(-1,1)
  x_dev_0 = dev_feature_df[feature_name].values.reshape(-1,1)
  x_test_0 = test_feature_df[feature_name].values.reshape(-1,1)
  model = XGBClassifier(n_estimators = 80, max_depth = 3,
min_child_weight = 1, gamma = 0, subsample = 1, colsample_bytree = 1,
reg_alpha = 0, reg_lambda = 1 , learning_rate = 0.1)
  model.fit(x_train_0, y_train)
  proba = model.predict_proba(x_dev_0)[:, 1]
  auc = roc_auc_score(y_dev, proba)
  aucs.append(auc)
  print("feature name: ", feature_name, ", dev AUC = ", auc)
```

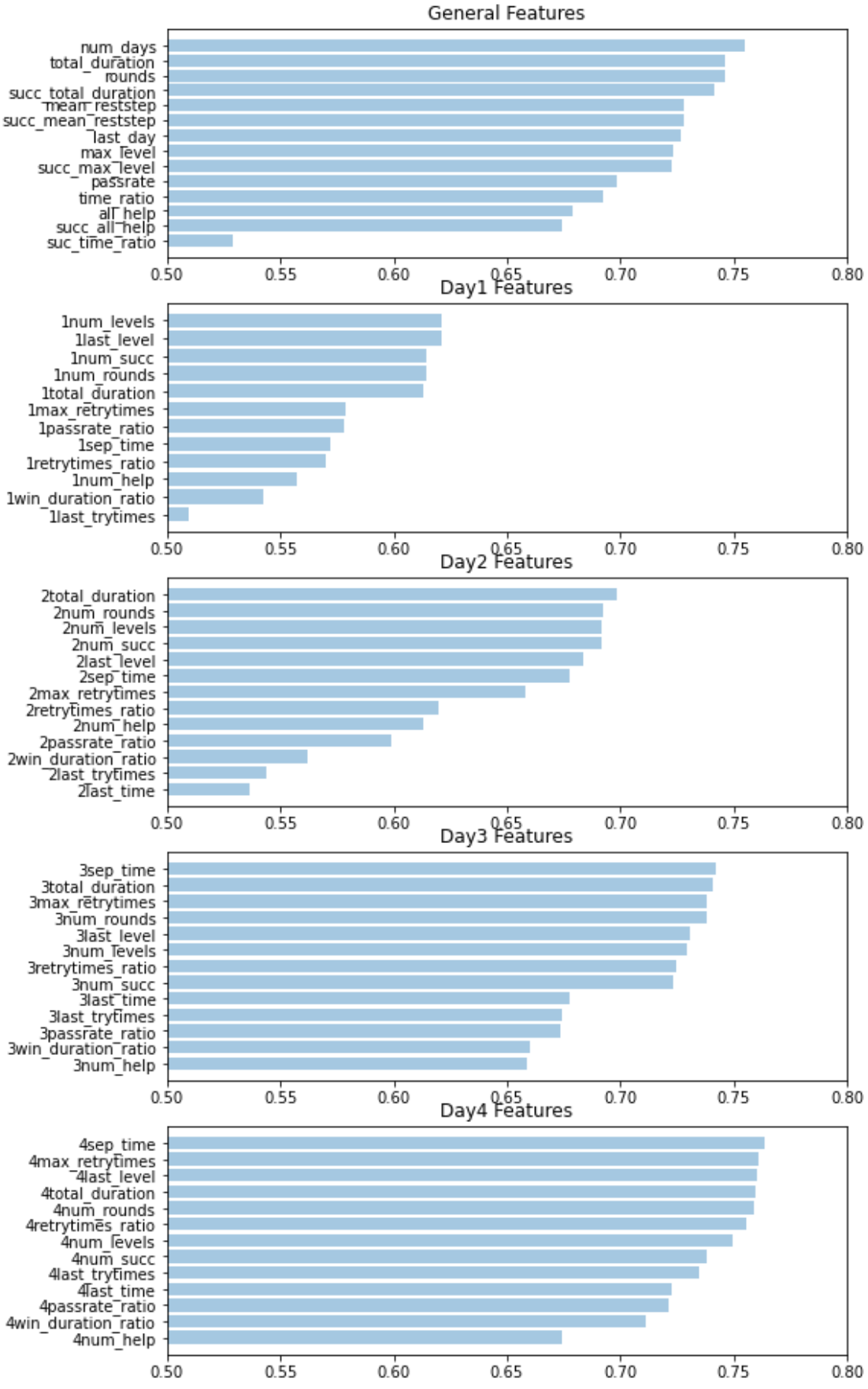


Figure 26. Features and AUC

仅使用单个特征用XGBoost模型进行训练和预测，可以发现，sep_time（每天第一次和最后一次玩的时间间隔）、total_duration（玩的总时长）、last_level（最大的一关）等标志着玩家玩游戏的时长的特征表现更好。
值得注意的是，4max_retrytimes(第四天的最大关卡的尝试次数）也是一个表现不错的特征，它一定程度上代表了玩家的耐心，但具体是正相关还是负相关这里并不能看出来。
随着天数的增长，玩家流失与否和当天的特征相关程度更高。

7.3 主要特征对比

新加入特征后，重新跑了LGBM和XGBoost模型，AUC均有提高。

这里以XGBoost为例分析特征和模型。

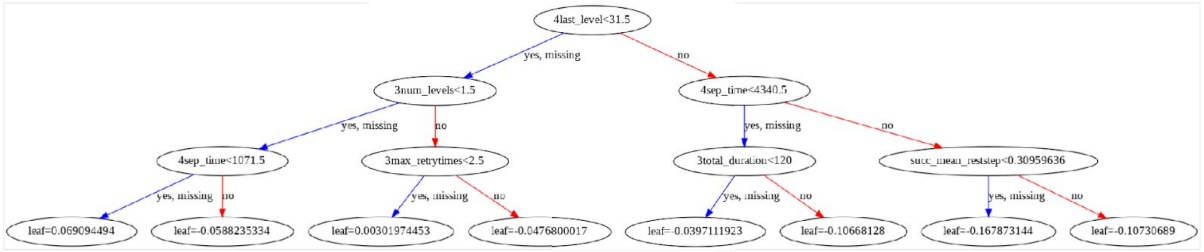


Figure 27. Old Tree

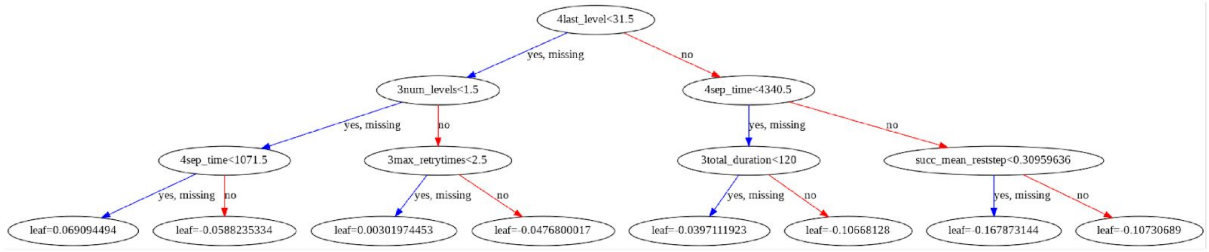


Figure 28. New Tree

老树使用到的特征有：

4last_level; 3num_levels, 4total_duration; 3max_retrytimes, 3retrytimes_ratio;

2total_duration.

层数	特征	特征单项得分	特征得分排名	是否在新树中出现
1	4last_level	0.7602	3	是
2	3num_levels	0.7291	19	是
2	4total_duration	0.7594	4	否
3	4total_duration	0.7594	4	否
3	3max_retrytimes	0.7383	14	是
3	3retrytimes_ratio	0.7247	23	否
3	2total_duration	0.6981	31	否

新树使用的特征有：

4last_level; 3num_levels, 4sep_time; 3max_retrytimes, 3total_duration, succ_mean_reststep.

层数	特征	特征单项得分	特征排名	是否在老树中出现
1	4last_level	0.7602	3	是
2	3num_levels	0.7291	19	是
2	4sep_time	0.7637	1	否（新增特征）
3	4sep_time	0.7637	1	否（新增特征）
3	3max_retrytimes	0.7383	14	是
3	3total_duration	0.7406	13	否
3	succ_mean_reststep	0.7278	21	否（新增特征）

从新老两棵树的结构中可以发现第一层选用的特征两棵树是一样的，last_level（最大尝试关卡， 它可能代表着玩家对这个游戏的兴趣），它不是排名最前面的，但也是排名很靠前的特征。

而重复在第二层和第三层出现的特征，老树中的4total_duration（第四天的游戏总时长）和新树中的4sep_time（第四天第一次登录和最后一次登录的时间差），他们都一定程度上反应了玩家在第四天对游戏的时间投入，（时长和频率），这两个特征的排名都是很靠前的，分别是第四和第一。

除此以外的其他特征并非是排名最靠前的特征，而只能算是中上层次。这可能是因为他们和last_level等已经在结点上的特征的互信息比较小，能为判断玩家是否流失提供更多有用的信息。

其中，3num_levels（第三天尝试的关卡数）, 3max_retrytimes（第三天最大关卡的尝试次数）同时出现在了两棵树中，由此可见，第三天的信息也是很重要的。

综上所述，选取多角度刻画玩家的特征，模型效果会更好。

7.4 Heatmap

让我们来画一张heatmap验证一下：

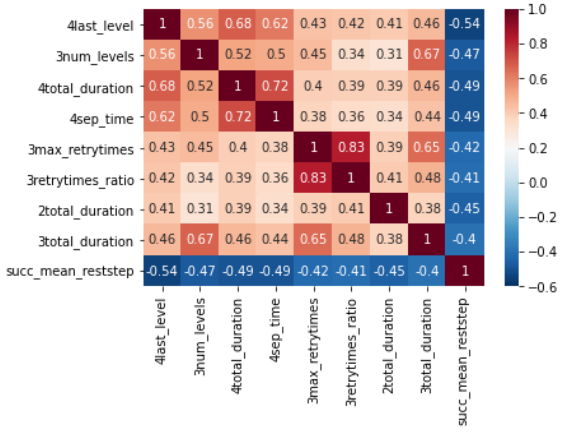


Figure 29. Features used - heatmap

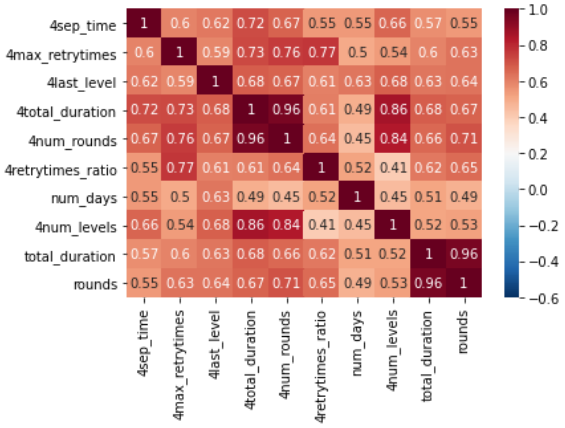


Figure 30. Top 10 features - heatmap

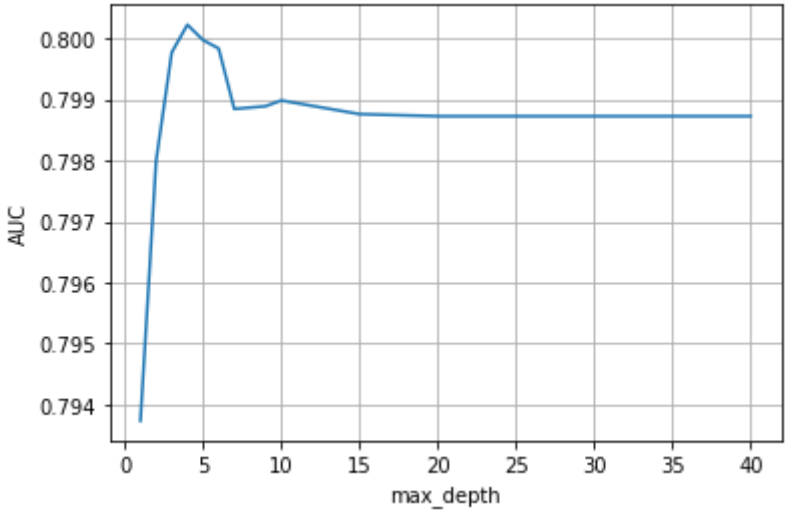
从上图可以发现：

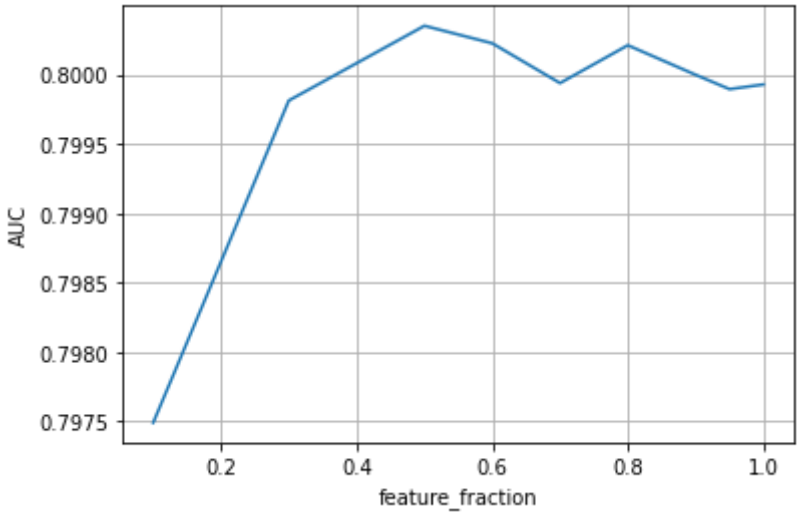
- 模型中选用的特征的互相的correlation值并不高，因此选取的新特征能提供更多信息，提高模型性能。
- 值得注意的是，每个母结点和她的子节点的互信息都不会很高。
- 而top10的feature互相间的关联性就比较大，因此在这里面选取新特征所能提供的信息量着实有限。

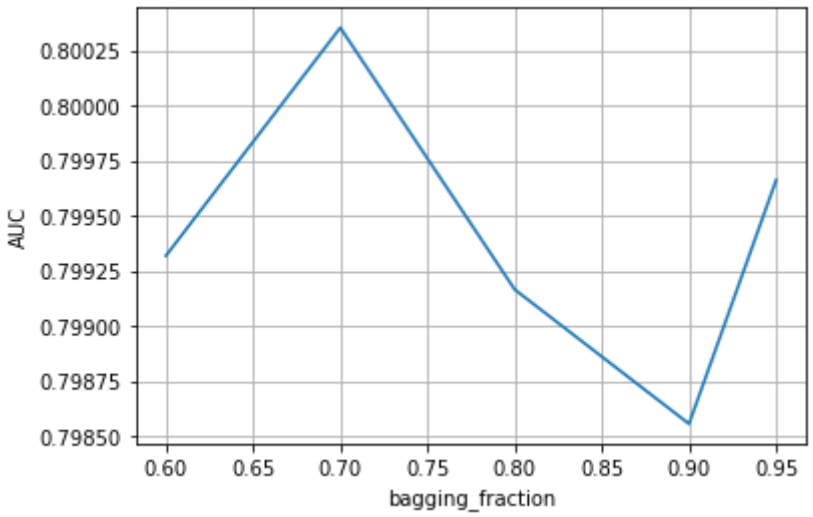
八、未来方向

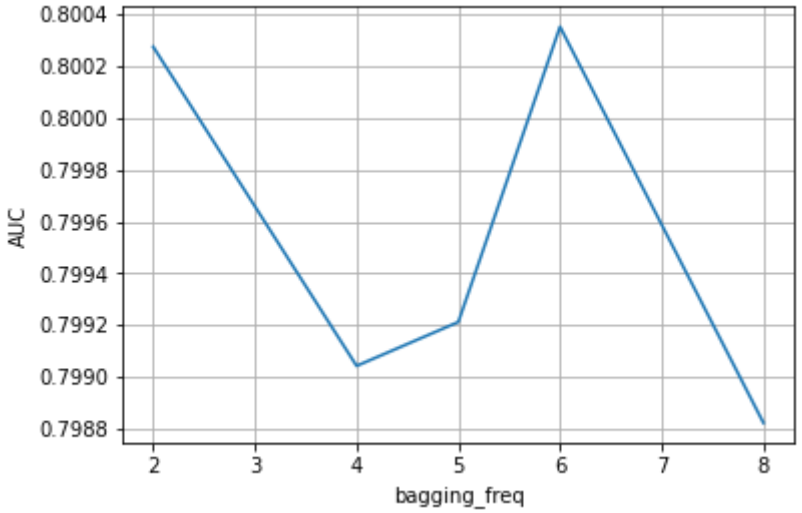
- 数据方面，可以收集更多天的数据，形成更长的数据序列，这样可能更有利于使用RNN模型分析玩家阶段性的特征，判断其流失的可能性。
- 提取特征方面，可以考虑从不同角度出发，有意识地提取不同维度的特征，从各个侧面来刻画玩家的形象。
- 模型选取方面，可以考虑尝试更多模型，例如LSTM等，也可以考虑集成多个模型。
- 模型调优方面，可以考虑grid Search，以多个parameter为变量，动态寻求更优解。但同时也要注意过拟合验证集的情况。

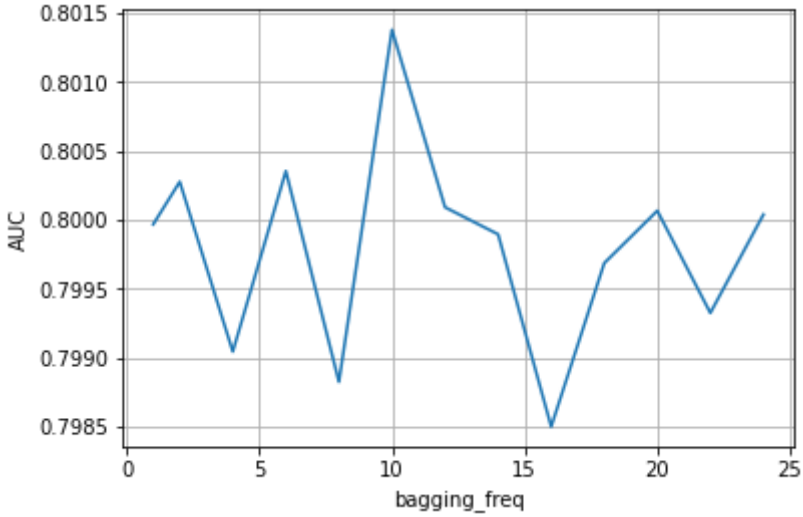
Appendix：LGBM调参图

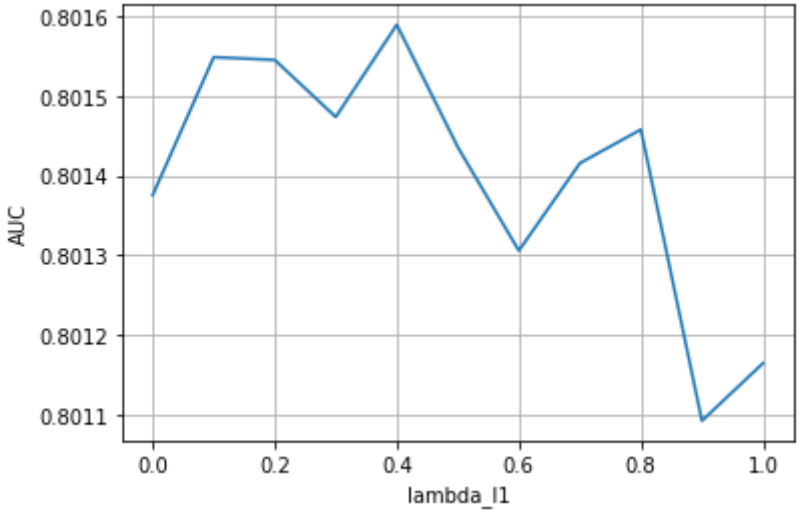


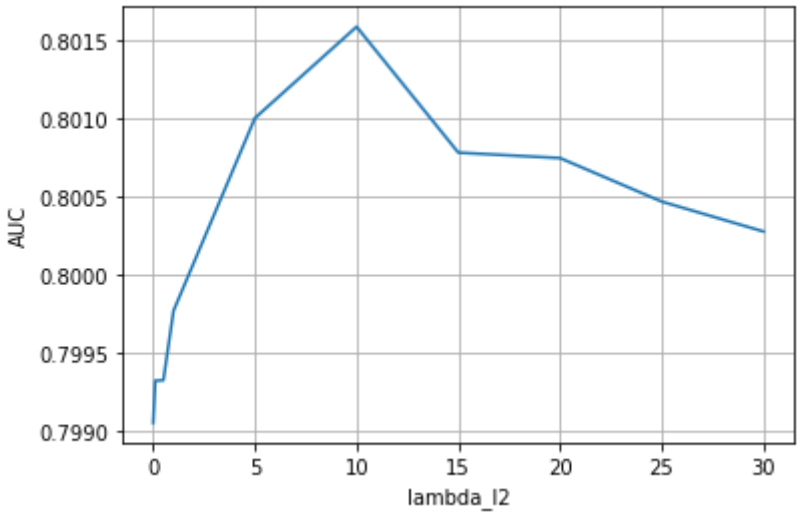


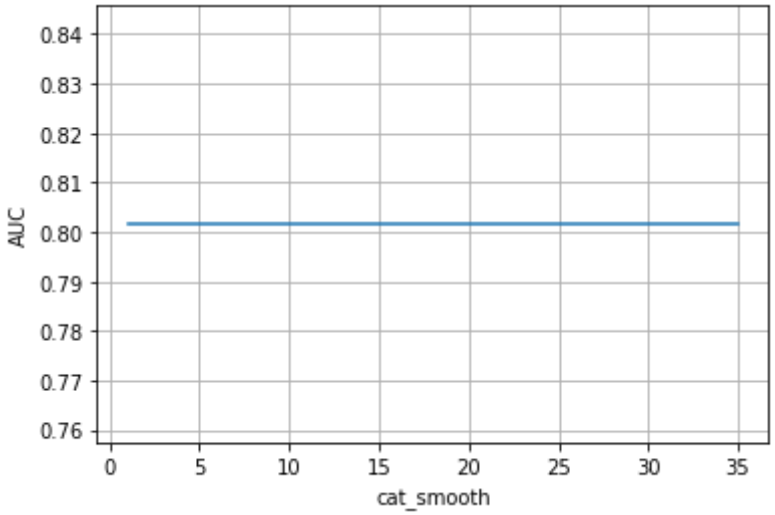


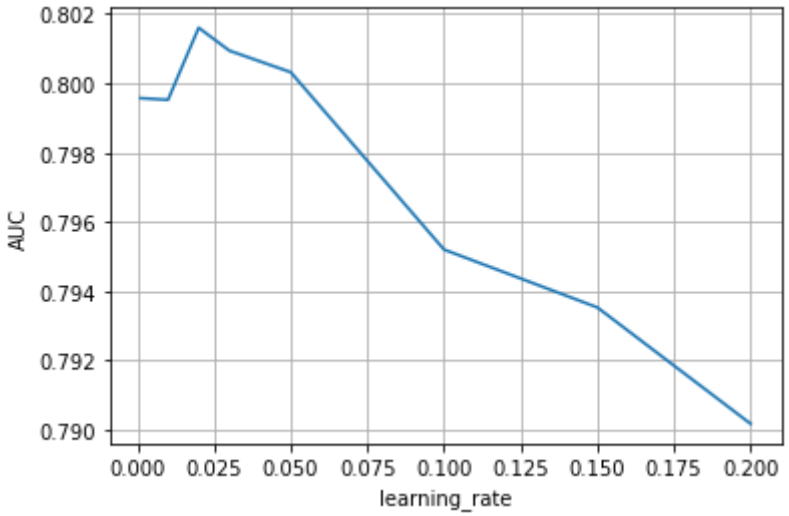

AUC
0.8004
0.8002
0.8000
0.7998
0.7996
0.7994
0.7992
0.7990
0.7988
2
3
4
5
6
7
8
bagging_freq


AUC
0.8015
0.8010
0.8005
0.8000
0.7995
0.7990
0.7985
0
5
10
15
20
25
bagging_freq


AUC
0.8016
0.8015
0.8014
0.8013
0.8012
0.8011
0.0
0.2
0.4
0.6
0.8
1.0
lambda_l1


AUC
0.8015
0.8010
0.8005
0.8000
0.7995
0.7990
0
5
10
15
20
25
30
lambda_l2


AUC
0.84
0.83
0.82
0.81
0.80
0.79
0.78
0.77
0.76
0
5
10
15
20
25
30
35
cat_smooth


AUC
0.802
0.800
0.798
0.796
0.794
0.792
0.790
0.000
0.025
0.050
0.075
0.100
0.125
0.150
0.175
0.200
learning_rate

References:

XGBoost Parameters — xgboost 1.4.0-SNAPSHOT documentation

XGboost数据比赛实战之调参篇(完整流程)_个人文章 - SegmentFault 思否

(3条消息) xgboost子树可视化_故园稻香的博客-CSDN博客

Parameters Tuning — LightGBM 3.1.1.99 documentation