华为 Kpi 异常检测赛

1. 赛题描述

根据网络状态的历史数据 (时间序列数据)，预测未来的网络状态 (二分类)。

2. 数据探索

2.1 训练集

	start_time	value	label	kpi
0	2019/1/27 12:00	96.984	0	SP create bearer context success ratio (%)
1	2019/1/27 13:00	92.361	0	SP create bearer context success ratio (%)
2	2019/1/27 14:00	92.824	0	SP create bearer context success ratio (%)
3	2019/1/27 15:00	96.76	0	SP create bearer context success ratio (%)
4	2019/1/27 16:00	87.29	0	SP create bearer context success ratio (%)

图 1

- start_time 当前时间 datetime64
- kpi 指标对象 object
- value 指标值 float64
- label 网络状态 int64

数据集共包含102个指标对象(kpi)，每个对象包含1032个时间点(2018.12.16 0:00 -- 2019.01.27 23:00 期间的每小时)，共105,264条记录。

2.2 测试集

	start_time	value	kpi
0	2019/1/28 0:00	794	Number of Answered Sessions After Domain Selec...
1	2019/1/28 1:00	400	Number of Answered Sessions After Domain Selec...
2	2019/1/28 2:00	247	Number of Answered Sessions After Domain Selec...
3	2019/1/28 3:00	156	Number of Answered Sessions After Domain Selec...
4	2019/1/28 4:00	120	Number of Answered Sessions After Domain Selec...

图 2

包含与训练集相同的102个指标对象(kpi)，每个对象包含408个时间点(2019.01.28 0:00 -- 2019.02.13 23:00 每小时)，共41,616条记录。缺少需要预测的label字段。

2.3 标签分布

```
df_train.label.mean()
# output:0.00956
```

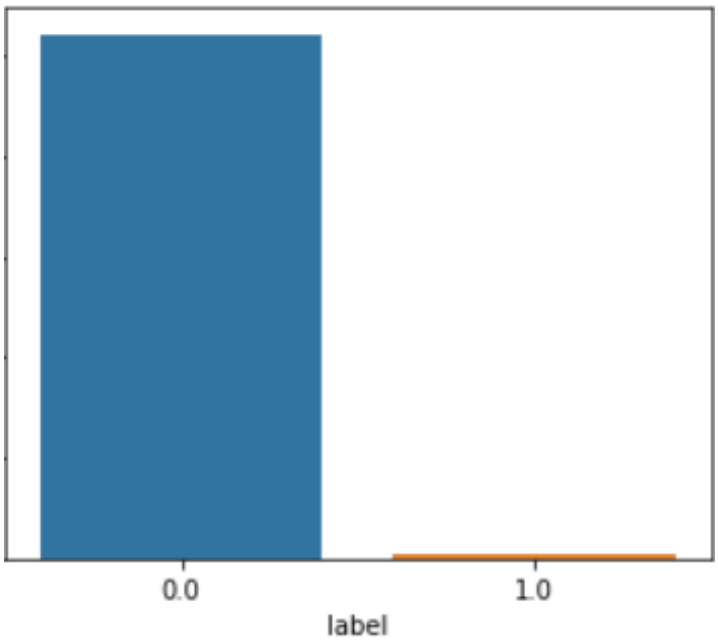

图 3

标签分布极不平衡。

2.4 检查训练集和测试集分布是否一致

```
df_train_temp = df_train[df_train['kpi']=='Number of Answered Sessions After Domain Selection (times)']
df_test_temp = df_test[df_test['kpi']=='Number of Answered Sessions After Domain Selection (times)']

g = sns.distplot(df_train_temp['value'], color="Red",)
g = sns.distplot(df_test_temp['value'], color="Green")
g = g.legend(["train","test"])
plt.show()
```

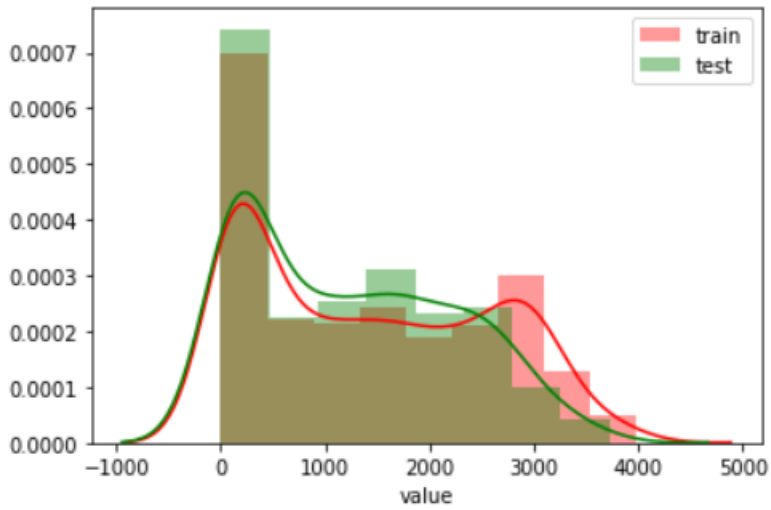

图 4

训练集和测试集数据分布基本一致，因此不需要额外处理

3. 数据预处理

（1）合并训练集和测试集，方便处理

```
df = pd.concat([df_train,df_test])
```

（2）将start_time属性转换为datetime64类型，并提取相关时间特征

```
df['date']=df['start_time'].dt.date
df['week']=df['start_time'].dt.weekday
df['hour']=df['start_time'].dt.hour
df['is_weekend']=(df['week']>5).astype(int)
df['is_night']=((df['hour'] >= 20) | (df['hour'] <= 7)).astype(int) # 是否在夜间
```

（3）按照时间顺序排列数据

```
df.sort_values(['kpi','start_time'], inplace=True)
df.reset_index(drop=True, inplace=True)
```

（4）将value属性从object类型转换为float，发现存在少量缺失值，使用向后填充。

```
df['value']=pd.to_numeric(df['value'],errors='coerce')
df['value']=df['value'].fillna(method='bfill')
```

4. 时间序列可视化

```
for cc in df.kpi.unique():
    dd=df[(df['kpi']==cc)]
    dd.set_index('start_time',inplace=True)
    anomalies=dd[dd['label']==1]

    fig=plt.figure(figsize=(18,3))
    fig=dd['value'].rolling(1).mean().plot()
    fig=anomalies['value'].rolling(1).mean().plot(color='red', marker='o', markersize=10)
    fig=plt.title(cc)
```

对时间序列数据进行可视化之后，发现以下两种类型的时间序列数据

（1）非周期型

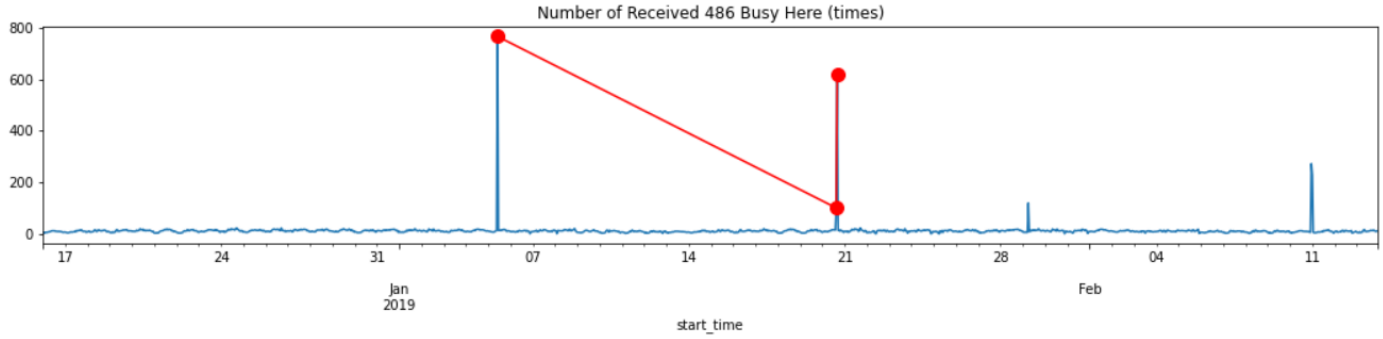

图 5

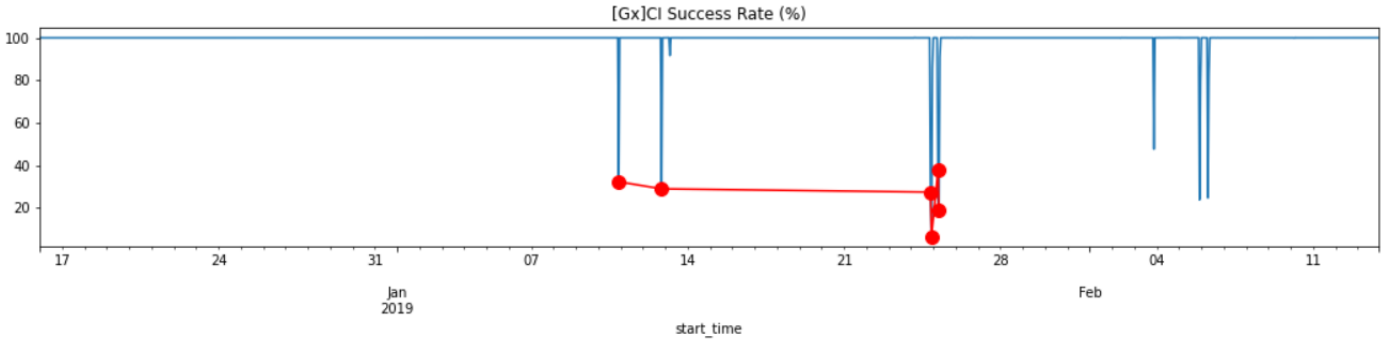

图 6

（2）周期型

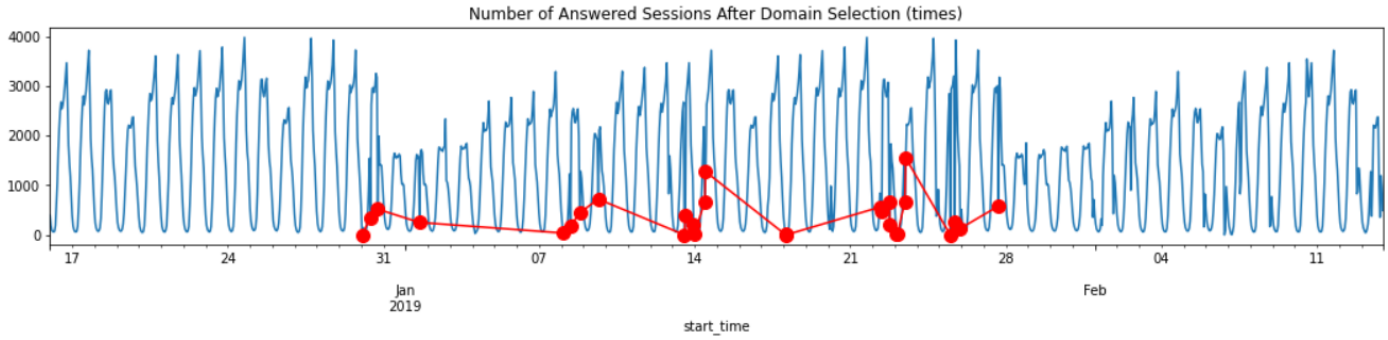

图 7

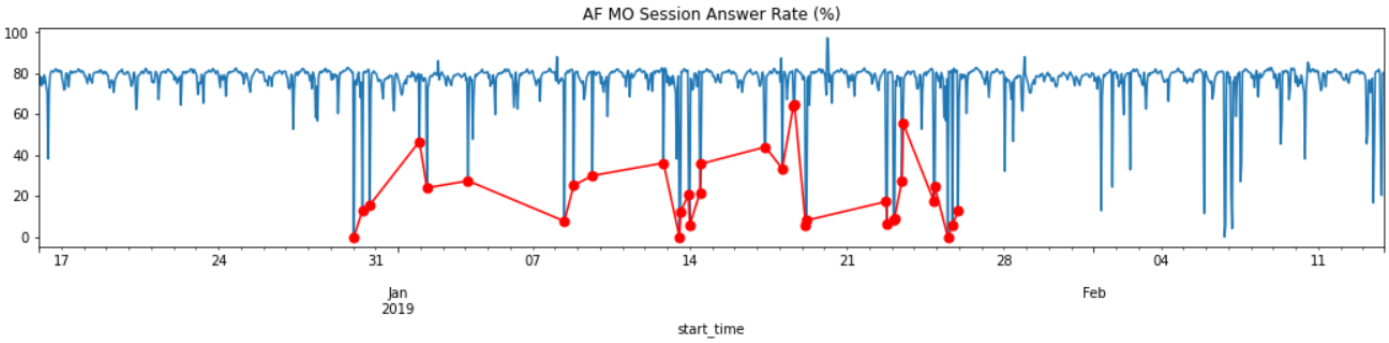

图 8

5. 非周期时间序列

对于少量的非周期时间序列，我们使用统计方法 (箱型图) 来筛选出每个kpi对象的时间序列中的异常值。可以利用训练集调整箱子大小，并对测试集做出预测。

```
def box_plot_outliers(data_ser, box_scale):
    iqr = box_scale * (data_ser.quantile(0.75) - data_ser.quantile(0.25))
    val_low = data_ser.quantile(0.25) - iqr*1.5
    val_up = data_ser.quantile(0.75) + iqr*1.5

    outlier = data_ser[(data_ser < val_low) | (data_ser > val_up)]

    return outlier

from sklearn.metrics import f1_score

for c in uncycle_c_name:
    outlier = box_plot_outliers(df_train_uncycle[df_train_uncycle['kpi']==c]['value'], 3)
    df_train_uncycle.loc[outlier.index.tolist(), 'y_hat'] = 1

score_validate = f1_score(y_true=df_train_uncycle['label'], y_pred=df_train_uncycle['y_hat'], average='binary')
```

在训练集上的F1 Score可以达到0.75142

6. 周期型时间序列

6.1 特征工程

（1）value 值归一化

不同指标对象(kpi)的 value 属性量纲是不一致的，导致它们无法比较。使用z-score方法对value值进行标准化：

$$std\_value = \frac{value - median}{\sigma} \tag{1}$$

不使用线性归一化(MinMaxScaler)，是为了避免数据中存在异常极值对归一化结果产生影响，value减去中值而不是均值也同理。计算公式的分母部分是标准差。

```
df_cycle['std_value'] = 0 # df_cycle是包含周期时间序列对应的数据
for c in cycle_c_name:
    d = df_cycle[df_cycle.kpi==c]
    d['std'] = d.groupby(['date']).value.transform(lambda x :x.std())
    std = d['std'].median()
    median = d['value'].median()

    df_cycle.loc[df_cycle.kpi==c, 'std_value'] =
(df_cycle.loc[df_cycle.kpi==c, 'value'] - median) / std
```

（2）滑动窗口特征

观察时间序列可以发现，异常点经常聚集发生(在邻近的几个小时内)，因此我们统计每个时间点前的滑动窗口特征。具体包括，我们统计每个时间点前 2、3、6、12个小时内的统计信息 (mean、std、skew、kurt)。

```
df_cycle['2_mean_value']= df_cycle.groupby("kpi")
[col].rolling(window=2,min_periods=1).mean().reset_index(0,drop=True)
df_cycle['2_std_value'.format(col)]= df_cycle.groupby("kpi")
[col].rolling(window=2,min_periods=1).std().reset_index(0,drop=True)
df_cycle['2_skew_value'.format(col)]= df_cycle.groupby("kpi")
[col].rolling(window=2,min_periods=1).skew().reset_index(0,drop=True)
df_cycle['2_kurt_value'.format(col)]= df_cycle.groupby("kpi")
[col].rolling(window=2,min_periods=1).kurt().reset_index(0,drop=True)
```

（3）趋势特征

每个时间点 与 “前两小时内的滑窗均值”、“前三小时内的滑窗均值” 的**差值**和**比值**。

```
df_cycle['trend_2']=df_cycle['std_value'] - df_cycle['2_mean_value']
df_cycle['trend_3']=df_cycle['std_value'] - df_cycle['3_mean_value']
```

（4）差分特征

当前时间点的滑动窗口信息 std_value、2_mean_value、3_mean_value 分别与前面对应时刻 (1h、24h、48h、144h(6d)、168h(7d)) 的**差值**和**比值**。

```
col = 'std_value'
for i in [1,24,48,144,168]:
    cc="shift_{}".format(i)
    df_cycle[cc] = df_cycle.groupby('kpi')[col].shift(i)
    df_cycle['x_y_{}_{}'.format(col,i)]=np.abs(df_cycle[col]-df_cycle[cc])
    df_cycle['xy_{}_{}'.format(col,i)]=df_cycle[col]/df_cycle[cc]
    df_cycle.drop(cc,axis=1,inplace=True)
```

（5）分割数据为训练集和测试集，并删除无用的特征

```
df_cycle_train = df_cycle[df_cycle['label'].notnull()]
df_cycle_test = df_cycle[df_cycle['label'].isnull()]

df_cycle_train.drop(['start_time', 'value', 'kpi',
'date'],axis=1,inplace=True)
df_cycle_test.drop(['start_time',  'value', 'kpi', 'date'
,'label'],axis=1,inplace=True)
```

（6）由于训练集中正负样本比例失衡，因此使用SOMTE库对少数类进行过采样

```
train_y = df_cycle_train['label']
train_x = df_cycle_train.drop('label',axis=1)

smo = SMOTENC(random_state=42, categorical_features=[1,2,3,4])
X_smo, y_smo = smo.fit_sample(train_x, train_y)
train_x = X_smo.reset_index(drop=True, inplace=True)
train_y = y_smo.reset_index(drop=True, inplace=True)
```

样本量：105264 -> 208516；正样本(label=1)占比：0.01 -> 0.5

6.2 XGBoost

本题评价指标是F1 Score，XGBoost需要自定义该评价函数。注意，XGBoost自定义评价函数的是用来度量当前损失，而F1 Score 越大表示当前模型效果越好（取值范围[0,1]），因此在自定义函数中需要取反操作。

自定义评价函数代码示例：https://github.com/dmlc/xgboost/tree/master/demo/guide-python

```
def f1_score(pred, data_validate):
    labels = data_validate.get_label()
    score_validate = f1_score(y_true=labels, y_pred=pred, average='binary')

    return 'f1_score', 1-score_validate
```

6.2.1 调参

通常有以下参数可以调整，默认值写在括号中：

1. 集成算法参数：

   num_boost_round；**eta**(0.3)

2. 弱学习器算法

   - **max_depth**(6) 每棵树的最大深度；**min_child_weight**(1) 每个叶节点具有的最小样本权重(hessian)，可以理解为每个叶结点具有的最小样本数
   - **gamma**(0) 树中每个节点进一步分枝所需的最小目标函数下降值
   - **subsample**(1) 从全部样本中采样的比例；**colsample_bytree**(1) 从全部特征中采样的比例
   - **alpha**(0) L1正则化强度；**lambda**(1) L2正则化强度

调参具体步骤如下：

- **num_boost_round**
  设置一个较大的学习步长 eta=0.3，以便快速收敛，寻找此时的最佳的boosting迭代次数

```
train_data = xgb.DMatrix(train_x, train_y)

xgb_params = {
    'booster':'gbtree',
    'objective':'binary:logistic',
    'eta': 0.3
}
cv_result = xgb.cv(xgb_params, train_data, num_boost_round = 1000,
nfold=3, feval = f1_score, early_stopping_rounds=30, verbose_eval=10)
```

	train-error-mean	train-error-std	test-error-mean	test-error-std	train-f1_score-mean	train-f1_score-std	test-f1_score-mean	test-f1_score-std
182	0.000000	0.000000	0.001481	0.000342	0.000000	0.000000	0.001481	0.000350
183	0.000003	0.000004	0.001480	0.000318	0.000003	0.000004	0.001480	0.000325
184	0.000000	0.000000	0.001464	0.000344	0.000000	0.000000	0.001464	0.000351
185	0.000000	0.000000	0.001469	0.000338	0.000000	0.000000	0.001469	0.000345
186	0.000000	0.000000	0.001458	0.000334	0.000000	0.000000	0.001458	0.000341

图 9

最佳迭代次数 cv_result.shape[0] = 187，此时验证集指标为1-F1_Score=0.001458

- **max_depth** & **min_child_weight**

  自定义网格搜索，测试参数范围保持在默认参数附近

```
result_dict = {}
for max_depth in range(4,9):
    for min_child_weight in range(0.5,2,0.5):
        key = "max_depth:%s && min_child_weight:%s" % (str(max_depth), str(min_child_weight))
        print("Current Param:", key)
        params['max_depth'] = max_depth
        params['min_child_weight'] = min_child_weight
        cv_results = lgb.cv(xgb_params, train_data, num_boost_round = 187, nfold=3, feval =
                            f1_score, verbose_eval=10)
        result_dict[key] = cv_result.loc[186,'test-f1_score-mean'].tolist()
```

  最佳参数为 max_depth:7 && min_child_weight:0.5，此时验证集指标为1-F1_Score=0.00138

  在最佳参数附近再次微调这两个参数，可以使得模型有更好的表现。

- **gamma**

  代入之前调整过的参数

```
xgb_params = {
    'booster':'gbtree',
    'objective':'binary:logistic',
    'eta': 0.3,
    'max_depth': 7,
    'min_child_weight': 0.5
}
```

  gamma参数测试范围 [0.2, 0.4, 0.6, 0.8, 1.0]

  最佳参数为 gamma:0.2，此时验证集指标为1-F1_Score=0.00151，大于gamma默认参数0时的表现。将gamma参数的取值向默认参数靠近，尝试[0.1, 0.05]。发现gamma=0.1时，验证集指标为1-F1_Score=0.00136，好于默认参数。

- **subsample** & **colsample_bytree**

  代入之前调整过的参数，并分别测试 subsample 和 colsample_bytree 在[0.2, 0.4, 0.6, 0.8, 1]时的表现。经过尝试 subsample=0.8, colsample_bytree=1时与默认参数表现一致，为了降低模型过拟合风险，设置subsample=0.8，而不是默认值。

- **alpha** & **lambda**

  经过之前的参数调整，弱学习器已经被很好地控制了，不易发生过拟合，两个正则化参数的作用不大。XGBoost默认lambda=1，即使用L2正则化。再尝试加一些L1正则化，当alpha:0.05时，模型得到更好的效果，验证集指标为1-F1_Score=0.00135。

- **eta**

  最后，尝试降低学习率，来提高精度，尝试参数[0.04,0.06,0.08,0.1]。最佳参数为 eta:0.08，此时验证集指标为1-F1_Score=0.00132

调整后参数为

```
xgb_params = {
    'booster':'gbtree',
    'objective':'binary:logistic',
    'eta': 0.08,
    'max_depth': 7,
    'min_child_weight': 0.5,
    'subsample': 0.8,
    'colsample_bytree': 1,
    'gamma': 0.1,
    'alpha': 0.05
}
```

调整参数前，测试集指标为0.00149，调整后降为0.00132，效果很明显。

6.2.2 预测

由于数据是经过过采样处理的，原始数据分布被破坏，在五折划分数据集前已取出少量数据(5%)，用于验证五折交叉模型在未见过的数据集上的表现。

```
from sklearn.model_selection import train_test_split

train, final_vaild = train_test_split(train, test_size=0.05, random_state=42)
```

使用五折交叉划分训练集，训练五个模型，并对验证集和测试集做出预测。注：

```
FOLDS =5
skf = StratifiedKFold(n_splits=FOLDS, shuffle=True, random_state=42)

xgb_y_preds = np.zeros(test_x.shape[0])
xgb_y_oof = np.zeros(train_x.shape[0])
X_test=xgb.DMatrix(test_x)

for tr_idx, val_idx in skf.split(train_x, train_y):
    X_tr, X_vl = train_x.iloc[tr_idx, :], train_x.iloc[val_idx, :]
    y_tr, y_vl = train_y.iloc[tr_idx], train_y.iloc[val_idx]
```

```
11
12      train_data = xgb.DMatrix(X_tr, y_tr)
13      validation_data = xgb.DMatrix(X_vl, y_vl)
14      watchlist = [(validation_data, 'validation_F1')]
15
16      xgb_clf = xgb.train(
17                      xgb_parrams,
18                      train_data,
19                      num_boost_round=1000,
20                      early_stopping_rounds=50,
21                      feval=f1_score,
22                      evals=watchlist,
23                      verbose_eval=10,
24                      #tree_method='gpu_hist',
25                      )
26      X_valid = xgb.DMatrix(X_vl)  # 转为xgb需要的格式
27      xgb_y_oof[val_idx] = xgb_clf.predict(X_valid)
28
29      # 测试集输出
30      xgb_y_preds += xgb_clf.predict(X_test)/FOLDS
31      # 未见过的验证集
32      xgb_final_vaild_preds += xgb_clf.predict(final_vaild_x_xgb)/FOLDS
```

计算 XGBoost 在未见过的验证集上表现：

```
1  y_oof_predict = np.where(xgb_final_vaild_preds>0.5, 1, 0)
2  f1_score(y_true=final_vaild_y, y_pred=y_oof_predict, average='binary')
```

XGboost 对未见过的验证集预测 F1 Score 为 0.81500

6.3 LightGBM

6.3.1 调参

LightGBM 模型需要调整的参数和 XGBoost 模型相似，具体为：

1. 选择较大的 **learning_rate**，确定调参的 **num_iterations**
2. 调整 **max_depth** 和 **num_leaves**，确定树结构
3. 调整 **min_data_in_leaf** 和 **min_sum_hessian_in_leaf**，控制树的分裂
4. 调整 **feature_fraction** 和 **bagging_fraction**，控制构建每棵树使用的样本和特征
5. 调整 **lambda_l1** 和 **lambda_l2**，避免过拟合。
6. 降低 **learning_rate**，提高精度。

最终得到参数

```
lgb_params = {
        'boosting_type': 'gbdt',
        'objective': 'binary',
        'max_depth': 7,
        'num_leaves': 80,
        'learning_rate': 0.07,
        'min_data_in_leaf':25,
        'min_sum_hessian_in_leaf':0.001,
        'feature_fraction': 0.8,
        'bagging_fraction': 0.9,
        'bagging_freq': 3,
        'min_split_gain':0.5
}
```

6.3.2 预测

和XGBoost 一样，使用五折交叉验证划分数据集，并将五个模型对测试集的预测结果取均值。

计算 LightGBM 在未见过的验证集上表现：

```
temp_y_oof = np.where(lgb_final_vaild_preds>0.5, 1, 0)
f1_score(y_true=final_vaild_y, y_pred=temp_y_oof, average='binary')
```

LightGBM 对未见过的验证集预测 F1 Score为 0.80119

6.4 模型融合

对 XGBoost 和 LightGBM 预测结果取平均，并调整分类阈值 threshold

```
merge_oof = 0.5 * xgb_final_vaild_preds + 0.5 * lgb_final_vaild_preds
for threshold in [0.4, 0.45, 0.5, 0.55, 0.6]:
    y_oof_merge = np.where(merge_oof>threshold, 1, 0)
    s = f1_score(y_true=final_vaild_y, y_pred=y_oof_merge, average='binary')
    print("threshold:%s, F1 Score:%s" % (str(threshold), str(s)))
```

当 threshold=0.55 时，模型效果最佳 F1 Score：0.82271，可见模型融合后表现提升。

7. 结果提交

赛题要求按照测试集中的 "start_time" 和 "kpi" 字段的顺序提交结果，我们重新读取测试集，并将对测试集的预测结果合并上去。

```
df_test_sub = data_reference.get_data_reference(dataset="DatasetService",dataset_entity="learning_competition_test_data").to_pandas_dataframe()
df_test_sub = df_test_sub['kpi','start_time']

# 合并 “非周期时间序列”与“周期时间序列”的预测结果
df_preds_all = pd.concat([df_test_uncycle, df_test_cycle_merge])

df_test_sub = pd.merge(df_test_sub, df_preds_all, how='left', on=['kpi', 'start_time'])
```

最终线上预测结果 F1 Score：0.7025，排名：41/725。

8.未来工作

8.1 非周期时间序列

仅使用箱型图寻找 value 字段异常值有一些简单。参考去年比赛的前排方案，可以先为每个时间点提取特征，再使用GMM聚类等无监督算法，往往异常值都是一些离群点。

8.2 周期时间序列

1. 模型在线下的表现为 0.82，而线上仅为0.7，发生了过拟合。之后可以尝试对特征进行筛序，或者利用降维算法减小特征数量。
2. 由于对训练集进行过采样，可能加剧了对训练集的过拟合，可以尝试只使用原始数据集建模。
3. 使用神经网络模型，加强对特征之间的交叉。
4. 参考去年的冠军方案，不同kpi的时间序列之间具有相关性，某些kpi的时间序列的Pearson相关系数甚至能达到0.9，因此我们可以对周期型时间序列进行聚类分组，为每一组单独构建一个模型。