《Kaggle Top 1%方案精讲与实践》

课程说明：

小伙伴好呀~欢迎来到《2021机器学习实战训练营》试学体验课！我是课程主讲老师，九天。

本次体验课为期三天（12月8-10号），期间每晚8点在我的B站直播间公开直播，直播间地址:https://live.bilibili.com/22678166

本期公开课的内容是在上一轮Kaggle竞赛公开课（《Elo Merchant Category Recommendation》）的基础上，进一步探讨如何进一步将排名提升至1%。在接下来的三天内容中，我们将进一步尝试多模型建模与优化、模型融合方法、特征优化与Trick优化等方法，从而在此前的结果基础上大幅提高模型预测准确率。

当然，没有参与上一轮公开课的小伙伴也不用担心，本轮公开课将在开始时我们将快速回顾上一轮公开课内容，并将提供上一轮公开课最终完成的数据处理结果与相关代码，帮助大家无门槛进入本轮课程内容的学习中。当然，如果时间允许，也希望大家能够通过观看上一轮公开课的直播录屏，以巩固相关知识。上一轮公开课直播录屏地址：
https://www.bilibili.com/video/BV1QU4y1u7Ph

课程资料/赛题数据/课程代码/付费课程信息，扫码添加客服“小可爱”回复【kaggle】即可领取哦~

另外，双十二年终大促持续进行中，十八周80+课时体系大课限时七折，扫码咨询小可爱回复“优惠”，还可领取额外折上折优惠，课程主页：https://appze9inzwc2314.pc.xiaoe-tech.com

---

【Kaggle】Elo Merchant Category Recommendation

竞赛案例解析公开课 Part II

Day 5.集成学习与模型融合

In [ ]:
```
import numpy as np
import pandas as pd
import lightgbm as lgb
from sklearn.model_selection import KFold
from hyperopt import hp, fmin, tpe
from numpy.random import RandomState
from sklearn.metrics import mean_squared_error
```

In [ ]:
```
train = pd.read_csv('preprocess/train.csv')
test = pd.read_csv('preprocess/test.csv')
```

一、单模训练策略（二）

Wrapper特征筛选+LightGBM建模+TPE调优

接下来我们进一步尝试Wrapper特征筛选+LightGBM建模+TPE调优的建模策略。相比随机森林这种老牌集成模型，LightGBM则算是Boosting家族中的新秀，LightGBM源自微软旗下的一个项目（Distributed Machine Learning Toolkit （DMKT）），同样是GBDT的一种实现方式，LightGBM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。

首次使用LightGBM时需要对其单独进行安装，可以直接使用pip工具按照如下指令完成安装：

pip install LightGBM

安装完即可按照如下方式进行导入：

In [ ]:
```
import lightgbm as lgb
```

此外，本次建模过程中将使用hyperopt优化器进行超参数搜索，hyperopt优化器也是贝叶斯优化器的一种，可以进行连续变量和离散变量的搜索，目前支持的搜索算法包括随机搜索（random search）、模拟退火（simulated annealing）和TPE（Tree of Parzen Estimator）算法，相比网格搜索，hyperopt效率更快、精度更高。首次使用hyperopt库可使用pip进行安装：

pip install hyperopt

安装完成后按照如下方式进行导入：

In [ ]:
```
from hyperopt import hp, fmin, tpe
```

其中hp是参数空间创建函数，fmin是参数搜索函数，tpe则是一种基于贝叶斯过程的搜索策略。

同时，在本次建模中，我们也将采用wrapper方法进行特征筛选，即根据模型输出结果来进行特征筛选，由于很多时候相关系数并不能很好的衡量特征实际对于标签的重要性，因此wrapper筛选的特征往往更加有效。当然，如果希望我们特征筛选结果更加具有可信度，则可以配合交叉验证过程对其进行筛选。

1.Wrapper特征筛选

接下来是特征筛选过程，此处先择使用Wrapper方法进行特征筛选，通过带入全部数据训练一个LightGBM模型，然后通过观察特征重要性，选取最重要的300个特征。当然，为了进一步确保挑选过程的有效性，此处我们考虑使用交叉验证的方法来进行多轮验证。实际多轮验证特征重要性的过程也较为清晰，我们只需要记录每一轮特征重要性，并在最后进行简单汇总即可。我们可以通过定义如下函数完成该过程：

In [ ]:

```
def feature_select_wrapper(train, test):
    """
    lgm特征重要性筛选函数
    :param train:训练数据集
    :param test:测试数据集
    :return:特征筛选后的训练集和测试集
    """

    # Part 1.划分特征名称，删除ID列和标签列
    print('feature_select_wrapper...')
    label = 'target'
    features = train.columns.tolist()
    features.remove('card_id')
    features.remove('target')

    # Step 2.配置lgb参数
    # 模型参数
    params_initial = {
        'num_leaves': 31,
        'learning_rate': 0.1,
        'boosting': 'gbdt',
        'min_child_samples': 20,
        'bagging_seed': 2020,
        'bagging_fraction': 0.7,
        'bagging_freq': 1,
        'feature_fraction': 0.7,
        'max_depth': -1,
        'metric': 'rmse',
        'reg_alpha': 0,
        'reg_lambda': 1,
        'objective': 'regression'
    }
    # 控制参数
    # 提前验证迭代效果或停止
    ESR = 30
    # 迭代次数
    NBR = 10000
    # 打印间隔
    VBE = 50

    # Part 3.交叉验证过程
    # 实例化评估器
    kf = KFold(n_splits=5, random_state=2020, shuffle=True)
    # 创建空容器
    fse = pd.Series(0, index=features)

    for train_part_index, eval_index in kf.split(train[features], train[label]):
        # 封装训练数据集
        train_part = lgb.Dataset(train[features].loc[train_part_index],
                                 train[label].loc[train_part_index])
        # 封装验证数据集
        eval = lgb.Dataset(train[features].loc[eval_index],
                           train[label].loc[eval_index])
        # 在训练集上进行训练，并同时进行验证
        bst = lgb.train(params_initial, train_part, num_boost_round=NBR,
                        valid_sets=[train_part, eval],
```

```
                          valid_names=['train', 'valid'],
                          early_stopping_rounds=ESR, verbose_eval=VBE)
            # 输出特征重要性计算结果，并进行累加
            fse += pd.Series(bst.feature_importance(), features)

        # Part 4.选择最重要的300个特征
        feature_select = ['card_id'] + fse.sort_values(ascending=False).index.tolist()[:
        print('done')
        return train[feature_select + ['target']], test[feature_select]
```

In [ ]:
```
train_LGBM, test_LGBM = feature_select_wrapper(train, test)
```

查看最终输出结果：

In [ ]:
```
train_LGBM.shape
```

接下来，我们即可带入经过筛选的特征进行建模。

2.LightGBM模型训练与TPE参数优化

接下来，我们进行LightGBM的模型训练过程，和此前的随机森林建模过程类似，我们需要在训练模型的过程同时进行超参数的搜索调优。为了能够更好的借助hyperopt进行超参数搜索，此处我们考虑使用LightGBM的原生算法库进行建模，并将整个算法建模流程封装在若干个函数 内执行。

- 参数回调函数

首先对于lgb模型来说，并不是所有的超参数都需要进行搜索，为了防止多次实例化模型过程中部分超参数被设置成默认参数，此处我们首先需要创建一个参数回调函数，用于在后续多次实例化模型过程中反复申明这部分参数的固定取值：

In [ ]:
```
def params_append(params):
    """
    动态回调参数函数，params视作字典
    :param params:lgb参数字典
    :return params:修正后的lgb参数字典
    """
    params['feature_pre_filter'] = False
    params['objective'] = 'regression'
    params['metric'] = 'rmse'
    params['bagging_seed'] = 2020
    return params
```

- 模型训练与参数优化函数

接下来就是更加复杂的模型训练与超参数调优的的过程。不同于sklearn内部的调参过程，此处由于涉及多个不同的库相互协同，外加本身lgb模型参数就较为复杂，因此整体模型训练与优化过程较为复杂，我们可以通过下述函数来执行该过程：

In [ ]:
```
def param_hyperopt(train):
    """
    模型参数搜索与优化函数
    :param train:训练数据集
    :return params_best:lgb最优参数
    """
    # Part 1.划分特征名称，删除ID列和标签列
```

```
    label = 'target'
    features = train.columns.tolist()
    features.remove('card_id')
    features.remove('target')

    # Part 2.封装训练数据
    train_data = lgb.Dataset(train[features], train[label])

    # Part 3.内部函数，输入模型超参数损失值输出函数
    def hyperopt_objective(params):
        """
        输入超参数，输出对应损失值
        :param params:
        :return:最小rmse
        """
        # 创建参数集
        params = params_append(params)
        print(params)

        # 借助lgb的cv过程，输出某一组超参数下损失值的最小值
        res = lgb.cv(params, train_data, 1000,
                     nfold=2,
                     stratified=False,
                     shuffle=True,
                     metrics='rmse',
                     early_stopping_rounds=20,
                     verbose_eval=False,
                     show_stdv=False,
                     seed=2020)
        return min(res['rmse-mean']) # res是个字典

    # Part 4.lgb超参数空间
    params_space = {
        'learning_rate': hp.uniform('learning_rate', 1e-2, 5e-1),
        'bagging_fraction': hp.uniform('bagging_fraction', 0.5, 1),
        'feature_fraction': hp.uniform('feature_fraction', 0.5, 1),
        'num_leaves': hp.choice('num_leaves', list(range(10, 300, 10))),
        'reg_alpha': hp.randint('reg_alpha', 0, 10),
        'reg_lambda': hp.uniform('reg_lambda', 0, 10),
        'bagging_freq': hp.randint('bagging_freq', 1, 10),
        'min_child_samples': hp.choice('min_child_samples', list(range(1, 30, 5)))
    }

    # Part 5.TPE超参数搜索
    params_best = fmin(
        hyperopt_objective,
        space=params_space,
        algo=tpe.suggest,
        max_evals=30,
        rstate=RandomState(2020))

    # 返回最佳参数
    return params_best
```

接下来我们带入训练数据，测试函数性能：

In [ ]:

```
best_clf = param_hyperopt(train_LGBM)
```

此时best_clf即为lgb模型的最优参数组。

In [ ]:

```
best_clf
```

3.LightGBM模型预测与结果排名

在搜索出最优参数后，接下来即可进行模型预测了。和此前一样，在实际执行预测时有两种思路，其一是单模型预测，即直接针对测试集进行预测并提交结果，其二则是通过交叉验证提交平均得分，并且在此过程中能同时保留下后续用于stacking集成时所需要用到的数据。

- 单模型预测

首先测试单独模型在测试集上的预测效果：

In [ ]:
```
# 再次申明固定参数
best_clf = params_append(best_clf)

# 数据准备过程
label = 'target'
features = train_LGBM.columns.tolist()
features.remove('card_id')
features.remove('target')

# 数据封装
lgb_train = lgb.Dataset(train_LGBM[features], train_LGBM[label])
```

In [ ]:
```
# 在全部数据集上训练模型
bst = lgb.train(best_clf, lgb_train)
```

In [ ]:
```
# 在测试集上完成预测
bst.predict(train_LGBM[features])
```

In [ ]:
```
# 简单查看训练集RMSE
np.sqrt(mean_squared_error(train_LGBM[label], bst.predict(train_LGBM[features])))
```

接下来，对测试集进行预测，并将结果写入本地文件

In [ ]:
```
test_LGBM['target'] = bst.predict(test_LGBM[features])
test_LGBM[['card_id', 'target']].to_csv("result/submission_LGBM.csv", index=False)
```

In [ ]:
```
test_LGBM[['card_id', 'target']].head(5)
```

提交该结果，得到公榜、私榜结果如下：

对比此前的随机森林提交的两组结果，汇总情况如下：

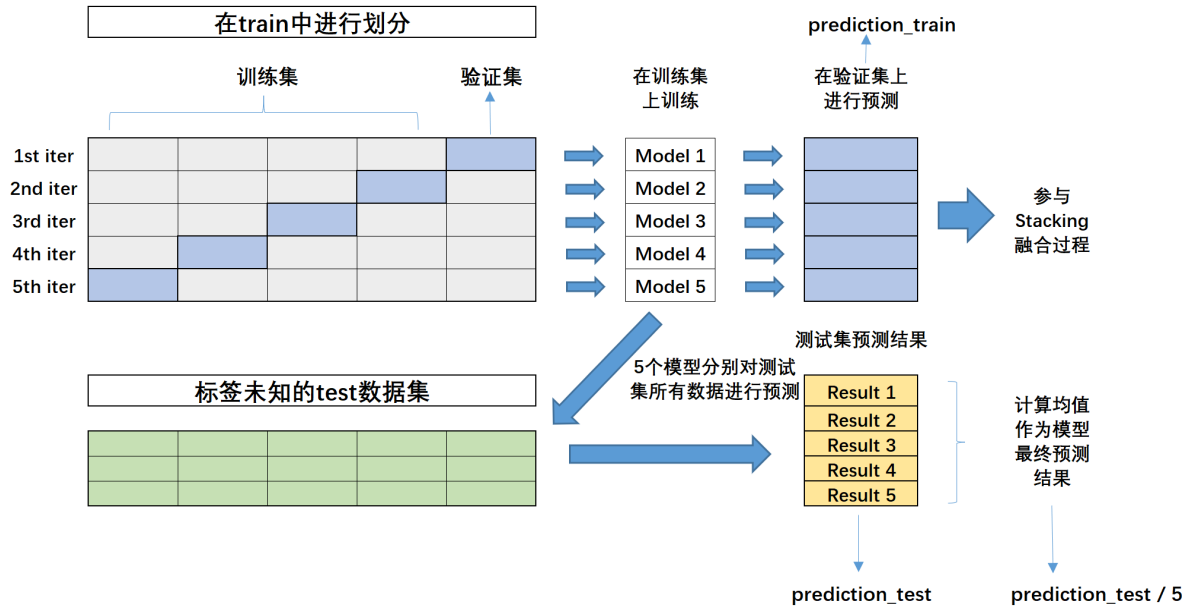
模型	Private Score	Public Score
randomforest	3.65455	3.74969
randomforest+validation	3.65173	3.74954
LightGBM	3.69723	3.80436

能够发现，在单模型预测情况下，lgb要略弱于rf，接下来考虑进行交叉验证，以提高lgb模型预测

效果。

- 结合交叉验证进行模型预测

和随机森林借助交叉验证进行模型预测的过程类似，lgb也需要遵照如下流程进行训练和预测，并同时创建后续集成所需数据集以及预测结果的平均值（作为最终预测结果）



执行过程如下：

In [ ]:

```
def train_predict(train, test, params):
    """

    :param train:
    :param test:
    :param params:
    :return:
    """
    # Part 1.选择特征
    label = 'target'
    features = train.columns.tolist()
    features.remove('card_id')
    features.remove('target')

    # Part 2.再次申明固定参数与控制迭代参数
    params = params_append(params)
    ESR = 30
    NBR = 10000
    VBE = 50

    # Part 3.创建结果存储容器
    # 测试集预测结果存储器，后保存至本地文件
    prediction_test = 0
    # 验证集的模型表现，作为展示用
    cv_score = []
    # 验证集的预测结果存储器，后保存至本地文件
    prediction_train = pd.Series()

    # Part 3.交叉验证
    kf = KFold(n_splits=5, random_state=2020, shuffle=True)
    for train_part_index, eval_index in kf.split(train[features], train[label]):
        # 训练数据封装
        train_part = lgb.Dataset(train[features].loc[train_part_index],
```

```
                                    train[label].loc[train_part_index])
        # 测试数据封装
        eval = lgb.Dataset(train[features].loc[eval_index],
                           train[label].loc[eval_index])
        # 依据验证集训练模型
        bst = lgb.train(params, train_part, num_boost_round=NBR,
                        valid_sets=[train_part, eval],
                        valid_names=['train', 'valid'],
                        early_stopping_rounds=ESR, verbose_eval=VBE)
        # 测试集预测结果并纳入prediction_test容器
        prediction_test += bst.predict(test[features])
        # 验证集预测结果并纳入prediction_train容器
        prediction_train = prediction_train.append(pd.Series(bst.predict(train[featur
                                                             index=eval_index))
        # 验证集预测结果
        eval_pre = bst.predict(train[features].loc[eval_index])
        # 计算验证集上得分
        score = np.sqrt(mean_squared_error(train[label].loc[eval_index].values, eval_
        # 纳入cv_score容器
        cv_score.append(score)

    # Part 4.打印/输出结果
    # 打印验证集得分与平均得分
    print(cv_score, sum(cv_score) / 5)
    # 将验证集上预测结果写入本地文件
    pd.Series(prediction_train.sort_index().values).to_csv("preprocess/train_lightgbm
    # 将测试集上预测结果写入本地文件
    pd.Series(prediction_test / 5).to_csv("preprocess/test_lightgbm.csv", index=False
    # 测试集平均得分作为模型最终预测结果
    test['target'] = prediction_test / 5
    # 将测试集预测结果写成竞赛要求格式并保存至本地
    test[['card_id', 'target']].to_csv("result/submission_lightgbm.csv", index=False)
    return
```

In [ ]:

```
train_LGBM, test_LGBM = feature_select_wrapper(train, test)
best_clf = param_hyperopt(train_LGBM)
train_predict(train_LGBM, test_LGBM, best_clf)
```

接下来即可在竞赛主页提交预测结果。最终公榜私榜评分如下：

对比此前结果：

模型	Private Score	Public Score
randomforest	3.65455	3.74969
randomforest+validation	3.65173	3.74954
LightGBM	3.69723	3.80436
LightGBM+validation	3.64403	3.73875

能够看出，经过交叉验证后输出的平均值结果，较此前的预测评分，有较大提升，这也是目前我们跑出的最好成绩。同时，交叉验证的作用已得到充分征明，后续在进行其他模型训练时仅考虑模型+交叉验证的输出结果，不再进行单模型结果输出。

二、单模训练策略（三）

NLP特征优化+XGBoost建模+贝叶斯优化器

在执行完随机森林与LightGBM后，我们已经对不同集成算法的竞赛建模流程有了一定的了解，大家可以照此思路继续尝试其他集成模型。当然，如果想进一步优化提升模型效果，我们可以考虑围绕数据集中的部分ID字段进行NLP特征优化。因此，接下来，我们考虑采用CountVectorizer, TfidfVectorizer两种方法对数据集中部分特征进行NLP特征衍生，并且采用XGBoost模型进行预测，同时考虑进一步使用另一种贝叶斯优化器（bayes_opt）来进行模型参数调优。

> 当然，此处训练的三个模型分别采用了不同的优化器、甚至是采用了不同的特征衍生的方法，也是为了令这三个模型尽可能的存在一定的差异性，从而为后续的模型融合的融合效果做铺垫。

1.NLP特征优化

首先我们注意到，在数据集中存在大量的ID相关的列（除了card_id外），包括'merchant_id'、'merchant_category_id'、'state_id'、'subsector_id'、'city_id'等，考虑到这些ID在出现频率方面都和用户实际的交易行为息息相关，例如对于单独用户A来说，在其交易记录中频繁出现某商户id（假设为B），则说明该用户A对商户B情有独钟，而如果在不同的用户交易数据中，都频繁的出现了商户B，则说明这家商户受到广泛欢迎，而进一步的说明A的喜好可能和大多数用户一致，而反之则说明A用户的喜好较为独特。为了能够挖掘出类似信息，我们可以考虑采用NLP中CountVector和TF-IDF两种方法来进行进一步特征衍生，其中CountVector可以挖掘类似某用户钟爱某商铺的信息，而TF-IDF则可进一步挖掘出类似某用户的喜好是否普遍或一致等信息。

此外，若要借助NLP方法进行进一步特征衍生，则需要考虑到新创建的特征数量过大所导致的问题，因此我们建议在使用上述方法的同时，考虑借助scipy中的稀疏矩阵相关方法，来进行新特征的存储与读取。即采用CSR格式来创建稀疏矩阵，用npz格式来进行本地数据文件保存。

需要导入的包如下：

In [ ]:
```
from sklearn.feature_extraction.text import CountVectorizer, TfidfVectorizer
from sklearn.preprocessing import OneHotEncoder, LabelEncoder
from scipy import sparse
```

接下来进行NLP特征创建。

In [ ]:
```
# 创建空DataFrame用于保存NLP特征
train_x = pd.DataFrame()
test_x = pd.DataFrame()

# 实例化CountVectorizer评估器与TfidfVectorizer评估器
cntv = CountVectorizer()
tfv = TfidfVectorizer(ngram_range=(1, 2), min_df=3, max_df=0.9, use_idf=1, smooth_idf

# 创建空列表用户保存修正后的列名称
vector_feature =[]
for co in ['merchant_id', 'merchant_category_id', 'state_id', 'subsector_id', 'city_i
    vector_feature.extend([co+'_new', co+'_hist', co+'_all'])

# 提取每一列进行新特征衍生
for feature in vector_feature:
```

```
    print(feature)
    cntv.fit(train[feature].append(test[feature]))
    train_x = sparse.hstack((train_x, cntv.transform(train[feature]))).tocsr()
    test_x = sparse.hstack((test_x, cntv.transform(test[feature]))).tocsr()

    tfv.fit(train[feature].append(test[feature]))
    train_x = sparse.hstack((train_x, tfv.transform(train[feature]))).tocsr()
    test_x = sparse.hstack((test_x, tfv.transform(test[feature]))).tocsr()

# 保存NLP特征衍生结果
sparse.save_npz("preprocess/train_nlp.npz", train_x)
sparse.save_npz("preprocess/test_nlp.npz", test_x)
```

In [ ]:

```
train_x.shape
```

2.XGBoost模型训练与优化

- 数据读取

接下来，来进行XGBoost的模型训练与优化，首先需要导入必要的包以及进行数据读取：

In [ ]:

```
import xgboost as xgb
from sklearn.feature_selection import f_regression
from numpy.random import RandomState
from bayes_opt import BayesianOptimization
```

In [ ]:

```
train = pd.read_csv('preprocess/train.csv')
test = pd.read_csv('preprocess/test.csv')
```

In [ ]:

```
features = train.columns.tolist()
features.remove('card_id')
features.remove('target')

train_x = sparse.load_npz("preprocess/train_nlp.npz")
test_x = sparse.load_npz("preprocess/test_nlp.npz")

train_x = sparse.hstack((train_x, train[features])).tocsr()
test_x = sparse.hstack((test_x, test[features])).tocsr()
```

- 模型训练与优化

接下来进行模型训练，本轮训练的流程和lgb模型流程类似，首先需要创建用于重复申明固定参数的函数，然后定义搜索和优化函数，并在优化函数内部调用参数回调函数，最后定义模型预测函数，该函数将借助交叉验证过程来进行测试集预测，并同步创建验证集预测结果与每个模型对测试集的预测结果。

In [ ]:

```
# 参数回调函数
def params_append(params):
    """

    :param params:
    :return:
    """
    params['objective'] = 'reg:squarederror'
    params['eval_metric'] = 'rmse'
```

```
    params["min_child_weight"] = int(params["min_child_weight"])
    params['max_depth'] = int(params['max_depth'])
    return params

# 模型优化函数
def param_beyesian(train):
    """

    :param train:
    :return:
    """
    # Part 1.数据准备
    train_y = pd.read_csv("data/train.csv")['target']
    # 数据封装
    sample_index = train_y.sample(frac=0.1, random_state=2020).index.tolist()
    train_data = xgb.DMatrix(train.tocsr()[sample_index, :
                             ], train_y.loc[sample_index].values, silent=True)

    # 借助cv过程构建目标函数
    def xgb_cv(colsample_bytree, subsample, min_child_weight, max_depth,
               reg_alpha, eta,
               reg_lambda):
        """

        :param colsample_bytree:
        :param subsample:
        :param min_child_weight:
        :param max_depth:
        :param reg_alpha:
        :param eta:
        :param reg_lambda:
        :return:
        """
        params = {'objective': 'reg:squarederror',
                  'early_stopping_round': 50,
                  'eval_metric': 'rmse'}
        params['colsample_bytree'] = max(min(colsample_bytree, 1), 0)
        params['subsample'] = max(min(subsample, 1), 0)
        params["min_child_weight"] = int(min_child_weight)
        params['max_depth'] = int(max_depth)
        params['eta'] = float(eta)
        params['reg_alpha'] = max(reg_alpha, 0)
        params['reg_lambda'] = max(reg_lambda, 0)
        print(params)
        cv_result = xgb.cv(params, train_data,
                           num_boost_round=1000,
                           nfold=2, seed=2,
                           stratified=False,
                           shuffle=True,
                           early_stopping_rounds=30,
                           verbose_eval=False)
        return -min(cv_result['test-rmse-mean'])

    # 调用贝叶斯优化器进行模型优化
    xgb_bo = BayesianOptimization(
        xgb_cv,
        {'colsample_bytree': (0.5, 1),
         'subsample': (0.5, 1),
         'min_child_weight': (1, 30),
         'max_depth': (5, 12),
         'reg_alpha': (0, 5),
         'eta':(0.02, 0.2),
         'reg_lambda': (0, 5)}
    )
    xgb_bo.maximize(init_points=21, n_iter=5)  # init_points表示初始点，n_iter代表迭代
```

```
    print(xgb_bo.max['target'], xgb_bo.max['params'])
    return xgb_bo.max['params']

# 交叉验证预测函数
def train_predict(train, test, params):
    """

    :param train:
    :param test:
    :param params:
    :return:
    """
    train_y = pd.read_csv("data/train.csv")['target']
    test_data = xgb.DMatrix(test)

    params = params_append(params)
    kf = KFold(n_splits=5, random_state=2020, shuffle=True)
    prediction_test = 0
    cv_score = []
    prediction_train = pd.Series()
    ESR = 30
    NBR = 10000
    VBE = 50
    for train_part_index, eval_index in kf.split(train, train_y):
        # 模型训练
        train_part = xgb.DMatrix(train.tocsr()[train_part_index, :],
                                 train_y.loc[train_part_index])
        eval = xgb.DMatrix(train.tocsr()[eval_index, :],
                           train_y.loc[eval_index])
        bst = xgb.train(params, train_part, NBR, [(train_part, 'train'),
                                                  (eval, 'eval')], verbose_eva
                        maximize=False, early_stopping_rounds=ESR, )
        prediction_test += bst.predict(test_data)
        eval_pre = bst.predict(eval)
        prediction_train = prediction_train.append(pd.Series(eval_pre, index=eval_ind
        score = np.sqrt(mean_squared_error(train_y.loc[eval_index].values, eval_pre))
        cv_score.append(score)
    print(cv_score, sum(cv_score) / 5)
    pd.Series(prediction_train.sort_index().values).to_csv("preprocess/train_xgboost.
    pd.Series(prediction_test / 5).to_csv("preprocess/test_xgboost.csv", index=False)
    test = pd.read_csv('data/test.csv')
    test['target'] = prediction_test / 5
    test[['card_id', 'target']].to_csv("result/submission_xgboost.csv", index=False)
    return
```

In [ ]:

```
best_clf = param_beyesian(train_x)
```

接下来输出模型预测结果：

In [ ]:

```
train_predict(train_x, test_x, best_clf)
```

并将结果提交至竞赛主页

Submission and Description	Private Score	Public Score	Use for Final Score
submission_xgboost.csv a few seconds to go by Hsail6 add submission details	3.62832	3.72358	☐

对比此前模型结果，能够发现目前XBG的模型结果较好，这其中差异极有可能是新增的NLP特征

所导致的。

模型	Private Score	Public Score
randomforest	3.65455	3.74969
randomforest+validation	3.65173	3.74954
LightGBM	3.69723	3.80436
LightGBM+validation	3.64403	3.73875
XGBoost	3.62832	3.72358

此外，课后同学们也可以进一步尝试在LightGBM和随机森林中带入NLP特征来进行计算，尝试是否能够进一步提分。当然，在各项单模型预测结束后，我们即可考虑进行模型融合了。

三、模型融合策略(一)

Voting融合

整体来看，常用模型融合的策略有两种，分别是Voting融合与Stacking融合，模型融合的目的和集成模型中的集成过程类似，都是希望能够尽可能借助不同模型的优势，最终输出一个更加可靠的结果。在Voting过程中，我们只需要对不同模型对测试集的预测结果进行加权汇总即可，而Stacking则相对复杂，需要借助此前不同模型的验证集预测结果和测试集预测结果再次进行模型训练，以验证集预测结果为训练集、训练集标签为标签构建新的训练集，在此模型上进行训练，然后以测试集预测结果作为新的预测集，并在新预测集上进行预测。

1.均值融合

首先我们来看Voting融合过程。一般来说Voting融合也可以分为均值融合（多组预测结果求均值）、加权融合（根据某种方式赋予不同预测结果不同权重而后进行求和）以及Trick融合（根据某种特殊的规则赋予权重而后进行求和）三种，此处先介绍均值融合与加权融合的基本过程。

- 拼接不同模型预测结果

首先，对随机森林、LightGBM和XGBoost模型预测结果进行读取，并简单查看三者相关系数：

In [ ]:
```
data = pd.read_csv("result/submission_randomforest.csv")
data['randomforest'] = data['target'].values

temp = pd.read_csv("result/submission_lightgbm.csv")
data['lightgbm'] = temp['target'].values


temp = pd.read_csv("result/submission_xgboost.csv")
data['xgboost'] = temp['target'].values

print(data.corr())
```

In [ ]:
```
data.head(5)
```

然后计算不同模型预测结果的均值：

In [ ]:
```
data['target'] = (data['randomforest'] + data['lightgbm'] + data['xgboost']) / 3
```

写入结果文件并提交：

In [ ]:
```
data[['card_id','target']].to_csv("result/voting_avr.csv", index=False)
```

Submission and Description	Private Score	Public Score	Use for Final Score
voting_avr.csv just now by Hsail6 add submission details	3.63650	3.73251	☐

模型	Private Score	Public Score
randomforest	3.65455	3.74969
randomforest+validation	3.65173	3.74954
LightGBM	3.69723	3.80436
LightGBM+validation	3.64403	3.73875
XGBoost	3.62832	3.72358
Voting_avr	3.63650	3.73251

能够发现，简单的均值融合并不能有效果上的提升，接下来我们尝试加权融合。

2.加权融合

加权融合的思路并不复杂，从客观计算流程上来看我们将赋予不同模型训练结果以不同权重，而具体权重的分配，我们可以根据三组模型在公榜上的评分决定，即假设模型A和B分别是2分和3分（分数越低越好的情况下），则在实际加权过程中，我们将赋予A模型结果3/5权重，B模型2/5权重，因此，加权融合过程如下：

In [ ]:
```
data['target'] = data['randomforest']*0.2+data['lightgbm']*0.3 + data['xgboost']*0.5
data[['card_id','target']].to_csv("result/voting_wei1.csv", index=False)
```

输出结果如下：

Submission and Description	Private Score	Public Score	Use for Final Score
voting_wei1.csv a few seconds ago by Hsail6 add submission details	3.63307	3.72877	☐

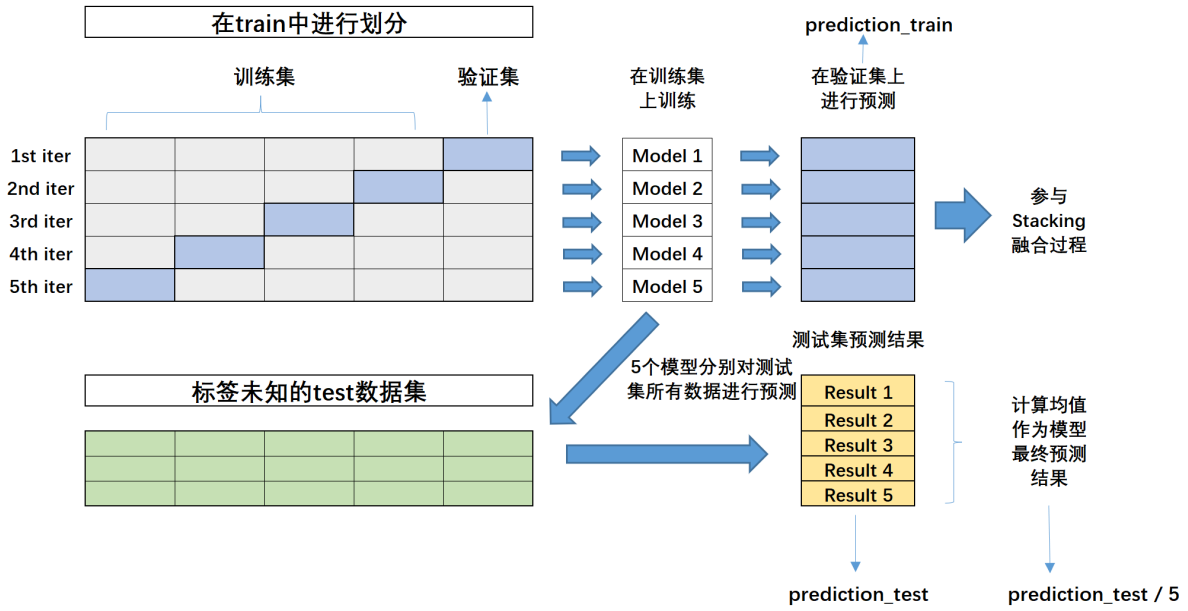
模型	Private Score	Public Score
randomforest	3.65455	3.74969
randomforest+validation	3.65173	3.74954
LightGBM	3.69723	3.80436
LightGBM+validation	3.64403	3.73875
XGBoost	3.62832	3.72358
Voting_avr	3.63650	3.73251
Voting_wei	3.633307	3.72877

能够发现结果略微有所改善，但实际结果仍然不如单模型结果，此处预测极有可能是因额外i标签中存在异常值导致。接下来继续尝试Stacking融合。

四、模型融合策略（二）

Stacking融合

此处我们考虑手动进行Stacking融合，在此前的模型训练中，我们已经创建了predication_train和predication_test数据集，这两个数据集将作为训练集、测试集带入到下一轮的建模中，而本轮建模也被称为Stacking融合。



- 数据集校验

首先快速查看此前创建的相关数据集，其中oof就是训练数据集的预测结果（多轮验证集预测结果拼接而来），而predictions则是此前单模型的预测结果：

In [ ]:
```
oof_rf  = pd.read_csv('./preprocess/train_randomforest.csv')
predictions_rf  = pd.read_csv('./preprocess/test_randomforest.csv')

oof_lgb  = pd.read_csv('./preprocess/train_lightgbm.csv')
predictions_lgb  = pd.read_csv('./preprocess/test_lightgbm.csv')

oof_xgb  = pd.read_csv('./preprocess/train_xgboost.csv')
predictions_xgb  = pd.read_csv('./preprocess/test_xgboost.csv')
```

In [ ]:
```
oof_rf.head(5)
```

In [ ]:
```
predictions_lgb.head(5)
```

In [ ]:
```
oof_rf.shape, oof_lgb.shape
```

In [ ]:
```

```

```
predictions_rf.shape, predictions_lgb.shape
```

接下来进行Stacking模型融合，该过程本身并不复杂，重点需要清洗不同数据集的创建过程与调用过程，同时为了保证模型泛化能力，我们需要对其进行交叉验证：

In [ ]:
```
def stack_model(oof_1, oof_2, oof_3, predictions_1, predictions_2, predictions_3, y):

    # Part 1.数据准备
    # 按行拼接列，拼接验证集所有预测结果
    # train_stack就是final model的训练数据
    train_stack = np.hstack([oof_1, oof_2, oof_3])
    # 按行拼接列，拼接测试集上所有预测结果
    # test_stack就是final model的测试数据
    test_stack = np.hstack([predictions_1, predictions_2, predictions_3])
    # 创建一个和验证集行数相同的全零数组
    # oof = np.zeros(train_stack.shape[0])
    # 创建一个和测试集行数相同的全零数组
    predictions = np.zeros(test_stack.shape[0])

    # Part 2.多轮交叉验证
    from sklearn.model_selection import RepeatedKFold
    folds = RepeatedKFold(n_splits=5, n_repeats=2, random_state=2020)

    # fold_为折数，trn_idx为每一折训练集index，val_idx为每一折验证集index
    for fold_, (trn_idx, val_idx) in enumerate(folds.split(train_stack, y)):
        # 打印折数信息
        print("fold n° {}".format(fold_+1))
        # 训练集中划分为训练数据的特征和标签
        trn_data, trn_y = train_stack[trn_idx], y[trn_idx]
        # 训练集中划分为验证数据的特征和标签
        val_data, val_y = train_stack[val_idx], y[val_idx]
        # 开始训练时提示
        print("-" * 10 + "Stacking " + str(fold_+1) + "-" * 10)
        # 采用贝叶斯回归作为结果融合的模型（final model）
        clf = BayesianRidge()
        # 在训练数据上进行训练
        clf.fit(trn_data, trn_y)
        # 在验证数据上进行预测，并将结果记录在oof对应位置
        # oof[val_idx] = clf.predict(val_data)
        # 对测试集数据进行预测，每一轮预测结果占比额外的1/10
        predictions += clf.predict(test_stack) / (5 * 2)

    # 返回测试集的预测结果
    return predictions
```

接下来执行模型融合过程：

In [ ]:
```
target = train['target'].values
```

In [ ]:
```
predictions_stack  = stack_model(oof_rf, oof_lgb, oof_xgb,
                                 predictions_rf, predictions_lgb, predictions_xgb, tar
```

查看输出结果，并将结果进行提交：

In [ ]:
```
predictions_stack
```

In [ ]:
```
sub_df = pd.read_csv('data/sample_submission.csv')
sub_df["target"] = predictions_stack
```

```
sub_df.to_csv('predictions_stack1.csv', index=False)
```

能够看到最终结果如下：

Submission and Description	Private Score	Public Score	Use for Final Score
predictions_stack1.csv just now by Hsail6 add submission details	3.62798	3.72055	☐

模型	Private Score	Public Score
randomforest	3.65455	3.74969
randomforest+validation	3.65173	3.74954
LightGBM	3.69723	3.80436
LightGBM+validation	3.64403	3.73875
XGBoost	3.62832	3.72358
Voting_avr	3.63650	3.73251
Voting_wei	3.633307	3.72877
Stacking	3.62798	3.72055

截至目前，我们已经尝试了在当前数据情况下的所有有效方法，而通过这一些列尝试，我们也得到了目前最好结果，而整体的排名也从Baseline的前40%提高到20%左右。当然，当这些通用方法都进行了尝试之后，接下来我们需要进一步从更加具体的角度来看当前数据集的数据情况，并据此提出一些跟进一步的优化方法，并更进一步的提高最终预测结果。