数据分析及实践-实验四

PB20000326　徐海阳

Part 1 分类算法实践

1.1 算法主要流程

Step 1. 将lab3中特征工程得到的DIY.csv转换为.npy文件，并进行5折交叉验证所需要的train/test数据集划分

```python
data = np.load('./data/DIY.npy')

data_part = cross_validation(data, 5)

for i in range(5):
    print('part: %d' % (i+1))
    train, test = split_train_test(data_part, i)
```

Step 2. 用logistic回归进行二分类

选用理由：实现的逻辑简单，并且我认为在lab3中我自己提取的特征DIY.npy是不错的，可以简单地认为这些因素加权求和后经过一个非线性变换就可以得到分类结果。

```python
class MLP(object):
    # ...
for i in range(5):
    print('part: %d' % (i+1))
    train, test = split_train_test(data_part, i)
    model = MLP(train, test, train.shape[1]-1, 1, 0.1, 5001)
    model.train_model()
```

1.2 算法关键技术

Step 1: 将DIY.npy数据分为5个part，其中四个concatenate成为train，剩余一个为test。

Step 2: 构造class: MLP，实现sigmoid，sigmoid_derivative（对sigmoid求导），forward（前向传播），judge（judge>0则代表predict结果正确），backward（反向传播），test_model（在test集上计算正确率），train_model （在train集上计算正确率）方法。

1.3 算法实现

关键代码结构如下。完整代码见附录。

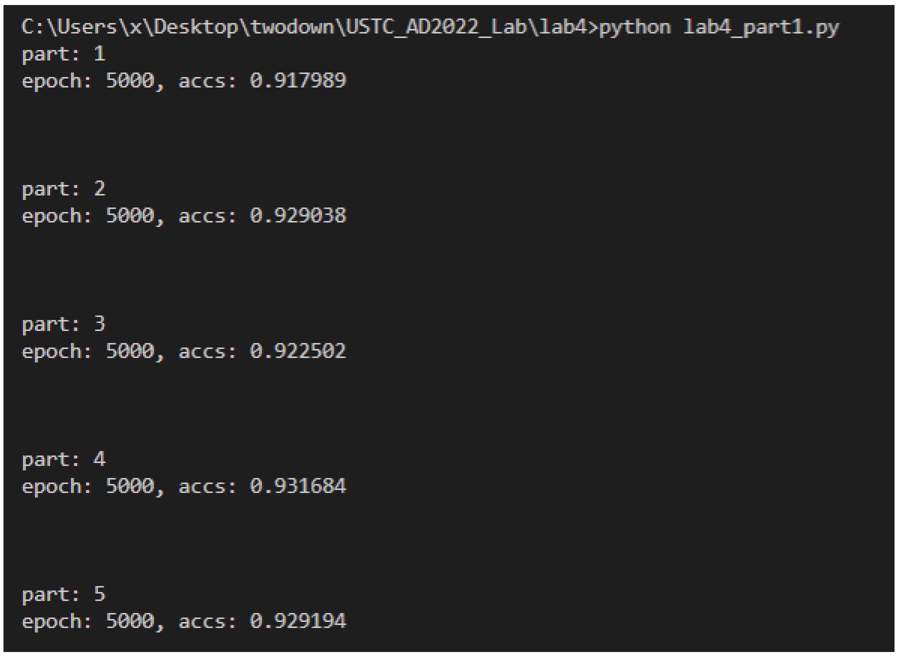
```python
import numpy as np

def cross_validation(data, k):
    # split the data into k parts
    # return the k parts
    return data.reshape(k, -1, data.shape[1])

def split_train_test(data_part, k):
    # split the data_part into train and test
    # return the train and test
    test = data_part[k, :, :]
    train = np.delete(data_part, k, axis=0)
    train = np.concatenate(train, axis=0)
    return train, test

class MLP(object):
    def __init__(self, train, test, n_in, n_out, lr, epoch):
        # ...

    def sigmoid(self, x):
        # ...

    def sigmoid_derivative(self, x):
        # ...

    def forward(self, x): # (bsz, n_in)
        # ...

    def judge(self, y_hat, y):
        """
        if (y_hat > 0.5 and y = 1) or (y_hat < 0.5 and y = 0),then return positive: right predict!
        create a function to judge whether the prediction is correct
        """
        # ...

    def backward(self, x, y): # (bsz, n_in) , (bsz, n_out)
        # ...


    def test_model(self, data, label):
        # ...

    def train_model(self):
        # ...
```

```python

def main():
    data = np.load('./data/DIY.npy')

    data_part = cross_validation(data, 5)

    for i in range(5):
        print('part: %d' % (i+1))
        train, test = split_train_test(data_part, i)
        model = MLP(train, test, train.shape[1]-1, 1, 0.1, 5001)
        model.train_model()
        print('\n\n')

if __name__ == '__main__':
    main()
```

1.4 实验记录

（k折交叉验证，4:1比例，共有5折）

在learning_rate=0.1，epoch=5001时的结果如下：



```
C:\Users\x\Desktop\twodown\USTC_AD2022_Lab\lab4>python lab4_part1.py
part: 1
epoch: 5000, accs: 0.917989



part: 2
epoch: 5000, accs: 0.929038



part: 3
epoch: 5000, accs: 0.922502



part: 4
epoch: 5000, accs: 0.931684



part: 5
epoch: 5000, accs: 0.929194
```

k	ACC
1	0.917989
2	0.929038
3	0.922502
4	0.931684
5	0.929194
平均值	0.926081

观察训练后网络的w可知，“过往是否复读”和“知识掌握程度”的weight是最大的，这也完美吻合了我在lab3特征工程中的数据分析。

Part 2 预测算法实践

1. sklearn.Adaboost, Bagging, DecisionTree, ExtraTree, GradientBoosting, KNN, Lasso, LinearReg, MLP, RandomForest, Ridge, SVM
2. lightgbm

2.0 算法主要流程

1. lab4\lab4_part2_lgb\lab4_p2_preprocess.ipynb：预处理

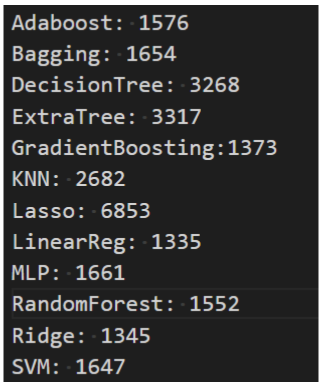
   用label=0延拓train_label的测试集到全体学生，与lab3中预处理好的数据合并，shape为(32130，406)。一方面将测试集部分存入“lab4\lab4_part2_lgb\data\D_test.npy”，训练集部分存入“lab4\lab4_part2_lgb\data\DatawithrawL.npy”；另一方面取这些特征中与MATH相关系数最大的100个特征，存入“lab4\lab4_part2_lgb\data\T100withrawL.npy”中。

2. 采用不同模型在训练集上进行kfold交叉验证
3. 挑选表现最好的模型在测试集上进行predict，保存结果

2.1 实验记录

2.1.1 sklearn

sklearn是简单常用的机器学习库。因此，针对回归问题，我选择了sklearn.ensemble集成学习库和sklearn.linear_model线性模型库进行模型训练。理论上针对回归问题，最经典的LinearRegression的效果应该不会差，可以将它作为baseline。同时，集成学习（Bagging，Adaboost等）应该也有不错的效果。


```
Adaboost: 1576
Bagging: 1654
DecisionTree: 3268
ExtraTree: 3317
GradientBoosting:1373
KNN: 2682
Lasso: 6853
LinearReg: 1335
MLP: 1661
RandomForest: 1552
Ridge: 1345
SVM: 1647
```

首先，我在DatawithrawL训练集上使用默认参数，对各种算法都简单地跑了一遍，结果如上图所示。

先说非线性模型。实验过程中我发现Tree模型耗时长，且效果不是非常好。这一定程度上可能是因为没有调参。于是进行了调参，调了max_depth, num_iterations等参数，发现速度极慢，而且效果仍然很差。我觉得这可能是因为特征数量太多，导致决策树更倾向于划分那些分的子类更多的子节点，这有害决策树的学习。于是我将训练集换为了T100withrawL（前100个最相关的特征），发现效果略有提升，但是速度仍旧很慢而且很难进一步提升。同时，我发现集成算法的效果确实不错。由于在下一部分我将用到lightgbm，它作为GBDT算法，效果优于仅仅利用Boost或Bagging的算法，也优于简单的单棵Tree算法，因此我决定在sklearn中不再对Tree和集成算法做调参，在下一部分lightgbm中详细调参。MLP在手动调参过后hidden_layer_sizes=(128,8),activation='relu',solver='adam',batch_size=1024,learning_rate_init=0.1效果变为1300左右，略优于LinearRegression。

再说线性模型：

1. LinearReg：线性回归，效果不错，作为baseline。
2. KNN：效果不好。我认为也是因为训练集中特征个数太多，并且有部分特征事实上对于MATH没有太多影响，所以在进行distance度量时影响结果。于是我将训练集换成了T100withrawL（前100个最相关的特征），并且在axis=1中只取[:5]，也就是只考虑5个印象因素最大的特征。然后手动调参，发现在n_neighbors=10时效果最好，达到了1800，不如baseline。

3. Lasso：默认情况下正则项alpha=1效果很差。手动调参，在alpha=0.0001时效果最好，达到1600左右，不如baseline。
4. Ridge：同样的，手动调参，在alpha=0.0001时效果最好，达到1300左右，略优于baseline。

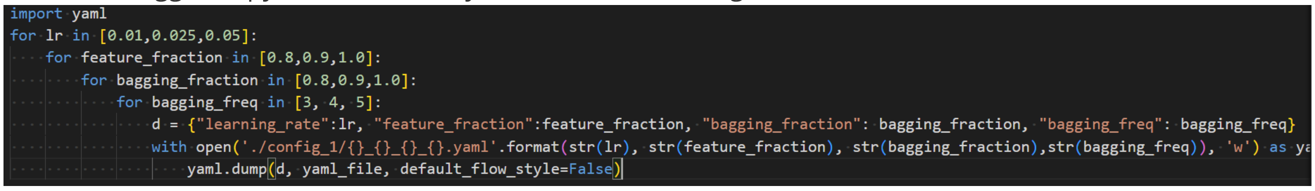
总结，在不使用集成学习的情况下，线性方法中Lasso的效果最好，非线性方法中MLP的效果最好。都达到了1300左右。

2.1.2 lightgbm

lightgbm使用GBDT算法，拥有集成学习的优点，同时相比Xgboost速度快很多。

第一次调参：learning_rate, feature_fraction, bagging_fraction, bagging_freq

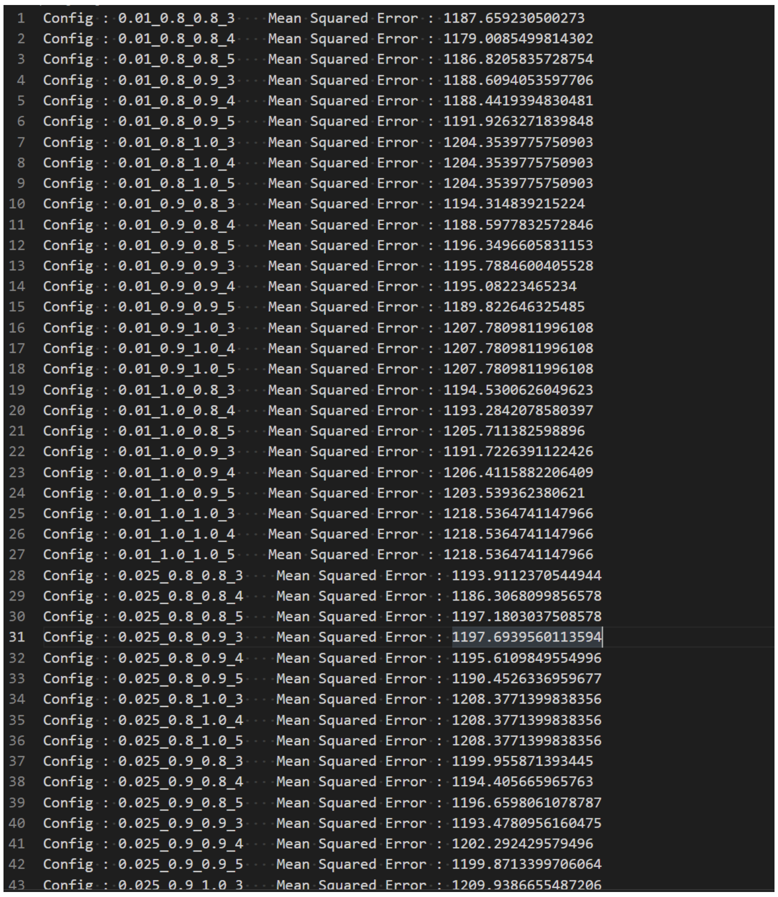
在如下cfggen_1.py中生成超参数的.yaml文件，存入./config_1文件夹。



```python
import yaml
for lr in [0.01,0.025,0.05]:
    for feature_fraction in [0.8,0.9,1.0]:
        for bagging_fraction in [0.8,0.9,1.0]:
            for bagging_freq in [3, 4, 5]:
                d = {"learning_rate":lr, "feature_fraction":feature_fraction, "bagging_fraction": bagging_fraction, "bagging_freq": bagging_freq}
                with open('./config_1/{}_{}_{}_{}.yaml'.format(str(lr), str(feature_fraction), str(bagging_fraction),str(bagging_freq)), 'w') as ya
                    yaml.dump(d, yaml_file, default_flow_style=False)
```

在lgb_1.py中定义函数：1. get_data() 2. get_config(config_file)（根据yaml文件中更新超参）

train_lgb_1.py批量跑这81个参数配置下的模型，结果写入result_1.txt中。部分结果如下：

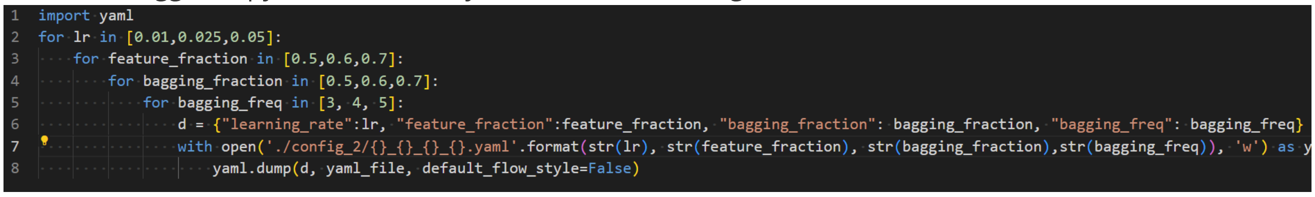


```
Config : 0.01_0.8_0.8_3    Mean Squared Error : 1187.659230500273
Config : 0.01_0.8_0.8_4    Mean Squared Error : 1179.0085499814302
Config : 0.01_0.8_0.8_5    Mean Squared Error : 1186.8205835728754
Config : 0.01_0.8_0.9_3    Mean Squared Error : 1188.6094053597706
Config : 0.01_0.8_0.9_4    Mean Squared Error : 1188.4419394830481
Config : 0.01_0.8_0.9_5    Mean Squared Error : 1191.9263271839848
Config : 0.01_0.8_1.0_3    Mean Squared Error : 1204.3539775750903
Config : 0.01_0.8_1.0_4    Mean Squared Error : 1204.3539775750903
Config : 0.01_0.8_1.0_5    Mean Squared Error : 1204.3539775750903
Config : 0.01_0.9_0.8_3    Mean Squared Error : 1194.314839215224
Config : 0.01_0.9_0.8_4    Mean Squared Error : 1188.5977832572846
Config : 0.01_0.9_0.8_5    Mean Squared Error : 1196.3496605831153
Config : 0.01_0.9_0.9_3    Mean Squared Error : 1195.7884600405528
Config : 0.01_0.9_0.9_4    Mean Squared Error : 1195.08223465234
Config : 0.01_0.9_0.9_5    Mean Squared Error : 1189.822646325485
Config : 0.01_0.9_1.0_3    Mean Squared Error : 1207.7809811996108
Config : 0.01_0.9_1.0_4    Mean Squared Error : 1207.7809811996108
Config : 0.01_0.9_1.0_5    Mean Squared Error : 1207.7809811996108
Config : 0.01_1.0_0.8_3    Mean Squared Error : 1194.5300626049623
Config : 0.01_1.0_0.8_4    Mean Squared Error : 1193.2842078580397
Config : 0.01_1.0_0.8_5    Mean Squared Error : 1205.711382598896
Config : 0.01_1.0_0.9_3    Mean Squared Error : 1191.7226391122426
Config : 0.01_1.0_0.9_4    Mean Squared Error : 1206.4115882206409
Config : 0.01_1.0_0.9_5    Mean Squared Error : 1203.539362380621
Config : 0.01_1.0_1.0_3    Mean Squared Error : 1218.5364741147966
Config : 0.01_1.0_1.0_4    Mean Squared Error : 1218.5364741147966
Config : 0.01_1.0_1.0_5    Mean Squared Error : 1218.5364741147966
Config : 0.025_0.8_0.8_3    Mean Squared Error : 1193.9112370544944
Config : 0.025_0.8_0.8_4    Mean Squared Error : 1186.3068099856578
Config : 0.025_0.8_0.8_5    Mean Squared Error : 1197.1803037508578
Config : 0.025_0.8_0.9_3    Mean Squared Error : 1197.6939560113594
Config : 0.025_0.8_0.9_4    Mean Squared Error : 1195.6109849554996
Config : 0.025_0.8_0.9_5    Mean Squared Error : 1190.4526336959677
Config : 0.025_0.8_1.0_3    Mean Squared Error : 1208.3771399838356
Config : 0.025_0.8_1.0_4    Mean Squared Error : 1208.3771399838356
Config : 0.025_0.8_1.0_5    Mean Squared Error : 1208.3771399838356
Config : 0.025_0.9_0.8_3    Mean Squared Error : 1199.955871393445
Config : 0.025_0.9_0.8_4    Mean Squared Error : 1194.405665965763
Config : 0.025_0.9_0.8_5    Mean Squared Error : 1196.6598061078787
Config : 0.025_0.9_0.9_3    Mean Squared Error : 1193.4780956160475
Config : 0.025_0.9_0.9_4    Mean Squared Error : 1202.292429579496
Config : 0.025_0.9_0.9_5    Mean Squared Error : 1199.8713399706064
Config : 0.025_0.9_1.0_3    Mean Squared Error : 1209.9386655487206
```

发现learning_rate=0.01时最优，同时feature_fraction和bagging_fraction在0.8时优于0.9和1.0，这说明这两个超参并不在最有区间内取值，取值范围还可以更小，小于0.8。根据这些信息进行第二次调参。

第二次调参: 调小feature_fraction和bagging_fraction的取值区间

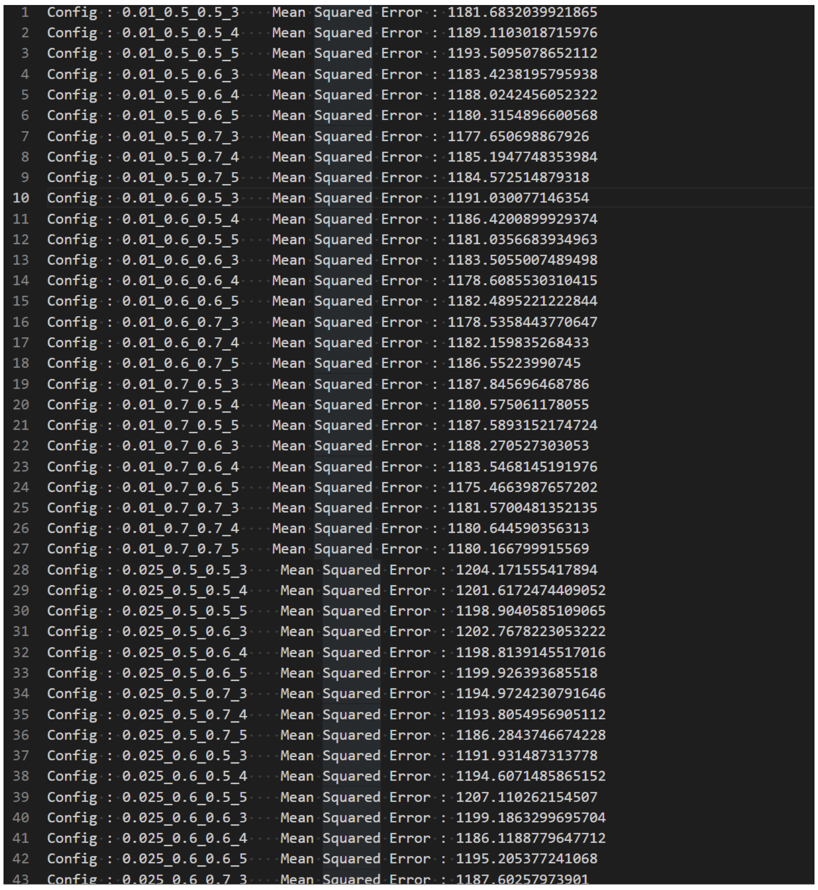
在如下cfggen_2.py中生成超参数的.yaml文件，存入./config_2文件夹。



```
import yaml
for lr in [0.01,0.025,0.05]:
    for feature_fraction in [0.5,0.6,0.7]:
        for bagging_fraction in [0.5,0.6,0.7]:
            for bagging_freq in [3, 4, 5]:
                d = {"learning_rate":lr, "feature_fraction":feature_fraction, "bagging_fraction": bagging_fraction, "bagging_freq": bagging_freq}
                with open('./config_2/{}_{}_{}_{}.yaml'.format(str(lr), str(feature_fraction), str(bagging_fraction),str(bagging_freq)), 'w') as y
                    yaml.dump(d, yaml_file, default_flow_style=False)
```

在lgb_2.py中定义函数：1. get_data() 2. get_config(config_file)（根据yaml文件中更新超参）

train_lgb_2.py批量跑这81个参数配置下的模型，结果写入result_2.txt中。部分结果如下：



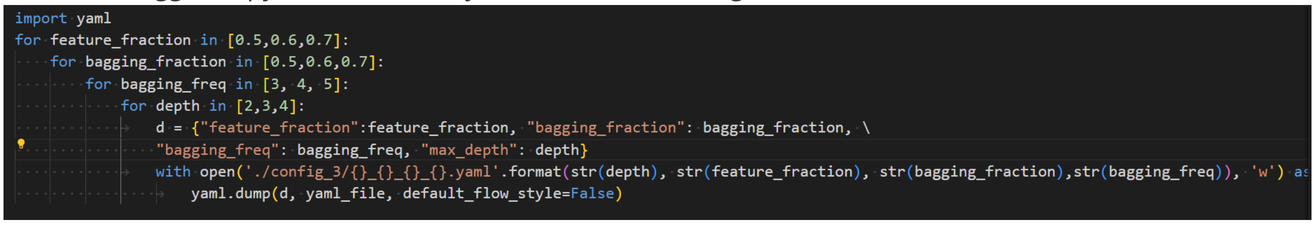
```
Config : 0.01_0.5_0.5_3    Mean Squared Error : 1181.6832039921865
Config : 0.01_0.5_0.5_4    Mean Squared Error : 1189.1103018715976
Config : 0.01_0.5_0.5_5    Mean Squared Error : 1193.5095078652112
Config : 0.01_0.5_0.6_3    Mean Squared Error : 1183.4238195795938
Config : 0.01_0.5_0.6_4    Mean Squared Error : 1188.0242456052322
Config : 0.01_0.5_0.6_5    Mean Squared Error : 1180.3154896600568
Config : 0.01_0.5_0.7_3    Mean Squared Error : 1177.650698867926
Config : 0.01_0.5_0.7_4    Mean Squared Error : 1185.1947748353984
Config : 0.01_0.5_0.7_5    Mean Squared Error : 1184.572514879318
Config : 0.01_0.6_0.5_3    Mean Squared Error : 1191.030077146354
Config : 0.01_0.6_0.5_4    Mean Squared Error : 1186.4200899929374
Config : 0.01_0.6_0.5_5    Mean Squared Error : 1181.0356683934963
Config : 0.01_0.6_0.6_3    Mean Squared Error : 1183.5055007489498
Config : 0.01_0.6_0.6_4    Mean Squared Error : 1178.6085530310415
Config : 0.01_0.6_0.6_5    Mean Squared Error : 1182.4895221222844
Config : 0.01_0.6_0.7_3    Mean Squared Error : 1178.5358443770647
Config : 0.01_0.6_0.7_4    Mean Squared Error : 1182.159835268433
Config : 0.01_0.6_0.7_5    Mean Squared Error : 1186.55223990745
Config : 0.01_0.7_0.5_3    Mean Squared Error : 1187.845696468786
Config : 0.01_0.7_0.5_4    Mean Squared Error : 1180.575061178055
Config : 0.01_0.7_0.5_5    Mean Squared Error : 1187.5893152174724
Config : 0.01_0.7_0.6_3    Mean Squared Error : 1188.270527303053
Config : 0.01_0.7_0.6_4    Mean Squared Error : 1183.5468145191976
Config : 0.01_0.7_0.6_5    Mean Squared Error : 1175.4663987657202
Config : 0.01_0.7_0.7_3    Mean Squared Error : 1181.5700481352135
Config : 0.01_0.7_0.7_4    Mean Squared Error : 1180.644590356313
Config : 0.01_0.7_0.7_5    Mean Squared Error : 1180.166799915569
Config : 0.025_0.5_0.5_3    Mean Squared Error : 1204.171555417894
Config : 0.025_0.5_0.5_4    Mean Squared Error : 1201.6172474409052
Config : 0.025_0.5_0.5_5    Mean Squared Error : 1198.9040585109065
Config : 0.025_0.5_0.6_3    Mean Squared Error : 1202.7678223053222
Config : 0.025_0.5_0.6_4    Mean Squared Error : 1198.8139145517016
Config : 0.025_0.5_0.6_5    Mean Squared Error : 1199.926393685518
Config : 0.025_0.5_0.7_3    Mean Squared Error : 1194.9724230791646
Config : 0.025_0.5_0.7_4    Mean Squared Error : 1193.8054956905112
Config : 0.025_0.5_0.7_5    Mean Squared Error : 1186.2843746674228
Config : 0.025_0.6_0.5_3    Mean Squared Error : 1191.931487313778
Config : 0.025_0.6_0.5_4    Mean Squared Error : 1194.6071485865152
Config : 0.025_0.6_0.5_5    Mean Squared Error : 1207.110262154507
Config : 0.025_0.6_0.6_3    Mean Squared Error : 1199.1863299695704
Config : 0.025_0.6_0.6_4    Mean Squared Error : 1186.1188779647712
Config : 0.025_0.6_0.6_5    Mean Squared Error : 1195.205377241068
Config : 0.025_0.6_0.7_3    Mean Squared Error : 1187.60257973901
```

发现效果优于第一次调参的结果。仍然是在learning_rate=0.01时最优。同时大部分feature_fraction和bagging_fraction在0.6要优于0.5，也有部分在0.7时更好，这说明这两个超参的取值区间已经可以大致确定在[0.5, 0.6, 0.7]。

这些参数的最优取值区间确定后，开始对最重要的max_depth（梯度提升树的深度）进行调参。

第三次调参: max_depth

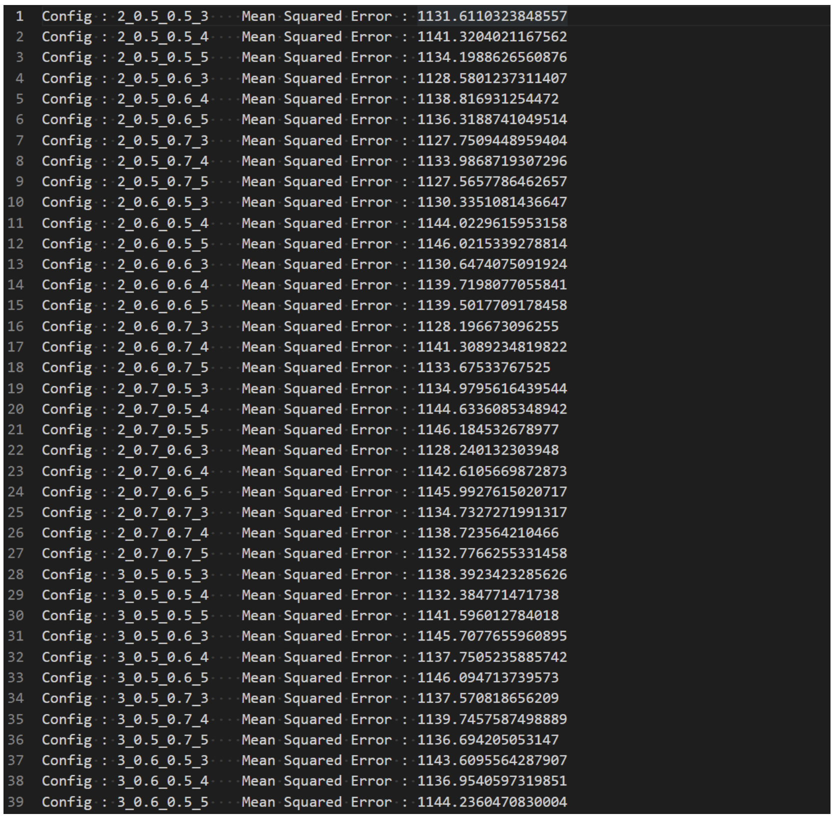
在如下cfggen_3.py中生成超参数的.yaml文件，存入./config_3文件夹。



```
import yaml
for feature_fraction in [0.5,0.6,0.7]:
    for bagging_fraction in [0.5,0.6,0.7]:
        for bagging_freq in [3, 4, 5]:
            for depth in [2,3,4]:
                d = {"feature_fraction":feature_fraction, "bagging_fraction": bagging_fraction, \
                "bagging_freq": bagging_freq, "max_depth": depth}
                with open('./config_3/{}_{}_{}_{}.yaml'.format(str(depth), str(feature_fraction), str(bagging_fraction),str(bagging_freq)), 'w') as
                    yaml.dump(d, yaml_file, default_flow_style=False)
```

在lgb_3.py中定义函数：1. get_data() 2. get_config(config_file)（根据yaml文件中更新超参）

train_lgb_3.py批量跑这81个参数配置下的模型，结果写入result_3.txt中。部分结果如下：



```
Config : 2_0.5_0.5_3    Mean Squared Error : 1131.6110323848557
Config : 2_0.5_0.5_4    Mean Squared Error : 1141.3204021167562
Config : 2_0.5_0.5_5    Mean Squared Error : 1134.1988626560876
Config : 2_0.5_0.6_3    Mean Squared Error : 1128.5801237311407
Config : 2_0.5_0.6_4    Mean Squared Error : 1138.816931254472
Config : 2_0.5_0.6_5    Mean Squared Error : 1136.3188741049514
Config : 2_0.5_0.7_3    Mean Squared Error : 1127.7509448959404
Config : 2_0.5_0.7_4    Mean Squared Error : 1133.9868719307296
Config : 2_0.5_0.7_5    Mean Squared Error : 1127.5657786462657
Config : 2_0.6_0.5_3    Mean Squared Error : 1130.3351081436647
Config : 2_0.6_0.5_4    Mean Squared Error : 1144.0229615953158
Config : 2_0.6_0.5_5    Mean Squared Error : 1146.0215339278814
Config : 2_0.6_0.6_3    Mean Squared Error : 1130.6474075091924
Config : 2_0.6_0.6_4    Mean Squared Error : 1139.7198077055841
Config : 2_0.6_0.6_5    Mean Squared Error : 1139.5017709178458
Config : 2_0.6_0.7_3    Mean Squared Error : 1128.196673096255
Config : 2_0.6_0.7_4    Mean Squared Error : 1141.3089234819822
Config : 2_0.6_0.7_5    Mean Squared Error : 1133.67533767525
Config : 2_0.7_0.5_3    Mean Squared Error : 1134.9795616439544
Config : 2_0.7_0.5_4    Mean Squared Error : 1144.6336085348942
Config : 2_0.7_0.5_5    Mean Squared Error : 1146.184532678977
Config : 2_0.7_0.6_3    Mean Squared Error : 1128.240132303948
Config : 2_0.7_0.6_4    Mean Squared Error : 1142.6105669872873
Config : 2_0.7_0.6_5    Mean Squared Error : 1145.9927615020717
Config : 2_0.7_0.7_3    Mean Squared Error : 1134.7327271991317
Config : 2_0.7_0.7_4    Mean Squared Error : 1138.723564210466
Config : 2_0.7_0.7_5    Mean Squared Error : 1132.7766255331458
Config : 3_0.5_0.5_3    Mean Squared Error : 1138.3923423285626
Config : 3_0.5_0.5_4    Mean Squared Error : 1132.384771471738
Config : 3_0.5_0.5_5    Mean Squared Error : 1141.596012784018
Config : 3_0.5_0.6_3    Mean Squared Error : 1145.7077655960895
Config : 3_0.5_0.6_4    Mean Squared Error : 1137.7505235885742
Config : 3_0.5_0.6_5    Mean Squared Error : 1146.094713739573
Config : 3_0.5_0.7_3    Mean Squared Error : 1137.570818656209
Config : 3_0.5_0.7_4    Mean Squared Error : 1139.7457587498889
Config : 3_0.5_0.7_5    Mean Squared Error : 1136.694205053147
Config : 3_0.6_0.5_3    Mean Squared Error : 1143.6095564287907
Config : 3_0.6_0.5_4    Mean Squared Error : 1136.9540597319851
Config : 3_0.6_0.5_5    Mean Squared Error : 1144.2360470830004
```

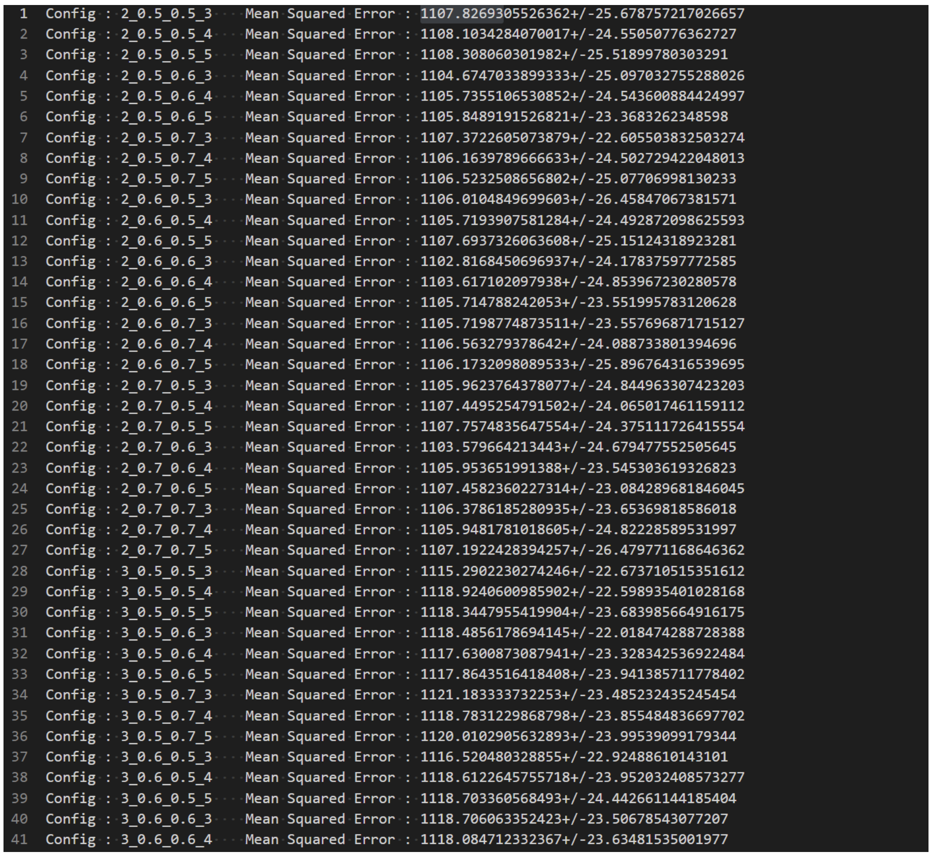
发现效果大大优于第二次调参的结果。仍然是在learning_rate=0.01时最优。树的深度在2时效果较好，也有些情况在3时最优。

此时，全部参数的最优取值区间确定下来。下一步根据kfold交叉验证进行细致实验。

kfold

取定k=6（5不整除训练集样本数）。在train_lgb_3.py的基础上加入kfold的代码得到kfold.py。计算MSE的均值和方差。

结果写入kfold.txt。部分结果如下：



```
Config : 2_0.5_0.5_3    Mean Squared Error : 1107.8269305526362+/-25.678757217026657
Config : 2_0.5_0.5_4    Mean Squared Error : 1108.1034284070017+/-24.55050776362727
Config : 2_0.5_0.5_5    Mean Squared Error : 1108.308060301982+/-25.51899780303291
Config : 2_0.5_0.6_3    Mean Squared Error : 1104.6747033899333+/-25.097032755288026
Config : 2_0.5_0.6_4    Mean Squared Error : 1105.7355106530852+/-24.543600884424997
Config : 2_0.5_0.6_5    Mean Squared Error : 1105.8489191526821+/-23.3683262348598
Config : 2_0.5_0.7_3    Mean Squared Error : 1107.3722605073879+/-22.605503832503274
Config : 2_0.5_0.7_4    Mean Squared Error : 1106.1639789666633+/-24.502729422048013
Config : 2_0.5_0.7_5    Mean Squared Error : 1106.5232508656802+/-25.07706998130233
Config : 2_0.6_0.5_3    Mean Squared Error : 1106.0104849699603+/-26.45847067381571
Config : 2_0.6_0.5_4    Mean Squared Error : 1105.7193907581284+/-24.492872098625593
Config : 2_0.6_0.5_5    Mean Squared Error : 1107.6937326063608+/-25.15124318923281
Config : 2_0.6_0.6_3    Mean Squared Error : 1102.8168450696937+/-24.17837597772585
Config : 2_0.6_0.6_4    Mean Squared Error : 1103.617102097938+/-24.853967230280578
Config : 2_0.6_0.6_5    Mean Squared Error : 1105.714788242053+/-23.551995783120628
Config : 2_0.6_0.7_3    Mean Squared Error : 1105.7198774873511+/-23.557696871715127
Config : 2_0.6_0.7_4    Mean Squared Error : 1106.563279378642+/-24.088733801394696
Config : 2_0.6_0.7_5    Mean Squared Error : 1106.1732098089533+/-25.896764316539695
Config : 2_0.7_0.5_3    Mean Squared Error : 1105.9623764378077+/-24.844963307423203
Config : 2_0.7_0.5_4    Mean Squared Error : 1107.4495254791502+/-24.065017461159112
Config : 2_0.7_0.5_5    Mean Squared Error : 1107.7574835647554+/-24.375111726415554
Config : 2_0.7_0.6_3    Mean Squared Error : 1103.579664213443+/-24.679477552505645
Config : 2_0.7_0.6_4    Mean Squared Error : 1105.953651991388+/-23.545303619326823
Config : 2_0.7_0.6_5    Mean Squared Error : 1107.4582360227314+/-23.084289681846045
Config : 2_0.7_0.7_3    Mean Squared Error : 1106.3786185280935+/-23.65369818586018
Config : 2_0.7_0.7_4    Mean Squared Error : 1105.9481781018605+/-24.82228589531997
Config : 2_0.7_0.7_5    Mean Squared Error : 1107.1922428394257+/-26.479771168646362
Config : 3_0.5_0.5_3    Mean Squared Error : 1115.2902230274246+/-22.673710515351612
Config : 3_0.5_0.5_4    Mean Squared Error : 1118.9240600985902+/-22.598935401028168
Config : 3_0.5_0.5_5    Mean Squared Error : 1118.3447955419904+/-23.683985664916175
Config : 3_0.5_0.6_3    Mean Squared Error : 1118.4856178694145+/-22.018474288728388
Config : 3_0.5_0.6_4    Mean Squared Error : 1117.6300873087941+/-23.328342536922484
Config : 3_0.5_0.6_5    Mean Squared Error : 1117.8643516418408+/-23.941385711778402
Config : 3_0.5_0.7_3    Mean Squared Error : 1121.183333732253+/-23.485232435245454
Config : 3_0.5_0.7_4    Mean Squared Error : 1118.7831229868798+/-23.855484836697702
Config : 3_0.5_0.7_5    Mean Squared Error : 1120.0102905632893+/-23.99539099179344
Config : 3_0.6_0.5_3    Mean Squared Error : 1116.520480328855+/-22.92488610143101
Config : 3_0.6_0.5_4    Mean Squared Error : 1118.6122645755718+/-23.952032408573277
Config : 3_0.6_0.5_5    Mean Squared Error : 1118.703360568493+/-24.442661144185404
Config : 3_0.6_0.6_3    Mean Squared Error : 1118.706063352423+/-23.50678543077207
Config : 3_0.6_0.6_4    Mean Squared Error : 1118.084712332367+/-23.63481535001977
```

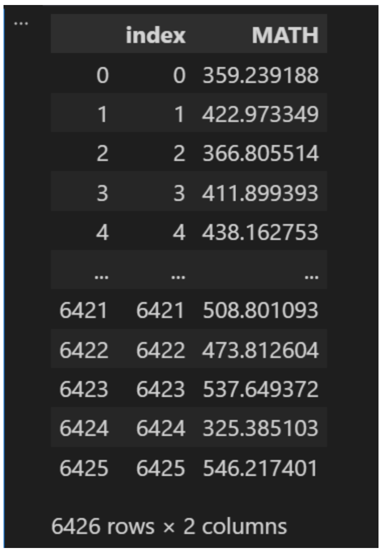
然后在bestiter.py中取MSE最小的5个模型，在k=12时再进行一次kfold，目的是得到best_iteration，然后在整个训练集上训练这么多轮。

结果如下：

```
Config : 2_0.6_0.6_3    Mean Squared Error : 1098.1580745340102+/-34.2975399549576    Best Iteration : 32451.166666666668+/-4717.377693756941
Config : 2_0.7_0.6_3    Mean Squared Error : 1097.676663516747+/-35.08556141294883    Best Iteration : 31914.0+/-4395.798069368822
Config : 2_0.6_0.6_4    Mean Squared Error : 1099.3406791853276+/-34.502537660792655    Best Iteration : 32134.5+/-5463.346265492118
Config : 2_0.5_0.6_3    Mean Squared Error : 1097.5901120691776+/-36.21014569132887    Best Iteration : 34260.666666666664+/-6025.1981617942565
Config : 2_0.6_0.6_5    Mean Squared Error : 1098.9752482861252+/-33.96305809696094    Best Iteration : 30407.416666666668+/-6040.322307326728
```

最后在predict.py中用这5个模型在整个训练集上进行训练，然后在测试集D_test上predict，对得到的5个predictions做平均作为最终的测试集预测结果。

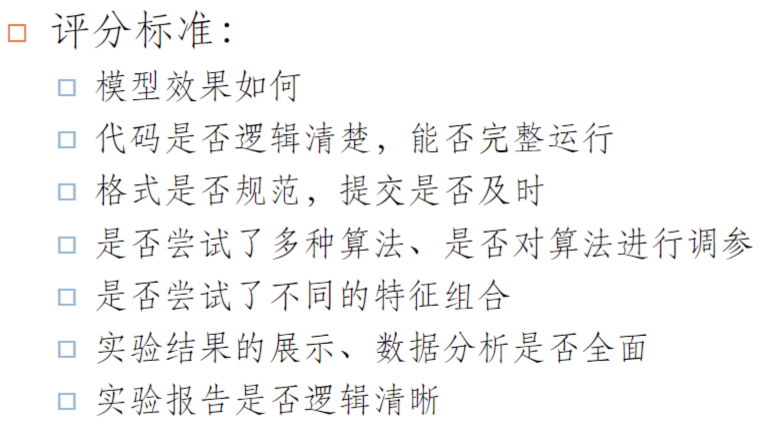
将结果按格式要求存入csv文件。



	index	MATH
0	0	359.239188
1	1	422.973349
2	2	366.805514
3	3	411.899393
4	4	438.162753
...	...	...
6421	6421	508.801093
6422	6422	473.812604
6423	6423	537.649372
6424	6424	325.385103
6425	6425	546.217401

6426 rows × 2 columns

3. 完成内容汇总



- 评分标准：
  - 模型效果如何
  - 代码是否逻辑清楚，能否完整运行
  - 格式是否规范，提交是否及时
  - 是否尝试了多种算法、是否对算法进行调参
  - 是否尝试了不同的特征组合
  - 实验结果的展示、数据分析是否全面
  - 实验报告是否逻辑清晰

1，2，3：完成

4：在part2中尝试了多种算法（sklearn的多种和lightgbm），并且详细调参

5：在part2中尝试了不同特征组合（Tree模型和KNN模型等用T100withrawL训练集，是取原始训练集中与MATH相关系数最大的100个特征组成的训练集）

6，7：完成

附录: part1完整代码

```python
import numpy as np
```

```python

def cross_validation(data, k):
    # split the data into k parts
    # return the k parts
    return data.reshape(k, -1, data.shape[1])

def split_train_test(data_part, k):
    # split the data_part into train and test
    # return the train and test
    test = data_part[k, :, :]
    train = np.delete(data_part, k, axis=0)
    train = np.concatenate(train, axis=0)
    return train, test

class MLP(object):
    def __init__(self, train, test, n_in, n_out, lr, epoch):
        # self.bsz = train.shape[0]
        # self.train = train
        # self.test = test
        self.n_in = n_in
        self.n_out = n_out
        self.lr = lr
        self.epoch = epoch
        self.train_data = train[:, :-1]
        train_label = train[:, -1]
        self.train_label = train_label.reshape(train_label.shape[0], 1)
        self.test_data = test[:, :-1]
        test_label = test[:, -1]
        self.test_label = test_label.reshape(test_label.shape[0], 1)
        self.w = np.random.randn(self.n_in, self.n_out)
        self.b = np.random.randn(self.n_out)

    def sigmoid(self, x):
        return 1 / (1 + np.exp(-x))

    def sigmoid_derivative(self, x):
        return self.sigmoid(x) * (1 - self.sigmoid(x))

    def forward(self, x): # (bsz, n_in)
        o = np.dot(x, self.w) + self.b # (bsz, n_out)
        y_hat = self.sigmoid(o) # (bsz, n_out)
        return o, y_hat

    def judge(self, y_hat, y):
        """
        if (y_hat > 0.5 and y = 1) or (y_hat < 0.5 and y = 0),then return positive: right predict!
        create a function to judge whether the prediction is correct
        """
        s = (y_hat - 0.5) * (y - 0.5)
        return s

    def backward(self, x, y): # (bsz, n_in) , (bsz, n_out)
        bsz = x.shape[0]
        o, y_hat = self.forward(x) # (bsz, n_out)

        # 链式法则
        d_L_d_y_hat = -y/y_hat + (np.ones_like(y)-y)/(np.ones_like(y)-y_hat) # (bsz, n_out)
        d_y_hat_d_o = self.sigmoid_derivative(y_hat) # (bsz, n_out)
        d_o_d_w = x # (bsz, n_in)
        d_o_d_b = np.ones((bsz, 1)) # (bsz, 1)

        d_L_d_w = np.mean(d_L_d_y_hat * d_y_hat_d_o * d_o_d_w, axis=0) # (n_in,)
        d_L_d_w = d_L_d_w.reshape(self.n_in, 1) # (n_in, 1)
        d_L_d_b = np.mean(d_L_d_y_hat * d_y_hat_d_o * d_o_d_b, axis=0) # (1,)
        self.w = self.w - self.lr * d_L_d_w
        self.b = self.b - self.lr * d_L_d_b


    def test_model(self, data, label):
        # pdb.set_trace()
        o, y_hat = self.forward(data)
        total = data.shape[0]
        """
        correct are those whose prediction is correct, i.e. y_hat > 0.5 and y = 1
        """
        correct = np.sum(self.judge(y_hat, label)>0)
        accs = correct / total
        return accs

    def train_model(self):
        for i in range(self.epoch):
            self.backward(self.train_data, self.train_label)
            if i!=0 and i%5000 == 0:
                accs = self.test_model(self.test_data, self.test_label)
                print('epoch: %d, accs: %f' % (i, accs))

def main():
    data = np.load('./data/DIY.npy')

    data_part = cross_validation(data, 5)

    for i in range(5):
        print('part: %d' % (i+1))
        train, test = split_train_test(data_part, i)
        model = MLP(train, test, train.shape[1]-1, 1, 0.1, 5001)
        model.train_model()
        print('\n\n')

if __name__ == '__main__':
    main()
```