LAB5 Comprehensive Experiment

袁雨 PB20151804

一、 实验目的

- 考察完整进行实验的能力。此处"实验"的含义是指整个任务的过程，不单单指模型的构建。

二、实验原理

- 一般来说，我们整个分类任务可以分为以下部分：

1. 获取数据集，对数据进行分析；
2. 对数据进行处理，形成测试集和测试集；
3. 对于任务，选择合适的模型；
4. 利用训练集训练模型，调整超参数以达到在测试集上达到更好的效果，保存模型；
5. 注意你的评价指标是否合适，同时进行假设检验；
6. 可视化你的结果；
7. 挑选一个最好的模型，用其对 test_label 进行预测，提交你的 pred。

- 二分类推广到多分类

  最经典的拆分策略有三种："一对一" (One vs. One, 简称 OvO)、"一对其余" (One vs. Rest, 简称 OvR)和 "多对多" (Many vs. Many, 简称 MvM)。本次实验尝试了OvO与OvR。

  给定数据集 $D=\{(\boldsymbol{x}_1,y_1),(\boldsymbol{x}_2,y_2),\ldots,(\boldsymbol{x}_m,y_m)\},y_i\in\{C_1,C_2,\ldots,C_N\}$。OvO 将这 N 个类别两两配对，从而产生 $N(N-1)/2$ 个二分类任务，例如 OvO 将为区分类别 C_i 和 C_j 训练一个分类器，该分类器把 D 中的 C_i 类样例作为正例，C_j 类样例作为反例。在测试阶段，新样本将同时提交给所有分类器，于是我们将得到 $N(N-1)/2$ 个分类结果，最终结果可通过投票产生：即把被预测得最多的类别作为最终分类结果。

  OvR 则是每次将一个类的样例作为正例、所有其他类的样例作为反例来训练 N 个分类器。在测试时若仅有一个分类器预测为正类，则对应的类别标记作为最终分类结果。若有多个分类器预测为正类，则通常考虑各分类器的预测置信度，选择置信度最大的类别标记作为分类结果。

  容易看出，OvR 只需训练 N 个分类器，而 OvO 需训练 $N(N-1)/2$ 个分 类器，因此，OvO 的存储开销和测试时间开销通常比 OvR 更大。但在训练时，OvR 的每个分类器均使用全部训练样例，而 OvO 的每个分类器仅用到两个类的样例，因此，在类别很多时，OvO 的训练时间开销通常比 OvR 更小。至于预测性能，则取决于具体的数据分布，在多数情形下两者差不多。

- 具体的模型可参考本课程涉及的所有分类模型及其原理。

- 评价指标

- $Acc(f;D)=1-E(f;D)$，分类精度，越大越好
- 运行时间

三、实验步骤

（一）导入数据

1. 读入数据

```
# 导入
df_train_feature = pd.read_csv("Dataset/train_feature.csv")
df_train_label = pd.read_csv("Dataset/train_label.csv")
df_test_feature = pd.read_csv("Dataset/test_feature.csv")
# 显示
pd.set_option('display.max_rows',None)
pd.set_option('display.max_columns',None)
# 合并
df_train = pd.concat([df_train_feature,df_train_label],axis=1)
```

使用pandas中的read_csv()函数读入数据，并将train_feature与train_label合并成一个新的DataFrame。

2. 基本信息

```
df_train.info(verbose=True, show_counts=True)
df_test_feature.info(verbose=True, show_counts=True)
df_train.sample(5)
df_train.describe()
df_test_feature.describe()
```

在`train_feature.csv`文件中，有10000条120维特征数据，数据类型为 float 或 int 。

在`train_label.csv`文件中，有10000条1维标签数据，数据类型为 int 。

在`test_feature.csv`文件中，有3000条120维特征数据。

获取训练集的具体统计信息如下表，因篇幅原因仅展示前十个特征。

	feature_0	feature_1	feature_2	feature_3	feature_4	feature_5	feature_6	feature_7	feature_8	feature_9
count	9960.000000	9946.000000	9955.000000	9952.000000	9954.000000	9.955000e+03	9956.000000	9956.000000	9935.000000	9954.000000
mean	1381.896486	58.715384	11.157463	215.609224	104.244826	5.159948e+03	1418.185014	193.551326	49.467136	228.066855
std	20342.092034	3067.018960	213.548216	3254.119426	1809.139214	8.325174e+04	21803.434424	3078.353897	1062.106891	4636.090534
min	26.000000	-72400.900016	0.000004	3.000000	0.000000	0.000000e+00	27.000000	3.000000	1.000000	0.042353
25%	45.000000	-5.442525	0.248601	7.000000	3.000000	1.620000e+02	45.000000	7.000000	1.000000	1.390866
50%	50.000000	-0.060218	0.504924	8.000000	4.000000	1.940000e+02	50.000000	8.000000	2.000000	2.730417
75%	55.000000	5.482893	0.751965	8.000000	5.000000	2.220000e+02	55.000000	8.000000	4.000000	5.413222
max	526876.000000	105832.015208	8448.902842	79794.000000	57936.000000	2.072000e+06	505945.000000	81060.000000	47816.000000	210388.669161

获取测试集的具体统计信息如下表。

	feature_0	feature_1	feature_2	feature_3	feature_4	feature_5	feature_6	feature_7	feature_8	feature_9
count	2989.000000	2978.000000	2986.000000	2993.000000	2982.000000	2.988000e+03	2978.000000	2985.000000	2985.000000	2985.000000
mean	1762.320843	91.910842	15.729200	265.862011	96.420188	4.984848e+03	1703.073875	255.758459	73.217755	109.578258
std	25164.050301	4582.805525	264.728244	3581.427976	1620.836020	7.474121e+04	24267.788219	3327.352630	1314.130579	3029.658060
min	31.000000	-120240.098513	0.001354	3.000000	0.000000	0.000000e+00	30.000000	3.000000	1.000000	0.080055
25%	45.000000	-5.573117	0.254053	7.000000	3.000000	1.630000e+02	45.000000	7.000000	1.000000	1.361981
50%	50.000000	-0.108952	0.493951	8.000000	4.000000	1.930000e+02	50.000000	7.000000	2.000000	2.667552
75%	55.000000	5.182033	0.758512	8.000000	5.000000	2.220000e+02	55.000000	8.000000	4.000000	5.384259
max	601705.000000	113501.350205	9410.525787	82062.000000	37962.000000	1.743624e+06	460649.000000	69288.000000	45012.000000	118302.525010

分析可知，训练集的每个特征列都含有50个左右的缺失值，测试集含20个左右。

大部分特征的均值和中位数都相差很大，数据中存在很离谱的outlier。

3. 关键变量

```
df_train['label'].value_counts()
```

因为不知道数据集的其他特征含义，故只对标签进行统计分析，结果如下表。

标签值	数量
0	2523
1	2521
2	2500
3	2456

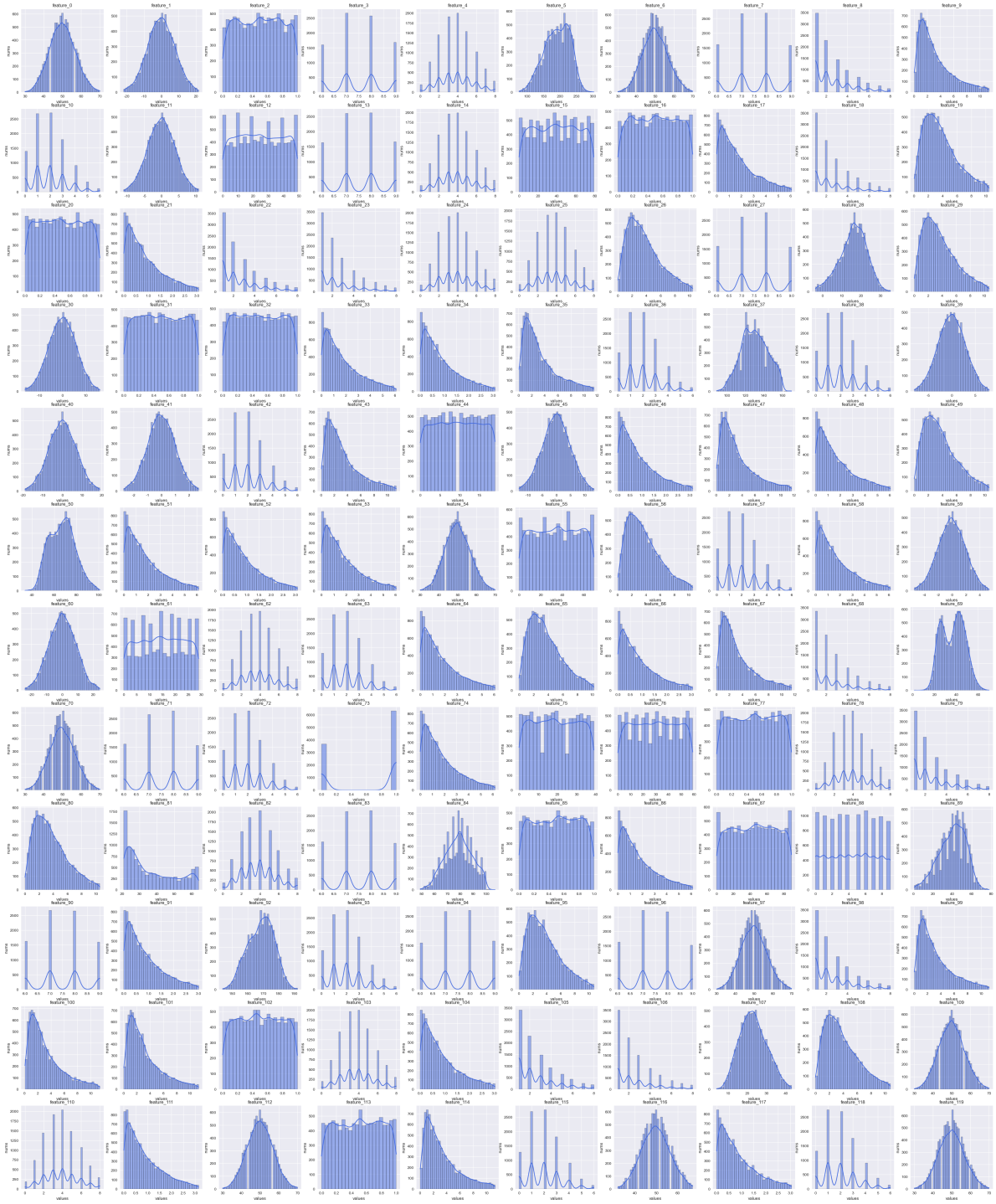
分析可知标签共4类，且分布较均匀。

4. 特征分布

```
plt.style.use('seaborn')

features_set=list(df_train_feature)
fig, axes = plt.subplots(ncols=10, nrows=12,figsize=[45,55])
for i, ax, feature in zip(range(df_train_feature.shape[1]),
axes.flat,features_set):
    ax.set_title(feature)
    ax.set_xlabel('values')
    ax.set_ylabel('nums')
    sns.histplot(df_train[feature].values,kde=True, ax=ax,color='royalblue')
plt.savefig("dataset.png")
```

在之后的数据处理中将异常值赋值为空值，然后作出每个特征的数据分布直方图如下。



可见数据分布主要包含正态分布、对数正态分布、均匀分布、两点分布、Gamma分布等。

（二）数据预处理

已知提供数据包含大量冗余随机特征、outlier数据以及Null数据。

1. **缺失值处理**

```
df_train = df_train.fillna(df_train.median())
```

（1）删除

每个特征列都含有50个左右的缺失值，但含有缺失值的行几乎不重复，若全部删除，则只剩下5486条数据，原始信息损失过多。

（2）填充

因为数据的异常值较多，极大地影响了均值，故选择用中位数进行填充。

2. **异常值处理**

```
from collections import Counter #计算每个值的个数

def detect_outliers(df,n,features):
    outlier_indices = []
    for col in features:
        Q1 = np.percentile(df[col], 25)
        Q3 = np.percentile(df[col],75)
        IQR = Q3 - Q1
        outlier_step = 1.5 * IQR
        outlier_list_col = df[(df[col] < Q1 - outlier_step) | (df[col] > Q3
+ outlier_step )].index

        for index in outlier_list_col:
            df[col][index]=None
        outlier_indices.extend(outlier_list_col)

    outlier_indices = Counter(outlier_indices)
    multiple_outliers = list( k for k, v in outlier_indices.items() if v >=
n )

    return df,multiple_outliers

df_train , Outliers_to_drop =
detect_outliers(df_train,1,df_train.columns.difference(['label']))
# df_train_drop =
df_train.drop(Outliers_to_drop,axis=0).reset_index(drop=True)
```

因为特征数较多，不方便一个个作图排查，故采用IQR方法进行特征值处理。

（1）删除

实验中发现含有缺失值的行几乎不重复，若全部删除，则只剩下133条数据，原始信息损失过多。故选择用空值填充异常值，之后当作缺失值进行填充。

一开始选择的策略是删除含5个以上缺失值的行，然后对剩下的行用中位数条件填充缺失值。在XGBoost算法上的表现很好，acc能够达到70%以上。

（2）填充

- 填充值

```
df_train_mean = df_train.fillna(df_train.mean())
df_train_median = df_train.fillna(df_train.median())
from sklearn.impute import KNNImputer
df_train_knn =
pd.DataFrame(KNNImputer(n_neighbors=10).fit_transform(df_train),columns=
df_train.columns)
```

①平均值

对每一列的缺失值，填充当列的均值。

②中位数

对每一列的缺失值，填充当列的中位数

③KNN

填充近邻的数据，先利用KNN计算临近的k个数据，然后填充他们的均值。

将以上三种填充值代入卡方过滤进行比较（此处只展示前十个特征）。

表a：填充平均值　　表b：填充中位数

Feature	Score	Feature	Score
feature_73	3.622972	feature_73	3.811723
feature_15	2.852665	feature_15	2.850349
feature_20	1.356439	feature_20	1.356109
feature_88	1.273343	feature_33	1.316590
feature_12	1.267401	feature_88	1.274713
feature_81	1.252109	feature_12	1.267853
feature_2	1.247899	feature_81	1.248780
feature_100	1.222590	feature_2	1.248270
feature_29	1.203006	feature_100	1.188736
feature_33	1.158411	feature_29	1.154131

表c：填充KNN(k=10)　表d：填充KNN(k=5) 表e：填充KNN(k=2)

Feature	Score		Feature	Score	Feature	Score
feature_73	3.784566	73	feature_73	3.796289	feature_73	3.726244
feature_15	2.852730	15	feature_15	2.834556	feature_15	2.881308
feature_20	1.361944	20	feature_20	1.340937	feature_33	1.367402
feature_33	1.289462	100	feature_100	1.314325	feature_100	1.358380
feature_2	1.270165	88	feature_88	1.290881	feature_88	1.349865
feature_12	1.269330	12	feature_12	1.274839	feature_20	1.326153
feature_88	1.262992	2	feature_2	1.260974	feature_12	1.284616
feature_100	1.218576	33	feature_33	1.254825	feature_2	1.248098
feature_81	1.180651	29	feature_29	1.173721	feature_29	1.229991
feature_29	1.169993	81	feature_81	1.167820	feature_83	1.227103

可见三种填充方法对特征与标签的相关性影响不大，填充中位数略好，选择填充中位数。

- 填充方法

①统一填充

每一列的缺失值填充相同的值。

②条件填充

```
df_train_con_median= df_train
for column in
list(df_train_con_median.columns[df_train_con_median.isna().sum() > 0]):
    median = df_train_con_median.groupby(['label'])[column].median()
    df_train_con_median = df_train_con_median.set_index(['label'])

 df_train_con_median[column]=df_train_con_median[column].fillna(median)
    df_train_con_median = df_train_con_median.reset_index()
```

每一列的缺失处填充的值从与该对象具有 label 的对象中取得。

综上所述，选择填充缺失值。常用的是均值填充，但由前述分析可知数据中离谱的outlier影响了均值，故选择用中位数填充。对于填充方法，选择条件填充，即填充和缺失样本具有相同标签的样本的中位数。

Feature	Score
feature_7	18.696107
feature_83	14.502335
feature_27	13.176440
feature_3	12.798772
feature_71	12.347897
feature_90	10.760328
feature_13	10.631434
feature_96	9.748034
feature_73	3.811723
feature_15	2.888730
feature_33	1.475745
feature_100	1.424703
feature_20	1.388104
feature_88	1.327696
feature_12	1.275835

可见特征与标签的相关性增强了，但选出的特征和之前选出的特征差别较大。统计每个特征的缺失值的数量，升序排列如下，可见选出的特征大多是缺失值数量多的，即是填充的条件中位数作为重要依据，增强了特征与标签的相关性，但是可能掩盖了原本有用的特征。

	num
count	0
feature_83	8488
feature_13	8510
feature_3	8519
feature_90	8533
feature_96	8538
feature_27	8546
feature_7	8549
feature_71	8576
feature_94	8597
feature_99	9166
feature_101	9170
feature_35	9187
feature_100	9188
feature_114	9191
feature_9	9201

```
xgbc = xgb.XGBClassifier(booster='gbtree',objective='multi:softmax',num_class=4,gamma=10,max_depth=5,eta=1)

scores = cross_validate(xgbc,X_c_med, y_c_med, cv=5,scoring=('accuracy'),return_train_score=True)
print(scores)
print(np.mean(scores['fit_time']))
print(np.mean(scores['train_score']))
print(np.mean(scores['test_score']))
```

```
{'fit_time': array([ 8.25424838, 10.6051476 , 10.74145889, 11.81823254, 12.16504502]), 'score_time': array([0.01396918, 0.01695395, 0.01408386, 0.01377702,
0.01404524]), 'test_score': array([0.7845, 0.7495, 0.773 , 0.7575, 0.766 ]), 'train_score': array([0.830375, 0.8055  , 0.8385  , 0.81475 , 0.8185  ])}
10.716826486587525
0.8215250000000001
0.7661
```

实验发现条件填充后的数据使用XGBoost能在自己划分的训练集和测试集上均取得70%以上的acc，但是应用到未条件填充的数据上时效果并不好。而test_feature是无法条件填充的，故不采用这个方法。

（3）部分删除后填充

```
df_train_drop = df_train.dropna(axis=0, thresh=115)
df_train_drop_median= df_train_drop.fillna(df_train_drop.median())
```

考虑到含缺失值太多的行被填充后就变得“普通”，尝试删除含缺失值超过5个以上的行，再进行填充。得到的特征与标签的相关性如下，效果并不太好。故不采用这个方法。

Feature	Score
feature_73	3.329589
feature_15	2.396067
feature_33	2.385988
feature_88	1.772811
feature_20	1.680228
feature_81	1.305162
feature_12	1.288869
feature_76	1.286406
feature_2	1.089799
feature_100	1.086633

综上所述，选择用IQR法检测异常值，然后赋值为该特征的中位数。

3. **独特率分析**

```
df_train_median.nunique()
```

```
feature_0      41
feature_1    9835
feature_2    9913
feature_3       4
feature_4       9
feature_5     216
feature_6      41
feature_7       4
feature_8       8
feature_9    9156
```

由于篇幅原因，仅展示前十个特征的不同值的个数。分析可知，变量均为float或int型，故不好以一个比例来剔除独特率高或者低的特征，对其分析可知，没有全部值都相同的特征。

4. **归一化**

```
a_X_med=preprocessing.MinMaxScaler().fit_transform(X_med)
df_X_med = pd.DataFrame(a_X_med, columns=X_med.columns)
```

为了统一量纲、方便比较距离、加快求解速度等，对数据进行归一化或标准化。

因为接下来的卡方过滤需要使用非负值，故选择对数据进行归一化，即

$$X = (X - X_{min})/(X_{max} - X_{min})$$

5. **相关性过滤**

```
from sklearn.feature_selection import SelectKBest,chi2
# from sklearn.feature_selection import mutual_info_classif as MIC
model_med = SelectKBest(score_func=chi2, k=19)
# model_med = SelectKBest(score_func=MIC, k=10)
fit_med = model_med.fit(df_X_med,y_med)
```

```
df_scores = pd.DataFrame(fit_med.scores_)
df_columns = pd.DataFrame(df_train_feature.columns)
# model.get_support(indices=True)
featureScores = pd.concat([df_columns,df_scores],axis=1)
featureScores.columns = ['Feature','Score']
print(featureScores.nlargest(30,'Score'))
# X_med_sel = model_med.transform(X_med)
select_list_med=featureScores.nlargest(20,'Score')['Feature'].tolist()
X_med_sel = df_X_med[select_list_med]
```

在sklearn当中们有三种常用的方法来评判特征与标签之间的相关性：卡方，F检验，互信息。其中卡方过滤是专门针对分类问题的相关性过滤，而F检验、互信息即可以做回归也可以做分类。又由于F检验只能够找出线性关系，而互信息法可以找出任意关系。故实验中对卡方和互信息法选择出的特征进行分析。

各挑选出相关性排名前30的特征如下，其中左为卡方过滤，右为互信息。

Feature	Score	Feature	Score
feature_73	3.811723	feature_18	0.016307
feature_15	2.850349	feature_49	0.014098
feature_20	1.356109	feature_77	0.013516
feature_33	1.316590	feature_97	0.012872
feature_88	1.274713	feature_34	0.011665
feature_12	1.267853	feature_9	0.010033
feature_81	1.248780	feature_67	0.010003
feature_2	1.248270	feature_106	0.009715
feature_100	1.188736	feature_99	0.009696
feature_29	1.154131	feature_33	0.009478
feature_22	1.083303	feature_52	0.009214
feature_79	1.054844	feature_109	0.008265
feature_7	1.036563	feature_29	0.007782
feature_49	0.927287	feature_62	0.007467
feature_17	0.905437	feature_61	0.007082
feature_82	0.870853	feature_35	0.006958
feature_110	0.867254	feature_116	0.006764
feature_56	0.827715	feature_55	0.006700
feature_76	0.818013	feature_90	0.006285
feature_31	0.810138	feature_3	0.006201
feature_106	0.805367	feature_76	0.006155
feature_75	0.787936	feature_72	0.005957
feature_65	0.744519	feature_114	0.005865
feature_48	0.741592	feature_5	0.005205
feature_57	0.720314	feature_87	0.004904
feature_117	0.719669	feature_65	0.004861
feature_8	0.700337	feature_103	0.004669
feature_61	0.693998	feature_8	0.004420
feature_23	0.692807	feature_21	0.004333
feature_64	0.683455	feature_17	0.004256

可见两种方法取出来的特征基本不重合。卡方过滤选出的特征间差异性较大，互信息选出的特征间差异性较小，故选择卡方过滤选出前20个特征进行训练。

6. 其他

实验中还尝试过PCA等数据降维方法，但由于大多数特征是随机生成的，降维时考虑入了太多无用信息，故最终的结果也不好，不采用。

（三）数据集划分

```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X_med_sel, y_med ,
test_size=0.2,stratify=y_med)
```

在二分类扩展为多分类时，因为本身时间消耗已较大，故使用留出法。采用train_test_split划分数据集，取test_size=0.2，即训练集：测试集=4：1。取stratify=y_med，进行分层采样。

```
from sklearn.model_selection import cross_validate
# 以SVM为例
svmc =
svm.SVC(decision_function_shape='ovo',kernel='rbf',C=2,gamma='auto',tol=0.01,max
_iter=3000)
scores = cross_validate(svmc,X_med_sel, y_med, cv=5,scoring=
('accuracy'),return_train_score=True)
print(scores)
print(np.mean(scores['fit_time']))
print(np.mean(scores['train_score']))
print(np.mean(scores['test_score']))
```

在其他模型训练时，使用交叉验证法，减少数据的偶然性等带来的误差。采用cross_validate，取cv=5，进行五折交叉验证。

（四）训练模型与调参

以下调参时大多以0.01、0.03、0.1、0.3、1、3、10.....这样近似3倍的规律粗调，找到最高的acc对应的参数，然后在附近微调。具体代码见`main.ipynb`。

1.线性回归

```
# OvO 4个类别两两配对
y_test_pred_list=[]
train_data = pd.concat([pd.DataFrame(X_train),pd.DataFrame(y_train)],axis=1)
for i in range(0,3):
    for j in range(i+1,4):
        df_train_i = train_data.loc[train_data['label'] == i]
        df_train_j = train_data.loc[train_data['label'] == j]
        df_train_ij = pd.concat([df_train_i,df_train_j],axis=0)
        # 将标签转换为0/1
        df_train_ij = df_train_ij.replace({"label":{i:0,j:1}})
        X_ij = df_train_ij.iloc[:,:-1]
        y_ij = df_train_ij.iloc[:,-1]

        # 调用自己的Logistic Regression
        lc_ij=MyLogisticRegression(penalty="l1", gamma=10, fit_intercept=True)
        iters, cost_list = lc_ij.fit(X_train,y_train,lr=1e-3,tol=1e-
1,max_iter=2000)
        lc_ij.fit(X_ij,y_ij)
        y_pred_ij = lc_ij.predict(X_ij)
        print(accuracy_score(y_ij, y_pred_ij))
        y_test_pred_ij = lc_ij.predict(X_test)
        y_test_pred_list.append(y_test_pred_ij)
```

```
# 将标签转换回来
for i,x in enumerate(y_test_pred_list[1]):
    if x==1:
        y_test_pred_list[1][i]=2
...
for i,x in enumerate(y_test_pred_list[5]):
    if x==0:
        y_test_pred_list[5][i]=2
    elif x==1:
        y_test_pred_list[5][i]=3

# 投票法选出最终结果
sum_test = [0,0,0,0]
y_test_pred=[]
for j in range(0,len(y_test)):
    max = 0
    label = 0
    for i in range(0,6):
        pred = y_test_pred_list[i][j]
        sum_test[pred]+=0
        for t,x in enumerate(sum_test):
            if x>=max:
                max = x
                label = t
    y_test_pred.append(label)
print(accuracy_score(y_test, y_test_pred))
```

在之前的实验中写的是二分类，这次实验使用OvO扩展为多分类。

实验中发现LogisticRegression模型较简单，收敛较快，调参起到的作用不大，最终的acc为0.25左右，故不多作赘述。

2.决策树

sklearn.tree模块包括决策树分类器（DecisionTreeClassifier）与决策树回归器（DecisionTreeRegressor），本次实验为四分类问题，故选择DecisionTreeClassifier。参数如下：

User guide: See the Decision Trees section for further details.

- criterion：{“gini”, “entropy”, “log_loss”}, default=”gini”

  衡量划分质量的函数。

criterion	训练集平均acc	测试集平均acc
gini	0.7180	0.2499
entropy	0.7183	0.2554
log_loss	0.7180	0.2553

  三种函数的预测结果类似，其中gini指数较低。

- splitter：{“best”, “random”}, default=”best”

  用于在每个节点选择划分的策略。

splitter	训练集平均acc	测试集平均acc
best	0.7183	0.2554
random	0.5955	0.2512

'best'略优于'random'。

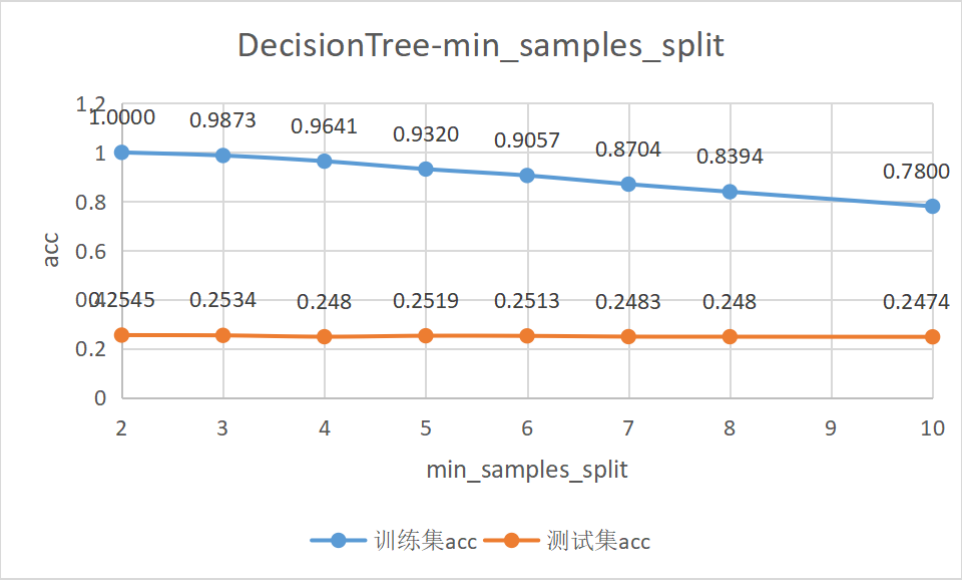
- max_depth：int, default=None

  树的最大深度。

  取为None，将节点划分直到所有树叶都为pure，或者直到所有树叶包含的样本少于min_samples_split。

- min_samples_split：int or float, default=2

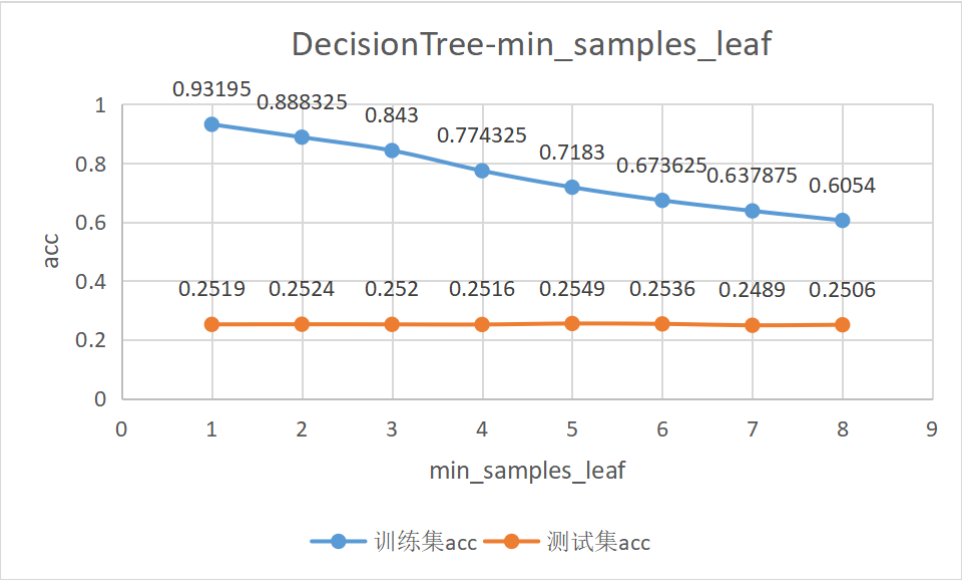
  拆分内部节点所需的最小样本数。



随着min_samples_split的减少，模型逐渐过拟合，训练集acc增加，测试集acc变化不大。

- min_samples_leaf： int or float, default=1

  叶节点所需的最小样本数。



随着min_samples_leaf的减少，模型逐渐过拟合，训练集acc增加，测试集acc变化不大。

- max_leaf_nodes：int, default=None

  以最佳优先方式生长具有max_leaf_nodes的树。

max_leaf_nodes	训练集平均acc	测试集平均acc
10	0.2768	0.2537
20	0.2912	0.2558
50	0.3179	0.2549
None	0.7181	0.2544

随着max_leaf_nodes的增加，模型逐渐过拟合，训练集acc增加，测试集acc变化不大。

3.神经网络

sklearn.neural_network包括多层感知机分类器（MLPClassifier）与多层感知机回归器（MLPRegressor），本次实验为四分类问题，故选择MLPRegressor。

- activation：{'identity'，'logic'，'tanh'，'relu'} ，默认值='relu'

  激活函数。

activation	训练集平均acc	测试集平均acc
identity	0.2837	0.2638
logistic	0.2553	0.2545
relu	0.3085	0.2617
tanh	0.3067	0.2588

四种激活函数训练出的acc近似，identity与relu略高，但考虑到identity只适合潜在行为是线性（与线性回归相似）的任务。当存在非线性，单独使用该激活函数是不够的。故选择relu作为激活函数。

- solver：{‘lbfgs’, ‘sgd’, ‘adam’}, default=’adam’

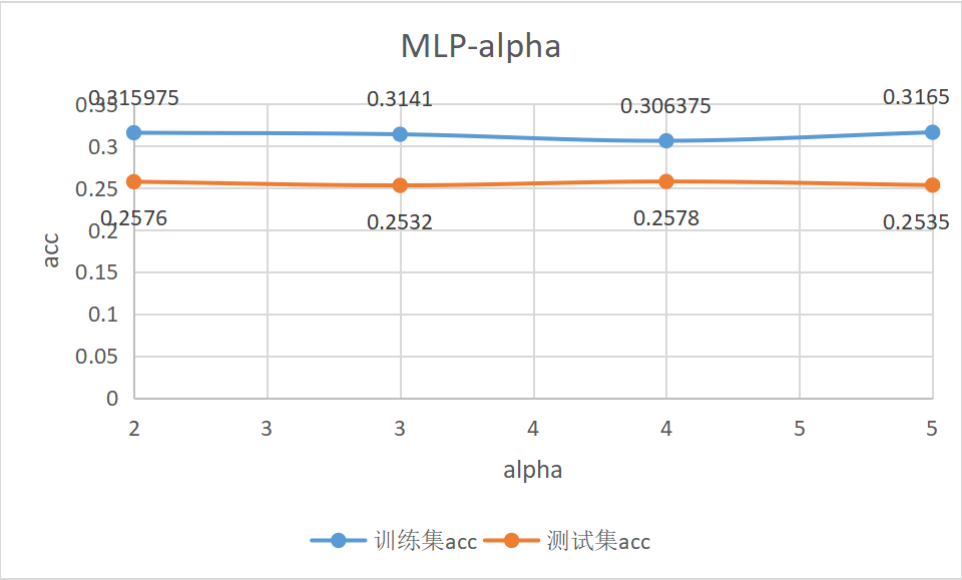
  权重优化的求解器。

solver	训练集平均acc	测试集平均acc
lbfgs	0.3237	0.2478
sgd	0.2592	0.2486
adam	0.3085	0.2617

'adam'的效果较好，选择'adam'。

- batch_size：int, default=’auto’

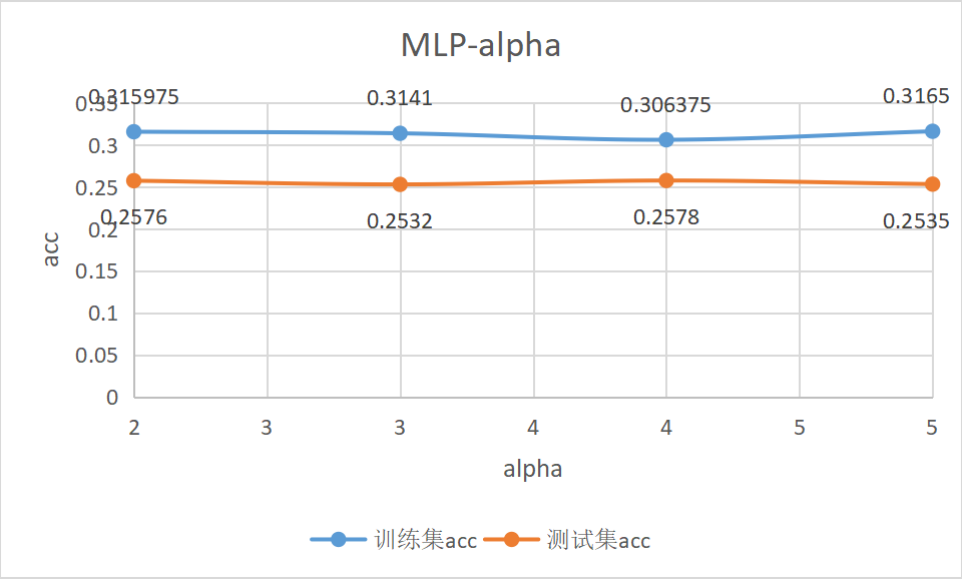
  随机优化器的minibatches大小。



可见该参数的变化对acc的影响不大。

- alpha： float, default=0.0001

  L2正则化项的强度。



可见该参数的变化对acc的影响不大。

- hidden_layer_sizes：array-like of shape(n_layers - 2,), default=(100,)

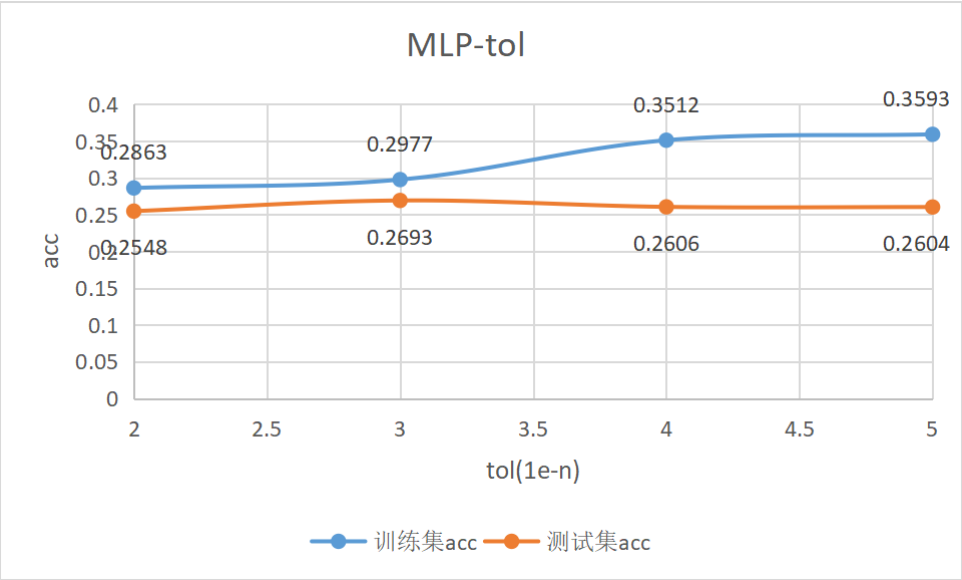
  第i个元素表示第i个隐藏层中的神经元数量。

hidden_layer_sizes	训练集平均acc	测试集平均acc
8	0.3014	0.2606
16	0.3284	0.2605
(8,8,)	0.3060	0.2602
(16,16,)	0.3512	0.2606
(32,32,)	0.4606	0.2468
(8,8,8,)	0.3123	0.2573
(16,16,16,)	0.3796	0.2569

随着网络层数和每层神经元个数的增加，模型逐渐变复杂，acc先上升后下降（出现了过拟合）。

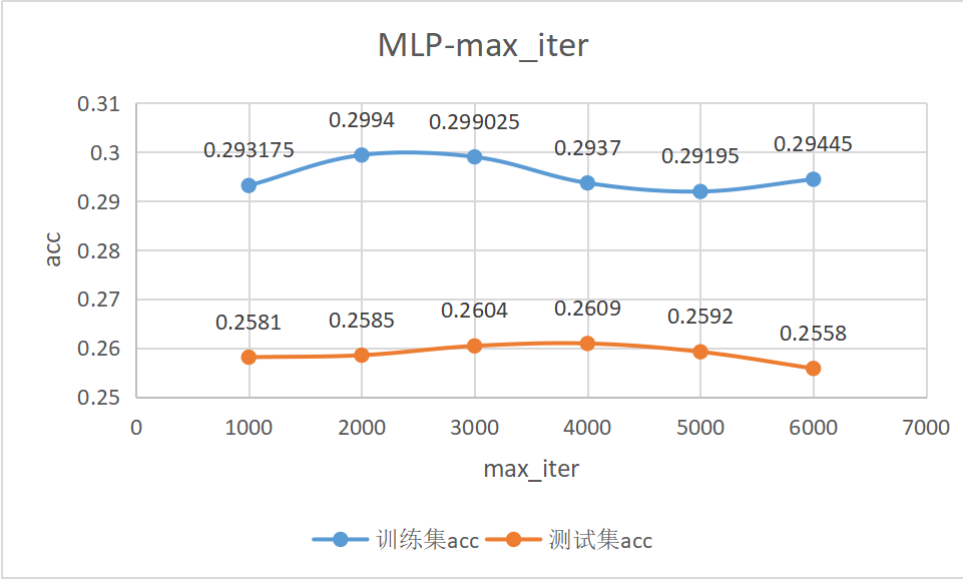
- tol：float, default=1e-4

  tolerance。



- max_iter：int, default=200

  最大迭代次数。



可见MLP迭代收敛较快，max_iter对acc的影响不大。

4.支持向量机

实验中对比发现自己写的SVM没有sklearn库中的SVM性能好，故使用后者进行调参与预测。选择sklearn.svm中的C-Support Vector Classification，即SVC。

- decision_function_shape：{'ovo', 'ovr'}, default='ovr'

  二分类扩展为多分类的方法。

decision_function_shape	训练集平均acc	测试集平均acc	平均训练时间(s)
ovo	0.2936	0.2646	11.68
ovr	0.2936	0.2646	11.33

可见二者的预测性能与训练时间开销均近似。

- kernel：{'linear', 'poly', 'rbf', 'sigmoid'} or callable, default='rbf'

  核函数。

kernel	训练集平均acc	测试集平均acc
linear	0.2867	0.2658
poly	0.2587	0.2504
rbf	0.2936	0.2646
sigmoid	0.2832	0.2616

- gamma：{'scale', 'auto'} or float, default='scale'

  'rbf'、'poly'和'sigmoid'的核系数。

  'scale'： 1 / (n_features * X.var())；

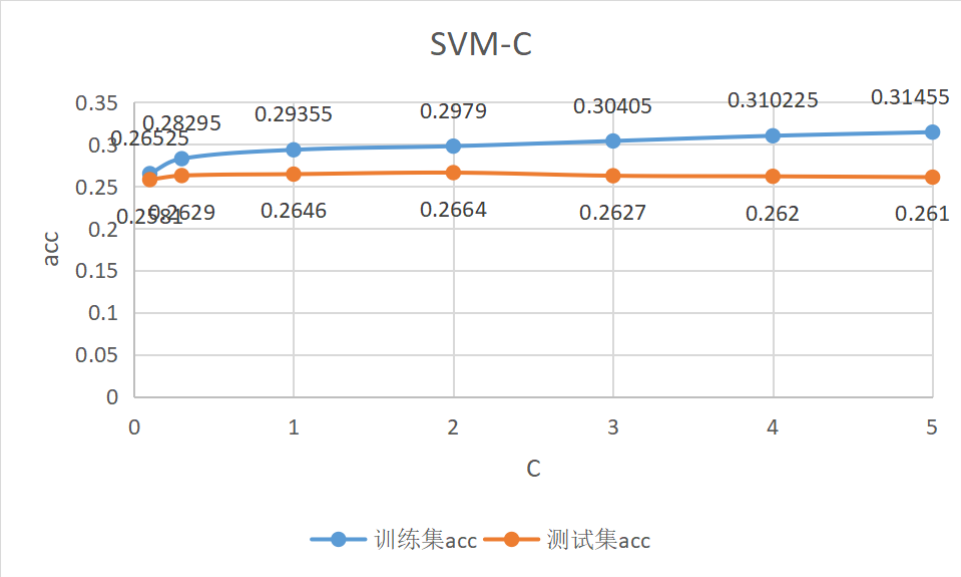
  'auto'： 1 / n_features。

gamma	训练集平均acc	测试集平均acc
scale	0.5432	0.2595
auto	0.2979	0.2661

gamma='scale'时出现了过拟合，gamma = 'auto'时测试集acc较高，取gamma = 'auto'。

- C：float, default=1.0

  正则化参数。正则化的强度与C成反比。



随着C的增加，训练集acc逐渐上升，测试集acc先略上升，后略下降。逐渐出现过拟合。取C=2。

- tol：float, default=1e-3

  迭代停止的tolerance。

tol	训练集平均acc	测试集平均acc
0.3	0.2983	0.2641
0.1	0.2933	0.2657
0.01	0.2938	0.2650
0.001	0.2936	0.2646
0.0001	0.2935	0.2646
0.00001	0.2935	0.2646

  随着tol的增加，acc变化不大。其中tol=0.1时测试集acc较高，缓解了过拟合。

- max_iter：int, default=-1

  最大迭代次数，-1表示无限制。

max_iter	训练集平均acc	测试集平均acc
2000	0.2985	0.2635
3000	0.2988	0.2655
4000	0.2988	0.2655
5000	0.2988	0.2655

  max_iter为2000时提示了ConvergenceWarning，算法还未收敛。max_iter取3000及以上时，算法以及收敛，acc不变。可取max_iter=3000。

5.XGBoost

- objective：default='reg:linear'

  损失函数。'binary:logistic'：二分类的逻辑回归，返回概率；'multi:softmax'：使用softmax的多分类器，返回类别。此时还需设置参数：num_class(类别数目)；'multi:softprob'：与multi:softmax参数相同，但返回概率。

  对于本次实验的四分类任务，选择 objective='multi:softmax',num_class=4。

- booster：default='gbtree'

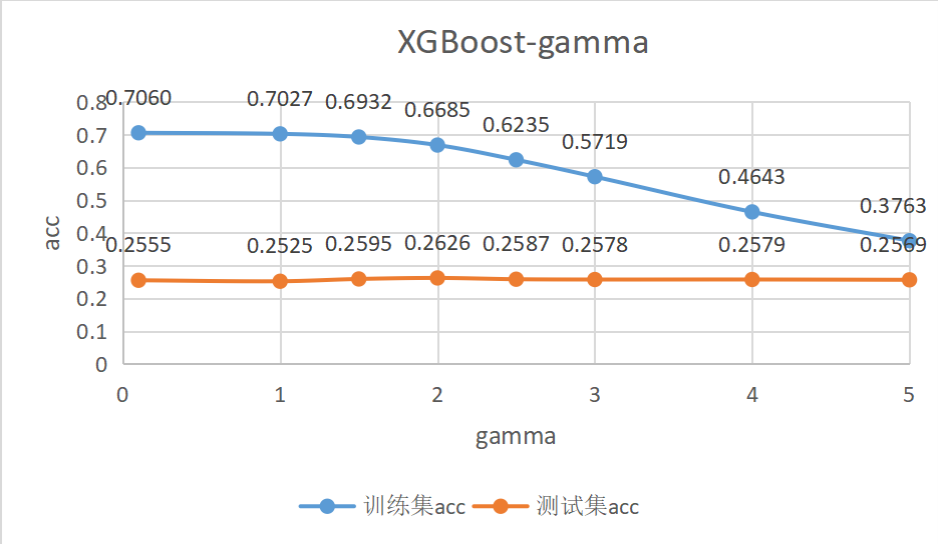
  基模型，'gblinear'：线性模型；'gbtree'：基于树的模型,default='gbtree'。

booster	训练集平均acc	测试集平均acc
gblinear	0.2844	0.2668
gbtree	0.6685	0.2626

  gbtree更复杂些，在训练集上的acc更高，但是训练集上二者的acc差不多。

- gamma：range: [0,∞]，default=0

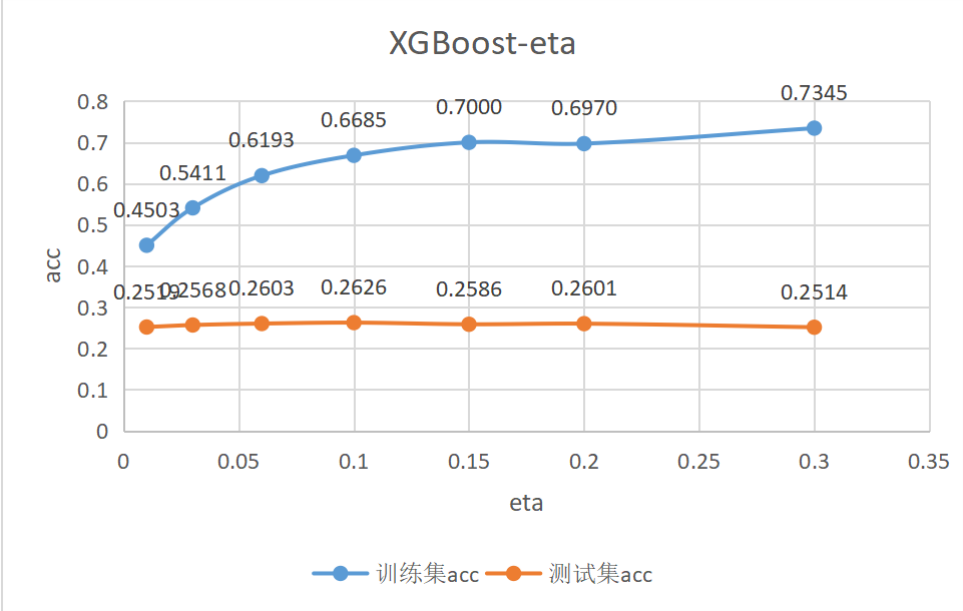
在树的叶节点上进行进一步分区所需的最小损失减少。越大，算法就越保守。



随着gamma的增大，训练集acc逐渐下降，测试集acc变化较小。

- eta：range: [0,1]，default=0.3

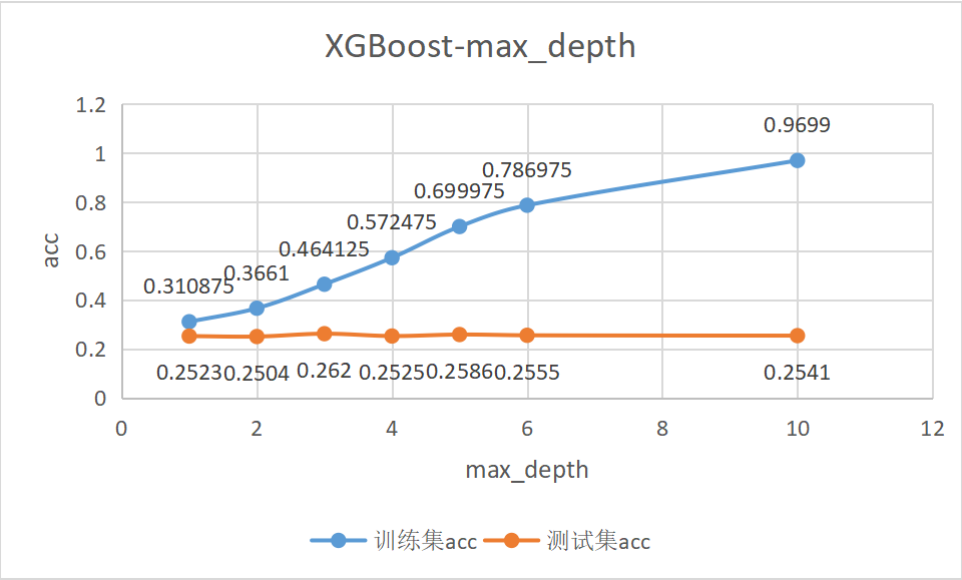
  类似学习率。



随着eta的增加，训练集acc增加，测试集acc变化不大。

- max_depth：　range: [1,∞]，default=6

  树的最大深度，增加该值将使模型更复杂，可能过拟合。



随着max_depth的增加，训练集acc增加，测试集acc先略增加再略下降，出现了过拟合。

四、实验结果

模型	测试集平均acc
MyLogisticRegression(penalty="l1", gamma=10, fit_intercept=True,lr=1e-3,tol=1e-1,max_iter=2000)	0.2510
MLPClassifier(activation = 'relu',solver = 'adam',alpha=1e-5,hidden_layer_sizes=(16,16,),batch_size=100,max_iter=3000,tol=1e-3)	0.2609
svm.SVC(decision_function_shape='ovo',kernel='rbf',C=2,gamma='auto',tol=0.01, max_iter=3000)	0.2661
DecisionTreeClassifier(criterion='entropy',max_depth=None, min_samples_split=5,min_samples_leaf=5,max_leaf_nodes=None,splitter='best')	0.2558
xgb.XGBClassifier(booster='gblinear',objective='multi:softmax',num_class=4, gamma=2,max_depth=5,eta=0.01)	0.2668

综上所述，5种分类器的表现结果相差不大，其中SVM和XGBoost的预测效果较高，但XGBoost的基模型为'gblinear'，较简单，偶然性较大。故选择SVM对test_feature进行预测。

```
# 对test_feature预处理
df_test_feature = df_test_feature.fillna(df_test_feature.median())
df_test_feature = detect_outliers(df_test_feature,1,df_test_feature.columns)
df_test_median = df_test_feature.fillna(df_test_feature.median())
a_X_test=preprocessing.MinMaxScaler().fit_transform(df_test_median)
df_X_test = pd.DataFrame(a_X_test, columns=df_test_median.columns)

# 特征选择，与train_feature相同
model_med = SelectKBest(score_func=chi2, k=20)
fit_med = model_med.fit(df_X_med,y_med)
df_scores = pd.DataFrame(fit_med.scores_)
df_columns = pd.DataFrame(df_train_feature.columns)
featureScores = pd.concat([df_columns,df_scores],axis=1)
```

```
featureScores.columns = ['Feature','Score']
print(featureScores.nlargest(30,'Score'))
select_list_med=featureScores.nlargest(20,'Score')['Feature'].tolist()
X_med_sel = df_X_med[select_list_med]
X_test_sel = df_X_test[select_list_med]

# 训练
svmc =
svm.SVC(decision_function_shape='ovo',kernel='rbf',C=2,gamma='auto',tol=0.01,max
_iter=3000)
svmc.fit(X_med_sel, y_med)
# 预测
y_train_pred = svmc.predict(X_med_sel)
y_test_pred = svmc.predict(X_test)

print(accuracy_score(y_train_pred,y_med))

# 生成test_label
y_test_pred = pd.DataFrame(y_test_pred,columns=['label'])
y_test_pred.to_csv("test_label.csv",index = False)
```

五、实验分析

本次实验因为提供数据包含大量冗余随机特征、outlier数据以及Null数据，故在数据预处理时尝试了很多方法，但是效果好像都不太好。

调参时也尽量做到"充分"调参，不过因为数据的原因，调参的效果不明显。

实验时把 acc 作为主要评价指标，因为数据集不大，故训练时间均较短，不额外比较。

在比较二分类扩展为多分类的两种方法（OvO与OvR）时将acc与训练时间都作为了评价指标，因为类别数不多，故二者的训练时间相差不大。acc也近似。

几种分类器的预测效果差别不大。

对SVM和XGBoost的预测效果进行交叉验证 t 检验，即成对 t 检验。原假设为二者测试集上的acc相同。

k	SVM_acc	XGBoost_acc
1	0.278	0.2895
2	0.2535	0.2585
3	0.2685	0.2725
4	0.2745	0.264
5	0.256	0.2495

配对样本统计

		平均值	个案数	标准 偏差	标准 误差平均值
配对 1	SVM_acc	.266100	5	.0109396	.0048923
	XGBoost_acc	.266800	5	.0151970	.0067963

SVM_acc的均值为0.2661，XGBoost_acc的均值为0.2668，差异不大。

配对样本相关性

		个案数	相关性	显著性
配对 1	SVM_acc & XGBoost_acc	5	.811	.095

相关系数检验的概率P-值为0.095，在显著性水平$\alpha = 0.05$时，SVM_acc与XGBoost的线性相关程度不明显。

配对样本检验

		配对差值					t	自由度	Sig.（双尾）
		平均值	标准 偏差	标准 误差平均值	差值 95% 置信区间 下限	差值 95% 置信区间 上限			
配对 1	SVM_acc - XGBoost_acc	-.0007000	.0089903	.0040206	-.0118629	.0104629	-.174	4	.870

SVM_acc与XGBoost_acc的平均差异为0.0007，t检验统计量观测值对应的双侧概率P-值为0.870，在显著性水平$\alpha = 0.05$时，应接受原假设，即SVM_acc与XGBoost_acc不存在显著差异。

因预测效果不太好，故不再做别的假设检验。