课程说明：

小伙伴好呀~欢迎来到《2021机器学习实战训练营》（第二期）试学体验课！我是课程主讲老师，九天。

本次体验课为期三天（10月21-23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

本期公开课将围绕一项kaggle竞赛案例进行深度剖析，并据此讨论算法竞赛中机器学习的一般建模流程，以及当前机器学习进行预测时最为有效的技术手段，也就是特征工程方法和集成算法的相关应用，对算法和竞赛感兴趣的小伙伴，欢迎积极参与讨论哦~

课程资料领取/数据技术交流/付费课程信息，扫码添加客服“小可爱”获取哦~

另外，《机器学习实战训练营》（第二期）本月25号即将开课，十八周80+课时体系大课限时半价，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

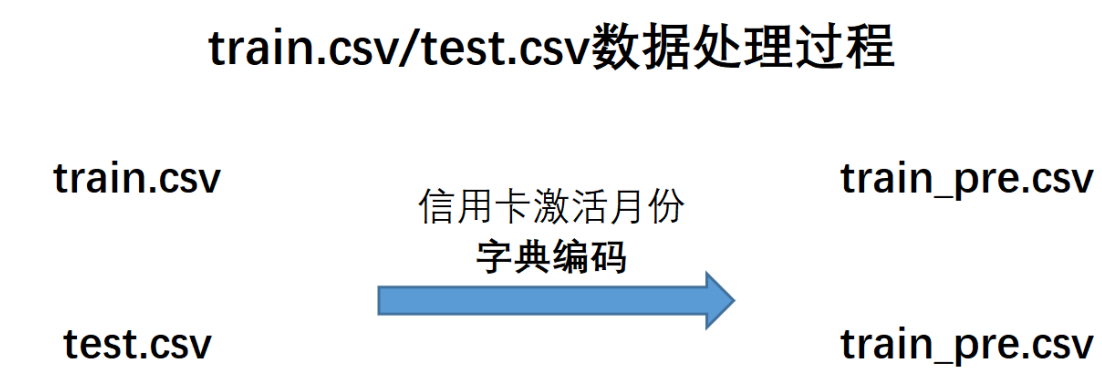
【Kaggle】Elo Merchant Category Recommendation

竞赛案例解析公开课

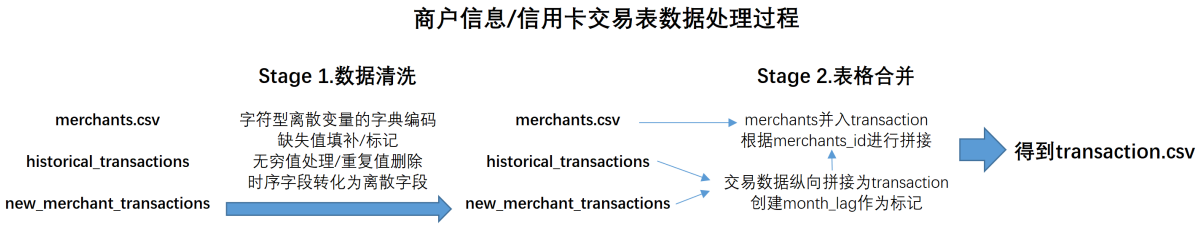
Day 3.特征工程与模型训练

在经历了漫长的数据解读、探索与清洗之后，接下来，我们将进入到特征工程与算法建模的环节。并在本小节的结尾，得出最终的预测结果。

在此前的内容中，我们最终得到了train.csv、test.csv和transaction.csv三张表。首先我们简单回顾下这三张数据表的构建过程，首先，目前得到的训练集和测试集都是由原始训练集/测试集将时间字段处理后得到：



而transaction数据集则相对复杂，该数据集是有一张商户数据merchants.csv和两张交易数据表处理后合并得到，该过程如下所示：



接下来，我们就依据这三张表进行后续操作。

一、特征工程

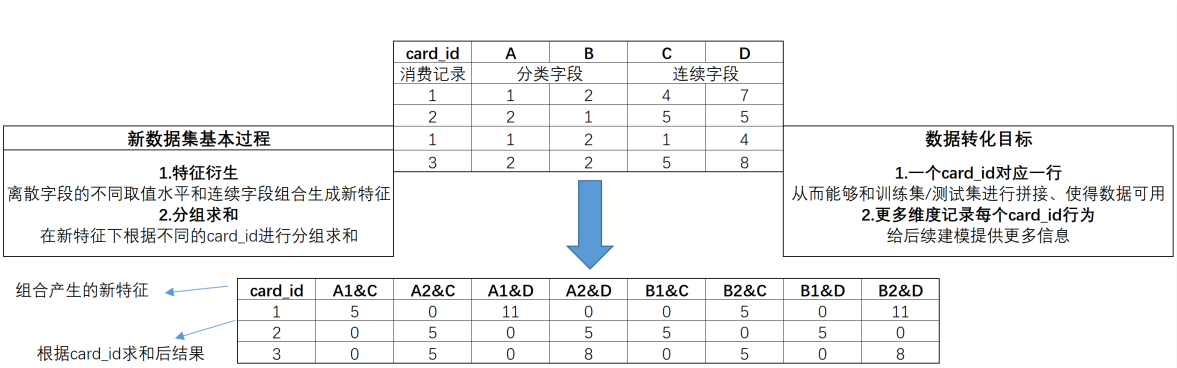
首先需要对得到的数据进一步进行特征工程处理。一般来说，对于已经清洗完的数据，特征工程部分核心需要考虑的问题就是特征创建（衍生）与特征筛选，也就是先尽可能创建/增加可能对模型结果有正面影响的特征，然后再对这些进行挑选，以保证模型运行稳定性及运行效率。当然，无论是特征衍生还是特征筛选，其实都有非常多的方法。此处为了保证公开课提供的思路和方法具有通用性，此处列举两种特征衍生的方法，即创建通用组合特征与业务统计特征；并在特征创建完毕后，介绍一种基础而通用的特征筛选的方法：基于皮尔逊相关系数的Filter方法进行特征筛选。这些方法都是非常通用且有效的方法，不仅能够帮助本次建模取得较好的成果，并且也能广泛适用到其他各场景中。

1.通用组合特征创建

1.1 通用组合特征的创建方法

首先是尝试创建一些通用组合特征。

所谓通用组合特征，指的是通过统计不同离散特征在不同取值水平下、不同连续特征取值之和创建的特征，并根据card_id进行分组求和。具体创建过程我们可以如下简例来进行理解：



通过该方法创建的数据集，不仅能够尽可能从更多维度表示每个card_id的消费情况，同时也能够顺利和训练集/测试集完成拼接，从而带入模型进行建模。相关过程我们可以借助Python中的字典对象类型来进行实现，上述简例实现过程如下：

In [1]:
```
import gc
import time
import numpy as np
import pandas as pd
from datetime import datetime
```

In [13]:
```
# 借助字典创建DataFrame
d1 = {'card_id':[1, 2, 1, 3],
      'A':[1, 2, 1, 2],
      'B':[2, 1, 2, 2],
      'C':[4, 5, 1, 5],
      'D':[7, 5, 4, 8],}

t1 = pd.DataFrame(t1)
t1
```

Out[13]:

	card_id	A	B	C	D
0	1	1	2	4	7
1	2	2	1	5	5
2	1	1	2	1	4
3	3	2	2	5	8

In [10]:
```
# 标注特征类别
numeric_cols = ['C', 'D']
category_cols = ['A', 'B']
```

In [11]:
```
# 创建一个以id为key、空字典为value的字典
features = {}
card_all = t1['card_id'].values.tolist()
for card in card_all:
    features[card] = {}
```

In [12]:
```
features
```

Out[12]:
```
{1: {}, 2: {}, 3: {}}
```

In [18]:
```
# 所有字段名称组成的list
columns = t1.columns.tolist()
columns
```

Out[18]: ['card_id', 'A', 'B', 'C', 'D']

In [19]:
```
# 其中card_id在list当中的索引值
idx = columns.index('card_id')
idx
```

Out[19]: 0

In [31]:
```
# 离散型字段的索引值
category_cols_index = [columns.index(col) for col in category_cols]
category_cols_index
```

Out[31]: [1, 2]

In [32]:
```
# 连续型字段的索引值
numeric_cols_index = [columns.index(col) for col in numeric_cols]
numeric_cols_index
```

Out[32]: [3, 4]

In [25]:
```
# 对离散型字段的不同取值和连续型字段两两组合
# 同时完成分组求和
for i in range(t1.shape[0]):
    va = t1.loc[i].values
    card = va[idx]
    for cate_ind in category_cols_index:
        for num_ind in numeric_cols_index:
            col_name = '&'.join([columns[cate_ind], str(va[cate_ind]), columns[num_in
            features[card][col_name] = features[card].get(col_name, 0) + va[num_ind]
```

然后查看features最终结果

In [26]:
```
features
```

Out[26]:
```
{1: {'A&1&C': 5, 'A&1&D': 11, 'B&2&C': 5, 'B&2&D': 11},
 2: {'A&2&C': 5, 'A&2&D': 5, 'B&1&C': 5, 'B&1&D': 5},
 3: {'A&2&C': 5, 'A&2&D': 8, 'B&2&C': 5, 'B&2&D': 8}}
```

能够发现，此时features就是一个已经包含了离散变量的不同取值和连续变量两两组合成新特征后在不同card_id下的分组求和结果。接下来我们将其转化为DataFrame：

In [33]:
```
# 转化成df
df = pd.DataFrame(features).T.reset_index()

# 标注所有列
cols = df.columns.tolist()

# 修改df的特征名称
df.columns = ['card_id'] + cols[1:]
df
```

Out[33]:

	card_id	A&1&C	A&1&D	B&2&C	B&2&D	A&2&C	A&2&D	B&1&C	B&1&D
0	1	5.0	11.0	5.0	11.0	NaN	NaN	NaN	NaN
1	2	NaN	NaN	NaN	NaN	5.0	5.0	5.0	5.0
2	3	NaN	NaN	5.0	8.0	5.0	8.0	NaN	NaN

至此我们就完成了在极简数据集上进行通用组合特征的创建工作。

当然，通过上述过程不难发现，这种特征创建的方式能够非常高效的表示更多数据集中的隐藏信息，不过该方法容易产生较多空值，在后续建模过程中需要考虑特征矩阵过于稀疏从而带来的问题。

1.2 基于transaction数据集创建通用组合特征

接下来，我们将上述过程应用于建模真实数据，即在此前已经清洗完的transaction数据集上来完成通用组合特征的创建工作。

> 此处需要注意的是，由于transaction数据集本身较大，尽管特征创建工作的求和部分会一定程度减少最终带入建模的数据体量，但操作transaction数据集本身就需要耗费大量的内容及一定的时间，如果要手动执行下述代码，建议至少配置32G及以上内存。当然，处理完的数据也会在课件领取地址中提供，大家也可以直接利用处理完的数据带入进行建模。

- 数据读取

  此处读取的transaction是此前创建的transaction_d_pre.csv数据集。

In [65]:
```
train = pd.read_csv('preprocess/train_pre.csv')
test =  pd.read_csv('preprocess/test_pre.csv')
transaction = pd.read_csv('preprocess/transaction_d_pre.csv')
```

- 字段类型标注

In [34]:
```
# 标注离散字段or连续型字段
numeric_cols = ['purchase_amount', 'installments']

category_cols = ['authorized_flag', 'city_id', 'category_1',
       'category_3', 'merchant_category_id','month_lag','most_recent_sales_range',
                 'most_recent_purchases_range', 'category_4',
                 'purchase_month', 'purchase_hour_section', 'purchase_day']

id_cols = ['card_id', 'merchant_id']
```

- 特征创建

In [7]:
```
# 创建字典用于保存数据
features = {}
card_all = train['card_id'].append(test['card_id']).values.tolist()
for card in card_all:
    features[card] = {}

# 标记不同类型字段的索引
columns = transaction.columns.tolist()
idx = columns.index('card_id')
```

```
category_cols_index = [columns.index(col) for col in category_cols]
numeric_cols_index = [columns.index(col) for col in numeric_cols]

# 记录运行时间
s = time.time()
num = 0

# 执行循环，并在此过程中记录时间
for i in range(transaction.shape[0]):
    va = transaction.loc[i].values
    card = va[idx]
    for cate_ind in category_cols_index:
        for num_ind in numeric_cols_index:
            col_name = '&'.join([columns[cate_ind], va[cate_ind], columns[num_ind]])
            features[card][col_name] = features[card].get(col_name, 0) + va[num_ind]
    num += 1
    if num%1000000==0:
        print(time.time()-s, "s")
del transaction
gc.collect()
```

```
142.746732711792 s
241.50783610343933 s
338.9149408340454 s
436.4667372703552 s
533.113107919693 s
629.66761469841 s
727.1969571113586 s
824.3946213722229 s
921.0717754364014 s
1017.7034878730774 s
1114.4361855983734 s
1211.2046930789948 s
1308.0264575481415 s
1404.8067374229431 s
1501.533932209015 s
1598.396145105362 s
1695.2529389858246 s
1792.5994687080383 s
1889.7299542427063 s
1987.0093190670013 s
2084.849946975708 s
2183.5546836853027 s
2281.704159259796 s
2379.819750070572 s
2478.2387039661407 s
2576.8626248836517 s
2676.383053302765 s
2777.3995122909546 s
2879.5466351509094 s
2981.8099772930145 s
3085.8226635456085 s
```

Out[7]: 0

能够发现，整体运行所需时间较长。此外，此处需要注意的是，card_id的提取并不是从transaction从提取，而是从训练集和测试集中提取，大家想想看是什么原因？

在提取完特征后，接下来即可将带有交易数据特征的合并入训练集和测试集了：

In [8]:

```
# 字典转dataframe
df = pd.DataFrame(features).T.reset_index()
del features
cols = df.columns.tolist()
df.columns = ['card_id'] + cols[1:]
```

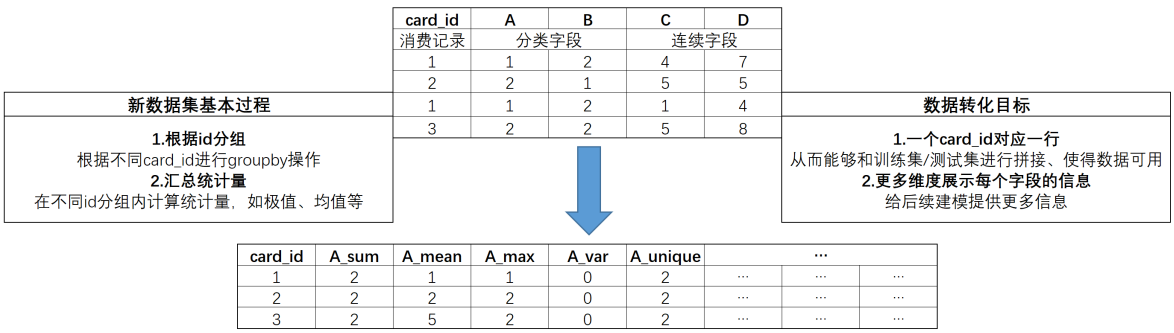
```
# 生成训练集与测试集
train = pd.merge(train, df, how='left', on='card_id')
test =  pd.merge(test, df, how='left', on='card_id')
del df
train.to_csv("preprocess/train_dict.csv", index=False)
test.to_csv("preprocess/test_dict.csv", index=False)

gc.collect()
```

至此，我们就完成了从transaction中提取通用特征的过程。简单查看数据集基本情况：

IP	IQ	IR	IS	IT	IU	IV	IW	IX	IY
purchase_month&3&	city_id&253&purchas	city_id&253&installm	merchant_category_ic	merchant_category_ic	merchant_category_ic	merchant_category_ic	merchant_category_ic	merchant_category_ic	merchant_category_ic
37	-3.46207826	5	-0.67945378	1	-0.99117878	4	-1.48179438	2	-0.56658965
0									
			-0.68995732	1	-1.13765721	4	-0.54478619	3	-0.65975403
							-0.51940641	1	
21									
0			-1.44272545	0	-0.59713855	0			
0			-0.70904099	0	-0.65607253	0			
0			-0.64187248	0	-1.23581038	0			
0									
2					-0.49661119	2			
80					-2.67604143	4			
1							-0.67177524	1	
0									
			-0.67973929	2					
62									
			-0.72436803	0	-7.47337553	0			
6									
					-4.68618299	0	-0.5966577	0	
12			-0.69732031	1	-0.63420896	1			
0									
					-0.56673992	2			
					-1.36335542	2			
3							-4.22293067	7	
1									
					-32.0133072	1			
12					-0.58732624	1			
									-0.67177524
0			-0.73188128	0					
0					-0.70925136	0			
0									
					-8.33293708	17			
					0.24409066	2			
21					-0.53868542	1			
36			-0.46290672	1					
					-0.71243698	0			

2.业务统计特征创建

当然，除了通用组合特征外，我们还可以考虑从另一个角度进行特征提取，那就是先根据card_id来进行分组，然后统计不同字段再各组内的相关统计量，再将其作为特征，带入进行建模。其基本构造特征思路如下：



该过程并不复杂，可以通过pandas中的groupby过程迅速实现。和此前特征构造的思路不同，通过该方法构造的特征，不会存在大量的缺失值，并且新增的列也将相对较少。代码实现过程如下：

- 数据读取：

In [36]:
```
transaction = pd.read_csv('preprocess/transaction_g_pre.csv')
```

- 字段类型标注

In [34]:
```
# 标注离散字段or连续型字段
numeric_cols = ['authorized_flag',  'category_1', 'installments',
       'category_3',  'month_lag','purchase_month','purchase_day','purchase_day_diff',
       'purchase_amount', 'category_2',
       'purchase_month', 'purchase_hour_section', 'purchase_day',
       'most_recent_sales_range', 'most_recent_purchases_range', 'category_4']
categorical_cols = ['city_id', 'merchant_category_id', 'merchant_id', 'state_id', 'sub
```

- 特征提取过程

In [7]:
```
# 创建空字典
aggs = {}

# 连续/离散字段统计量提取范围
for col in numeric_cols:
    aggs[col] = ['nunique', 'mean', 'min', 'max','var','skew', 'sum']
for col in categorical_cols:
    aggs[col] = ['nunique']
aggs['card_id'] = ['size', 'count']
cols = ['card_id']

# 借助groupby实现统计量计算
for key in aggs.keys():
    cols.extend([key+'_'+stat for stat in aggs[key]])

df = transaction[transaction['month_lag']<0].groupby('card_id').agg(aggs).reset_index
df.columns = cols[:1] + [co+'_hist' for co in cols[1:]]

df2 = transaction[transaction['month_lag']>=0].groupby('card_id').agg(aggs).reset_ind
df2.columns = cols[:1] + [co+'_new' for co in cols[1:]]
df = pd.merge(df, df2, how='left',on='card_id')

df2 = transaction.groupby('card_id').agg(aggs).reset_index()
df2.columns = cols
df = pd.merge(df, df2, how='left',on='card_id')
del transaction
gc.collect()

# 生成训练集与测试集
train = pd.merge(train, df, how='left', on='card_id')
test =  pd.merge(test, df, how='left', on='card_id')
del df
train.to_csv("preprocess/train_groupby.csv", index=False)
test.to_csv("preprocess/test_groupby.csv", index=False)

gc.collect()
```

执行完毕后，我们也可以简单查看数据集基本情况：

CG	CH	CI	CJ	CK
purchase_hour_section_mean_hist	purchase_hour_section_min_hist	purchase_hour_section_max_hist	purchase_hour_section_var_hist	purchase_hour_section_skew_hist
1.864978903	0	3	0.659658156	-0.609027459
2.073578595	0	3	0.80664856	-0.900811166
2.523809524	1	3	0.401858304	-0.998006085
1.967213115	0	3	0.698907104	-0.998971289
1.714285714	0	3	0.969072165	-0.25530887
1.814814815	0	3	0.618233618	-0.674940694
2.077235772	0	3	0.308296001	-0.255175126
2.263157895	0	3	1.204678363	-1.430645466
1.8	0	3	1.457142857	-0.68597415
1.971962617	0	3	0.404866867	-0.872793249
1.976744186	0	3	0.705335157	-0.565592242
2	1	3	0.666666667	0
1.806451613	0	3	1.227956989	-0.376684756
2.052631579	1	3	0.385964912	-0.025822943
2.279069767	0	3	0.825027685	-1.397690507
2.028571429	0	3	0.947454844	-0.700574901
1.723809524	0	3	1.00952381	-0.403901879
1.7	1	3	0.465822785	0.460623723
1.92	0	3	0.66	-0.86034303
1.980263158	0	3	0.634774436	-0.436026555
1.895424837	0	3	0.962676299	-0.549909847
1.923076923	1	3	0.393846154	0.049795112
2	0	3	0.714285714	-0.819149508
2.032967033	0	3	0.765567766	-0.47179565
1.960264901	0	3	0.891743929	-0.594069801
1.568181818	0	3	0.399895906	0.290958964
1.857142857	0	3	0.895008606	-0.406945773
1.533333333	0	3	1.40952381	-0.386562239
1.705882353	0	3	0.744321491	-0.237043466
1.78030303	0	3	0.630754106	0.322756891
2.641025641	0	3	0.394062078	-2.250468855
2.040358744	0	3	0.615480952	-0.918334996

3.数据合并

至此，我们即完成了从两个不同角度提取特征的相关工作。不过截至目前上述两套方案的特征仍然保存在不同数据文件中，我们需要对其进行合并，才能进一步带入进行建模，合并过程较为简单，只需要将train_dict(test_dict)与train_group(test_group)根据card_id进行横向拼接、然后剔除重复列即可，实现过程如下所示：

- 数据读取

In [2]:
```
train_dict = pd.read_csv("preprocess/train_dict.csv")
test_dict = pd.read_csv("preprocess/test_dict.csv")
train_groupby = pd.read_csv("preprocess/train_groupby.csv")
test_groupby = pd.read_csv("preprocess/test_groupby.csv")
```

- 剔除重复列

In [3]:
```
for co in train_dict.columns:
    if co in train_groupby.columns and co!='card_id':
        del train_groupby[co]
for co in test_dict.columns:
    if co in test_groupby.columns and co!='card_id':
        del test_groupby[co]
```

- 拼接特征

In [4]:
```
train = pd.merge(train_dict, train_groupby, how='left', on='card_id').fillna(0)
test = pd.merge(test_dict, test_groupby, how='left', on='card_id').fillna(0)
```

> 注，上述操作对缺失值进行了0的填补，此处缺失值并非真正的缺失值，该缺失值只是在特征创建过程没有统计结果的值，这些值从逻辑上来讲其实也都是0。因此此处缺失值填补相当于是数据补全。

- 数据保存与内存管理

In [5]:
```
train.to_csv("preprocess/train.csv", index=False)
test.to_csv("preprocess/test.csv", index=False)

del train_dict, test_dict, train_groupby, test_groupby
gc.collect()
```

Out[5]: 352

二、随机森林模型预测

在准备好了基础特征之后，终于迎来了算法建模环节。限于公开课篇幅，此处重点介绍关于集成算法中的随机森林模型的建模及优化流程，并最终展示建模结果。

In [5]:
```
train = pd.read_csv("preprocess/train.csv")
test = pd.read_csv("preprocess/test.csv")
```

- 特征选择

由于此前创建了数千条特征，若带入全部特征进行建模，势必极大程度延长模型建模时间，并且带入太多无关特征对模型结果提升有限，因此此处我们借助皮尔逊相关系数，挑选和标签最相关的300个特征进行建模。当然此处300也可以自行调整。

In [7]:
```
# 提取特征名称
features = train.columns.tolist()
features.remove("card_id")
features.remove("target")
featureSelect = features[:]

# 计算相关系数
corr = []
for fea in featureSelect:
    corr.append(abs(train[[fea, 'target']].fillna(0).corr().values[0][1]))

# 取top300的特征进行建模，具体数量可选
se = pd.Series(corr, index=featureSelect).sort_values(ascending=False)
feature_select = ['card_id'] + se[:300].index.tolist()

# 输出结果
train = train[feature_select + ['target']]
test = test[feature_select]
```

> 注意，此处可以通过皮尔逊相关系数进行特征提取的主要原因也是在于我们在特征创建的过程中，将所有特征都默认为连续性变量

- 借助网格搜索进行参数调优

接下来，我们将借助sklearn中基础调参工具—网格搜索（Gridsearch）进行参数搜索与调优。

首先导入相关包，包括均方误差计算函数、随机森林评估器和网格搜索评估器：

In [30]:
```
from sklearn.metrics import mean_squared_error
from sklearn.ensemble import RandomForestRegressor
```

```
from sklearn.model_selection import GridSearchCV
```

然后根据网格搜索的要求，我们需要根据随机森林的参数情况，有针对性的创造一个参数空间，随机森林基本参数基本情况如下：

Name	Description
criterion	规则评估指标或损失函数，默认基尼系数，可选信息熵
splitter	树模型生长方式，默认以损失函数取值减少最快方式生长，可选随机根据某条件进行划分
max_depth	树的最大生长深度，类似max_iter，即总共迭代几次
min_samples_split	内部节点再划分所需最小样本数
min_samples_leaf	叶节点包含最少样本数
min_weight_fraction_leaf	叶节点所需最小权重和
max_features	在进行切分时候最多带入多少个特征进行划分规则挑选
random_state	随机数种子
max_leaf_nodes	叶节点最大个数
min_impurity_decrease	数据集再划分至少需要降低的损失值
min_impurity_split	数据集再划分所需最低不纯度，将在0.25版本中移除
class_weight	各类样本权重

其中我们挑选"n_estimators"、"min_samples_leaf"、"min_samples_split"、"max_depth"和"max_features"进行参数搜索：

然后是关于网格搜索工具的选择。随着sklearn不断完善，有越来越多的网格搜索工具可供选择，但整体来看其实就是在效率和精度之间做权衡，有些网格搜索工具由于是全域枚举（如GridSearchCV），所以执行效率较慢、但结果精度有保障，而如果愿意牺牲精度换执行效率，则也有很多工具可以选择，如RandomizedSearchCV。当然，在最新的sklearn版本中，还出现了一种更高效的搜索策略——HalvingGridSearchCV，该方法先两两比对、然后逐层筛选的方法来进行参数筛选，并且同时支持HalvingGridSearchCV和HalvingRandomSearchCV。注意，这是sklearn最新版、也就是0.24版才支持的功能，该功能的出现也是0.24版最大的改动之一，而该功能的加入，也将进一步减少网格搜索所需计算资源、加快网格搜索的速度。

围绕本次竞赛的数据，在实际执行网格搜索的过程中，建议先使用RandomizedSearchCV确定大概范围，然后再使用GridSearchCV高精度搜索具体参数取值。当然，如果是使用最新版的sklearn，也可以考虑使用Halving方法进行搜索。在公开课讲解过程中，由于时间有限，此处我们在大致确定最优参数范围的前提下设置在一个相对较小的参数空间内来进行搜索：

In [43]:

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


parameter_space = {
    "n_estimators": [79, 80, 81],
    "min_samples_leaf": [29, 30, 31],
    "min_samples_split": [2, 3],
    "max_depth": [9, 10],
    "max_features": ["auto", 80]
}
```

然后构建随机森林评估器，并输入其他超参数取值

In [44]:
```
clf = RandomForestRegressor(
    criterion="mse",
    n_jobs=15,
    random_state=22)
```

准备完毕后，开始进行网格搜索：

In [ ]:
```
grid = GridSearchCV(clf, parameter_space, cv=2, scoring="neg_mean_squared_error")
grid.fit(train[features].values, train['target'].values)
```

搜索完毕后模型也随之训练完成，接下来我们即可查看训练结果了。首先是在参数空间内最终搜索出来的最优参数组：

In [14]:
```
grid.best_params_
```

Out[14]:
```
{'max_depth': 9,
 'max_features': 80,
 'min_samples_leaf': 30,
 'min_samples_split': 2,
 'n_estimators': 80}
```

同时，我们也可以直接查看或调用最优参数组成评估器：

In [15]:
```
grid.best_estimator_
```

Out[15]:
```
RandomForestRegressor(max_depth=9, max_features=80, min_samples_leaf=30,
                      n_estimators=80, n_jobs=4, random_state=2020)
```

当然，我们也可以直接查看在训练集上的最终评分：

In [21]:
```
np.sqrt(-grid.best_score_)
```

Out[21]:
```
3.691443363139941
```

In [23]:
```
grid.best_estimator_.predict(test[features])
```

Out[23]:
```
array([-3.47007473, -0.78764767, -0.42686863, ...,  0.55621849,
       -2.34277367,  0.3091642 ])
```

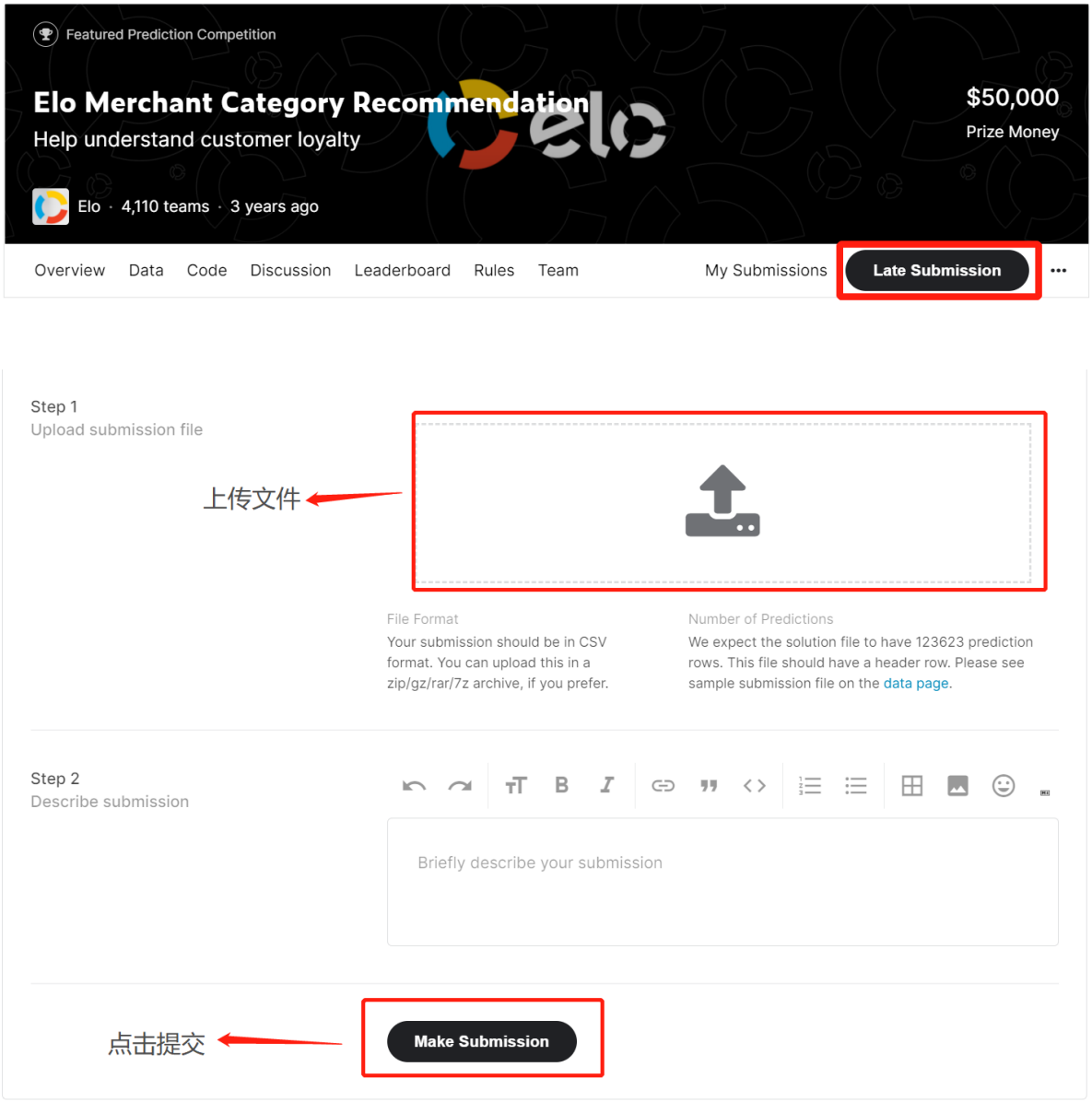
然后将结果按照所需要提交的格式写入csv文档

In [24]:
```
test['target'] = grid.best_estimator_.predict(test[features])
test[['card_id', 'target']].to_csv("result/submission_randomforest.csv", index=False)
```

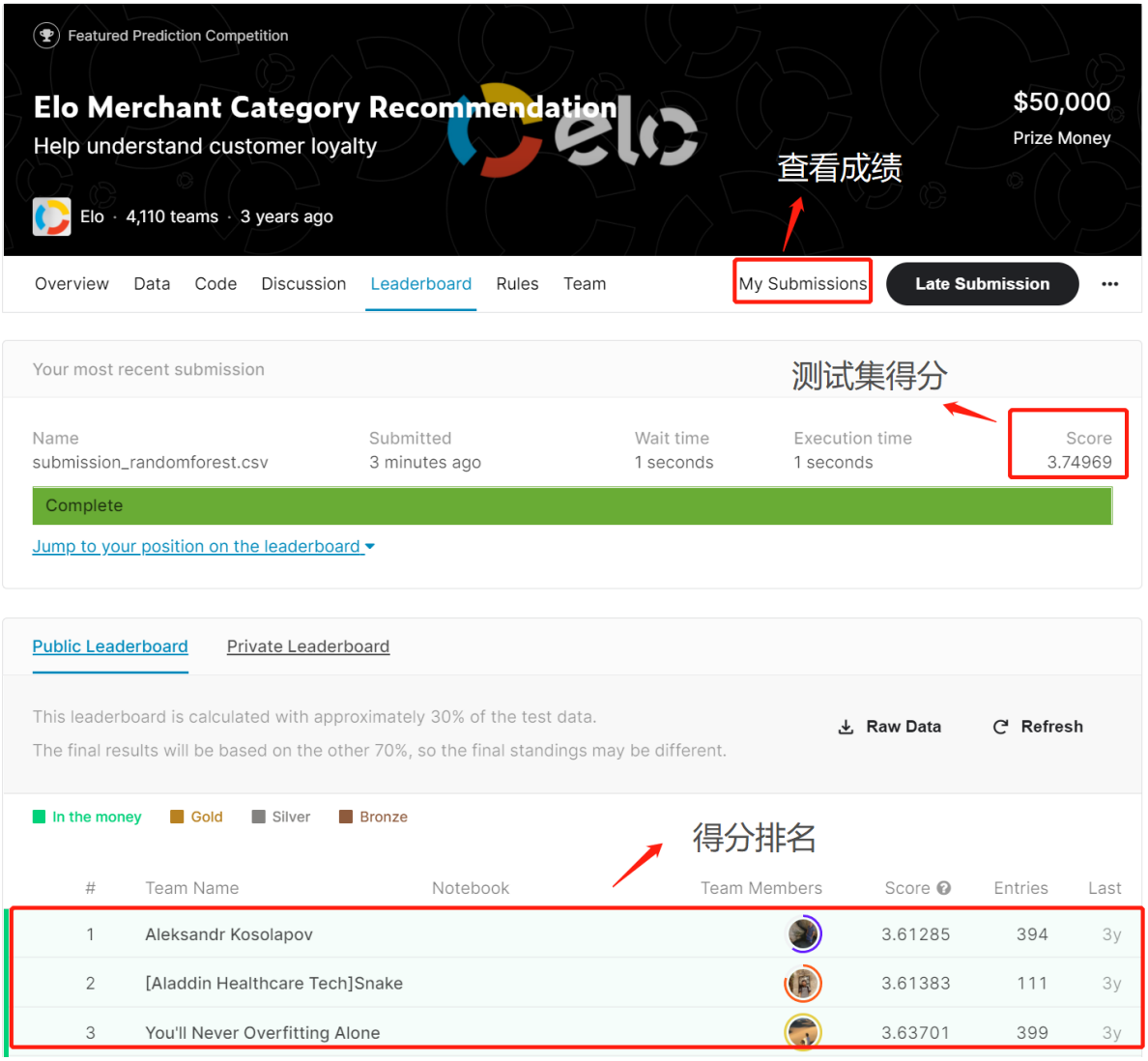
数据文档写入完毕后，接下来就可以直接在Kaggle上提交了。上传提交结果数据和下载数据过程类似，都可以直接利用网页功能实现，或者通过命令行的方式实现。

- 结果提交

在Kaggle竞赛主页找到Late Submission进行结果提交，只需将结果文件在线提交即可：



提交完成后，即可在我的提交结果中看到成绩了：



最终在测试集上MSE为3.749，算是基本合格的成绩，至此我们也就相当于从零开始完成了一项Kaggle赛事。三天公开课时间有限，但如果能够完成到这一步，也是非常有收获的。

三、后续优化策略

当然，围绕上述结果还有许多可以优化的地方，在公开课的最后，我们也将提供一些后续优化建议：

- 文本特征挖掘

  在特征处理的过程中，可以尝试使用NLP领域的TF-IDF进行词频统计，增加离散变量特征；

- 更多衍生特征

  除了对离散变量进行词频统计外，我们还可以考虑构建更多特征，如全局card_id特征、最近两个月 card_id特征、二阶特征和补充特征等，来更深程度挖掘数据集信息；

- 更多集成算法

  除了随机森林外，还有许多功能非常强大的集成模型，包括LightGBM、XGBoost等，都是可以尝试使用的算法；

- 模型融合方法

  既然使用了多集成模型来进行建模，那么模型融合也势在必行。模型融合能够很好的综合各

集成模型的输出结果，来做出最后更加综合的判断。当然模型融合可以考虑简单加权融合或者stacking融合方法；

- 更加细致的数据处理

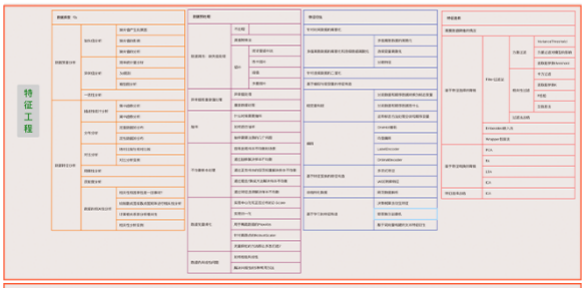
除了技术手段外，我们可也可以围绕此前得出的业务分析结论，对数据集进行更加细致的处理，如此前标签中出现的异常值的处理、13家商户没有过去一段时间营销信息等，通过更加细致的处理，能够让模型达到更好的效果。

当然，如果对机器学习算法或者机器学习竞赛感兴趣，也欢迎各位小伙伴参与付费课程的学习！《机器学习实战训练营》（第二期）即将开课，十八周80+小时体系大课，由我和菜菜老师主讲，从零开始补充机器学习必备基础，深入讲解集成算法与特征工程方法，并包含四项大型Kaggle案例/企业级应用案例！对机器学习感兴趣的小伙伴扫描下方二维码查看课程详情哦！也可以添加客服VX：little_bird0229咨询课程，还有惊喜优惠券哦~

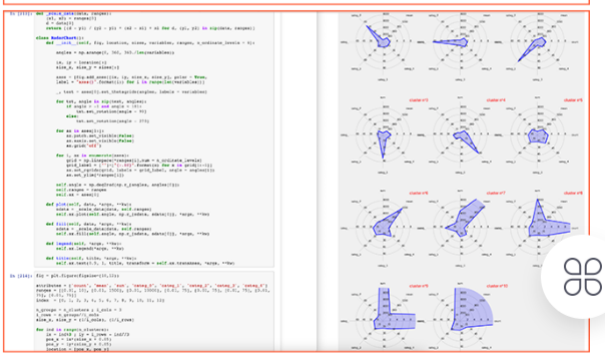
竞赛经验倾囊相授

拒绝纸上谈兵，快速提分有迹可循！

竞赛常用特征工程技巧



竞赛常用模型融合技巧



扫码查看更多课程详情

本次公开课圆满结束！接下来有任何想听的内容，也欢迎在我的B站账号下留言~各位小伙伴，大家下次见！