阿里云安全恶意程序检测

小组名：　　菜鸟裹裹

小组成员：　　余晓 3220200998 （组长）

张嘉辉 3220201005

李洁 3220200900

岳佳琪 3220201000

郑慧娴 3220201026

目录

1. 研究框架

1.1 题目说明

本题提供的数据来自经过沙箱程序模拟运行后的 API 指令序列，全为 Windows 二进制可执行程序，经过脱敏处理；样本数据均来自互联网，其中恶意文件的类型有感染型病毒、木马程序、挖矿程序、DDos 木马、勒索病毒等，共 6 亿条数据。

1.2 数据说明

训练数据调用记录近 9000 万次，文件 1 万多个（以文件编号汇总）。测试数据调用记录近 8000 万次，文件 1 万多个。每个文件调用的 API 可能有很多，并且 API 之间可能存在一些序列关系。

原始训练数据字段如下表所述，测试数据除了没有 label 字段，数据格式与训练数据一致。

字段	类型	解释
field_id	bigint	文件编号
label	bigint	文件标签，0-正常/1-勒索病毒/2-挖矿程序/3-DDos 木马/4-蠕虫病毒/5-感染型病毒/6-后门程序/7-木马程序
api	bigint	文件调用的 API 名称
tid	bigint	调用 API 的线程编号
index	bigint	线程中 API 调用的顺序编号

1.3 评价指标

$$logloss = \frac{1}{N}\sum_{j}^{N}\sum_{i}^{M}\left[y_{ij}\log\left(P_{ij}\right) + \left(1 - y_{ij}\right)\log\left(1 - P_{ij}\right)\right] \qquad (1\text{-}1)$$

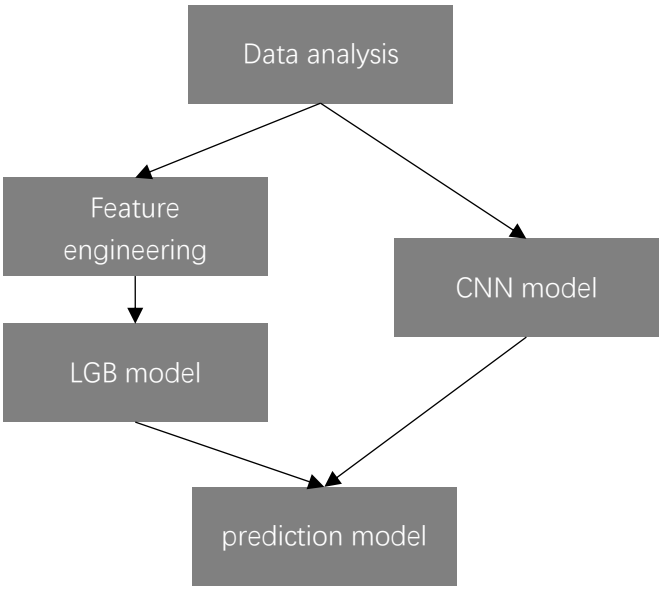
其中，M 代表分类数，N 代表测试集样本数，y_{ij}代表第 i 个样本是否为类别 j（1 为是，0 为否），P_{ij}代表选手提交的第 i 个样本被预测为类别 j的概率， $logloss$ 最终保留小数点后 6 位。

1.4 研究思路

本题的特征主要是 API 接口的名称，这是融合时序与文本的数据，同时接口名称基本表达了接口用途。因此，可以对所有 API 数据构造 CountVectorizer 特征，即 API 看作一个词，文件的 API 调用看作一个文本，对于每个训练文本，只考虑每种词汇在该训练文本中出现的频率，所以需要解决的分类问题可以看作是在 NLP 字段中具有词序的文本分类问题。

我们根据官方提供的每个文件对 API 调用顺序及线程的相关信息按文件进行分类，将文件属于每个类的概率作为最终结果进行提交，并采用 logloss 作为最终评分。

整体研究框架如下图所示。



2. 数据探索与基线模型构建

2.1 数据探索

2.1.1 训练集数据探索

（1）数据特征类型

题目数据是 CSV 文件，导入数据包，读取数据，并查看训练集数据前 10 行信息：

```
#导入必要的包
import pandas as pd
import numpy as np
# import seaborn as sns
import  matplotlib.pyplot as plt
#忽略警告信息
import warnings
warnings.filterwarnings("ignore")
#读取数据
train = pd.read_csv("./security_train.csv")
test = pd.read_csv("./security_test.csv")
#查看训练数据的前10行信息
print(train.head(10))
```

	file_id	label	api	tid	index
0	1	5	LdrLoadDll	2488	0
1	1	5	LdrGetProcedureAddress	2488	1
2	1	5	LdrGetProcedureAddress	2488	2
3	1	5	LdrGetProcedureAddress	2488	3
4	1	5	LdrGetProcedureAddress	2488	4
5	1	5	LdrGetProcedureAddress	2488	5
6	1	5	LdrGetProcedureAddress	2488	6
7	1	5	LdrGetProcedureAddress	2488	7
8	1	5	LdrGetProcedureAddress	2488	8
9	1	5	LdrGetProcedureAddress	2488	9

用 info()函数查看训练集的大小、数据类型等信息。

```
#查看训练集的大小、类型等信息
train.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 89806693 entries, 0 to 89806692
Data columns (total 5 columns):
 #   Column   Dtype
---  ------   -----
 0   file_id  int64
 1   label    int64
 2   api      object
 3   tid      int64
 4   index    int64
dtypes: int64(4), object(1)
memory usage: 3.3+ GB
```

由运行结果可知，整个数据集的大小 3.3GB，共有 89 806 692 条记录。每条数据中有 4 个 int64 类型(file_id, label, tid, index)的数据和 1 个 object 类型的数据（api）。

用 describe()函数查看训练集数据的统计信息。

```
train.describe()
```

	file_id	label	tid	index
count	8.980669e+07	8.980669e+07	8.980669e+07	8.980669e+07
mean	7.078770e+03	3.862835e+00	2.533028e+03	1.547521e+03
std	3.998794e+03	2.393783e+00	6.995798e+02	1.412249e+03
min	1.000000e+00	0.000000e+00	1.000000e+02	0.000000e+00
25%	3.637000e+03	2.000000e+00	2.356000e+03	3.490000e+02
50%	7.161000e+03	5.000000e+00	2.564000e+03	1.085000e+03
75%	1.055100e+04	5.000000e+00	2.776000e+03	2.503000e+03
max	1.388700e+04	7.000000e+00	2.089600e+04	5.000000e+03

（2）数据分布

用 nunique()函数查看训练集中变量取值分布：

```
#用nunique函数查看训练集中的变量取值分布
train.nunique()
```

```
file_id    13887
label          8
api          295
tid         2782
index       5001
dtype: int64
```

由运行结果可知，file_id 有 13 887 个不同的值；共 8 种标签 label；有 295 个不同的 API；有 2782 个不同的 tid；有 5001 个不同的 index。

（3）缺失值

查看训练集数据的缺失情况：

```
#查看训练集数据的缺少情况—结论不存在缺少
train.isnull().sum()
```

```
file_id    0
label      0
api        0
tid        0
index      0
dtype: int64
```

由运行结果可知，数据不存在缺失值。

（4）异常值

分析训练集的“index”特征：

```
#判断是否有异常值—最大值最小值符合范围
train['index'].describe()
```

```
count    8.980669e+07
mean     1.547521e+03
std      1.412249e+03
min      0.000000e+00
25%      3.490000e+02
50%      1.085000e+03
75%      2.503000e+03
max      5.000000e+03
Name: index, dtype: float64
```

由结果可知，“index”特征最小值为 0，最大值为 5000，刚好 5001 个值，因此无异常值。

分析训练集的“tid”特征：

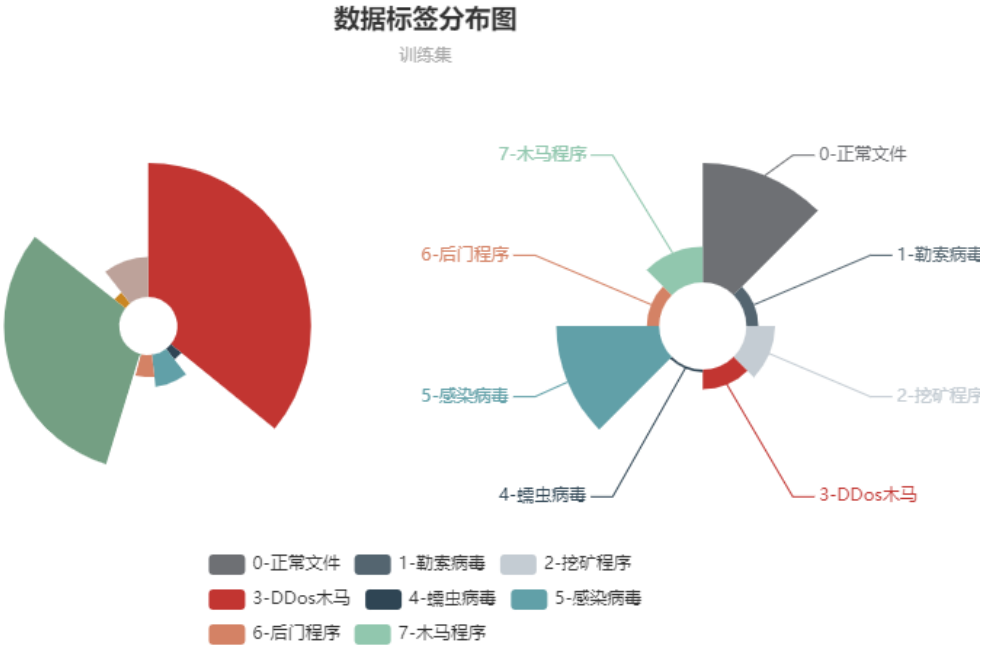
```
#判断是否有异常值—最大值最小值符合范围
train['tid'].describe()
```

```
count    8.980669e+07
mean     2.533028e+03
std      6.995798e+02
min      1.000000e+02
25%      2.356000e+03
50%      2.564000e+03
75%      2.776000e+03
max      2.089600e+04
Name: tid, dtype: float64
```

由结果可知，“tid”特征的最小值为 100，最大值为 9196，因为这个字段表示线程，因此无法判断是否有异常值。

（5）标签分布

统计标签取值分布情况并作可视化处理。训练集中共有 4978 个正常文件（label=0）；502 个勒索病毒（label=1）；1196 个挖矿程序（label=2）；820 个 DDos 木马（label=3）；100 个蠕虫病毒（label=4）；4289 个感染病毒（label=5）；515 个后门程序（label=6）；1487 个木马程序（label=7）。



2.1.2 测试集数据探索

（1）数据特征类型

查看测试集数据前 10 行信息：

```
#测试集数据探索
test.head()
```

	file_id	api	tid	index
0	1	RegOpenKeyExA	2332	0
1	1	CopyFileA	2332	1
2	1	OpenSCManagerA	2332	2
3	1	CreateServiceA	2332	3
4	1	RegOpenKeyExA	2468	0

用 info()函数查看测试集的大小、数据类型等信息。

```
#查看测试集大小、数据类型信息
test.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 79288375 entries, 0 to 79288374
Data columns (total 4 columns):
 #   Column   Dtype
---  ------   -----
 0   file_id  int64
 1   api      object
 2   tid      int64
 3   index    int64
dtypes: int64(3), object(1)
memory usage: 2.4+ GB
```

由运行结果可知，整个数据集的大小 2.4GB，共有 79 288 375 条记录。每条数据中有 3 个 int64 类型(file_id, tid, index)的数据和 1 个 object 类型的数据（api）。

（2）数据分布

查看测试集中变量值取值的分布：

```
#查看测试集中变量的取值分布
test.nunique()
```

```
file_id    12955
api          298
tid         2047
index       5001
dtype: int64
```

由运行结果可知，file_id 有 12 955 个不同的值；有 298 个不同的 API；有 2047 个不同的 tid；有 5001 个不同的 index。

（3）缺失值

查看测试集数据的缺失情况：

```
#查看测试集数据的缺失情况
test.isnull().sum()
```

```
file_id    0
api        0
tid        0
index      0
dtype: int64
```

由运行结果可知，数据不存在缺失值。

（4）异常值

分析测试集的“index”特征：

```
#异常值
test['index'].describe()
```

```
count    7.928838e+07
mean     1.584815e+03
std      1.411116e+03
min      0.000000e+00
25%      3.900000e+02
50%      1.131000e+03
75%      2.547000e+03
max      5.000000e+03
Name: index, dtype: float64
```

由结果可知，“index”特征最小值为 0，最大值为 5000，刚好 5001 个值，因此无异常值。

分析测试集的“tid”特征：

```
test['tid'].describe()
```

```
count    7.928838e+07
mean     2.491914e+03
std      5.824600e+02
min      1.000000e+02
25%      2.360000e+03
50%      2.556000e+03
75%      2.752000e+03
max      9.196000e+03
Name: tid, dtype: float64
```

由结果可知，“tid”特征的最小值为 100，最大值为 9196，因为这个字段表示线程，因此无法判断是否有异常值。

2.1.3 数据集联合分析

（1）file_id 分析

对比分析“file_id”变量在训练集和测试集中分布的重合情况：

```
train_fields = train['file_id'].unique()
test_fileds = test['file_id'].unique()
len(set(train_fields)-set(test_fileds))
```

```
932
```

运行结果表明，有 932 个训练集数据在测试集中是没有的。

```
len(set(test_fileds)-set(train_fields))
```

```
0
```

运行结果表明，测试集有的文件在训练集中都有。

我们发现训练集和测试集的 file_id 存在交叉，因此不能直接合并，需要进行其他处理来区分训练集和测试集。

（2）API 分析

对比分析“API”变量在训练集和测试集中分布的重合情况：

```
#API分析
train_apis = train['api'].unique()
test_apis = test['api'].unique()
set(train_apis)-set(test_apis)
```

```
{'EncryptMessage', 'RtlCompressBuffer', 'WSASendTo'}
```

```
set(test_apis)-set(train_apis)
```

```
{'CreateDirectoryExW',
 'InternetGetConnectedStateExA',
 'MessageBoxTimeoutW',
 'NtCreateUserProcess',
 'NtDeleteFile',
 'TaskDialog'}
```

运行结果表明，测试集中有 CreateDirectoryExW 等 6 个 API 未出现在训练集中，训练集中有 EncryptMessage 等 3 个 API 未出现在测试集中。

2.2 特征工程

（1）利用 count()函数和 nunique()函数生成特征：反应样本调用 api，tid，index 的频率信息。

```
def simple_sts_feature(df):
    simple_feature = pd.DataFrame()
    simple_feature['file_id'] = df['file_id'].unique()
    simple_feature['file_id'] = simple_feature.sort_values('file_id')

    df_grp = df.groupby('file_id')
    simple_feature['file_id_api_count'] = df_grp['api'].count().values

    simple_feature['file_id_api_nunique'] = df_grp['api'].nunique().values
```

```
    simple_feature['file_id_tid_count'] = df_grp['tid'].count().values
    simple_feature['file_id_tid_nunique'] = df_grp['tid'].nunique().values

    simple_feature['file_id_index_count'] = df_grp['index'].count().values
    simple_feature['file_id_index_nunique'] = df_grp['index'].nunique().values

    return simple_feature
```

（2）利用 mean()函数、min()函数、max()函数、std()函数生成特征：tid, index 可认为是数值特征，提取统计特征。

```
def simple_numerical_sts_feature(df):
    simple_feature = pd.DataFrame()
    simple_feature['file_id'] = df['file_id'].unique()
    simple_feature['file_id'] = simple_feature.sort_values('file_id')

    df_grp = df.groupby('file_id')

    simple_feature['file_id_tid_mean'] = df_grp['tid'].mean().values
    simple_feature['file_id_tid_min'] = df_grp['tid'].min().values
    simple_feature['file_id_tid_max'] = df_grp['tid'].max().values
    simple_feature['file_id_tid_std'] = df_grp['tid'].std().values

    simple_feature['file_id_index_mean'] = df_grp['index'].mean().values
    simple_feature['file_id_index_min'] = df_grp['index'].min().values
    simple_feature['file_id_index_max'] = df_grp['index'].max().values
    simple_feature['file_id_index_std'] = df_grp['index'].std().values

    return simple_feature
```

（3）利用定义的特征生成函数，并生成训练集和测试集的统计特征。

```
#构建特征
simple_train_feature1 = simple_sts_feature(train)
simple_test_feature1 = simple_sts_feature(test)


simple_train_feature2 = simple_numerical_sts_feature(train)
simple_test_feature2 = simple_numerical_sts_feature(test)
```

2.3 基线构建

2.3.1LightGBM 模型

LightGBM 是一个梯度 boosting 框架，使用基于学习算法的决策树。它可以说是分布式的，高效的。LGB 主要有以下特点：

- 基于 Histogram 的决策树算法
- 带深度限制的 Leaf-wise 的叶子生长策略
- 直方图做差加速

- 直接支持类别特征(Categorical Feature)
- Cache 命中率优化
- 基于直方图的稀疏特征优化
- 多线程优化

LGB 有以下优势：

- 更快的训练效率
- 低内存使用
- 更高的准确率
- 支持并行化学习
- 可处理大规模数据

2.3.2 基线模型构建

获取标签，构造训练集和测试集。

模型评估函数使用 *logloss*，代码如下：

```
def lgb_logloss(preds,data):
    labels = data.get_label()
    classes = np.unique(labels)
    preds_prob = []
    for i in range(len(classes)):
        preds_prob.append(preds[i*len(labels):(i+1)*len(labels)])
    preds_prob =  np.vstack(preds_prob)
    loss=[]
    for i in range(preds_prob.shape[1]):#样本个数
        sum = 0
        for j in range(preds_prob.shape[0]):#类别个数
            pred = preds_prob[j,i]#第i个样本预测为第j类的概率
            if j==labels[i]:
                sum+=np.log(pred)
            else:
                sum+=np.log(1-pred)
        loss.append(sum)
    return 'loss is:',-1*(np.sum(loss)/preds_prob.shape[1]),False
```

使用 5 折交叉验证，采用 LightGBM 模型来构建线下验证集，模型训练代码与结果如下：

```
#线下验证
train_features = [col for col in train_data.columns if col not in ['label','file_id']]
train_label = 'label'
from sklearn.model_selection import StratifiedKFold,KFold
params = {
    'task':'train',
    'num_leaves':255,
    'objective':'multiclass',
    'num_class':8,
    'min_data_in_leaf':50,
    'learning_rate':0.05,
    'feature_fraction':0.85,
    'bagging_fraction':0.85,
    'bagging_freq':5,
```

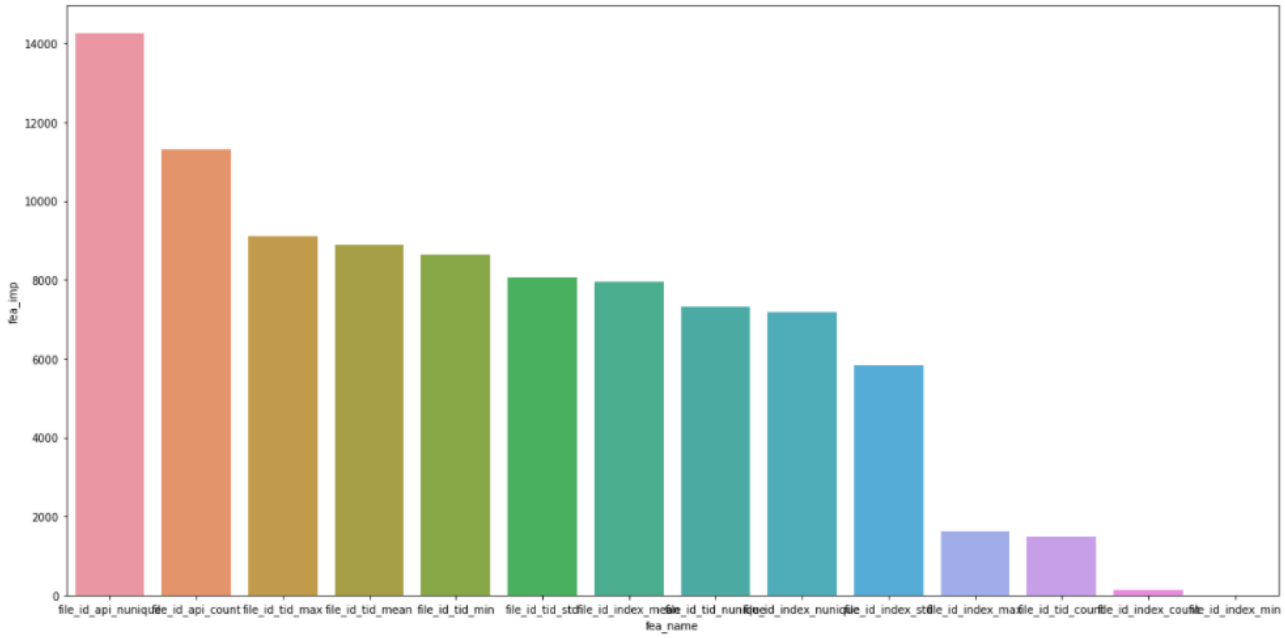
```
    'max_bin':128,
    'random_state':100,
    'verbosity': -1
}
folds =KFold(n_splits=5,shuffle=True,random_state=15)
oof = np.zeros(len(train))

predict_res = 0
models = []
import lightgbm as lgb
for fold,(trn_id,val_id) in enumerate(folds.split(train_data)):
    print("fold n {}".format(fold))
    trn_data = lgb.Dataset(train_data.iloc[trn_id][train_features],label=train_data.iloc[trn
    val_data = lgb.Dataset(train_data.iloc[val_id][train_features], label=train_data.iloc[va
    clf = lgb.train(params,trn_data,num_boost_round=2000,valid_sets=[trn_data,val_data],verb
    models.append(clf)
```

```
fold n 0
Training until validation scores don't improve for 100 rounds
[50]    training's multi_logloss: 0.431369      training's loss is:: 0.771962   valid_1's multi_logloss: 0.680269       valid_1's loss is::
1.13155
[100]   training's multi_logloss: 0.201085      training's loss is:: 0.382615   valid_1's multi_logloss: 0.671298       valid_1's loss is::
1.11585
[150]   training's multi_logloss: 0.101343      training's loss is:: 0.19816    valid_1's multi_logloss: 0.704873       valid_1's loss is::
1.16574
Early stopping, best iteration is:
[73]    training's multi_logloss: 0.30019       training's loss is:: 0.555744   valid_1's multi_logloss: 0.666186       valid_1's loss is::
1.1093
fold n 1
Training until validation scores don't improve for 100 rounds
[50]    training's multi_logloss: 0.428075      training's loss is:: 0.766616   valid_1's multi_logloss: 0.709282       valid_1's loss is::
1.16272
[100]   training's multi_logloss: 0.198723      training's loss is:: 0.378419   valid_1's multi_logloss: 0.700122       valid_1's loss is::
1.14116
[150]   training's multi_logloss: 0.100872      training's loss is:: 0.19734    valid_1's multi_logloss: 0.738649       valid_1's loss is::
1.19481
Early stopping, best iteration is:
[73]    training's multi_logloss: 0.297361      training's loss is:: 0.551098   valid_1's multi_logloss: 0.694235       valid_1's loss is::
1.13517

fold n 2
Training until validation scores don't improve for 100 rounds
[50]    training's multi_logloss: 0.425288      training's loss is:: 0.761576   valid_1's multi_logloss: 0.745304       valid_1's loss is::
1.22324
[100]   training's multi_logloss: 0.195328      training's loss is:: 0.372065   valid_1's multi_logloss: 0.742432       valid_1's loss is::
1.21693
[150]   training's multi_logloss: 0.0968182     training's loss is:: 0.189483   valid_1's multi_logloss: 0.792177       valid_1's loss is::
1.29193
Early stopping, best iteration is:
[73]    training's multi_logloss: 0.294585      training's loss is:: 0.545868   valid_1's multi_logloss: 0.731312       valid_1's loss is::
1.20156
fold n 3
Training until validation scores don't improve for 100 rounds
[50]    training's multi_logloss: 0.429625      training's loss is:: 0.768713   valid_1's multi_logloss: 0.720675       valid_1's loss is::
1.19433
[100]   training's multi_logloss: 0.198977      training's loss is:: 0.378212   valid_1's multi_logloss: 0.709815       valid_1's loss is::
1.17413
[150]   training's multi_logloss: 0.099573      training's loss is:: 0.194517   valid_1's multi_logloss: 0.758826       valid_1's loss is::
1.2462
Early stopping, best iteration is:
[74]    training's multi_logloss: 0.294028      training's loss is:: 0.544535   valid_1's multi_logloss: 0.702428       valid_1's loss is::
1.16501
fold n 4
Training until validation scores don't improve for 100 rounds
[50]    training's multi_logloss: 0.431106      training's loss is:: 0.771531   valid_1's multi_logloss: 0.714685       valid_1's loss is::
1.17941
[100]   training's multi_logloss: 0.200899      training's loss is:: 0.382212   valid_1's multi_logloss: 0.706153       valid_1's loss is::
1.16353
[150]   training's multi_logloss: 0.100749      training's loss is:: 0.196847   valid_1's multi_logloss: 0.744389       valid_1's loss is::
1.22088
Early stopping, best iteration is:
[72]    training's multi_logloss: 0.304554      training's loss is:: 0.563107   valid_1's multi_logloss: 0.699542       valid_1's loss is::
1.15491
```

2.4 特征重要性分析

通过特征重要性分析，可以看到当前指标显示的成绩下，影响因子最高的特征因素，同时进行可视化处理，结果如下：

```
#特征重要性分析
import seaborn as sns
import matplotlib.pyplot as plt
feature_importance = pd.DataFrame()
feature_importance['fea_name'] = train_features
feature_importance['fea_imp' ] = clf.feature_importance()
feature_importance = feature_importance.sort_values('fea_imp',ascending=False)
plt.figure(figsize=[20,10,])

sns.barplot(x=feature_importance['fea_name'],y=feature_importance['fea_imp'])
plt.show()
```



由运行结果可以看出：

（1）API 的调用次数和 API 的调用类别数是最重要的两个特征，即不同的病毒会调用不同的 API，而且由于有些病毒需要复制自身，因此调用 API 的次数会明显比其他不同类别的病毒多。

（2）第 3~5 强的特征都是线程统计特征，这是由于木马等病毒经常需要通过线程监听一些内容，所以在线程等使用上会表现得略有不同。

2.5 模型测试

通过前期的数据探索分析过和基础的特征工程，我们采用 LightGBM 进行 5 折交叉验证，即，用上面训练的 5 个 LightGBM 分别对测试集进行预测，将所有预测结果的均值作为最终结果，以此作为 baseline。

```
pred_res = 0
fold = 5
for model in models:
    pred_res += model.predict(test_submit[train_features])*1.0/fold
test_submit['prob0'] = 0
test_submit['prob1'] = 0
test_submit['prob2'] = 0
test_submit['prob3'] = 0
test_submit['prob4'] = 0
test_submit['prob5'] = 0
test_submit['prob6'] = 0
test_submit['prob7'] = 0
test_submit[['prob0 ','prob1','prob2','prob3','prob4','prob5','prob6','prob7']]=pred_res
test_submit[['file_id','prob0','prob1','prob2','prob3','prob4','prob5','prob6','prob7']].
```

Logloss=1.087292

3. 高阶数据探索与优化方案

3.1 多变量交叉探索

（1）通过统计特征 file_id_cnt，分析 file_id 变量和 api 变量之间的关系。

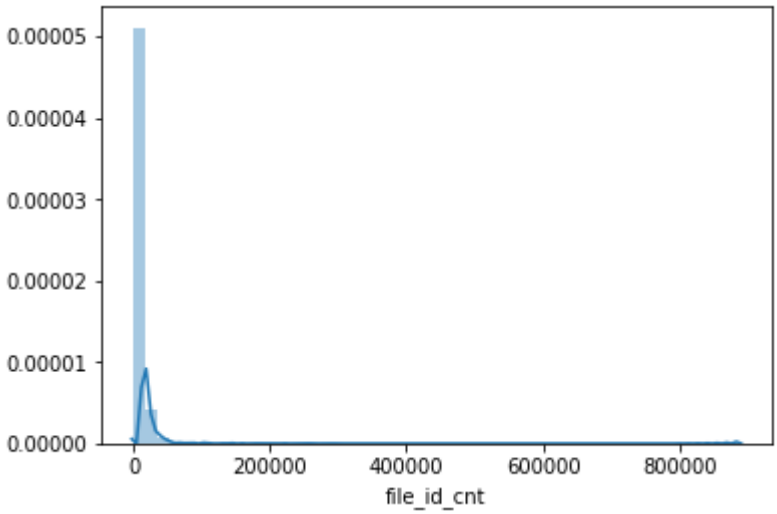
```
train_analysis = train[['file_id','label']].drop_duplicates(subset = ['file_id','label'], keep = 'last')
```

```
dic_ = train['file_id'].value_counts().to_dict()
train_analysis['file_id_cnt'] = train_analysis['file_id'].map(dic_).values
```

运行结果：

```
train_analysis['file_id_cnt'].value_counts()
```

```
5001     448
268      211
44       186
4        160
16       149
2        143
10002    141
257      135
20004    126
3        120
15003    114
22        96
43        87
```

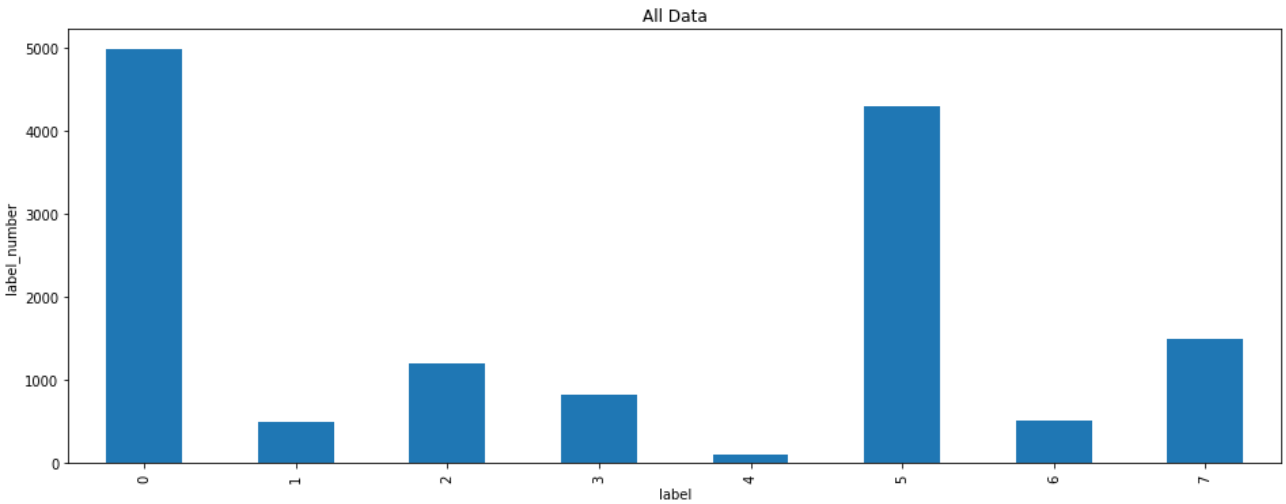


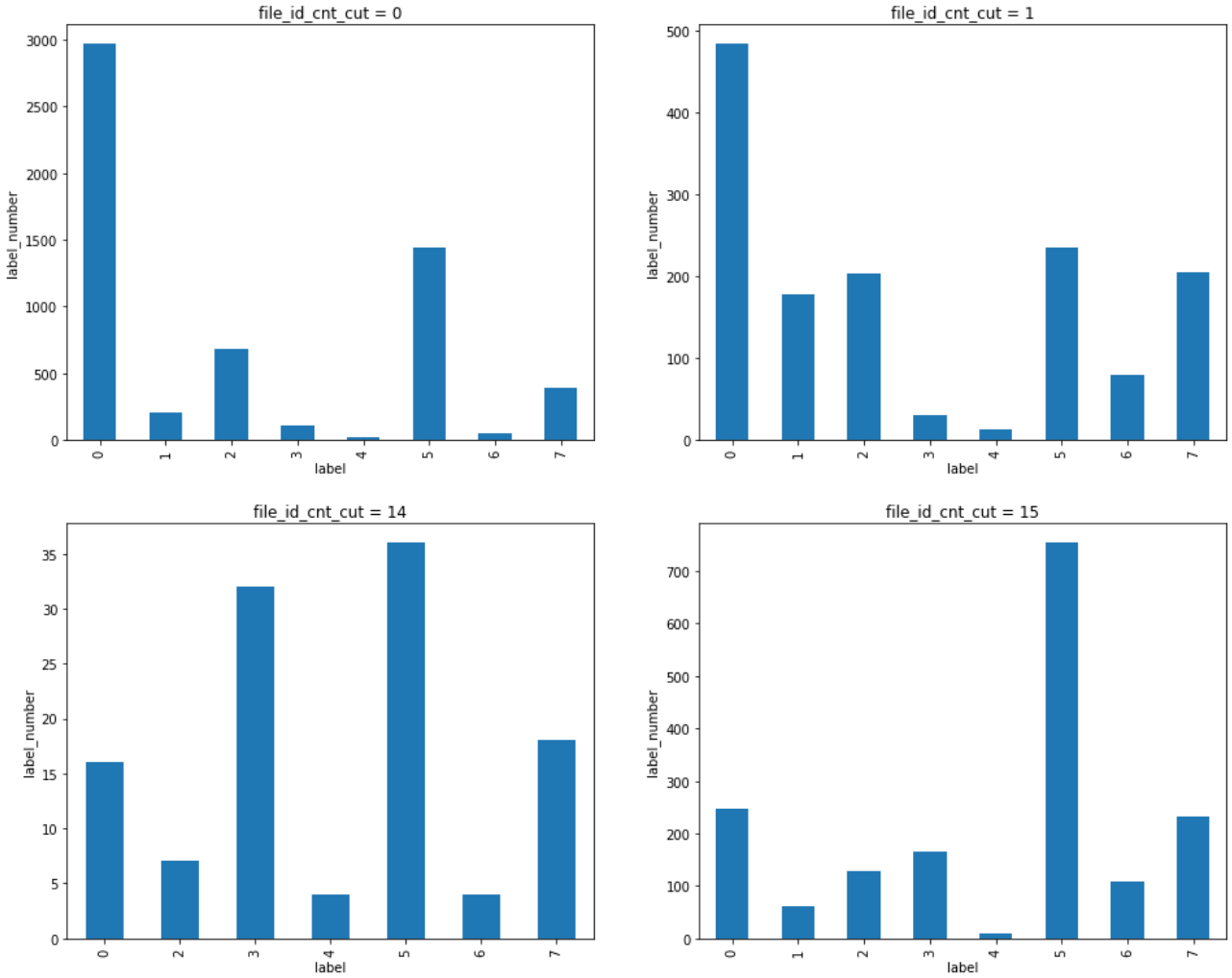
可以看到，文件调用 API 次数出现最多的是 5001 次，API 调用次数的 80%都集中在 10，000 次以下。

（2）为了便于分析 file_id_cnt 变量与 label 变量的关系，首先将数据按 file_id_cnt 变量取值划分为 16 个区间。

```
def file_id_cnt_cut(x):
    if x< 15000:
        return x // 1e3
    else:
        return 15

train_analysis['file_id_cnt_cut'] = train_analysis['file_id_cnt'].map(file_id_cnt_cut).values
```

随机选取 4 个区间进行查看，运行结果如下。

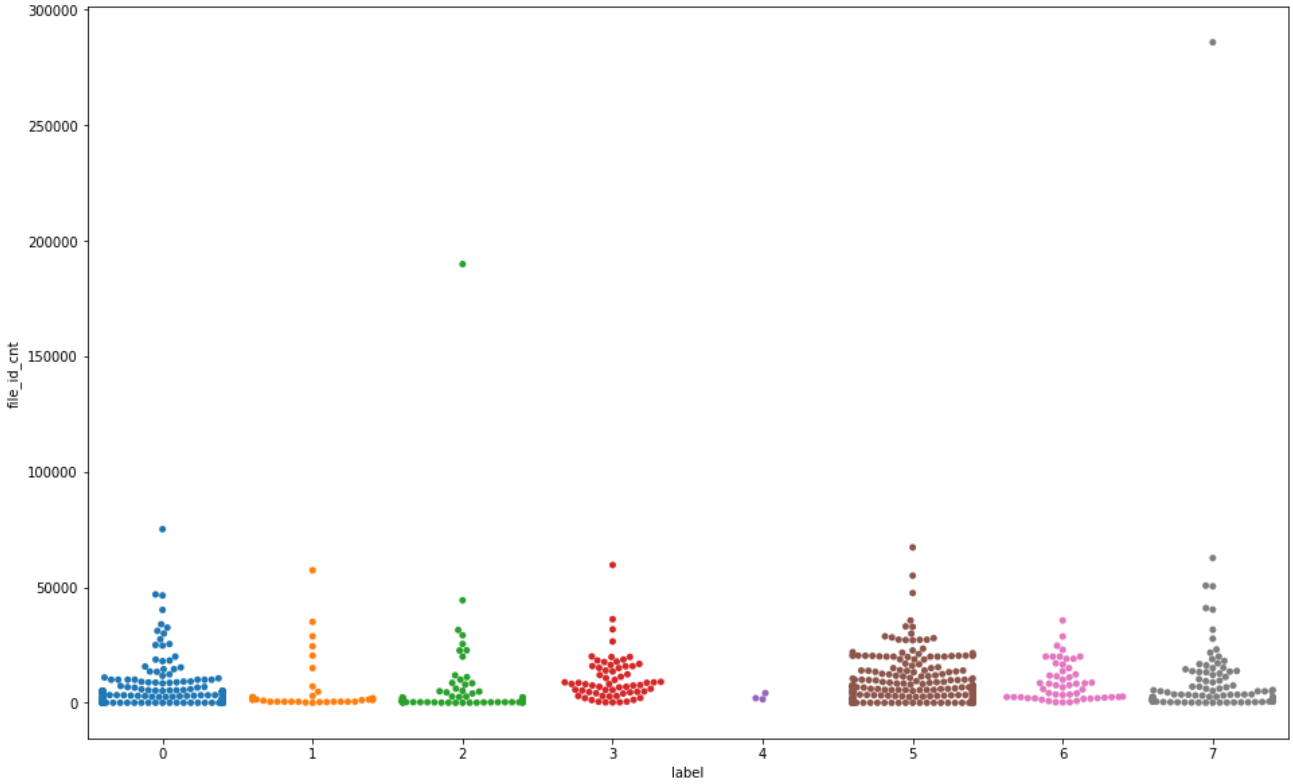




可以看到：当 API 调用次数越多，是第 5 类病毒（感染型病毒）的可能性就越大。这是因为感染型病毒需要找到宿主并自身传播,所以调用 api 的次数会相对多一些。

用分簇散点图查看 label 下 file_id_cnt 的分布，由于散点图绘制时间过长，我们采样其中 1000 个样本点进行绘制。

```
plt.figure(figsize=[16,10])
sns.swarmplot(x =train_analysis.iloc[:1000]['label'], y = train_analysis.iloc[:1000]['file_id_cnt'])
```



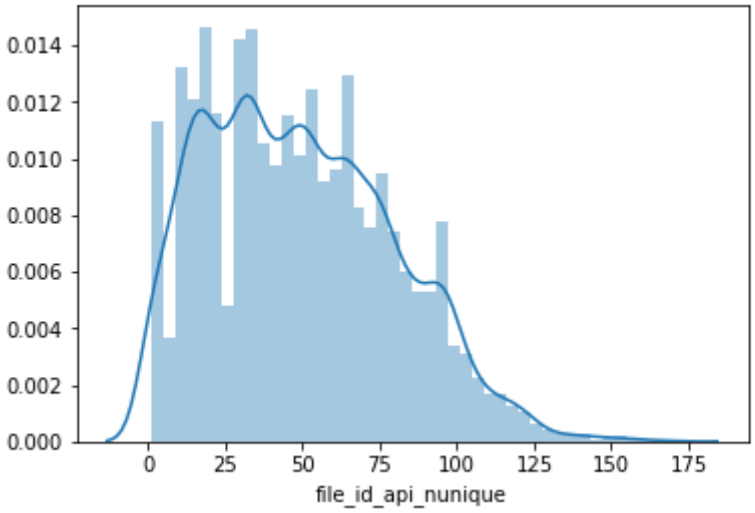
可以看到：从频次上看，第五类病毒调用 API 的次数最多；从调用峰值上看，第 2 类和第 7 类病毒有时能调用 150，000 次的 API。

（3）首先通过文件调用 API 类别数 file_id_api_nunique，分析变量 file_id 和 API 的关系。

```
dic_ = train.groupby('file_id')['api'].nunique().to_dict()
train_analysis['file_id_api_nunique'] = train_analysis['file_id'].map(dic_).values
```

```
sns.distplot(train_analysis['file_id_api_nunique'])
```

运行结果：



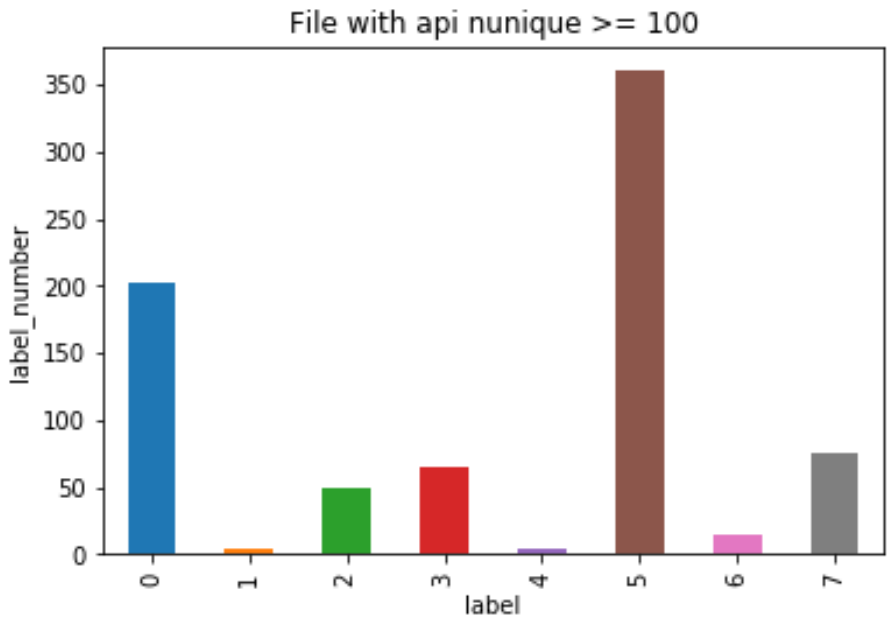
```
train_analysis['file_id_api_nunique'].describe()
```

```
count    13887.000000
mean        49.263700
std         30.338888
min          1.000000
25%         24.000000
50%         47.000000
75%         71.000000
max        170.000000
Name: file_id_api_nunique, dtype: float64
```

文件调用 API 的类别对大部分在 100 以内，最少是 1 个，最多是 170 个。
然后分析 file_id_api_nunique 和标签 label 的关系。

```
train_analysis.loc[train_analysis.file_id_api_nunique >=100]['label'].value_counts().sort_index().plot(kind = 'bar')
plt.title('File with api nunique >= 100')
plt.xlabel('label')
plt.ylabel('label_number')
```

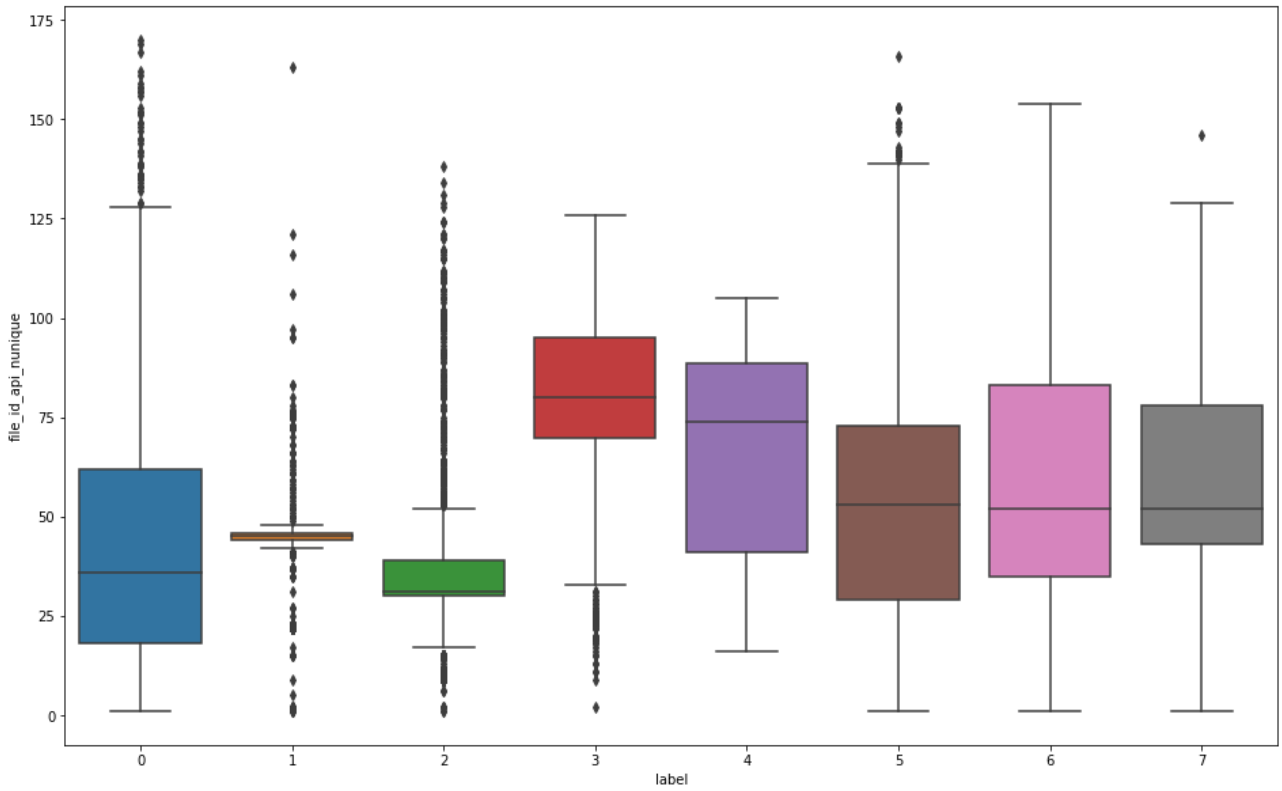
运行结果：



结果可见，第 5 类病毒调用不同 API 的次数最多。
用 boxplot 检查每个 label 对应的 file_id_api_nunique 分布。

```
plt.figure(figsize=[16,10])
sns.boxplot(x =train_analysis['label'], y = train_analysis['file_id_api_nunique'])
```

运行结果：



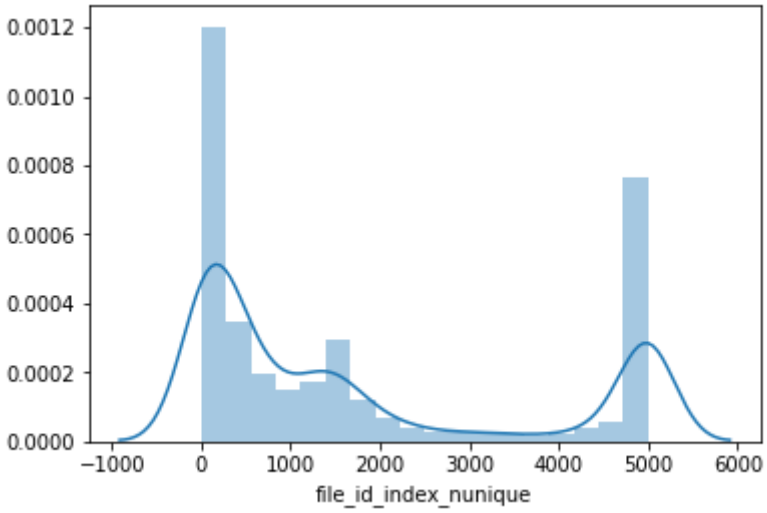
从图中可以看出，第 3 类病毒调用不同 API 的次数相对较多，第 2 类病毒调用不同 API 的次数最少；第 4，6，7 类病毒的离群点较少，第 1 类病毒离群点最多，第 3 类病毒的离群点主要在下方，第 0 类和第 5 类离群点集中在上方。

（4）首先通过 file_id_index_nunique 和 file_id_index_max 两个统计特征，分析变量 file_id 和 index 之间的关系。

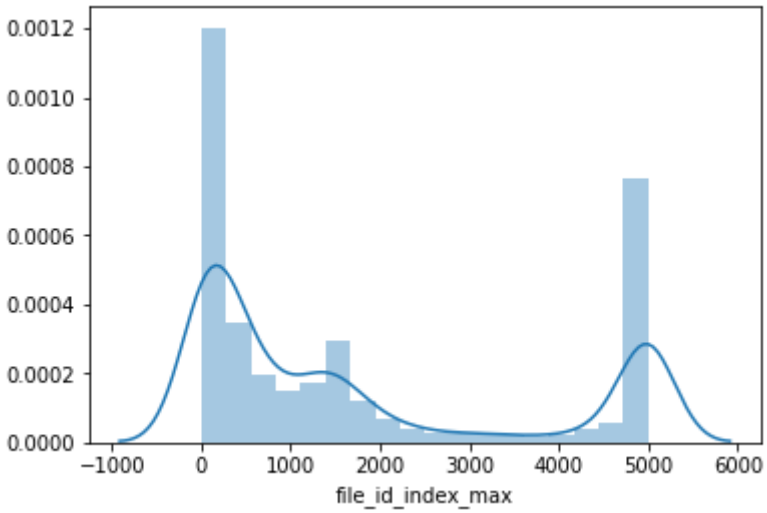
```
dic_ = train.groupby('file_id')['index'].nunique().to_dict()
train_analysis['file_id_index_nunique'] = train_analysis['file_id'].map(dic_).values
```

```
train_analysis['file_id_index_nunique'].describe()
```

```
count    13887.000000
mean      1770.645136
std       1934.542352
min          1.000000
25%        135.000000
50%        924.000000
75%       3628.000000
max       5001.000000
Name: file_id_index_nunique, dtype: float64
```



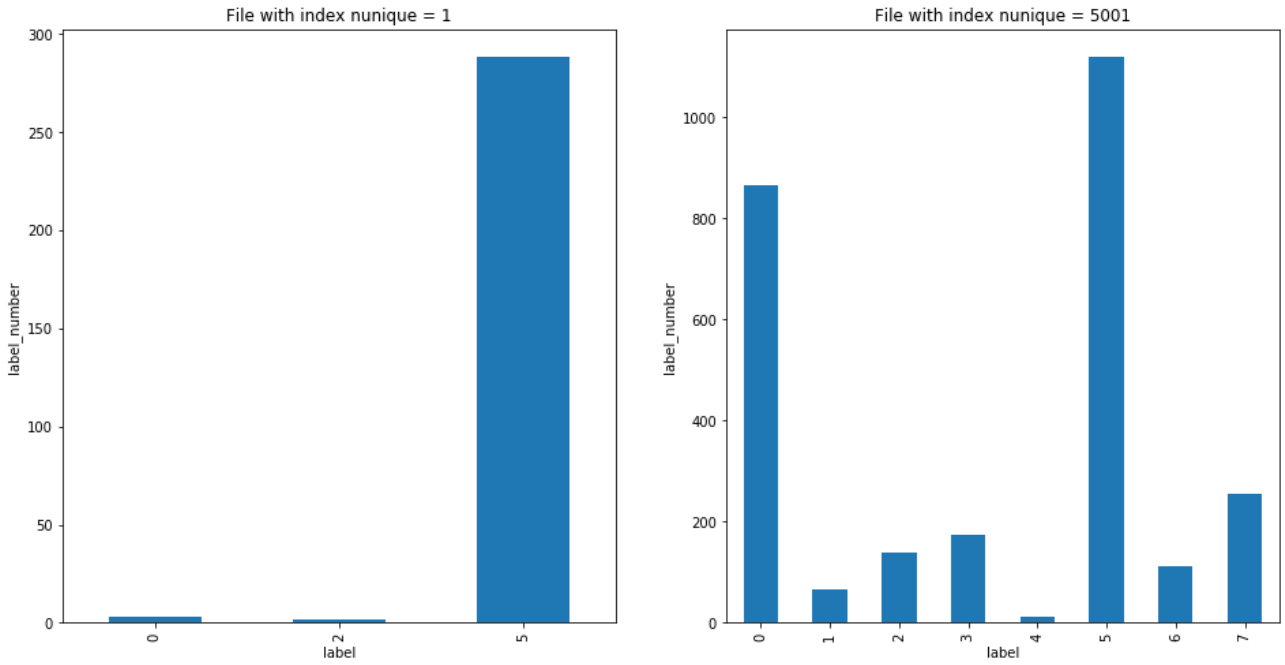

```
dic_ = train.groupby('file_id')['index'].max().to_dict()
train_analysis['file_id_index_max'] = train_analysis['file_id'].map(dic_).values
```



从图中可以看出，文件调用 index 有两个极端：一个在 1 附近，另一个在 5000 附近。

然后分析 file_id_index_nunique 和 file_id_index_max 与标签 label 之间的关系。

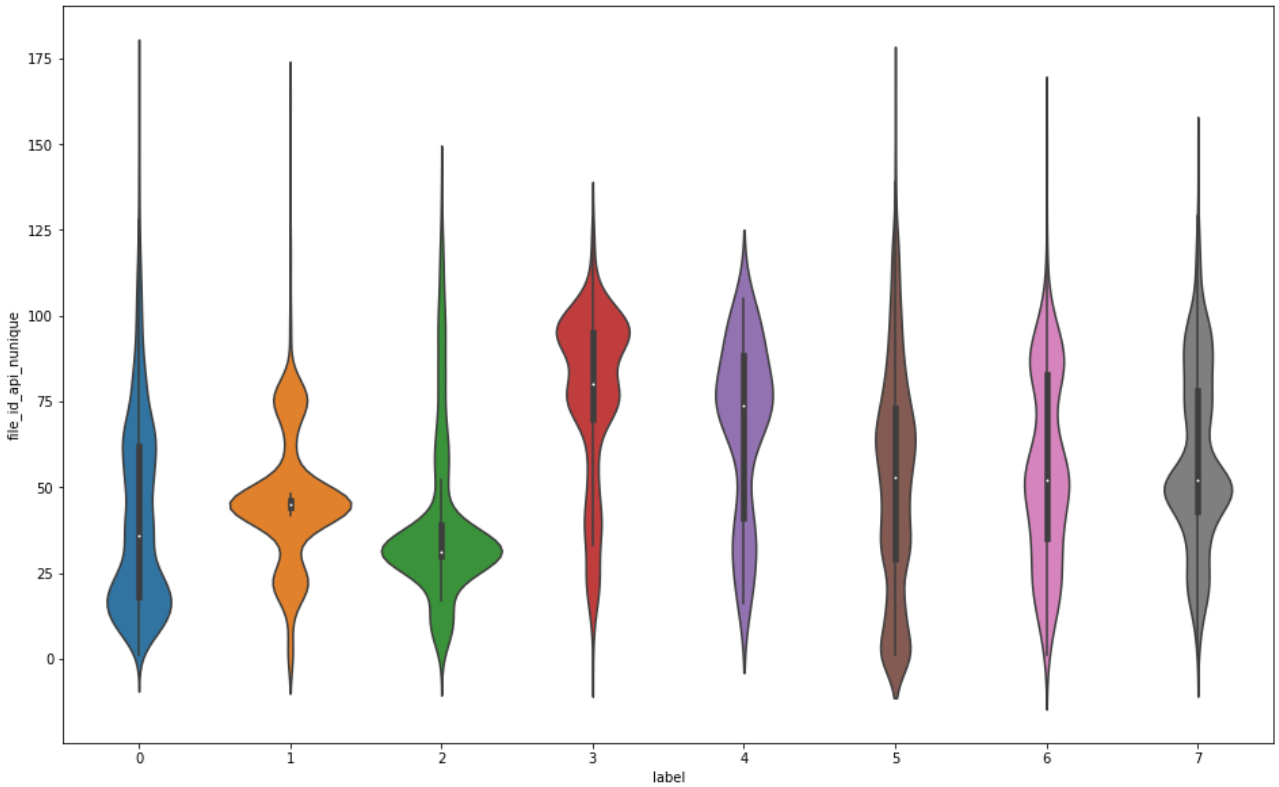
```
plt.figure(figsize=[16,8])
plt.subplot(121)
train_analysis.loc[train_analysis.file_id_index_nunique == 1]['label'].value_counts().sort_index().plot(kind = 'bar')
plt.title('File with index nunique = 1')
plt.xlabel('label')
plt.ylabel('label_number')

plt.subplot(122)
train_analysis.loc[train_analysis.file_id_index_nunique == 5001]['label'].value_counts().sort_index().plot(kind = 'bar')
plt.title('File with index nunique = 5001')
plt.xlabel('label')
plt.ylabel('label_number')
```



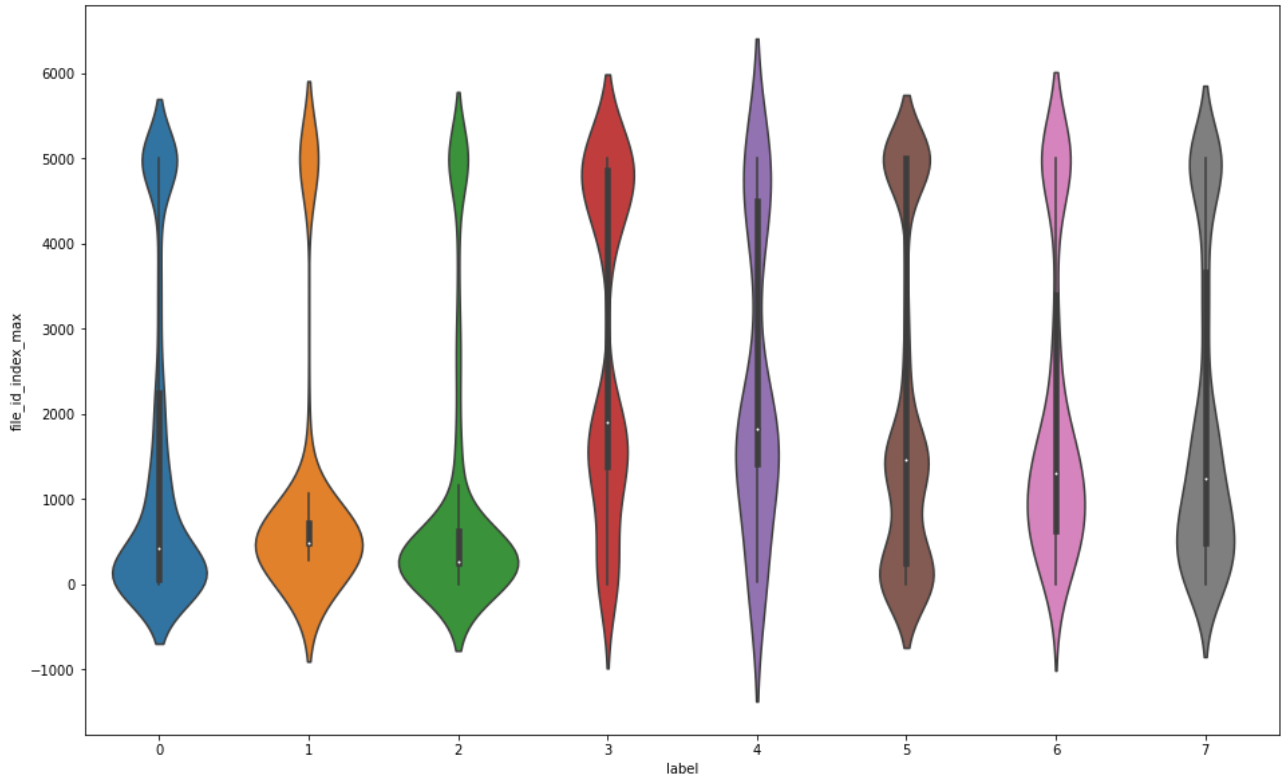
从图中可以看出，在文件顺序编号只有一个时，文件标签只会是 0、2、5，且最大概率是 5。

通过绘制小提琴图、分类散点图分析，代码和结果如下：

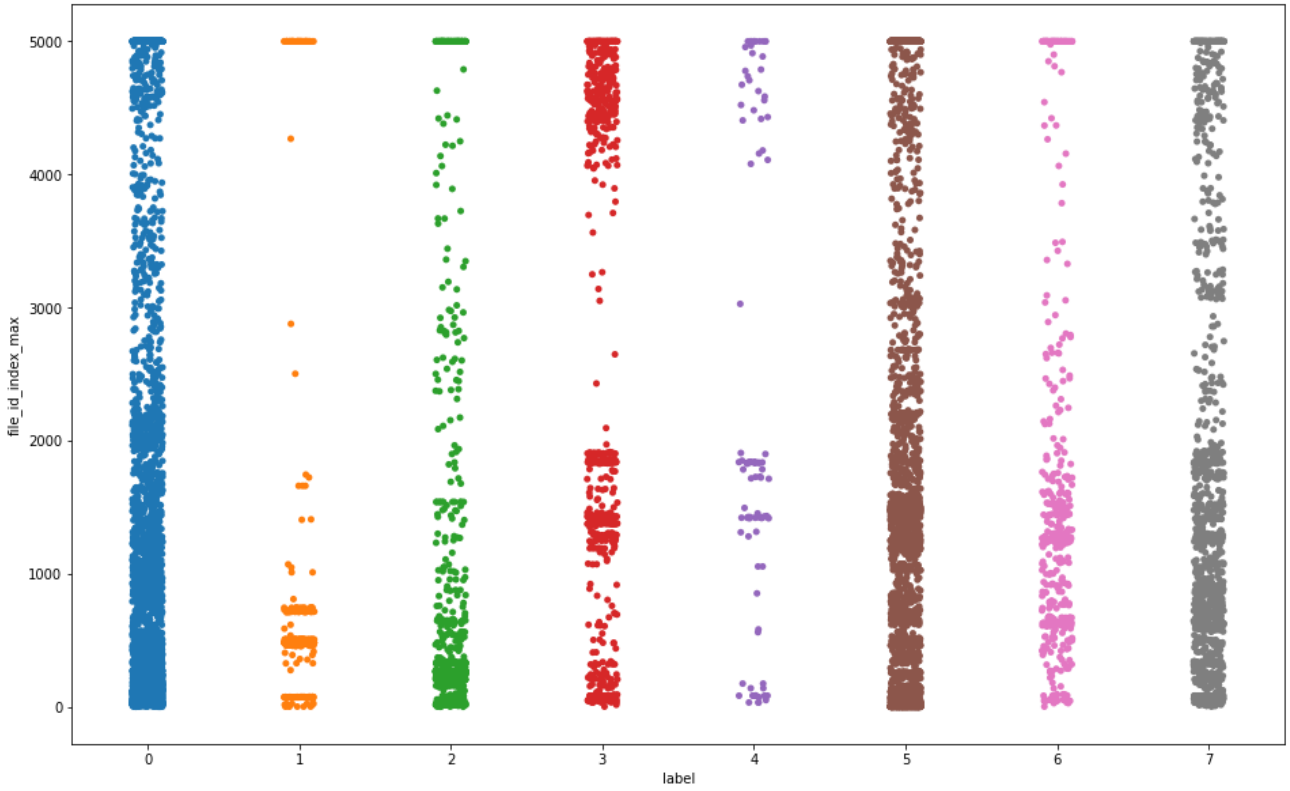
```
plt.figure(figsize=[16,10])
sns.violinplot(x =train_analysis['label'], y = train_analysis['file_id_api_nunique'])#绘制小提琴图
```




```
plt.figure(figsize=[16,10])
sns.violinplot(x =train_analysis['label'], y = train_analysis['file_id_index_max'])
```




```
plt.figure(figsize=[16,10])
#绘制散点图
sns.stripplot(x =train_analysis['label'], y = train_analysis['file_id_index_max'])
```



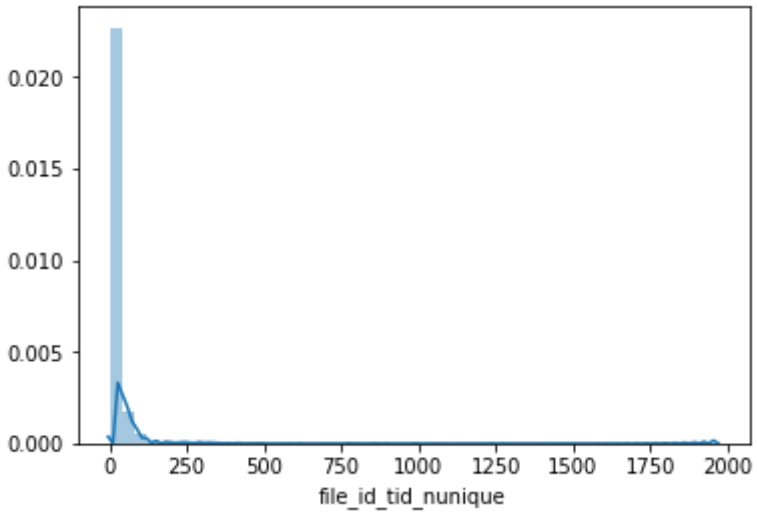
从图中可以看出，第 3 类病毒调用不同 index 次数的平均值最大；第 2 类病毒调用不同 index 次数的平均值最小；第 5、6、7 类病毒表用不同 index 次数的平均值相似。

（5）首先通过 file_id_tid_nunique 和 file_id_tid_max 两个统计特征，分析变量 file_id 和 tid 之间的关系，代码和运行结果如下：

```
dic_ = train.groupby('file_id')['tid'].nunique().to_dict()
train_analysis['file_id_tid_nunique'] = train_analysis['file_id'].map(dic_).values
```

```
train_analysis['file_id_tid_nunique'].describe()
```

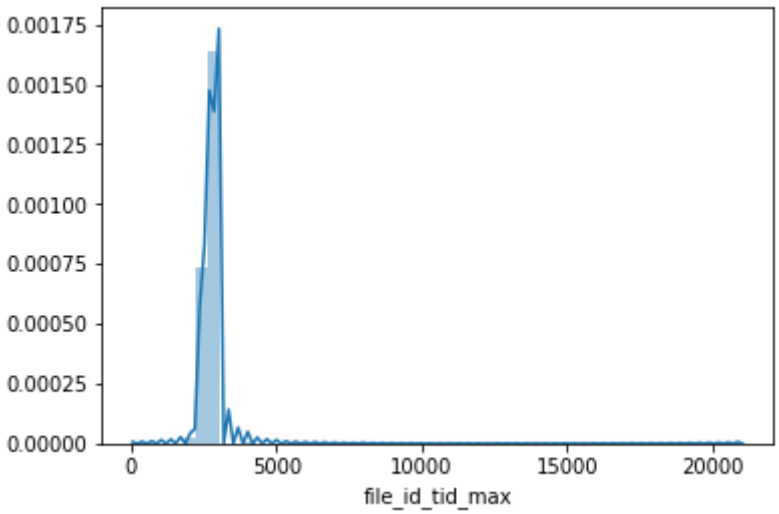
```
count    13887.000000
mean        18.797724
std         55.212772
min          1.000000
25%          2.000000
50%          4.000000
75%         17.000000
max       1965.000000
Name: file_id_tid_nunique, dtype: float64
```




```
dic_ = train.groupby('file_id')['tid'].max().to_dict()
train_analysis['file_id_tid_max'] = train_analysis['file_id'].map(dic_).values
```

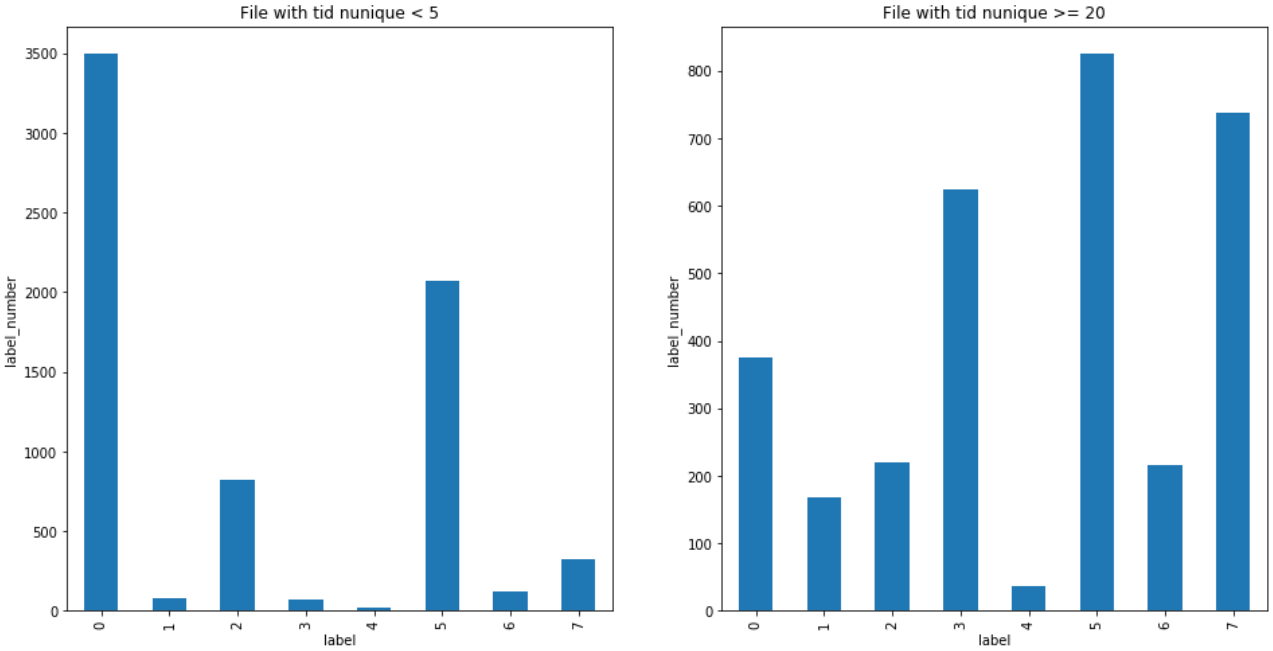
```
train_analysis['file_id_tid_max'].describe()
```

```
count    13887.000000
mean      2782.530424
std        420.516683
min        184.000000
25%       2612.000000
50%       2792.000000
75%       2964.000000
max      20896.000000
Name: file_id_tid_max, dtype: float64
```



线程的调用 75%都在 20 个以下,最多的调用了 1965 个线程，从图中可见线程基本都在 200 个以内。文件中 90%的文件调用的线程的最大值都小于 5000。

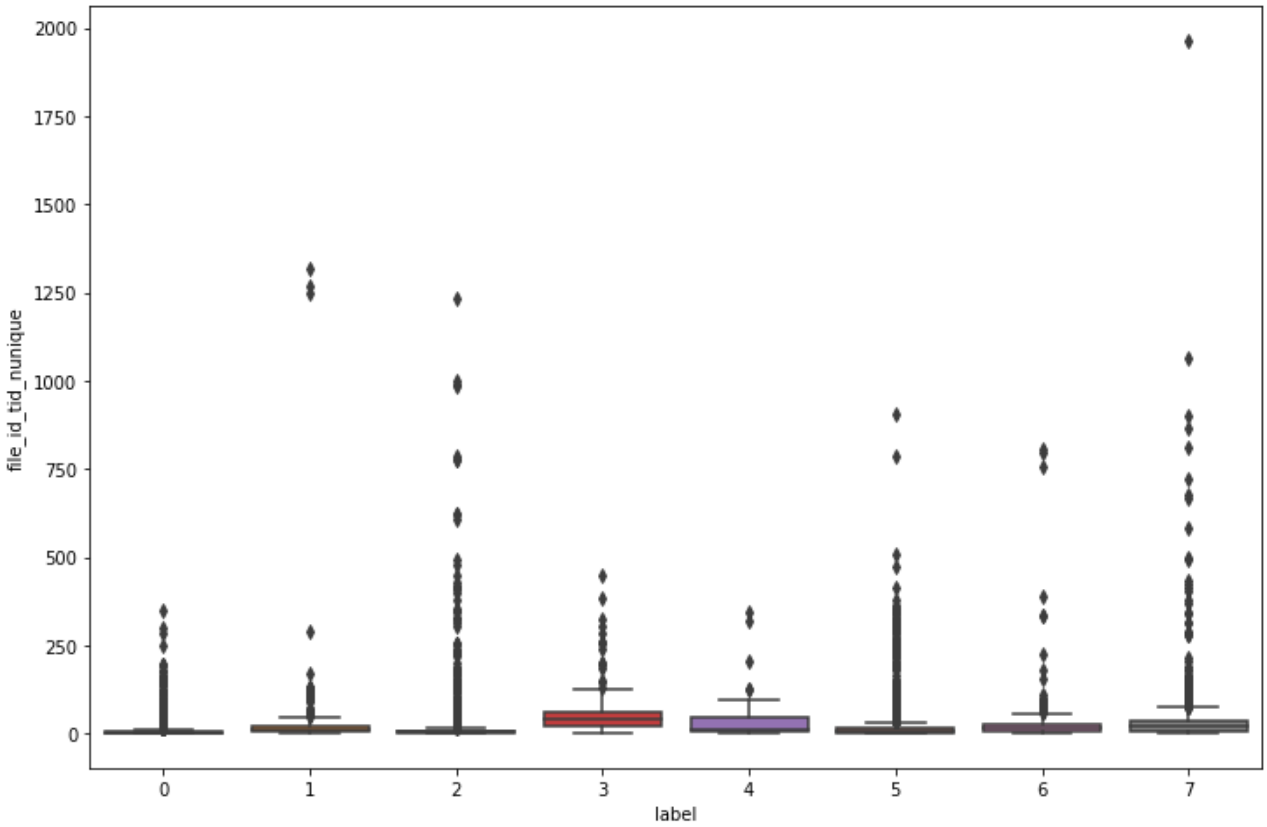
然后分析 file_id_tid_nunique 和 file_tid_max 与标签 label 的关系。

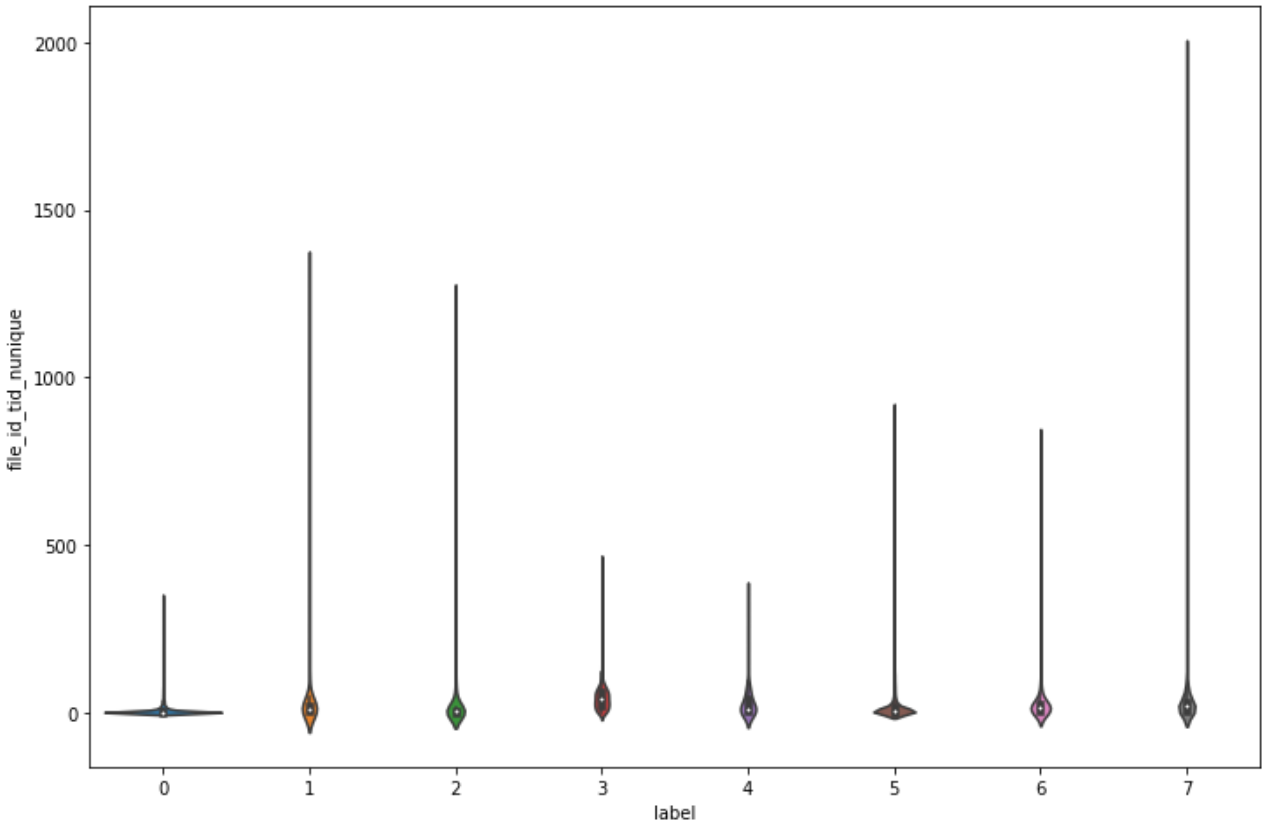


从图中可以看出 DDoS 木马（3 号）和木马病毒（7 号）经常会调用多个线程,可能木马经常会用一些线程来监听各种信息,所以线程的调用次数会较多。

通过箱线图和小提琴图进一步分析。

```
plt.figure(figsize=[12,8])
sns.boxplot(x =train_analysis['label'], y = train_analysis['file_id_tid_nunique'])#盒图
```





从图中可以得出结论，所有文件调用的线程都相对较少；第 7 类病毒调用的线程数的范围最大；第 0、3、4 类调用不同线程数相似。

（6）分析 API 变量和 label 的关系，因为不同的病毒对 api 的调用不一样,所以探索 api 的调用情况：主要看 label 调用最多的 api，代码和运行结果如下：

```
train['api_label'] = train['api'] + '_' + train['label'].astype(str)
dic_ = train['api_label'].value_counts().to_dict()
```

```
df_api_label = pd.DataFrame.from_dict(dic_,orient = 'index').reset_index()
df_api_label.columns = ['api_label', 'api_label_count']
```

```
df_api_label['label'] = df_api_label['api_label'].apply(lambda x:int(x.split('_')[-1]))
```

```
labels = df_api_label['label'].unique()
for label in range(8):
    print('*' * 50, label,'*' * 50)
    print(df_api_label.loc[df_api_label.label == label].sort_values('api_label_count').iloc[-5:][['api_label','api_label_count']])
    print('*' * 103)
```

每个类型统计结果：

```
************************************************** 0 **************************************************
                  api_label  api_label_count
20     CryptDecodeObjectEx_0           808724
19           RegOpenKeyExW_0           815653
11  LdrGetProcedureAddress_0          1067389
9                  NtClose_0          1150929
5          RegQueryValueExW_0          1793509

************************************************** 1 **************************************************
                   api_label  api_label_count
180             RegCloseKey_1            83134
160              NtReadFile_1           101051
102  LdrGetProcedureAddress_1           199218
75                  NtClose_1           268922
72          RegQueryValueExW_1           283562

************************************************** 2 **************************************************
                  api_label  api_label_count
47              NtReadFile_2           429733
34          Process32NextW_2           609066
28        RegQueryValueExW_2           704073
27  LdrGetProcedureAddress_2           711169
12                 NtClose_2          1044951
```

********** 3 **********

	api_label	api_label_count
32	NtClose_3	614574
31	RegCloseKey_3	616165
25	RegQueryValueExW_3	749380
24	LdrGetProcedureAddress_3	762139
13	RegOpenKeyExW_3	937860

********** 4 **********

	api_label	api_label_count
270	RegCloseKey_4	43475
257	LdrGetProcedureAddress_4	46977
238	RegQueryValueExW_4	53934
236	NtClose_4	54087
211	RegOpenKeyExW_4	68092

********** 5 **********

	api_label	api_label_count
6	GetSystemMetrics_5	1381193
3	NtClose_5	2076013
2	GetCursorPos_5	2397779
1	Thread32Next_5	4973322
0	LdrGetProcedureAddress_5	5574419

********** 6 **********

	api_label	api_label_count
105	RegOpenKeyExW_6	193608
99	RegQueryValueExW_6	206940
82	NtClose_6	254385
40	LdrGetProcedureAddress_6	503839
8	NtDelayExecution_6	1197309

********** 7 **********

	api_label	api_label_count
18	RegQueryValueExW_7	837933
17	Process32NextW_7	856303
14	NtDelayExecution_7	937033
10	NtClose_7	1120847
4	LdrGetProcedureAddress_7	1839155

****

可得结论：LdrGetProcedureAddress，所有病毒和正常文件均调用多；第 5 类病毒(感染型病毒)调用 Thread32Next 较多；第 6、7 类病毒(后门程序&木马程序)调用 NtDelayExecution 较多；第 2、7 类病毒(挖矿程序&木马程序)调用 Process32NextW 较多。

3.2 特征工程构造

在 baseline 的基础上，通过对多变量的交叉分析，可以增加新的 pivot 特征：
（1）每个 API 调用线程 tid 的次数
（2）每个 API 调用不同线程 tid 的次数

```
from tqdm import tqdm, tqdm_notebook
#特征工程进阶
def api_pivot_count_features(df):
    tmp = df.groupby(
        ['file_id','api']
    )['tid'].count().to_frame('api_tid_count').reset_index()
    tmp_pivot = pd.pivot_table(data = tmp,index='file_id',columns='api',values='api_tid_count',fill_value=0)

    tmp_pivot.columns = [tmp_pivot.columns.names[0]+'_pivot_'+str(col) for col in tmp_pivot.columns]

    tmp_pivot.reset_index(inplace=True)
#     tmp_pivot = meomory_process._memory_process(tmp_pivot)

    return tmp_pivot

def api_pivot_nunique_features(df):
    tmp = df.groupby(
        ['file_id', 'api']
    )['tid'].nunique().to_frame('api_tid_nunique').reset_index()
    tmp_pivot = pd.pivot_table(data=tmp, index='file_id', columns='api', values='api_tid_nunique', fill_value=0)
    tmp_pivot.columns = [tmp_pivot.columns.names[0] + '_pivot_' + str(col) for col in tmp_pivot.columns]
    tmp_pivot.reset_index(inplace=True)
#     tmp_pivot = meomory_process._memory_process(tmp_pivot)
    return tmp_pivot
```

3.3 基于 LightGBM 的模型验证

（1）获取标签

```
#获取标签
train_label = train[['file_id','label']].drop_duplicates(subset = ['file_id','label'],keep = 'first')
test_submit = test[['file_id']].drop_duplicates(subset = ['file_id'],keep = 'first')

train_label
test_submit
```

	file_id
0	1
97	2
1458	3
1474	4
1667	5
...	...
79277890	12951
79278179	12952
79278291	12953
79283386	12954
79286337	12955

12955 rows × 1 columns

（2）训练集与测试集的构建

```
#训练集和测试集的构建
train_data = train_label.merge(simple_train_feature1,on='file_id',how = 'left')
train_data = train_data.merge(simple_train_feature2,on='file_id',how = 'left')
print(simple_train_feature3)
train_data = train_data.merge(simple_train_feature3,on='file_id',how = 'left')
train_data = train_data.merge(simple_train_feature4,on='file_id',how = 'left')

test_submit = test_submit.merge(simple_test_feature1,on='file_id',how = 'left')
test_submit = test_submit.merge(simple_test_feature2,on='file_id',how = 'left')
test_submit = test_submit.merge(simple_test_feature3,on='file_id',how = 'left')
test_submit = test_submit.merge(simple_test_feature4,on='file_id',how = 'left')
```

```
       file_id  api_pivot_AssignProcessToJobObject  \
0            1                                   0
1            2                                   0
2            3                                   0
3            4                                   0
4            5                                   0
...        ...                                 ...
13882    13883                                   0
13883    13884                                   0
13884    13885                                   0
13885    13886                                   0
13886    13887                                   0
```

（3）评估指标构建

```
def lgb_logloss(preds,data):
    labels = data.get_label()
    classes = np.unique(labels)
    preds_prob = []
    for i in range(len(classes)):
        preds_prob.append(preds[i*len(labels):(i+1)*len(labels)])
    preds_prob =  np.vstack(preds_prob)
    loss=[]
    for i in range(preds_prob.shape[1]):#样本个数
        sum = 0
        for j in range(preds_prob.shape[0]):#类别个数
            pred = preds_prob[j,i]#第i个样本预测为第j类的概率
            if j==labels[i]:
                sum+=np.log(pred)
            else:
                sum+=np.log(1-pred)
        loss.append(sum)
    return 'loss is:',-1*(np.sum(loss)/preds_prob.shape[1]),False
```

（4）模型采用 5 折交叉验证方式，去预测均值作为最终结果：

```
#线下验证
train_features = [col for col in train_data.columns if col not in ['label','file_id']]
train_label = 'label'
from sklearn.model_selection import StratifiedKFold,KFold
params = {
    'task':'train',
    'num_leaves':255,
    'objective':'multiclass',
    'num_class':8,
    'min_data_in_leaf':50,
    'learning_rate':0.05,
    'feature_fraction':0.85,
    'bagging_fraction':0.85,
    'bagging_freq':5,
    'max_bin':128,
    'random_state':100,
    'verbosity': -1
}
folds =KFold(n_splits=5,shuffle=True,random_state=15)
oof = np.zeros(len(train))

predict_res = 0
models = []
import lightgbm as lgb
for fold,(trn_id,val_id) in enumerate(folds.split(train_data)):
    print("fold n {}".format(fold))
    trn_data = lgb.Dataset(train_data.iloc[trn_id][train_features],label=train_data.iloc[trn_id][train_label].values)
    val_data = lgb.Dataset(train_data.iloc[val_id][train_features], label=train_data.iloc[val_id][train_label].values)
    clf = lgb.train(params,trn_data,num_boost_round=2000,valid_sets=[trn_data,val_data],verbose_eval=50,early_stopping_rounds=100,fev
    models.append(clf)
```

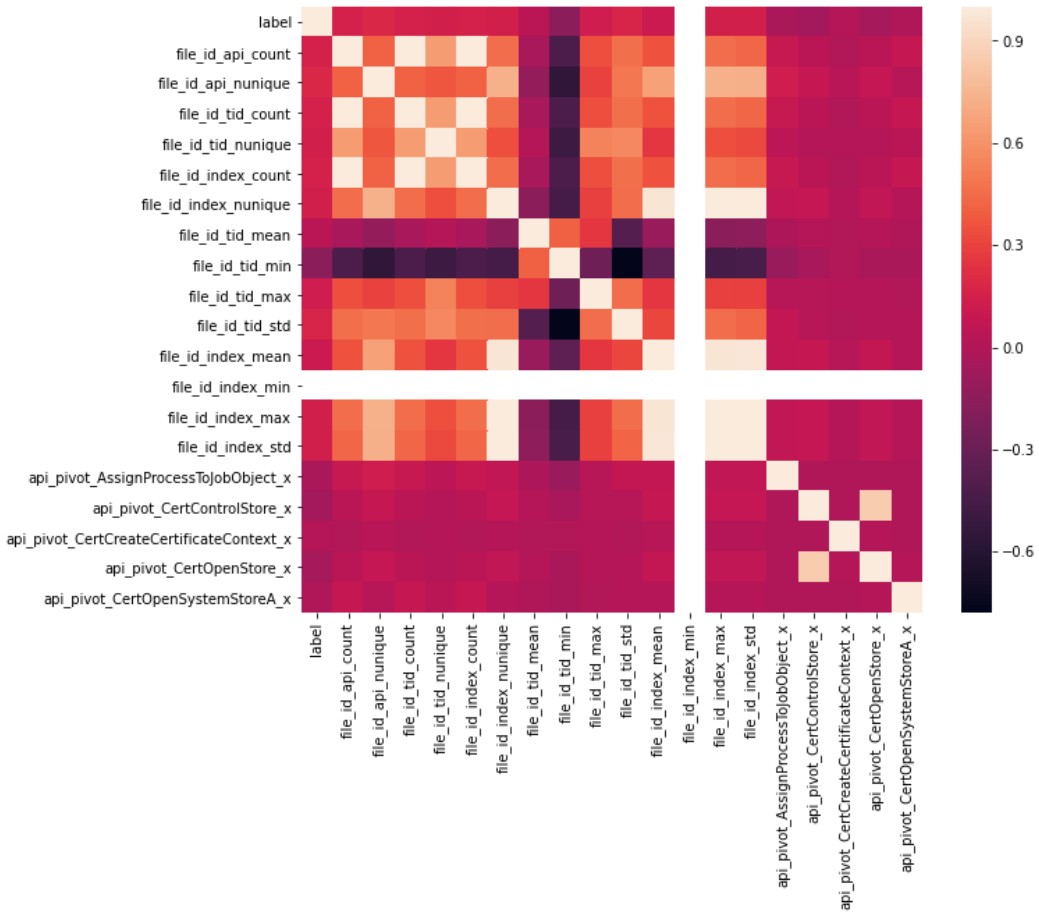
（5）运行结果：

```
fold n 0
Training until validation scores don't improve for 100 rounds
[50]    training's multi_logloss: 0.183626      training's loss is:: 0.341959   valid_1's multi_logloss: 0.341377       valid_1's loss i
s:: 0.586353
[100]   training's multi_logloss: 0.0363207     training's loss is:: 0.0715909  valid_1's multi_logloss: 0.329024       valid_1's loss i
s:: 0.569227
[150]   training's multi_logloss: 0.00770209    training's loss is:: 0.0153632  valid_1's multi_logloss: 0.374909       valid_1's loss i
s:: 0.654556
Early stopping, best iteration is:
[73]    training's multi_logloss: 0.0871876     training's loss is:: 0.168299   valid_1's multi_logloss: 0.32096        valid_1's loss i
s:: 0.552405
fold n 1
Training until validation scores don't improve for 100 rounds
[50]    training's multi_logloss: 0.1821        training's loss is:: 0.339742   valid_1's multi_logloss: 0.371503       valid_1's loss i
s:: 0.626954
[100]   training's multi_logloss: 0.0361332     training's loss is:: 0.0712726  valid_1's multi_logloss: 0.361585       valid_1's loss i
s:: 0.612793
[150]   training's multi_logloss: 0.00768985    training's loss is:: 0.0153376  valid_1's multi_logloss: 0.415334       valid_1's loss i
s:: 0.707089
Early stopping, best iteration is:
[72]    training's multi_logloss: 0.0889021     training's loss is:: 0.171715   valid_1's multi_logloss: 0.350917       valid_1's loss i
s:: 0.593296
fold n 2
Training until validation scores don't improve for 100 rounds
[50]    training's multi_logloss: 0.175981      training's loss is:: 0.328958   valid_1's multi_logloss: 0.388225       valid_1's loss i
s:: 0.657502
[100]   training's multi_logloss: 0.0344636     training's loss is:: 0.068053   valid_1's multi_logloss: 0.383424       valid_1's loss i
s:: 0.65584
[150]   training's multi_logloss: 0.00713817    training's loss is:: 0.0142411  valid_1's multi_logloss: 0.450465       valid_1's loss i
s:: 0.778353
Early stopping, best iteration is:
```

（6）结果分析：

特征相关性分析：计算特征之间的相关性系数，并用热力图可视化显示。这里采样 10 000 个样本，观察其中 20 个特征的线性相关性。

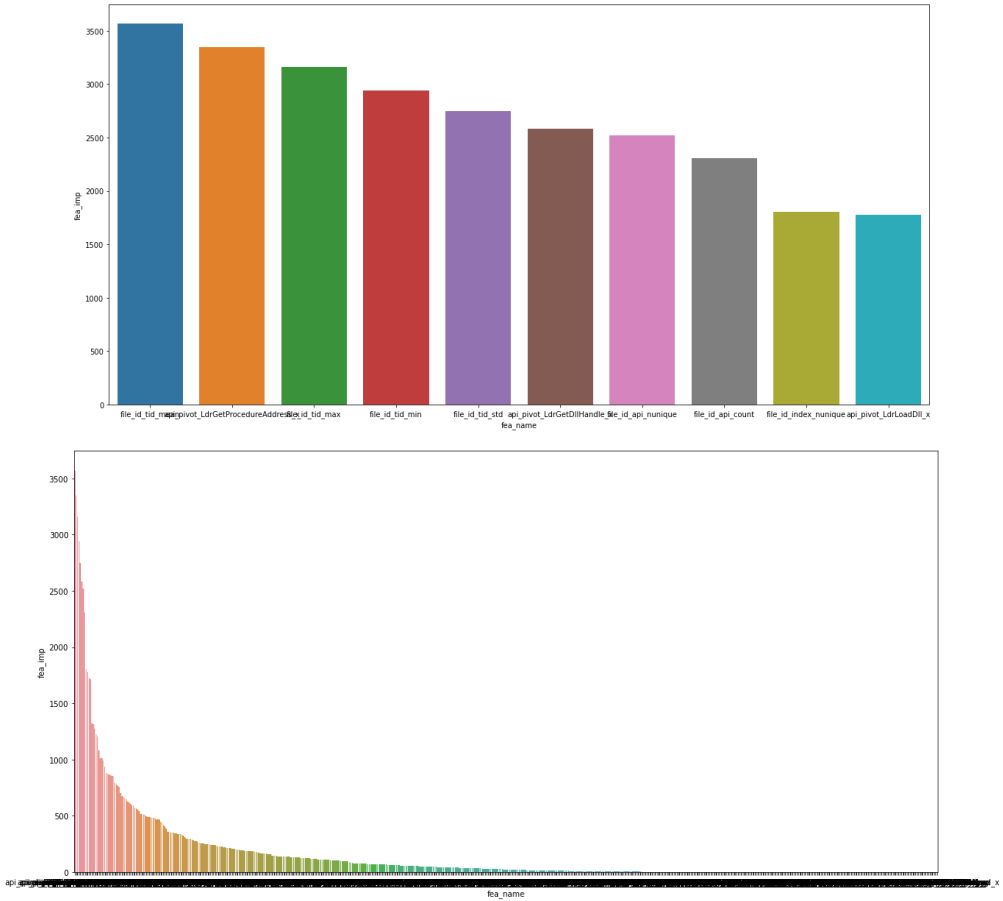
```
plt.figure(figsize=[10,8])
sns.heatmap(train_data.iloc[:10000,1:21].corr())
plt.show()
```



通过查看特征变量与标签 label 的相关性，再次验证数据探索 EDA 部分的结论：每个文件调用的 API 次数与病毒类型是强相关的。

特征重要性分析的代码和结果如下：

```
#特征重要性分析
import seaborn as sns
import matplotlib.pyplot as plt
feature_importance = pd.DataFrame()
feature_importance['fea_name'] = train_features
feature_importance['fea_imp' ] = clf.feature_importance()
feature_importance = feature_importance.sort_values('fea_imp',ascending=False)
plt.figure(figsize=[20,10,])

sns.barplot(x=feature_importance.iloc[:10]['fea_name'],y=feature_importance.iloc[:10]['fea_imp'])
plt.show()


plt.figure(figsize=[20,10,])

sns.barplot(x=feature_importance['fea_name'],y=feature_importance['fea_imp'])
plt.show()
```



结果也验证了 API 的调用次数及 API 的调用类别书是最重要的两个特征。

3.4TextCNN 建模

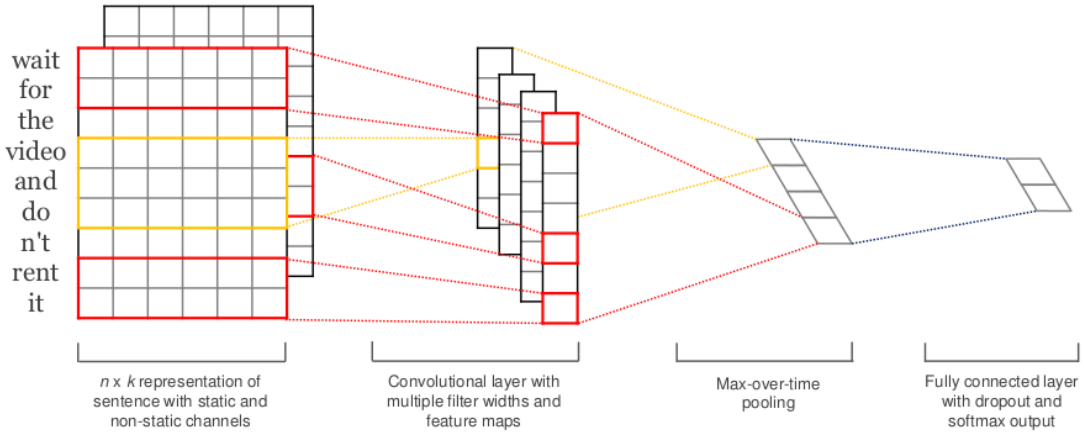
	file_id	label	api	tid	index	api_idx
0	1	5	LdrLoadDll	2488	0	1
1	1	5	LdrGetProcedureAddress	2488	1	2
2	1	5	LdrGetProcedureAddress	2488	2	2
3	1	5	LdrGetProcedureAddress	2488	3	2
4	1	5	LdrGetProcedureAddress	2488	4	2

	file_id	seq
0	1	[46.0, 106.0, 113.0, 173.0, 46.0, 106.0, 113.0...
97	2	[46.0, 46.0, 46.0, 4.0, 37.0, 37.0, 37.0, 37.0...
1458	3	[55.0, 13.0, 18.0, 63.0, 63.0, 41.0, 41.0, 41....
1474	4	[1.0, 2.0, 108.0, 150.0, 151.0, 4.0, 1.0, 2.0,...
1667	5	[55.0, 9.0, 10.0, 9.0, 12.0, 2.0, 2.0, 2.0, 2....

此外，我们还用深度学习对该问题进行建模，将该问题进行相应的问题转换，由 API 序列的调用可知：每个文件都对应一个 API 的调用序列，将每个 API 的序列进行拼接，将本问题转变为一个文本分类问题。

3.4.1 数据预处理

（1）字符串转换为数字：对 API 构建映射表将其转换为数字
（2）获取每个文件对应的字符串序列：获取每个文件调用的 API 序列



3.4.2TextCNN 网络结构

使用 Keras 深度学习框架进行实现：

```
def TextCNN(max_len,max_cnt,embed_size, num_filters,kernel_size,conv_action, mask_zero):

    _input = Input(shape=(max_len,), dtype='int32')
    _embed = Embedding(max_cnt, embed_size, input_length=max_len, mask_zero=mask_zero)(_input)
    #_embed = SpatialDropout1D(0.15)(_embed)
    _embed = SpatialDropout1D(0.25)(_embed)
    warppers = []

    for _kernel_size in kernel_size:
        conv1d = Conv1D(filters=num_filters, kernel_size=_kernel_size, activation=conv_action)(_embed)
        warppers.append(GlobalMaxPooling1D()(conv1d))

    fc = concatenate(warppers)
    fc = Dropout(0.5)(fc)
    #fc = BatchNormalization()(fc)
    fc = Dense(256, activation='relu')(fc)
    fc = Dropout(0.25)(fc)
    #fc = BatchNormalization()(fc)
    preds = Dense(8, activation = 'softmax')(fc)

    model = Model(inputs=_input, outputs=preds)

    model.compile(loss='categorical_crossentropy',
        optimizer='adam',
        metrics=['accuracy'])
    return model
```

3.4.3TextCNN 训练和测试

运行结果：

```
174/174 [==============================] - 816s 5s/step - loss: 0.2715 - accuracy: 0.9023 - val_loss: 0.3326 - va
l_accuracy: 0.8970
Epoch 22/100
174/174 [==============================] - 816s 5s/step - loss: 0.2825 - accuracy: 0.8988 - val_loss: 0.3409 - va
l_accuracy: 0.8895
Epoch 23/100
174/174 [==============================] - 816s 5s/step - loss: 0.2849 - accuracy: 0.9021 - val_loss: 0.3342 - va
l_accuracy: 0.8970
```

Logloss=0.529231

4. 结果分析

4.1 结果

我们分别使用 LightGBM 和 TextCNN 模型预测了测试集，最终结果如表所示，可以看出，LightGBM 和 TextCNN 均可以取得良好的预测结果。

Model	Result
LGB	0.568286
CNN	0.529231
Baseline	1.087292

4.2 总结与展望

根据领域知识来了解 API 序列的特点是非常重要的，但由于我们对安全性领域了解不多，所以只能从 NLP 和统计学习的角度提取特征。

在模型的选择上用了 LGB 和 CNN，后续可以尝试其他方法与模型融合，最后得到的结果再 bagging，这样可能能得到更好的效果。