**中国矿业大学计算机学院**

**2019 级本科生课程作业**

|  |  |
| --- | --- |
| 课程名称 | 信息内容安全 |
| 报告题目 | 基于机器学习的恶意软件检测技术 |
| 报告时间 | 2022年6月26日 |
| 姓 名 | 许万鹏 |
| 学 号 | 05191643 |
| 任课教师 | 曹天杰 |

**2021-2022(二)《信息内容安全》评分表**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **考核类别** | **考核内容** | **支撑课程目标** | **试题类型与分值比例** | **分数** |
| 结课考核 | 课程报告（论文综述、设计、实现、写作规范） | **目标3**：掌握信息内容安全的基础知识，针对具体问题和要求选择正确的技术路线，通过在实验环境中进行仿真实验并能根据算法特点进行攻击测试和综合性能评价，得到具有参考价值的结论。 | 课程报告，100% |  |
| 过程考核 | 1.基本概念、原理 | **目标1**:掌握信息内容安全的基本概念、分类、原理和相关技术，能够根据课程基本知识对信息内容安全领域出现的问题进行归类、分析、并有初步分析和解决问题的能力。 | 系统演示及解说，30% |  |
| 2.系统设计与分析 | **目标2**：掌握信息内容安全处理相关的理论、技术以及健全的评价体系，能够根据具体问题分析算法、设计算法、实现算法并能综合评价算法。 | PPT讲解与答辩，50% |  |
| 3. 基本概念、原理 |  | 作业或测试，20% |  |
| 结课考核与过程考核比例 | | 结课考核：60% | 过程考核：40% |  |

评阅人：

2022年7月10日

基于机器学习的恶意软件检测技术

Malware Detection Technology Based on Machine Learning

作 者：许万鹏

摘要

信息安全是当今互联网社会必须重视的问题，恶意软件的高发增长给互联网信息安全带来了很大的风险。有效地检测恶意软件成为现今必须克服的问题。目前基于特征码的恶意软件检测技术己经很难全面检测数量如此庞大的恶意软件，特别是无法检测新型恶意软件。基于行为的恶意软件检测分析技术应运而生，近年来基于Windows API调用行为的恶意软件动态分析技术成为了研究的热点。本文就基于Windows API调用行为的恶意软件检测技术进行了研究，提高恶意软件的检测率。

本文结合文本分析技术和机器学习技术，对软件的Windows API调用行为进行分析研究。本文立足阿里天池平台的比赛——阿里云安全恶意程序检测，对软件的Windows API调用行为进行数据探索，对恶意软件的调用行为构建特征。接下来，针对构建的特征集，使用三种基线模型进行预测，最终使用TextCNN算法构建模型并提交。同时，本文对软件Windows API调用频率和Windows API调用之间的关联关系进行了探究。再者，使用TF-IDF和CNN-LSTM对样本进行文本分类和预测。接着，使用了文本分类评价指标对实验的结果进行分析与评价。最终，本文训练的模型可以识别正常文件、勒索病毒、挖矿程序、DDoS木马、蠕虫病毒、感染型病毒、后门程序、木马程序共七种恶意软件。

本文多方面的实验结果都表明，Windows API调用日志包含的内容极为丰富，基于Windows API调用行为的恶意软件动态检测技术，可取得更好的恶意软件检测效果，具有较强实用价值。本文在后续的研究中会继续探究Windows API调用的其他方面对恶意软件检测的影响。

**关键词**：Windows API调用；恶意软件动态检测；LightGBM；TextCNN

Abstract

Information security is a very important problem that we should pay attention to innowaday's internet society. The number of malwares is growing fast and it is bringing a bigrisk to the information security of internet It's a big problem that we need to solve to detectthe malwares efficiently. It's difficult for the signature code based malwares detecting technology to technology to detect the huge number of malwares especially the new types. So the behaviors based malwares detecting technology is appearing. It's a hot point to research on the malwares deleting based on the Windows API call behaviors. I did some research on theWindows API call behaviors of malwares and benign softwares to improve the detecting rate of malwares.

This paper combines text analysis technology and machine learning technology to analyze and study the Windows API invocation behavior of software. In this paper, based on the aliyun security malicious program detection, a competition on the aliyun TIANCHI platform, data exploration of the Windows API invocation behavior of software is conducted to construct features for the invocation behavior of malware. Next, the predictions are performed using three baseline models for the constructed feature sets, and the models are finally constructed and submitted using the TextCNN algorithm. Meanwhile, the correlation between the frequency of software Windows API calls and Windows API calls is explored in this paper. Further, text classification and prediction of samples are performed using TF-IDF and CNN-LSTM. Then, text classification evaluation metrics are used to analyze and evaluate the results of the experiments. Finally, the model trained in this paper can identify a total of seven types of malware: normal files, ransomware, mining programs, DDoS Trojans, worms, infectious viruses, backdoor programs, and Trojan horses.

We find that the Windows API call logfiles have rich informatio through our research. It's a better and useful way to detect malwares based on the Windows API call behaviors. In the follow-up study, I will go on the research on the other aspects of the Windows API call logfiles.

**Keywords**: Windows API Call, Malware Deletion, LightGBM, TextCNN

**目 录**

[摘要 IV](#_Toc107291370)

[目录 VI](#_Toc107291372)

[1 绪论 1](#_Toc107291374)

[1.1 研究背景及意义 1](#_Toc107291376)

[1.2 国内外研究现状 2](#_Toc107291377)

[1.3 论文组织结构 4](#_Toc107291378)

[2 相关概念与技术 4](#_Toc107291379)

[2.1 LightGBM框架 4](#_Toc107291381)

[2.2 CNN算法 6](#_Toc107291382)

[2.3 TextCNN算法 9](#_Toc107291383)

[3 系统设计与实现 11](#_Toc107291384)

[3.1 赛题理解 11](#_Toc107291386)

[3.2 数据探索 13](#_Toc107291387)

[3.3 特征工程与基线模型 22](#_Toc107291388)

[3.4 高阶数据探索 27](#_Toc107291389)

[3.5 特征工程进阶与方案优化 41](#_Toc107291390)

[3.6 优化技巧与解决方案升级 51](#_Toc107291391)

[4 系统测试与改进 62](#_Toc107291392)

[4.1 系统测试 62](#_Toc107291394)

[4.2 改进 63](#_Toc107291395)

[5 总结与展望 64](#_Toc107291396)

[5.1 总结 64](#_Toc107291398)

[5.2 进一步研究 65](#_Toc107291399)

[参考文献 66](#_Toc107291400)

**Contents**

[Abstract V](#_Toc107291371)

[Contents VIII](#_Toc107291373)

[1 Introduction 1](#_Toc107291375)

[1.1 Research Background and Significance 1](#_Toc107291376)

[1.2 Research Status at Home and Abroad 2](#_Toc107291377)

[1.3 The Organizational Structure of the Thesis 4](#_Toc107291378)

[2 Related Concepts and Technologies 4](#_Toc107291380)

[2.1 LightGBM Frame 4](#_Toc107291381)

[2.2 CNN Algorithm 6](#_Toc107291382)

[2.3 TextCNN Algorithm 9](#_Toc107291383)

[3 System Design and Implementation 11](#_Toc107291385)

[3.1 Understand the Competition 11](#_Toc107291386)

[3.2 Data Explore 13](#_Toc107291387)

[3.3 Feature Engineering and Baseline Model 22](#_Toc107291388)

[3.4 Advanced Data Explore 27](#_Toc107291389)

[3.5 Advanced Feature Engineering and Program Optimization 41](#_Toc107291390)

[3.6 Optimization Tips and Solution Upgrades 51](#_Toc107291391)

[4 System Test and Improve 62](#_Toc107291393)

[4.1 System Test 62](#_Toc107291394)

[4.2 Improve 63](#_Toc107291395)

[5 Conclusions and Prospects 64](#_Toc107291397)

[5.1 Conclusions 64](#_Toc107291398)

[5.2 Further Research 65](#_Toc107291399)

[References 66](#_Toc107291400)

# 绪论

# 1 Introduction

## 研究背景及意义(Research Background and Significance)

互联网技术的发展给人们带来了好处,而风险也伴随而来。高速发展的互联网中信息安全问题不断出现，基础和新型网络产品带来的风险日益增加。根据McAfee2021年的报告[1]显示,恶意软件的数量年年上升，且上升的速度十分惊人。如图1-1所示，2021年第一季度的恶意软件数量统计达到87.6兆，较2020年第四季度增长将近3兆。



图 1-1 2021年第一季度恶意软件数量

Figure 1-1 Number of malware in Q1 2021

所谓恶意软件是指在计算机系统上执行恶意任务的病毒、蠕虫和木马等程序，通过破坏软件进程来实施控制。恶意软件出现的最初因素主要是兴趣的驱动，例如显示出安全缺陷或者是自己的技术能力，然而现在它是以试图破坏计算机系统和其他电子设备完整性为目的。因此，如何保护计算机系统和数据不因偶然或恶意的原因而遭到破坏、更改或泄露，如何降低计病毒感染算机的机会等信息安全技术，是当今计算机网络领域的研究重点。

目前主流的反病毒方法是使用之前生成的特征库，进行基于特征的反病毒扫描[2]。这种基于特征码的恶意软件检测是在上个世纪中期，基于特征码的检测被提出[3]。它的检测效果很大一部分决定于所具有的特征库，特征库越大则检测的准确率越高。特征码是一段短的字节字符串，对于每个己知的恶意软件类型都是唯一的，所对恶意软件及其后代有很高的检测率[4]。但是该技术存在的最大两个问题是：首先这些特征码都来自于人工的分析，这就给出错留下了空间[5]。而且根据迈克菲的报告2020年最后一个季度到2021年第一季度的恶意软件每日平均增长速率是500000个。如果使用人工来分析如此大量的新型恶意软件，无疑需要极大的时间代价[6]。而且分析得到新的特征码后，需要更新数据库同时需要部署到所有基于该数据库的客户端[5]。其次，特征码的检测没有办法检测多态、变异和未知的恶意软件，能检测的只是特征己经存在于数据库中的恶意软件，这是基于特征码的恶意软件检测目前最大的不足[7-9]。

目前基于行为的恶意软件检测主要分为静态分析检测和动态分析检测两种方式[2, 10-12]。其中，静态分析是通过对代码的分析获得软件的行为信息，但它依然无法良好应对精良的代码混淆技术。而动态的分析是通过实际运行软件，捕捉行为进行分析，它可以有效地应对代码混淆技术，更真实地反映软件意图。

本文认为无论经过怎样的掩饰，行为始终会暴露恶意软件的意图。而一个恶意软件实现自己的意图，需要通过调用现有的系统API，否则就需要自己再编写大量的代码，不只增加了代码复杂度，同时还会增加被检测的几率[13]。且Shankarapani等的研究表明Windows API调用可反映一段代码的特殊行为，可帮助检测恶意软件[14]。所以Windows API调用是值得分析的软件行为之一。因此，本文设计基于动态检测的思想，分析Windows操作系统中软件的Windows API调用行为，实现恶意软件检测。通过收集大量的恶意软件和非恶意软件样本，然后监测它们的Windows API调用，结合文本分类和数据挖掘的技术进行分析。以弥补基于特征码和静态分析的恶意软件检测的不足。

总之，本文旨在通过对软件的Windows API调用的分析，能够探究出一个有效的检测恶意软件方法，提高目前的恶意软件检测率。

## 国内外研究现状(Research Status at Home and Abroad)

### 静态恶意软件检测(Static Malware Detection)

静态检测其实就是对软件的代码的分析。它并不会实际运行恶意软件，其主要通过分析恶意软件的字节码，指令等。如果可获得程序的源代码，静态分析的效果会较好，然而这对很多的恶意软件来说是不可行的。而且即使在知道恶意软件的源代码的情况下，代码混淆技术所带来的代码执行的不确定性，也给分析带来了挑战。而且部分恶意软件的代码调用或指令执行只有在动态的情况下才会体现[5]。静态的恶意软件检测方法并不实际运行软件，因此难应对加壳、变形及多态等技术。同时相关的恶意行为并没有真实展现，通过静态检测，不一定能确保软件行为的安全性[15]。

最常见的就是基于PE文件的分析。2004年A.H.Sung等提出了应对代码混淆技术的恶意软件检测，使用PE中的Windows API作为特征进行检测，改善了原有恶意软件的检测方法[16]。他们使用了导入地址表来发现Windows API调用。K.Rozinov也使用了类似的方法进行研究。还有比较经典的n-gram方法，己经在基于文本的恶意软件检测中得到应用，但是并没有有效的检测出所有类型的恶意代码[12]。

国内也有很多关于PE文件恶意软件检测的研究，白等人结合数据挖掘的方法，很好地区分了恶意软件及非恶意软件，取得了99.1%的检测率[17]。付文等人考虑到了恶意的API调用，对病毒变体具有较好的检测效果[18]。白等使用了API调用图来提取API调用图对恶意软件进行检测也是一种重要的恶意软件检测方法[19]。

静态检测的方法对基于特征码的检测方法有一定的改进，对变体病毒也有较好的检测效果。然而静态的检测方法并不会真正的运行恶意软件，这样恶意软件就有了可乘之机。

### 动态恶意软件检测(Dynamic Malware Detection)

动态的恶意软件检测方法，通过实际运行恶意软件，在运行的过程中捕捉到其行为，以其行为作为分析的资料。因此动态的检测方法，需要实际的执行软件，从软件执行产生的行为来获得分析的元素。分析一个程序在被执行的过程中的动作称之为动态分析[5]。无论一个恶意软件的制作者使用怎样的技术来混淆和掩藏恶意程序的代码，恶意代码的行为都是无法掩饰的。对于一个恶意软件来说，它的变体的功能基本上都是没有改变的[20]。行为可更加真实的反应一个程序的意图。而在Windows平台之下，很多的行为都是通过调用系统带的API来完成的。使用API作动态的分析因素成为了信息安全领域的一个研究热点。

国内韩等人使用APIOVERRIDE工具对软件的Windows API调用进行了监测，使用了450个非恶意软件和950个恶意软件样本进行分析。采用信息增益作为特征提取的方法，提取了APL参数和API参数组合H组特征，使用eka分类器进行分类，取得了90%以上的分类准确率[21]。王等人分析程序PE文件调用的API，在特征选取时对信息增益进行了改进，将每个日志文件中API出现的频率加考虑。提取特征之后使用多种方法进行分类，其中随机森林分类方法使得检测率达到了96%上[22]。白等人使用Native API的调用频率，结合数据挖掘得到了96%上的检测率[15]。

国外动态检测到的方式较多，最开始的使用API调用的频率进行特征的选择，例如文献23也达到了90%上的准确率[23]。后续的研究对算法进行了改进，效果也得到了提高。Dolly等对使用Fisher得分进行特征选取，使用SVM分类取得了最佳的准确率，高达98%[24]。G.Ganesh Sundarkumar等使用LDA结合文档频率对特征进行提取，使用两个数据集进行实验，只选用了几个特征进行分类，其准确率相对于前面两种方法有所降低，不过依然保持在80%以上[25]。

## 论文组织结构(The Organizational Structure of the Thesis)

本文的主要研究目的在于：通过对恶意软件行为的分析，实现恶意软件的检测。

所以本文的主要研究内容如下：

(1)通过国内外恶意软件分享网站及论坛，收集恶意软件样本。

(2)在虚拟机中，运行收集到的恶意软件样本，收集恶意软件API调用日志。日志文件作为分析对象，将恶意软件动态行为分析转换为对文本的分析。

注：实际上我通过“阿里云安全恶意程序检测”竞赛获取了恶意软件API调用数据集。

(3)对参数取值的影响进行探究。使用文本分析的方法，从收集到的日志文本文件中提取特征，进行数据探索及数据预处理。

(4)使用三种不同的基线模型分别预测并计算logloss。对三组基线模型选择不同数量的特征进行实验。同时选用多种分类算法进行实验，取得了更好的恶意软件检测效果。

(5)其他如下方面的探究：把Windows API调用频率、Windows API调用之间的关联关系等恶意行为检测和实验结果分析。

# 相关概念与技术

# 2 Related Concepts and Technologies

## LightGBM框架(LightGBM Frame)

### 简介

GBDT (Gradient Boosting Decision Tree) 是机器学习中一个长盛不衰的模型，其主要思想是利用弱分类器（决策树）迭代训练以得到最优模型，该模型具有训练效果好、不易过拟合等优点。GBDT不仅在工业界应用广泛，通常被用于多分类、点击率预测、搜索排序等任务；在各种数据挖掘竞赛中也是致命武器，据统计Kaggle上的比赛有一半以上的冠军方案都是基于GBDT。而LightGBM（Light Gradient Boosting Machine）是一个实现GBDT算法的框架，支持高效率的并行训练，并且具有更快的训练速度、更低的内存消耗、更好的准确率、支持分布式可以快速处理海量数据等优点。

### 优化

为了能够在不损害准确率的条件下加快GBDT模型的训练速度，lightGBM在传统的GBDT算法上进行了如下优化：

* 基于Histogram的决策树算法。
* 单边梯度采样 Gradient-based One-Side Sampling(GOSS)：使用GOSS可以减少大量只具有小梯度的数据实例，这样在计算信息增益的时候只利用剩下的具有高梯度的数据就可以了，相比XGBoost遍历所有特征值节省了不少时间和空间上的开销。
* 互斥特征捆绑 Exclusive Feature Bundling(EFB)：使用EFB可以将许多互斥的特征绑定为一个特征，这样达到了降维的目的。
* 带深度限制的Leaf-wise的叶子生长策略：大多数GBDT工具使用低效的按层生长 (level-wise) 的决策树生长策略，因为它不加区分的对待同一层的叶子，带来了很多没必要的开销。实际上很多叶子的分裂增益较低，没必要进行搜索和分裂。LightGBM使用了带有深度限制的按叶子生长 (leaf-wise) 算法。
* 直接支持类别特征(Categorical Feature)
* 支持高效并行
* Cache命中率优化

这里不再对这些优化进行具体说明。

### 工程优化

#### 直接支持类别特征

类别特征的使用在实践中是很常见的。且为了解决one-hot编码处理类别特征的不足，LightGBM优化了对类别特征的支持，可以直接输入类别特征，不需要额外的0/1展开。LightGBM采用many-vs-many的切分方式将类别特征分为两个子集，实现类别特征的最优切分。假设某维特征有个类别，则有种可能，时间复杂度为，LightGBM基于Fisher的《On Grouping For Maximum Homogeneity》论文实现了的时间复杂度。

算法流程如下图所示，在枚举分割点之前，先把直方图按照每个类别对应的label均值进行排序；然后按照排序的结果依次枚举最优分割点。从下图可以看到，为类别的均值。当然，这个方法很容易过拟合，所以LightGBM里面还增加了很多对于这个方法的约束和正则化。

在Expo数据集上的实验结果表明，相比0/1展开的方法，使用LightGBM支持的类别特征可以使训练速度加速8倍，并且精度一致。更重要的是，LightGBM是第一个直接支持类别特征的GBDT工具。

#### 支持高效并行

高效并行具体包含：

1. 特征并行

特征并行的主要思想是不同机器在不同的特征集合上分别寻找最优的分割点，然后在机器间同步最优的分割点。XGBoost使用的就是这种特征并行方法。

2. 数据并行

LightGBM在数据并行中使用分散规约 (Reduce scatter) 把直方图合并的任务分摊到不同的机器，降低通信和计算，并利用直方图做差，进一步减少了一半的通信量。

3. 投票并行

基于投票的数据并行则进一步优化数据并行中的通信代价，使通信代价变成常数级别。在数据量很大的时候，使用投票并行的方式只合并部分特征的直方图从而达到降低通信量的目的，可以得到非常好的加速效果。

#### Cache命中率优化

LightGBM 所使用直方图算法对Cache友好：

1. 所有的特征都采用相同的方式获得梯度（区别于XGBoost的不同特征通过不同的索引获得梯度），只需要对梯度进行排序并可实现连续访问，大大提高了缓存命中率；

2. 因为不需要存储行索引到叶子索引的数组，降低了存储消耗，而且也不存在Cache Miss的问题。

## CNN算法(CNN Algorithm)

### 简介

卷积神经网络（Convolutional Neural Network, CNN）是一种前馈神经网络，它的人工神经元可以响应一部分覆盖范围内的周围单元，对于大型图像处理有出色表现。

卷积神经网络由一个或多个卷积层和顶端的全连通层（对应经典的神经网络）组成，同时也包括关联权重和池化层（Pooling layer）。这一结构使得卷积神经网络能够利用输入数据的二维结构。与其他深度学习结构相比，卷积神经网络在图像和语音识别方面能够给出更好的结果。这一模型也可以使用反向传播算法进行训练。相比较其他深度、前馈神经网络，卷积神经网络需要考量的参数更少，使之成为一种颇具吸引力的深度学习结构。

### 基本原理

典型的 CNN 由3个部分构成：卷积层、池化层、全连接层。

卷积层负责提取图像中的局部特征；池化层用来大幅降低参数量级(降维)；全连接层类似传统神经网络的部分，用来输出想要的结果。

1. 卷积——提取特征

卷积层的运算过程如下图，用一个卷积核扫完整张图片：

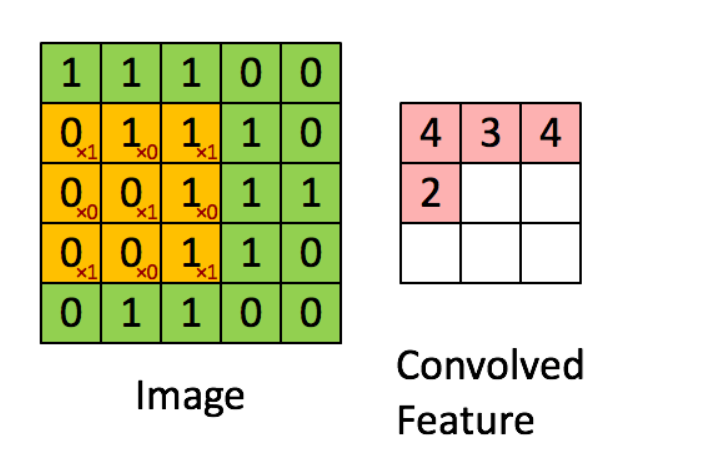


图 2-1 CNN算法

Figure 2-1 CNN Algorithm

这个过程我们可以理解为我们使用一个过滤器（卷积核）来过滤图像的各个小区域，从而得到这些小区域的特征值。

卷积层的通过卷积核的过滤提取出图片中局部的特征，和人类视觉的特征提取类似。

2. 池化层——数据降维，避免过拟合

池化层简单说就是下采样，他可以大大降低数据的维度。其过程如下：

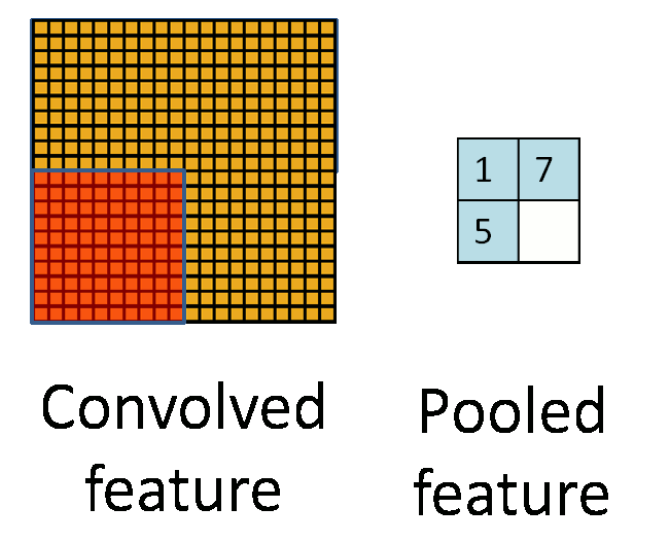


图 2-2 池化操作

Figure 2-2 Pooling

上图中，我们可以看到，原始图片是20×20的，我们对其进行下采样，采样窗口为10×10，最终将其下采样成为一个2×2大小的特征图。因为即使做完了卷积，图像仍然很大（因为卷积核比较小），所以为了降低数据维度，就对其进行池化。

池化层相比卷积层可以更有效的降低数据维度，这么做不但可以大大减少运算量，还可以有效的避免过拟合。

3. 全连接层——输出结果

经过卷积层和池化层降维过的数据，全连接层才能”跑得动”，不然数据量太大，计算成本高，效率低下。

一个典型的5层CNN——LeNet-5的结构：

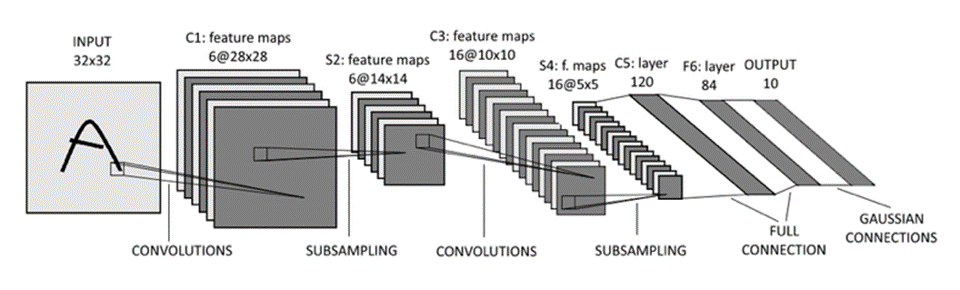


图 2-3 LeNet-5

Figure 2-3 LeNet-5

其包含卷积层——池化层——卷积层——池化层——卷积层——全连接层。

## TextCNN算法(TextCNN Algorithm)

### 简介

2014年，Yoon Kim针对CNN的输入层做了一些变形，提出了文本分类模型TextCNN[26]。

与传统图像的CNN网络相比，TextCNN在网络结构上没有任何变化，甚至更加简单了，TextCNN其实只有一层卷积，一层Max-Pooling，最后将输出外接Softmax进行N分类。

### 基本原理

#### 架构

TextCNN由n\*k的文本、卷积层、最大池化层和全连接层组成，如图：

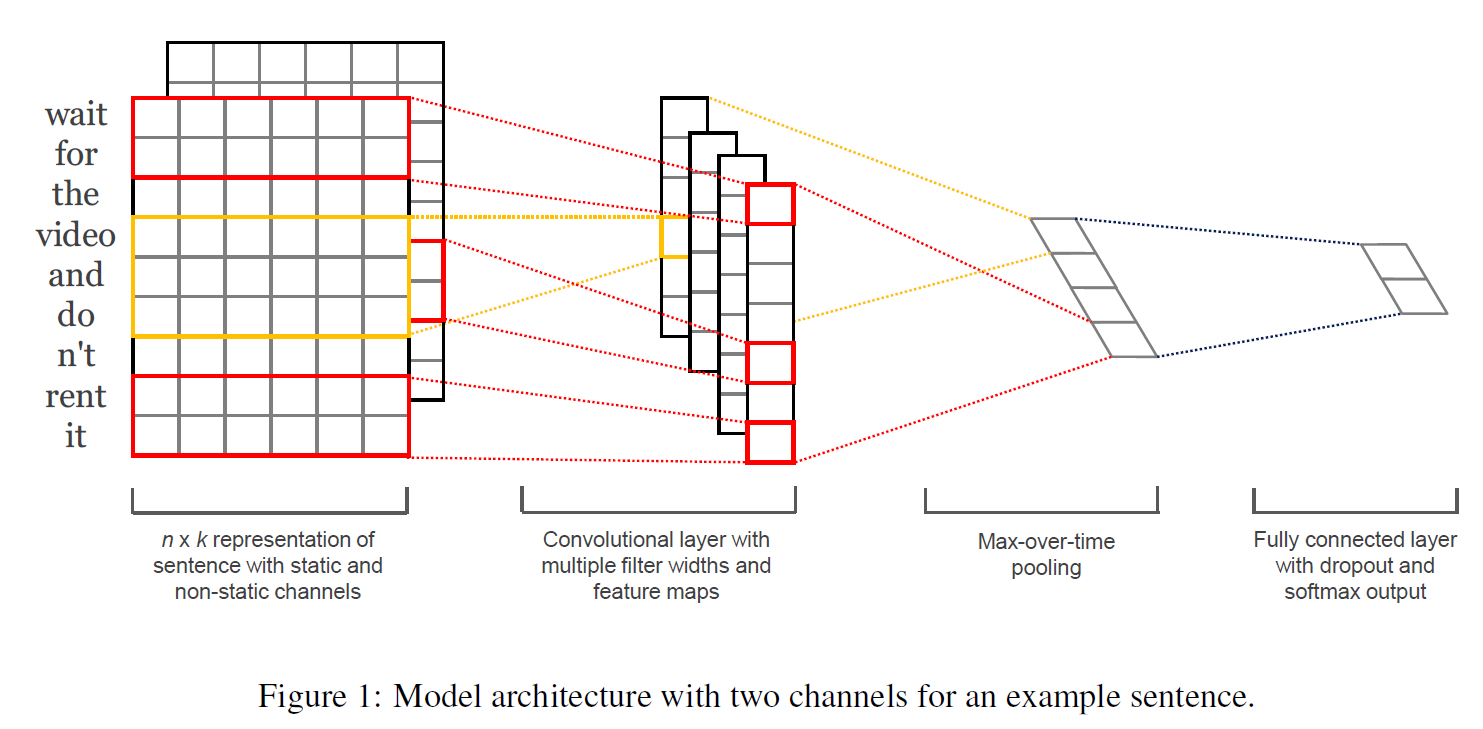


图 2-4 TextCNN架构

Figure 2-4 TextCNN Frame

#### 流程

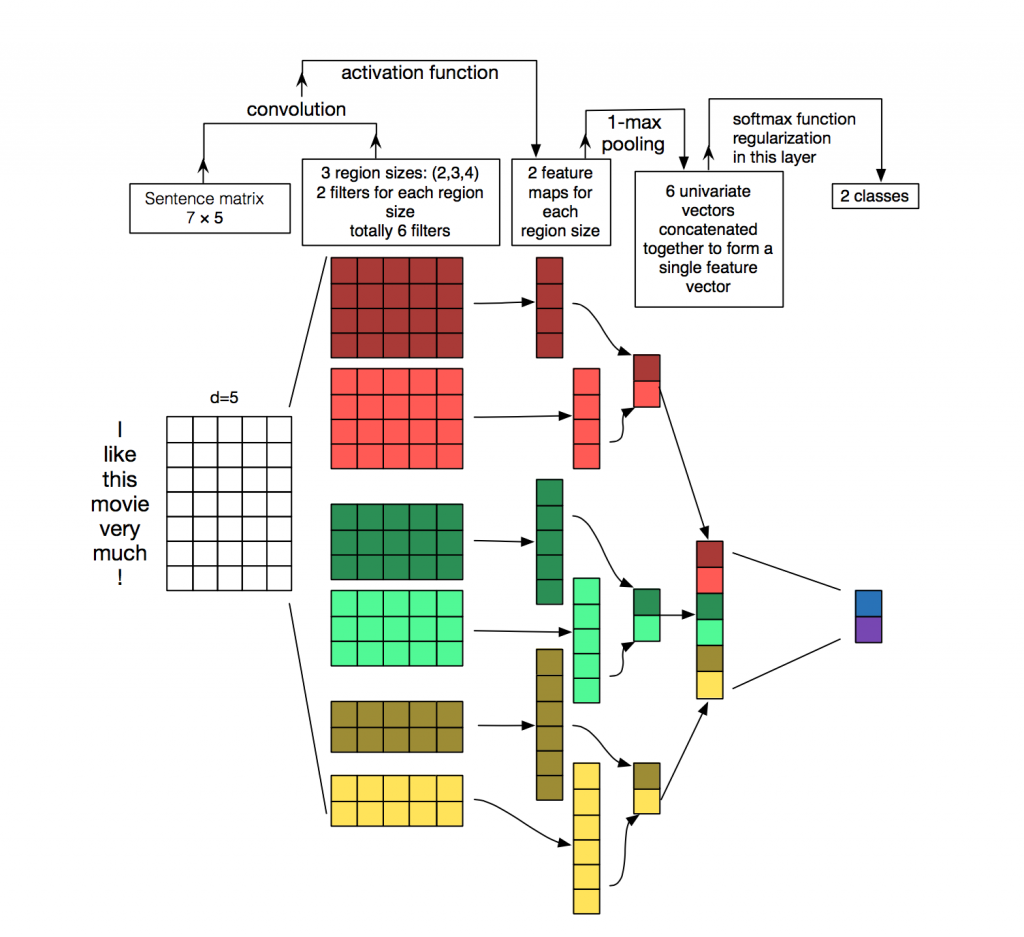


图 2-4 TextCNN流程

Figure 2-4 TextCNN Process

如图所示，其与CNN流程相似，要注意两点

1. 需要两个filter

2. 每个filter 的宽度与词向量等宽，只能进行一维滑动。

#### 总结

流程：先将文本分词做Embeeding得到词向量, 将词向量经过一层卷积,一层Max-Pooling, 最后将输出外接Softmax 来做N分类。

优势：模型简单, 训练速度快，效果不错。

缺点：模型可解释型不强，在调优模型的时候，很难根据训练的结果去针对性的调整具体的特征，因为在TextCNN中没有类似GBDT模型中特征重要度（feature importance）的概念, 所以很难去评估每个特征的重要度。

# 系统设计与实现

# 3 System Design and Implementation

## 赛题理解(Understand the Competition)

### 题意理解

恶意软件是一种被设计用来对目标计算机造成破坏或者占用目标计算机资源的软件，传统的恶意软件包括蠕虫、木马等，这些恶意软件严重侵犯用户合法权益，甚至将为用户及他人带来巨大的经济或其他形式的利益损失。近年来随着虚拟货币进入大众视野，挖矿类的恶意程序也开始大量涌现，黑客通过入侵恶意挖矿程序获取巨额收益。当前恶意软件的检测技术主要有特征码检测、行为检测和启发式检测等，配合使用机器学习可以在一定程度上提高泛化能力，提升恶意样本的识别率。

### 数据特征

表 3-1 Windows API数据特征

Table 3-1 Windows API Data Features

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | 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 |

本赛题的特征主要是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，最基本、最简单的特征思路是对所有API数据进行CountVertorizer特征。

### 标签理解

样本提供了在沙箱中的API调用序列，但在对病毒进行分类之后，我们需要了解标签中提到的不同病毒的属性。为了加深我们对题目的理解，下面简单地介绍一下，每种病毒的特性。

#### 勒索病毒

勒索软件，又称勒索病毒，是一种特殊的恶意软件，又被人归类为“阻断访问式攻击”（denial-of-access attack），其与其他病毒最大的不同在于手法以及中毒方式。其中一种勒索软件仅是单纯地将受害者的计算机锁起来，而另一种则系统性地加密受害者硬盘上的文件。所有的勒索软件都会要求受害者缴纳赎金以取回对计算机的控制权，或是取回受害者根本无从自行获取的解密密钥以便解密文件。勒索软件通常透过木马病毒的形式传播，将自身为掩盖为看似无害的文件，通常会通过假冒成普通的电子邮件等社会工程学方法欺骗受害者点击链接下载，但也有可能与许多其他蠕虫病毒一样利用软件的漏洞在联网的计算机间传播。

因为勒索病毒经常会有木马病毒及蠕虫病毒等的属性，所以模型中我们可能会将其混淆，需要重点寻找勒索病毒与二者的区别。

#### 挖矿程序

随着虚拟加密货币越来越热门，网络犯罪集团整积极开发、微调各种虚拟加密货币恶意程序。这些挖矿程序恶意消耗用户的计算机运算资源，消耗用户的CPU算力，但往往都隐藏在系统的各种进程中，并不容易被发现。

因此，挖矿病毒具有消耗资源和隐蔽的特点，也会和木马病毒一样进行传播。

#### DDoS木马

DDoS是Distributed Denial of Service（分布式拒绝服务）的缩写。DDoS是一种DoS（拒绝服务）攻击类型，其中多个受到威胁的系统（通常受特洛伊木马感染）被用于针对单个系统，从而导致DoS攻击。DDoS攻击的受害者包括最终目标系统和黑客在分布式攻击中恶意使用和控制的所有系统。

由此可知，DDoS会经常和木马病毒相关，而且和网络相关。

#### 蠕虫病毒

与计算机病毒相似，是一种能够自我复制的计算机程序。与计算机病毒不同的是，计算机蠕虫不需要附在别的程序内，可能不用使用者介入操作也能自我复制或执行。

计算机蠕虫未必会直接破坏被感染的系统，却几乎都对网络有害。计算机蠕虫可能会执行垃圾代码以发动分散式阻断服务攻击，令计算机的执行效率极大程度降低，从而影响计算机的正常使用，可能会损毁或修改目标计算机的档案，亦可能只是浪费带宽。

恶意的计算机蠕虫可根据其目的分成2类：一类是面对大规模计算机使用网络发动拒绝服务的计算机蠕虫，虽说会绑架计算机，但使用者可能还可以正常使用，只是会被占用一部分运算、连网能力。另一类是针对个人用户的以执行大量垃圾代码的计算机蠕虫。计算机蠕虫多不具有跨平台性，但是在其他平台下，可能会出现其平台特有的非跨平台性的平台版本。第一个被广泛注意的计算机蠕虫名为：“莫里斯蠕虫”，由罗伯特·泰潘·莫里斯编写，于1988年11月2日释出第一个版本。这个计算机蠕虫间接和直接地造成了近1亿美元的损失。这个计算机蠕虫释出之后，引起了各界对计算机蠕虫的广泛关注。

通过上面的分析，我们发现蠕虫病毒经常和DDoS相关，而且会使计算机资源消耗严重。

#### 感染型病毒

感染型将自身加入在其它的程序或动态库文件(DLL的一种)中，从而实现随被感染程序同步运行的功能，进而对感染电脑进行破坏和自身传播。感染型病毒由于其自身的特性，需要附加到其他宿主程序上进行运行，并且为了躲避杀毒软件的查杀，通常感染型病毒都会将自身分割、变形或加密后，再将自身的一部分或者全部附加到宿主程序上。一旦一个病毒文件执行，它很有可能就将系统中的绝大多数程序文件都加入病毒代码，进而传播给其它计算机。

由此可知，感染型病毒具有很强的破坏性，而且会自身传播，需要有宿主（和蠕虫病毒一样），同时经常会通过分割、变形、加密来保护自己不被发现。

#### 后门程序

软件后门则指绕过软件的安全性控制，而从比较隐秘的通道获取对程序或系统访问权的黑客方法。在软件开发时，设置后门可以方便修改和测试程序中的缺陷。

更多的后门程序是黑客制作的加壳程序，即在正常程序基础上加入后门，用来盗取其他用户的个人信息，甚至是远程控制对方的计算机而加壳制作，然后通过各种手段传播或者骗取目标用户执行该程序，以达到盗取密码等各种数据资料等目的。与病毒相似，木马程序有很强的隐秘性，随操作系统启动而启动。

#### 木马程序

木马程序（Trojan horse program）通常称为木马，恶意代码等，是指潜伏在电脑中，可受外部用户控制以窃取本机信息或者控制权的程序。木马指的是特洛伊木马，英文叫做“Trojan horse”，其名称取自希腊神话的特洛伊木马记。

木马程序带来很多危害，例如占用系统资源，降低电脑效能，危害本机信息安全（盗取QQ帐号、游戏帐号甚至银行帐号），将本机作为工具来攻击其他设备等

木马程序是比较流行的病毒文件，与一般的病毒不同，它不会自我繁殖，也不会刻意地去感染其他文件，它通过将自身伪装吸引用户下载执行，向施种木马者提供打开被种者电脑的门户，使施种者可以任意毁坏、窃取被种者的文件，甚至远程操控被种者的计算机。

通过上面的理解可知，木马与感染型病毒和蠕虫病毒略有区别，但是和后门程序的相关性较大。

### 解题思路

根据官方提供的每个文件对API的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的结果进行提交，并采用官方的logloss作为最终评分，属于典型的多分类问题。

## 数据探索(Data Explore)

下面对数据的基本特征进行了解。虽然通过题目的介绍我们已经知道了数据的格式，但是这对于特征提取和建模来说，还远远不够。

### 训练集数据探索

#### 数据特征类型

本赛题的数据是csv文件，读取数据的代码如下：

# 导入相关库

import pandas as pd

import numpy as py

import seaborn as sns

import matplotlib.pyplot as plt

# 忽略警告信息

import warnings

warnings.filterwarnings('ignore')

# 在jupyter中显示图像

%matplotlib inline

# 读取数据

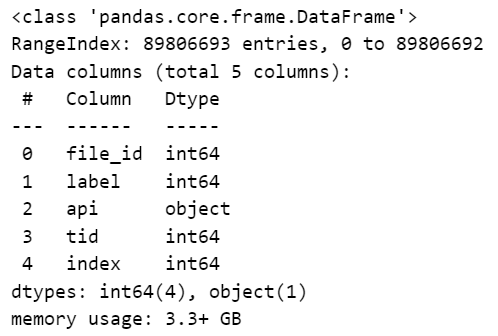
path = '../datasets/'

train = pd.read\_csv(path + 'security\_train.csv')

test = pd.read\_csv(path + 'security\_test.csv')

用DataFrame.info()函数查看训练集的大小、数据等信息。

train.info()

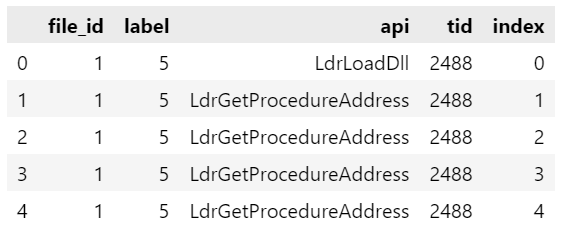


从运行结果可以看出：

* 数据中有4个int64类型的数据和1个object类型的数据（api）；
* 整个数据集的大小为3.3GB
* 数据一共有89 806 693条记录

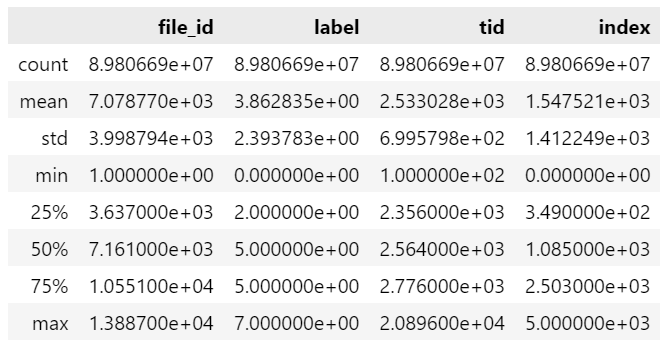
用DataFrame.head()函数查看训练集的头几行数据：

train.head()



用DataFrame.describe()函数查看训练集的统计信息：

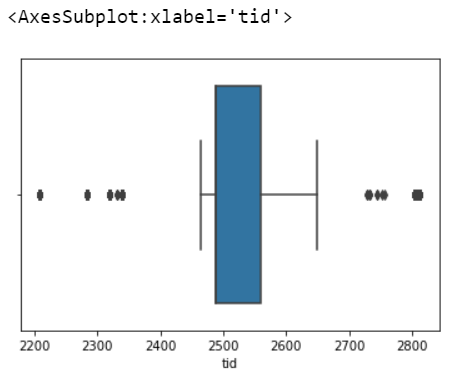
train.describe()



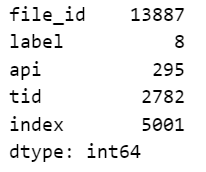
#### 数据分布

使用箱线图查看单个变量的分布情况。下面以训练集为例，取前10000条数据绘制“tid”变量的箱线图，代码和运行结果如下：

sns.boxplot(x=train.iloc[:10000]["tid"])



用nunique()函数查看训练集中变量取值的分布：

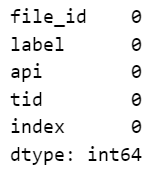


由运行结果可知：file\_id有13887个不同的值；有8种不同的label；有295个不同的API；有2782个不同的tid；有5001个不同的index。

#### 缺失值

查看训练集数据的缺失情况：

train.isnull().sum()

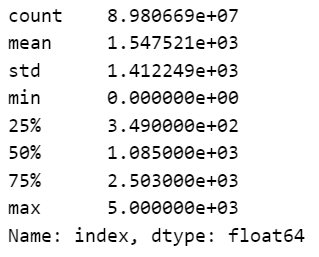


从运行结果看，数据不存在缺失的情况。

#### 异常值

分析训练集的“index”特征

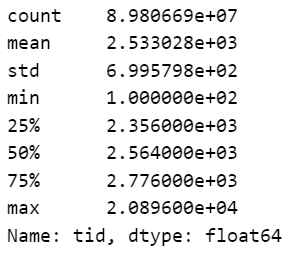
train['index'].describe()



“index”特征的最小值为0，最大值为5000，刚好是5001个值，看不出异常值。

分析训练集的“tid”特征：

train['tid'].describe()



“tid”特征的最小值为100，最大值为20896，因为这个字段表示的是线程，所以我们目前也没法判断是否有异常值。

#### 标签分布

“label”对应的含义如：

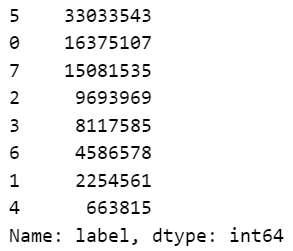
表 3-2 label标签与程序类型的映射

Table 3-2 Mapping of label tags to program types

|  |  |
| --- | --- |
| 值 | 分类 |
| 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 |

统计标签取值的分布情况：

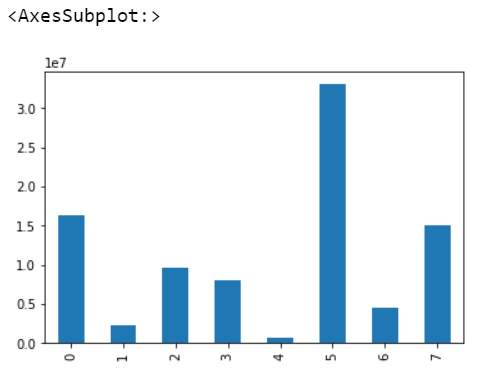
train['label'].value\_counts()



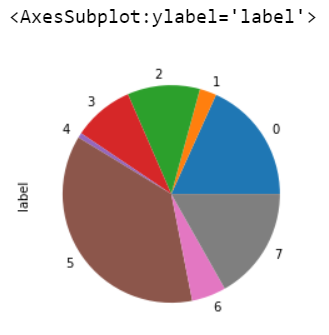
由标签分布可以发现：训练集中一共有16375107个正常文件（label=0）；2254561个勒索病毒（label=1）；9693969个挖矿程序（label=2）；8117585个DDoS木马（label=3）；663815个蠕虫病毒（label=4）；33033543个感染型病毒（label=5）；4586578后门程序（label=6）；15081535个木马程序（label=7）。

为了直观化，我们可以通过条形图看数据的大小，同时用饼图看数据的比例，代码和结果如下：

train['label'].value\_counts().sort\_index().plot(kind='bar')



train['label'].value\_counts().sort\_index().plot(kind='pie')

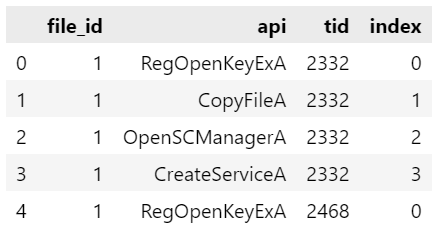


### 测试集数据探索

#### 数据信息

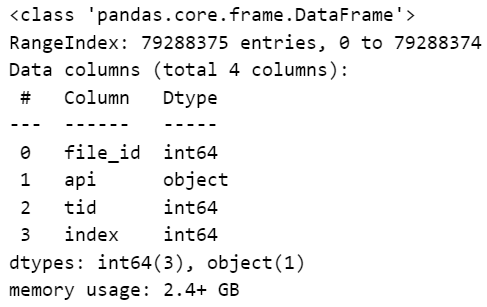
查看测试集的头几行数据：

test.head()



查看测试集的大小、数据类型等信息：

test.info()



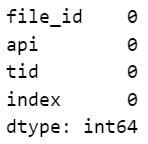
从运行结果可以看出：

* 数据中有3个int64类型的数据和1个object类型的数据（api）；
* 整个数据集的大小为2.4GB
* 数据一共有79 288 375条记录

#### 缺失值

查看测试集数据的缺失情况：

test.isnull().sum()

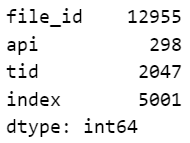


可知，数据不存在缺失的情况。

#### 数据分布

查看测试集中变量取值的分布：

test.nunique()

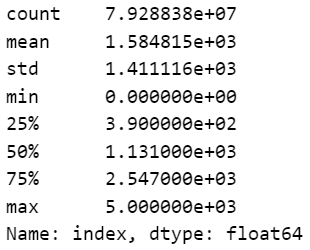


由运行结果可知：file\_id有12955个不同值；有298个不同的API；有2047个不同的tid；有5001个不同的index。

#### 异常值

查看测试集的“index”特征：

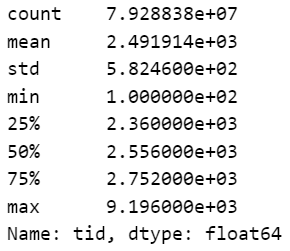
test['index'].describe()



由结果可知，“index”特征的最小值为0，最大值为5000，刚好是5001个值，看不出任何异常的情况。

查看“测试集”的“tid”特征：

test['tid'].describe()



由结果可知，“tid”特征的最小值为100，最大值为9196，因为这个字段表示的是线程，所以我们目前也没办法判断是否有异常值。

### 数据集联合分析

#### 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\_fileids = train['file\_id'].unique()

test\_fileids = test['file\_id'].unique()

len(set(train\_fileids) - set(test\_fileids))



运行结果表明，有932个训练文件是测试文件中没有的。

len(set(test\_fileids) - set(train\_fileids))



运行结果表明，测试文件中有的文件，在训练文件中都有。

我们发现训练数据集和测试数据集的file\_id存在交叉，也就是说file\_id在训练集和测试集不是完全无交集的，这个时候不能直接合并，需要进行其他的处理来区分训练集和测试集。

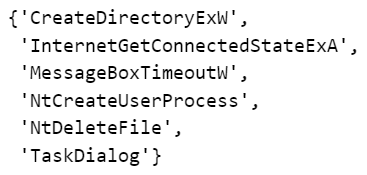
#### API分析

对比分析“API”变量在训练集和测试集中分布的重合情况：

train\_apis = train['api'].unique()

test\_apis = test['api'].unique()

set(test\_apis) - set(train\_apis)



运行结果表明，测试集中有6个API未出现在训练集中，分别是CreateDirectoryExW, InternetGetConnectedStateExA, MessageBoxTimeoutW, NtCreateUserProcess, NtDeleteFile, TaskDialog。

set(train\_apis) - set(test\_apis)



运行结果表明，训练集中有3个API为出现在测试集中，分别是EncryptMessage, RtlCompressBuffer, WSASendTo。

## 特征工程与基线模型(Feature Engineering and Baseline Model)

通过简单的数据探索，我们对赛题数据有了大概了解，那么接下来就可以通过简单的特征工程，快速构建基线（Baseline）模型了。一方面，快速构建基线模型可以让我们对基础数据在评价指标上的基线值有所了解；另一方面，一个良好的基线模型结构是进一步迭代提升成绩的基石。

### 数据读取

导入库，读取数据

import numpy as np

import pandas as pd

import matplotlib.pyplot as plt

import seaborn as sns

import lightgbm as lgb

import warnings

warnings.filterwarnings("ignore")

%matplotlib inline

path = '../datasets/'

train = pd.read\_csv(path + 'security\_train.csv')

test = pd.read\_csv(path + 'security\_test.csv')

### 特征工程

1. 利用count()函数和nunique()函数生成特征：反映样本调用api、tid、index的频率信息。

def simple\_sts\_features(df):

    simple\_fea = pd.DataFrame()

    simple\_fea['file\_id'] = df['file\_id'].unique()

    simple\_fea = simple\_fea.sort\_values('file\_id')

    df\_grp = df.groupby('file\_id')

    simple\_fea['file\_id\_api\_count'] = df\_grp['api'].count().values

    simple\_fea['file\_id\_api\_nunique'] = df\_grp['api'].nunique().values

    simple\_fea['file\_id\_tid\_count'] = df\_grp['tid'].count().values

    simple\_fea['file\_id\_tid\_nunique'] = df\_grp['tid'].nunique().values

    simple\_fea['file\_id\_index\_count'] = df\_grp['index'].count().values

    simple\_fea['file\_id\_index\_nunique'] = df\_grp['index'].nunique().values

    return simple\_fea

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

def simple\_numberical\_sts\_features(df):

    simple\_numberical\_fea = pd.DataFrame()

    simple\_numberical\_fea['file\_id'] = df['file\_id'].unique()

    simple\_numberical\_fea = simple\_numberical\_fea.sort\_values('file\_id')

    df\_grp = df.groupby('file\_id')

    simple\_numberical\_fea['file\_id\_tid\_mean'] = df\_grp['tid'].mean().values

    simple\_numberical\_fea['file\_id\_tid\_min'] = df\_grp['tid'].min().values

    simple\_numberical\_fea['file\_id\_tid\_std'] = df\_grp['tid'].std().values

    simple\_numberical\_fea['file\_id\_tid\_max'] = df\_grp['tid'].max().values

    simple\_numberical\_fea['file\_id\_index\_mean'] = df\_grp['index'].mean().values

    simple\_numberical\_fea['file\_id\_index\_min'] = df\_grp['index'].min().values

    simple\_numberical\_fea['file\_id\_index\_std'] = df\_grp['index'].std().values

    simple\_numberical\_fea['file\_id\_index\_max'] = df\_grp['index'].max().values

    return simple\_numberical\_fea

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。

反映样本调用api、tid、index的频率信息的统计特征：

%%time

simple\_train\_fea1 = simple\_sts\_features(train)



%%time

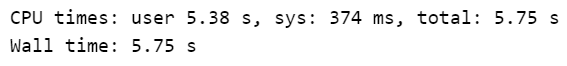
simple\_test\_fea1 = simple\_sts\_features(test)



反映tid、index等数值特征的统计特征：

%%time

simple\_train\_fea2 = simple\_numberical\_sts\_features(train)



%%time

simple\_test\_fea2 = simple\_numberical\_sts\_features(test)



### 基线构建

获取标签：

train\_label = train[['file\_id', 'label']].drop\_duplicates(subset=['file\_id', 'label'], keep='first')

test\_label = test[['file\_id']].drop\_duplicates(subset=['file\_id'], keep='first')

训练集和测试集的构建：

train\_data = train\_label.merge(simple\_train\_fea1, on='file\_id', how='left')

train\_data = train\_data.merge(simple\_train\_fea2, on='file\_id', how='left')

test\_submit = test\_label.merge(simple\_test\_fea1, on='file\_id', how='left')

test\_submit = test\_submit.merge(simple\_test\_fea2, on='file\_id', how='left')

自定义logloss指标：

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]):

        sum\_ = 0

        for j in range(preds\_prob\_.shape[0]):

            pred = preds\_prob\_[j, i]

            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

线下验证。因为数据与时间的相关性并不是非常大，所以此处我们用传统的5折交叉验证来构建线下验证集。

train\_features = [col for col in train\_data.columns if col not in ['label', 'file\_id']]

train\_label = 'label'

使用5折交叉验证，采用LightGBM模型，代码和运行结果如下：

%%time

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

}

folds = KFold(n\_splits=5, shuffle=True, random\_state=15)

oof = np.zeros(len(train))

predict\_res = 0

models = []

for fold\_, (trn\_idx, val\_idx) in enumerate(folds.split(train\_data)):

    print(f"fold n°{fold\_}")

    trn\_data = lgb.Dataset(train\_data.iloc[trn\_idx][train\_features], label=train\_data.iloc[trn\_idx][train\_label].values)

    val\_data = lgb.Dataset(train\_data.iloc[val\_idx][train\_features], label=train\_data.iloc[val\_idx][train\_label].values)

    clf = lgb.train(params,

                    trn\_data,

                    num\_boost\_round=2000,

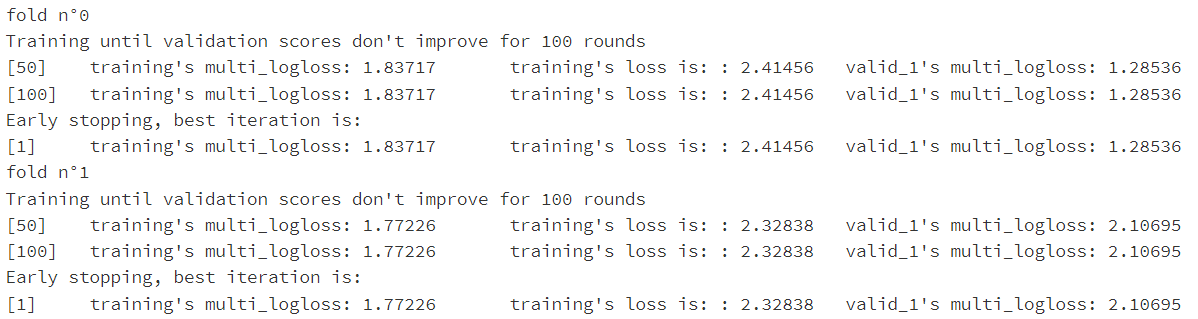
                    valid\_sets=[trn\_data, val\_data],

                    verbose\_eval=50,

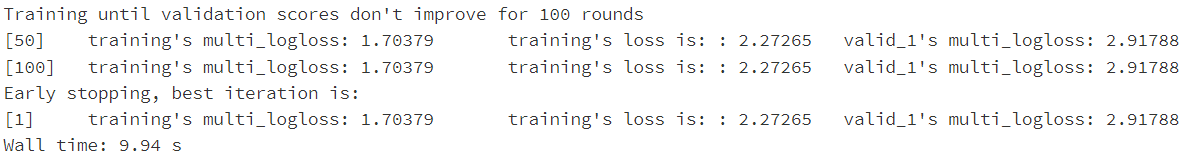
                    early\_stopping\_rounds=100,

                    feval=lgb\_logloss)

    models.append(clf)



…



### 特征重要性分析

通过特征重要性分析，可以看到在当前指标显示的成绩下，影响因子最高的特征因素，从而更好地理解题目，同时在此基础上进行特征工程的延伸。相应的代码如下：

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'])

### 模型测试

至此，通过前期的数据探索分析和基础的特征工程，我们可以用上面训练的5个LightGBM分别对测试集进行预测，然后将所有测试结果的均值作为最终结果。其预测结果并不能100%超过单折全量数据的LightGBM，但是大多数还是不错的。

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']].to\_csv('baseline.csv', index=None)

线下成绩：1.087292

## 高阶数据探索(Advanced Data Explore)

我们可以进一步分析变量之间的关系，发现更多有用信息，进一步探索数据之间的规律。

### 数据读取

导入库

import numpy as np

import pandas as pd

import matplotlib.pyplot as plt

import seaborn as sns

import lightgbm as lgb

import warnings

warnings.filterwarnings("ignore")

%matplotlib inline

读取数据集

path = '../datasets/'

train = pd.read\_csv(path + 'security\_train.csv')

test = pd.read\_csv(path + 'security\_test.csv')

### 多变量交叉探索

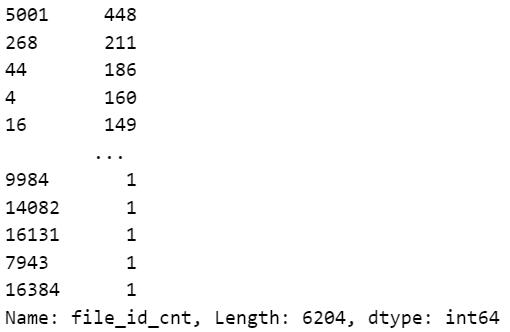
1. 通过统计特征file\_id\_cnt，分析file\_id变量和api变量之间的关系。

train\_analysis = train[['file\_id', 'label']].drop\_duplicates(subset=['file\_id', 'label'], keep='last')

dic\_ = train['file\_id'].value\_counts().to\_dict()

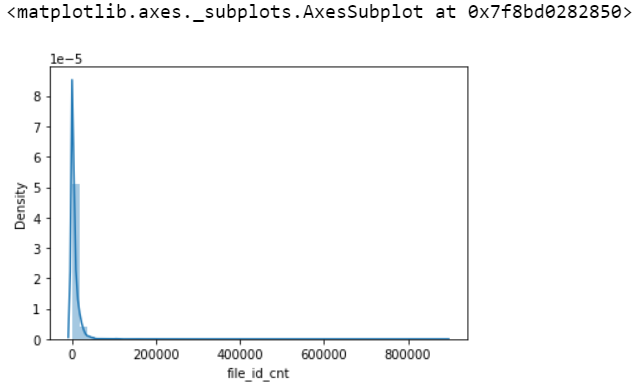
train\_analysis['file\_id\_cnt'] = train\_analysis['file\_id'].map(dic\_).values

train\_analysis['file\_id\_cnt'].value\_counts()



可以看到，文件调用API次数出现最多的是5001次。

sns.distplot(train\_analysis['file\_id\_cnt'])



API调用次数的80%都集中在1000次以下。

为了便于分析file\_id\_cnt变量和label变量之间的关系，首先将数据按file\_id\_cnt变量（即API的调用次数）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

然后随机选取4个区间查看，代码及运行结果如下所示。

plt.figure(figsize=[16,20])

plt.subplot(321)

train\_analysis[train\_analysis['file\_id\_cnt\_cut'] == 0]['label'].value\_counts().sort\_index().plot(kind = 'bar')

plt.title('file\_id\_cnt\_cut = 0')

plt.xlabel('label')

plt.ylabel('label\_number')

plt.subplot(322)

train\_analysis[train\_analysis['file\_id\_cnt\_cut'] == 1]['label'].value\_counts().sort\_index().plot(kind = 'bar')

plt.title('file\_id\_cnt\_cut = 1')

plt.xlabel('label')

plt.ylabel('label\_number')

plt.subplot(323)

train\_analysis[train\_analysis['file\_id\_cnt\_cut'] == 14]['label'].value\_counts().sort\_index().plot(kind = 'bar')

plt.title('file\_id\_cnt\_cut = 14')

plt.xlabel('label')

plt.ylabel('label\_number')

plt.subplot(324)

train\_analysis[train\_analysis['file\_id\_cnt\_cut'] == 15]['label'].value\_counts().sort\_index().plot(kind = 'bar')

plt.title('file\_id\_cnt\_cut = 15')

plt.xlabel('label')

plt.ylabel('label\_number')

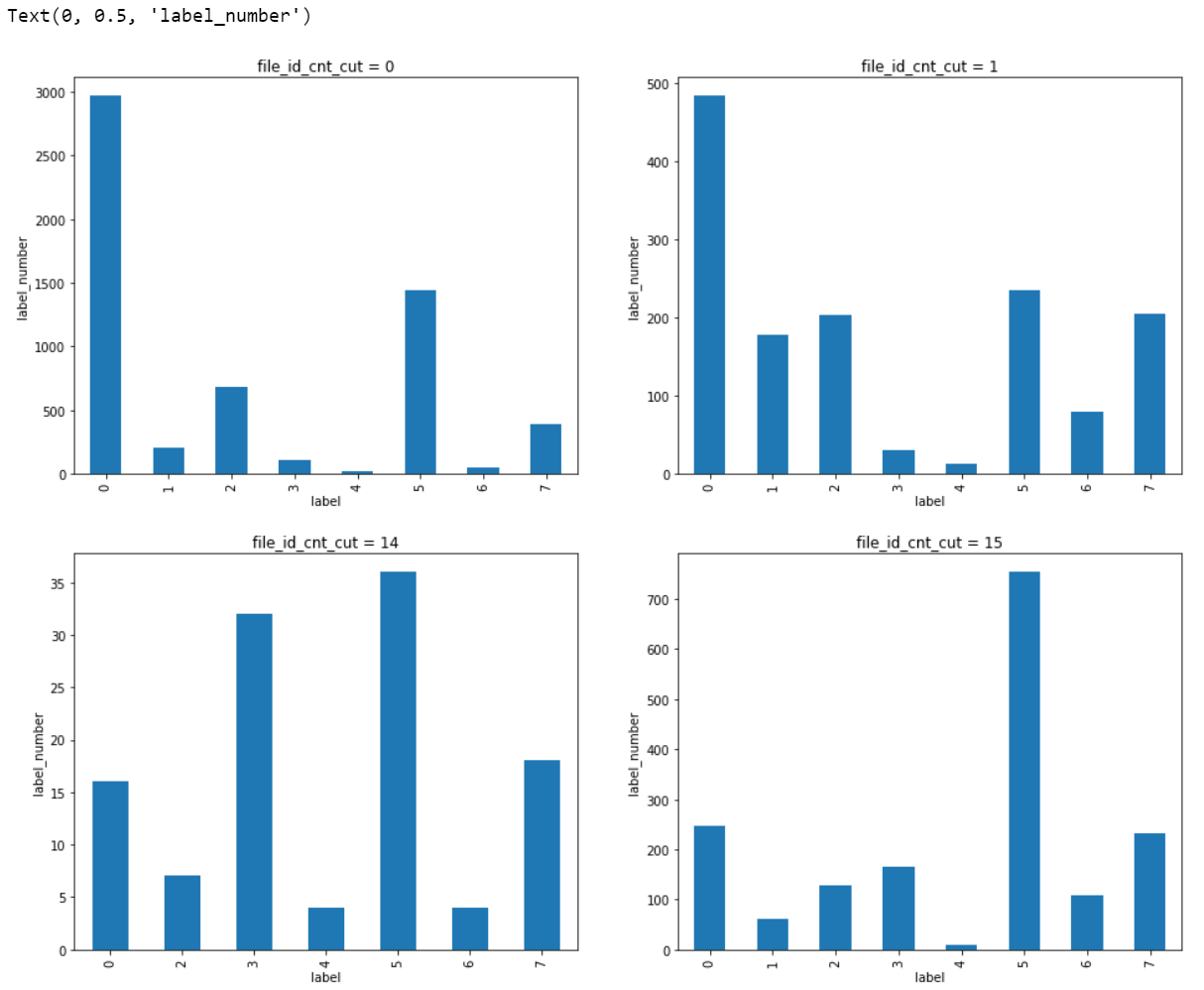
plt.subplot(313)

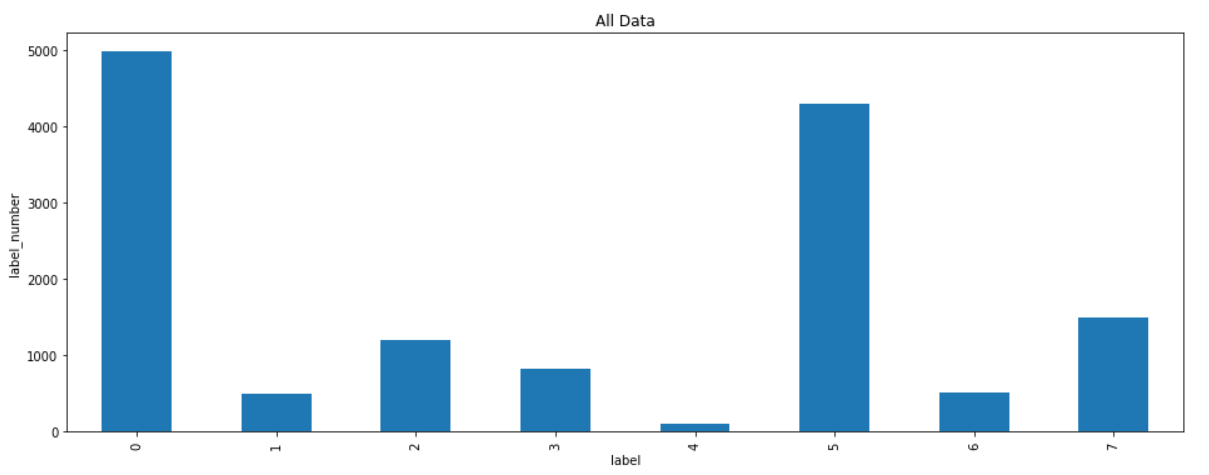
train\_analysis['label'].value\_counts().sort\_index().plot(kind = 'bar')

plt.title('All Data')

plt.xlabel('label')

plt.ylabel('label\_number')



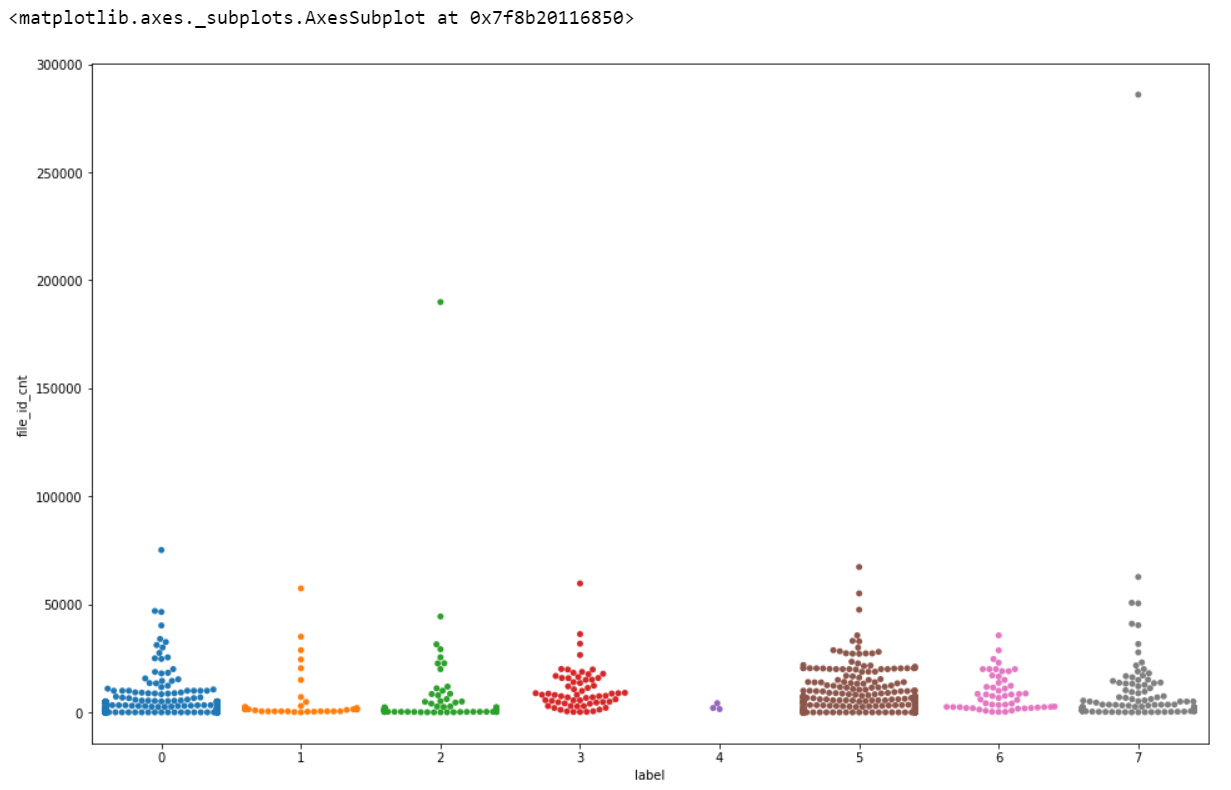


从图中可以看到：当API调用次数越多时，该API是第五类病毒（感染型病毒）的可能性就越大。

用分簇散点图查看label下file\_id\_cnt的分布，由于绘制分簇散点图比较耗时，因此我们采用1000个样本点。

plt.figure(figsize=[16, 10])

sns.swarmplot(x=train\_analysis.iloc[:1000]['label'], y=train\_analysis.iloc[:1000]['file\_id\_cnt'])



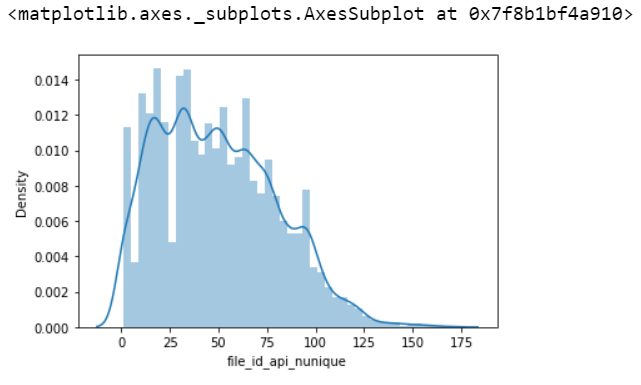
从图中可以得到以下结论：从频次上看，第5类病毒调用API的次数最多；从调用峰值上看，第2类和第7类病毒有时能调用150000次的API。

3. 首先通过文件调用API的类型数file\_id\_api\_nunique，分析变量file\_id和API的关系。

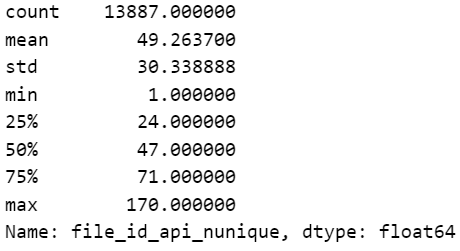
dic\_ = train.groupby('file\_id')['api'].nunique().to\_dict()

train\_analysis['file\_id\_api\_nunique'] = train\_analysis['file\_id'].map(dic\_).values

sns.distplot(train\_analysis['file\_id\_api\_nunique'])



train\_analysis['file\_id\_api\_nunique'].describe()



文件调用API的类别数绝大部分都在100以内，最少的是1个，最多的是170个。

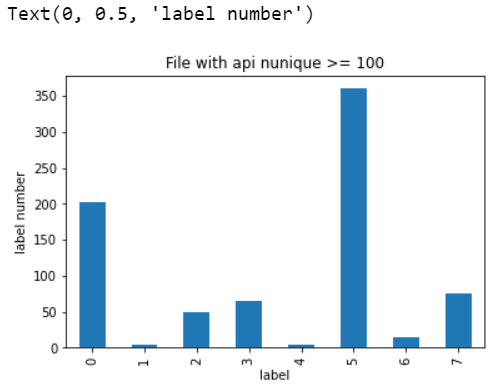
然后分析file\_id\_api\_nunique和标签label的关系。

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')

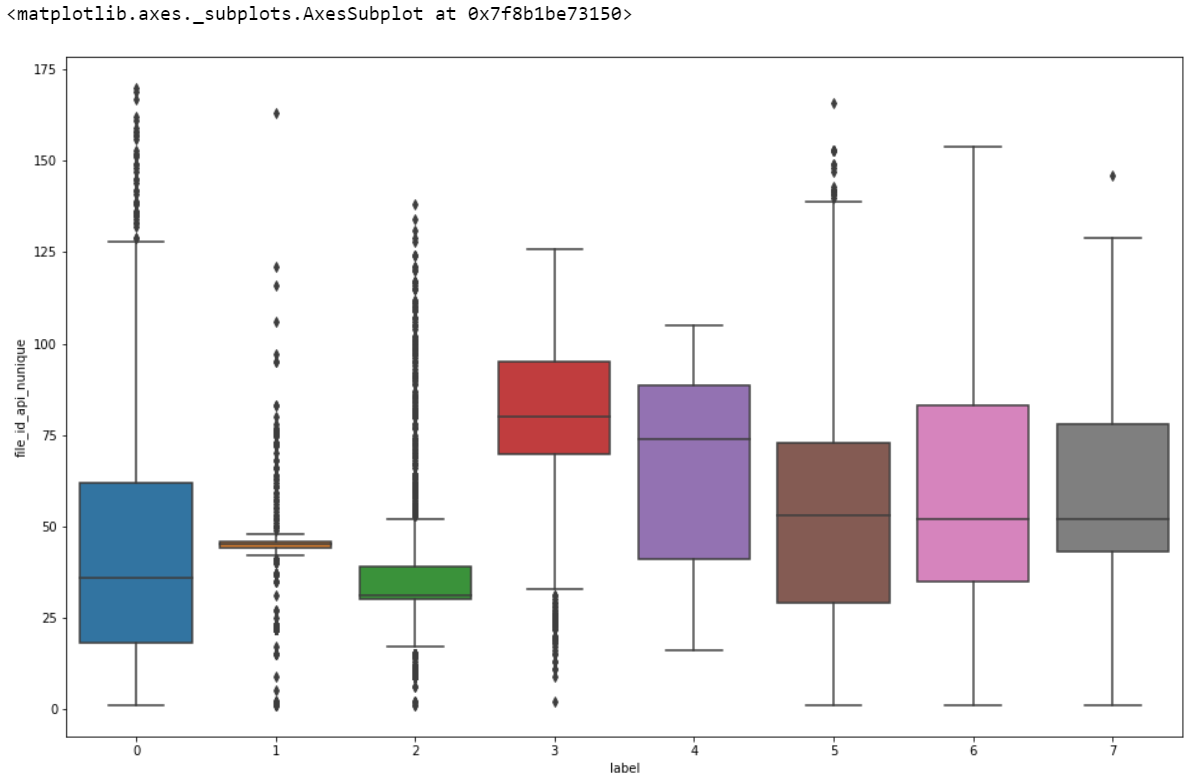
plt.ylabel('label number')



从图中可以发现，第5类病毒调用不同API的次数是最多的。在上面的分析中，我们也发现第5类病毒调用API的次数最多，调用不同API的次数多也是可以理解的。

plt.figure(figsize=[16, 10])

sns.boxplot(x=train\_analysis['label'], y=train\_analysis['file\_id\_api\_nunique'])



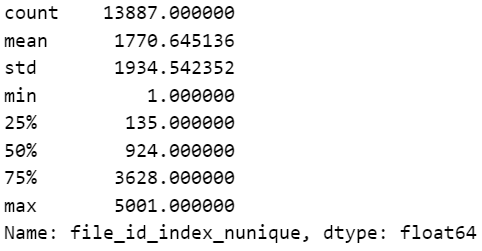
从图中可以得到以下结论：第3类病毒调用不同的API的次数相对较多，第2类病毒调用不同API的次数最少，第4,6,7类病毒的离群点较少，第1类病毒的离群点最多，第3类病毒的离群点主要在下方，第0类和第5类的离群点则集中在上方。

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。我们发现调用API顺序编号的两个边缘（0和5001）的样本数是最多的，因此可以单独看一下这两个点的label分布。

dic\_ = train.groupby('file\_id')['index'].nunique().to\_dict()

train\_analysis['file\_id\_index\_nunique'] = train\_analysis['file\_id'].map(dic\_).values

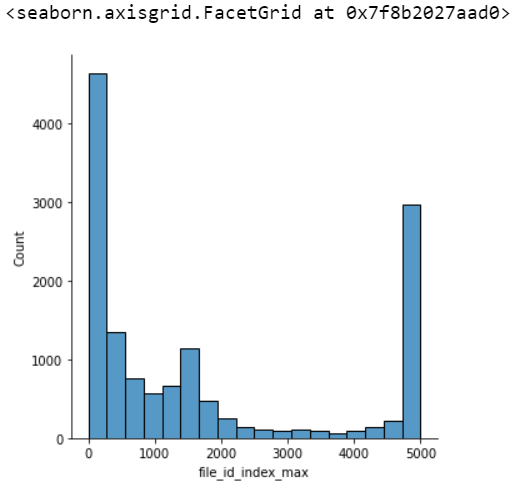
train\_analysis['file\_id\_index\_nunique'].describe()



dic\_ = train.groupby('file\_id')['index'].max().to\_dict()

train\_analysis['file\_id\_index\_max'] = train\_analysis['file\_id'].map(dic\_).values

sns.displot(train\_analysis['file\_id\_index\_max'])



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

然后分析file\_id\_index\_nunique和file\_id\_index\_max和标签label变量的关系。

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')

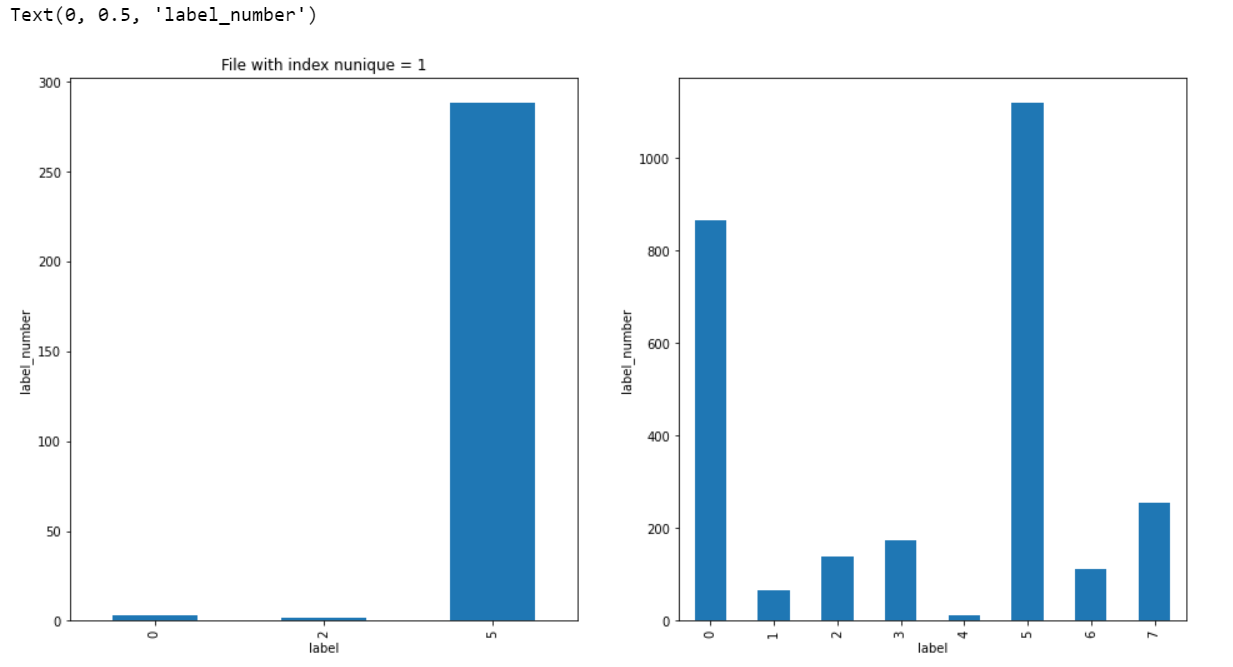
plt.ylabel('label\_number')

plt.subplot(122)

train\_analysis.loc[train\_analysis.file\_id\_index\_nunique == 5001]['label'].value\_counts().sort\_index().plot(kind='bar')

plt.xlabel('label')

plt.ylabel('label\_number')

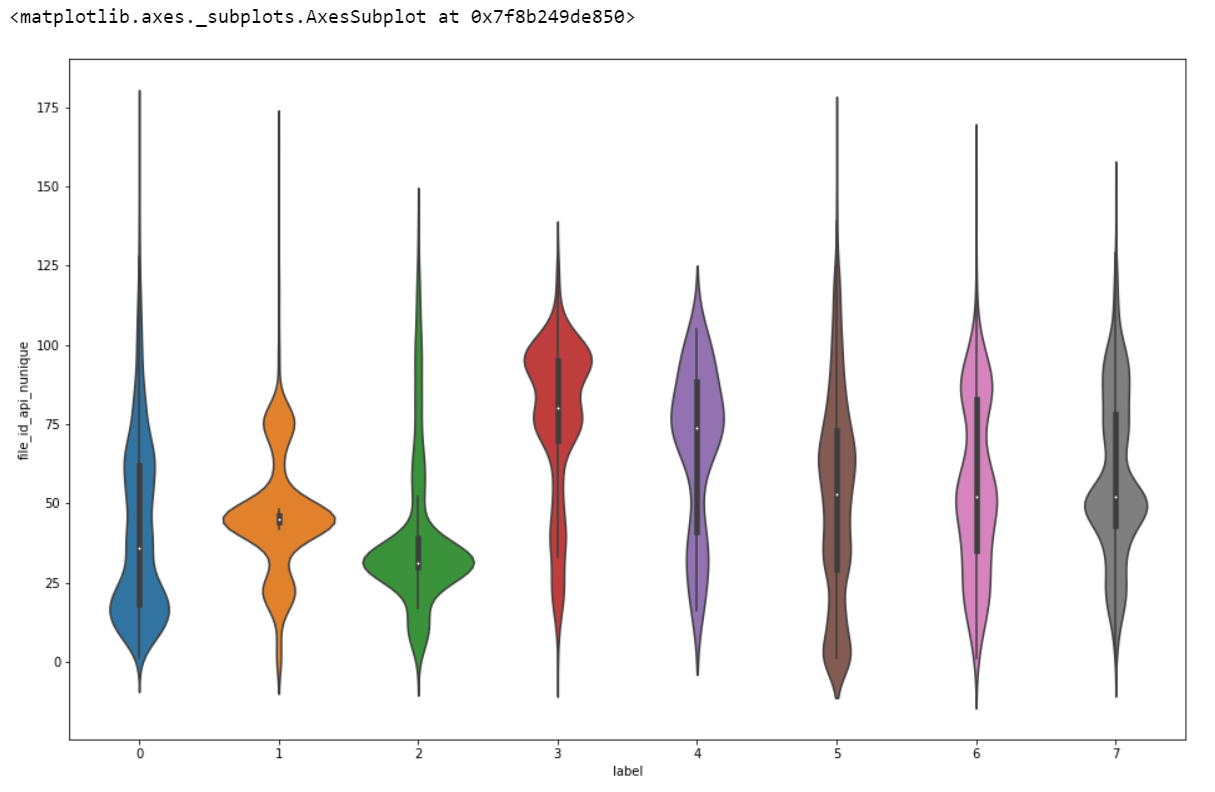


从图中可以发现，在文件顺序编号只有一个时，文件的标签只会是0（正常）、2（挖矿程序）或5（感染型病毒），而不会是其他病毒，而且最大概率可能是5；对于顺序次数大于5000个的文件，其和上面调用API次数很大时类似。

还可以通过绘制小提琴图、分类散点图分析，代码和结果如下：

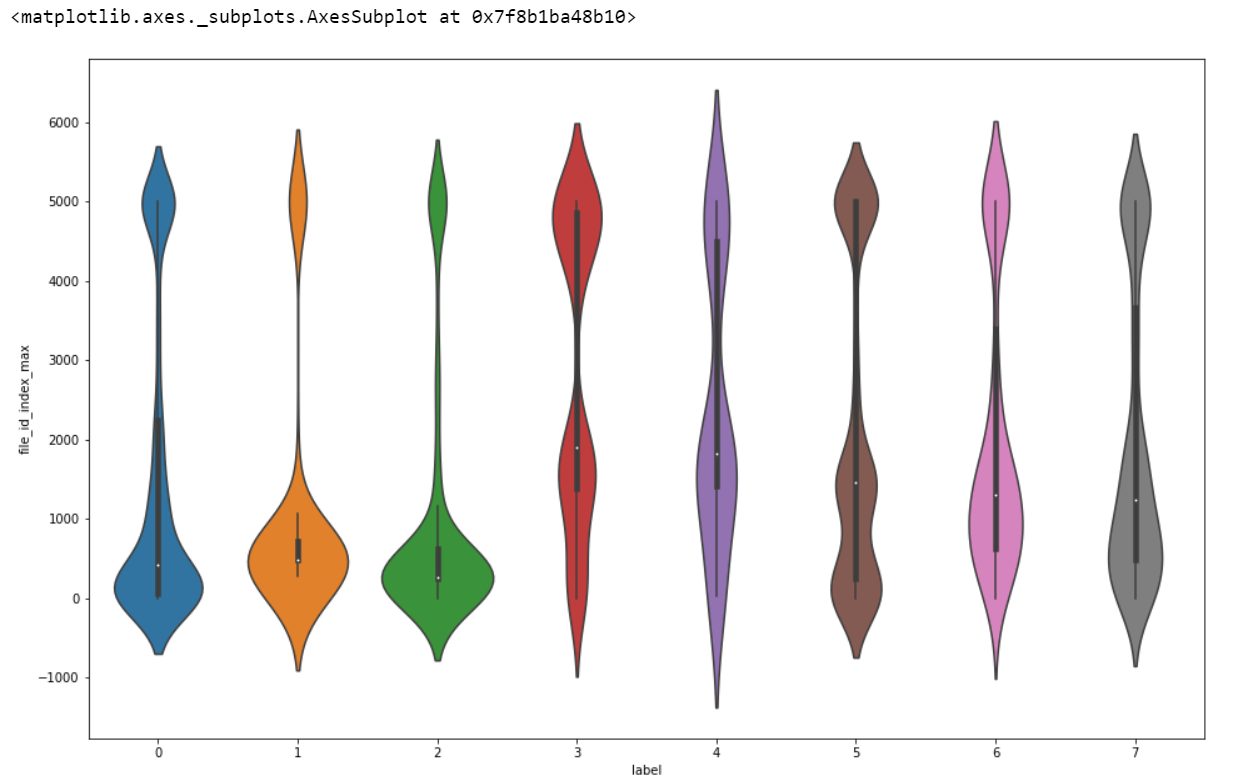
plt.figure(figsize=[16, 10])

sns.violinplot(x=train\_analysis['label'], y=train\_analysis['file\_id\_api\_nunique'])



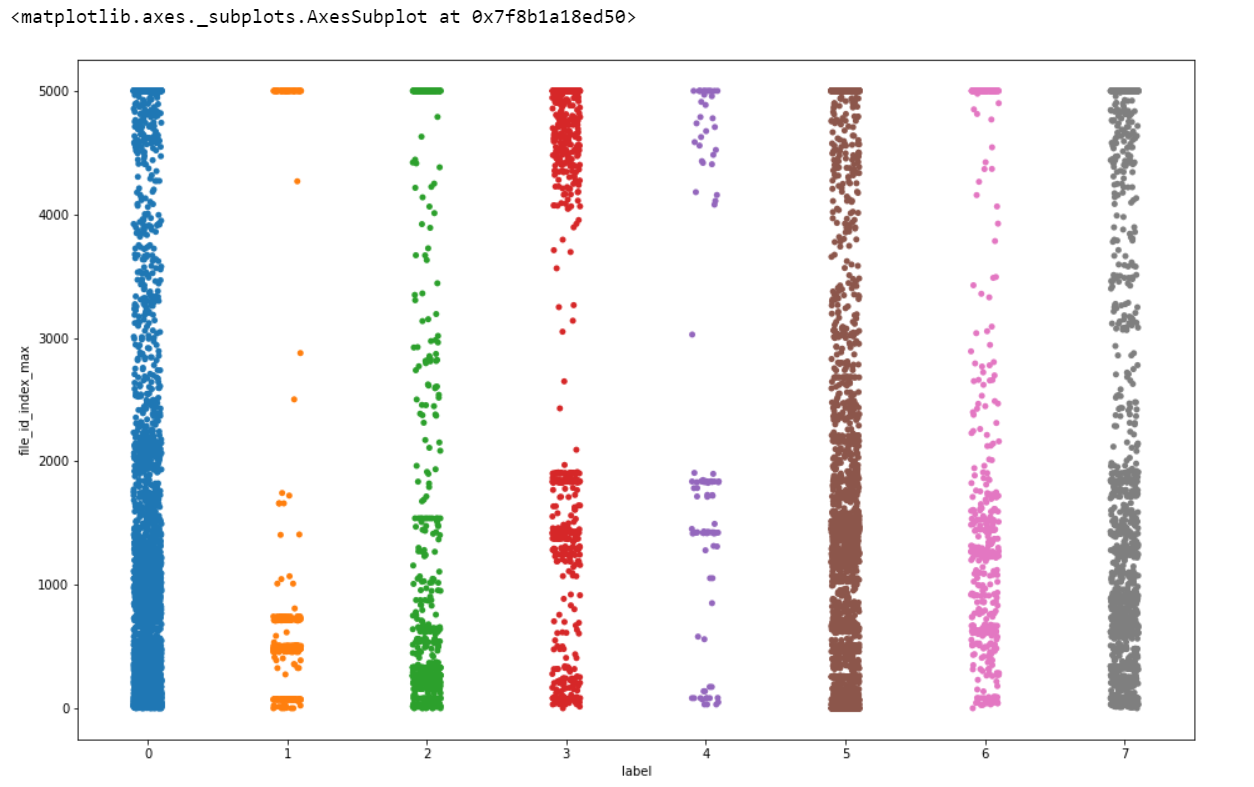
plt.figure(figsize=[16, 10])

sns.violinplot(x=train\_analysis['label'], y=train\_analysis['file\_id\_index\_max'])



plt.figure(figsize=[16, 10])

sns.stripplot(x=train\_analysis['label'], y=train\_analysis['file\_id\_index\_max'])



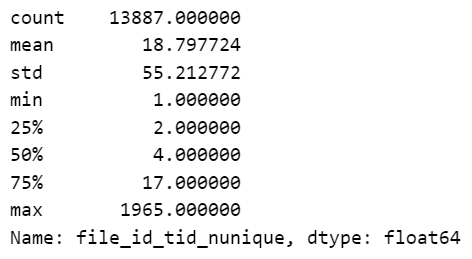
从图中得到的结论：第3类病毒调用不同index次数的平均值最大；第2类病毒调用不同index的平均值最小；第5,6,7类病毒调用不同index次数的平均值相似。

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。

dic\_ = train.groupby('file\_id')['tid'].nunique().to\_dict()

train\_analysis['file\_id\_tid\_nunique'] = train\_analysis['file\_id'].map(dic\_).values

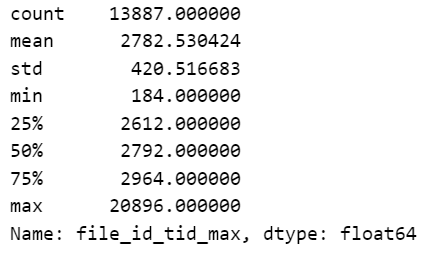
train\_analysis['file\_id\_tid\_nunique'].describe()



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()



然后分析file\_id\_tid\_nunique和file\_id\_tid\_max与标签label变量间的关系。

plt.figure(figsize=[16, 8])

plt.subplot(121)

train\_analysis.loc[train\_analysis.file\_id\_tid\_nunique < 5]['label'].value\_counts().sort\_index().plot(kind='bar')

plt.title('File with tid nunique < 5')

plt.xlabel('label')

plt.ylabel('label\_number')

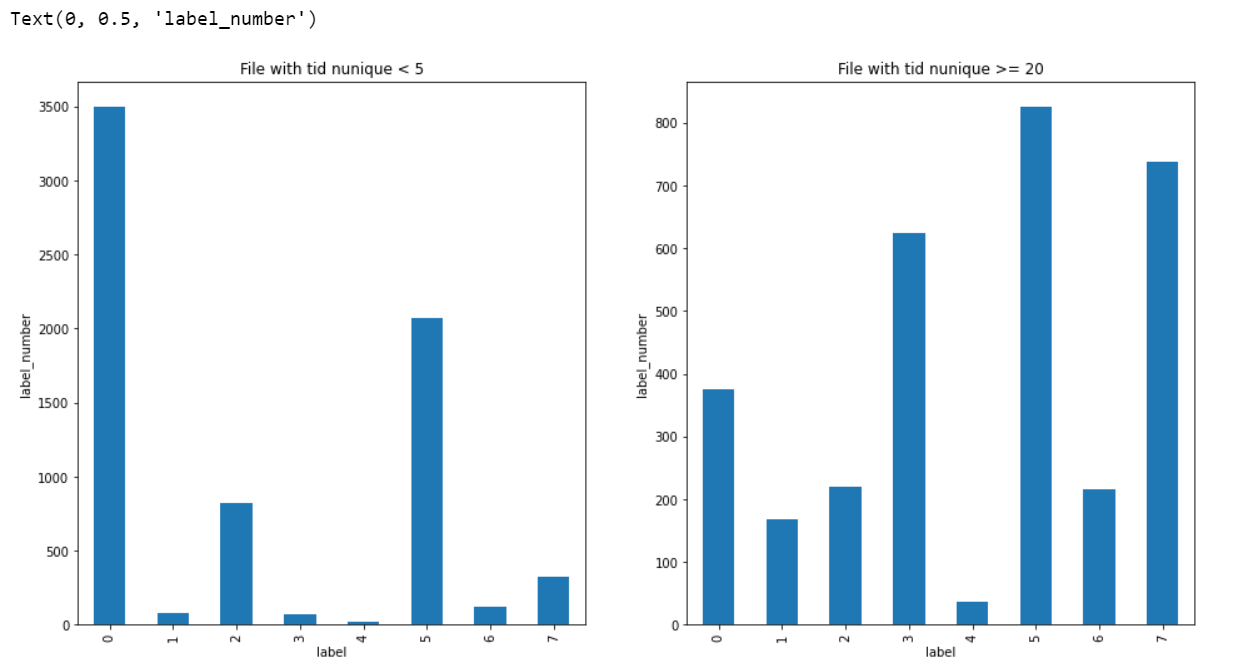
plt.subplot(122)

train\_analysis.loc[train\_analysis.file\_id\_tid\_nunique >= 20]['label'].value\_counts().sort\_index().plot(kind='bar')

plt.title('File with tid nunique >= 20')

plt.xlabel('label')

plt.ylabel('label\_number')

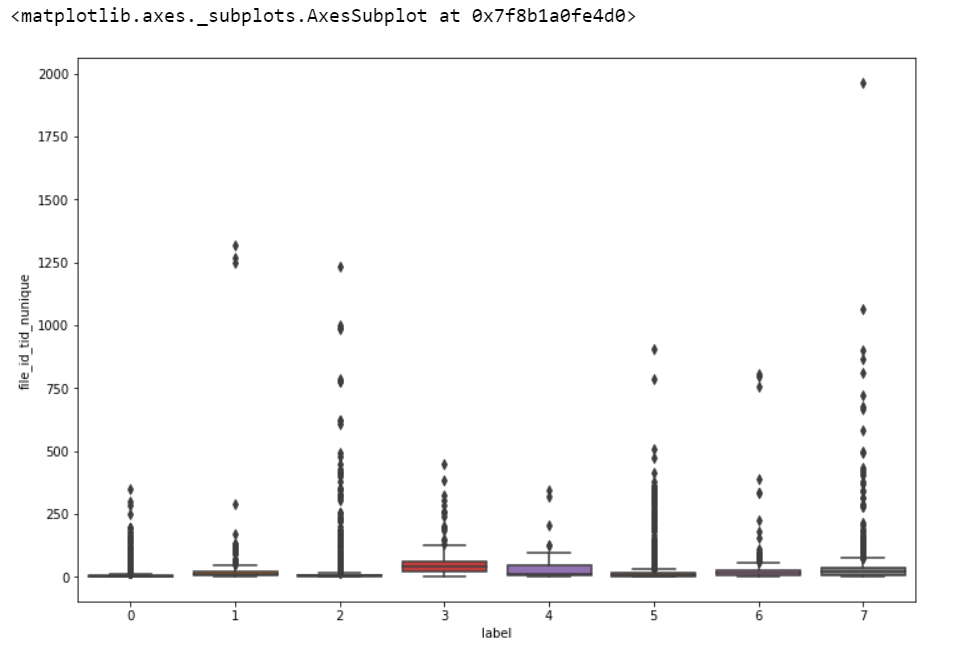


其中，0：正常文件；1：勒索病毒；2：挖矿程序；3：DDoS木马；4：蠕虫病毒；5：感染型病毒；6：后门程序；7：木马程序。

还可以通过箱线图和小提琴图进一步分析。

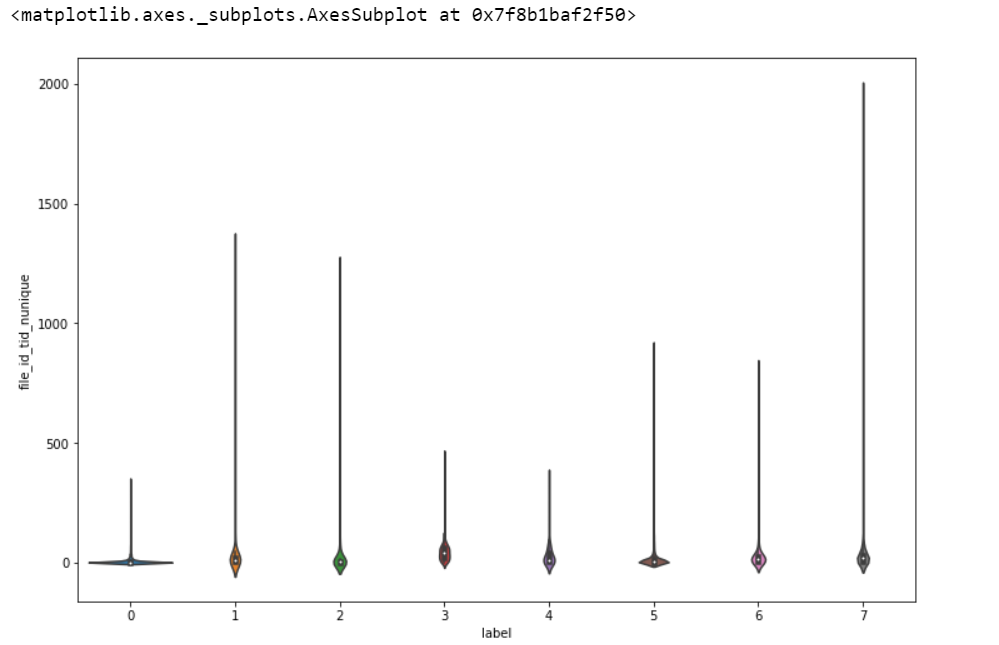
plt.figure(figsize=[12, 8])

sns.boxplot(x=train\_analysis['label'], y=train\_analysis['file\_id\_tid\_nunique'])



plt.figure(figsize=[12, 8])

sns.violinplot(x=train\_analysis['label'], y=train\_analysis['file\_id\_tid\_nunique'])



分析file\_id和tid的max特征，我们将tid的最大值大于3000的数据和整体比较，发现分布差异并不是非常大。

plt.figure(figsize=[12, 8])

plt.subplot(121)

train\_analysis.loc[train\_analysis.file\_id\_tid\_max >= 3000]['label'].value\_counts().sort\_index().plot(kind='bar')

plt.title('File with tid max >= 3000')

plt.xlabel('label')

plt.ylabel('label\_number')

plt.subplot(122)

train\_analysis['label'].value\_counts().sort\_index().plot(kind='bar')

plt.title('All Data')

plt.xlabel('label')

plt.ylabel('label\_number')

从图中得到结论：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类调用的不同线程数类似。

6. 分析API变量与label变量的关系，代码及运行结果如下：

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]))

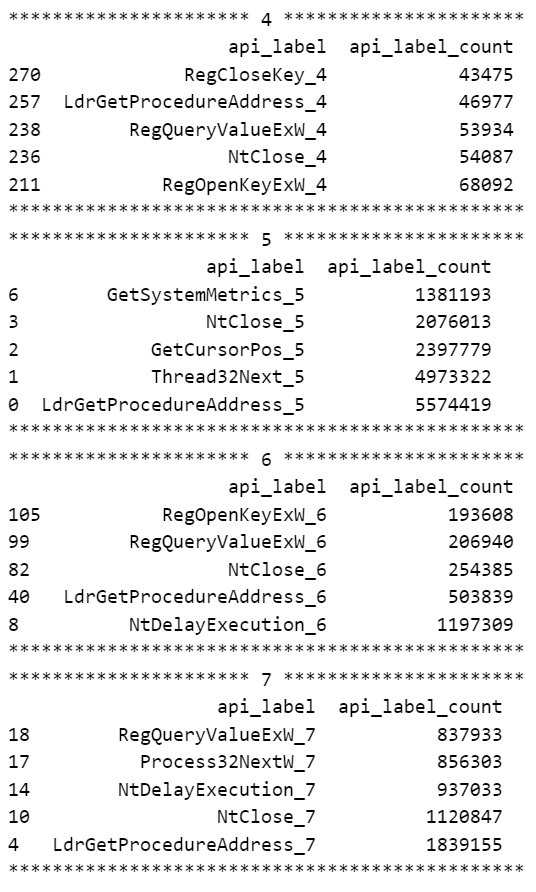
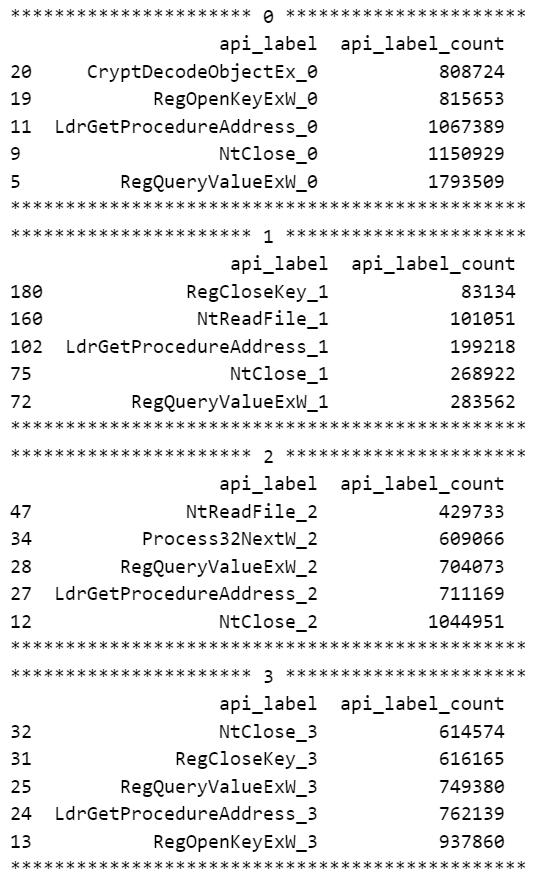
labels = df\_api\_label['label'].unique()

for label in range(8):

    print('\*' \* 22, label, '\*' \* 22)

    print(df\_api\_label.loc[df\_api\_label.label == label].sort\_values('api\_label\_count').iloc[-5:][['api\_label', 'api\_label\_count']])

    print('\*' \* 47)



从结果可以得到以下结论：LdrGetProcedureAddress，所有病毒和正常文件都是调用比较多的；第5类病毒：Thread32Next调用得较多；第6,7类病毒：NtDelayExecution调用得较多；第2,7类病毒：Process32NextW调用得较多。

## 特征工程进阶与方案优化(Advanced Feature Engineering and Program Optimization)

在快速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，迭代优化一般新的模型。

pivot特征是采用pandas.pivot操作获得的特征，其本质是分层统计特征，同时也是一种组合特征。很多时候因为样本在每层的表现都不一样，所以需要我们先对特征进行分层，然后在新层对特征进行构建，此时的特征相较于直接用所有层构建得到的特征更加细化，也更具有代表性。

### 特征工程基础部分

导入库

import pandas as pd

import numpy as np

import seaborn as sns

import matplotlib.pyplot as plt

import lightgbm as lgb

from sklearn.model\_selection import train\_test\_split

from sklearn.preprocessing import OneHotEncoder

from tqdm import tqdm\_notebook

import warnings

warnings.filterwarnings('ignore')

%matplotlib inline

读取数据

path  = '../security\_data/'

train = pd.read\_csv(path + 'security\_train.csv')

test  = pd.read\_csv(path + 'security\_test.csv')

内存管理

import numpy as np

import pandas as pd

from tqdm import tqdm

class \_Data\_Preprocess:

    def \_\_init\_\_(self):

        self.int8\_max = np.iinfo(np.int8).max

        self.int8\_min = np.iinfo(np.int8).min

        self.int16\_max = np.iinfo(np.int16).max

        self.int16\_min = np.iinfo(np.int16).min

        self.int32\_max = np.iinfo(np.int32).max

        self.int32\_min = np.iinfo(np.int32).min

        self.int64\_max = np.iinfo(np.int64).max

        self.int64\_min = np.iinfo(np.int64).min

        self.float16\_max = np.finfo(np.float16).max

        self.float16\_min = np.finfo(np.float16).min

        self.float32\_max = np.finfo(np.float32).max

        self.float32\_min = np.finfo(np.float32).min

        self.float64\_max = np.finfo(np.float64).max

        self.float64\_min = np.finfo(np.float64).min

    def \_get\_type(self, min\_val, max\_val, types):

        if types == 'int':

            if max\_val <= self.int8\_max and min\_val >= self.int8\_min:

                return np.int8

            elif max\_val <= self.int16\_max <= max\_val and min\_val >= self.int16\_min:

                return np.int16

            elif max\_val <= self.int32\_max and min\_val >= self.int32\_min:

                return np.int32

            return None

        elif types == 'float':

            if max\_val <= self.float16\_max and min\_val >= self.float16\_min:

                return np.float16

            if max\_val <= self.float32\_max and min\_val >= self.float32\_min:

                return np.float32

            if max\_val <= self.float64\_max and min\_val >= self.float64\_min:

                return np.float64

            return None

    def \_memory\_process(self, df):

        init\_memory = df.memory\_usage().sum() / 1024 \*\* 2 / 1024

        print('Original data occupies {} GB memory.'.format(init\_memory))

        df\_cols = df.columns

        for col in tqdm\_notebook(df\_cols):

            try:

                if 'float' in str(df[col].dtypes):

                    max\_val = df[col].max()

                    min\_val = df[col].min()

                    trans\_types = self.\_get\_type(min\_val, max\_val, 'float')

                    if trans\_types is not None:

                        df[col] = df[col].astype(trans\_types)

                elif 'int' in str(df[col].dtypes):

                    max\_val = df[col].max()

                    min\_val = df[col].min()

                    trans\_types = self.\_get\_type(min\_val, max\_val, 'int')

                    if trans\_types is not None:

                        df[col] = df[col].astype(trans\_types)

            except:

                print(' Can not do any process for column, {}.'.format(col))

        afterprocess\_memory = df.memory\_usage().sum() / 1024 \*\* 2 / 1024

        print('After processing, the data occupies {} GB memory.'.format(afterprocess\_memory))

        return df

memory\_process = \_Data\_Preprocess()

基础特征工程构造

def simple\_sts\_features(df):

    simple\_fea = pd.DataFrame()

    simple\_fea['file\_id'] = df['file\_id'].unique()

    simple\_fea = simple\_fea.sort\_values('file\_id')

    df\_grp = df.groupby('file\_id')

    simple\_fea['file\_id\_api\_count'] = df\_grp['api'].count().values

    simple\_fea['file\_id\_api\_nunique'] = df\_grp['api'].nunique().values

    simple\_fea['file\_id\_tid\_count'] = df\_grp['tid'].count().values

    simple\_fea['file\_id\_tid\_nunique'] = df\_grp['tid'].nunique().values

    simple\_fea['file\_id\_index\_count'] = df\_grp['index'].count().values

    simple\_fea['file\_id\_index\_nunique'] = df\_grp['index'].nunique().values

    return simple\_fea

def simple\_numerical\_sts\_features(df):

    simple\_numerical\_fea = pd.DataFrame()

    simple\_numerical\_fea['file\_id'] = df['file\_id'].unique()

    simple\_numerical\_fea = simple\_numerical\_fea.sort\_values('file\_id')

    df\_grp = df.groupby('file\_id')

    simple\_numerical\_fea['file\_id\_tid\_mean'] = df\_grp['tid'].mean().values

    simple\_numerical\_fea['file\_id\_tid\_min'] = df\_grp['tid'].min().values

    simple\_numerical\_fea['file\_id\_tid\_std'] = df\_grp['tid'].std().values

    simple\_numerical\_fea['file\_id\_tid\_max'] = df\_grp['tid'].max().values

    simple\_numerical\_fea['file\_id\_index\_mean'] = df\_grp['index'].mean().values

    simple\_numerical\_fea['file\_id\_index\_min'] = df\_grp['index'].min().values

    simple\_numerical\_fea['file\_id\_index\_std'] = df\_grp['index'].std().values

    simple\_numerical\_fea['file\_id\_index\_max'] = df\_grp['index'].max().values

    return simple\_numerical\_fea

特征获取

simple\_train\_fea1 = simple\_sts\_features(train)

simple\_test\_fea1 = simple\_sts\_features(test)

simple\_train\_fea2 = simple\_numerical\_sts\_features(train)

simple\_test\_fea2 = simple\_numerical\_sts\_features(test)

### 特征工程进阶部分

每个API调用线程tid的次数

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 = memory\_process.\_memory\_process(tmp\_pivot)

    return tmp\_pivot

每个API调用不同线程tid的次数

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 = memory\_process.\_memory\_process(tmp\_pivot)

    return tmp\_pivot

特征获取

simple\_train\_fea3 = api\_pivot\_count\_features(train)

simple\_test\_fea3 = api\_pivot\_count\_features(test)

simple\_train\_fea4 = api\_pivot\_nunique\_features(train)

simple\_test\_fea4 = api\_pivot\_nunique\_features(test)

### 基于LightGBM的模型验证

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')

训练集与测试集的构建，此处将之前提取的特征与新生成的特征进行合并。

train\_data = train\_label.merge(simple\_train\_fea1, on ='file\_id', how='left')

train\_data = train\_data.merge(simple\_train\_fea2, on ='file\_id', how='left')

train\_data = train\_data.merge(simple\_train\_fea3, on ='file\_id', how='left')

train\_data = train\_data.merge(simple\_train\_fea4, on ='file\_id', how='left')

test\_submit = test\_submit.merge(simple\_test\_fea1, on ='file\_id', how='left')

test\_submit = test\_submit.merge(simple\_test\_fea2, on ='file\_id', how='left')

test\_submit = test\_submit.merge(simple\_test\_fea3, on ='file\_id', how='left')

test\_submit = test\_submit.merge(simple\_test\_fea4, on ='file\_id', how='left')

构建评估指标logloss

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]):

        sum\_ = 0

        for j in range(preds\_prob\_.shape[0]):

            pred = preds\_prob\_[j,i]

            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

模型采用5折交叉验证方式

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

    }

folds = KFold(n\_splits=5, shuffle=True, random\_state=15)

oof = np.zeros(len(train))

predict\_res = 0

models = []

for fold\_, (trn\_idx, val\_idx) in enumerate(folds.split(train\_data)):

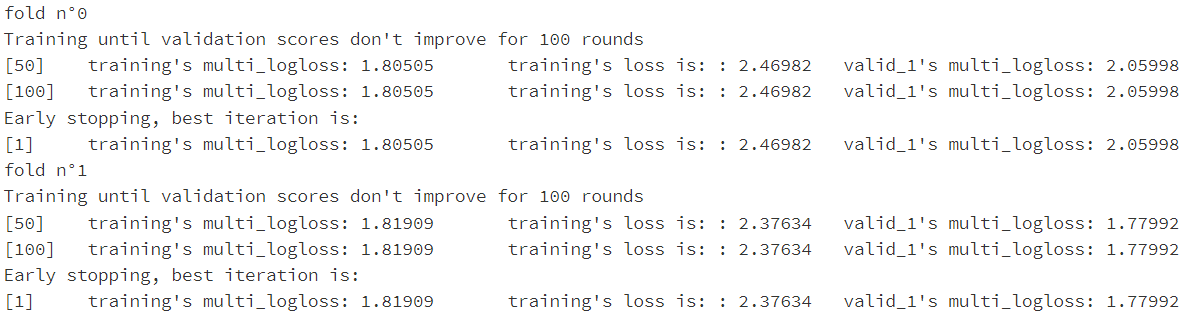
    print("fold n°{}".format(fold\_))

    trn\_data = lgb.Dataset(train\_data.iloc[trn\_idx][train\_features], label=train\_data.iloc[trn\_idx][train\_label].values)

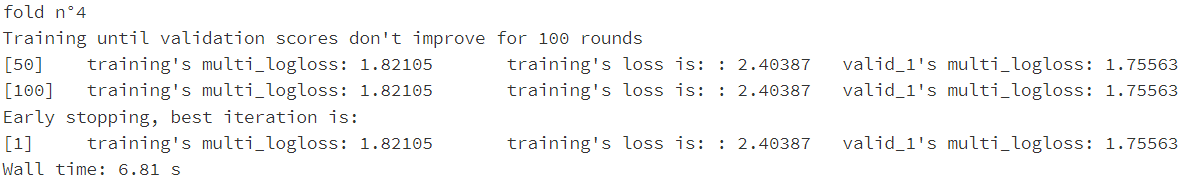
    val\_data = lgb.Dataset(train\_data.iloc[val\_idx][train\_features], label=train\_data.iloc[val\_idx][train\_label].values)

    clf = lgb.train(params, trn\_data, num\_boost\_round=2000,valid\_sets=[trn\_data,val\_data], verbose\_eval=50, early\_stopping\_rounds=100, feval=lgb\_logloss)

    models.append(clf)



…



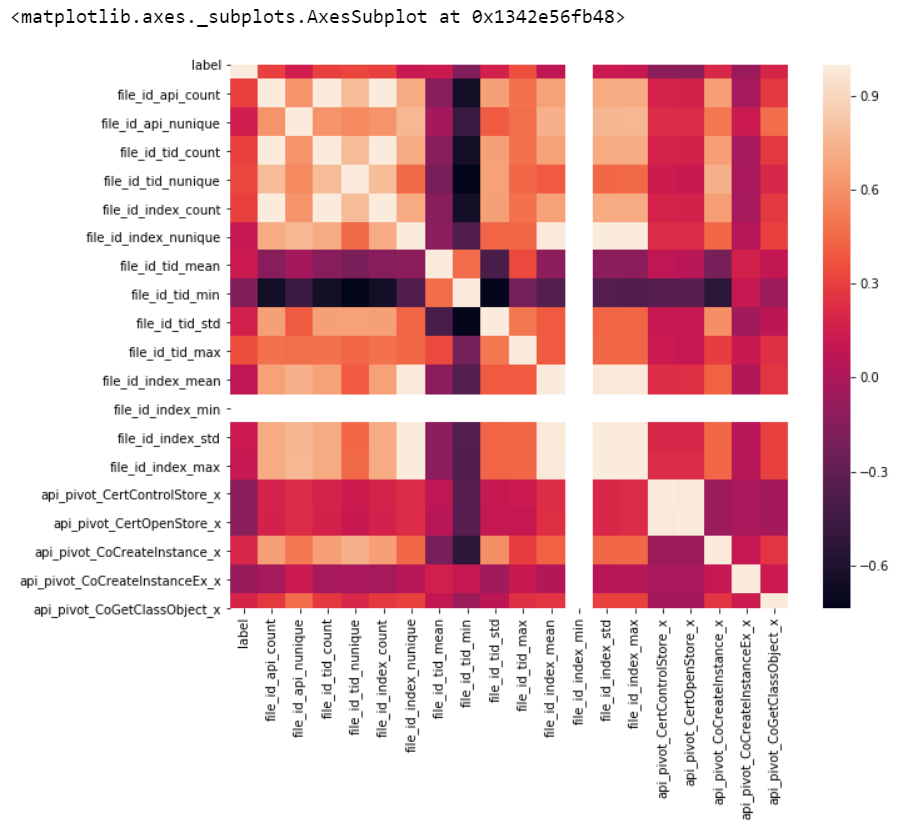
### 模型结果分析

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10000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())



通过查看特征变量与label的相关性，我们也可以再次验证之前数据探索EDA部分的结论，每个文件调用API的次数与病毒类型是强相关的。

特征重要性分析

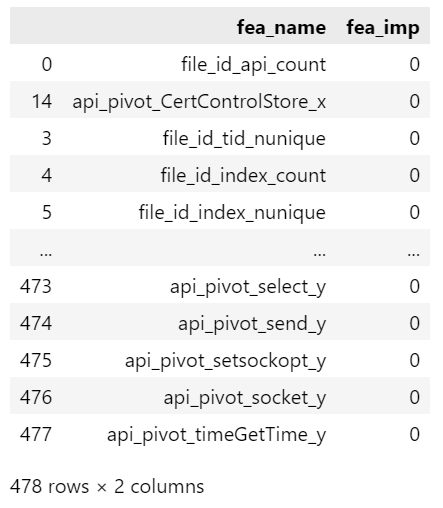
feature\_importance = pd.DataFrame()

feature\_importance['fea\_name'] = train\_features

feature\_importance['fea\_imp'] = clf.feature\_importance()

feature\_importance = feature\_importance.sort\_values('fea\_imp', ascending=False)

feature\_importance.sort\_values('fea\_imp', ascending=False)



对特征重要性分析也再一次验证了我们的想法：API调用次数及API调用类别数是最重要的两个特征，也就是说不同的病毒常常会调用不同的API，而且因为有些病毒需要复制自身，调用API的次数会非常多；第3到第5强的都是线程统计特征，这也较为容易理解，因为木马等病毒经常需要通过线程监听一些内容，所以在线程数量的使用上也会表现的略有不同。

### 模型测试

此处，我们采用LightGBM模型进行5折交叉验证，取预测均值作为最终结果。

LightGBM模型进行5折交叉验证的预测并不能100%超过单折全量数据的LightGBM，但是在大多数时候取得的效果还是不错的。

pred\_res = 0

flod = 5

for model in models:

    pred\_res += model.predict(test\_submit[train\_features]) \* 1.0 / flod

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']].to\_csv('baseline2.csv',index = None)

线下成绩：0.568286

## 优化技巧与解决方案升级(Optimization Tips and Solution Upgrades)

### Python处理大数据

#### 内存管理控制

利用数据类型控制内存。首先判断特征列取值的最小表示范围，然后进行类型转换，如float64转换为float16等。

import numpy as np

import pandas as pd

from tqdm import tqdm

class \_Data\_Preprocess:

    def \_\_init\_\_(self):

        self.int8\_max = np.iinfo(np.int8).max

        self.int8\_min = np.iinfo(np.int8).min

        self.int16\_max = np.iinfo(np.int16).max

        self.int16\_min = np.iinfo(np.int16).min

        self.int32\_max = np.iinfo(np.int32).max

        self.int32\_min = np.iinfo(np.int32).min

        self.int64\_max = np.iinfo(np.int64).max

        self.int64\_min = np.iinfo(np.int64).min

        self.float16\_max = np.finfo(np.float16).max

        self.float16\_min = np.finfo(np.float16).min

        self.float32\_max = np.finfo(np.float32).max

        self.float32\_min = np.finfo(np.float32).min

        self.float64\_max = np.finfo(np.float64).max

        self.float64\_min = np.finfo(np.float64).min

    def \_get\_type(self, min\_val, max\_val, types):

        if types == 'int':

            if max\_val <= self.int8\_max and min\_val >= self.int8\_min:

                return np.int8

            elif max\_val <= self.int16\_max <= max\_val and min\_val >= self.int16\_min:

                return np.int16

            elif max\_val <= self.int32\_max and min\_val >= self.int32\_min:

                return np.int32

            return None

        elif types == 'float':

            if max\_val <= self.float16\_max and min\_val >= self.float16\_min:

                return np.float16

            if max\_val <= self.float32\_max and min\_val >= self.float32\_min:

                return np.float32

            if max\_val <= self.float64\_max and min\_val >= self.float64\_min:

                return np.float64

            return None

    def \_memory\_process(self, df):

        init\_memory = df.memory\_usage().sum() / 1024 \*\* 2 / 1024

        print('Original data occupies {} GB memory.'.format(init\_memory))

        df\_cols = df.columns

        for col in tqdm\_notebook(df\_cols):

            try:

                if 'float' in str(df[col].dtypes):

                    max\_val = df[col].max()

                    min\_val = df[col].min()

                    trans\_types = self.\_get\_type(min\_val, max\_val, 'float')

                    if trans\_types is not None:

                        df[col] = df[col].astype(trans\_types)

                elif 'int' in str(df[col].dtypes):

                    max\_val = df[col].max()

                    min\_val = df[col].min()

                    trans\_types = self.\_get\_type(min\_val, max\_val, 'int')

                    if trans\_types is not None:

                        df[col] = df[col].astype(trans\_types)

            except:

                print(' Can not do any process for column, {}.'.format(col))

        afterprocess\_memory = df.memory\_usage().sum() / 1024 \*\* 2 / 1024

        print('After processing, the data occupies {} GB memory.'.format(afterprocess\_memory))

        return df

memory\_process = \_Data\_Preprocess()

#### 加速数据处理

1. 加速Pandas的merge

当数据量较大时，Pandas的merge操作相比基于object的merge操作耗时，在比赛中index经常会编码成一个非常复杂的字符串序列，此时我们可以直接将index编码为简单的数字，然后存储映射的字典，再对数字进行merge，最后通过字典映射回来。

2. 加速Pandas分位数的特征提取

在比赛中经常会遇到各种分位数的问题，这时有些选手经常会枚举分位数特征，但如果每次都自定义一个分位数提取函数会及极其耗时。这时就可以考虑将所有分位数的提取用一个函数实现，返回一个分位数的list，这样就无须再进行多次分位数的提取了。

3. 用Numpy替换Pandas

由于Numpy的操作比Pandas操作得快，因此当Pandas全部是数值等特征时，可以考虑将其转换为Numpy再进行特征提取。

#### 其他开源工具包

1. Dask

Dask 是一个灵活的 Python 并行计算库。

Dask由两部分组成：

i. 针对计算优化的动态任务调度。这类似于 Airflow、Luigi、Celery 或 Make，但针对交互式计算工作负载进行了优化。

ii. “大数据”集合，如并行数组、数据帧和列表，将NumPy、Pandas 或 Python 迭代器等常见接口扩展到大于内存或分布式环境。这些并行集合在动态任务调度程序之上运行。

Dask 有以下优点：

熟悉：提供并行化的 NumPy 数组和 Pandas DataFrame 对象

灵活：为更多自定义工作负载和与其他项目的集成提供任务调度界面。

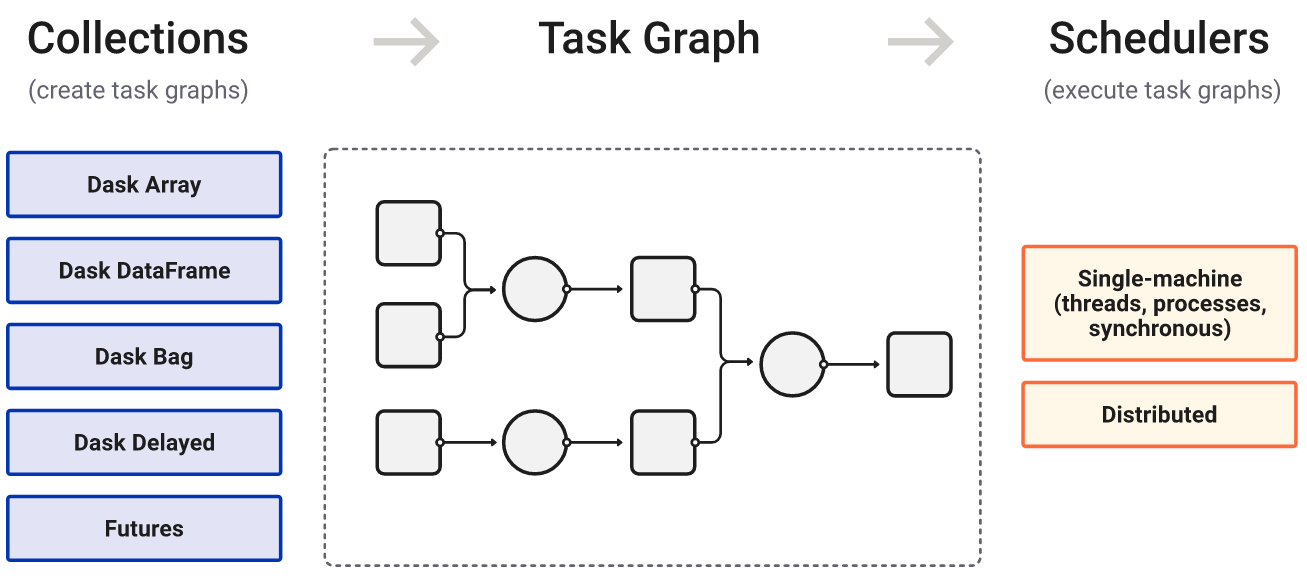
原生：在纯 Python 中启用分布式计算，并访问 PyData 堆栈。

快速：以低开销、低延迟和快速数值算法所需的最小序列化运行

规模增大：在具有 1000 个内核的集群上弹性运行

规模缩小：在单个进程中在笔记本电脑上设置和运行微不足道

响应：设计时考虑到交互式计算，它提供快速反馈和诊断以提供帮助



2. Numba

Numba是适用于Python的即时编译器，最适合于使用NumPy数组和函数以及循环的代码。使用Numba最常见的方法是通过它的装饰器，这些装饰器可以应用于您的函数以指示Numba编译它们。当调用一个Numba修饰的函数时，它会被编译为“即时”的机器代码以供执行，并且全部或部分代码随后可以以本机机器代码的速度运行。

以装饰器JIT为例，对于如下代码

**from** numba **import** jit

**import** time

SIZE**=**10000

**def** normal\_test**():**

**for** \_ **in** **range(**SIZE**):**

**for** \_ **in** **range(**SIZE**):**

**pass**

*@jit*

**def** jit\_test**():**

**for** \_ **in** **range(**SIZE**):**

**for** \_ **in** **range(**SIZE**):**

**pass**

**def** main**():**

start**=**time**.**time**()**

normal\_test**()**

end**=**time**.**time**()**

**print(**end**-**start**)**

start**=**time**.**time**()**

jit\_test**()**

end**=**time**.**time**()**

**print(**end**-**start**)**

start**=**time**.**time**()**

jit\_test**()**

end**=**time**.**time**()**

**print(**end**-**start**)**

**if** \_\_name\_\_**==**'\_\_main\_\_'**:**

main**()**

其输出为

1.5333094596862793

0.2395479679107666

0.0

可以发现后两次执行的函数相比与第一次执行的函数在速度上有巨大的提升。

这是因为JIT使用懒编译（Lazy Compilation）技术，在执行到@jit代码块时才将其编译，而纯编译型语言C会在程序运行前整个编译好。

因为被@jit修饰的代码块已经编译过了，，所以第二次运行时编译时间=0，又因为循环体内为pass，所以运行时间=0，故总时间=0

另外，原生Python慢的原因还有一点在于解释器需要推断变量类型，@jit可以使用Eager Compilation技术，指定变量类型，这样可以提高编译速度。

### 深度学习解决方案——TextCNN

#### 问题转化

和传统机器学习模型相比，当用深度学习对该问题进行建模时，需要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，这样问题就变成了一个文本分类问题，同时API序列之间的顺序具有物理意义，等价于文本分类问题。

#### TextCNN建模

导入库

import pandas as pd

import numpy as np

import seaborn as sns

import matplotlib.pyplot as plt

import lightgbm as lgb

from sklearn.model\_selection import train\_test\_split

from sklearn.preprocessing import OneHotEncoder

from tqdm import tqdm\_notebook

from sklearn.preprocessing import LabelBinarizer,LabelEncoder

import warnings

warnings.filterwarnings('ignore')

%matplotlib inline

读取数据

path  = '../security\_data/'

train = pd.read\_csv(path + 'security\_train.csv')

test  = pd.read\_csv(path + 'security\_test.csv')

内存管理

import numpy as np

import pandas as pd

from tqdm import tqdm

class \_Data\_Preprocess:

    def \_\_init\_\_(self):

        self.int8\_max = np.iinfo(np.int8).max

        self.int8\_min = np.iinfo(np.int8).min

        self.int16\_max = np.iinfo(np.int16).max

        self.int16\_min = np.iinfo(np.int16).min

        self.int32\_max = np.iinfo(np.int32).max

        self.int32\_min = np.iinfo(np.int32).min

        self.int64\_max = np.iinfo(np.int64).max

        self.int64\_min = np.iinfo(np.int64).min

        self.float16\_max = np.finfo(np.float16).max

        self.float16\_min = np.finfo(np.float16).min

        self.float32\_max = np.finfo(np.float32).max

        self.float32\_min = np.finfo(np.float32).min

        self.float64\_max = np.finfo(np.float64).max

        self.float64\_min = np.finfo(np.float64).min

    def \_get\_type(self, min\_val, max\_val, types):

        if types == 'int':

            if max\_val <= self.int8\_max and min\_val >= self.int8\_min:

                return np.int8

            elif max\_val <= self.int16\_max <= max\_val and min\_val >= self.int16\_min:

                return np.int16

            elif max\_val <= self.int32\_max and min\_val >= self.int32\_min:

                return np.int32

            return None

        elif types == 'float':

            if max\_val <= self.float16\_max and min\_val >= self.float16\_min:

                return np.float16

            if max\_val <= self.float32\_max and min\_val >= self.float32\_min:

                return np.float32

            if max\_val <= self.float64\_max and min\_val >= self.float64\_min:

                return np.float64

            return None

    def \_memory\_process(self, df):

        init\_memory = df.memory\_usage().sum() / 1024 \*\* 2 / 1024

        print('Original data occupies {} GB memory.'.format(init\_memory))

        df\_cols = df.columns

        for col in tqdm\_notebook(df\_cols):

            try:

                if 'float' in str(df[col].dtypes):

                    max\_val = df[col].max()

                    min\_val = df[col].min()

                    trans\_types = self.\_get\_type(min\_val, max\_val, 'float')

                    if trans\_types is not None:

                        df[col] = df[col].astype(trans\_types)

                elif 'int' in str(df[col].dtypes):

                    max\_val = df[col].max()

                    min\_val = df[col].min()

                    trans\_types = self.\_get\_type(min\_val, max\_val, 'int')

                    if trans\_types is not None:

                        df[col] = df[col].astype(trans\_types)

            except:

                print(' Can not do any process for column, {}.'.format(col))

        afterprocess\_memory = df.memory\_usage().sum() / 1024 \*\* 2 / 1024

        print('After processing, the data occupies {} GB memory.'.format(afterprocess\_memory))

        return df

memory\_process = \_Data\_Preprocess()

#### 数据预处理

1. 字符串转换为数字

为了将数据输入神经网络，需要将字符串转换为数字，此处可以对API构建映射表将其转换为数字。

unique\_api = train['api'].unique()

api2index = {item:(i+1) for i,item in enumerate(unique\_api)}

index2api = {(i+1):item for i,item in enumerate(unique\_api)}

train['api\_idx'] = train['api'].map(api2index)

test['api\_idx']  = test['api'].map(api2index)

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

def get\_sequence(df,period\_idx):

    seq\_list = []

    for \_id,begin in enumerate(period\_idx[:-1]):

        seq\_list.append(df.iloc[begin:period\_idx[\_id+1]]['api\_idx'].values)

    seq\_list.append(df.iloc[period\_idx[-1]:]['api\_idx'].values)

    return seq\_list

train\_period\_idx = train.file\_id.drop\_duplicates(keep='first').index.values

test\_period\_idx  = test.file\_id.drop\_duplicates(keep='first').index.values

train\_df = train[['file\_id','label']].drop\_duplicates(keep='first')

test\_df  = test[['file\_id']].drop\_duplicates(keep='first')

train\_df['seq'] = get\_sequence(train,train\_period\_idx)

test\_df['seq']  = get\_sequence(test,test\_period\_idx)

#### TextCNN网络结构

1. 导入库

from keras.preprocessing.text import Tokenizer

from keras.preprocessing.sequence import pad\_sequences

from keras.layers import Dense, Input, LSTM, Lambda, Embedding, Dropout, Activation,GRU,Bidirectional

from keras.layers import Conv1D,Conv2D,MaxPooling2D,GlobalAveragePooling1D,GlobalMaxPooling1D, MaxPooling1D, Flatten

from keras.layers import CuDNNGRU, CuDNNLSTM, SpatialDropout1D

from keras.layers.merge import concatenate, Concatenate, Average, Dot, Maximum, Multiply, Subtract, average

from keras.models import Model

from keras.optimizers import RMSprop,Adam

from keras.layers.normalization import BatchNormalization

from keras.callbacks import EarlyStopping, ModelCheckpoint

from keras.optimizers import SGD

from keras import backend as K

from sklearn.decomposition import TruncatedSVD, NMF, LatentDirichletAllocation

from keras.layers import SpatialDropout1D

from keras.layers.wrappers import Bidirectional

2. 定义TextCNN网络

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)

    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

#### TextCNN训练和测试

考虑到GPU的存储空间，这里对序列进行了一定长度的截断，此处将长度设置为6000。

train\_labels = pd.get\_dummies(train\_df.label).values

train\_seq = pad\_sequences(train\_df.seq.values, maxlen = 6000)

test\_seq = pad\_sequences(test\_df.seq.values, maxlen = 6000)

模型训练和预测

1. 导入库，设置KFold

import os

from sklearn.model\_selection import StratifiedKFold,KFold

skf = KFold(n\_splits=5, shuffle=True)

2. 设定参数

max\_len = 6000

max\_cnt = 295

embed\_size = 256

num\_filters = 64

kernel\_size = [2,4,6,8,10,12,14]

conv\_action = 'relu'

mask\_zero = False

TRAIN = True

3. 模型训练和预测

os.environ["CUDA\_VISIBLE\_DEVICES"] = "0,1"

meta\_train = np.zeros(shape = (len(train\_seq),8))

meta\_test = np.zeros(shape = (len(test\_seq),8))

FLAG = True

i = 0

for tr\_ind,te\_ind in skf.split(train\_labels):

    i +=1

    print('FOLD: '.format(i))

    print(len(te\_ind),len(tr\_ind))

    model\_name = 'benchmark\_textcnn\_fold\_'+str(i)

    X\_train,X\_train\_label = train\_seq[tr\_ind],train\_labels[tr\_ind]

    X\_val,X\_val\_label     = train\_seq[te\_ind],train\_labels[te\_ind]

    model = TextCNN(max\_len,max\_cnt,embed\_size,num\_filters,kernel\_size,conv\_action,mask\_zero)

    model\_save\_path = './NN/%s\_%s.hdf5'%(model\_name,embed\_size)

    early\_stopping =EarlyStopping(monitor='val\_loss', patience=3)

    model\_checkpoint = ModelCheckpoint(model\_save\_path, save\_best\_only=True, save\_weights\_only=True)

    if TRAIN and FLAG:

        model.fit(X\_train,X\_train\_label,validation\_data=(X\_val,X\_val\_label),epochs=100,batch\_size=64,shuffle=True,callbacks=[early\_stopping,model\_checkpoint] )

    model.load\_weights(model\_save\_path)

    pred\_val = model.predict(X\_val,batch\_size=128,verbose=1)

    pred\_test = model.predict(test\_seq,batch\_size=128,verbose=1)

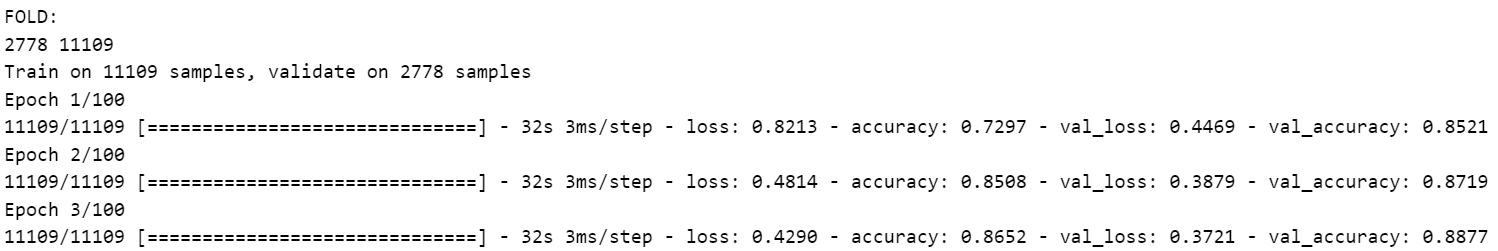
    meta\_train[te\_ind] = pred\_val

    meta\_test += pred\_test

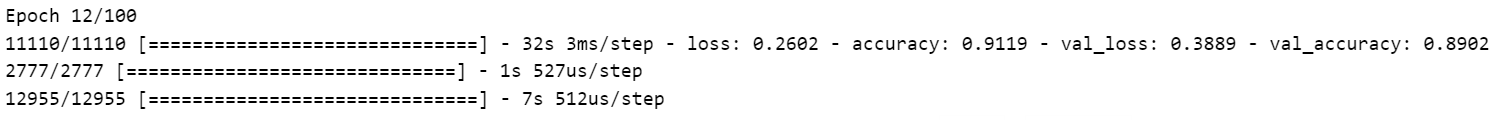
    K.clear\_session()

meta\_test /= 5.0

运行结果



…



#### 结果提交

将模型进行5折交叉验证后得到的预测均值输出，提交结果。

test\_df['prob0'] = 0

test\_df['prob1'] = 0

test\_df['prob2'] = 0

test\_df['prob3'] = 0

test\_df['prob4'] = 0

test\_df['prob5'] = 0

test\_df['prob6'] = 0

test\_df['prob7'] = 0

test\_df[['prob0','prob1','prob2','prob3','prob4','prob5','prob6','prob7']] = meta\_test

test\_df[['file\_id','prob0','prob1','prob2','prob3','prob4','prob5','prob6','prob7']].to\_csv('nn\_baseline\_5fold.csv',index = None)

线上成绩：0.529231

# 系统测试与改进

# 4 System Test and Improve

## 系统测试(System Test)

经阿里云官方测试，以为评分标准，成绩达到0.525310，排名位于331名，达到全国前10%，属于不错的成绩。





将训练好的模型取出，可制作实用程序，通过导入程序的API序列来预测程序在本文中提到的7种类别。

## 改进(Improve)

### TF-IDF

尝试使用TF-IDF获取样本整体各个API调用序列的分布情况。

vectorizer = TfidfVectorizer(ngram\_range=(1, 5), min\_df=3, max\_df=0.9, )

### XGBoost

XGBoost是在LightGBM前最有名的GBDT工具，它基于预排序方法的决策树算法。优点是能精确地找到分割点。

meta\_train = np.zeros(shape=(len(files), 8))

meta\_test = np.zeros(shape=(len(outfiles), 8))

skf = StratifiedKFold(n\_splits=5, random\_state=4, shuffle=True)

for i, (tr\_ind, te\_ind) in enumerate(skf.split(train\_features, labels)):

    X\_train, X\_train\_label = train\_features[tr\_ind], labels[tr\_ind]

    X\_val, X\_val\_label = train\_features[te\_ind], labels[te\_ind]

    print('FOLD: {}'.format(str(i)))

    print(len(te\_ind), len(tr\_ind))

    dtrain = xgb.DMatrix(X\_train, label=X\_train\_label)

    dtest = xgb.DMatrix(X\_val, X\_val\_label)

    dout = xgb.DMatrix(out\_features)

    param = {'max\_depth': 6, 'eta': 0.1, 'eval\_metric': 'mlogloss', 'silent': 1, 'objective': 'multi:softprob',

             'num\_class': 8, 'subsample': 0.8,

             'colsample\_bytree': 0.85}  # 参数

    evallist = [(dtrain, 'train'), (dtest, 'val')]  # 测试 , (dtrain, 'train')

    num\_round = 300  # 循环次数

    bst = xgb.train(param, dtrain, num\_round, evallist, early\_stopping\_rounds=50)

    # dtr = xgb.DMatrix(train\_features)

    pred\_val = bst.predict(dtest)

    pred\_test = bst.predict(dout)

    meta\_train[te\_ind] = pred\_val

    meta\_test += pred\_test

### CNN-LSTM

我们可以尝试使用CNN-LSTM来获取序列的上下文信息。

def CNN\_LSTM:

    main\_input = Input(shape=(maxlen,), dtype='float64')

    embedder = Embedding(304, 256, input\_length=maxlen)

    embed = embedder(main\_input)

    # avg = GlobalAveragePooling1D()(embed)

    # cnn1模块，kernel\_size = 3

    conv1\_1 = Conv1D(64, 3, padding='same', activation='relu')(embed)

    conv1\_2 = Conv1D(64, 3, padding='same', activation='relu')(conv1\_1)

    cnn1 = MaxPool1D(pool\_size=2)(conv1\_2)

    conv1\_1 = Conv1D(64, 3, padding='same', activation='relu')(cnn1)

    conv1\_2 = Conv1D(64, 3, padding='same', activation='relu')(conv1\_1)

    cnn1 = MaxPool1D(pool\_size=2)(conv1\_2)

    conv1\_1 = Conv1D(64, 3, padding='same', activation='relu')(cnn1)

    conv1\_2 = Conv1D(64, 3, padding='same', activation='relu')(conv1\_1)

    cnn1 = MaxPool1D(pool\_size=2)(conv1\_2)

    rl = CuDNNLSTM(256)(cnn1)

    # flat = Flatten()(cnn3)

    # drop = Dropout(0.5)(flat)

    fc = Dense(256)(rl)

    main\_output = Dense(8, activation='softmax')(rl)

    model = Model(inputs=main\_input, outputs=main\_output)

    return model

# 总结与展望

# 5 Conclusions and Prospects

## 总结(Conclusions)

随着互联网技术的普及,恶意软件给网络安全带来了极大的威胁。目前主要的恶意软件检测技术及基于特征码的恶意软件检测,该技术在上世纪被提出之后在恶意软件检测上取得了极大的成功。然而随着技术的发展,恶意软件惊人的速度增长,不断出现新型的恶意软件类型。基于特征码的恶意软件检测技术己经出现了明显的局限性一一不能检测新型的恶意软件,对经过代码混淆技术掩饰的部分恶意软件也无法检测。因此出现了基于软件行为的恶意软件检测技术,分为静态检测和动态检测。

静态检测从代码和指令流的角度对软件的行为进行分析,并不会真正的运行恶意软件,因而难应对越来越精良的代码混淆技术。因此动态分析的恶意软件检测技术应运而生,目前基于Windows API调用行为的恶意软件动态检测成为了研究热点。本文就针对基于Windows API调用行为的恶意软件检测技术进行了深入的研究。

本文首先在国内外专业网站上收集了大量的恶意软件样本,包括病毒、蠕虫、后门、木马、勒索软件等等。本文扩大了对API调用的监测范围。使用专业的API调用监测工具,在虚拟机中安装Windows操作系统,实际运行恶意软件样本和非恶意软件样本,对恶意软件的API调用行为进行监测形成API调用行为日志。形成恶意软件和非恶意软件API调用日志数据集。结合数据挖掘的方式进行分析。

本文首先使用特征选择和特征重构两种方式,从原始日志文件中提取出了API,API+参数,API+参数+参数取值H组特征,使用文档频率和信息增益相结合的方式进行了特征选择,特征细节上层层深入。其中API+参数+参数取值的特征为本文构建。使用筛选出的特征形成用于文本分类的数据,选用朴素贝叶斯、支持向量机(SMO)、决策树(J48)、随机森林四种经典的分类方法进行实验,均取得了较好的分类准确率。随着特征细节的深入,分类的准确率不断提高,特别是在将参数取值加考虑之后,分类准确率得到了极大的提高。

本文也从其他角度进行了探究:API调用频率,API调用之间的关联关系和API+全部参数作为特征。在API调用频率的探究中,本文使用改进的逆文档频率进行特征向量的构造,准确率优于单纯考虑频率的准确率。关联关系的探巧中准确率没有单独考虑API时高,因此API调用么间的关联关系对恶意软件检测贡献不大。使用API+全部参数作为新形式特征,使得特征的数量得到了减少,同时提高了检测的准确率。

本文从多个方面对Windows API调用的行为进行了探究,均取得了很高的恶意软件检测率,因此动态的恶意软件检测方法应该是恶意软件检测研究的一个方向。无论经过怎样精心的代码掩饰,行为始终会暴露它的意图。

## 进一步研究(Further Research)

本文从多方面对软件行为进行了探究，取得了较好的恶意软件检测效果，动态的恶意软件检测技术是一个值得探究的恶意软件检测方向。但是其中也有很多不足之处和很多未做的探究。笔者在今后会尝试整合更多机器学习和深度学习的方法，将降至更低的水平。

# 参考文献

1. McAfeeLabs. McAfee Labs Threats Report, June 2021[R]. America: McAfeeLabs, 2021.
2. Uppal D, Sinha R, Mehra V, et al. Malware detection and classification based on extraction of API sequences[C]
3. Sami A, Yadegari B, Rahimi H, et al. Malware detection based on mining API calls[C]
4. Kephart J O. Automatic extraction of computer virus signatures[C]
5. Egele M, Scholte T, Kirda E, et al. A survey on automated dynamic malware-analysis techniques and tools[J]. ACM computing surveys (CSUR), 2008, 44(2): 1-42.
6. Eskandari M, Khorshidpur Z, Hashemi S. To incorporate sequential dynamic features in malware detection engines[C]
7. Sundarkumar G G, Ravi V. Malware detection by text and data mining[C]
8. Zhuang W, Ye Y, Chen Y, et al. Ensemble clustering for internet security applications[J]. IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews), 2012, 42(6): 1784-1796.
9. Hou S, Chen L, Tas E, et al. Cluster-oriented ensemble classifiers for intelligent malware detection[C]
10. Ali M A M, Maarof M A. Dynamic innate immune system model for malware detection[C]
11. Salehi Z, Ghiasi M, Sami A. A miner for malware detection based on API function calls and their arguments[C]
12. Alazab M, Venkatraman S, Watters P, et al. Zero-day malware detection based on supervised learning algorithms of API call signatures[J]. 2010.
13. 陈志云, 薛质. 基于 Win32 API 调用监控的恶意代码检测技术研究[J]. 信息安全与通信保密, 2009 (7): 73-75.
14. Shankarapani M, Kancherla K, Ramammoorthy S, et al. Kernel machines for malware classification and similarity analysis[C]
15. 白金荣, 王俊峰, 赵宗渠, 等. 基于敏感 Native API 的恶意软件检测方法[J]. 计算机工程, 2012, 38(13): 9-12.
16. Sung A H, Xu J, Chavez P, et al. Static analyzer of vicious executables (save)[C]//20th Annual Computer Security Applications Conference. IEEE, 2004: 326-334.
17. 白金荣, 王俊峰, 赵宗渠. 基于 PE 静态结构特征的恶意软件检测方法[J]. 计算机科学, 2013, 40(1): 122-126.
18. 付文, 赵荣彩, 庞建民, 等. 隐式 API 调用行为的静态检测方法[J]. 计算机工程, 2010, 36(14): 108-110.
19. Bai L, Pang J, Zhang Y, et al. Detecting malicious behavior using critical api-calling graph matching[C]
20. Fu W, Pang J, Zhao R, et al. Static detection of api-calling behavior from malicious binary executables[C]
21. 韩兰胜, 高昆仑, 赵保华, 等. 基于 API 函数及其参数相结合的恶意软件行为检测[J]. 计算机应用研究, 2013, 30(11): 3407-3410.
22. 王长志, 梁刚, 杨进, 等. 基于信息增益特征优化选择的恶意软件检测方法[J]. 计算机安全, 2013 (4): 13-17.
23. Fan C I, Hsiao H W, Chou C H, et al. Malware detection systems based on API log data mining[C]
24. Uppal D, Sinha R, Mehra V, et al. Exploring behavioral aspects of API calls for malware identification and categorization[C]
25. Sundarkumar G G, Ravi V, Nwogu I, et al. Malware detection via API calls, topic models and machine learning[C]
26. Kim Y . Convolutional Neural Networks for Sentence Classification[C]