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已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

Contents

1 绪论

1 Introduction

1.1 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，对于每个已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，能检测的只是特征已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。

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

1.2.1 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，已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。

1.2.2 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]。

1.3 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。

2 相关概念与技术

2 Related Concepts and Technologies

2.1 LightGBM 框架(LightGBM Frame)

2.1.1 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。

2.1.2 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。

2.1.3 工程优化

(1) 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k个类别，则有$2^{k-1}-1$种可能，时间复杂度为$O(2^k)$，LightGBM 基于 Fisher 的《On Grouping For Maximum Homogeneity》论文实现了$O(nlogn)$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，$\frac{Sum(y)}{Count(y)}$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。

(2) 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。

(3) 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。

2.2 CNN 算法(CNN Algorithm)

2.2.1 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。

2.2.2 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：

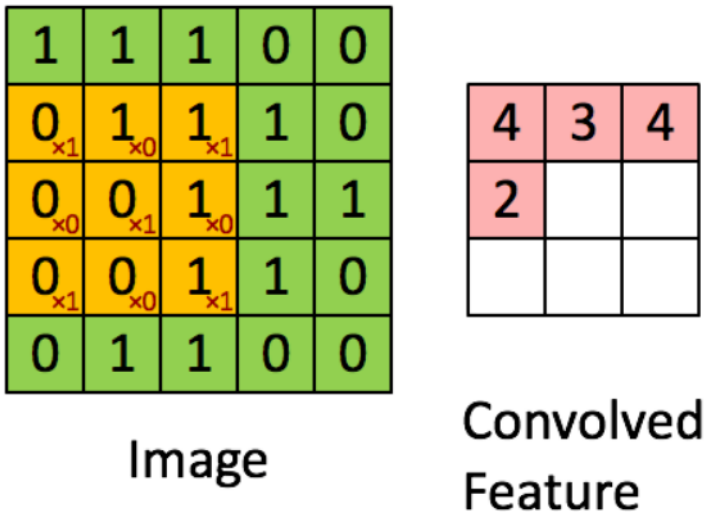


图 2-1 CNN 算法

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：

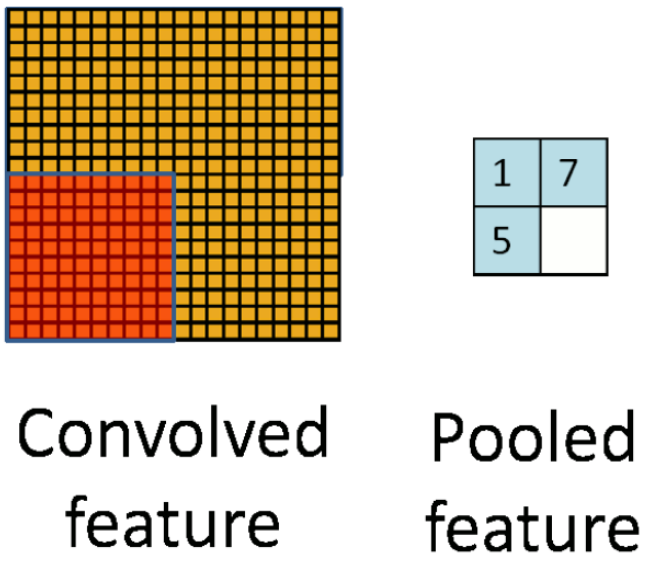


图 2-2 池化操作

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：

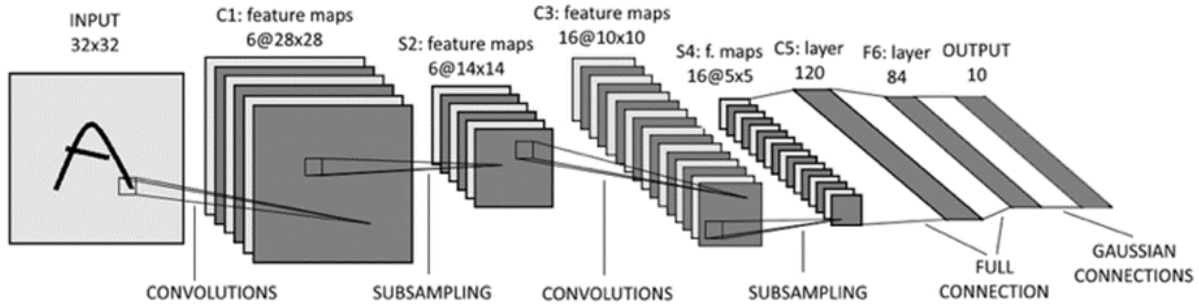


图 2-3 LeNet-5

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。

2.3 TextCNN 算法(TextCNN Algorithm)

2.3.1 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。

2.3.2 基本原理

(1) 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：

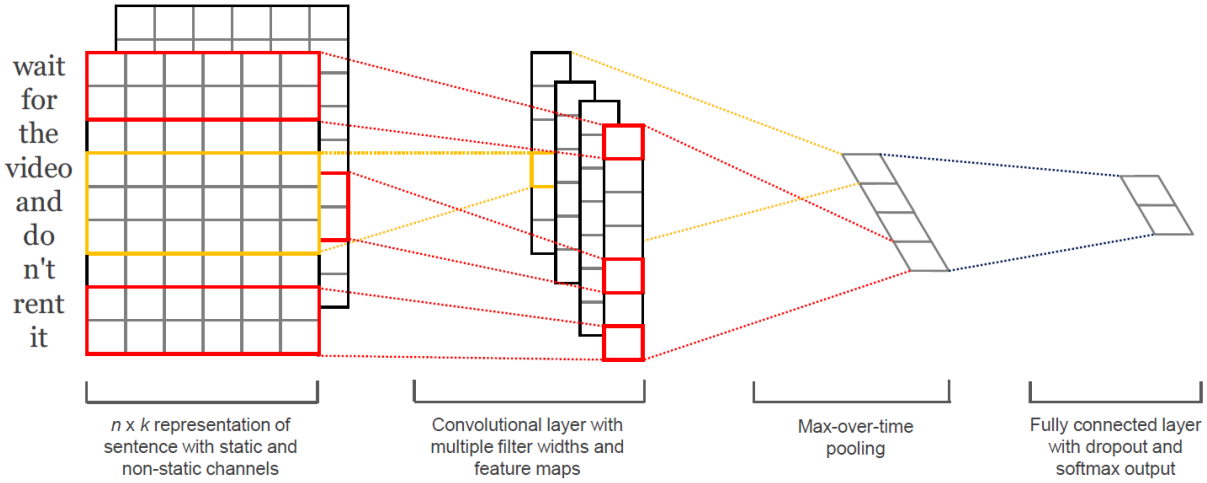


图 2-4 TextCNN 架构

Figure 2-4 TextCNN Frame

(2) 流程

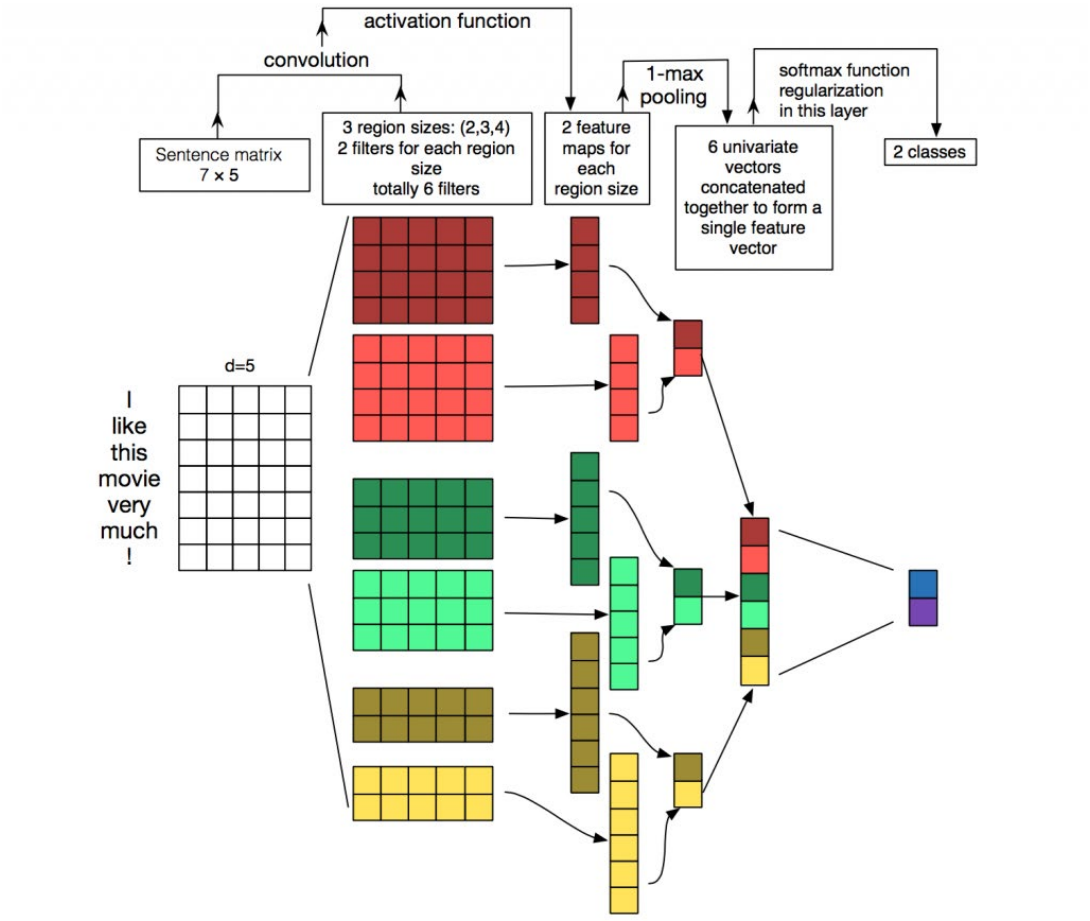


图 2-4 TextCNN 流程

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。

(3) 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。

3 系统设计与实现

3 System Design and Implementation

3.1 赛题理解(Understand the Competition)

3.1.1 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。

3.1.2 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。

3.1.3 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。

(1) 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。

(2) 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。

(3) 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。

(4) 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。

(5) 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。

(6) 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。

(7) 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。

3.1.4 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。

3.2 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。

3.2.1 训练集数据探索

(1) 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()
```

```
<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
```

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()
```

	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

用 DataFrame.describe()函数查看训练集的统计信息：

```
train.describe()
```

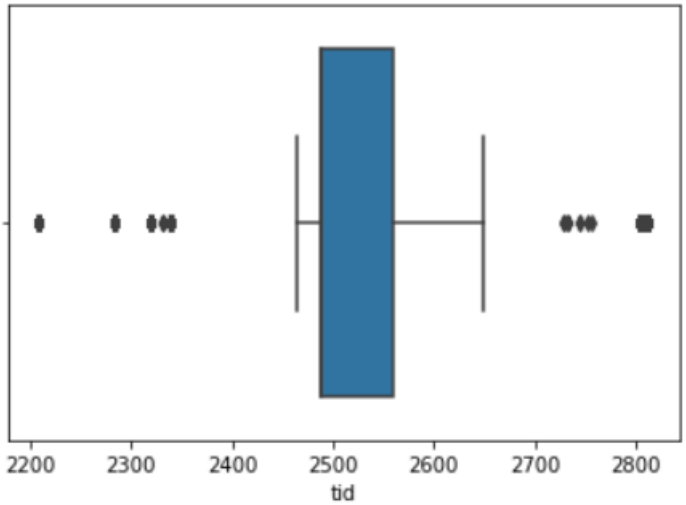
	file_id	label	tid	index
count	8.980669e+07	8.980669e+07	8.980669e+07	8.980669e+07
mean	7.078770e+03	3.862835e+00	2.533028e+03	1.547521e+03
std	3.998794e+03	2.393783e+00	6.995798e+02	1.412249e+03
min	1.000000e+00	0.000000e+00	1.000000e+02	0.000000e+00
25%	3.637000e+03	2.000000e+00	2.356000e+03	3.490000e+02
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

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"])
```

```
<AxesSubplot:xlabel='tid'>
```



用 nunique()函数查看训练集中变量取值的分布：

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

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。

(3) 缺失值

查看训练集数据的缺失情况：

```
train.isnull().sum()
```

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

从运行结果看，数据不存在缺失的情况。

(4) 异常值

分析训练集的“index”特征

```
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
```

“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()
```

```
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
```

“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。

(5) 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()
```

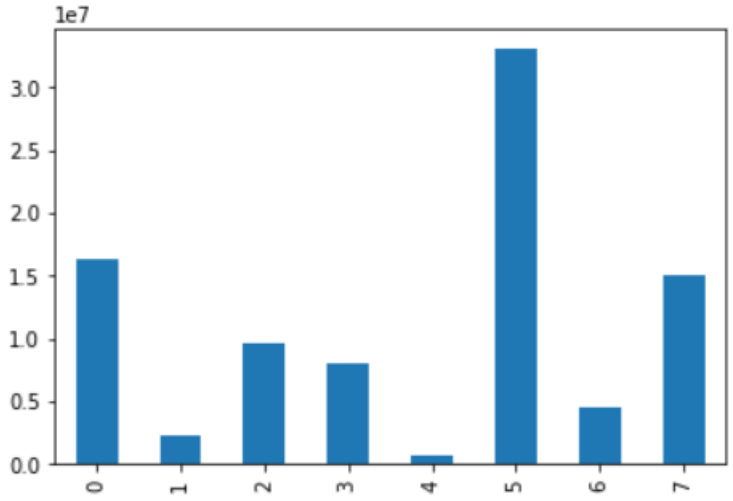
```
5    33033543
0    16375107
7    15081535
2     9693969
3     8117585
6     4586578
1     2254561
4      663815
Name: label, dtype: int64
```

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：

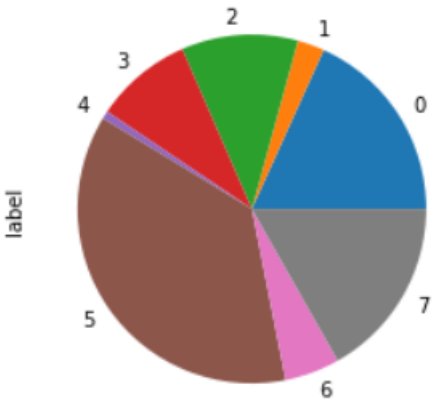
```
train['label'].value_counts().sort_index().plot(kind='bar')
```

```
<AxesSubplot:>
```




```
train['label'].value_counts().sort_index().plot(kind='pie')
```

```
<AxesSubplot:ylabel='label'>
```



3.2.2 测试集数据探索

(1) 数据信息

查看测试集的头几行数据：

```
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

查看测试集的大小、数据类型等信息：

```
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
```

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

(2) 缺失值

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
```

可知，数据不存在缺失的情况。

(3) 数据分布

查看测试集中变量取值的分布：

```
test.nunique()
```

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

由运行结果可知：file_id 有 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。

(4) 异常值

查看测试集的“index”特征：

```
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()
```

```
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，因为这个字段表示的是线程，所以我们目前也没办法判断是否有异常值。

3.2.3 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_fileids = train['file_id'].unique()
test_fileids = test['file_id'].unique()
```

```
len(set(train_fileids) - set(test_fileids))
```

```
932
```

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))
```

```
0
```

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。

(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：

```
train_apis = train['api'].unique()
test_apis = test['api'].unique()
```

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

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

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)
```

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

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。

3.3 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。

3.3.1 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')
```

3.3.2 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)
```

```
CPU times: user 52.7 s, sys: 10.5 s, total: 1min 3s
Wall time: 1min 3s
```

```
%%time
simple_test_fea1 = simple_sts_features(test)
```

```
CPU times: user 42.1 s, sys: 9.21 ms, total: 42.1 s
Wall time: 42.1 s
```

反映 tid、index 等数值特征的统计特征：

```
%%time
simple_train_fea2 = simple_numberical_sts_features(train)
```

```
CPU times: user 5.38 s, sys: 374 ms, total: 5.75 s
Wall time: 5.75 s
```

```
%%time
simple_test_fea2 = simple_numberical_sts_features(test)
```

```
CPU times: user 4.68 s, sys: 417 ms, total: 5.09 s
Wall time: 5.09 s
```

3.3.3 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：

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

```
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)
```

```
fold n°0
Training until validation scores don't improve for 100 rounds
[50]    training's multi_logloss: 1.83717       training's loss is: : 2.41456   valid_1's multi_logloss: 1.28536
[100]   training's multi_logloss: 1.83717       training's loss is: : 2.41456   valid_1's multi_logloss: 1.28536
Early stopping, best iteration is:
[1]     training's multi_logloss: 1.83717       training's loss is: : 2.41456   valid_1's multi_logloss: 1.28536
fold n°1
Training until validation scores don't improve for 100 rounds
[50]    training's multi_logloss: 1.77226       training's loss is: : 2.32838   valid_1's multi_logloss: 2.10695
[100]   training's multi_logloss: 1.77226       training's loss is: : 2.32838   valid_1's multi_logloss: 2.10695
Early stopping, best iteration is:
[1]     training's multi_logloss: 1.77226       training's loss is: : 2.32838   valid_1's multi_logloss: 2.10695

    ...
Training until validation scores don't improve for 100 rounds
[50]    training's multi_logloss: 1.70379       training's loss is: : 2.27265   valid_1's multi_logloss: 2.91788
[100]   training's multi_logloss: 1.70379       training's loss is: : 2.27265   valid_1's multi_logloss: 2.91788
Early stopping, best iteration is:
[1]     training's multi_logloss: 1.70379       training's loss is: : 2.27265   valid_1's multi_logloss: 2.91788
Wall time: 9.94 s
```

3.3.4 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'])
```

3.3.5 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

3.4 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。

3.4.1 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')
```

3.4.2 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。

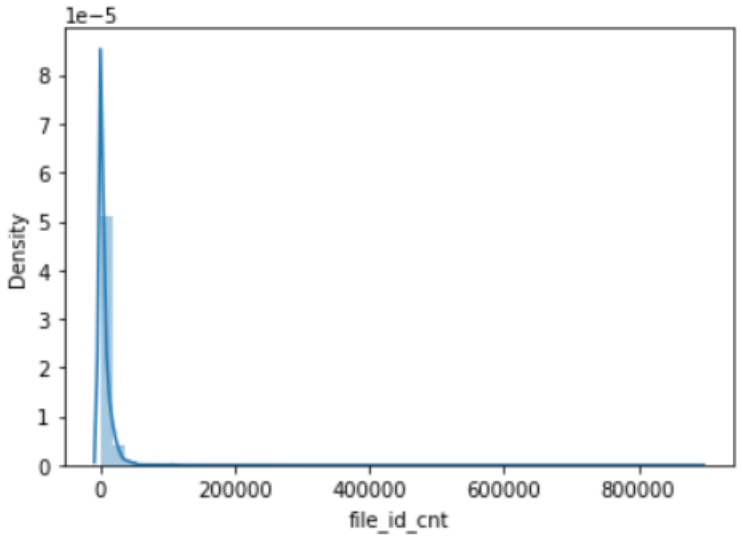
```
    train_analysis = train[['file_id', 'label']].drop_duplicates(subset=['file_id',
'label'], keep='last')
    dic_ = train['file_id'].value_counts().to_dict()
    train_analysis['file_id_cnt'] = train_analysis['file_id'].map(dic_).values
    train_analysis['file_id_cnt'].value_counts()
```

```
5001     448
268      211
44       186
4        160
16       149
         ...
9984       1
14082      1
16131      1
7943       1
16384      1
Name: file_id_cnt, Length: 6204, dtype: int64
```

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'])
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x7f8bd0282850>
```



API 调用次数的 80%都集中在 1000 次以下。

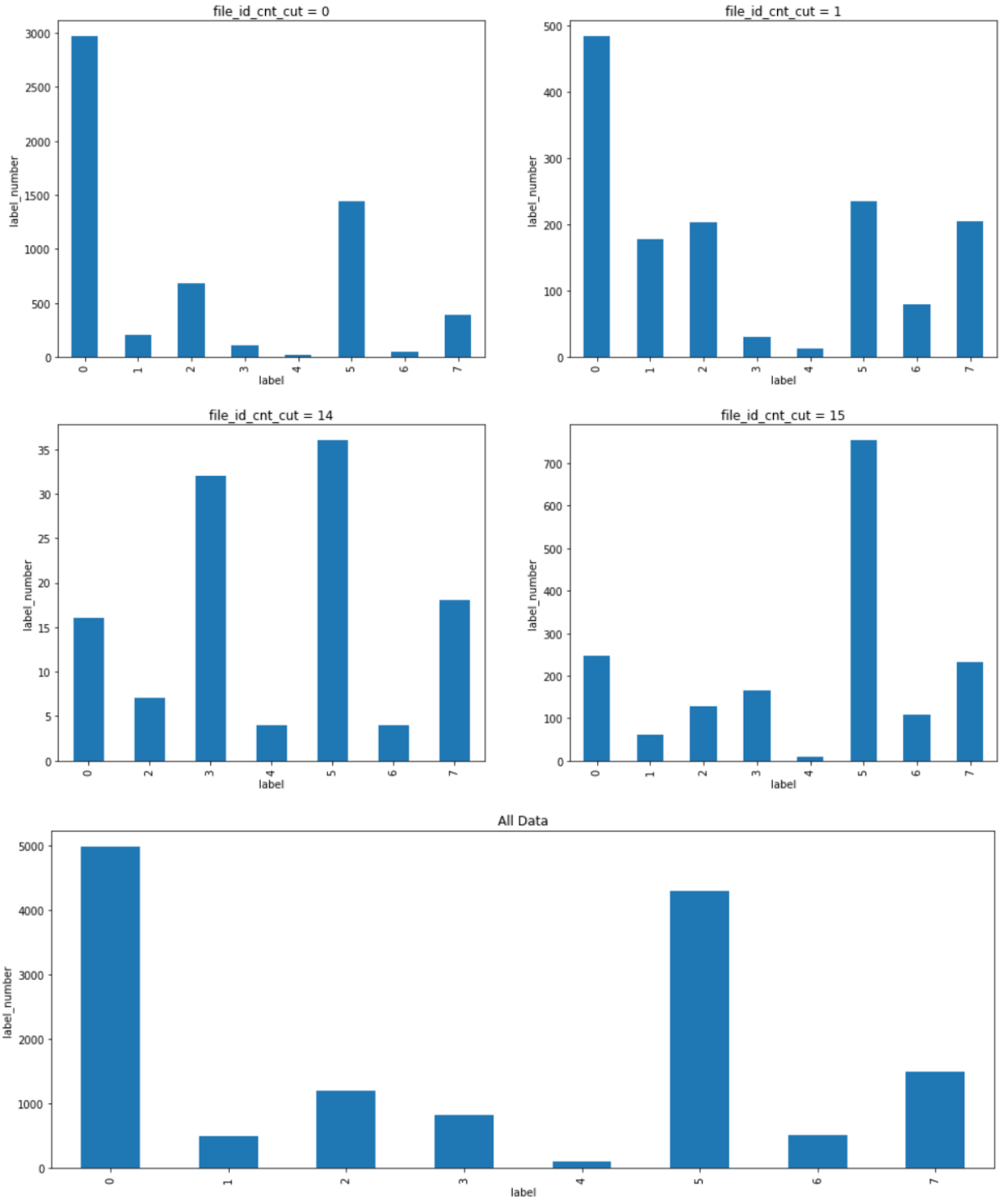
为了便于分析 file_id_cnt 变量和 label 变量之间的关系，首先将数据按 file_id_cnt 变量（即 API 的调用次数）取值划分为 16 个区间。

```
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')
```

```
Text(0, 0.5, 'label_number')
```



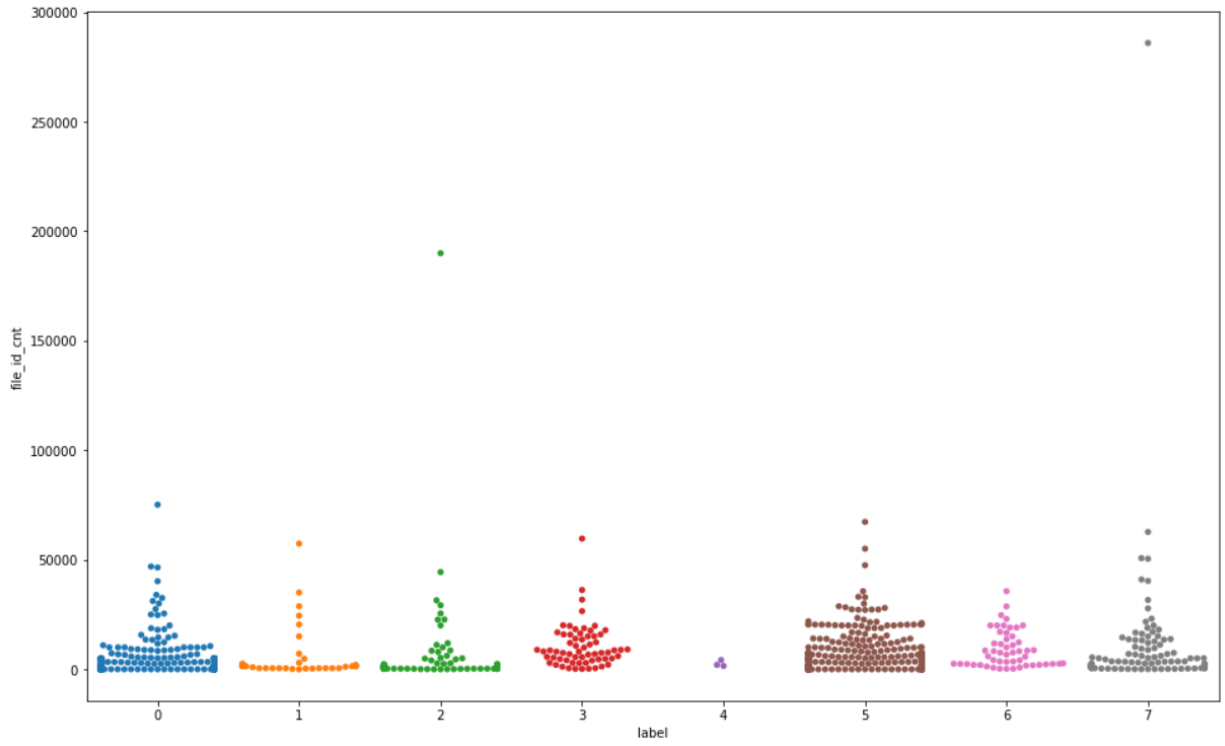
从图中可以看到：当 API 调用次数越多时，该 API 是第五类病毒（感染型病毒）的可能性就越大。

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'])
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x7f8b20116850>
```

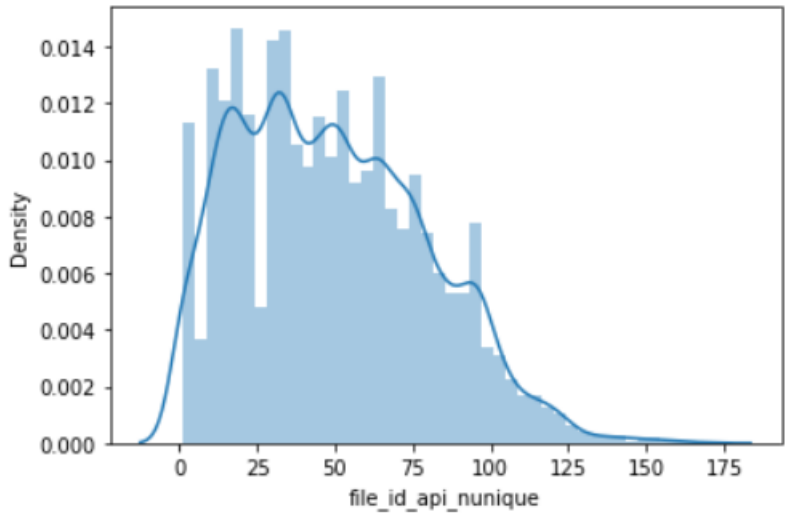


从图中可以得到以下结论：从频次上看，第 5 类病毒调用 API 的次数最多；从调用峰值上看，第 2 类和第 7 类病毒有时能调用 150000 次的 API。

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'])
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x7f8b1bf4a910>
```



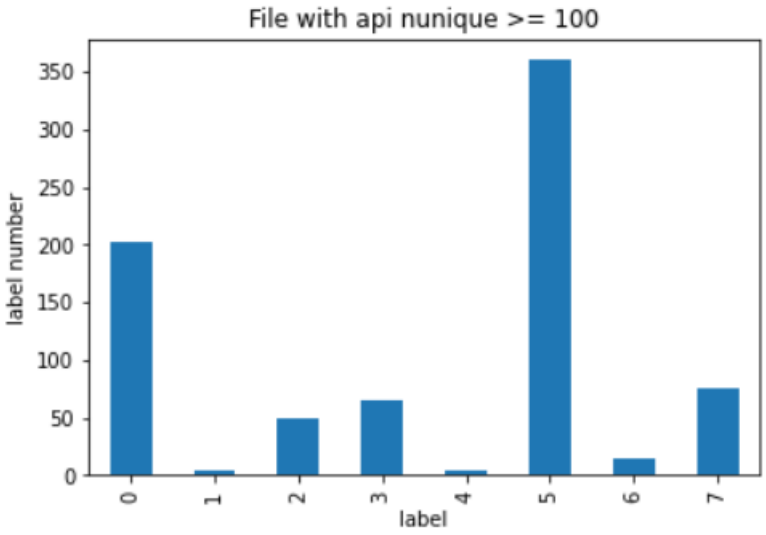

```
train_analysis['file_id_api_nunique'].describe()
```

```
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')
```

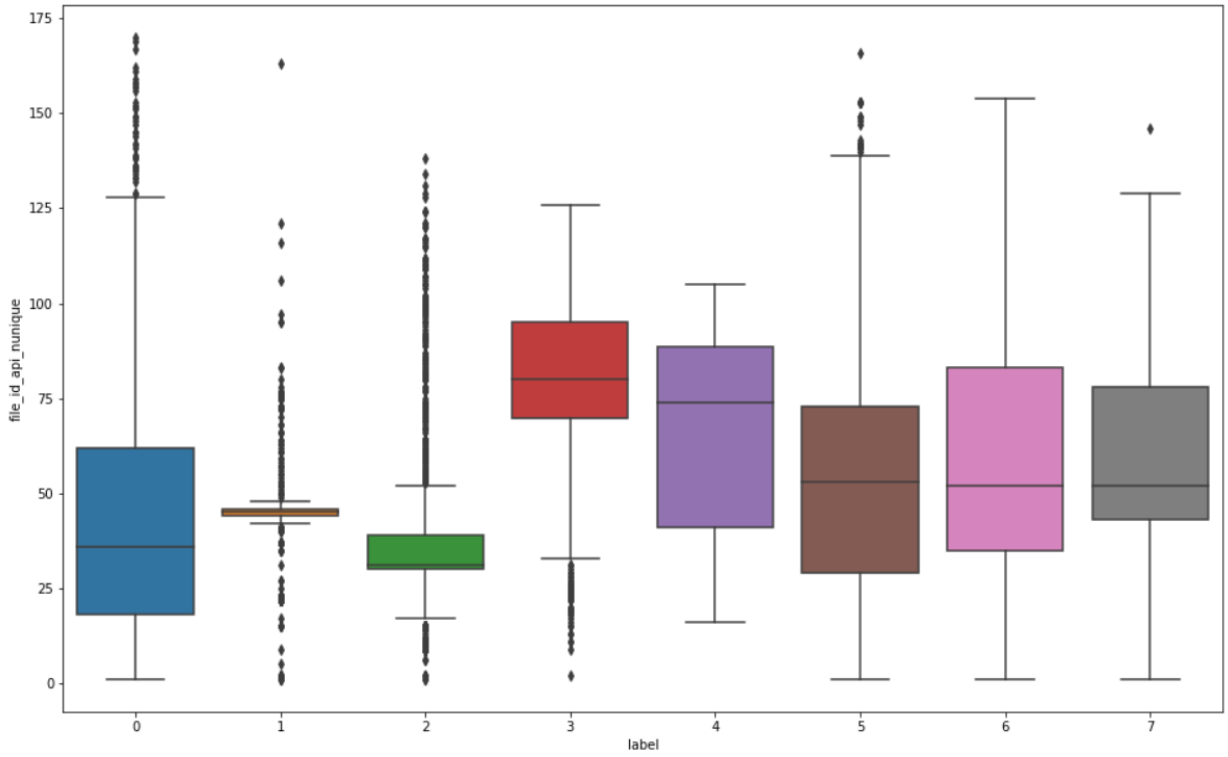
```
Text(0, 0.5, 'label number')
```



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'])
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x7f8b1be73150>
```



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。

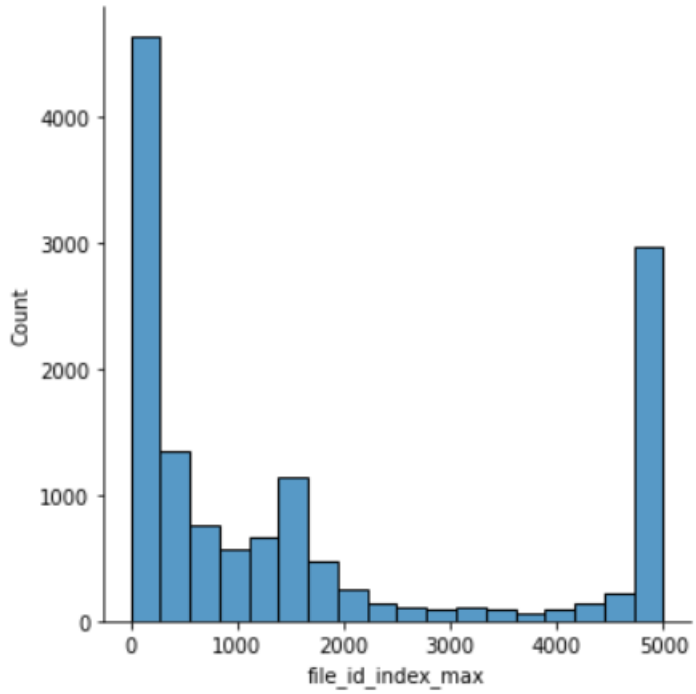
4. 首先，通过 file_id_index_nunique 和 file_id_index_max 两个统计特征，分析变量 file_id 和 index 之间的关系。我们发现调用 API 顺序编号的两个边缘（0 和 5001）的样本数是最多的，因此可以单独看一下这两个点的 label 分布。

```
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()
```

```
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
sns.displot(train_analysis['file_id_index_max'])
```

```
<seaborn.axisgrid.FacetGrid at 0x7f8b2027aad0>
```

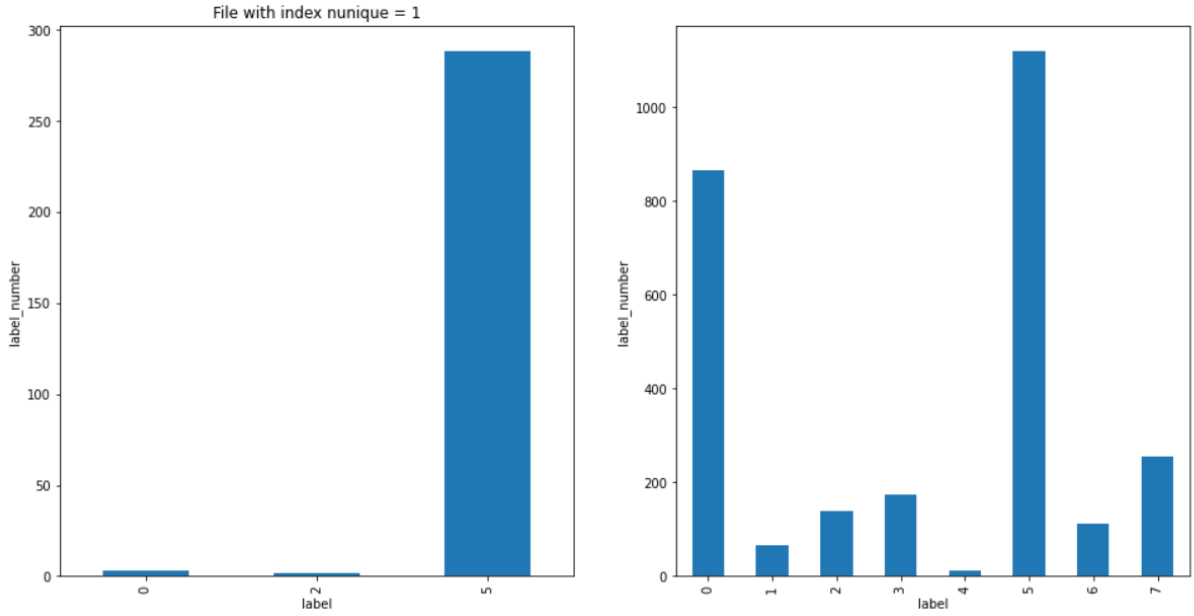


从图中可以看出，文件调用 index 有两个极端，一个是在 1 附近，另一个是在 5000 附近。

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')
```

```
Text(0, 0.5, 'label_number')
```

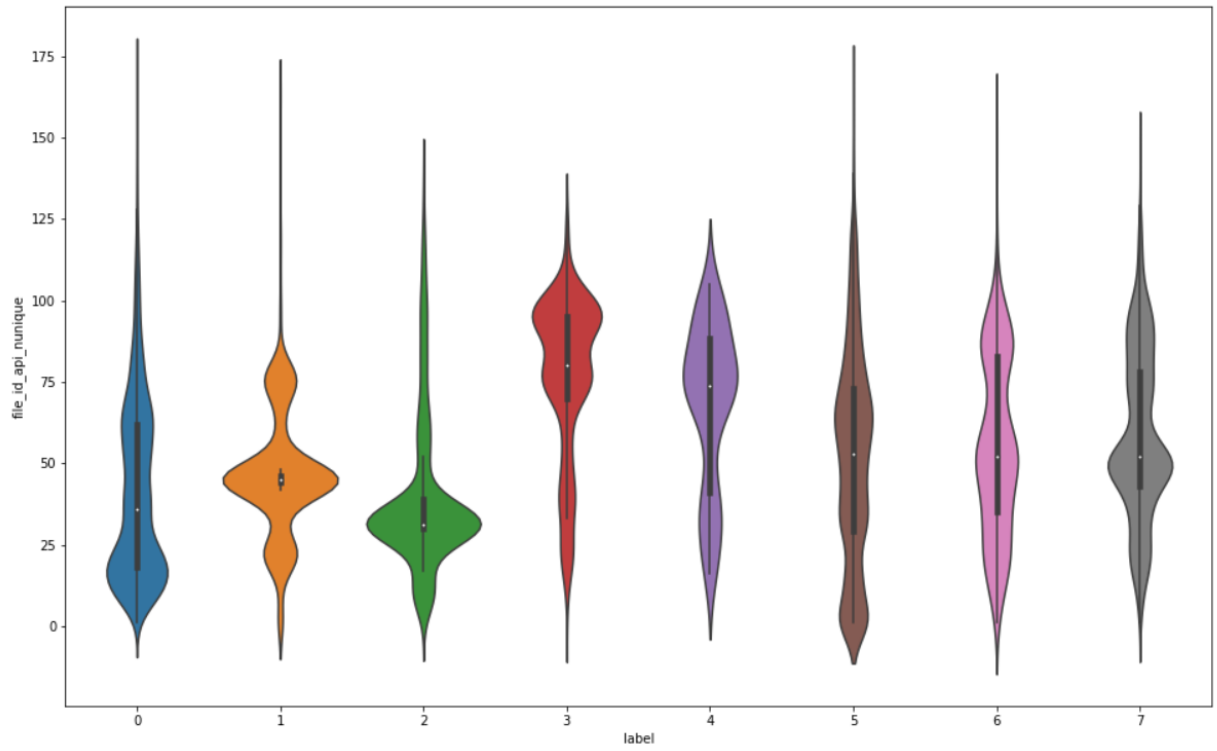


从图中可以发现，在文件顺序编号只有一个时，文件的标签只会是 0（正常）、2（挖矿程序）或 5（感染型病毒），而不会是其他病毒，而且最大概率可能是 5；对于顺序次数大于 5000 个的文件，其和上面调用 API 次数很大时类似。

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：

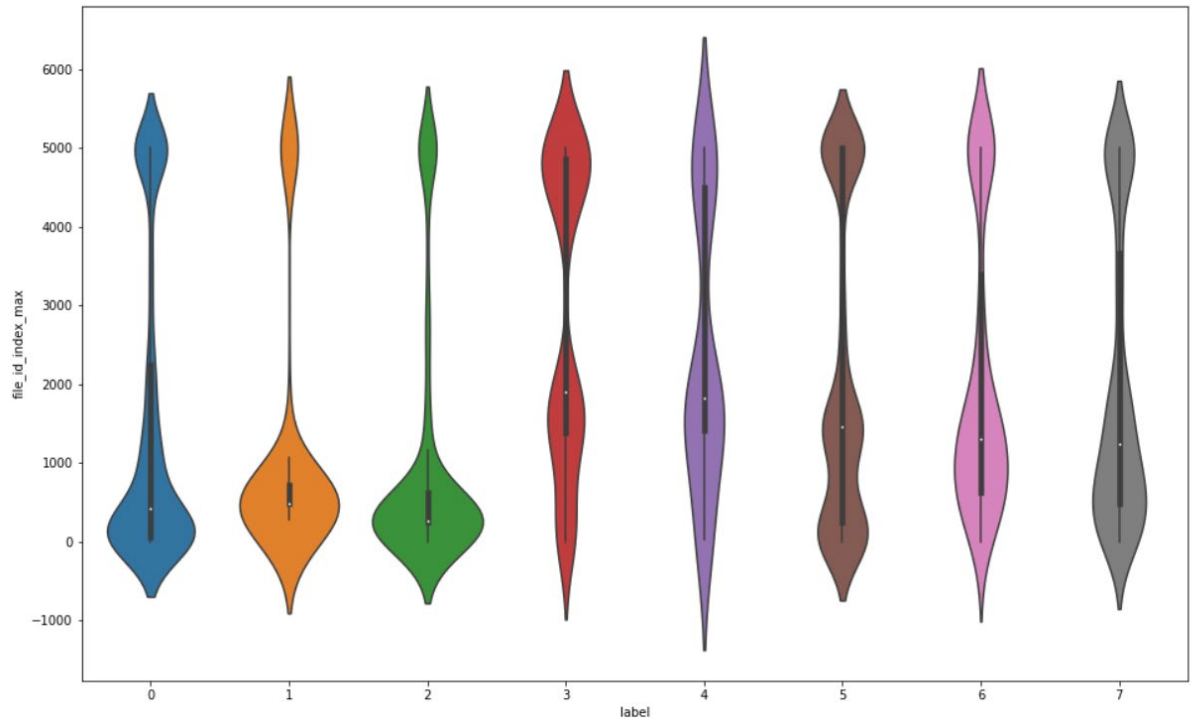
```
    plt.figure(figsize=[16, 10])
    sns.violinplot(x=train_analysis['label'], y=train_analysis['file_id_api_nunique'])
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x7f8b249de850>
```



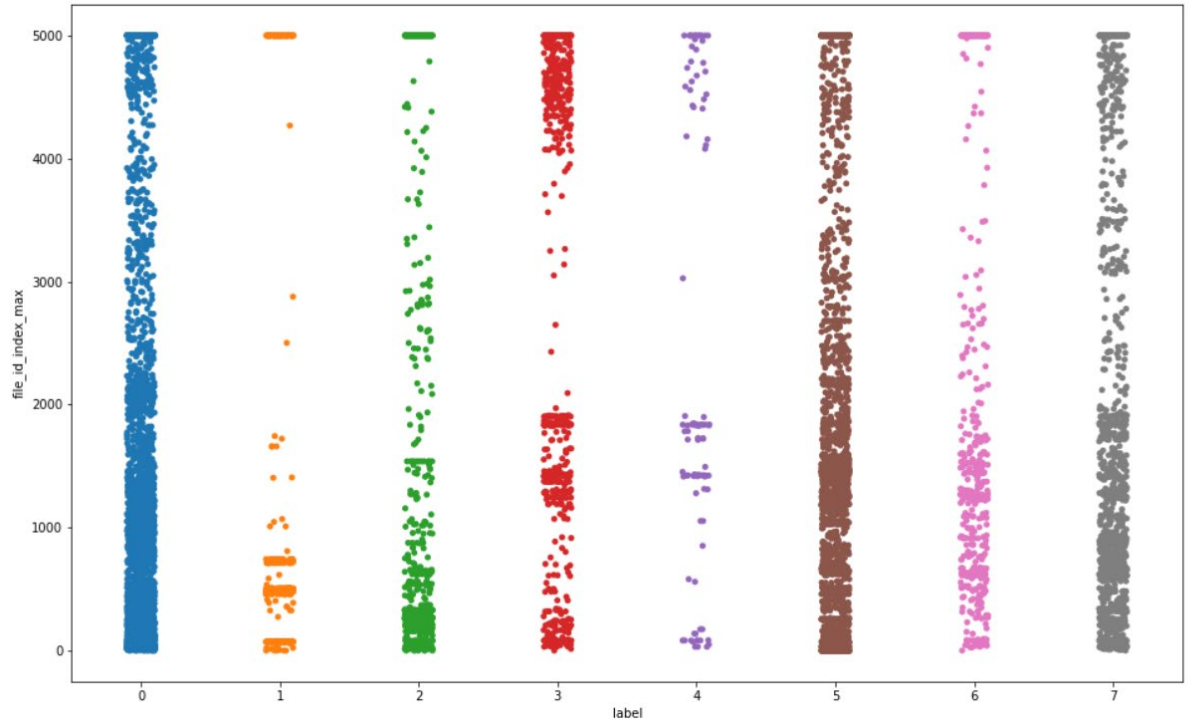

```
plt.figure(figsize=[16, 10])
sns.violinplot(x=train_analysis['label'], y=train_analysis['file_id_index_max'])
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x7f8b1ba48b10>
```




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

```
<matplotlib.axes._subplots.AxesSubplot at 0x7f8b1a18ed50>
```



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()
```

```
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
```

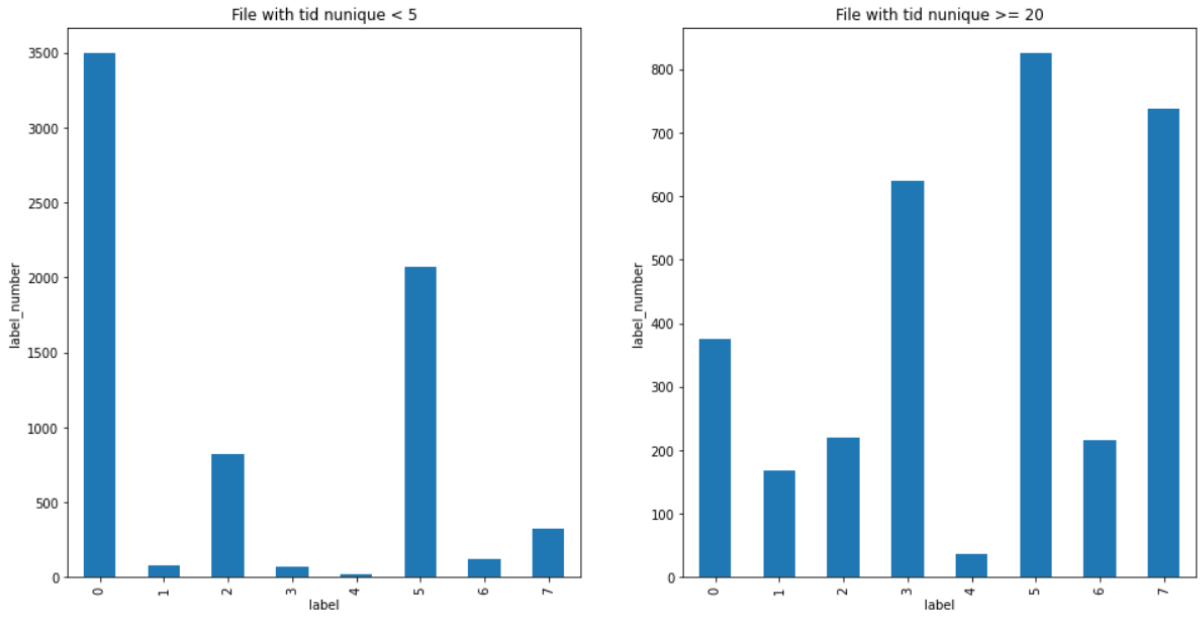
```
dic_ = train.groupby('file_id')['tid'].max().to_dict()
train_analysis['file_id_tid_max'] = train_analysis['file_id'].map(dic_).values
train_analysis['file_id_tid_max'].describe()
```

```
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
```

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().so
rt_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')
```

```
Text(0, 0.5, 'label_number')
```



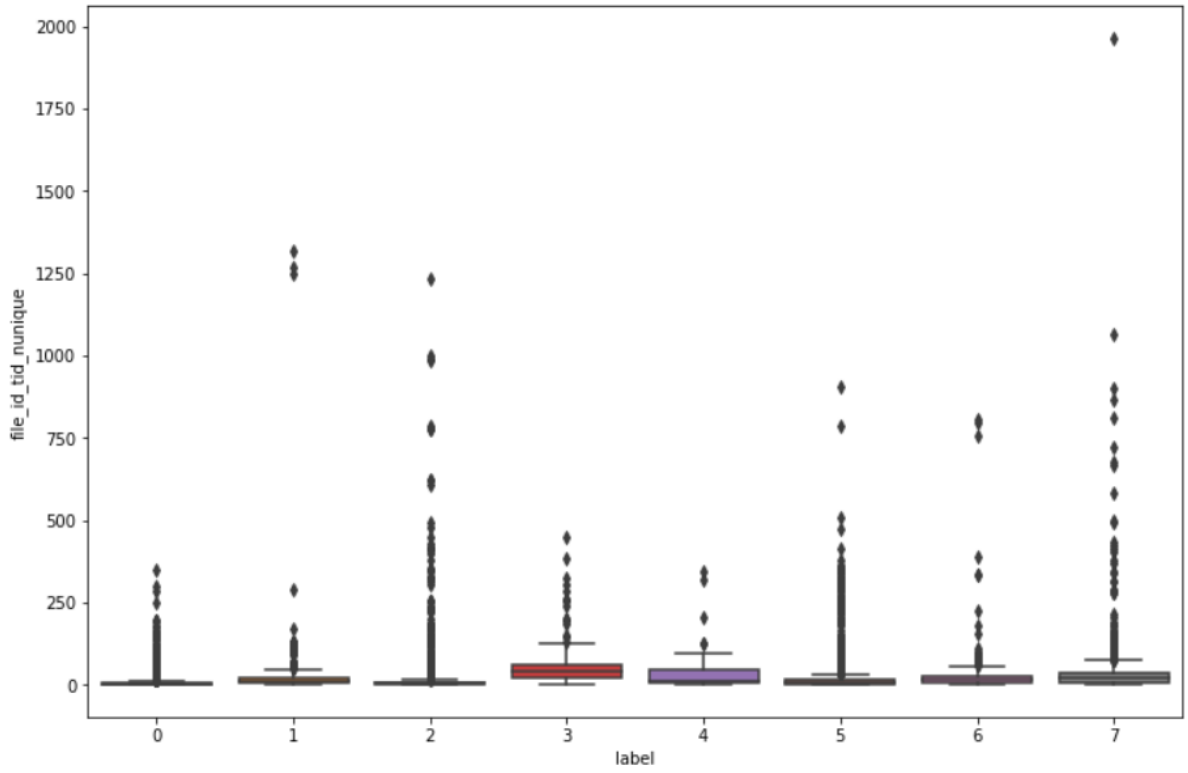
其中，0：正常文件；1：勒索病毒；2：挖矿程序；3：DDoS 木马；4：蠕虫病毒；5：

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。

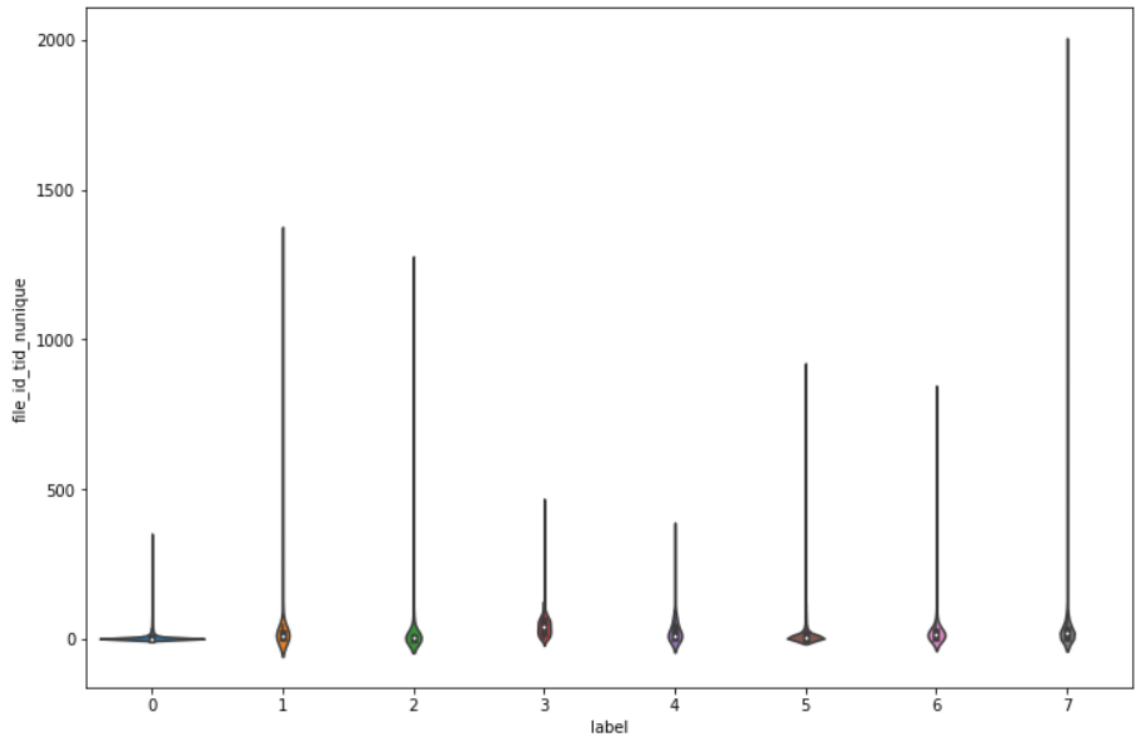
```
plt.figure(figsize=[12, 8])
sns.boxplot(x=train_analysis['label'], y=train_analysis['file_id_tid_nunique'])
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x7f8b1a0fe4d0>
```




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

```
<matplotlib.axes._subplots.AxesSubplot at 0x7f8b1baf2f50>
```



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().so
rt_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)
```

```
********** 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
***********************************************
```

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。

3.5 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。

3.5.1 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)
```

3.5.2 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').re
set_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 i
n 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 i
n 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)
```

3.5.3 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_da
ta.iloc[trn_idx][train_label].values)
            val_data = lgb.Dataset(train_data.iloc[val_idx][train_features], label=train_da
ta.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)
```

```
fold n°0
Training until validation scores don't improve for 100 rounds
[50]    training's multi_logloss: 1.80505       training's loss is: : 2.46982   valid_1's multi_logloss: 2.05998
[100]   training's multi_logloss: 1.80505       training's loss is: : 2.46982   valid_1's multi_logloss: 2.05998
Early stopping, best iteration is:
[1]     training's multi_logloss: 1.80505       training's loss is: : 2.46982   valid_1's multi_logloss: 2.05998
fold n°1
Training until validation scores don't improve for 100 rounds
[50]    training's multi_logloss: 1.81909       training's loss is: : 2.37634   valid_1's multi_logloss: 1.77992
[100]   training's multi_logloss: 1.81909       training's loss is: : 2.37634   valid_1's multi_logloss: 1.77992
Early stopping, best iteration is:
[1]     training's multi_logloss: 1.81909       training's loss is: : 2.37634   valid_1's multi_logloss: 1.77992

    ...
fold n°4
Training until validation scores don't improve for 100 rounds
[50]    training's multi_logloss: 1.82105       training's loss is: : 2.40387   valid_1's multi_logloss: 1.75563
[100]   training's multi_logloss: 1.82105       training's loss is: : 2.40387   valid_1's multi_logloss: 1.75563
Early stopping, best iteration is:
[1]     training's multi_logloss: 1.82105       training's loss is: : 2.40387   valid_1's multi_logloss: 1.75563
Wall time: 6.81 s
```

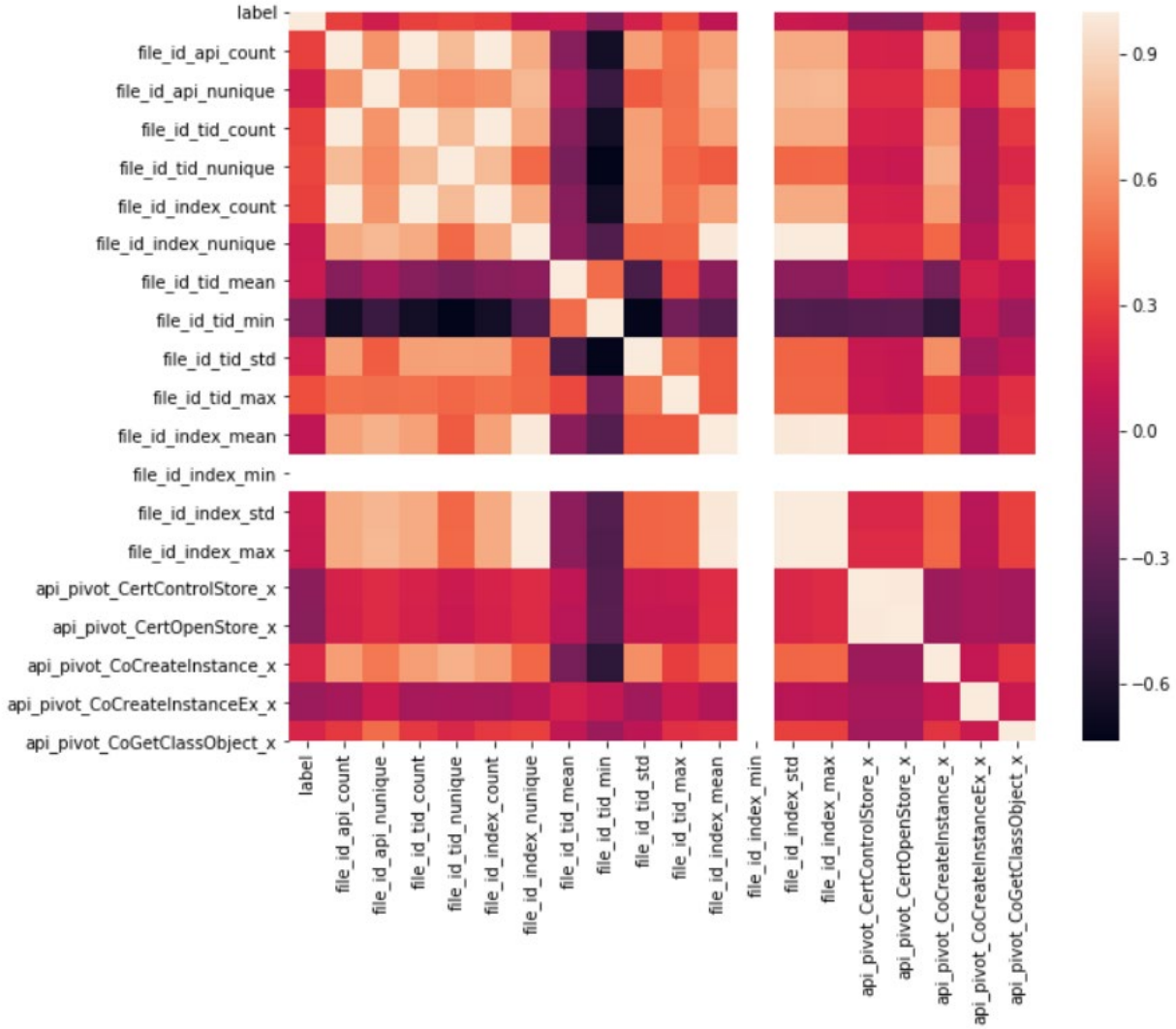
3.5.4 模型结果分析

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())
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x1342e56fb48>
```



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)
```

	fea_name	fea_imp
0	file_id_api_count	0
14	api_pivot_CertControlStore_x	0
3	file_id_tid_nunique	0
4	file_id_index_count	0
5	file_id_index_nunique	0
...	...	...
473	api_pivot_select_y	0
474	api_pivot_send_y	0
475	api_pivot_setsockopt_y	0
476	api_pivot_socket_y	0
477	api_pivot_timeGetTime_y	0

478 rows × 2 columns

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。

3.5.5 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

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

3.6.1 Python 处理大数据

(1) 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(afterpro
cess_memory))
                return df

    memory_process = _Data_Preprocess()
```

(2) 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。

(3) 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。

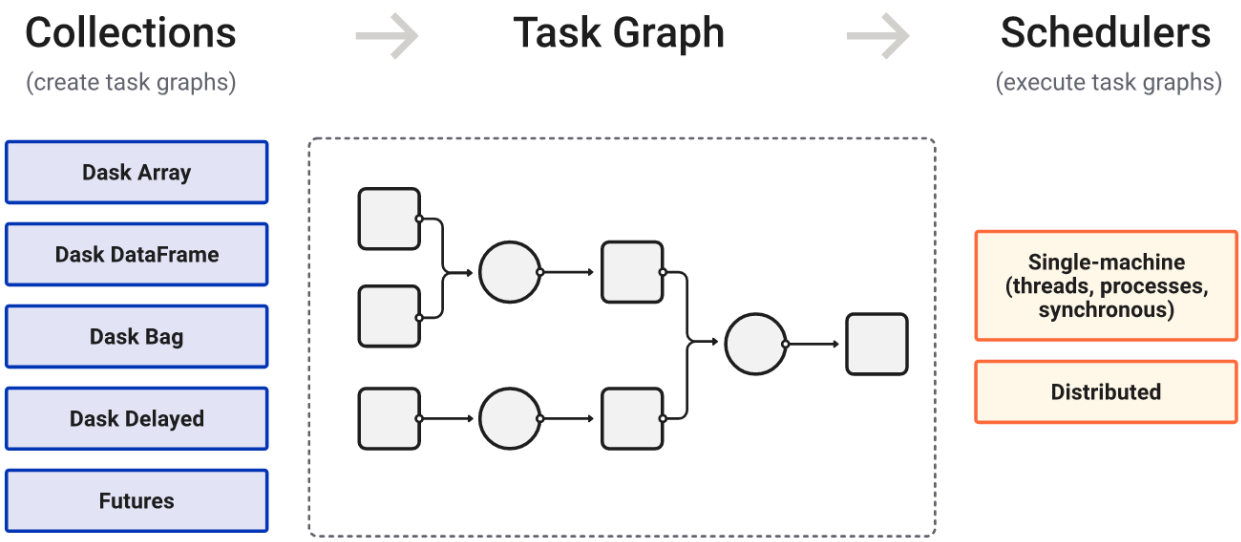
原生：在纯 Python 中启用分布式计算，并访问 PyData 堆栈。

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，总时间 = 编译时间 + 运行时间，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。

3.6.2 深度学习解决方案——TextCNN

(1) 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。

(2) 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(afterpro
cess_memory))
```

```
            return df

memory_process = _Data_Preprocess()
```

(3) 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)
```

(4) 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_zer
o):

          _input = Input(shape=(max_len,), dtype='int32')
          _embed = Embedding(max_cnt, embed_size, input_length=max_len, mask_zero=mask_ze
ro)(_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
```

(5) 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_w
eights_only=True)
      if TRAIN and FLAG:
            model.fit(X_train,X_train_label,validation_data=(X_val,X_val_label),epoch
s=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

```
FOLD:
2778 11109
Train on 11109 samples, validate on 2778 samples
Epoch 1/100
11109/11109 [==============================] - 32s 3ms/step - loss: 0.8213 - accuracy: 0.7297 - val_loss: 0.4469 - val_accuracy: 0.8521
Epoch 2/100
11109/11109 [==============================] - 32s 3ms/step - loss: 0.4814 - accuracy: 0.8508 - val_loss: 0.3879 - val_accuracy: 0.8719
Epoch 3/100
11109/11109 [==============================] - 32s 3ms/step - loss: 0.4290 - accuracy: 0.8652 - val_loss: 0.3721 - val_accuracy: 0.8877

    ...

Epoch 12/100
11110/11110 [==============================] - 32s 3ms/step - loss: 0.2602 - accuracy: 0.9119 - val_loss: 0.3889 - val_accuracy: 0.8902
2777/2777 [==============================] - 1s 527us/step
12955/12955 [==============================] - 7s 512us/step
```

(6) 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_tes
t
    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

4 系统测试与改进

4 System Test and Improve

4.1 系统测试(System Test)

经阿里云官方测试，以*logloss*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。

330	BBTDXSS	XSS	0.524308	2020-03-22
331	Aoi达西	中国矿业大学	0.525310	2022-06-14
332 ↓1	xinruzhishui_0802	兰州市救灾物资储备中心	0.526404	2021-07-25
333 ↓1	逍遥风影李	西北工业大学	0.527541	2019-06-21
334 ↓1	卓著功勋	浙江工业大学	0.527950	2021-05-06
335 ↓1	manassehzhou	北京邮电大学	0.527964	2019-11-09
336 ↓1	锦夏工作室	无	0.528702	2019-06-16
337 ↓1	jiajiao1	新仁公司	0.528932	2019-06-13
338 ↓1	[退役]老师社会	南京大学	0.529231	2019-03-03
339 ↓1	小汇汇	电子科技大学	0.529255	2019-02-22
340 ↓1	LimxMia	中国科学技术大学	0.529378	2020-01-14

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。

4.2 改进(Improve)

4.2.1 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, )
```

4.2.2 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
```

4.2.3 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
```

5 总结与展望

5 Conclusions and Prospects

5.1 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已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。

5.2 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$logloss$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]