**上海电力大学**

**本科毕业设计（论文）**



题 目： 基于集成学习的恶意PE软件特征检测与识别

院 系： 计算机科学与技术学院

专业年级： 信息安全2021级

学生姓名： 郑蜀杭 学号： 20210521

指导教师： 陈思远

2025 年4 月 20日

上海电力大学

本科毕业设计（论文）原创性声明

本人郑重申明：本人所呈交的毕业论文，是在指导老师的指导下独立进行研究所取得的成果。论文中凡引用他人已经发布或未发表的成果、数据、观点等，均已明确注明出处。论文中除已经注明引用的内容外，不包含任何其他个人或集体已经发表或撰写过的研究成果。对本文的研究成果做出重要贡献的个人和集体，均已在论文中以明确的方式标明。

本声明的法律责任由本人承担。



论文作者签名： 日期：2025年4月25日

上海电力大学

本科毕业设计（论文）使用授权声明

本人在指导老师的指导下所完成的论文及相关的资料，知识产权归属上海电力大学。本人完全了解上海电力大学有关保存、使用毕业论文的规定，同意学校保存或向国家有关部门或机构送交论文的纸质版或电子版，允许论文被查阅或借阅。本人授权上海电力大学可以将本毕业论文的全部或部分内容编入有关数据库进行检索，可以采用任何复制手段保存或编汇本毕业论文。如果发表相关成果，一定征得指导教师同意，且第一署名单位为上海电力大学。本人毕业后使用毕业论文或与该论文直接相关的学术论文或成果时，第一署名单位仍然为上海电力大学。



论文作者签名： 日期：2025年5月28日

指导教师签名： 日期：2025年5月28日

摘 要

本文提出了一种新的基于编码压缩的JPEG图像可逆数据隐藏方案。在用流密码对未压缩图像的全部数据进行加密后，可以通过修改一小部分加密数据将附加数据嵌入到图像中。对于包含附加数据的加密图像，可以首先使用加密密钥对其进行解密，解密后的版本与原始图像相似。根据数据隐藏密钥，借助自然图像中的空间相关性，可以成功提取嵌入数据，完美恢复原始图像。

关键词：图像加密；图像恢复；可逆信息隐藏

Reversible Information Hiding in JPEG Images Based on Coded Compression

Abstract

This work proposes a novel reversible data hiding scheme for encrypted image. After encrypting the entire data of an uncompressed image by a stream cipher, the additional data can be embedded into the image by modifying a small proportion of encrypted data. With an encrypted image containing additional data, one may firstly decrypt it using the encryption key, and the decrypted version is similar to the original image. According to the data-hiding key, with the aid of spatial correlation in natural image, the embedded data can be successfully extracted and the original image can be perfectly recovered.

**Key Words**: Image encryption, image recovery, reversible data hiding.

目录

[第 1 章 绪论 1](#_Toc194881812)

[1.1 研究背景与意义 1](#_Toc194881813)

[1.2 相关工作 2](#_Toc194881814)

[1.3 研究内容及创新点 4](#_Toc194881815)

[第 2 章 样本检测关键技术 5](#_Toc194881816)

[2.1 PE恶意样本常见特征 5](#_Toc194881817)

[2.1.1 DOS头及PE 文件头 5](#_Toc194881818)

[2.1.2 节表及节区数据 5](#_Toc194881819)

[2.1.3 关键数据结构 5](#_Toc194881820)

[2.1.4 其他重要数据 5](#_Toc194881821)

[2.2 机器学习算法选择 5](#_Toc194881822)

[2.2.1 常见机器学习算法比较 5](#_Toc194881823)

[2.2.2 经典机器学习与深度学习的选择 5](#_Toc194881824)

[2.3 集成学习技术 6](#_Toc194881825)

[2.3.1 Bagging 6](#_Toc194881826)

[2.3.2 Boosting 6](#_Toc194881827)

[2.3.3 Stacking 6](#_Toc194881828)

[2.3.4 基模型与元模型 6](#_Toc194881829)

[2.3.5 加权融合 6](#_Toc194881830)

[2.4 特征哈希技术 6](#_Toc194881831)

[第 3 章 模型架构 7](#_Toc194881832)

[3.1 三层模型集成架构 7](#_Toc194881833)

[3.1.1 基模型一层 8](#_Toc194881834)

[3.1.2 基模型二层 8](#_Toc194881835)

[3.1.3 元模型决策层 8](#_Toc194881836)

[3.2 直方图特征模型 8](#_Toc194881837)

[3.2.1 字节直方图 8](#_Toc194881838)

[3.2.2 字节熵直方图 8](#_Toc194881839)

[3.2.3 卷积神经网络构建 9](#_Toc194881840)

[3.3 PE静态特征模型 9](#_Toc194881841)

[3.3.1 LIEF解析库 10](#_Toc194881842)

[3.3.2 特征哈希结构化处理 10](#_Toc194881843)

[3.3.3 9基模型堆叠 10](#_Toc194881844)

[3.3.4 随机森林二层基模型构建 10](#_Toc194881845)

[3.4 特征工程模型 10](#_Toc194881846)

[3.4.1 Section信息 11](#_Toc194881847)

[3.4.2 字符匹配 11](#_Toc194881848)

[3.4.3 Yara匹配 12](#_Toc194881849)

[3.4.4 Opcode 12](#_Toc194881850)

[3.4.5 其他布尔信息 13](#_Toc194881851)

[3.4.6 LightGBM模型构建 14](#_Toc194881852)

[第 4 章 实验结果 15](#_Toc194881853)

[4.1 数据集 15](#_Toc194881854)

[4.2 基模型一层集成提升 16](#_Toc194881855)

[4.3 基模型二层集成提升 17](#_Toc194881856)

[4.4 元模型加权融合提升 18](#_Toc194881857)

[4.5 模型评估 19](#_Toc194881858)

[第 5 章 总结与展望 22](#_Toc194881859)

[致谢 23](#_Toc194881860)

[参考文献 24](#_Toc194881861)

# 绪论

## 研究背景与意义

恶意软件作为一种具有恶意目的的软件程序，旨在非法入侵并危害计算机系统、盗取个人敏感信息、破坏隐私数据，甚至影响计算机和网络的正常运行.恶意软件不仅攻击手段复杂多样，还具备快速衍生变种的能力，致使其逐渐成为影响现代网络信息安全的主要威胁之一。近年来，恶意软件攻击频率和规模在全球范围内出现持续增长的趋势。同时，恶意混淆技术的不断发展和自动化代码编写工具的广泛应用，使得恶意软件新变种在数量上呈指数级增长，给现有的防御机制带来严峻挑战。AV-TEST [1]最新统计报告显示, 每天约有 35 万多个新型恶意软件和潜在有害的应用程序被注册。到 2020 年 2 月 9 日, 恶意软件总量超过 1021.19 百万, 其中仅 2019年新生成的恶意软件总量就接近 144.9 百万。目前，恶意软件总量在此时期内逐年递增，且年增长率整体保持在较高水平。因此,数量快速增长的恶意软件仍然是当前严重影响网络信息安全的一个重要因素,如何高效检测并清除恶意软件新变种，已成为网络安全领域函需解决的热点问题。

常见的针对恶意软件的检测分类方法主要包括静态分析方法、动态分析方法、基于机器学习的检测分类方法等静态分析方法是一种在不运行恶意软件样本的情况下对代码或二进制文件进行分析的检测方法.该方法虽然无法提供关于二进制源文件的所有相关信息，但能够快速获取恶意软件样本中的关键结构属性并完成检测分类，但该方法的缺陷在于容易受到例如加密、代码变形和API调用等混淆技术的干扰。动态分析方法是一种在受控环境下运行恶意软件样本，并利用预先安装的进程监控行为特征的检测方法.由于提取动态特征环节需要投入大量资源，因此该方法需要较高的时间和人工成本以基于机器学习的检测方法可以通过自动提取恶意软件样本特征信息实现高效检测，但该方法高度依赖人工提取特征和训练数据的质量，对应用复杂混淆和加壳技术的恶意软件检测效果较差。随着反检测技术和混淆技术的发展，恶意软件新变种不仅可以有效规避常见检测分类方法的识别还能通过插入余代码和改变程序结构等手段掩盖真实意。通过多个渠道在网络中进行传播并攻击目标用户。

在此背景下，通过集成学习的方式，利用多种机器学习模型对恶意软件进行特征提取与分类的检测方法，已成为应对恶意软件混淆技术的有效手段之一。与传统检测方法相比，基于集成学习的检测方法，在多个方面表现出显著优势。首先，该方法能够通过多种机器学习、神经网络模型分析提取恶意样本中的特征，能够更准确地识别分类恶意软件，无需领域专家对特征进行手动定义，极大降低了对时间和人力资源的依赖，从而显著提高检测过程的自动化程度。其次，该方法具有较强的适应性和泛化能力,在面对高度混淆、加密的恶意软件或未知攻击场景时基于集成学习的检测方法仍能保持较高的检测准确率。

## 相关工作

目前针对恶意软件分析的方法有基于特征匹配技术检测方法、基于机器学习技术检测方法、基于集成学习技术检测方法等。特征匹配技术是一种快速识别恶意软件的技术。首先, 检测引擎对待检测的可执行文件进行相关分析，并获得该文件的特征码。然后, 该特征码与检测引擎检索库中的特征集进行匹配, 如果发生匹配项, 则该检测引擎对该检测文件发出恶意警报。若未发现匹配项, 则对该文件进一步分析。如果发现该文件为恶意软件, 则将该文件的特征码加入检索库以对特征集更新。其中特征码是唯一代表文件的序列, 该序列可以是文件的哈希值, 字节或指令, 而传统的恶意软件检测引擎几乎都是依赖特征匹配技术。因此, 当恶意软件随着时间不改变自己的特征标签时,商业检测引擎是可以准确的识别该软件。但是, 当恶意软件制造者对恶意软件进行一些简单的混淆变化,例如增加无效代码, 改变代码结构等, 该文件的特征码就会随之改变。即使简单混淆操作并没有改变恶意软件的功能, 还是能够绕过检测引擎的检测[2]。Christodorescu 等人[3]认为传统的模式匹配方式只考虑了语法的信息匹配, 而忽略了指令的语义信息。于是, 提出了一种合并指令语义的措施来改进匹配策略。该方法将恶意软件的恶意行为形式化表示, 因此当某些恶意行为以不同的形式出现时, 它就可以被发现。上述方式可以抵御黑客使用的常见混淆技术。后来, Christodorescu 和 Jha 等人[3] 认为想要检测混淆的恶意软件, 检测引擎必须首先消除恶意软件作者所使用的混淆变换。因此设计了一种检测可执行文件中恶意模式的体系结构, 该体系结构在进行匹配前可以抵消常见的混淆转换。后来, 研究者们认为提取有价值的特征来对恶意样本生成特征标签是非常重要的, 于是 Zhang 和 Reeves [4]提出了一种利用恶意软件所执行的系统调用特征来实现恶意软件检测。他们认为系统调用可以表征代码片段的语义或功能, 同时还考虑了为系统调用所需要的参数指令。恶意软件检测者利用这一弱点将恶意软件检测看成一个分类问题, 它可以学习二进制代码文件中的模式, 以对未知恶意软件进行分类。Schultz 等人[5]首次提出了利用机器学习技术来实现恶意软件检测。使用机器学习技术进行恶意行为检测的第一步是确定可执行文件的表示形式,该方法使用了三种不同的特征: 字节序列 n-gram,可移植的可执行文件(Portable Executable , PE)特征与字符串特征。其中字节 n-gram 特征是从可执行文件中提取的 n 字节序列, 而 n-gram 已被广泛应用于语言建模与语言识别领域[6]。Schultz 等人[5]设计了四个分类器进行对比实验, 分别为基于签名分类, 基于规则RIPPER 算法的分类, 基于朴素贝叶斯(Naive Bayes,NB) 分类及基于多重朴素贝叶斯(Multi-Naive Bayes)分类。实验结果表明使用机器学习技术可以更有效地实现恶意软件检测。在微结构特征级别, Malone 等人[7]与 Xia 等人[8]将 HPC 数据特征应用到异常检测的问题上, 并取得了有效的检测结果。于是, Demme 等人[9] 提出了利用 HPC 特征数据来实现恶意软件检测。该方法收集样本在执行过程中产生的多维 HPC 数据, 并利用机器学习技术实现恶意软件检测。但是该方法的最高检测准确率只有 90%, 检测准确率低的主要原因可能是检测器直接对所收集到的数据进行分析而未进行有效地数据预处理, 因此数据包含了大量噪声。用GPRs微结构特征进行恶意软件检测, 而如何从CPU中收集恶意样本与良性样本在执行过程中所产生的行为信息至关重要。基于动态分析的恶意软件检测技术都是将待分析的可执行文件在隔离环境中进行执行, 因为直接将恶意软件在物理机中执行而不隔离会对计算机系统及网络造成恶劣的影响。为了进一步降低 HPC 特征的维度, Sayadi 等人[10]采用主成分分析(Principal Component Analysis , PCA)技术从收集的特征数据中选择最佳的 HPC 特征数据。为了提高检测性能, Yerima 等人[11]利用集成学习技术构建最终的恶意软件检测模型。同 Sayadi 等人[10]提出的方法类似, Khasawneh 等人[12]也利用集成学习技术设计恶意软件检测模型, 但是该方法主要验证了 Opcode 特征在计算机微结构级别的检测性能。虽然 Sayadi 等人[10]与 Khasawneh 等人[12] 能够利用微结构特征并使用机器学习技术实现有效的恶意软件检测, 但是这两种方法都需要依赖过长的检测窗口。

综合现有文献评述，利用高级别特征以及微结构特征都可以实现对恶意软件的检测。但是使用微结构特征实现恶意软件检测能够有效避免恶意软件在应用层或操作系统层逃避检测的情况发生[7], 并且不需要等待低级别特征转换成高级别特征, 就可以直接依赖底层低级别数据直接实现决策, 提高检测灵敏度和检测效率。通过以上对比可以发现, 目前微结构特征实现恶意软件检测所依赖的数据集较小, 主要原因为在计算机底层, 短时间内会收集到大量的微结构特征, 在模型的设计过程中需要权衡所需要分析的数据量。虽然目前存在的微结构特征检测技术 FPr 相比较与某些高级别特征检测技术较高, 例如 Khasawneh等人[8] FPr 为 0.056, 但是在计算机底层获得的微结构特征数据复杂度较高, 并且容易受到噪声的干扰。而本文与目前存在的微结构检测技术相比, 能够依赖较小的检测窗口(分析 0.7K 指令信息)就可以获得较高的检测性能。使用 TF-IDF 技术来降低特征数据的复杂性, 并抽取有区分性的信息以进行恶意软件检测，该预处理方式有利于检测模型的学习，通过构造集成分类的模型, 可以进一步提高检测性能，在未知数据集和真实数据上的综合实验证明, 基于集成学习的恶意PE软件特征检测与识别比其他代表性的模型具有更好的检测性能。

## 研究内容及创新点

Xxx

# 样本检测关键技术

## PE恶意样本常见特征

### DOS头及PE 文件头

### 节表及节区数据

### 关键数据结构

导入表、导出表、资源表、重定位表

### 其他重要数据

调试信息：可包含PDB文件路径。

TLS（Thread Local Storage）：定义线程启动/结束时调用的回调函数。

安全证书：数字签名信息（位于WIN\_CERTIFICATE结构）。

## 机器学习算法选择

### 常见机器学习算法比较

### 经典机器学习与深度学习的选择

## 集成学习技术

### Bagging

### Boosting

### Stacking

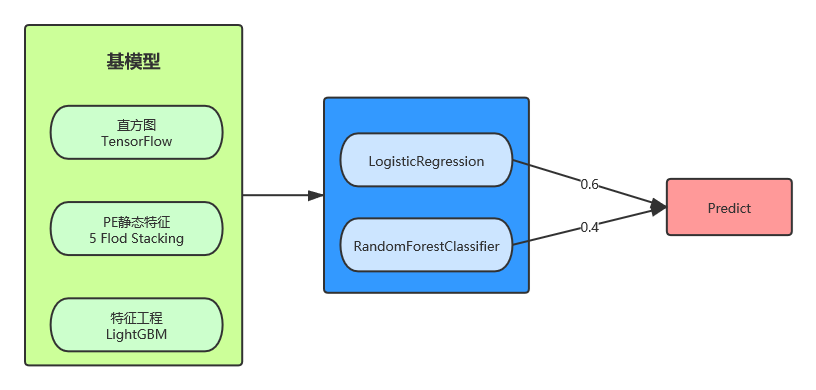
### 基模型与元模型

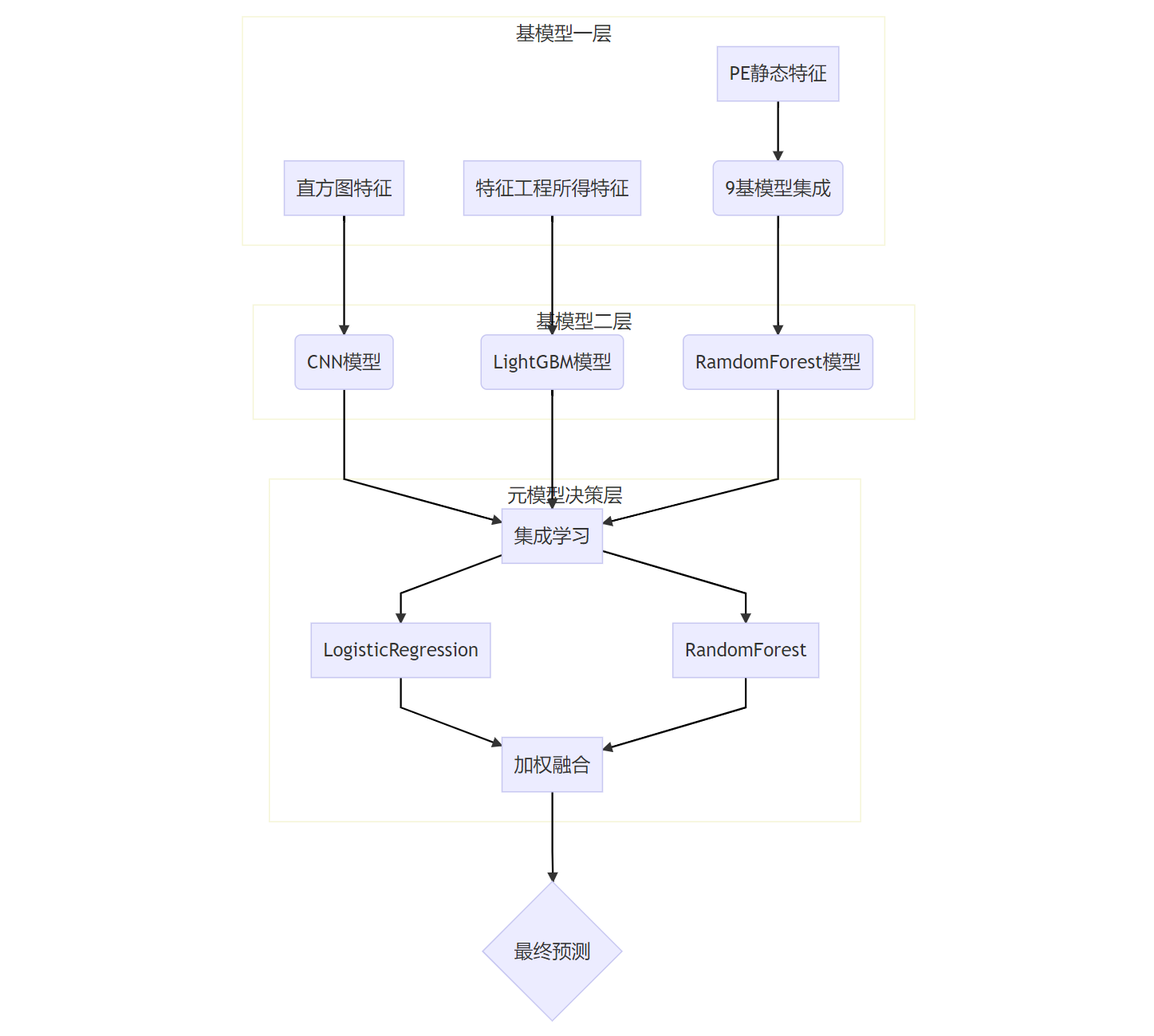
### 加权融合

## 特征哈希技术

# 模型架构

## 三层模型集成架构





### 基模型一层

### 基模型二层

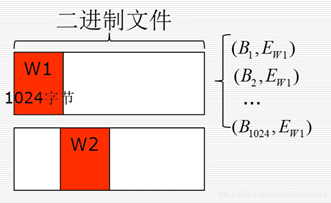
### 元模型决策层

## 直方图特征模型

在本实验中主要使用两类直方图：

（1）字节直方图：统计字节0-255出现个数；

（2）字节熵直方图：滑动一个1024字节的窗口，步长为256字节，计算每个1024字节窗口的熵，同时统计滑动窗口的（字节，熵值）对，最后转换成1x256维的特征向量。



### 字节直方图

### 字节熵直方图

### 卷积神经网络构建

将获取到的特征数据导入模型，通过卷积神经网络（CNN）构建图像分类模型。这里使用Keras定义一个CNN，输入层接收形状为(LENGTH, 1)的直方图数据，将其重塑为二维图像；网络包括两个卷积层（Conv2D），每个卷积层后接一个最大池化层（MaxPooling2D），用于特征提取和降维；接下来是一个全连接层（Dense），用于学习更高级别的抽象特征，以及一个Dropout层用于正则化，防止过拟合。输出层是一个单节点的Dense层，使用sigmoid激活函数，输出一个介于0和1之间的概率值，用于二分类任务。

## PE静态特征模型

虽然提供的样本被抹掉了样本PE结构中的MZ、PE、导入导出表等信息，但我们只需要恢复MZ头和PE\0\0即可使用常规的分析工具对PE样本进行分析。

但恢复也只是能解析PE文件的静态格式和特征，并不能将其运行，所以我们只能从静态特征入手。最后使用的是著名EMBER数据集提到的PE文件静态特征提取方法。虽然原文用于检测恶意Windows PE文件，但是我们也将其移植过来检测挖矿软件。

本实验所提取的PE文件静态特征如下：

（1）ByteHistogram、ByteEntropyHistogram：直方图；

（2）GeneralFileInfo：调试信息、TLS段、重定位信息等；

（3）HeaderFileInfo：PE头基本所有信息；

（4）ExportsInfo：导出表个数、名称；

（5）SectionInfo：Section名、大小、熵、属性等；

（6）ImportsInfo：导入表被破坏，无法解析导入函数信息；

（7）StringExtractor：字符串提取，放在特征工程。

### LIEF解析库

### 特征哈希结构化处理

### 9基模型堆叠

PE静态特征所使用的模型包括袋装集成（Bagging Classifier），梯度提升（Gradient Boosting），逻辑回归（Logistic Regression），线性支持向量机（Linear Support Vector Classifier），决策树（Decision Tree），优化梯度提升（eXtreme Gradient Boosting），随机森林（Random Forest），额外树（Extremely Randomized Trees），自适应增强（AdaBoost）等9种机器学习模型用于构建基模型。

将所有模型训练后，通过堆叠（Stacking）的方式，将这些模型的预测结果作为输入，再训练一个元模型来进行最终预测。

### 随机森林二层基模型构建

## 特征工程模型

本实验中所用特征工程主要包括五部分，分别为：Section信息、字符匹配、Yara匹配、Opcode和其他布尔信息。

### Section信息

Section信息所包含的节区特征是PE文件一种重要特征，过多的节区、异常的节区名、异常的资源节区个数等指标都可以指示这个PE文件的可疑程度，因此我们首先针对节区进行特征统计：

（1）OEP所在节区名长度：OEP所在节区名一般为.text，如果过长或过短说明很可能被混淆，比如UPX壳OEP处节区名为UPX1；

（2）各可读、可写、可执行节区大小和熵，和各属性节区占文件大小比例：如果可执行节区占比过小，很可能加壳了（压缩壳）；

（3）资源节区个数：资源节区一般藏又一些压缩数据，比如挖矿恶意载荷；

（4）节区总个数：恶意软件节区数一般比较多。

### 字符匹配

根据挖矿样本中较常出现的字符进行相应的正则匹配，其中包括：

（1）路径、注册表、URL、IP地址正则匹配：认为操作注册表必然存在相应函数，而这些函数名基本含有“reg”，匹配注册表就简单地匹配“reg”实现；

（2）比特币钱包地址正则匹配：主要写了三种货币：比特币、莱特币、门罗币；

（3）一些重要字符串匹配：“MZ”“PE”表面可能含别的PE文件，“pool”“cpu”“gpu”“coin”则是认为挖矿软件普遍存在的字符串。

### Yara匹配

Yara规则是基于二进制文件中包含的文本或二进制字符串的描述创建的。首先使用Yara-Rules提供的规则进行匹配，其中包括：

（1）壳规则：包含许多已知壳的Yara匹配规则；

（2）密码学常量规则：特别是哈希算法初始值，匹配时间略长。

除此之外还使用了yarGen工具，提取训练集黑样本特征，其原理是先解析出样本集中的共同的字符串，然后经过白名单库的过滤，最后通过启发式、机器学习等方式筛选出最优的Yara规则。根据得到的Yara规则集结果，选择匹配度大于某一阈值的规则形成新的规则集，查看匹配黑白样本的分布比例，筛选部分白样本规则。通过不断的调整阈值参数与筛除比例，在尽可能泛化的同时匹配到更多的黑样本，最后人工结合挖矿特征筛选出更值得关注的部分，优化规则集。

最终得到的自定义Yara规则集阈值为20，即每条规则在训练集黑样本中匹配样本个数大于等于20。

### Opcode

在样本分析中，Opcode特征指的是从程序的机器代码中提取出的操作码序列。操作码（Opcode）是计算机指令集架构（ISA）中每条指令的识别码，它告诉处理器执行什么样的操作。在二进制代码层面，每个操作码对应着特定的二进制模式。

通过传统逆向工具解析PE文件中的函数实在太耗时，因此打算通过简单的正则搜索识别代码中的函数，然后提取函数片段中的Opcode并保存。例如x86下，按栈匹配push ebp; mov ebp, esp; ……; ret如下代码段：

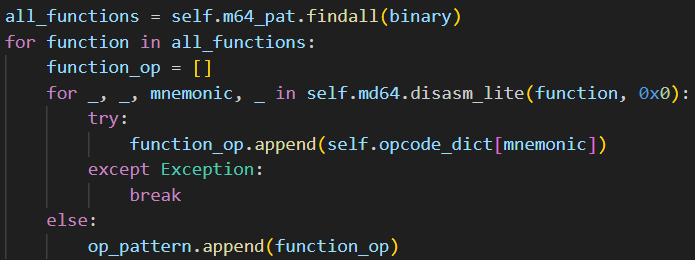


图6-9 Opcode特征提取代码实现

原因是发现在挖矿样本中有大量样本间共有的Opcode特征，而白样本中却不明显。因此可以统计匹配出的函数个数、Opcode种类个数、平均值、方差等特征。

### 其他布尔信息

根据各种类别的进程名、数字货币名、密码学算法名、杀毒软件的进程名称、Opcode词汇表、常见矿池二级域名、虚拟机内软件名等信息，并将它们存储在./data中，以检测它们是否在给定PE文件内。考虑到挖矿软件需要反分析、反调试来保证持久化，这里主要以常见的调试器进程名作为挖矿软件的反调试行为的特征。

考虑到挖矿软件会进行反沙箱对抗，于是考虑搭建了各种不同的虚拟机软件环境，整理了挖矿软件可能检测的虚拟机环境特征。

由于挖矿过程必然存在矿池，因此想到检测矿池二级域名来作为挖矿软件的一个有效特征，于是编写了爬虫脚本通过正则匹配的方式爬取了目前常用的矿池域名300余条。

### LightGBM模型构建

通过LightGBM算法对特征工程产生的特征进行模型构建。

直方图特征

融合字节直方图（256维）和字节熵直方图（256维），最终生成512维复合特征向量，特征值以ASCII明文存储（每行一个浮点数，并且归一化了）

```python

# normalization

Histogram = np.array(Histogram) / file\_size

Byte\_Entropy = np.array(Byte\_Entropy) / Sum

# 特征融合（512维复合特征）

feature = np.concatenate((Histogram, Byte\_Entropy), axis=-1)

feature = list(feature)

```

- 256维：全局字节分布特征 （检测加密/填充数据）

- 256维：局部熵值变化特征 （识别代码段/混淆区域）

- 两者结合可提升对高级恶意软件（如多态病毒、内存驻留木马）的检测能力

字节直方图（ByteHistogram）

计算每个字节值（0-255）的出现次数

特征维度 ：输出 256 维的特征向量（每个维度对应一个字节值的出现概率）

Histogram = ByteHistogram().raw\_features(data, None)

print(len(Histogram))

print(Histogram)

字节熵直方图（Byte\_Entropy）

\* 滑动一个1024字节的窗口，步长为256字节

\* 计算每个1024字节窗口的熵

\* 统计滑动窗口的（字节，熵值）对，最后转换成1x256维的特征向量

双维度特征：

X轴（字节值）：将256种字节值压缩为16个bin（每个bin对应16个连续字节值）

Y轴（局部熵）：计算每个窗口的熵值并量化为16个等级

滑动窗口机制：

window=2048 # 分析窗口大小（2KB）

step=1024 # 滑动步长（1KB）产生50%重叠

这种50%重叠的窗口设计可以捕捉更细粒度的局部特征

熵值计算优化：

使用右移运算 >> 4 将字节值从256维压缩到16维，降低计算复杂度

添加 1e-10 避免零概率导致的数学错误

熵值乘以2补偿信息损失（从8bit熵空间压缩到4bit）

原始方法提取了许多PE文件静态特征，如下：

- [x] ByteHistogram、ByteEntropyHistogram：直方图

- [x] GeneralFileInfo：调试信息、TLS段、重定位信息……

- [x] HeaderFileInfo：PE头基本所有信息

- [x] ExportsInfo：导出表个数、名称

- [x] SectionInfo：Section名、大小、熵、属性等……

- [ ] ImportsInfo：导入表被破坏，无法解析导入函数信息

- [ ] StringExtractor：字符串提取在特征工程里做，这里删掉一是为了节省时间，二是防止特征重叠

特征维度：256+256+10+62+128+255 = 967维

LIEF（Library to Instrument Executable Formats） 是一个跨平台的二进制文件解析库，支持PE/ELF/Mach-O等格式。

```python

try:

# 使用LIEF库解析PE文件

lief\_binary = lief.PE.parse(list(bytez))

except (lief.bad\_format, lief.bad\_file, lief.pe\_error, lief.parser\_error, RuntimeError) as e:

print("lief error: ", str(e))

lief\_binary = None # 解析失败时设为None

except Exception: # 其他异常直接抛出

raise

```

异常处理逻辑 ：

- 处理损坏PE文件（如加壳样本）

- 跳过无法解析的文件（返回空特征）

作用：

- 检测节区属性异常（如可写可执行节区）

- 识别非常规导入函数（如无Kernel32导入）

- 分析资源段内容（如隐藏的恶意脚本）

LIEF分析结果示例：

```python

fp = '../tmp/00a8b0ff6c1a48a69f85657e57d9ed99'

with open(fp, 'rb') as f:

bytez = f.read()

lief\_binary = lief.PE.parse(list(bytez))

print(lief\_binary)

```

```python

Dos Header

==========

Magic: 5a4d

Used Bytes In The LastPage: 90

File Size In Pages: 3

Number Of Relocation: 0

Header Size In Paragraphs: 4

...

OEM info: 0

Address Of New Exe Header: 130

Rich Header

===========

Key: f6401d48

- ID: 0x1020 Build ID: 0x64eb Count: 1

- ID: 0x1000 Build ID: 0x64eb Count: 1

- ID: 0x1040 Build ID: 0x64eb Count: 5

...

- ID: 0xf100 Build ID: 0x9cb4 Count: 19

- ID: 0x0000 Build ID: 0x0000 Count: 0

Header

======

Signature: 50 45 0 0

Machine: I386

Number Of Sections: 4

...

Time Date Stamp: 5a451f1f

Optional Header

===============

Magic: 10b

Major Linker Version: d

Minor Linker Version: 19

Size Of Code: 7ac00

...

Loader Flags: 0

Number Of RVA And Size: 10

Data directories

================

Data directory "EXPORT\_TABLE"

RVA: 0x 97090

Size: 0x 4c

Section: .rdata

Data directory "IMPORT\_TABLE"

RVA: 0x 0

Size: 0x 0

...

Data directory "CLR\_RUNTIME\_HEADER"

RVA: 0x 0

Size: 0x 0

Sections

========

.text 7abca 1000 7ac00 400 0 7.8773 CNT\_CODE - MEM\_EXECUTE - MEM\_READ

.rdata 1cdf2 7c000 1ce00 7b000 0 7.35952 CNT\_INITIALIZED\_DATA - MEM\_READ

.data a6c60 99000 a5000 97e00 0 3.29984 CNT\_INITIALIZED\_DATA - MEM\_READ - MEM\_WRITE

.reloc 5778 140000 5800 13ce00 0 5.65389 CNT\_INITIALIZED\_DATA - MEM\_DISCARDABLE - MEM\_READ

TLS

===

Address Of Index: 1e49e52c

Address Of Callbacks: 1e3dc5e8

Virtual Address of RawData (start): 1e3f248c

Virtual Address of RawData (end): 1e3f2494

Size Of Zero Fill: 0

Associated section: .rdata

Debug

=====

Characteristics: 8a84c6d4

Timestamp: 82c6e175

Major version: ae33

...

Relocations

===========

1000 e0

- HIGHLOW a

- HIGHLOW f

- HIGHLOW 18

- HIGHLOW 1e

...

2000 98

- HIGHLOW 1b

- ABSOLUTE 0

- HIGHLOW 7a

...

...

13b000 34

- ABSOLUTE 0

- ABSOLUTE 0

- ABSOLUTE 0

- HIGHLOW 30

Export

======

m64j\_\_\_hw.dll

DllInstall 1 5ef0

Symbols

=======

Load Configuration

==================

Version: UNKNOWN

Characteristics: 0xa0

Timedatestamp: 0

Major version: 0

...

Edit list: 0

Security cookie: 0x1e48a5e8

```

这五部分特征，除了字节直方图和字节熵直方图外（这俩可以直接算），其余均是根据LIEF分析结果来处理的。

```python

features = [ # 集成了5个对象

ByteHistogram(), ByteEntropyHistogram(), GeneralFileInfo(),

HeaderFileInfo(), ExportsInfo(), SectionInfo()

]

```

以section特征为例

展示`SectionInfo`对象的定义

```python

class SectionInfo():

name = 'section'

dim = 255 # 特征维度

# 用做处理lief分析结果

def raw\_features(self, bytez, lief\_binary):

...

# 进一步处理，标准化、结构化结果

def process\_raw\_features(self, raw\_obj):

...

...

```

lief处理结果输出看看

```python

fe = features[5]

print(len(fe.raw\_features(bytez, lief\_binary)))

print({fe.name: fe.raw\_features(bytez, lief\_binary)})

```

```python

{

'section': {

'entry': '.text',

'sections': [

{'name': '.text', 'size': 502784, 'entropy': 7.8772955321013445, 'vsize': 502730, 'props': ['CNT\_CODE', 'MEM\_EXECUTE', 'MEM\_READ']},

{'name': '.rdata', 'size': 118272, 'entropy': 7.359519722739576, 'vsize': 118258, 'props': ['CNT\_INITIALIZED\_DATA', 'MEM\_READ']},

{'name': '.data', 'size': 675840, 'entropy': 3.2998353275291605, 'vsize': 683104, 'props': ['CNT\_INITIALIZED\_DATA', 'MEM\_READ', 'MEM\_WRITE']},

{'name': '.reloc', 'size': 22528, 'entropy': 5.653888730136691, 'vsize': 22392, 'props': ['CNT\_INITIALIZED\_DATA', 'MEM\_DISCARDABLE', 'MEM\_READ']}

]

}

}

```

进一步处理数据，使其结构化

```python

fe = features[5]

raw\_obj = {fe.name: fe.raw\_features(bytez, lief\_binary)}

print(len(raw\_obj[fe.name]))

print(raw\_obj[fe.name])

print(len(fe.process\_raw\_features(raw\_obj[fe.name])))

print(fe.process\_raw\_features(raw\_obj[fe.name])) # normalization

```

```python

2

{'entry': '.text',

'sections': [

{'name': '.text', 'size': 502784, 'entropy': 7.8772955321013445, 'vsize': 502730, 'props': ['CNT\_CODE', 'MEM\_EXECUTE', 'MEM\_READ']},

{'name': '.rdata', 'size': 118272, 'entropy': 7.359519722739576, 'vsize': 118258, 'props': ['CNT\_INITIALIZED\_DATA', 'MEM\_READ']},

{'name': '.data', 'size': 675840, 'entropy': 3.2998353275291605, 'vsize': 683104, 'props': ['CNT\_INITIALIZED\_DATA', 'MEM\_READ', 'MEM\_WRITE']},

{'name': '.reloc', 'size': 22528, 'entropy': 5.653888730136691, 'vsize': 22392, 'props': ['CNT\_INITIALIZED\_DATA', 'MEM\_DISCARDABLE', 'MEM\_READ']}

]}

```

```python

255

[ 4.0000000e+00 0.0000000e+00 0.0000000e+00 1.0000000e+00

1.0000000e+00 0.0000000e+00 0.0000000e+00 0.0000000e+00

0.0000000e+00 0.0000000e+00 -2.2528000e+04 0.0000000e+00

0.0000000e+00 0.0000000e+00 0.0000000e+00 0.0000000e+00

0.0000000e+00 0.0000000e+00 0.0000000e+00 1.1827200e+05

0.0000000e+00 0.0000000e+00 0.0000000e+00 0.0000000e+00

...

0.0000000e+00 1.0000000e+00 0.0000000e+00 0.0000000e+00

0.0000000e+00 0.0000000e+00 0.0000000e+00 0.0000000e+00

0.0000000e+00 0.0000000e+00 0.0000000e+00 0.0000000e+00

0.0000000e+00 0.0000000e+00 0.0000000e+00 0.0000000e+00

0.0000000e+00 0.0000000e+00 0.0000000e+00 0.0000000e+00

0.0000000e+00 0.0000000e+00 0.0000000e+00]

```

`process\_raw\_features`函数的核心部分：

- 怎么做到能够将非量化的数据转化为量化数据的？

- 通过特征哈希（FeatureHasher）技术将非结构化数据转换为固定维度的数值向量

处理LIEF分析结果后，得到的原始数据包括entry节名称和各个section的name、size、entropy、vsize、props等。这些数据中有字符串和类别型数据，比如name和props，这些非数值型数据需要转换为数值特征

```python

from sklearn.feature\_extraction import FeatureHasher

```

```python

fe = features[5]

raw\_obj = {fe.name: fe.raw\_features(bytez, lief\_binary)}[fe.name]

print(raw\_obj)

```

```python

{'entry': '.text',

'sections': [

{'name': '.text', 'size': 502784, 'entropy': 7.8772955321013445, 'vsize': 502730, 'props': ['CNT\_CODE', 'MEM\_EXECUTE', 'MEM\_READ']},

{'name': '.rdata', 'size': 118272, 'entropy': 7.359519722739576, 'vsize': 118258, 'props': ['CNT\_INITIALIZED\_DATA', 'MEM\_READ']},

{'name': '.data', 'size': 675840, 'entropy': 3.2998353275291605, 'vsize': 683104, 'props': ['CNT\_INITIALIZED\_DATA', 'MEM\_READ', 'MEM\_WRITE']},

{'name': '.reloc', 'size': 22528, 'entropy': 5.653888730136691, 'vsize': 22392, 'props': ['CNT\_INITIALIZED\_DATA', 'MEM\_DISCARDABLE', 'MEM\_READ']}

]}

```

```python

sections = raw\_obj['sections']

# 第一部分：基础统计特征（5维）

general = [

len(sections), # 总节区数量

sum(1 for s in sections if s['size'] == 0), # 空尺寸节区数量

sum(1 for s in sections if s['name'] == ""), # 无名节区数量

sum(1 for s in sections if 'MEM\_READ' in s['props'] and 'MEM\_EXECUTE' in s['props']), # 可读可执行节区

sum(1 for s in sections if 'MEM\_WRITE' in s['props']) # 可写节区

]

print(general)

```

```python

[4, 0, 0, 1, 1]

```

```python

# 第二部分：哈希技巧转换（每个FeatureHasher生成50维）

# 节区尺寸哈希（名称+尺寸）

section\_sizes = [(s['name'], s['size']) for s in sections]

section\_sizes\_hashed = FeatureHasher(50, input\_type="pair").transform([section\_sizes]).toarray()[0]

print(section\_sizes)

print(section\_sizes\_hashed)

```

```python

[('.text', 502784), ('.rdata', 118272), ('.data', 675840), ('.reloc', 22528)]

[ 0. 0. 0. 0. 0. -22528. 0. 0.

0. 0. 0. 0. 0. 0. 118272. 0.

0. 0. 0. 0. 0. 0. 0. 0.

0. 0. 0. 0. 0. 0. 0. 0.

-675840. 0. 0. 0. 0. 0. 0. 0.

0. -502784. 0. 0. 0. 0. 0. 0.

0. 0.]

```

`FeatureHasher`解析：

```python

section\_sizes\_hashed = FeatureHasher(50, input\_type="pair").transform([section\_sizes]).toarray()[0]

```

- \*\*参数解释\*\*：

- `n\_features=50`：输出向量的维度为 50。

- `input\_type="pair"`：输入是键值对列表。

- \*\*内部过程\*\*：

1. \*\*哈希键名\*\*：对每个键（如 `.text`）计算两次哈希：

- 第一次确定索引位置（`hash(key) % 50`）。

- 第二次确定符号（`1` 或 `-1`，通过奇偶性判断）。

2. \*\*累加值\*\*：将值乘以符号后累加到对应索引位置。

```python

# 原始数据（假设）

sections = [('.text', 1024), ('.data', 2048), ('.text', 4096)]

# 经过哈希函数映射（示意）

hash('.text') % 50 → 12

hash('.data') % 50 → 35

# 特征向量生成（伪代码）

feature\_vector[12] += 1024 + 4096 = 5120

feature\_vector[35] += 2048 = 2048

```

- \*\*负值的来源\*\*：

- 键 `.text` 被哈希到索引 `40`，且符号为 `-1`，因此 `502784 \* -1 = -502784`。

- 类似地，`.rdata` 的符号为 `+1`，哈希到索引 `14`，因此值为 `118272`。

- \*\*稀疏性\*\*：

- 大部分位置为 `0`，因为只有 4 个键值对参与哈希。

符号对哈希冲突的影响：

1. \*\*哈希冲突\*\*：

- 不同键可能映射到同一索引（例如 `.text` 和 `.data` 哈希到同一位置），但通过符号随机性可以部分抵消冲突的影响。

- 冲突概率与 `n\_features` 相关，维度越大冲突越少。

2. \*\*符号的作用\*\*：

- 例如，假设两个冲突的键值对 `('A', 100)` 和 `('B', 200)`，若符号分别为 `+1` 和 `-1`，则累加结果为 `100 - 200 = -100`，冲突带来的误差小于直接相加 `100 + 200 = 300`。

使用特征哈希的原因：

- \*\*非结构化数据的处理\*\*：特征哈希天然支持键值对、字符串、分类变量等非结构化数据，解决类别型数据（如节区名称）的编码问题，同时保留名称与数值的关联关系（通过` input\_type="pair" `），无需预先构建词表。

- \*\*内存友好\*\*：避免存储庞大的词表。

- \*\*在线学习\*\*：动态处理新特征（例如新出现的段名 `.new\_section`）。

- \*\*维度可控\*\*：输出维度固定，将不定长的节区信息压缩为固定维度，适合机器学习模型输入。

这个例子中的映射结果：

| 键名 | 原始值 | 哈希位置 | 符号 | 贡献值 |

| :------- | :----- | :------- | :--- | :------ |

| `.text` | 502784 | 40 | -1 | -502784 |

| `.rdata` | 118272 | 14 | +1 | +118272 |

| `.data` | 675840 | 32 | -1 | -675840 |

| `.reloc` | 22528 | 5 | -1 | -22528 |

最终向量中只有索引 `5, 14, 32, 40` 处有非零值，其余为 0。

通过这种技术，即使是非结构化的键值对数据（如二进制文件的段信息），也能被高效地转换为固定维度的数值向量，供机器学习模型直接使用。

同理可得：

```python

# 节区熵值哈希（名称+熵）

section\_entropy = [(s['name'], s['entropy']) for s in sections]

section\_entropy\_hashed = FeatureHasher(50, input\_type="pair").transform([section\_entropy]).toarray()[0]

print(section\_entropy)

print(section\_entropy\_hashed)

```

```python

[('.text', 7.8772955321013445), ('.rdata', 7.359519722739576), ('.data', 3.2998353275291605), ('.reloc', 5.653888730136691)]

[ 0. 0. 0. 0. 0. -5.65388873

0. 0. 0. 0. 0. 0.

0. 0. 7.35951972 0. 0. 0.

0. 0. 0. 0. 0. 0.

0. 0. 0. 0. 0. 0.

0. 0. -3.29983533 0. 0. 0.

0. 0. 0. 0. 0. -7.87729553

0. 0. 0. 0. 0. 0.

0. 0. ]

```

```python

# 节区虚拟大小哈希（名称+虚拟大小）

section\_vsize = [(s['name'], s['vsize']) for s in sections]

section\_vsize\_hashed = FeatureHasher(50, input\_type="pair").transform([section\_vsize]).toarray()[0]

```

```python

[('.text', 502730), ('.rdata', 118258), ('.data', 683104), ('.reloc', 22392)]

[ 0. 0. 0. 0. 0. -22392. 0. 0.

0. 0. 0. 0. 0. 0. 118258. 0.

0. 0. 0. 0. 0. 0. 0. 0.

0. 0. 0. 0. 0. 0. 0. 0.

-683104. 0. 0. 0. 0. 0. 0. 0.

0. -502730. 0. 0. 0. 0. 0. 0.

0. 0.]

```

```python

# 入口节名称哈希

entry\_name\_hashed = FeatureHasher(50, input\_type="string").transform([raw\_obj['entry']]).toarray()[0]

print(raw\_obj['entry'])

print(entry\_name\_hashed)

```

```python

.text

[ 0. 0. 0. -1. 0. 0. 0. 0. 0. 1. 0. 0. 0. 0. 0. 0. 0. 0.

0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.

0. 0. 0. -2. 0. 0. 0. 1. 0. 0. 0. 0. 0. 0.]

```

```python

# 入口节属性哈希

characteristics = [p for s in sections for p in s['props'] if s['name'] == raw\_obj['entry']]

characteristics\_hashed = FeatureHasher(50, input\_type="string").transform([characteristics]).toarray()[0]

print(characteristics)

print(characteristics\_hashed)

```

```python

['CNT\_CODE', 'MEM\_EXECUTE', 'MEM\_READ']

[ 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. -1. 0. 0. 0. 0.

0. 0. 0. 0. 0. 1. 0. 0. 0. 0. 1. 0. 0. 0. 0. 0. 0. 0.

0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]

```

```python

# 第三部分：特征拼接（5 + 50\*5 = 255维）

return np.hstack([

general, # 5

section\_sizes\_hashed, # 50

section\_entropy\_hashed, # 50

section\_vsize\_hashed, # 50

entry\_name\_hashed, # 50

characteristics\_hashed # 50

]).astype(np.float32)

```

### 0x30 特征工程

主要包括五部分，分别为：Section信息、字符匹配、Yara匹配、Opcode和其他布尔信息

```python

def get\_feature\_engineering(self, sample\_data):

# 阶段1：节区信息特征（16个基础特征）

tmp\_section = self.get\_section\_infomation(sample\_data) # 获取节区基本信息

...

# 阶段2：字符串模式匹配（26个特征）

tmp\_match = self.string\_match(sample\_data) # 匹配钱包地址、路径、URL等模式

# 阶段3：YARA规则检测（2个特征）

tmp\_yara = self.yara\_match(sample\_data) # 检测加壳和通用恶意规则

# 阶段4：关键字扫描（5个特征）

tmp\_count = self.string\_count(sample\_data) # 统计AV/调试工具/矿池等关键词

# 阶段5：操作码分析（7个统计特征）

tmp\_opcode = self.opcodes(sample\_data) # 提取函数操作码序列特征

# 特征融合（共16+26+2+5+7=56维）

res\_dict = ChainMap(tmp\_section, tmp\_match, tmp\_yara, tmp\_count, tmp\_opcode)

res = [res\_dict[key] for key in self.keys]

return res

```

\* 关键方法调用关系

```plaintext

get\_feature\_engineering

├─ get\_section\_infomation → 节区属性分析（LIEF解析）

├─ string\_match → 正则模式匹配（钱包/URL/路径等）

├─ yara\_match → 恶意规则检测

├─ string\_count → 关键字扫描

└─ opcodes → 反汇编特征提取

```

\* 典型特征示例

1. entr\_X\_weight （可执行节熵值占比）：

- 正常样本：0.15-0.25（代码节中等熵值）

- 恶意样本：>0.3（加壳/加密代码的高熵特征）

2. pool\_name\_count （矿池关键词）：

- 挖矿软件：≥3个常见矿池名称

- 正常软件：0-1个偶然匹配

3. opcode\_uniq （唯一操作码数量）：

- 勒索软件：200+（复杂逻辑）

- 下载器：50-100（简单循环结构）

例子

```python

from feature\_engineering import Feature\_engineering

fn = Feature\_engineering()

fp = test\_fixed\_path[0] # ../tmp/00a8b0ff6c1a48a69f85657e57d9ed99

with open(fp, 'rb') as f:

sample\_data = f.read()

```

```python

tmp\_section = fn.get\_section\_infomation(sample\_data)

section\_keys = ["size\_R", "size\_W", "size\_X", "entr\_R", "entr\_W", "entr\_X"]

for k in section\_keys:

file\_size = tmp\_section['file\_size']

tmp = tmp\_section[k]

tmp\_section["{}\_weight".format(k)] = tmp / file\_size

print(len(tmp\_section))

print(tmp\_section)

```

```python

16

{'entry': 5, 'size\_R': 329856.0, 'size\_W': 675840.0, 'size\_X': 502784.0, 'entr\_R': 6.047634828126693, 'entr\_W': 3.2998353275291605, 'entr\_X': 7.8772955321013445, 'rsrc\_num': 0, 'section\_num': 4, 'file\_size': 1320448, 'size\_R\_weight': 0.24980612640558356, 'size\_W\_weight': 0.5118262892594029, 'size\_X\_weight': 0.3807677394338891, 'entr\_R\_weight': 4.579987116589743e-06, 'entr\_W\_weight': 2.499027093478244e-06, 'entr\_X\_weight': 5.96562343394162e-06}

```

```python

tmp\_match = fn.string\_match(sample\_data)

print(len(tmp\_match))

print(tmp\_match)

```

```python

26

{'btc\_count': 0, 'btc\_mean': 0, 'ltc\_count': 1, 'ltc\_mean': 26.0, 'xmr\_count': 0, 'xmr\_mean': 0, 'paths\_count': 2, 'paths\_mean': 4.0, 'regs\_count': 1, 'regs\_mean': 3.0, 'urls\_count': 0, 'urls\_mean': 0, 'ips\_count': 1, 'ips\_mean': 7.0, 'mz\_count': 11, 'mz\_mean': 2.0, 'pe\_count': 19, 'pe\_mean': 2.0, 'pool\_count': 7, 'pool\_mean': 4.0, 'cpu\_count': 10, 'cpu\_mean': 3.0, 'gpu\_count': 0, 'gpu\_mean': 0, 'coin\_count': 0, 'coin\_mean': 0}

```

```python

tmp\_yara = fn.yara\_match(sample\_data)

print(len(tmp\_yara))

print(tmp\_yara)

```

```python

2

{'packer\_count': 1, 'yargen\_count': 1}

```

```python

tmp\_count = fn.string\_count(sample\_data)

print(len(tmp\_count))

print(tmp\_count)

```

```python

5

{'av\_count': 9, 'dbg\_count': 0, 'pool\_name\_count': 14, 'algorithm\_name\_count': 2, 'coin\_name\_count': 7}

```

```python

tmp\_opcode = fn.opcodes(sample\_data)

print(len(tmp\_opcode))

print(tmp\_opcode)

```

```python

7

{'opcode\_min': 3, 'opcode\_max': 950, 'opcode\_sum': 17497, 'opcode\_mean': 52.86102719033233, 'opcode\_var': 5681.738994715272, 'opcode\_count': 331, 'opcode\_uniq': 179}

```

#### 0x31 Section信息

节区特征是PE文件一种重要特征，过多的节区、异常的节区名、异常的资源节区个数等指标都可以指示这个PE文件的可疑程度，因此我们首先针对节区进行特征统计：

\* OEP所在节区名长度

\* OEP所在节区名一般为`.text`，如果过长或过短说明很可能被混淆

\* 比如UPX壳OEP处节区名为`UPX1`。

\* 各可读、可写、可执行节区大小和熵，和各属性节区占文件大小比例

\* 举例：如果可执行节区占比过小，很可能加壳了（压缩壳）

\* 资源节区个数

\* 资源节区一般藏又一些压缩数据，比如挖矿恶意载荷

\* 节区总个数

\* 恶意软件节区数一般比较多

```Python

# OEP处section名长度

section\_info["entry"] = len(entry\_section)

section\_info["section\_num"] = len(lief\_binary.sections)

# 可读、可写、可执行sections大小均值

sR, sW, sX = [], [], []

# 可读、可写、可执行sections熵值均值

entrR, entrW, entrX = [], [], []

# 资源section个数

rsrc\_num = 0

for s in lief\_binary.sections:

props = [str(c).split('.')[-1] for c in s.characteristics\_lists]

if "MEM\_READ" in props:

sR.append(s.size)

entrR.append(s.entropy)

if "MEM\_WRITE" in props:

sW.append(s.size)

entrW.append(s.entropy)

if "MEM\_EXECUTE" in props:

sX.append(s.size)

entrX.append(s.entropy)

if 'rsrc' in s.name:

rsrc\_num += 1

section\_info['size\_R'], section\_info['size\_W'], section\_info['size\_X'] = np.mean(sR), np.mean(sW), np.mean(sX)

section\_info['entr\_R'], section\_info['entr\_W'], section\_info['entr\_X'] = np.mean(entrR), np.mean(entrW), np.mean(entrX)

section\_info['rsrc\_num'] = rsrc\_num

```

#### 0x32 字符匹配

根据资格赛获得的启发，队员们手写相应的正则匹配模式，其中包括

\* 路径、注册表、URL、IP地址正则匹配

\* 其中因为注册表正则模式存在回溯问题，有的样本存在特别长的字符串，导致一个样本可能匹配了八分钟，所以我们复赛简单粗暴改成匹配字符串”reg”。主要原因是我们认为操作注册表必然存在相应函数，而这些函数名基本含有”reg”。

\* 比特币钱包地址正则匹配

\* 主要写了三种货币：比特币、莱特币、门罗币

\* 一些重要字符串匹配

\* ”MZ”、”PE”指示可能含别的PE文件

\* ”pool”、”cpu”、”gpu”、”coin”则是我们认为挖矿软件普遍存在的字符串

```Python

self.path\_pattern = re.compile(b'[C-Zc-z]:(?:(?:\\\\|/)[^\\\\/:\*?"<>|"\x00-\x19\x7f-\xff]+)+(?:\\\\|/)?')

self.regs\_pattern = re.compile(b'reg', re.IGNORECASE)# re.compile(b'[A-Z\_ ]{5,}(?:\\\\[a-zA-Z ]+)+')

self.urls\_pattern = re.compile(b'https?://(?:[-\w.]|(?:%[\da-fA-F]{2}))+')

# self.strings\_pattern = re.compile(b'[\x20-\x7f]{5,}')

self.ip\_pattern = re.compile(b'(?:(?:25[0-5]|2[0-4]\d|[01]?\d{1,2})\.){3}(?:25[0-5]|2[0-4]\d|[01]?\d{1,2})')

​

# #比特币钱包地址

self.wallet\_pattern\_btc = re.compile(b'(?:1|3|bc1|bitcoincash:q)(?:(?![0OIi])[0-9A-Za-z]){25,34}')

self.wallet\_pattern\_ltc = re.compile(b'(?:ltc1|M|L)[A-Za-z0-9]{25,36}')

self.wallet\_pattern\_xmr = re.compile(b'[0-9A-Za-z]{90,100}') #门罗币

​

self.mz\_pattern = re.compile(b'MZ')

self.pe\_pattern = re.compile(b'PE')

self.pool\_pattern = re.compile(b'pool', re.IGNORECASE)

self.cpu\_pattern = re.compile(b'cpu', re.IGNORECASE)

self.gpu\_pattern = re.compile(b'gpu', re.IGNORECASE)

self.coin\_pattern = re.compile(b'coin', re.IGNORECASE)

```

Yara规则是基于二进制文件中包含的文本或二进制字符串的描述创建的。

最后我还使用了[yarGen][5]工具，提取训练集黑样本特征，其原理是先解析出样本集中的共同的字符串，然后经过白名单库的过滤，最后通过启发式、机器学习等方式筛选出最优的Yara规则。根据得到的Yara规则集结果，选择匹配度大于某一阈值的规则形成新的规则集，查看匹配黑白样本的分布比例，筛选部分白样本规则。通过不断的调整阈值参数与筛除比例，在尽可能泛化的同时匹配到更多的黑样本，最后人工结合挖矿特征筛选出更值得关注的部分，优化规则集。

最终得到的自定义Yara规则集阈值为20，即每条规则在训练集黑样本\*\*中匹配样本个数大于等于20。

通过传统逆向工具解析PE文件中的函数实在太耗时，因此我们打算通过简单的正则搜索识别代码中的函数，然后提取函数片段中的Opcode并保存。例如x86下，按栈匹配`push ebp; mov ebp, esp; ……; ret`如下代码段。

```Python

self.m32\_pat = re.compile(b'\x55\x8b\xec[^\xc3]\*\xc3')

# …………

all\_functions = self.m32\_pat.findall(binary)

for function in all\_functions:

function\_op = []

for \_, \_, mnemonic, \_ in self.md32.disasm\_lite(function, 0x0):

try:

function\_op.append(self.opcode\_dict[mnemonic])

except Exception:

break

else:

op\_pattern.append(function\_op)

```

原因是发现在挖矿样本中有大量样本间共有的opcode特征，而白样本中却不明显。因此可以统计匹配出的函数个数、opcode种类个数、平均值、方差等特征。

其他布尔信息

我还收集了各种类别的进程名、数字货币名、密码学算法名等信息，并将它们存储在`./data`中，以检测它们是否在给定PE文件内，具体如下表所示。

| 文件名 | 注释 |

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

| algorithm.txt | 常见密码学算法名 |

| av.json | [常见的杀毒软件的进程名称][6] |

| coin.txt | 数字货币名 |

| dbg.txt | 调试器名 |

| domain\_suffix.txt | 常见顶级域名（复赛时未使用 |

| OPCODE.txt | Opcode词汇表 |

| pool.txt | [常见矿池二级域名][7] |

| vm.txt | 虚拟机内软件名 |

我们搜集了常见的杀毒软件的进程名称，以此作为挖矿软件对杀毒软件的检测行为的特征。

考虑到挖矿软件需要反分析、反调试来保证持久化，我们还以常见的调试器进程名作为挖矿软件的反调试行为的特征，主要为我们日常使用的调试工具。

考虑到挖矿软件会进行反沙箱对抗，我们搭建了各种不同的虚拟机软件环境，整理了挖矿软件可能检测的虚拟机环境特征。

由于挖矿过程必然存在矿池，因此我们想到检测矿池二级域名来作为挖矿软件的一个有效特征，于是编写了爬虫脚本通过正则匹配的方式爬取了目前常用的矿池域名300余条。

针对直方图特征设计了一个卷积神经网络(CNN)模型，主要架构如下：

复现论文：Deep Neural Network Based Malware Detection Using Two Dimensional

模型架构：

- 输入重塑（32x16）：将512维直方图转为二维矩阵，保留字节/熵值的空间分布特征

- 小卷积核（2x2）：捕捉局部统计模式，识别异常字节簇（如加密段）

- 通道数递增（60→200）：逐步提取高阶特征，发现复杂模式（如多态代码特征）

- 大密度层（500节点）：增强非线性表达能力，处理高度非线性的恶意模式

- Sigmoid输出：适配二分类任务（恶意/正常），输出恶意概率值

```plaintext

Input(512) → Reshape(32x16) → Conv(60) → Pool → Conv(200) → Pool → Dense(500) → Output(1)

```

```python

# 输入层：接收512维直方图特征

inputs = layers.Input(shape=(LENGTH, 1)) # LENGTH=512

# 特征重塑：将1D特征转换为2D伪图像 (32x16)

re\_inputs = tf.reshape(inputs, [-1, WIDTH, HEIGHT, 1]) # WIDTH=32, HEIGHT=16

# 双卷积层设计：捕捉局部统计特征

Conv\_1 = layers.Conv2D(60, (2,2), activation='relu')(re\_inputs) # 捕捉字节分布模式

Conv\_2 = layers.Conv2D(200, (2,2), activation='relu')(pool\_1) # 提取高阶熵特征

# 池化层

pool\_1 = layers.MaxPooling2D()(Conv\_1) # 捕捉字节分布模式

pool\_2 = layers.MaxPooling2D()(Conv\_2) # 提取高阶熵特征

Flat = layers.Flatten()(pool\_2)

# 全连接层：特征融合与分类

Dense\_1 = layers.Dense(500, activation='relu')(Flat) # 综合卷积特征

outputs = layers.Dense(1, activation='sigmoid')(Dense\_1) # 二分类输出

```

优化部分：

1. \*\*正则化配置\*\*：

```python

dropout = layers.Dropout(0.2)(Dense\_1) # 20%的神经元丢弃率

```

2. \*\*训练优化：\*\*

```python

# 早停机制（6轮耐心值）

EarlyStopping(monitor='val\_loss', patience=6)

# 动态学习率调整（4轮无改善后降半）

ReduceLROnPlateau(patience=4, factor=0.5)

```

适合检测具有以下行为的样本：

1. 代码加密/混淆 （高熵值区域异常分布）

2. 数据填充攻击 （连续重复字节模式）

3. 内存驻留木马 （特殊字节分布模式）

#### 0x42 PE特征模型部分

通过PE文件的967维结构化特征（节区信息/导入表/字符串等）训练多个基模型

```python

# 训练的9个基模型

models = [

'lr' : LogisticRegression, # 逻辑回归

'gbc' : GradientBoostingClassifier, # 梯度提升树

'bc' : BaggingClassifier, # 袋装法集成

'xgb' : XGBClassifier, # XGBoost

'dt' : DecisionTreeClassifier, # 决策树

'svm' : LinearSVC, # 支持向量机

'rfc' : RandomForestClassifier, # 随机森林

'etc' : ExtraTreesClassifier, # 极端随机树

'ada' : AdaBoostClassifier # AdaBoost

]

```

采用二级集成学习架构进行集成：

```plaintext

第一层：9个异构基模型 → 第二层：随机森林元模型（rfc\_pe\_model）

```

基模型通过5折交叉验证生成OOF预测结果，将OOF处理后的训练集堆叠，再训练随机森林元模型

OOF的作用：

\* 对每个折叠，模型仅在 `kf\_x\_train`（训练集部分）训练，并在 `kf\_x\_test`（验证集部分）预测，确保预测结果未见过验证集数据。

- 最终 `oof\_train` 是完整训练集的OOF预测（每个样本的预测来自未训练该样本的模型），`oof\_test` 是测试集的平均预测。

- OOF保证堆叠特征无过拟合风险，因为每个样本的预测来自未见过该样本的模型。

训练随机森林元模型的作用：

将9个基模型的OOF预测结果通过随机森林进一步抽象为单一训练集，作为后续模型的输入。使用 `rfc\_pe\_model` 对9个经OOF后的训练集进行堆叠和训练，理论上可以捕捉基模型之间的非线性关系。

该方法是集成学习中的一种，名为Stacking（堆叠）：用元模型整合基模型的预测结果。另外还有两种，分别是Bagging和Boosting。Bagging（如随机森林）：并行训练多个基模型，通过投票或平均融合结果。Boosting（如AdaBoost、GBDT）：顺序训练基模型，调整样本权重或模型权重。

基模型处理

```python

# 多进程并行预测（L40-75）

for name in raw\_feature\_names: # 遍历9个基模型

pool = Pool(10)

for i, model in enumerate(pe\_raw\_models[name]):

pool.apply\_async(pe\_raw\_predict, (i, model)) # 并行预测

# 取5折结果的平均值

oof\_test = np.mean(oof\_test\_skf)

stacking\_test.append(oof\_test) # 堆叠预测结果

```

0x62 预测融合

```python

# 三级融合策略

test = np.hstack([

histogram\_test, # 直方图CNN模型预测结果

raw\_feature\_test, # PE特征集成模型结果

feature\_engineerin\_test # 特征工程模型结果

])

# 加权投票（6:4比例）

for x, y in zip(lr\_prob, rfc\_prob):

if x[1]\*0.6 + y[1]\*0.4 < 0.5:

final\_label = 0

else:

final\_label = 1

```

# 实验结果

本实验将从直方图特征构建、PE静态特征构建、特征工程三种方式进行特征的选取和创建。根据特征类别，每类特征将单独生成一个模型作为基模型，通过对基模型的预测结果，通过堆叠、融合等集成学习的方式，将新生成的特征集合并，使用新的特征集训练一个元模型作为最终预测模型。

在实验中，直方图特征使用深度学习图像分类模型处理二分类任务；PE静态特征所使用的模型包括袋装集成（Bagging Classifier，BC），梯度提升（Gradient Boosting,GB），逻辑回归（Logistic Regression, LR），线性支持向量机（Linear Support Vector Classifier, Liner-SVC），决策树（Decision Tree, DT），优化梯度提升（eXtreme Gradient Boosting, XGB），随机森林（Random Forest, RF），额外树（Extremely Randomized Tree, ERT），自适应增强（AdaBoost）等9种机器学习模型用于构建基模型；特征工程主要包括五部分，分别为：Section信息、字符匹配、Yara匹配、Opcode和其他布尔信息，最终使用LightGBM算法进行模型训练。

通过对不同类别的特征进行模型构建、预测后，将生成的预测集以6 : 4的比例导入逻辑回归和随机森林两个模型进行模型复合，最终将融合模型的预测结果作为整个模型的输出值。

## 数据集

随着挖矿型恶意代码数量爆发式增长，越来越多的个人用户、企业用户受到感染，沦为免费矿工。本课题所用的数据集来自奇安信科技集团股份有限公司从现网捕获的大量挖矿型恶意代码和非挖矿型恶意代码，经过数据清洗后从中提取大小在20KB至10MB之间的样本，并通过代码相似性分析等方法去掉过于相似的样本，保证样本具有一定的多样性。

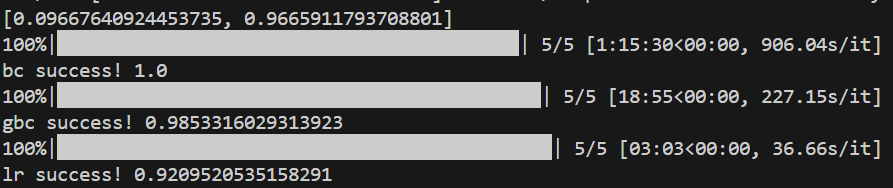
未避免样本运行，样本PE结构中的MZ头、PE头、导入导出表等区域均已被抹去。清洗后的样本虽然无法动态分析，但其挖矿功能的代码指令特征依然存在。

[1] DataCon社区.DataCon开放数据集-DataCon2020-恶意代码数据集方向开放数据[DB/OL].2021-11-11.https://datacon.qianxin.com/opendata/openpage?resourcesId=5



## 基模型一层集成提升

对于基模型而言，由于特征较充足，模型拟合效果总体良好，个别模型准确率较低：



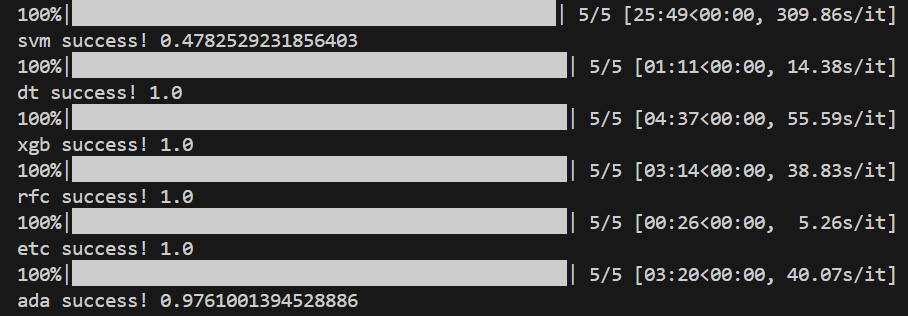


图 基模型拟合效果

可以看到score值为100分：

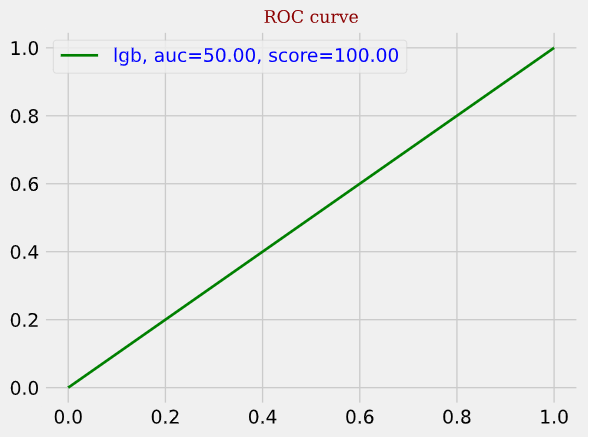


图 基模型ROC曲线

## 基模型二层集成提升

对于神经网络模型而言，可以看到在样本数量为100时，随着迭代次数的增加，平均损失不断降低，准确度达到了95.06%

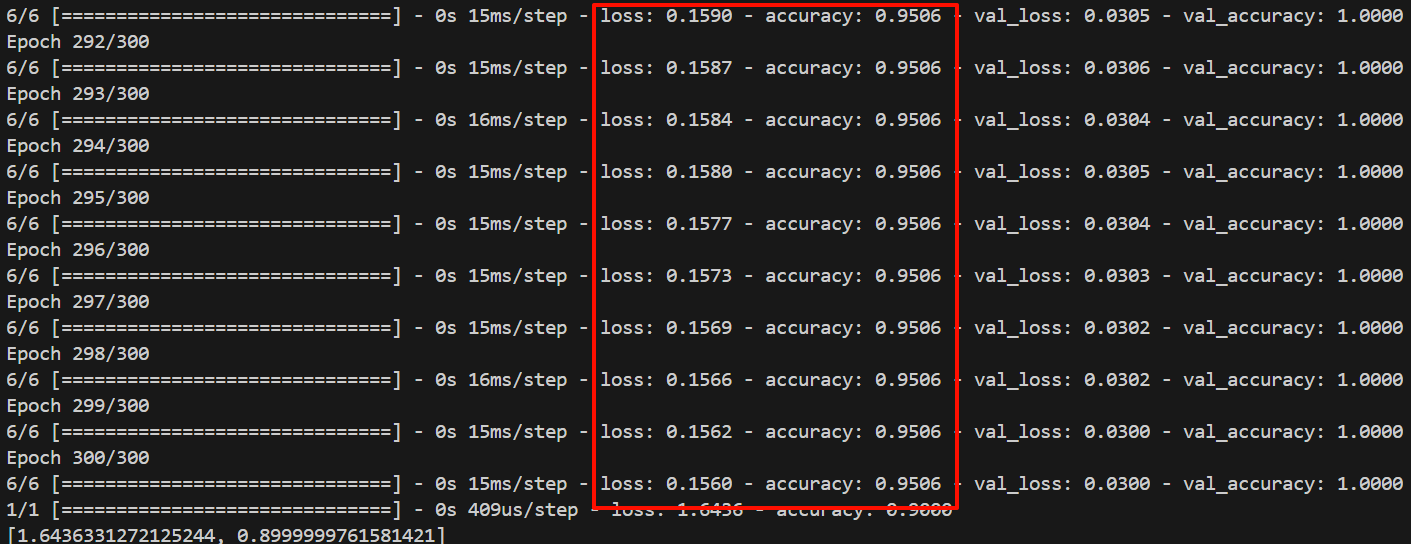


图 深度学习基模型准确度（样本数=100）

而样本数量达到1万+时，迭代次数达到了894次，模型准确度达到97.15%，相较之前更加高。

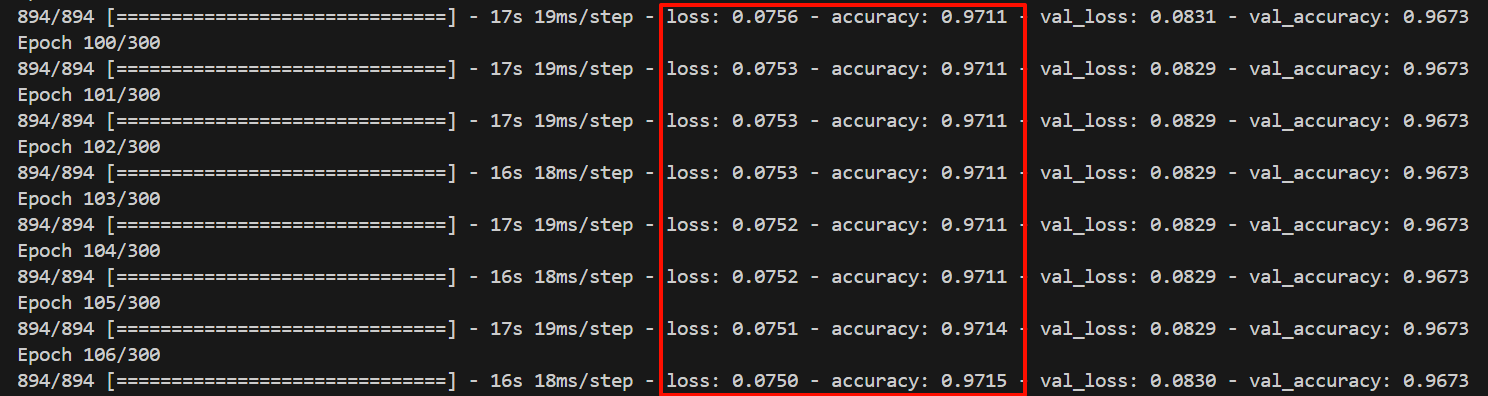


图 深度学习基模型准确度（样本数=17657）

## 元模型加权融合提升

将基模型融合、压缩后，可以得到堆叠后的元模型，其拟合效果相较基模型而言泛化性更好：

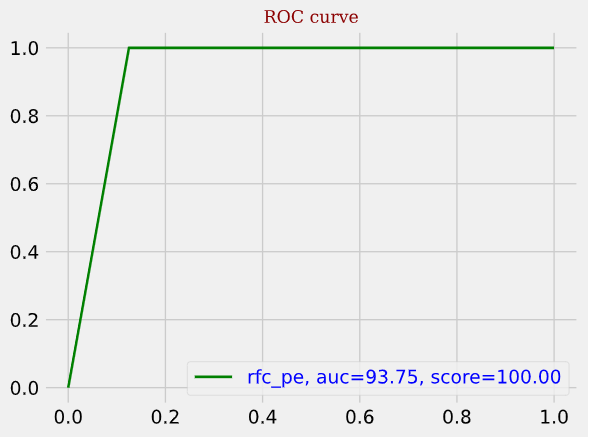


图 7-5 随机森林元模型ROC曲线

对于逻辑回归模型，针对二分类问题也有良好的拟合结果：

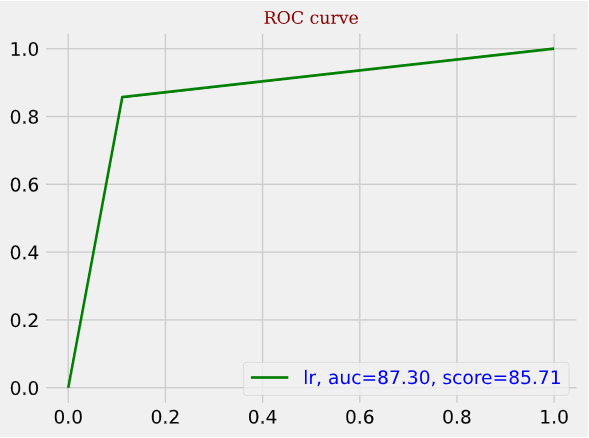


图 7-6 逻辑回归模型ROC曲线

## 模型评估

最终训练好的模型，经过5折交叉验证有97.23%的准确度，对于50个样本预测情况，其结果全部正确，准确率达100%

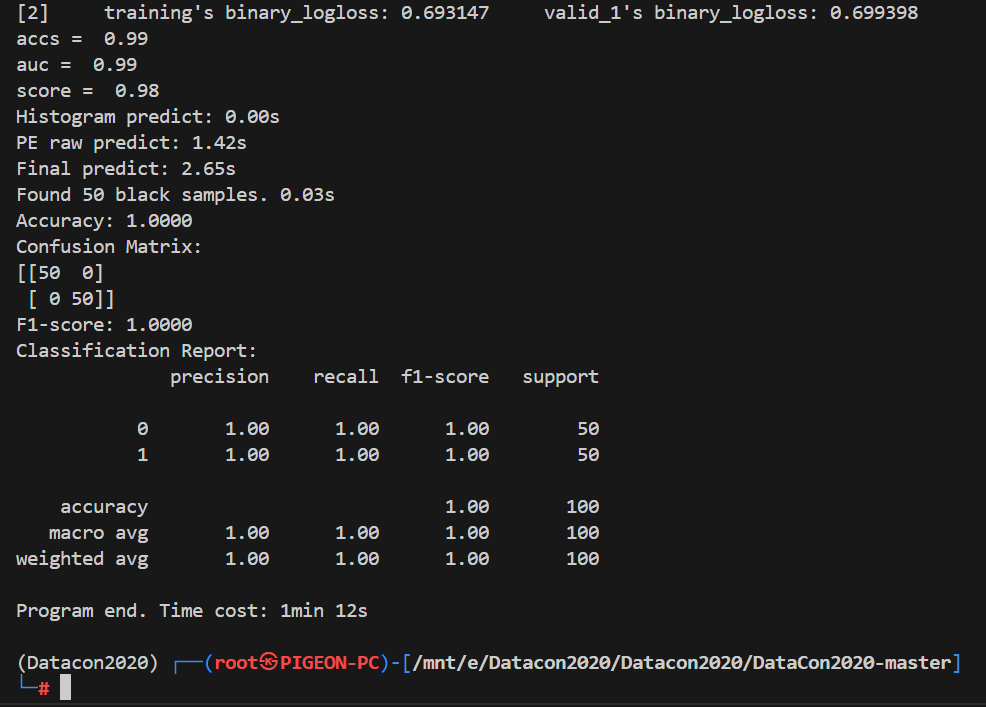


图 模型评估结果（样本数=100）

对于17657个样本训练的模型，训练的准确率达到98.61%，对于预测结果而言，仅存在2个假阳性样本，准确率达99.99%，F1-score达99.98%。

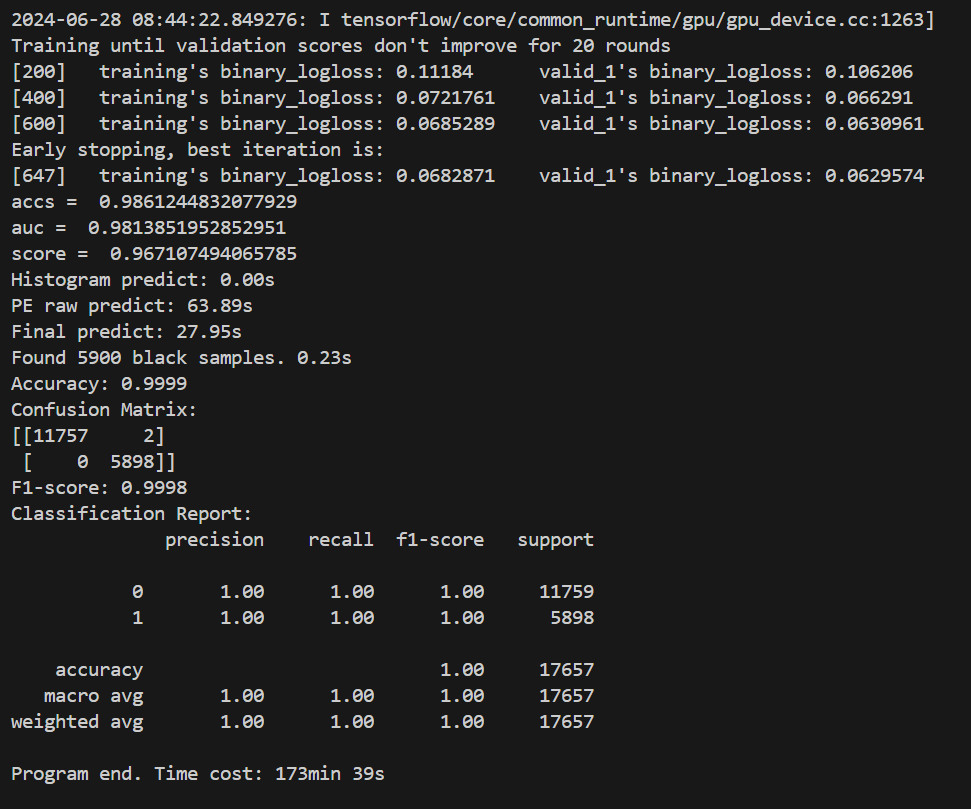


图 模型评估结果（样本数=17657）

# 总结与展望

致谢

陈思远老师是我这次毕业设计的导师。自大三暑假我参与专家系统课题以来，他便经常讲解设计思路，宏观上帮助我把握好设计方向；微观上提供各种专业知识方面的咨询，并协助我查找了大量的资料，而且给我提供了先进的仪器设备及舒适的工作环境，解决了大量的技术难题，同时为我的软件测试提供了很好的平台，我所有成绩的取得和陈老师悉心指导是分不开的。

参考文献

1. Malware Statistics. AV-TEST. https://www.av-test.org/en/statistics/malware/. Feb. 2020.
2. Christodorescu M, Jha S. Static Analysis of Executables to Detect Malicious Patterns[C]. The 12th conference on USENIX Security Symposium - Volume 12, 2003: 12.
3. Christodorescu M, Jha S, Seshia S A, et al. Semantics-Aware Malware Detection[C]. 2005 IEEE Symposium on Security and Privacy, 2005: 32-46.
4. Zhang Q H, Reeves D S. MetaAware: Identifying Metamorphic Malware[C]. Twenty-Third Annual Computer Security Applications Conference, 2008: 411-420.
5. Schultz M G, Eskin E, Zadok F, et al. Data Mining Methods for Detection of New Malicious Executables[C]. Proceedings 2001 IEEE Symposium on Security and Privacy. S&P, 2002: 38-49.
6. Jurafsky D. Speech and language processing[M]. Pearson Education India, 2000.
7. Malone C, Zahran M, Karri R. Are Hardware Performance Counters a Cost Effective Way for Integrity Checking of Programs[C]. The sixth ACM workshop on Scalable trusted computing, 2011:71-76.
8. Xia Y B, Liu Y T, Chen H B, et al. CFIMon: Detecting Violation of Control Flow Integrity Using Performance Counters[C]. IEEE/IFIP International Conference on Dependable Systems and Networks, 2012: 1-12.
9. Demme J, Maycock M, Schmitz J, et al. On the Feasibility of Online Malware Detection with Performance Counters[J]. ACM SIGARCH Computer Architecture News, 2013, 41(3): 559-570.
10. Sayadi H, Mohammadi Makrani H, Randive O, et al. Customized Machine Learning-Based Hardware-Assisted Malware Detection in Embedded Devices[C]. 2018 17th IEEE International Conference on Trust, Security and Privacy In Computing and Communications/12th IEEE International Conference on Big Data Science and Engineering, 2018: 1685-1688.
11. Yerima S Y, Sezer S, Muttik I. High Accuracy Android Malware Detection Using Ensemble Learning[J]. IET Information Security, 2015, 9(6): 313-320.
12. Khasawneh K N, Ozsoy M, Donovick C, et al. EnsembleHMD: Accurate Hardware Malware Detectors with Specialized Ensemble Classifiers[J]. IEEE Transactions on Dependable and Secure Computing, 2020, 17(3): 620-633.