**微软恶意代码分类系统开发实践**

**开发文档**

[1 功能概述及使用说明 3](#_Toc493021234)

[1.1 背景 3](#_Toc493021235)

[1.2 功能 3](#_Toc493021236)

[1.3 方法简述 4](#_Toc493021237)

[1.4 环境搭建 5](#_Toc493021238)

[1.4.1 TensorFlow平台 5](#_Toc493021239)

[1.4.2 神经网络模型 9](#_Toc493021240)

[1.5 使用说明 9](#_Toc493021241)

[2 系统设计概述 9](#_Toc493021242)

[2.1架构设计 10](#_Toc493021243)

[2.1.1 系统架构设计 10](#_Toc493021244)

[2.1.2 硬件系统平台 12](#_Toc493021245)

[2.2 系统需求分析与设计 12](#_Toc493021246)

[2.2 系统测试计划 13](#_Toc493021247)

[2.2.1 目的 13](#_Toc493021248)

[2.2.2 范围 13](#_Toc493021249)

[2.2.3 测试进度 13](#_Toc493021250)

[2.2.4 测试质量目标 14](#_Toc493021251)

[2.2.5 功能测试 15](#_Toc493021252)

[2.2.6 测试风险管理 15](#_Toc493021253)

[2.3 系统的编码 16](#_Toc493021254)

[2.4系统实施方案 16](#_Toc493021255)

[2.4.1 进度计划 16](#_Toc493021256)

[3 开发过程概述 17](#_Toc493021257)

[3.1 系统实现 17](#_Toc493021258)

[3.1.1 灰度图绘制 17](#_Toc493021259)

[3.1.2 GIST特征 19](#_Toc493021260)

[3.1.3 卷积神经网络模型 19](#_Toc493021261)

[3.1.4 SVM、随机森林、GDBT和逻辑回归等其他分类算法 20](#_Toc493021262)

[3.2 开发代码详解 20](#_Toc493021263)

[3.3 运行结果比较分析 21](#_Toc493021264)

## 1 功能概述及使用说明

### 1.1 背景

80年代末期，随着恶意代码的诞生，­­­­反恶意代码软件（或称反病毒软件）随之诞生。这个时期的恶意代码所采用的技术较为简单，这使得对应的检测技术也较为容易，早期的反病毒软件大都单一的采用特征匹配的方法，简单的利用特征串完成检测。随着恶意代码技术的发展，恶意代码开始在传播过程中进行变形以躲避查杀，此时同一个恶意代码的变种数量急剧提升，形态较本体也发生了较大的变化，反病毒软件已经很难提取出一段代码作为恶意代码的特征码。在这种情况下，广谱特征码随之诞生，广谱特征码将特征码进行了分段，通过掩码字节对需要进行比较的和不需要进行比较的区段进行划分。然而无论是特征码扫描还是广谱特征，都需要在获得恶意代码样本后，进行特征的提取，随后才能进行检测，这使得对恶意代码的查杀具有一定的滞后性，始终走在恶意代码的后面。为了针对变种病毒和未知病毒，启发式扫描应运而生，启发式扫描利用已有的经验和知识对未知的二进制代码进行检测，这种技术抓住了恶意代码具有普通二进制文件所不具有的恶意行为，例如非常规读写文件，终结自身，非常规切入零环等等。启发式扫描的重点和难点在于如何对恶意代码的恶意行为特征进行提取。特征码扫描、查找广谱特征、启发式扫描，这三种查杀方式均没有实际运行二进制文件，因此均可归为恶意代码静态检测的方法。随着反恶意代码技术的逐步发展，主动防御技术、云查杀技术已越来越多的被安全厂商使用，但恶意代码静态检测的方法仍是效率最高，被运用最广泛的恶意代码查杀技术。

### 1.2 功能

实现了基于Tensorflow平台的恶意代码的检测，具体是分别对老师提供的803个恶意样本的.asm和.bytes文件进行特征提取，然后构建两层神经网络模型，训练出分类器，用来对恶意代码进行分类，同时综合运用如决策树、逻辑回归和GBDT（梯度提升树）等算法实现分类器训练和样本预测。

值得说明的是：（1）我们的检测主要是对恶意代码的分类，不是病毒查杀。（2）采用的方法是纯静态分析的方法，不涉及行为分析等动态分析方法。因此这不意味着这个方法能够取代现有的方法，但是了解它能够为安全研究人员提供一个崭新的思路，至于能否在工业界应用仍待进一步研究。

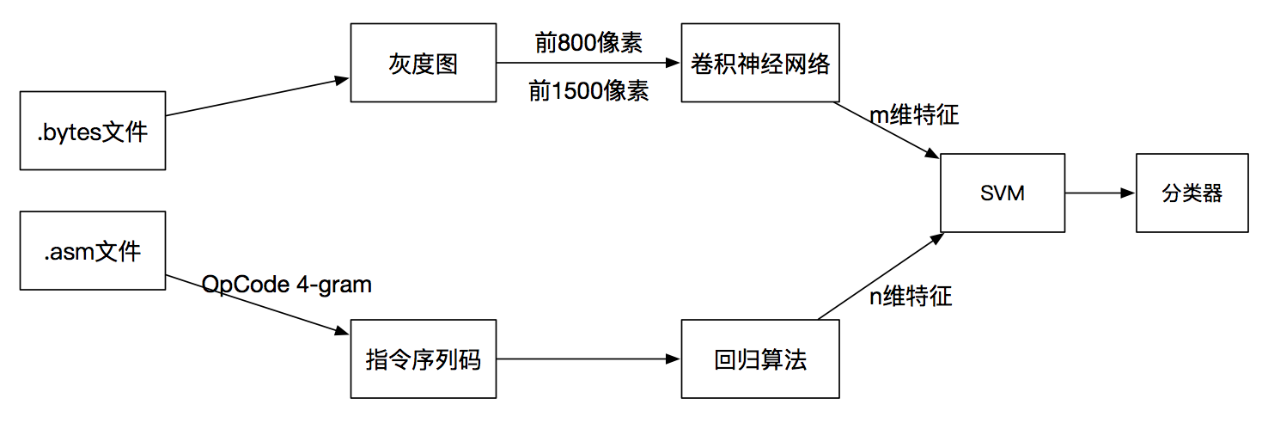
### 1.3 方法简述

我们分别对提供的.bytes文件和.asm文件，提取它的恶意代码图像、OpCode n-gram这两个“黄金”特征，机器学习部分采用的是基于TensorFlow平台的卷积神经网络和线性回归两种算法。具体流程如下图：

随机森林

GBDT

逻辑回归



决策树

两层神经网络

OpCode 2-gram

OpCode 3-gram

前2500像素

（1）恶意代码图像：

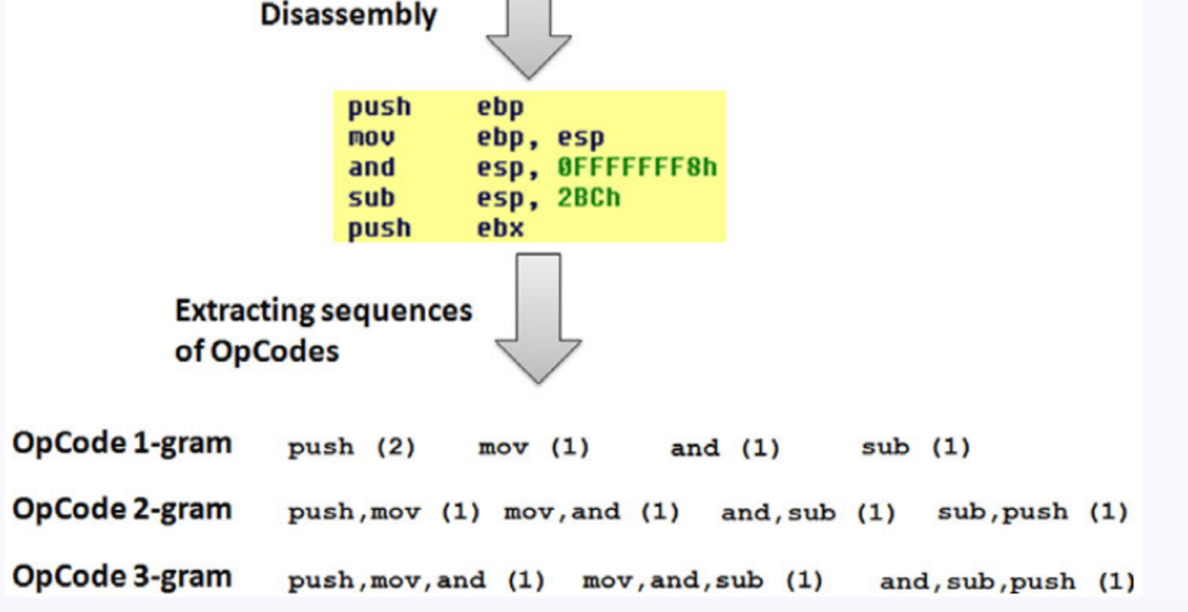
①恶意代码图像绘制方法：对一个二进制文件，每个字节范围在00~FF之间，刚好对应灰度图0~255(0为黑色，255为白色)。将一个二进制文件转换为一个矩阵(矩阵元素对应文件中的每一个字节，矩阵的大小可根据实际情况进行调整)，该矩阵又可以非常方便的转换为一张灰度图。因此我们很容易就把恶意程序的.bytes文件转化为相应的灰度图。

②图像特征提取：同一个家族的恶意代码图像在纹理上存在一定的相似性，不同的恶意代码家族是有一定区别的。如何用计算机发现和提取这些纹理的相似特征用以分类呢？我们采用的是构建在TensorFlow平台下的卷积神经网络分别对①中生成的灰度图的前800个像素、1500个像素和2500个像素进行特征提取（采用800、1500和2500个像素是为了形成对比，寻找最合适的范围）。

（2）OpCode 2-gram 、OpCode 3-gram 和OpCode 4-gram:

①n-gram：是自然语言处理领域的概念，早期的语音识别技术和统计语言模型与它密不可分。n-gram基于一个简单的假设，即认为一个词出现的概率仅与它之前的n-1个词有关，这个概率可从大量语料中统计得到。例如“吃”的后面出现“苹果”或“披萨”的概率就会比“公路”的概率大(正常的语料中基本不会出现“吃公路”这种组合)，可以看出n-gram在一定程度上包含了部分语言特征。

②：特征提取：我们直接对给的.asm文件，用python提取出其n-gram特征，过程如下：



### 1.4 环境搭建

#### 1.4.1 ­­­TensorFlow平台

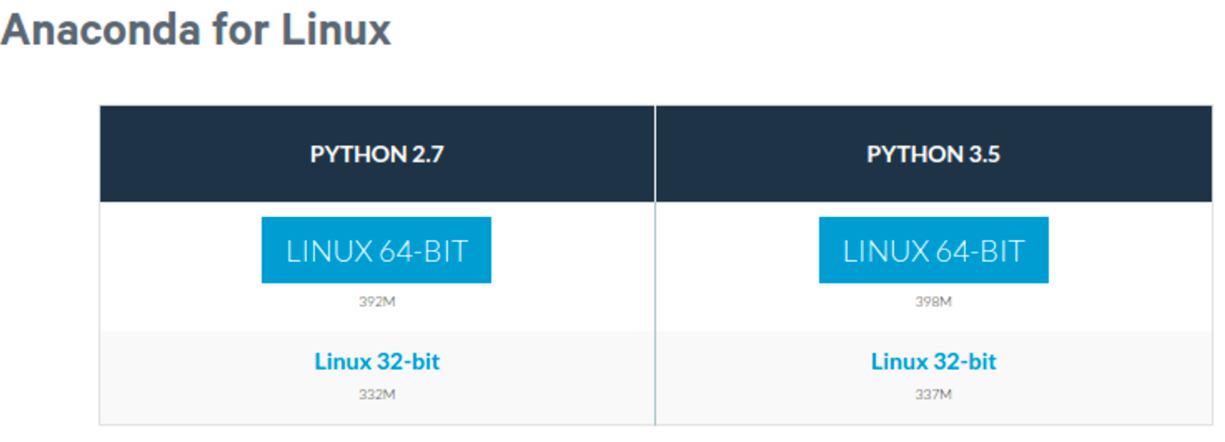
TensorFlow是谷歌基于DistBelief进行研发的第二代人工智能学习系统，其命名来源于本身的运行原理。Tensor(张量)意味着N维数 组，Flow(流)意味着基于数据流图的计算 ，TensorFlow为张量从图象的一端流动到另 一端计算过程。TensorFlow是将复杂的数据 结构传输至人工智能神经网中进行分析和处 理过程的系统。

TensorFlow可被用于语音识别或图像识别等 多项机器深度学习领域，对2011年开发的深度学习基础架构DistBelief进行了各方面的改 进，它可在小到一部智能手机、大到数千台数据中心服务器的各种设备上运行。

我们组配置的是CPU版本的TensorFlow，原因是：图像提取特征，可以通过GPU进行加速，而GPU版本的TensorFlow要求是英伟达的显卡，而我们全组的电脑都不是。然而我们的训练集只有800多个样本，用CPU版本加速，耗时并没有增加多少。

我们配置的系统是：Ubuntu，配置过程参考了谷歌官方教程（链接：<https://www.tensorflow.org/install/install_linux>），我们是：基于Anaconda的tensorflow安装，具体过程如下：

1. **下载linux版本的Anaconda安装包**



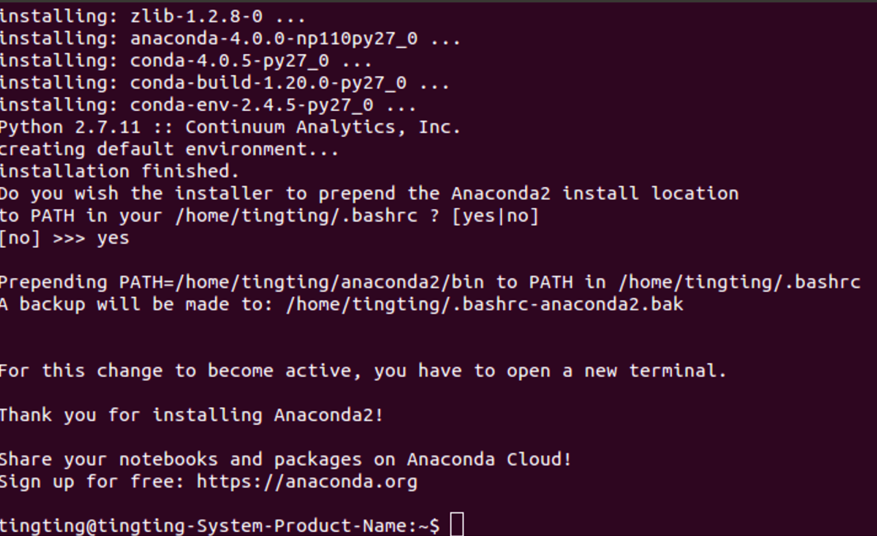
下载地址：<https://www.continuum.io/downloads>   
 这里选择64位linux的Python 2.7版本，点击下载，下载完成后，得到Anaconda2-4.0.0-Linux-x86\_64.sh安装文件

1. **安装Anaconda**

打开terminal,输入如下命令,然后回车：  
 bash /home/tingting/Downloads/Anaconda2-4.0.0-Linux-x86\_64.sh

（这里的/home/tingting/Downloads/是存放Anaconda2-4.0.0-Linux-x86\_64.sh的路径）

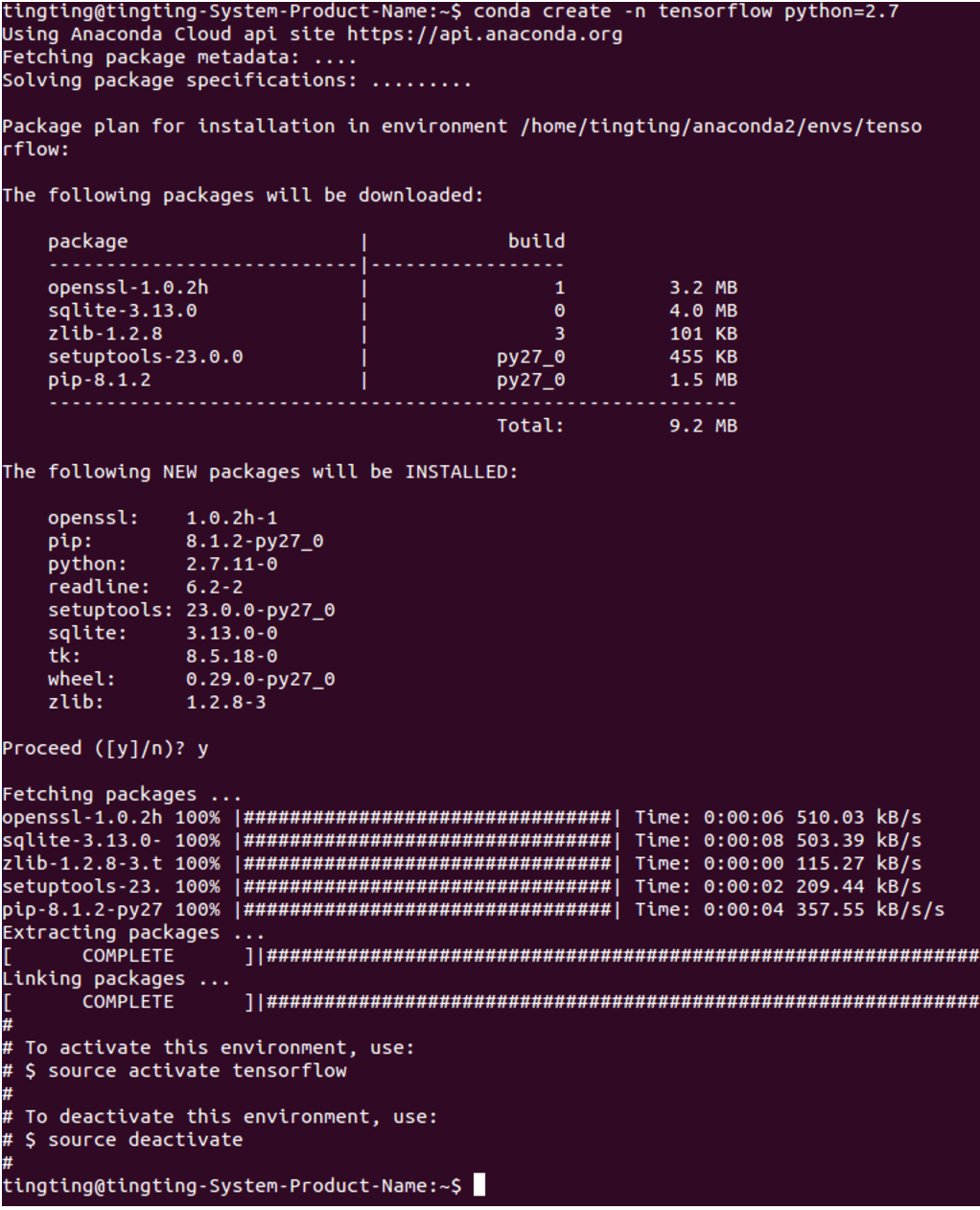
安装完成：



1. **利用anaconda安装tensorflow**

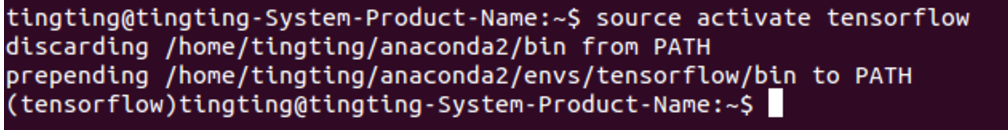
①建立一个 conda 计算环境

输入如下命令：conda create -n tensorflow python=2.7



②激活环境,使用 conda 安装 TensorFlow

输入如下命令：source activate tensorflow

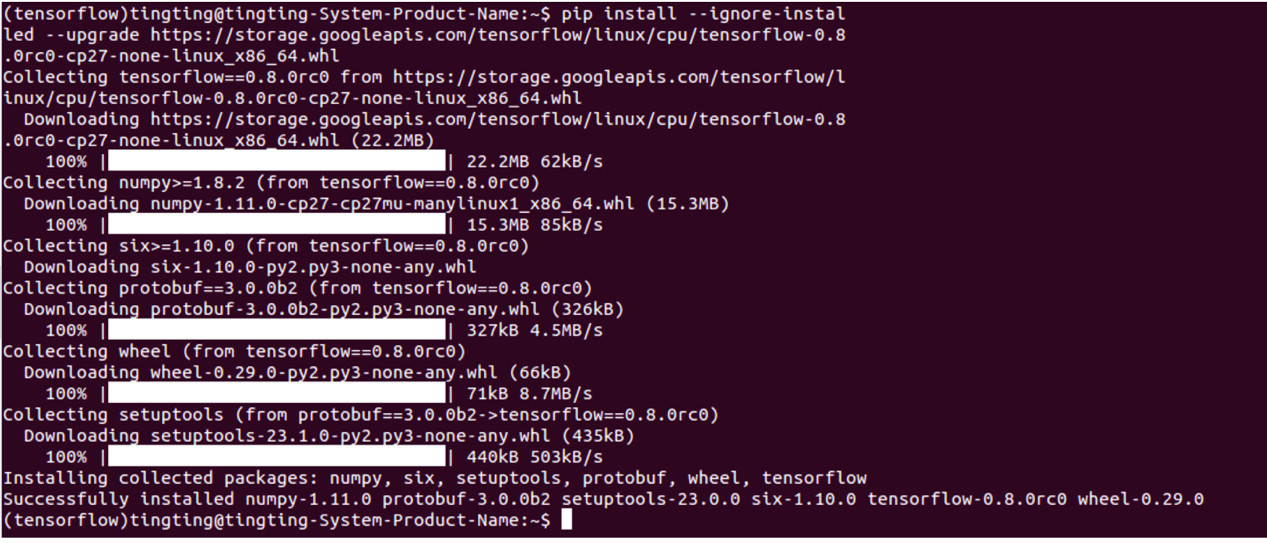


1. **安装tensorflow**

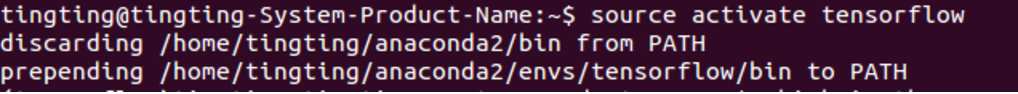
①安装：

输入如下命令：

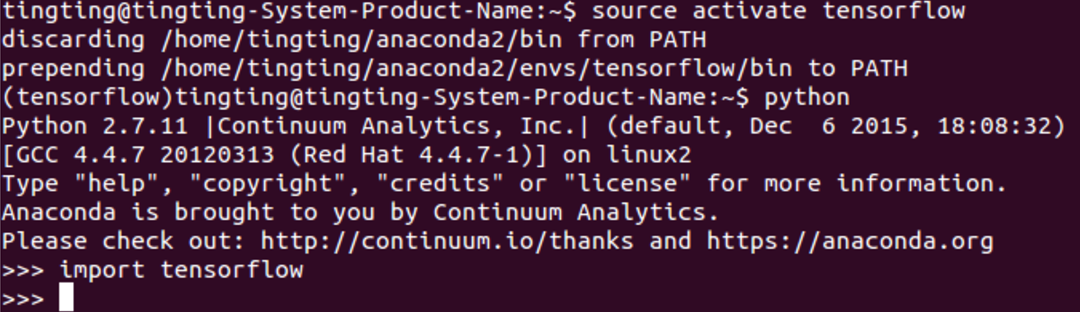
pip install --ignore-installed –upgrade <https://storage.googleapis.com/tensorflow/linux/cpu/tensorflow-0.8.0rc0-cp27-none-linux_x86_64.whl>



**②**安装成功后,每次使用 TensorFlow 的时候需要激活 conda 环境,可以看到,在正常情况下,是anaconda的bin路径在环境变量中,但激活conda-tensorflow环境后,环境变量中存储的是tensorflow下的bin路径



③测试tensorflow是否安装成功   
I.激活conda环境   
II.进入python   
III.import tensorflow   
整个过程比较顺利,tensorflow被成功import了



#### 1.4.2 神经网络模型

模拟人类实际神经网络的[数学方法](https://baike.baidu.com/item/%E6%95%B0%E5%AD%A6%E6%96%B9%E6%B3%95)问世以来，人们已慢慢习惯了把这种[人工神经网络](https://baike.baidu.com/item/%E4%BA%BA%E5%B7%A5%E7%A5%9E%E7%BB%8F%E7%BD%91%E7%BB%9C)直接称为神经网络。神经网络在[系统辨识](https://baike.baidu.com/item/%E7%B3%BB%E7%BB%9F%E8%BE%A8%E8%AF%86)、[模式识别](https://baike.baidu.com/item/%E6%A8%A1%E5%BC%8F%E8%AF%86%E5%88%AB)、智能控制等领域有着广泛而吸引人的前景，特别在智能控制中，人们对神经网络的自学习功能尤其感兴趣，并且把神经网络这一重要特点看作是解决自动控制中控制器适应能力这个难题的关键钥匙之一。

神经网络模型是以神经元的数学模型为基础来描述的。人工神经网络(ArtificialNuearlNewtokr)s,是对人类大脑系统的一阶特性的一种描。简单地讲,它是一个数学模型。神经网络模型由网络拓扑．节点特点和学习规则来表示。神经网络对人们的巨大吸引力主要在下列几点：

1．并行分布处理。

2．高度鲁棒性和容错能力。

3．分布存储及学习能力。

4．能充分逼近复杂的非线性关系。

在控制领域的研究课题中，不确定性系统的控制问题长期以来都是控制理论研究的中心主题之一，但是这个问题一直没有得到有效的解决。利用神经网络的学习能力，使它在对不确定性系统的控制过程中自动学习系统的特性，从而自动适应系统随时间的特性变异，以求达到对系统的最优控制；显然这是一种十分振奋人心的意向和方法

### 1.5 使用说明

当安装完Tensorflow之后，使用PyCharm编写python代码时，会自动关联到用到的库，可以直接按照自己的意愿进行代码书写。

## 2 系统设计概述

Windows恶意代码分类系统是基于静态恶意代码分析的基本理论，以及两层神经网络模型、SVM、随机森立、GDBT等分类算法所得出的实践性内容。主要介绍如何将恶意代码的特征选择方法并利用其特征进行分类进行系统化实现。使用两层神经网络模型来进行灰度图的特征提取并分类，同是采纳SVM等分类方法进行特征分类，并与两层神经网路模型形成对比进行选取最优分类算法，从而实现对灰度图的分类，以此为依据实现对windows恶意代码的分类。

### 2.1架构设计

#### 2.1.1 系统架构设计

windows恶意代码分类系统的实现分为两个核心方面以及五个阶段。两个核心内容分别为恶意代码的特征及特征选择的方法和不同分类算法的选择与调优。前者是不同的特征描述的是恶意代码不同的层面的信息，特征描述能力是否全面是恶意代码能否正确分类的前提；后者是不同的分类算法其分类效率与精度也有所不同，其泛化能力也各有不同。不过这两个方面的目标都是提高恶意代码检测的准确率。windows恶意代码分类系统的实现的五个阶段分别为特征提取、特征处理、特征降维、分类学习、对测试集进行分析与评估。

首先是特征提取阶段，经过查找资料以及参考相关文件，我们采用.asm文件的前1500像为基本，提取出恶意代码的“黄金”特征——恶意代码图像。这个概念最早是2011年由加利福尼亚大学的Nataraj和Karthikeyan在他们的论文 Malware Images: Visualization and Automatic Classification 中提出来的，思路非常新颖，把一个二进制文件以灰度图的形式展现出来，利用图像中的纹理特征对恶意代码进行聚类。此后，有许多研究人员在这个思路基础上进行了改进和探索。就目前发表的文章来看，恶意代码图像的形式并不固定，研究人员可根据实际情况进行调整和创新。至于国内，对这方面的研究还较少，相关论文也比较少见。

接下来是特征处理阶段，根据前1500像素提取上文提及到恶意代码图像。对于恶意代码图像的绘制，首先对一个二进制文件，每个字节范围在00~FF之间，刚好对应灰度图0~255(0为黑色，255为白色)。将一个二进制文件转换为一个矩阵(矩阵元素对应文件中的每一个字节，矩阵的大小可根据实际情况进行调整)，该矩阵又可以非常方便的转换为一张灰度图。需要我们处理的是，先对提供的.asm文件提取前1500像素值，然后转换为二进制文件，对此绘制灰度图。

在绘制灰度图后采用卷积神经网络模型对其进行卷积，提取特征向量。同时为了降低计算维度，本文将针对关键特征向量的重要程度对特征向量进行排序。

提取后的特征向量将采用两种分类算法来进行对比，从而取最优。首先采用卷积神经网络模型自带的分类算法对所得到的特征向量进行分类对比，输出相应的灰度图的分类结果。另一类分类算法则采用SVM分类算法进行特征向量的分类对比。通过训练集的进行分类训练，用以生成分类器。通过分类器可以将测试机的恶意代码进行分类输出。

最后通过对两种算法的输出结果进行对比，采取最优算法作为分类器，实现对测试集的分类。

 图2.1 系统整体流程图

对于实际操作阶段，在数据流上前期和训练阶段一样，需要进行特征提取、特征表示、最后使用训练好的分类器模型进行分类。最后将分类结果与人工分类结果进行拟合判断，以验证系统的有效性。

#### 2.1.2 硬件系统平台

（1） 硬件环境

操作系统 Ubuntu 14.0

处理器 英特尔 第四代酷睿 i5-4210U 双核

主板 联想 Erazer Y4 ( 英特尔 Haswell-ULT - Lynx Point-LP )

内存 8 GB ( 三星 DDR3L 1600MHz / 海力士 DDR3L 1600MHz )

主硬盘 西数 WDC WD10SPCX-24HWST0 ( 1 TB / 5400 转/分 )

显卡 AMD Radeon R9 M275X ( 2 GB )

显示器 友达 AUO143D ( 14 英寸 )

声卡 瑞昱 @ 英特尔 Lynx Point-LP 高保真音频

网卡 瑞昱 RTL8168/8111/8112 Gigabit Ethernet Controller / 联想

（2） 开发环境

Tensorflow 1.2版本

Python 2.7.3

（3） 开发平台

Pycharm 2016.1.2

### 2.2 系统需求分析与设计

恶意代码分类系统的两个核心内容分别为恶意代码的特征及特征选择的方法和不同分类算法的选择与调优。五个重要阶段是特征提取、特征处理、特征降维、分类学习、对测试集进行分析与评估。根据数据流的先后顺序，本文系统首先会对待测程序进行特征提取，包括提取前1500像素作为提取点，根据其特征转化二进制文件，绘制灰度图。第二步是对灰度图进行特征处理，使用卷积神经网络模型对其提取特征向量。第三步对特征表示，根据生成的特征字典生成相应的特征表示文本，为下一步分类做准备。恶意代码分类阶段，本文采用两种分类算法一种是基于卷积神经网络模型的分类算法，另一种是基于SVM等不同分类算法，我们在不同的分类算法进行选择与调优。

### 2.2 系统测试计划

#### 2.2.1 目的

本文主要阐述“基于两层神经网络的恶意代码分类系统”的测试初步计划，并为“基于卷积神经网络的恶意代码分类系统”的测试工作提供一个框架和规范：

（1）确定项目的策略、范围和方法；

（2）使项目测试工作的所有参与人员（客户方参与人员、测试管理者、测试人员）对本项目测试的目标、范围、策略、方法、组织、资源等有一个清晰的认识；

（3）从策略角度说明本项目测试的组织和管理，知道测试进展，并作为项目测试工作实施的依据。

本测试计划的预期读者主要有两类受众：测试管理人员（项目经理、客户参与人员）和测试人员。

（1）项目经理根据该测试大纲指定进一步的计划、安排（工作任务分配、时间进度安排）和控制测试过程。

（2）客户参与人员通过该测试大纲了解测试过程和相关信息。

（3）测试人员根据该测试大纲中指定的岗位、方法确定测试需求、设计测试用例、执行和记录测试过程并记录和报告缺陷。

#### 2.2.2 范围

本测试计划是针对“基于卷积神经网络的恶意代码分类系统”所提供的功能点以及项目文档中的内容的测试计划，包括：

卷积神经网络模型的运行

SVM等算法模型的运行

恶意代码的分类结果

#### 2.2.3 测试进度

|  |  |  |  |
| --- | --- | --- | --- |
| 测试活动 | 开始时间 | 结束时间 | 测试人员 |
| 制定测试计划 | 2017.8.29 | 2017.8.29 | 小组全体 |
| 环境搭建 | 2017.8.29 | 2017.8.30 | 小组全体 |
| 卷积神经网络测试 | 2017.9.4 | 2017.9.6 | 秦勇 |
| SVM算法结果测试 | 2017.9.8 | 2017.9.10 | 李斌斌 |
| 恶意代码分类结果测试 | 2017.9.9 | 2017.9.10 | 胡鹏、莫开胜 |
| 文档测试 | 2017.9.9 | 2017.9.10 | 杨晓雷 |

#### 2.2.4 测试质量目标

|  |  |
| --- | --- |
| 测试质量目标 | 确认者 |
| 所有的测试案例已经执行 | 杨晓雷 |
| 所有的自动测试脚本已经执行通过 | 杨晓雷 |
| 所有的重要等级为1/2的BUG已经解决并由测试验证 | 杨晓雷 |
| 每一部分的测试已经被TestLead确认完成 | 杨晓雷 |
| 重要的功能不允许有等级为1/2/3的BUG | 胡鹏 |
| 一般的功能或与最终使用者不直接联系的功能不允许有等级为1/2的BUG，且BUG等级为3的问题不得超过1/3功能 | 胡鹏 |
| 轻量的功能允许有少量2/3等级的错误 | 胡鹏 |
| 发现错误等级为1/2/3的BUG的速率正在下降并接近0 | 胡鹏 |
| 在最后的三天内没有发现错误等级为1/2/3类的BUG | 胡鹏 |

#### 2.2.5 功能测试

|  |  |
| --- | --- |
| 测试目标 | 确保测试的功能正常，能满足pdf中所提及的要求 |
| 测试范围 | 整个恶意代码分类系统所实现的功能 |
| 技术 | 利用测试集来执行该系统其功能以核实一下内容；  输入恶意代码，得到预期的结果；  各业务规则都得到正确的应用 |
| 开始标准 | 列出测试计划，并发现缺陷 |
| 完成标准 | 所有计划的测试已全部执行  所有发现的缺陷已全部解决 |
| 测试重点和优先级 | 首先测试整个系统能够正常运行，然后测试系统的功能、性能以及输入输出的要求。 |
| 需要考虑的特殊事项 | 确定或说明那些将对功能测试的实施和执行造成影响的事项或因素（内部的或外部的） |

#### 2.2.6 测试风险管理

|  |  |  |
| --- | --- | --- |
| 序号 | 风险名称 | 风险控制措施 |
| 1 | 数据丢失 | 及时备份 |
| 2 | 设备故障 | 所准备几台电脑 |
| 3 | 网络故障 | 去教室连无线 |
| 4 | 测试工作不全面 | 补全测试计划 |
| 5 | 测试进度出现危机 | 预算进度时留下周旋时间 |
| 6 | 测试进度不统一 | 定期小组会议 |
| 7 | 测试时间不充足 | 合理安排测试计划，留给其他项目进行时间 |
| 8 | 某项测试人员特殊原因进度暂停 | 其他测试人员分担测试内容 |

### 2.3 系统的编码

在软件编码阶段，开发者根据《软件系统详细设计报告》中对数据结构、算法分析和模块实现等方面的设计要求，开始具体的编写程序工作，分别实现各模块的功能，从而实现对目标系统的功能、性能、接口、界面等方面的要求。

软件编码的要求：

a. 模块化编码

b. 代码可读性

c. 可维护性

d. 模块接口标准化

e. 界面风格统一

f. 注释的应用。

### 2.4系统实施方案

#### 2.4.1 进度计划

|  |  |
| --- | --- |
| 时间规划 | 目标计划 |
| 8.29-9.1 | 阅读相关资料，了解项目 |
| 9.2-9.5 | 配置项目环境，继续学习深度学习相关知识 |
| 9.6-9.8 | 配置运行tensorflow |
| 9.8-9.12 | 编写开发文档，测试系统，完成系统必要的功能 |
| 8.13-9.15 | 修改bug，完善系统，完成开发文档 |

## 3 开发过程概述

### 3.1 系统实现

#### 3.1.1 灰度图绘制

在整个过程中灰度图（即恶意代码图像）是整个系统非常关键的一步，也是整个系统设计的开端。

绘制方法如下，对一个二进制文件，每个字节范围在00~FF之间，刚好对应灰度图0~255(0为黑色，255为白色)。将一个二进制文件转换为一个矩阵(矩阵元素对应文件中的每一个字节，矩阵的大小可根据实际情况进行调整)，该矩阵又可以非常方便的转换为一张灰度图。

对于给出的.asm文件，先转化为二进制文件，通过恶意代码图像绘制，得到灰度图。

pythion代码如下：

import os

import numpy

from collections import \*

import pandas as pd

import binascii

def getMatrixfrom\_bin(filename, width = 512, oneRow = False):

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

content = f.read()

hexst = binascii.hexlify(content)

fh = numpy.array([int(hexst[i:i+2],16) for i in range(0, len(hexst), 2)])

if oneRow is False:

rn = len(fh)/width

fh = numpy.reshape(fh[:rn\*width],(-1,width))

fh = numpy.uint8(fh)

return fh

def getMatrixfrom\_asm(filename, startindex = 0, pixnum = 5000):

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

f.seek(startindex, 0)

content = f.read(pixnum)

hexst = binascii.hexlify(content)

fh = numpy.array([int(hexst[i:i+2],16) for i in range(0, len(hexst), 2)])

fh = numpy.uint8(fh)

return fh

def getMatrixfrom\_hex(filename, width):

hexar = []

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

for line in f:

hexar.extend(int(el,16) for el in line.split()[1:] if el != "??")

rn = len(hexar)/width

fh = numpy.reshape(hexar[:rn\*width],(-1,width))

fh = numpy.uint8(fh)

return fh

def read\_hexbytes(filename):

hexar = []

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

for line in f:

hexar.extend(int(el,16) for el in line.split()[1:] if el != "??")

rn = len(hexar)/256

fh = numpy.reshape(hexar[:rn\*256],(-1,256))

fh = numpy.uint8(fh)

return fh

basepath = "/home/qinyong/subtrain/"

mapimg = defaultdict(list)

subtrain = pd.read\_csv('subtrainLabels.csv')

i = 0

for sid in subtrain.Id:

i += 1

print "dealing with {0}th file...".format(str(i))

filename = basepath + sid + ".asm"

im = getMatrixfrom\_asm(filename, startindex = 0, pixnum = 2500)

mapimg[sid] = im

dataframelist = []

for sid,imf in mapimg.iteritems():

standard = {}

standard["Id"] = sid

for index,value in enumerate(imf):

colName = "pix{0}".format(str(index))

standard[colName] = value

dataframelist.append(standard)

df = pd.DataFrame(dataframelist)

df.to\_csv("imgfeature\_2500.csv",index=False)

#### 3.1.2 GIST特征

使用上图代码就可以绘制出恶意代码图像——灰度图，但是我们从肉眼上可以观察出哪些代码在图像特征上有类似的特征，哪些代码具有一定的区别。但是我们需要教会计算机来完成这一步，这就需要利用计算机视觉里面的一些技术。其中最为主要的是在Nataraj和Karthikeyan的论文中采用的GIST特征。对灰度图的GIST描述进行数值化，采用一个向量来表示，便于之后的特征字典的生成和分类器的训练分类。

#### 3.1.3 卷积神经网络模型

在使用卷积神经网络之前需要先对图像进行预处理，由于图像的大小直接关系到卷积核的选取，因此不宜太大；如果图像太小，又会引起图像关键细节的丢失。一般选取256\*256像素作为图像的标准大小，由于我们采用了.asm的生成灰度图，所以在这一方面符合图像的选取

实现深度卷积神经网络已经有了成熟的框架，本文实现使用的是Tensorflow。Tensorflow是一个可读性、简洁性和性能都很优秀的深度学习框架，并且直接集成了卷积神经网络神经层。由于深度卷积网络本身的特性，有时候用CPU加速运算可以大大缩短算法训练时间，Tensorflow也提供了相应的接口。

使用Tensorflow实现卷积神经网络时，需要预处理数据，然后根据设计好的网络结构配置响应的文件，即可使用Tensorflow训练并测试结果。

在大多数的深度学习算法里，网络的深度都是很重要的参数，深度卷积网络也不例外。在深度卷积网络中，增加深度并不会引起训练时间的爆炸性增长，而是一个相对缓慢的增长；网络所能够取得的结果却往往能够有明显的改善。我们经过试验对比发现，在2到7层之间，2层会取得较好的结果，所以在我们采用两层卷积层。

#### 3.1.4 SVM、随机森林、GDBT和逻辑回归等其他分类算法

在恶意代码最后的分类过程中我们采用了多种分算法做对比，一种是直接使用两层神经网络模型进行恶意代码分类，另一种方法就是采用SVM、随机森林、GBDT和逻辑回归等分类算法进行分类器的构建，实现对恶意代码的分类，这种方法主要是基于Python的scikit-learn库。

## 3.2 开发代码详解

DrawASMFileImageFeature.py

提取ASM文件图像纹理特征（即800pixel特征、1500pixel特征和2500pixel特征）

DrawOpcode\_n-gramFeature.py

提取恶意代码文件opcode—n-gram特征（即2-gram特征、3-gram特征和4-gram特征）

ImgfeaturePrediction\_800pixel.py

基于ASM文件图像纹理800像素特征的随机森林、支持向量分类、核支持向量分类、朴素贝叶斯分类、逻辑回归、决策树、

GBDT（梯度提升树）算法实现模型训练和分类预测；以及使用tensorflow实现两层神经网络模型训练和分类预测

ImgfeaturePrediction\_1500pixel.py

基于ASM文件图像纹理1500像素特征的随机森林、支持向量分类、核支持向量分类、朴素贝叶斯分类、逻辑回归、决策树、

GBDT（梯度提升树）算法实现模型训练和分类预测；以及使用tensorflow实现两层神经网络模型训练和分类预测

ImgfeaturePrediction\_2500pixel.py

基于ASM文件图像纹理2500特征的随机森林、支持向量分类、核支持向量分类、朴素贝叶斯分类、逻辑回归、决策树、

GBDT（梯度提升树）算法实现模型训练和分类预测；以及使用tensorflow实现两层神经网络模型训练和分类预测

NGramfeaturePrediction\_2.py

基于opcode-2-gram特征的随机森林、支持向量分类、核支持向量分类、朴素贝叶斯分类、逻辑回归、决策树、

GBDT（梯度提升树）算法实现模型训练和分类预测

NGramfeaturePrediction\_3.py

基于opcode-3-gram特征的随机森林、支持向量分类、核支持向量分类、朴素贝叶斯分类、逻辑回归、决策树、

GBDT（梯度提升树）算法实现模型训练和分类预测

NGramfeaturePrediction\_4.py

基于opcode-4-gram特征的随机森林、支持向量分类、核支持向量分类、朴素贝叶斯分类、逻辑回归、决策树、

GBDT（梯度提升树）算法实现模型训练和分类预测

CombinePrediction.py

基于ASM文件图像纹理800特征和opcode-2-gram特征、ASM文件图像纹理800特征和opcode-3-gram特征、

ASM文件图像纹理800特征和opcode-4-gram特征、ASM文件图像纹理1500特征和opcode-2-gram特征、

ASM文件图像纹理1500特征和opcode-3-gram特征、ASM文件图像纹理2500特征和opcode-4-gram特征、

ASM文件图像纹理2500特征和opcode-2-gram特征、ASM文件图像纹理2500特征和opcode-3-gram特征、

ASM文件图像纹理2500特征和opcode-4-gram特征的随机森林、支持向量分类、核支持向量分类、朴素贝叶斯分类、

逻辑回归、决策树、GBDT（梯度提升树）算法实现模型训练和分类预测

BestPrediction.py

基于ASM文件图像纹理2500像素特征，使用两层神经网络实现最优预测结果

## 3.3 运行结果比较分析

**NGramfeaturePrediction\_2.py**

**/usr/bin/python2.7 /home/qinyong/PycharmProjects/DistributedSystem/NGramfeaturePrediction\_2.py**

使用随机森林算法分类结果：

0.930555555556

使用支持向量分类算法分类结果：

0.536111111111

使用支持向量分类算法分类结果：

0.527777777778

使用朴素贝叶斯分类算法分类结果：

0.722222222222

使用逻辑回归算法分类结果：

0.908333333333

使用决策树算法分类结果：

0.861111111111

使用GBDT算法分类结果：

0.930555555556

**Process finished with exit code 0**

**NGramfeaturePrediction\_3.py**

**/usr/bin/python2.7 /home/qinyong/PycharmProjects/DistributedSystem/NGramfeaturePrediction\_3.py**

使用随机森林算法分类结果：

0.916666666667

使用支持向量分类算法分类结果：

0.666666666667

使用支持向量分类算法分类结果：

0.641666666667

使用朴素贝叶斯分类算法分类结果：

0.666666666667

使用逻辑回归算法分类结果：

0.866666666667

使用决策树算法分类结果：

0.861111111111

使用GBDT算法分类结果：

0.911111111111

**Process finished with exit code 0**

**NGramfeaturePrediction\_4.py**

**/usr/bin/python2.7 /home/qinyong/PycharmProjects/DistributedSystem/NGramfeaturePrediction\_4.py**

使用随机森林算法分类结果：

0.930555555556

使用支持向量分类算法分类结果：

0.738888888889

使用支持向量分类算法分类结果：

0.680555555556

使用朴素贝叶斯分类算法分类结果：

0.747222222222

使用逻辑回归算法分类结果：

0.902777777778

使用决策树算法分类结果：

0.894444444444

使用GBDT算法分类结果：

0.922222222222

**Process finished with exit code 0**

**CombinePrediction.py**

**/usr/bin/python2.7 /home/qinyong/PycharmProjects/DistributedSystem/CombinePrediction.py**

使用**2-gram特征和800像素特征**，各种算法预测结果如下：

使用随机森林算法分类结果：

0.966666666667

使用支持向量分类算法分类结果：

0.630555555556

使用支持向量分类算法分类结果：

0.627777777778

使用朴素贝叶斯分类算法分类结果：

0.816666666667

使用逻辑回归算法分类结果：

0.930555555556

使用决策树算法分类结果：

0.936111111111

使用GBDT算法分类结果：

0.961111111111

使用**2-gram特征和1500像素特征**，各种算法预测结果如下：

使用随机森林算法分类结果：

0.980555555556

使用支持向量分类算法分类结果：

0.661111111111

使用支持向量分类算法分类结果：

0.663888888889

使用朴素贝叶斯分类算法分类结果：

0.872222222222

使用逻辑回归算法分类结果：

0.95

使用决策树算法分类结果：

0.913888888889

使用GBDT算法分类结果：

0.983333333333

使用**2-gram特征和2500像素特征**，各种算法预测结果如下：

使用随机森林算法分类结果：

0.955555555556

使用支持向量分类算法分类结果：

0.416666666667

使用支持向量分类算法分类结果：

0.430555555556

使用朴素贝叶斯分类算法分类结果：

0.9

使用逻辑回归算法分类结果：

0.952777777778

使用决策树算法分类结果：

0.930555555556

使用GBDT算法分类结果：

0.972222222222

使用**3-gram特征和800像素特征**，各种算法预测结果如下：

使用随机森林算法分类结果：

0.966666666667

使用支持向量分类算法分类结果：

0.7

使用支持向量分类算法分类结果：

0.691666666667

使用朴素贝叶斯分类算法分类结果：

0.847222222222

使用逻辑回归算法分类结果：

0.955555555556

使用决策树算法分类结果：

0.944444444444

使用GBDT算法分类结果：

0.972222222222

使用**3-gram特征和1500像素特征**，各种算法预测结果如下：

使用随机森林算法分类结果：

0.991666666667

使用支持向量分类算法分类结果：

0.705555555556

使用支持向量分类算法分类结果：

0.677777777778

使用朴素贝叶斯分类算法分类结果：

0.894444444444

使用逻辑回归算法分类结果：

0.963888888889

使用决策树算法分类结果：

0.933333333333

使用GBDT算法分类结果：

0.988888888889

使用**3-gram特征和2500像素特征**，各种算法预测结果如下：

使用随机森林算法分类结果：

0.986111111111

使用支持向量分类算法分类结果：

0.486111111111

使用支持向量分类算法分类结果：

0.502777777778

使用朴素贝叶斯分类算法分类结果：

0.861111111111

使用逻辑回归算法分类结果：

0.947222222222

使用决策树算法分类结果：

0.958333333333

使用GBDT算法分类结果：

0.986111111111

使用**4-gram特征和800像素特征，**各种算法预测结果如下：

使用随机森林算法分类结果：

0.963888888889

使用支持向量分类算法分类结果：

0.769444444444

使用支持向量分类算法分类结果：

0.758333333333

使用朴素贝叶斯分类算法分类结果：

0.869444444444

使用逻辑回归算法分类结果：

0.947222222222

使用决策树算法分类结果：

0.947222222222

使用GBDT算法分类结果：

0.963888888889

使用**4-gram特征和1500像素特征**，各种算法预测结果如下：

使用随机森林算法分类结果：

0.966666666667

使用支持向量分类算法分类结果：

0.758333333333

使用支持向量分类算法分类结果：

0.75

使用朴素贝叶斯分类算法分类结果：

0.833333333333

使用逻辑回归算法分类结果：

0.955555555556

使用决策树算法分类结果：

0.95

使用GBDT算法分类结果：

0.947222222222

使用**4-gram特征和2500像素特征**，各种算法预测结果如下：

使用随机森林算法分类结果：

0.980555555556

使用支持向量分类算法分类结果：

0.455555555556

使用支持向量分类算法分类结果：

0.472222222222

使用朴素贝叶斯分类算法分类结果：

0.863888888889

使用逻辑回归算法分类结果：

0.944444444444

使用决策树算法分类结果：

0.938888888889

使用GBDT算法分类结果：

0.969444444444

**Process finished with exit code 0**

**ImgfeaturePrediction\_800pixel.py**

**/usr/bin/python2.7 /home/qinyong/PycharmProjects/DistributedSystem/ImgfeaturePrediction\_800pixel.py**

使用随机森林算法分类结果：

0.513888888889

使用支持向量分类算法分类结果：

0.511111111111

使用支持向量分类算法分类结果：

0.522222222222

使用朴素贝叶斯分类算法分类结果：

0.397222222222

使用逻辑回归算法分类结果：

0.516666666667

使用决策树算法分类结果：

0.513888888889

使用GBDT算法分类结果：

0.513888888889

**使用RMSProp优化器**：

Step 1, Minibatch Loss= 1638169.2500, Training Accuracy= 0.119

Step 100, Minibatch Loss= 131226.3906, Training Accuracy= 0.415

Step 200, Minibatch Loss= 61090.5742, Training Accuracy= 0.415

Step 300, Minibatch Loss= 125655.9375, Training Accuracy= 0.365

Step 400, Minibatch Loss= 15839.9014, Training Accuracy= 0.407

Step 500, Minibatch Loss= 38663.3789, Training Accuracy= 0.296

Step 600, Minibatch Loss= 19549.9844, Training Accuracy= 0.354

Step 700, Minibatch Loss= 26579.9258, Training Accuracy= 0.407

Step 800, Minibatch Loss= 39421.0586, Training Accuracy= 0.363

Step 900, Minibatch Loss= 86282.5469, Training Accuracy= 0.352

Step 1000, Minibatch Loss= 24406.2070, Training Accuracy= 0.400

Optimization Finished!

('Testing Accuracy:', 0.40000001)

**使用Adam优化器：**

Step 1, Minibatch Loss= 591237.2500, Training Accuracy= 0.109

Step 100, Minibatch Loss= 139549.8906, Training Accuracy= 0.394

Step 200, Minibatch Loss= 28572.3516, Training Accuracy= 0.400

Step 300, Minibatch Loss= 13349.6475, Training Accuracy= 0.531

Step 400, Minibatch Loss= 13737.1387, Training Accuracy= 0.519

Step 500, Minibatch Loss= 11749.1104, Training Accuracy= 0.374

Step 600, Minibatch Loss= 8293.5371, Training Accuracy= 0.446

Step 700, Minibatch Loss= 9496.6553, Training Accuracy= 0.420

Step 800, Minibatch Loss= 6816.3926, Training Accuracy= 0.448

Step 900, Minibatch Loss= 4337.7241, Training Accuracy= 0.530

Step 1000, Minibatch Loss= 2563.3315, Training Accuracy= 0.556

Optimization Finished!

('Testing Accuracy:', 0.55555558)

**使用GradientDescent优化器**：

Step 1, Minibatch Loss= 115736657920.0000, Training Accuracy= 0.122

Step 100, Minibatch Loss= nan, Training Accuracy= 0.122

Step 200, Minibatch Loss= nan, Training Accuracy= 0.122

Step 300, Minibatch Loss= nan, Training Accuracy= 0.122

Step 400, Minibatch Loss= nan, Training Accuracy= 0.122

Step 500, Minibatch Loss= nan, Training Accuracy= 0.122

Step 600, Minibatch Loss= nan, Training Accuracy= 0.122

Step 700, Minibatch Loss= nan, Training Accuracy= 0.122

Step 800, Minibatch Loss= nan, Training Accuracy= 0.122

Step 900, Minibatch Loss= nan, Training Accuracy= 0.122

Step 1000, Minibatch Loss= nan, Training Accuracy= 0.122

Optimization Finished!

('Testing Accuracy:', 0.12222222)

**使用Adagrad优化器**：

Step 1, Minibatch Loss= 589969.0000, Training Accuracy= 0.217

Step 100, Minibatch Loss= 70675.4844, Training Accuracy= 0.346

Step 200, Minibatch Loss= 47542.5117, Training Accuracy= 0.344

Step 300, Minibatch Loss= 41443.7266, Training Accuracy= 0.383

Step 400, Minibatch Loss= 38411.0391, Training Accuracy= 0.374

Step 500, Minibatch Loss= 33280.7109, Training Accuracy= 0.356

Step 600, Minibatch Loss= 16663.0059, Training Accuracy= 0.428

Step 700, Minibatch Loss= 13471.1592, Training Accuracy= 0.450

Step 800, Minibatch Loss= 18855.3262, Training Accuracy= 0.431

Step 900, Minibatch Loss= 20714.4258, Training Accuracy= 0.346

Step 1000, Minibatch Loss= 17029.8496, Training Accuracy= 0.398

Optimization Finished!

('Testing Accuracy:', 0.39814815)

**使用Adadelta优化器：**

Step 1, Minibatch Loss= 695206.1875, Training Accuracy= 0.185

Step 100, Minibatch Loss= 636916.9375, Training Accuracy= 0.185

Step 200, Minibatch Loss= 573867.3750, Training Accuracy= 0.185

Step 300, Minibatch Loss= 506822.2812, Training Accuracy= 0.185

Step 400, Minibatch Loss= 441447.1875, Training Accuracy= 0.185

Step 500, Minibatch Loss= 378790.3438, Training Accuracy= 0.215

Step 600, Minibatch Loss= 315664.5312, Training Accuracy= 0.104

Step 700, Minibatch Loss= 252374.8750, Training Accuracy= 0.104

Step 800, Minibatch Loss= 207475.7344, Training Accuracy= 0.133

Step 900, Minibatch Loss= 167870.1094, Training Accuracy= 0.198

Step 1000, Minibatch Loss= 132636.7344, Training Accuracy= 0.154

Optimization Finished!

('Testing Accuracy:', 0.1537037)

**使用FtrlO优化器：**

Step 1, Minibatch Loss= 564882.5000, Training Accuracy= 0.130

Step 100, Minibatch Loss= 52041.2500, Training Accuracy= 0.415

Step 200, Minibatch Loss= 45683.0234, Training Accuracy= 0.419

Step 300, Minibatch Loss= 26898.0996, Training Accuracy= 0.420

Step 400, Minibatch Loss= 23441.1602, Training Accuracy= 0.448

Step 500, Minibatch Loss= 29416.5254, Training Accuracy= 0.337

Step 600, Minibatch Loss= 15605.0703, Training Accuracy= 0.420

Step 700, Minibatch Loss= 18878.4414, Training Accuracy= 0.422

Step 800, Minibatch Loss= 25387.6348, Training Accuracy= 0.367

Step 900, Minibatch Loss= 23120.8457, Training Accuracy= 0.417

Step 1000, Minibatch Loss= 20199.2207, Training Accuracy= 0.413

Optimization Finished!

('Testing Accuracy:', 0.41296297)

**Process finished with exit code 0**

**ImgfeaturePrediction\_1500pixel.py**

**/usr/bin/python2.7 /home/qinyong/PycharmProjects/DistributedSystem/ImgfeaturePrediction\_1500pixel.py**

使用随机森林算法分类结果：

0.963888888889

使用支持向量分类算法分类结果：

0.922222222222

使用支持向量分类算法分类结果：

0.891666666667

使用朴素贝叶斯分类算法分类结果：

0.716666666667

使用逻辑回归算法分类结果：

0.952777777778

使用决策树算法分类结果：

0.947222222222

使用GBDT算法分类结果：

0.972222222222

**使用RMSProp优化器：**

Step 1, Minibatch Loss= 2913299.0000, Training Accuracy= 0.104

Step 100, Minibatch Loss= 93746.2422, Training Accuracy= 0.611

Step 200, Minibatch Loss= 341751.0000, Training Accuracy= 0.402

Step 300, Minibatch Loss= 43495.2969, Training Accuracy= 0.683

Step 400, Minibatch Loss= 86184.2812, Training Accuracy= 0.563

Step 500, Minibatch Loss= 29647.2324, Training Accuracy= 0.713

Step 600, Minibatch Loss= 17665.8223, Training Accuracy= 0.754

Step 700, Minibatch Loss= 35361.3242, Training Accuracy= 0.676

Step 800, Minibatch Loss= 30245.2188, Training Accuracy= 0.726

Step 900, Minibatch Loss= 193658.8594, Training Accuracy= 0.631

Step 1000, Minibatch Loss= 17998.9570, Training Accuracy= 0.739

Optimization Finished!

('Testing Accuracy:', 0.73888886)

**使用Adam优化器：**

Step 1, Minibatch Loss= 1027085.6250, Training Accuracy= 0.120

Step 100, Minibatch Loss= 62895.8672, Training Accuracy= 0.706

Step 200, Minibatch Loss= 26676.3438, Training Accuracy= 0.733

Step 300, Minibatch Loss= 17374.2324, Training Accuracy= 0.793

Step 400, Minibatch Loss= 13605.6357, Training Accuracy= 0.765

Step 500, Minibatch Loss= 7663.7842, Training Accuracy= 0.824

Step 600, Minibatch Loss= 8325.1338, Training Accuracy= 0.789

Step 700, Minibatch Loss= 7595.5684, Training Accuracy= 0.857

Step 800, Minibatch Loss= 7319.0474, Training Accuracy= 0.724

Step 900, Minibatch Loss= 3999.5046, Training Accuracy= 0.887

Step 1000, Minibatch Loss= 6684.0166, Training Accuracy= 0.870

Optimization Finished!

('Testing Accuracy:', 0.87037039)

**使用GradientDescent优化器：**

Step 1, Minibatch Loss= 483477848064.0000, Training Accuracy= 0.120

Step 100, Minibatch Loss= nan, Training Accuracy= 0.109

Step 200, Minibatch Loss= nan, Training Accuracy= 0.109

Step 300, Minibatch Loss= nan, Training Accuracy= 0.109

Step 400, Minibatch Loss= nan, Training Accuracy= 0.109

Step 500, Minibatch Loss= nan, Training Accuracy= 0.109

Step 600, Minibatch Loss= nan, Training Accuracy= 0.109

Step 700, Minibatch Loss= nan, Training Accuracy= 0.109

Step 800, Minibatch Loss= nan, Training Accuracy= 0.109

Step 900, Minibatch Loss= nan, Training Accuracy= 0.109

Step 1000, Minibatch Loss= nan, Training Accuracy= 0.109

Optimization Finished!

('Testing Accuracy:', 0.10925926)

**使用Adagrad优化器：**

Step 1, Minibatch Loss= 1097340.5000, Training Accuracy= 0.028

Step 100, Minibatch Loss= 27578.7891, Training Accuracy= 0.698

Step 200, Minibatch Loss= 30010.3496, Training Accuracy= 0.715

Step 300, Minibatch Loss= 11465.5957, Training Accuracy= 0.715

Step 400, Minibatch Loss= 18191.6914, Training Accuracy= 0.719

Step 500, Minibatch Loss= 19215.3086, Training Accuracy= 0.774

Step 600, Minibatch Loss= 22028.4512, Training Accuracy= 0.720

Step 700, Minibatch Loss= 19590.2812, Training Accuracy= 0.761

Step 800, Minibatch Loss= 15232.8838, Training Accuracy= 0.752

Step 900, Minibatch Loss= 10877.8633, Training Accuracy= 0.739

Step 1000, Minibatch Loss= 15374.3877, Training Accuracy= 0.761

Optimization Finished!

('Testing Accuracy:', 0.76111114)

**使用Adadelta优化器：**

Step 1, Minibatch Loss= 1022685.0625, Training Accuracy= 0.033

Step 100, Minibatch Loss= 904664.4375, Training Accuracy= 0.022

Step 200, Minibatch Loss= 784700.6875, Training Accuracy= 0.004

Step 300, Minibatch Loss= 673935.3125, Training Accuracy= 0.006

Step 400, Minibatch Loss= 566350.9375, Training Accuracy= 0.028

Step 500, Minibatch Loss= 466077.5000, Training Accuracy= 0.057

Step 600, Minibatch Loss= 369697.4062, Training Accuracy= 0.093

Step 700, Minibatch Loss= 288936.2812, Training Accuracy= 0.128

Step 800, Minibatch Loss= 224709.5469, Training Accuracy= 0.181

Step 900, Minibatch Loss= 176683.1719, Training Accuracy= 0.250

Step 1000, Minibatch Loss= 141714.0469, Training Accuracy= 0.265

Optimization Finished!

('Testing Accuracy:', 0.26481482)

**使用FtrlO优化器：**

Step 1, Minibatch Loss= 811241.1875, Training Accuracy= 0.031

Step 100, Minibatch Loss= 24702.9219, Training Accuracy= 0.661

Step 200, Minibatch Loss= 28545.7773, Training Accuracy= 0.665

Step 300, Minibatch Loss= 28925.9785, Training Accuracy= 0.720

Step 400, Minibatch Loss= 14326.9434, Training Accuracy= 0.663

Step 500, Minibatch Loss= 10634.3623, Training Accuracy= 0.756

Step 600, Minibatch Loss= 12714.1826, Training Accuracy= 0.770

Step 700, Minibatch Loss= 8635.5166, Training Accuracy= 0.685

Step 800, Minibatch Loss= 4564.2632, Training Accuracy= 0.807

Step 900, Minibatch Loss= 8629.2354, Training Accuracy= 0.781

Step 1000, Minibatch Loss= 14596.1855, Training Accuracy= 0.743

Optimization Finished!

('Testing Accuracy:', 0.74259257)

**Process finished with exit code 0**

**ImgfeaturePrediction\_2500pixel.py**

**/usr/bin/python2.7 /home/qinyong/PycharmProjects/DistributedSystem/ImgfeaturePrediction\_2500pixel.py**

使用随机森林算法分类结果：

0.952777777778

使用支持向量分类算法分类结果：

0.575

使用支持向量分类算法分类结果：

0.572222222222

使用朴素贝叶斯分类算法分类结果：

0.758333333333

使用逻辑回归算法分类结果：

0.938888888889

使用决策树算法分类结果：

0.905555555556

使用GBDT算法分类结果：

0.933333333333

**使用RMSProp优化器：**

Step 1, Minibatch Loss= 3327918.7500, Training Accuracy= 0.169

Step 100, Minibatch Loss= 71217.0234, Training Accuracy= 0.752

Step 200, Minibatch Loss= 167000.8281, Training Accuracy= 0.569

Step 300, Minibatch Loss= 22833.0059, Training Accuracy= 0.869

Step 400, Minibatch Loss= 149259.6094, Training Accuracy= 0.809

Step 500, Minibatch Loss= 6542.1396, Training Accuracy= 0.931

Step 600, Minibatch Loss= 1477.5553, Training Accuracy= 0.972

Step 700, Minibatch Loss= 916.3538, Training Accuracy= 0.983

Step 800, Minibatch Loss= 918.4988, Training Accuracy= 0.981

Step 900, Minibatch Loss= 372.1985, Training Accuracy= 0.994

Step 1000, Minibatch Loss= 1047.3105, Training Accuracy= 0.983

Optimization Finished!

('Testing Accuracy:', 0.98333335)

**使用Adam优化器：**

Step 1, Minibatch Loss= 1656577.5000, Training Accuracy= 0.126

Step 100, Minibatch Loss= 1344.0509, Training Accuracy= 0.985

Step 200, Minibatch Loss= 7.4170, Training Accuracy= 0.996

Step 300, Minibatch Loss= 18.3046, Training Accuracy= 0.998

Step 400, Minibatch Loss= 6.0604, Training Accuracy= 0.998

Step 500, Minibatch Loss= 317.0854, Training Accuracy= 0.991

Step 600, Minibatch Loss= 0.0000, Training Accuracy= 1.000

Step 700, Minibatch Loss= 0.0000, Training Accuracy= 1.000

Step 800, Minibatch Loss= 0.0000, Training Accuracy= 1.000

Step 900, Minibatch Loss= 0.0000, Training Accuracy= 1.000

Step 1000, Minibatch Loss= 0.0000, Training Accuracy= 1.000

Optimization Finished!

('Testing Accuracy:', 1.0)

**使用GradientDescent优化器：**

Step 1, Minibatch Loss= 891264499712.0000, Training Accuracy= 0.120

Step 100, Minibatch Loss= nan, Training Accuracy= 0.102

Step 200, Minibatch Loss= nan, Training Accuracy= 0.102

Step 300, Minibatch Loss= nan, Training Accuracy= 0.102

Step 400, Minibatch Loss= nan, Training Accuracy= 0.102

Step 500, Minibatch Loss= nan, Training Accuracy= 0.102

Step 600, Minibatch Loss= nan, Training Accuracy= 0.102

Step 700, Minibatch Loss= nan, Training Accuracy= 0.102

Step 800, Minibatch Loss= nan, Training Accuracy= 0.102

Step 900, Minibatch Loss= nan, Training Accuracy= 0.102

Step 1000, Minibatch Loss= nan, Training Accuracy= 0.102

Optimization Finished!

('Testing Accuracy:', 0.10185185)

**使用Adagrad优化器：**

Step 1, Minibatch Loss= 1540325.0000, Training Accuracy= 0.094

Step 100, Minibatch Loss= 47917.2227, Training Accuracy= 0.739

Step 200, Minibatch Loss= 20182.1680, Training Accuracy= 0.841

Step 300, Minibatch Loss= 9696.7627, Training Accuracy= 0.889

Step 400, Minibatch Loss= 6364.7676, Training Accuracy= 0.909

Step 500, Minibatch Loss= 4047.3945, Training Accuracy= 0.943

Step 600, Minibatch Loss= 2219.7725, Training Accuracy= 0.970

Step 700, Minibatch Loss= 1329.3962, Training Accuracy= 0.972

Step 800, Minibatch Loss= 940.1345, Training Accuracy= 0.976

Step 900, Minibatch Loss= 633.1077, Training Accuracy= 0.980

Step 1000, Minibatch Loss= 278.4106, Training Accuracy= 0.987

Optimization Finished!

('Testing Accuracy:', 0.98703706)

**使用Adadelta优化器**：

Step 1, Minibatch Loss= 1701650.2500, Training Accuracy= 0.115

Step 100, Minibatch Loss= 1415770.1250, Training Accuracy= 0.117

Step 200, Minibatch Loss= 1156349.8750, Training Accuracy= 0.096

Step 300, Minibatch Loss= 936227.5625, Training Accuracy= 0.076

Step 400, Minibatch Loss= 739614.3125, Training Accuracy= 0.074

Step 500, Minibatch Loss= 607102.6875, Training Accuracy= 0.089

Step 600, Minibatch Loss= 511723.8438, Training Accuracy= 0.111

Step 700, Minibatch Loss= 434235.1562, Training Accuracy= 0.141

Step 800, Minibatch Loss= 370788.6250, Training Accuracy= 0.161

Step 900, Minibatch Loss= 313023.0625, Training Accuracy= 0.194

Step 1000, Minibatch Loss= 265282.1562, Training Accuracy= 0.239

Optimization Finished!

('Testing Accuracy:', 0.23888889)

**使用FtrlO优化器：**

Step 1, Minibatch Loss= 2092522.6250, Training Accuracy= 0.154

Step 100, Minibatch Loss= 22759.4141, Training Accuracy= 0.837

Step 200, Minibatch Loss= 7693.4067, Training Accuracy= 0.920

Step 300, Minibatch Loss= 5026.8545, Training Accuracy= 0.904

Step 400, Minibatch Loss= 1663.7523, Training Accuracy= 0.970

Step 500, Minibatch Loss= 1071.6084, Training Accuracy= 0.980

Step 600, Minibatch Loss= 1080.4586, Training Accuracy= 0.981

Step 700, Minibatch Loss= 338.2769, Training Accuracy= 0.991

Step 800, Minibatch Loss= 1276.6757, Training Accuracy= 0.976

Step 900, Minibatch Loss= 1.2261, Training Accuracy= 0.998

Step 1000, Minibatch Loss= 0.0000, Training Accuracy= 1.000

Optimization Finished!

('Testing Accuracy:', 1.0)

Process finished with exit code 0

**BestPrediction.py**

**/usr/bin/python2.7 /home/qinyong/PycharmProjects/DistributedSystem/BestPrediction.py**

**使用Adam优化器：**

**Step 1, Minibatch Loss= 1241454.6250, Training Accuracy= 0.228**

**Step 100, Minibatch Loss= 1034.0033, Training Accuracy= 0.961**

**Step 200, Minibatch Loss= 24.9628, Training Accuracy= 0.996**

**Step 300, Minibatch Loss= 37.0244, Training Accuracy= 0.996**

**Step 400, Minibatch Loss= 218.4113, Training Accuracy= 0.985**

**Step 500, Minibatch Loss= 0.0000, Training Accuracy= 1.000**

**Step 600, Minibatch Loss= 15.6786, Training Accuracy= 0.998**

**Step 700, Minibatch Loss= 0.0000, Training Accuracy= 1.000**

**Step 800, Minibatch Loss= 0.0000, Training Accuracy= 1.000**

**Step 900, Minibatch Loss= 0.0000, Training Accuracy= 1.000**

**Step 1000, Minibatch Loss= 0.0000, Training Accuracy= 1.000**

**Optimization Finished!**

**('Testing Accuracy:', 1.0)**

**使用FtrlO优化器：**

**Step 1, Minibatch Loss= 1253328.1250, Training Accuracy= 0.217**

**Step 100, Minibatch Loss= 39214.8320, Training Accuracy= 0.774**

**Step 200, Minibatch Loss= 15838.2441, Training Accuracy= 0.874**

**Step 300, Minibatch Loss= 8806.9033, Training Accuracy= 0.917**

**Step 400, Minibatch Loss= 4442.7100, Training Accuracy= 0.931**

**Step 500, Minibatch Loss= 1808.0312, Training Accuracy= 0.965**

**Step 600, Minibatch Loss= 887.6412, Training Accuracy= 0.974**

**Step 700, Minibatch Loss= 274.9190, Training Accuracy= 0.985**

**Step 800, Minibatch Loss= 27.2812, Training Accuracy= 0.991**

**Step 900, Minibatch Loss= 0.0000, Training Accuracy= 1.000**

**Step 1000, Minibatch Loss= 0.0000, Training Accuracy= 1.000**

**Optimization Finished!**

**('Testing Accuracy:', 1.0)**

**Process finished with exit code 0**

分析：从上述数据可以看出：

**仅n-gram特征：**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | 随机森林 | 支持向量 | 核支持向量 | 朴素贝叶斯 | 逻辑回归 | 决策树 | GBDT |
| 2-gram | 0.930555555556 | 0.536111111111 | 0.527777777778 | 0.722222222222 | 0.908333333333 | 0.861111111111 | 0.930555555556 |
| 3-gram | 0.916666666667 | 0.666666666667 | 0.641666666667 | 0.666666666667 | 0.866666666667 | 0.861111111111 | 0.911111111111 |
| 4-gram | 0.930555555556 | 0.738888888889 | 0.680555555556 | 0.747222222222 | 0.902777777778 | 0.894444444444 | 0.922222222222 |

**仅前n个像素特征：**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | | 800pixel | 1500pixel | 2500pixel |
| 随机森林 | | 0.513888888889 | 0.963888888889 | 0.952777777778 |
| 支持向量 | | 0.511111111111 | 0.922222222222 | 0.575 |
| 核支持向量 | | 0.522222222222 | 0.891666666667 | 0.572222222222 |
| 朴素贝叶斯 | | 0.397222222222 | 0.716666666667 | 0.758333333333 |
| 逻辑回归 | | 0.516666666667 | 0.952777777778 | 0.938888888889 |
| 决策树 | | 0.513888888889 | 0.947222222222 | 0.905555555556 |
| GBDT | | 0.513888888889 | 0.972222222222 | 0.933333333333 |
| 两层神经网络模型 | RMSProp优化器 | 0.40000001 | 0.73888886 | 0.98333335 |
| 使用Adam优化器 | 0.55555558 | 0.87037039 | 1.0 |
| 使用GradientDescent优化器 | 0.12222222 | 0.10925926 | 0.10185185 |
| 使用Adagrad优化器 | 0.39814815 | 0.76111114 | 0.98703706 |
| 使用Adadelta优化器 | 0.1537037 | 0.26481482 | 0.23888889 |
| 使用FtrlO优化器 | 0.41296297 | 0.74259257 | 1.0 |

**n-gram特征和前N个特征相结合：**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | 随机森林 | 支持向量 | 核支持向量 | 朴素贝叶斯 | 逻辑回归 | 决策树 | GBDT |
| 2-gram和800像素 | 0.966666666667 | 0.630555555556 | 0.627777777778 | 0.816666666667 | 0.930555555556 | 0.936111111111 | 0.961111111111 |
| 3-gram和800像素 | 0.966666666667 | 0.7 | 0.691666666667 | 0.847222222222 | 0.955555555556 | 0.944444444444 | 0.972222222222 |
| 4-gram和800像素 | 0.963888888889 | 0.769444444444 | 0.758333333333 | 0.869444444444 | 0.947222222222 | 0.947222222222 | 0.963888888889 |
| 2-gram和1500像素 | 0.980555555556 | 0.661111111111 | 0.663888888889 | 0.872222222222 | 0.95 | 0.913888888889 | 0.983333333333 |
| 3-gram和1500像素 | 0.991666666667 | 0.705555555556 | 0.677777777778 | 0.894444444444 | 0.963888888889 | 0.933333333333 | 0.988888888889 |
| 4-gram和1500像素 | 0.966666666667 | 0.758333333333 | 0.75 | 0.833333333333 | 0.955555555556 | 0.95 | 0.947222222222 |
| 2-gram和2500像素 | 0.955555555556 | 0.416666666667 | 0.430555555556 | 0.9 | 0.952777777778 | 0.930555555556 | 0.972222222222 |
| 3-gram和2500像素 | 0.986111111111 | 0.486111111111 | 0.502777777778 | 0.861111111111 | 0.947222222222 | 0.958333333333 | 0.986111111111 |
| 4-gram和2500像素 | 0.980555555556 | 0.455555555556 | 0.472222222222 | 0.863888888889 | 0.944444444444 | 0.938888888889 | 0.969444444444 |

由此可知：当使用两层神经网络模型，使用恶意代码文件的前2500个像素特征时，使用Adam优化器和Ftrl0优化器效果是最好的，达到了100%的准确率。