项目：Android系统中的系统异常及检测机制研究

**模型训练手册**

二○二三年八月

**目 录**

[1 性能异常模型训练 1](#_Toc135901939)

[1.1 异常检测模型概述 1](#_Toc135901940)

[1.2 异常检测决策树模型结构与参数 3](#_Toc135901941)

[1.3 异常检测决策树模型数据集 4](#_Toc135901942)

[1.4 异常检测决策树模型训练过程 5](#_Toc135901943)

[2 功耗异常模型训练 10](#_Toc135901944)

[2.1 功耗异常检测模型结构与参数 10](#_Toc135901945)

[2.1.1 短时异常 10](#_Toc135901946)

[2.1.2 长时异常 11](#_Toc135901947)

[2.2 功耗异常检测数据集 13](#_Toc135901948)

[2.3 功耗异常检测模型训练过程 14](#_Toc135901949)

[2.3.1 短时异常 14](#_Toc135901950)

[2.3.2 长时异常 16](#_Toc135901951)

[3 附录 19](#_Toc135901952)

本文将分别介绍性能异常和功耗异常检测模型的结构、训练方法、数据集与参数。

训练环境：NVIDIA RTX 3060显卡，12GB显存，Windows 11系统，Python 3.7。

# **性能异常模型训练**

本章先对异常检测模型进行一个概述，然后会从性能异常检测决策树模型结构与参数、训练数据集以及训练流程四个方面对性能异常模拟进行说明。

* 1. **异常检测模型概述**

决策树模型是一种常见的分类模型，是一种基于树状结构来进行决策的模型，决策树算法的基本思想是对数据进行递归划分，在这个过程中，决策树会选择最优的特征来进行划分，从而生成一系列节点和边，代表不同的决策路径和决策规则，最后构建一棵决策树来实现分类或回归任务。在决策树模型中，每个内部节点表示一个属性或者特征，每个叶子结点表示一个类别，决策树的生成过程包括最优划分属性的选择、数据集的划分以及生成叶子节点等步骤，直到所有的数据都被正确的分类或者达到预定的停止条件。

本项目性能的异常检测采用决策树模型，原因是决策树模型具有以下优点：

（1）直观易懂：决策树是白盒模型，可以自然地表现出特征之间的关系，具有很好的可解释性和可视化能力，可根据决策树推断相应的逻辑表达式；

（2）可处理非线性关系：决策树可以很好地处理非线性数据关系，能够同时处理数据型和分类型属性，可以应对Android系统的不同类型指标

（3）无需数据标准化：决策树对于不同尺度和单位的特征可以进行处理，无需对数据进行特征缩放或归一化等数据预处理步骤，一般可以自动处理各种数据类型和缺失值，可以解决监控中可能出现的数据缺失问题；

（4）计算复杂度不高，在相对短的时间内能够对大数据集合做出可行且效果良好的分类结果，而且推理速度快，如果需要重新训练，开销较小，适用于移动端方案。

本文针对后台应用抢占资源导致的前台应用性能异常，提出一种性能异常的检测模型，如表1.1显示了主要检测的性能异常类型：

表1.1 主要异常类型

|  |  |
| --- | --- |
| **性能异常类型** | **内容** |
| CPU异常 | 当后台应用过度使用CPU资源时，会导致前台应用分到的CPU时间片减少，并且前台应用和系统进程可能会被调度到频率较低的小核上，造成帧率下降。 |
| 内存异常 | 当后台应用占用过多内存时，尽管Android系统会采用kswap和LMK回收内存保证前台正常运行，但前台应用的内存增长速率受限，并且内存PSI（Pressure Stall Information）显著增高，导致性能下降 |
| GPU异常 | 在异常模拟阶段，发现后台应用对GPU的使用也会造成性能下降，但由于目前高通提供的接口中对GPU的监控信息非常有限，无法获取各应用的使用情况，因此本项目暂不考虑对GPU异常应用的定位。 |
| 混合异常 | 在系统中，可能同时存在上述多种异常同时出现的情况 |

模型的检测原理如下：实验发现：当手机资源不足时，整机和前台应用的资源使用情况都会和正常情况下有明显不同。利用这一现象，分别在系统正常和异常运行的情况下采集与资源使用相关的监控数据，用于训练机器学习模型：模型的输入为各项监控数据，模型的输出为异常分类结果。模型训练完成后，可对实时数据进行推理，分类各种情况的异常。如果用户在使用中安装了不在训练集中的应用，可以使用预训练的整机资源模型对陌生前台应用进行推理，用于训练的数据是开启多种前台应用时的整机资源使用情况，其他情况使用特定的前台应用资源模型进行诊断。

为了对于混合异常有更加好的检测，本文不使用多分类模型，转而采取使用三个二分类模型进行性能异常的检测，模型包括：CPU异常检测模型、GPU异常检测模型以及内存异常检测模型，CPU异常检测模型的输出为：0-CPU异常未检出，1-CPU异常检出；GPU异常检测模型的输出为：0-GPU异常未检出，1-CPU异常检出；内存异常监测模型的输出为：0-内存异常未检出，1-内存异常检出；判断是否是混合异常只需确定是否同时检出多种异常即可。

* 1. **异常检测决策树模型结构与参数**

本文的异常检测决策树模型针对特定前台资源的模型输入如表1.2，整机资源的模型输入如表1.3：

表1.2 针对特定前后的异常检测决策树模型输入

|  |  |  |
| --- | --- | --- |
| **序号** | **指标名** | **含义** |
| 1 | *FrontAppCpuUsage* | 前台应用平均CPU利用率 |
| 2 | *usedNativeMemPercent* | 前台应用Native堆内存使用占分配比例 |
| 3 | *FrontAppJavaHeapPercent* | 前台应用的JavaHeap占总内存比例 |
| 4 | *FrontAppNativeHeapPercent* | 前台应用的NativeHeap占总内存比例 |
| 5 | *FrontAppCodeMemPercent* | 前台应用的CodeMem占总内存比例 |
| 6 | *FrontAppStackMemPercent* | 前台应用的StackMem占总内存比例 |
| 7 | *FrontAppGraphicsMemPercent* | 前台应用的GraphicsMem占总内存比例 |
| 8 | *FrontAppPrivateOtherPercent* | 前台应用PrivateOtherMem占总内存比例 |
| 9 | *FrontAppSystemMemPercent* | 前台应用的SystemMem占总内存比例 |
| 10-17 | *tempCPU0~tempCPU7* | CPU八个核心的温度 |
| 18 | *batteryTemp* | 电池温度 |
| 19 | *thermalStatus* | 系统热状态 |
| 20 | *GpuUsage* | GPU利用率 |
| 21 | *GpuFreq* | GPU频率 |
| 22-29 | *cpu0Freq~cpu7Freq* | CPU八个核心的频率 |

表1.2 针对整机的异常检测决策树模型输入

|  |  |  |
| --- | --- | --- |
| **序号** | **指标名** | **含义** |
| 1 | *CpuUsage* | 全局平均CPU利用率 |
| 2 | *Memavailable* | 可用内存内总内存比例 |
| 3 | *MemFree* | 空闲内存占总内存比例 |
| 4 | *GpuUsage* | GPU利用率 |
| 5 | *GpuFreq* | GPU频率 |
| 6-13 | *cpu0Freq~cpu7Freq* | CPU八个核心的频率 |
| 14-21 | *tempCPU0~tempCPU7* | CPU八个核心的温度 |
| 22 | *batteryTemp* | 电池温度 |
| 23 | *thermalStatus* | 系统热状态 |

文本的异常检测决策树模型输出为是否异常，具体包括CPU异常检测模型：0-CPU异常未检出，1-CPU异常检出；GPU异常检测模型的输出为：0-GPU异常未检出，1-GPU异常检出；内存异常监测模型的输出为：0-内存异常未检出，1-内存异常检出；

异常检测决策树模型的训练使用scikit-learn机器学习库，模型的定义为：

model=DecisionTreeClassifier(class\_weight='balanced', criterion ='gini')

相关主要参数值见表1.3：

表1.3 决策树模型主要参数说明

|  |  |  |
| --- | --- | --- |
| **参数名称** | **参数值** | **参数含义** |
| criterion | gini | 分裂节点的采取标准,这里采取基尼系数，基尼系数越小，不纯度越低，效果越好 |
| class\_weight | balanced | 每个类的权重进行平衡，平衡数据集数量差异的影响 |
| max\_depth | none | 树最大深度不限，节点展开直到所有叶子都是一个类 |
| splitter | best | 用于在每个节点上选择拆分的策略，选择最佳拆分 |
| min\_samples\_split | 2 | 拆分内部节点所需的最少样本数为2 |
| min\_samples\_leaf | 1 | 在叶节点处需要的最小样本数为1 |
| max\_features | none | 寻找最佳分割时考虑所有的特征数量 |

* 1. **异常检测决策树模型数据集**

异常检测决策树模型的数据集为在异常开启与关闭不同时刻收集的各项前台应用指标与手机指标，包括但不限于上述决策树模型的输入，部分数据格式示例如图1.1所示，相关训练数据以csv表格的形式存储，间隔的各个数据量均在1000条以上：

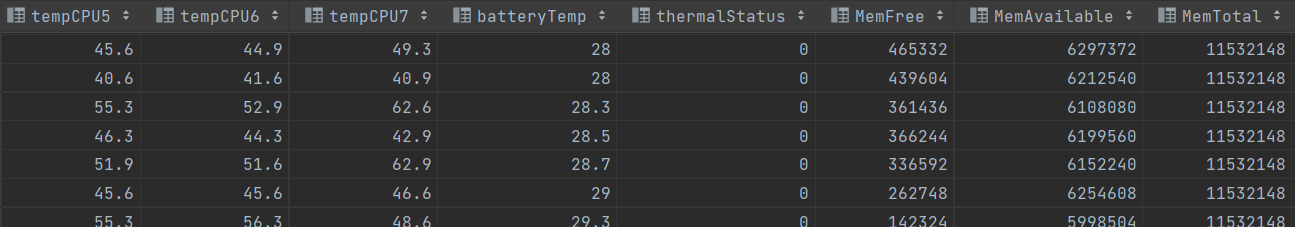


图1.1 数据集部分概览

* 1. **异常检测决策树模型训练过程**

异常检测决策树模型作为一个Python语言实现的项目在Windows设备上实现，用于训练模型的Windows设备主要参数与依赖库版本如下表1.4：

表1.4 训练平台

|  |  |
| --- | --- |
| 平台 | 配置 |
| 操作系统 | Windows10/11 |
| 开发平台 | Pycharm 2022.1 |
| Python版本 | Python 3.7 |
| Scikit-learn库版本 | 0.23.1 |
| Sklearn2pmml库版本 | 0.63.1 |
| Joblib库版本 | 0.16.0 |

以淘宝的CPU异常检测模型训练项目目录对代码文件结构进行说明如下，GPU异常与内存异常检测模型效果训练方式类似，其他应用模型以及整机模型的代码目录与淘宝也相似：

C:.

│ foreDeal.py #用于对采集的数据进行预处理

│ options.py #用于选择输入输出参数

│

├─taobao #淘宝异常检测模型目录

│ ├─cpuData #淘宝CPU异常检测模型目录

│ │ │ draw.py #绘图相关代码

│ │ │ main.py #决策树模型训练与结果展示代码

│ │ │ others1.py #baseline模型：SVM

│ │ │ others2.py #baseline模型：朴素贝叶斯

│ │ │ tree.png #生成决策树可视化

│ │ │

│ │ ├─abnormal #异常数据目录

│ │ │ Abnormal.csv # 存放的CPU性能异常数据目录

│ │ │ data\_AfterDeal.csv # 对CPU异常数据的处理结果

│ │ │ train.csv # 存放生成用于模型的训练的数据

│ │ │

│ │ ├─normal #正常数据目录

│ │ │ data\_AfterDeal.csv # 存放的CPU性能异常数据目录

│ │ │ Normal.csv # 对CPU异常数据的处理结果

│ │ │ train.csv # 存放生成用于模型的训练的数据

│ │ │

│ │ └─res #存放决策树模型结果目录

│ │ cpuDecisionTreeClassifier.pkl #pkl文件

│ │ cpuDecisionTreeClassifier.pmml #pmml文件

│ │

│ ├─gpuData #淘宝GPU异常检测模型目录

│ │ │…

│ │ ├─abnormal…

│ │ ├─normal…

│ │ └─res…

│ └─memData #淘宝内存异常检测模型目录

│ │ …

│ ├─abnormal…

│ ├─normal…

│ └─res…

│

├─douyin #抖音异常检测模型目录

│ ├─cpuData… #抖音CPU异常检测模型目录

│ ├─gpuData… #抖音GPU异常检测模型目录

│ └─memData… #抖音内存异常检测模型目录

│

├─yuanshen #原神异常检测模型目录

│ ├─cpuData… #原神CPU异常检测模型目录

│ ├─gpuData… #原神GPU异常检测模型目录

│ └─memData… #原神内存异常检测模型目录

│

└─entire #整机异常检测模型目录

├─cpuData… # 整机CPU异常检测模型目录

├─gpuData… # 整机GPU异常检测模型目录

└─memData… # 整机内存异常检测模型目录

根据目录所示，训练过程如下，首先将正常数据以及异常数据放到相应模型目录的normal目录以及abnormal目录下，然后运行该异常检测模型下的main.py（直接在pycharm运行或者终端运行 “python + 文件路径”）即可训练保存模型。

main文件主要内容如下：包括对文件预处理后通过模型进行训练并最后输出模型评价以及可视化决策树图片，保存模型为pkl形式以及跨平台通用pmml形式。

|  |
| --- |
| # 添加标签  def getdata(dir, label):      df\_s = pd.read\_csv(dir)      df\_s['label'] = label      df = df\_s['label']      df\_s = df\_s.drop(['label'], axis=1)      return df\_s, df  # 决策树  def decsionTreeSolution(feature):      prepare\_log("normal/", "Normal.csv", feature)      prepare\_log("abnormal/", "Abnormal.csv", feature)      df\_s1, df\_st1 = getdata('normal/train.csv', 0)      print(df\_s1)      df\_s2, df\_st2 = getdata('abnormal/train.csv', 1)      df\_train = pd.concat([df\_s1, df\_s2], ignore\_index=True)      df\_test = pd.concat([df\_st1, df\_st2], ignore\_index=True)      # 划分训练集和测试集7:3      x\_train, x\_test, y\_train, y\_test = train\_test\_split(df\_train, df\_test, test\_size=0.3)      # 训练，预测      clf = DecisionTreeClassifier(class\_weight='balanced', criterion='gini')      clf.fit(x\_train, y\_train)      predict\_target = clf.predict(x\_test)      # 保存pkl模型      joblib.dump(clf, filename='res/cpuDecisionTreeClassifier.pkl')      # 预测结果与真实结果的对比      print('预测数据量：', len(y\_test), '正确数据量：', len(predict\_target == y\_test))      # 输出准确率，召回率，F值      print(metrics.classification\_report(y\_test, predict\_target))      print(metrics.confusion\_matrix(y\_test, predict\_target))      # 预测模型得分      sorce = clf.score(x\_test, y\_test)      print("此模型得分为%s" % sorce)      # 获取测试数据集两列数据集      X = x\_test      print(X)      L1 = [n[0] for n in X]      print(L1)      L2 = [n[1] for n in X]      print(L2)      text\_representation = tree.export\_text(clf, feature\_names=feature\_names)      print(text\_representation)      # plot\_tree函数      # 用图片画出      plt.figure(figsize=(100,50))      a = tree.plot\_tree(clf,                         feature\_names=feature\_names,                         class\_names=['Normal', 'CpuAbnormal'],                         rounded=True,                         filled=True,                         fontsize=14)      # plt.show()      plt.savefig('tree.png')      # 导出pmml模型      clf2 = DecisionTreeClassifier()      pipeline = PMMLPipeline([("classifier", clf2)])      pipeline.fit(x\_train, y\_train)      sklearn2pmml(pipeline, 'res/cpuDecisionTreeClassifier.pmml', with\_repr=True)  if \_\_name\_\_ == '\_\_main\_\_':      feature\_names = [options['target9'], options['target15'], options['target14'], options['target16'],                       options['target17'], options['target18'], options['target19'], options['target20'],                       options['target21'], options['target28'], options['target29'], options['target30'],                       options['target31'], options['target32'], options['target33'], options['target34'],                       options['target35'], options['target1'], options['target44'], options['target5'],                       options['target23'], options['target36'], options['target37'], options['target38'],                       options['target39'], options['target40'], options['target41'], options['target42'],                       options['target43']                       ]      decsionTreeSolution(feature\_names) |

运行模型进行训练，训练结束输出结果如图1.2所示：

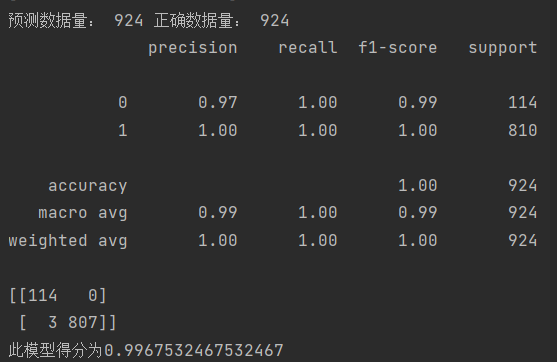


图1.2 训练结果展示

终端打印树结构如图1.3所示，保存整体树图如图1.4所示，保存模型为pkl和pmml文件在目录res下。

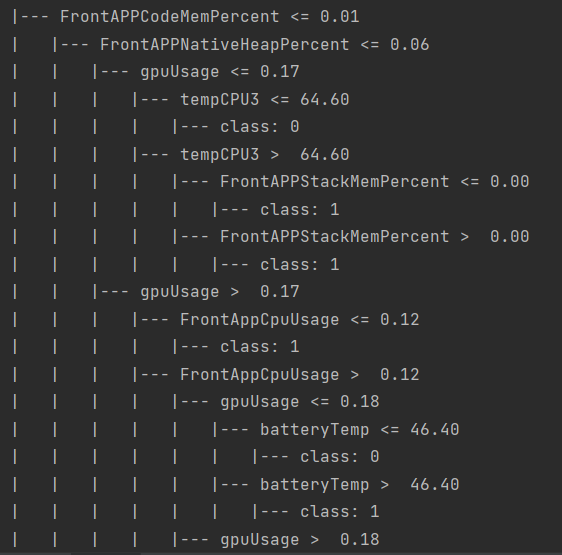


图1.3 终端打印树结果

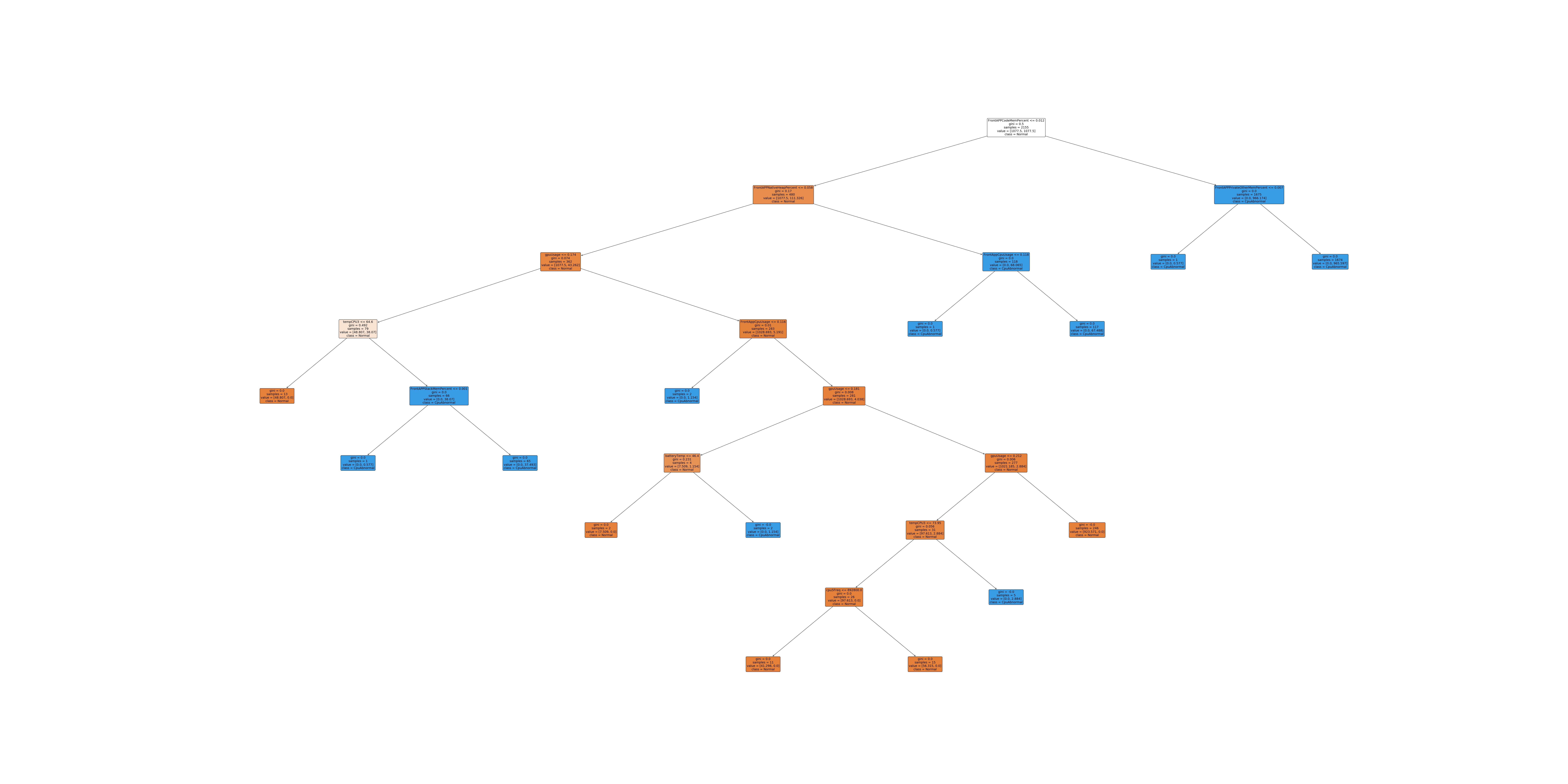


图1.4 保存的异常检测树模型图



图1.5 保存的pmml模型部分内容

# **功耗异常模型训练**

本章先介绍了从功耗异常检测中卷积神经网络模型的一些基本组成部分，再具体说明短时异常和长时异常的模型结构与参数、训练数据集以及训练流程。

## **功耗异常检测模型结构与参数**

功耗异常检测中的模型，包括了短时异常检测和长时异常检测都主要使用了基于卷积神经网络的异常检测模型。与使用全连接层的多层感知机相比，卷积神经网络采用局部感知模式的卷积层，可有效缓解过拟合和参数过多的问题。局部连接权重带来了网络学习的数据局部相关性和位置信息，能够将输入数据映射到更高维度的特征空间。在训练数据量相同的情况下，具有更好的表示能力。卷积神经网络更注重数据的局部相关性和位置信息，因此特别适用于处理和分析图像、视频、语音和文本等结构化数据和时序数据。通常，卷积神经网络由输入层、卷积层、池化层、全连接层和输出层几个组成部分组成。

### 短时异常

短时异常检测算法需要每 30 秒输入一次表 2.1中的特征变量，并判断是否为 CPU 或网络异常。由于 CPU 异常和网络异常一般不会同时出现，本项目将短时数据分为三类：正常，CPU 异常，网络异常，并使用多类别分类算法进行检测。本项目使用了基于全卷积网络（FCN）的深度学习分类算法来检测短时异常。

表2.1 短时异常特征

|  |  |  |
| --- | --- | --- |
| 特征名 | 含义 | 单位 |
| brightness | 亮度 | 0~255 |
| CPUUtilityRate | CPU利用率 | 0~100% |
| CPUCluster0UtilityRate | CPU小核利用率 | 0~100% |
| CPUCluster1UtilityRate | CPU中核利用率 | 0~100% |
| CPUCluster2UtilityRate | CPU大核利用率 | 0~100% |
| services | 正在运行的服务个数 | / |
| foregroundApps | 正在运行的应用个数 | / |
| temperature | CPU温度 | 摄氏度 |
| foregroundConsumption | 前台应用平均耗电 | mA |
| backgroundConsumption | 后台应用平均耗电 | mA |
| CPUConsumption | CPU平均耗电 | mA |
| wifiConsumption | Wifi模块平均耗电 | mA |
| mobileRadioConsumption | 数据网络平均耗电 | mA |
| foregroundAppType | 前台应用类型 | 应用 Category 中的 10 种类型 |
| foregroundCPU | 前台应用 CPU 时间 | jiffies/s |
| backgroundCPU | 后台应用 CPU 时间 | jiffies/s |
| foregroundNetwork | 前台应用网络流量 | Kbytes/s |
| backgroundNetwork | 后台应用网络流量 | Kbytes/s |
| radioType | 网络类型 | 0(其它), 1(Wifi), 2(2G), 3(3G), 4(4G), 5(5G) |
| radioStrength | 网络信号强度 | dBm |

本项目使用了三层的全卷积网络进行短时异常分类，结构如图2.1所示，FCN 中的三个基本块作为特征提取器提取数据的特征，其中的每一个基本块由一个一维卷积层、一个批量归一化层和一个ReLU激活层组成，计算步骤如公式 2.1 所示。三个基本块的一维卷积层的卷积核大小分别为7、5、3；输出通道数分别为128，256，128；步长均为1。最后使用 的平均池化层降低维度并使用 softmax 层求出每类异常的概率，从而输出分类结果。模型优化器为 SGD，每次迭代学习率为0.01。

图示

描述已自动生成

图 2.1 FCN结构

其中，为输入矩阵，为卷积运算符，为卷积核，为偏置变量。

### 长时异常

长时异常指应用在后台频繁触发 Alarm 或频繁触发 JobScheduler、长时间使用传感器和 GPS 模块从而在后台大量耗电导致的异常，这类异常特征较多，且没有固定频率。本项目每隔 90s 采集一次数据，采集的特征变量见表2.2，将 20 段数据连接成为一条包含 30 分钟内数据的多维度时间序列数据。在数据标注中，本项目将这30 分钟内异常应用启动次数大于 2 次或异常发生总时间大于 2 分钟的数据标注为异常。

与短时异常不同，长时异常中一段时间序列里可能会发生多种异常，且可能是由不同的多个应用引起的。因此，长时异常的分类结果应该是发生的所有异常的类型，不可以使用只输出单个分类结果的多类别分类器来解决。本项目采用了 5 个二元分类器的分类方式，这 5 个分类器分别对应 CPU 异常、网络异常、GPS 异常、传感器异常和后台唤醒异常。每个模型都输入相同的所有的特征数据并判断是否有指定的异常类型。

表2.2 长时异常特征

|  |  |  |
| --- | --- | --- |
| 特征名 | 含义 | 单位 |
| wakelock | 最长的 Wakelock 的持续时间 | ms |
| alarmTime | Alarm 触发次数 | / |
| alarmWake | Alarm 触发唤醒次数 | / |
| alarmTotal | 所有 Alarm 触发总时间 | ms |
| jobTime | Job 触发次数 | / |
| jobTotal | 所有 Job 触发总时间 | ms |
| AppCPU | 后台应用 CPU 时间 | jiffies/s |
| AppNetwork | 后台应用网络流量 | Kbytes/s |
| AppCPUConsumption | 后台应用平均 CPU 耗电 | mA |
| APPGPSConsumption | 后台应用平均 GPS 耗电 | mA |
| AppSensorConsumption | 后台应用平均传感器耗电 | mA |

实验尝试了多种机器学习算法来判断是否存在长时异常以及异常类型（实验结果可参考测试报告），综合考虑模型准确率、模型复杂度和资源消耗，本文选择了两层的一维 CNN 模型作为部署的模型，结构如图 2.2所示，模型输入为 的多维时间序列数据，通过两次的卷积层和的平均池化层提取特征并降维，最后经过全连接层和softmax层将特征分类为两类——正常和异常。卷积层的输出通道数分别为6、12；步长均为1，激活函数为 sigmoid。模型优化器为 Adam，每次迭代学习率为0.01，批量大小为 64。

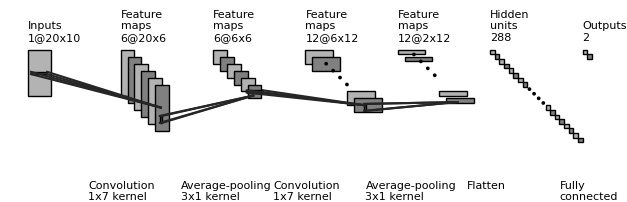


图 2.2 CNN结构

## **功耗异常检测数据集**

本项目的数据采集以一加 9 Pro 手机作为硬件平台，其配置如表2.3所示。为了防止非原生系统可能对实验的干扰，禁用了设备上的 Bootloader 安全机制，并通过 fastboot 刷入了基于 Android 12 的 LineageOS 19.2 类原生操作系统，其 Linux 内核版本为 5.4.191，同时刷入了 Magisk 工具以获取设备上的 Root 权限。数据集中运行的前后台应用如表2.4所示。本项目设计了一个功耗异常模拟程序，可以使用 Service 模拟持续运行并占用资源的异常应用，或调用 Alarm 和 JobScheduler API 周期性运行模拟频繁在后台占用资源的异常应用，异常应用可以使用设备上的 CPU、网络、GPS 和传感器资源。设备异常状态的数据集由异常模拟应用模拟各种异常情景提供。

表2.3 实验测试平台

|  |  |
| --- | --- |
| 相关平台 | 名称 |
| 手机型号 | OnePlus 9 Pro LE2120 |
| Android版本 | Android 12 |
| 操作系统版本 | Lineage OS 19-20221101-NIGHTLY-lemonadep |
| 内存大小 | 12GB(可用内存：10.99GB) |
| CPU版本 | 高通骁龙™ 888 移动平台 |
| CPU核心数目 | 8 |
| GPU版本 | Qualcomm, Adreno (TM) 660 |

表2.4 采集时运行的前后台应用

|  |  |  |  |
| --- | --- | --- | --- |
| 类型 | 应用名 | 包名 | 版本号 |
| 购物 | 淘宝 | com.taobao.taobao | 10.17.0 |
| 视频 | 抖音 | com.ss.android.ugc.aweme | 22.7.0 |
| 游戏 | 原神 | com.miHoYo.Yuanshen | 3.4.0 |
| 社交 | 微博 | com.sina.weibo | 12.10.0 |
| 新闻 | 今日头条 | com.ss.android.article.news | 8.9.9 |
| 地图 | 百度地图 | com.baidu.BaiduMap | 16.4.5 |
| 社交 | 微信 | com.tencent.mm | 8.0.33 |
| 社交 | QQ | com.tencent.mobileqq | 8.9.13 |
| 音乐 | 网易云音乐 | com.netease.cloudmusic | 8.9.50 |

对于短时异常，本项目在实验设备上采集了超过 5000 条（约 41 小时）的数据作为数据集，其中训练集和测试集的比例约为 ；对于长时异常，采集了超过 1300 条（约 217 小时）的数据作为数据集，训练集和测试集的比例约为。相关训练数据以csv表格的形式存储。

* 1. **功耗异常检测模型训练过程**

异常检测作为一个Python语言实现的项目在Windows设备上实现，训练模型的设备主要参数与依赖库版本如下表2.5所示：

表2.5 训练平台

|  |  |
| --- | --- |
| 平台 | 配置 |
| 操作系统 | Windows10/11 |
| 开发平台 | Pycharm 2022.1 |
| Python版本 | Python 3.7 |
| torch | 1.11.0 |
| tqdm | 4.64.1 |
| matplotlib | 3.6.2 |
| pandas | 1.5.2 |
| numpy | 1.22.4 |
| scipy | 1.9.3 |
| seaborn | 0.12.0 |
| scikit-learn | 1.1.3 |
| sktime | 0.16.1 |

模型训练项目目录对代码文件结构如下：

power\_model

├── long-term # 长时异常

│   ├── data # 数据集

│   │   ├── long\_test.csv

│   │   └── long\_train.csv

│   ├── preprocess.py # 预处理程序

│   └── main.py # 长时异常主程序

└── short-term # 短时异常

├── data # 数据集

│   ├── new\_test.arff

│   └── new\_train.arff

├── ConvNet.py # 全卷积网络模型

├── data.py # 数据类

├── datasets.txt # 数据集名称

├── main.py # 深度学习主程序

├── MLmethods.py # 机器学习主程序

├── MultiLayerPerceptron.py # 多层感知机模型

├── train.py # 训练类

└── utils.py # 工具类

下面分别介绍短时异常检测模型和长时异常检测模型的训练过程：

### 短时异常

首先执行short-term文件夹下的 preprocess.py 预处理程序，将数据采集应用采集到的csv文件进行归一化并转换为训练使用的arff格式，然后执行main.py（直接在pycharm运行或者终端运行 “python main.py”）即可自动训练并保存FCN模型。

main文件主要内容如下：包括从dataset.txt文件中读取需要训练的数据集，通过模型进行训练并最后输出模型训练过程中的准确率和损失函数变化图，保存模型为移动端使用的ptl格式。

1. **from** functools **import** reduce
2. **from** utils **import** \*
3. **from** train **import** \*
4. **from** MultiLayerPerceptron **import** \*
5. **from** ConvNet **import** \*
6. **import** matplotlib.pyplot as plt
7. # 参数设置
8. datasets = np.loadtxt('datasets.txt', dtype=str)
9. dataset\_dictionary = data\_dictionary(datasets)
10. epochs = 500
11. device = torch.device('cuda:0' **if** torch.cuda.is\_available() **else** 'cpu')
12. learning\_rate = 0.0001
13. # 读取数据集
14. **for** dataset, dataloader **in** dataset\_dictionary.items():
15. print\_dataset\_info(dataset, dataloader)
16. time\_steps = dataloader['test'].dataset.x.shape[-1]
17. n\_classes = len(np.unique(dataloader['test'].dataset.y))
19. # MLP 模型训练
20. model = MultiLayerPerceptron(time\_steps, n\_classes)
21. model.to(device)
22. model1, history1 = train(dataloader['train'],
23. dataloader['test'],
24. device, model,
25. epochs, learning\_rate)
26. acc(model, dataloader['test'], 'cpu')
27. **print**(reduce(**lambda** x, y: (max(x[0], y[0]), 0), history1)[0])
29. # FCN 模型训练并保存
30. model = ConvNet(time\_steps, n\_classes)
31. model.to(device)
32. model2, history2 = train(dataloader['train'],
33. dataloader['test'],
34. device, model,
35. epochs, learning\_rate)
36. acc(model, dataloader['test'], 'cpu')
37. **print**(reduce(**lambda** x, y: (max(x[0], y[0]), 0), history2)[0])
38. # 绘制训练过程图
39. plt.figure()
40. plt.subplot(121)
41. plt.title("accuracy")
42. plt.xlabel("epoch")
43. plt.ylabel("accuracy")
44. plt.plot(list(map(**lambda** x: x[0], history1)), label='MLP')
45. plt.plot(list(map(**lambda** x: x[0], history2)), label='FCN')
46. plt.legend(loc='center right')
48. plt.subplot(122)
49. plt.title("loss")
50. plt.xlabel("epoch")
51. plt.ylabel("loss")
52. plt.plot(list(map(**lambda** x: float(x[2]), history2)), label='FCN-train')
53. plt.plot(list(map(**lambda** x: x[1], history2)), label='FCN-test')
54. plt.legend(loc='center right')
55. plt.show()

### 长时异常

首先执行long-term文件夹下的preprocess.py 预处理程序，将数据集应用采集到的csv文件进行归一化并转换为训练使用的csv格式，然后执行 main.py（直接在pycharm运行或者终端运行 “python main.py”）。main文件主要内容如下：包括从data目录下读取需要训练的数据集，通过模型进行训练并最后输出模型的准确率，由于机器学习模型没有，保存CNN模型。

1. **import** datetime
2. **from** sktime.classification.dictionary\_based **import** MUSE
3. **from** sktime.classification.kernel\_based **import** RocketClassifier
4. **from** sktime.classification.shapelet\_based **import** ShapeletTransformClassifier
5. **from** sktime.classification.deep\_learning.cnn **import** CNNClassifier
6. **from** sktime.classification.deep\_learning.mlp **import** MLPClassifier
7. **from** sktime.classification.deep\_learning.lstmfcn **import** LSTMFCNClassifier
8. **from** sktime.classification.deep\_learning.tapnet **import** TapNetClassifier
9. **import** numpy as np
10. **import** pandas as pd

13. **def** to\_series(s: str):
14. t = list(map(float, s.split('\\n')[:-1]))
15. **return** np.array(t)
17. # 读取数据集
18. dataset = pd.read\_csv('data/long\_train.csv', sep=',', header=None)
19. data = dataset.iloc[:, 0:10]
20. target = dataset.iloc[:, 10:16]
21. X\_train = np.empty([data.shape[0], data.shape[1], 20], dtype=float)
22. y\_train = np.empty([data.shape[0]])
23. **for** i **in** range(data.shape[0]):
24. **for** j **in** range(data.shape[1]):
25. X\_train[i][j] = to\_series(data.iloc[i, j])
27. testset = pd.read\_csv('data/long\_mytest.csv', sep=',', header=None)
28. **print**(testset.shape)
29. tdata = testset.iloc[:, 0:10]
30. ttarget = testset.iloc[:, 10:16]
31. X\_test = np.empty([tdata.shape[0], tdata.shape[1], 20], dtype=float)
32. y\_test = np.empty([tdata.shape[0]])
33. **for** i **in** range(tdata.shape[0]):
34. **for** j **in** range(tdata.shape[1]):
35. X\_test[i][j] = to\_series(tdata.iloc[i, j])
37. acc = {}
38. # 循环训练每一类异常，异常类型与label对应关系：
39. # ALARM JOB CPU NET GPS SENSOR
40. # 0 1 2 3 4 5
41. **for** predict\_label **in** range(0, 6):
42. **for** i **in** range(target.shape[0]):
43. y\_train[i] = int(target.iloc[i, predict\_label])
44. **for** i **in** range(ttarget.shape[0]):
45. y\_test[i] = int(ttarget.iloc[i, predict\_label])
46. # MUSE模型训练
47. now = datetime.datetime.now()
48. muse = MUSE(window\_inc=4, use\_first\_order\_differences=False)
49. muse.fit(X\_train, y\_train)
50. **print**("MUSE:", (datetime.datetime.now() - now).microseconds / 1000000 + (datetime.datetime.now() - now).seconds)
51. now = datetime.datetime.now()
52. # ROCKET 模型训练
53. rocket = RocketClassifier(num\_kernels=100)
54. rocket.fit(X\_train, y\_train)
55. **print**("rocket:", (datetime.datetime.now() - now).microseconds / 1000000 + (datetime.datetime.now() - now).seconds)
56. now = datetime.datetime.now()
57. # Shapelet模型训练
58. shape = ShapeletTransformClassifier(n\_shapelet\_samples=100)
59. shape.fit(X\_train, y\_train)
60. **print**("shape:", (datetime.datetime.now() - now).microseconds / 1000000 + (datetime.datetime.now() - now).seconds)
61. now = datetime.datetime.now()
62. # MLP 模型训练
63. mlp = MLPClassifier(n\_epochs=200)
64. mlp.fit(X\_train, y\_train)
65. **print**("mlp:", (datetime.datetime.now() - now).microseconds / 1000000 + (datetime.datetime.now() - now).seconds)
66. now = datetime.datetime.now()
67. # CNN模型训练并保存
68. cnn = CNNClassifier(n\_epochs=200)
69. cnn.fit(X\_train, y\_train)
70. cnn.save("cnn" + str(predict\_label))
71. **print**("cnn:", (datetime.datetime.now() - now).microseconds / 1000000 + (datetime.datetime.now() - now).seconds)
72. now = datetime.datetime.now()
73. # LSTM-FCN模型训练
74. lstmfcn = LSTMFCNClassifier(n\_epochs=200)
75. lstmfcn.fit(X\_train, y\_train)
76. **print**("lstmfcn:", (datetime.datetime.now() - now).microseconds / 1000000 + (datetime.datetime.now() - now).seconds)
77. now = datetime.datetime.now()
78. # Tapnet 模型训练
79. tapnet = TapNetClassifier(n\_epochs=200)
80. tapnet.fit(X\_train, y\_train)
81. **print**("tapnet:", (datetime.datetime.now() - now).microseconds / 1000000 + (datetime.datetime.now() - now).seconds)
82. now = datetime.datetime.now()
83. # 将准确率保存到文件中
84. acc\_temp = {"MUSE": mlp.score(X\_test, y\_test),
85. "ROCKET": rocket.score(X\_test, y\_test),
86. "Shapelet": shape.score(X\_test, y\_test),
87. "MLP": mlp.score(X\_test, y\_test),
88. "CNN": cnn.score(X\_test, y\_test),
89. "LSTMFCN": lstmfcn.score(X\_test, y\_test),
90. "TapNet": tapnet.score(X\_test, y\_test)}
91. acc[predict\_label] = acc\_temp
92. **print**(acc)

# **附录**

本阶段提交的文件清单如下所示：

.

├─code

│ ├─performance

│ │ ├─performance\_Decision # 性能异常检测机器学习模型

│ │ ├─pmml-android-master # 决策树模型序列化

│ │ ├─server # 远程服务器代码

│ │ └─updatemonitor # 性能异常检测APP代码

│ └─power

│ ├─power\_model # 功耗异常检测深度学习模型代码

│ ├─powermonitorapp # 功耗异常检测应用

│ └─powerdataacquisitionapp # 功耗异常数据采集应用

├─模型训练手册 # 本文件

├─代码使用手册

└─异常检测程序测试报告