小组成员及分工

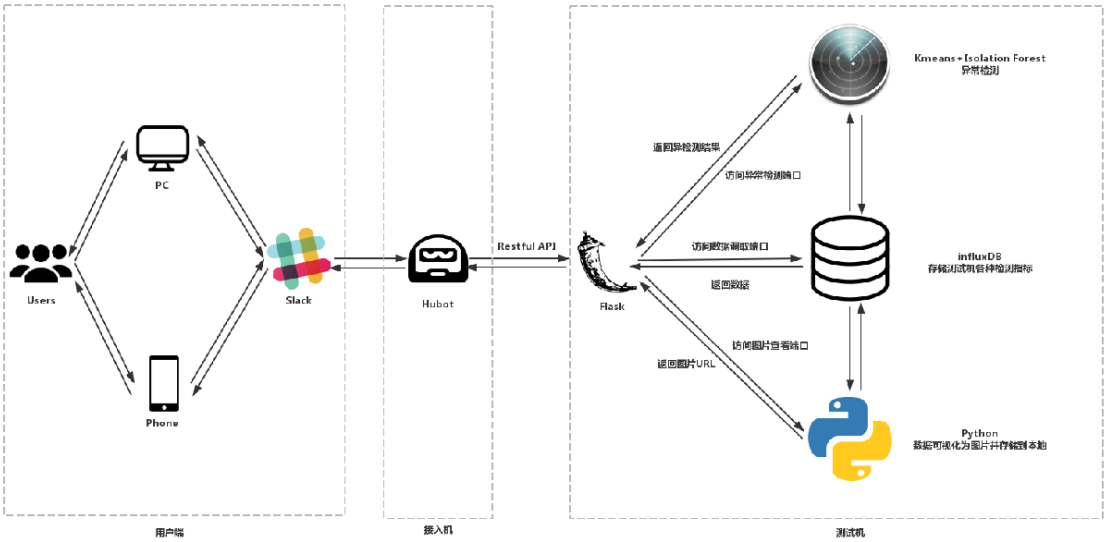
学号	姓名	承担工作
16340048	陈远洋	环境配置，算法
16340196	苏依晴	环境配置，前端
16340198	孙肖冉	环境配置，后端

一、阶段任务

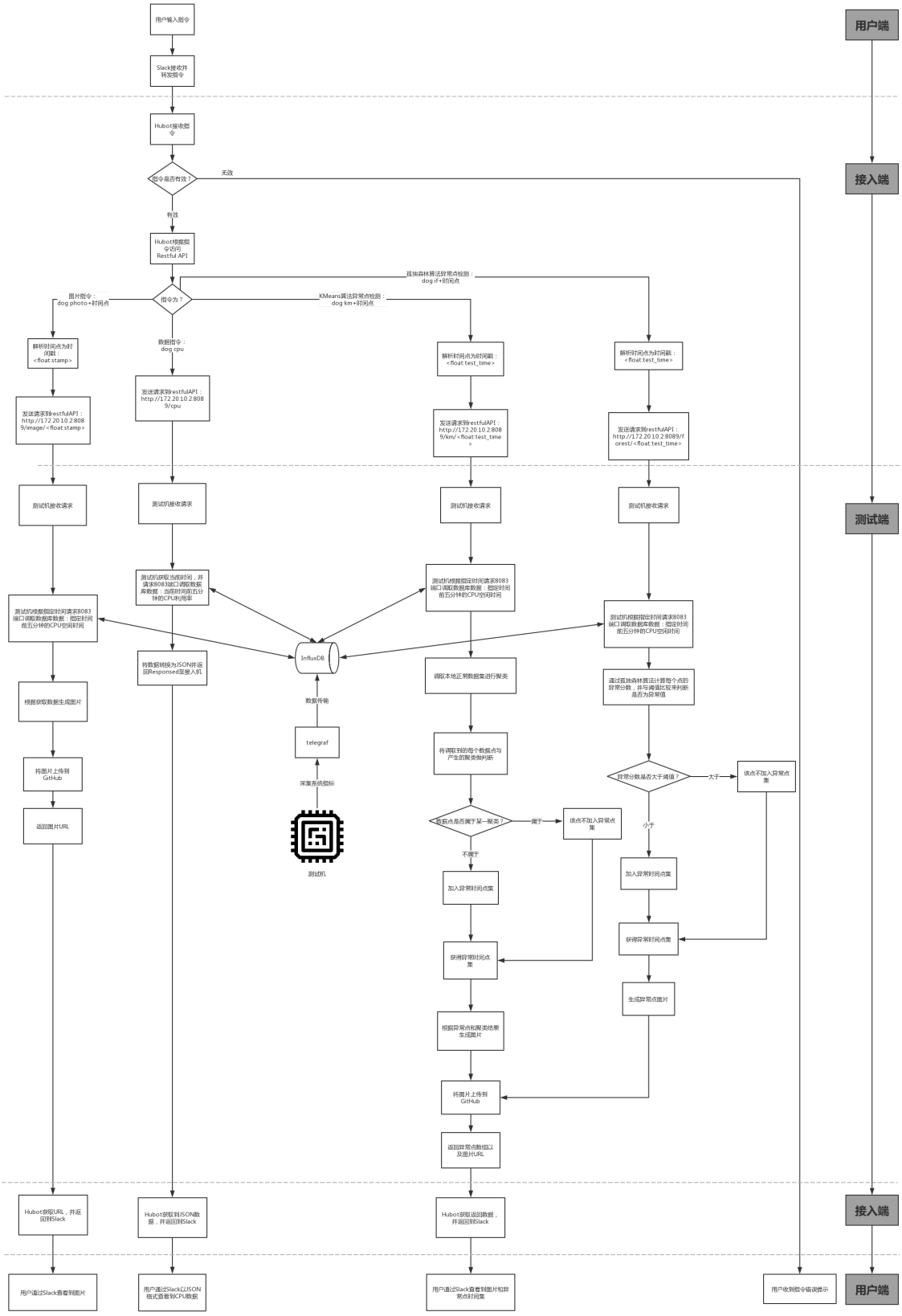
阶段任务	时间	实现情况
环境搭建	第一周	hubot+slack
	第二周	InfluxDB +Telegraf+Grafana
	第三周	
	第四周	Ansible +Nginx
	第五周	
	第六周	前期文档汇总
前后端构建	第七周	查找资料+导出数据
	第八周	
	第九周	restful-api设计+flask学习+apache
	第十周	
算法实现	第十一周	后端其他功能的实现+异常检测K-means算法+训练数据的选择+异常注入
	第十二周	
	第十三周	
	第十四周	异常检测算法 Isolation forest + 实现图片可预览
	第十五周	

二、 实现情况

架构图:


PC
Users
Slack
Phone
用户端
Hubot
接入机
Restful API
Flask
返回异检测结果
访问异常检测端口
访问数据调取端口
返回数据
访问图片查看端口
返回图片URL
Kmeans+Isolation Forest
异常检测
influxDB
存储测试机各种检测指标
Python
数据可视化为图片并存储到本地
测试机

流程图：



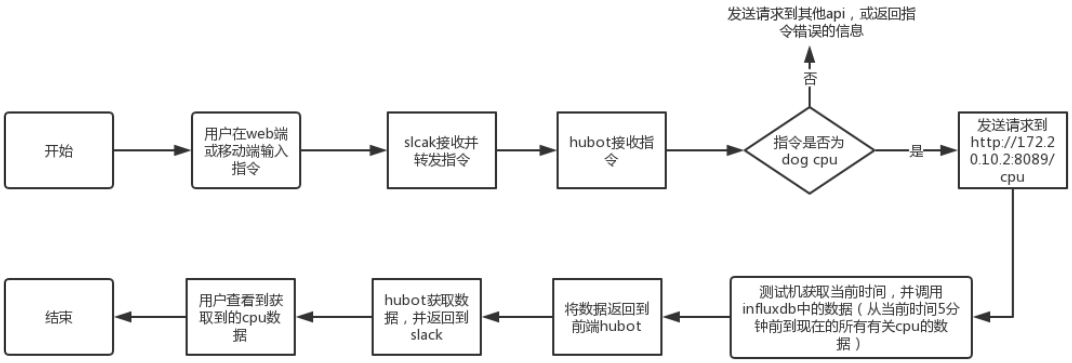
主要功能

功能一：直接读取有关cpu在当前时间前五分钟内的数据信息

命令：dog cpu

返回测试机上的有关cpu的数据

流程图：



具体场景：

RazerSou 7:45 PM
dog cpu

dog APP 7:45 PM


```
[
 [
  {
   "cpu": "cpu-total",
   "host": "sun",
   "time": "2019-06-14T11:45:00Z",
   "usage_guest": 0,
   "usage_guest_nice": 0,
   "usage_idle": 93.80020080321023,
   "usage_iowait": 0.60240963855421,
   "usage_irq": 0,
   "usage_nice": 0,
   "usage_softirq": 0.1255020080321152,
   "usage_steal": 0,
   "usage_system": 0.9538152610441896,
   "usage_user": 4.518072289156289
  },
  {
   "cpu": "cpu0",
   "host": "sun",
   "time": "2019-06-14T11:45:00Z",
   "usage_guest": 0,
   "usage_guest_nice": 0,
   "usage_idle": 94.67336683416899,
   "usage_iowait": 0.5025125628140994,
   "usage_irq": 0,
   "usage_nice": 0,
   "usage_softirq": 0.2010050251256362,
   "usage_steal": 0,
   "usage_system": 1.1055276381909902,
   "usage_user": 3.517587939698339
  },
  {
   "cpu": "cpu1",
   "host": "sun",
```

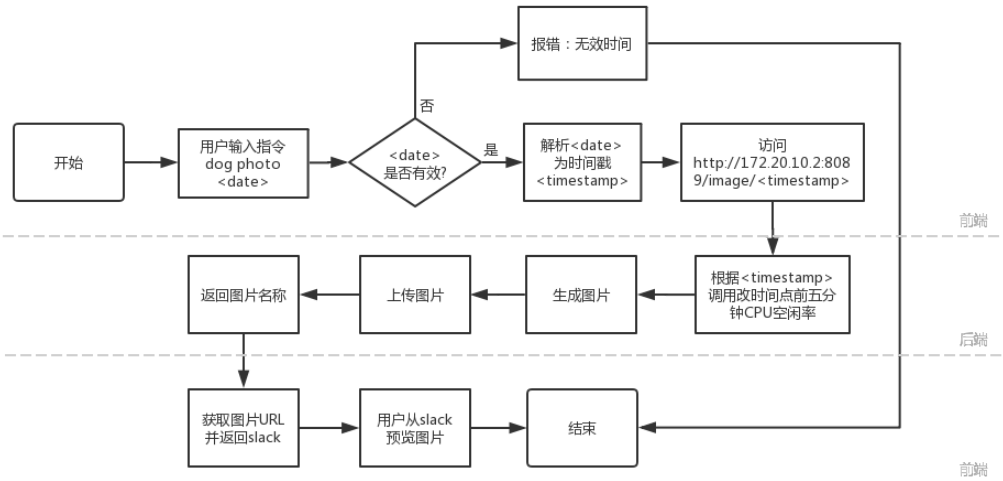

+ Message dog

功能二：获取时间点之前五分钟内的CPU空闲率的数据图像

命令：dog photo 时间点

返回CPU在时间点之前五分钟内的CPU空闲率的数据图像

流程图：



具体场景：

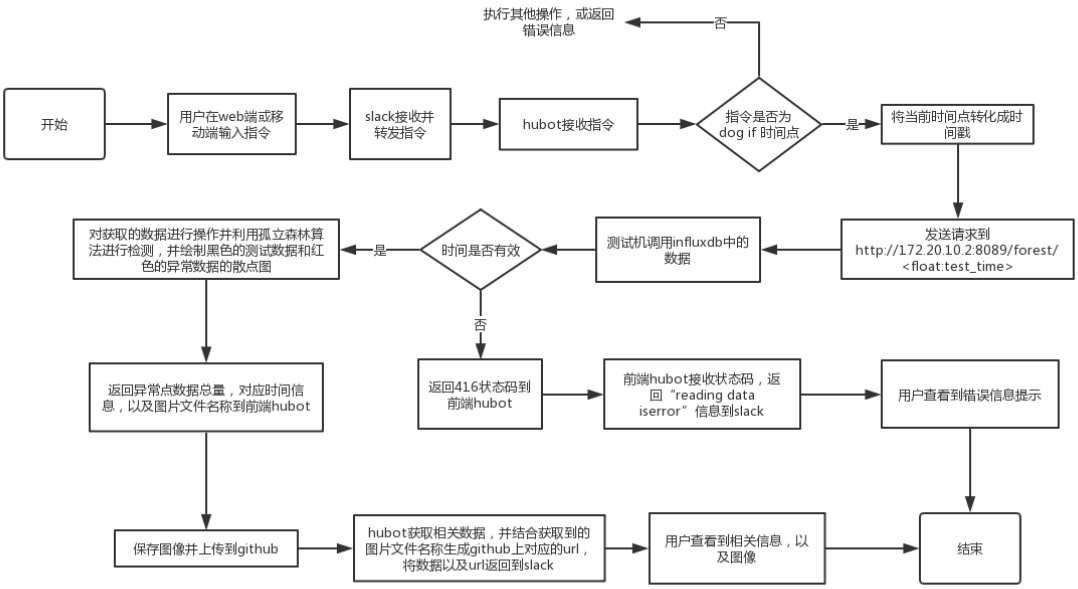
###

功能三：获取时间点前五分钟内的net指标的异常时间点(孤立森林)

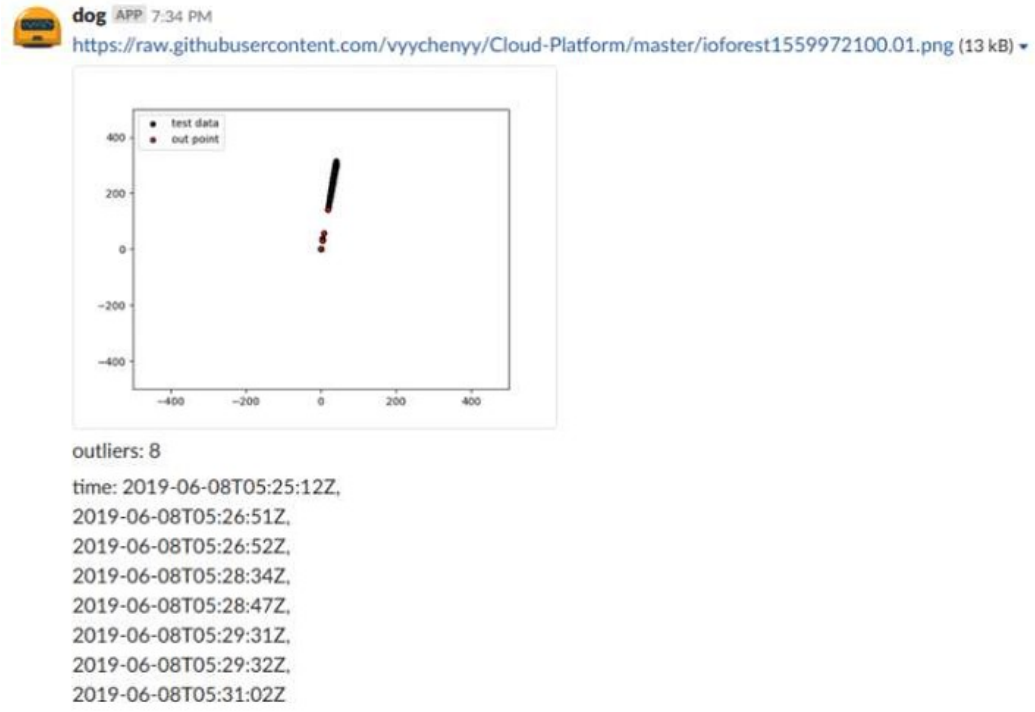
命令：dog if 时间点

返回时间点前五分钟内异常点，图像以及所在时间

流程图：



具体场景：

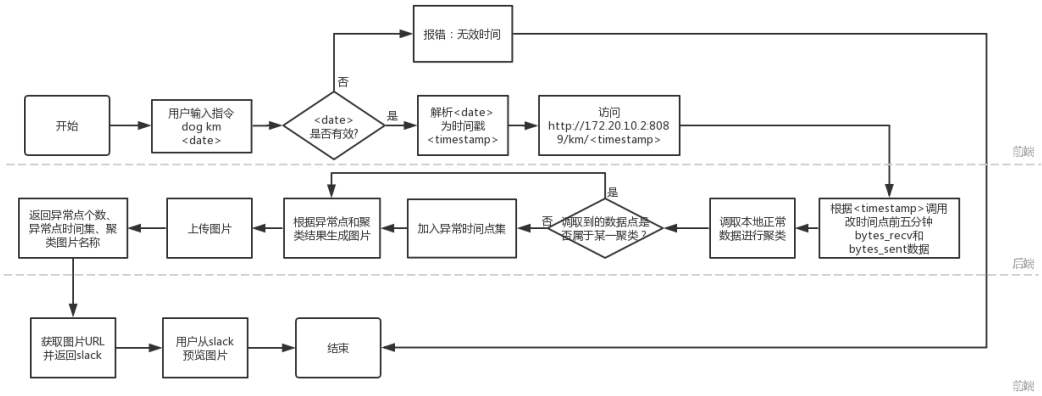


功能四：获取时间点前五分钟内的net指标的异常时间点(K-means)

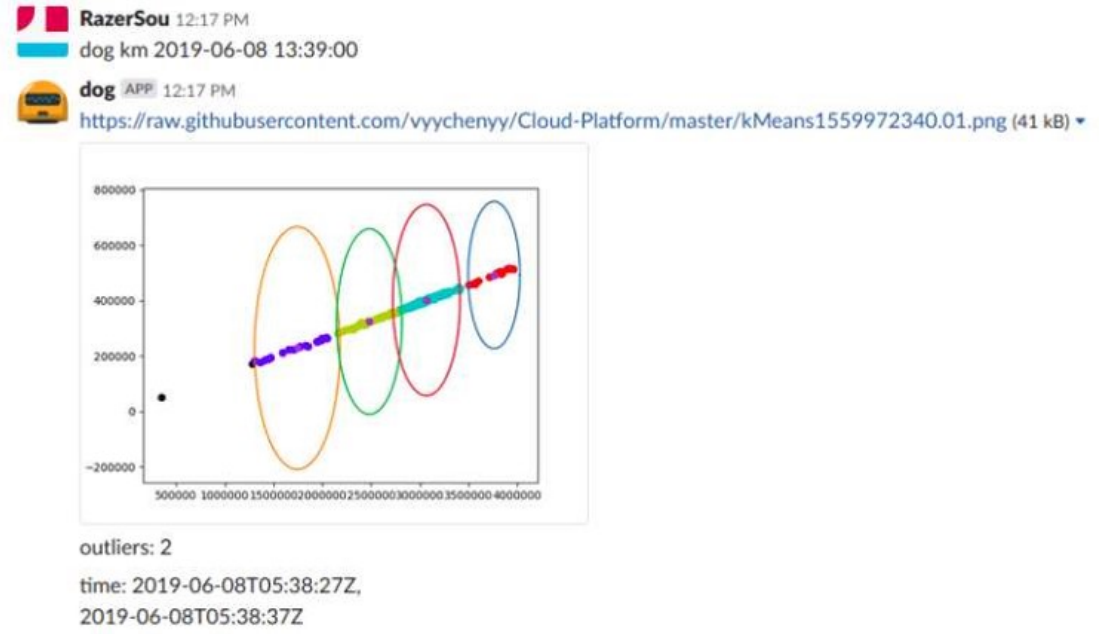
命令格式：dog km 时间点

返回时间点前五分钟内异常点，图像以及所在时间

流程图：



具体场景：



异常检测算法

Isolation forest 算法

简单介绍

孤立森林算法是一种离群点检测算法，它尝试直接去刻画数据的“疏离”程度，较为简单易用。但是该算法对于异常数据有两点要求：

1. 异常数据跟样本中大多数数据不太一样。
2. 异常数据在整体数据样本中占比比较小。

Isolation Forest 是无监督的异常检测算法，在实际应用时需要注意的是：（1）如果训练样本中异常样本的比例比较高，违背了先前提到的异常检测的基本假设，可能最终的效果会受影响；（2）异常检测跟具体的应用场景紧密相关，算法检测出的“异常”不一定是我们实际想要的。所以，在阈值设定时，要根据具体场景进行调整，以免检测出并不异常的“异常数据”。

算法的缺点：iForest不适用于特别高维的数据。由于每次切数据空间都是随机选取一个维度，建完树后仍然有大量的维度信息没有被使用，导致算法可靠性降低。高维空间还可能存在大量噪音维度或无关维度（irrelevant attributes），影响树的构建 。iForest仅对Global Anomaly 敏感，即全局稀疏点敏感，不擅长处理局部的相对稀疏点 （Local Anomaly）。

具体实现

我们在这次算法实现中利用有关net 的bytes_recv和bytes_sent两个参数作为检测对象

调用 **Isolationforest()**函数接口，并使用测试数据进行**拟合**，然后再对数据进行**异常评分**

评分越低代表该点为异常点的可能性越大

与K-means算法作比较，得出该情况下适用的阈值 **-0.15**

若是异常评分低于设定的阈值，我们则认为该点为异常点

重要代码：

- 读取数据

```
client = InfluxDBClient('localhost',port = 8086,database = 'telegraf');
#stamp = time.time()
num1 = int(stamp)
num2 = num1 - 300
str1 = 'select "bytes_recv" , "bytes_sent" from net where time >= '+ str(num2) +'s and time \
      <= '+str(num1)+'s and bytes_recv != 0'
temp = client.query(str1)
test1 = temp.get_points()
```

- 数据处理

```
for item in test1:
  apache1.append(item[u'bytes_recv'])
  apache21.append(item[u'bytes_sent'])
  time_store.append(item[u'time'])
  if j != 0:
    test_req = float(apache1[j]) - float(apache1[j-1])
    test_sec = float(apache21[j]) - float(apache21[j-1])
    origin1.append([test_req/10240,test_sec/10240])
  j += 1

# train = np.array(origin)
ceshi = np.array(origin1)
#print(np.shape(ceshi))
```

- 进行异常检测

```
rng = np.random.RandomState(42)
clf = IsolationForest(max_samples=300, random_state=rng)


clf.fit(ceshi)
anomaly_score = clf.decision_function(ceshi)
#print(anomaly_score)


bad_domains = []
out_point1 = []

threshold = -0.15
i = 0
count = 0

for item in anomaly_score:
  if item < threshold:
    bad_domains.append(time_store[i])
    out_point1.append(origin1[i])
    count += 1
  i += 1
```

K-Means 算法

简单介绍

k-means算法是一种聚类算法，所谓聚类，即根据相似性原则，将具有较高相似度的数据对象划分至同一类簇，将具有较高相异度的数据对象划分至不同类簇。聚类与分类最大的区别在于，聚类过程为无监督过程，即待处理数据对象没有任何先验知识，而分类过程为有监督过程，即存在有先验知识的训练数据集。

- 算法思路： 先随机选取K个对象作为初始的聚类中心。然后计算每个对象与各个种子聚类中心之间的距离，把每个对象分配给距离它最近的聚类中心。聚类中心以及分配给它们的对象就代表一个聚类。一旦全部对象都被分配了，每个聚类的聚类中心会根据聚类中现有的对象被重新计算。这个过程将不断重复直到满足某个终止条件。终止条件可以是以下任何一个： 1)没有（或最小数目）对象被重新分配给不同的聚类。 2)没有（或最小数目）聚类中心再发生变化。 3)误差平方和局部最小。

输入：类簇个数 K，迭代终止阈值 δ
输出：聚类结果

For t=1,2,...,T
    For every x_i（对于所有数据对象）
        根据公式(1)，计算 $\mathrm{dist}(x_i, \mathrm{Center}_k)$；
        将 x_i 划分至距离其最近的类簇中心所在类簇中；
    End for
    根据公式(2)，更新所有类簇中心；
    根据公式(3)，计算两次迭代的差值 ΔJ；
    If $\Delta J < \delta$
        Then 输出聚类结果；
        break;
    End if
End for

- 将K-Means算法应用于异常点检测步骤： a.从数据集中随机挑K个数据当簇心； b.对数据中的所有点求到这K个簇心的距离，假如点Pi离簇心Si最近，那么Pi属于Si对应的簇； c.根据每个簇的数据，更新簇心，使得簇心位于簇的中心； d.重复步骤b和步骤c，直到簇心不再移动，继续下一步； e.计算每个聚类的最大半径，即每个聚类中距离簇心最远的点的距离； f.传入一个检测点，计算该点属不属于某个聚类所在的最大圆内，若属于则说明非异常的，反之则为异常点。
- 优点： 算法简单易实现；
- 缺点： 需要用户事先指定类簇个数； 聚类结果对初始类簇中心的选取较为敏感； 容易陷入局部最优； 只能发现球形类簇；

具体实现

我们这次使用了本地固定的正常数据文件为训练集生成聚类，传入两个维度的数据进行k=4的算法聚类，在生成聚类后，进行每个聚类内最大半径的计算，完成计算后即可生成4个圆形区域，此时即可传入测试点检测数据是否为异常点，若测试点落在四个圆形之内，则说明不是异常点，反之则为异常点。

重要代码：

- 读取数据

```
# 加载数据
def loadDataSet(fileName):  # 解析文件，按逗号分割字段，得到一个浮点数字类型的矩阵
    dataMat = []                # 文件的最后一个字段是类别标签
    with open('net.csv') as file:
      reader = csv.DictReader(file)
      test = [test for test in reader]

    key1 = 'bytes_recv'
    key2 = 'bytes_sent'
    i = 1
    while i < len(test):
        temp = []
        num1 = float(test[i]['bytes_sent']) - float(test[i-1]['bytes_sent'])
        num2 = float(test[i]['bytes_recv']) - float(test[i-1]['bytes_recv'])
        temp.append(num1)
        temp.append(num2)
        dataMat.append(temp)
        i += 1
    return dataMat
```

- 聚类算法

```
# k-means 聚类算法
def kMeans(dataSet, k, distMeans =distEclud, createCent = randCent):
    m = shape(dataSet)[0]
    clusterAssment = mat(zeros((m,2)))    # 用于存放该样本属于哪类及质心距离
    # clusterAssment第一列存放该数据所属的中心点，第二列是该数据到中心点的距离
    centroids = createCent(dataSet, k)
    clusterChanged = True   # 用来判断聚类是否已经收敛
    while clusterChanged:
        clusterChanged = False;
        for i in range(m):  # 把每一个数据点划分到离它最近的中心点
            minDist = inf; minIndex = -1;
            for j in range(k):
                distJI = distMeans(centroids[j,:], dataSet[i,:])
                if distJI < minDist:
                    minDist = distJI; minIndex = j  # 如果第i个数据点到第j个中心点更近，则将i归属为j
            if clusterAssment[i,0] != minIndex: clusterChanged = True;  # 如果分配发生变化，则需要继续迭代
            clusterAssment[i,:] = minIndex,minDist   # 并将第i个数据点的分配情况存入字典
        #print centroids
        for cent in range(k):   # 重新计算中心点
            ptsInClust = dataSet[nonzero(clusterAssment[:,0].A == cent)[0]]   # 去第一列等于cent的所有列
            if len(ptsInClust)!=0:
                centroids[cent,:] = mean(ptsInClust, axis = 0)  # 算出这些数据的中心点
    return centroids, clusterAssment
```

- 进行异常检测

```
def maxDis(centroids,clusterAssment):
    max = zeros(4)
    flag = 1 #返回1则表示为异常点
    for i in range(len(clusterAssment)):
        tep = int(clusterAssment[i,0])
        if clusterAssment[i,1]>max[tep]:
            max[tep] = clusterAssment[i,1]
    return max

def detect(max,centroids,clusterAssment,x,y):
    point = array([x, y])
    for j in range(len(max)):
        dis = distEclud(centroids[j,:],point[0])
        if dis < max[j] and max[j] != 0:  #则属于其中一个有效聚类
            flag = 0
            break
    return flag
```

三、 遇到的问题及解决方法

图片自动化下载问题

问题描述：无法实现从grafana 自动下载图片

解决方案：后端直接借用matplotlib库绘制图片

另一种可行方案：使用ansible ，调用hubot的shellcmd 使用grafana的curl +api-key + url 的命令语句自动化下载图片

Slack预览图片问题

问题描述：我们将绘制的图片存到自己搭建的内网服务器上，输入对应命令，slack会接收到对应的url，但是无法访问、预览内网图片

解决方案：刚开始我们选择让后端将response 的格式设为img/png类型，只返回url，但需要点击链接才可访问，最后我们选择由后端将生成的图片存储在github的本地分支中，并使用git语句自动上传到github，返回生成图片的名称到前端，然后由前端设置对应的url，并推送给slack，slack可以直接预览到对应的图片

InfluxDB的配置问题

问题描述：无法使用influxdb 的8083管理页面端口查看数据

解决方案：查看资料发现这个问题是influxdb的版本问题造成的，influxbd1.3以上就不再支持8083端口的访问，但是无法进行版本降级，也没有查找到uninstall文件，于是我们重新装载了ubuntu系统并重新下载了版本较低的influxdb

异常检测的相关问题

1.问题描述：无法正确选取训练数据

解决方案：我们不了解数据库中参数的对应含义，无法正确选取要进行异常检测的数据，后来在网上查找资料找到了influxdb的相关文档，查阅文档并询问老师之后，选择了有关net的bytes_recv和bytes_send两个参数对应的数据，将数据处理之后，进行训练和异常注入

另一种待解决方案：选取多种参数，在注入异常的情况下观察grafana的对应图像变化，选取数据，其他小组的展示中有提到cpu的部分数据也与丢包、延时的异常有关

2.问题描述：在进行算法评价时，有关异常注入的部分遇到了一些困难。

解决方案：在测试异常时，一开始我们参考了其他同学的方法，尝试用跑循环的方式制造CPU异常，但是跑了几分钟后还是完全没有效果。后来我们使用了siege和tc两个工具，接入端这边用siege给测试端发送get请求，然后测试端这边用tc来注入丢包和网络延迟，这时产生的有关net的bytes_recv和bytes_send两个参数对应的数据可以被认为是异常，但是我们无法确定异常点的个数，所以无法利用这样的数据对异常检测算法进行评估。后来发现可以手动改变数据注入异常值，但是这样与现实情况可能又有较大出入，只能简单地进行评估。

四、 后续工作

增加采集数据的种类

我们这次只使用了Telegraf这个工具来采集数据，而采集的数据都是一些性能参数。根据开题时的讲座，我们之后可以增添其他工具，用来采集其他方面的数据，比如各种日志内容，端到端的响应时间等等。前者可以帮助我们记录完整日志，让运营人员可以方便地查看，后者可以排查系统性能或者记录可聚合的数据等等。

增加异常检测算法

每种算法适合检测的异常不同，这次我们使用的两个算法是K-means和Isolation Forest。K-means容易实现，在簇近似为高斯分布时效果很好，但只收敛到局部最优。Isolation Forest是离群点的检测，只根据异常数据少且不同的特点找出异常，但是不适用于特别高维的数据，也不擅长处理局部的相对稀疏点。所以在异常检测算法方面，可以根据需求，多加一些检测算法，比如时间序列建模ARIMA，可以处理低维的稳定数据。又比如One-Class SVM，在高维空间下或特征维数大于样本数时仍有效，适合数值型和标称型数据。

优化日志存储

目前运维人员能通过hubot查询某个时间点前五分钟的数据和图像，但是不能查询完整日志。之后可以增加让用户查询完整日志以及异常日志的功能，方便运维人员操作。

把图片数据库和GitHub目录分开

我们项目目前图片是直接上传到GitHub项目的根目录下的，难以管理，但由于我们用的时python的os模块，而不是GitPython，没有足够的资料可以查询，所以到项目结束时还没有找到解决方法。之后我们可以通过os模块把图片传到我们之前建立的图片服务器来解决这个问题，转用GitPython库或许也可以解决这个问题。

增加功能

为了能让运维人员直接对有问题机器进行操作解决问题，应该要增加重启重配置功能。参考其他同学的做法，我们可以通过接入端使用shellcmd启动ansible来操控测试机重启重配置。