目录

[一、背景](#_Toc9074)

[1.云计算框架](#_Toc30519)

[2.显示层](#_Toc20203)

[3.中间层](#_Toc4152)

[4.基础设施层](#_Toc28353)

[5.管理层](#_Toc14538)

[二、树莓派数据处理](#_Toc753)

[1.配置树莓派](#_Toc24613)

[2.收集数据](#_Toc14399)

[3.数据处理并存储](#_Toc14528)

[三、数据挖掘](#_Toc21757)

[1.58同城二手车的信息爬取](#_Toc15993)

[2. 猫眼电影网《风语咒》的影评爬取](#_Toc16668)

[四、分类与回归算法](#_Toc20189)

[1. 算法概述](#_Toc11217)

[2. 分类算法种类以及特征](#_Toc27836)

[2.1 决策树（decision tree）分类算法](#_Toc9043)

[3.1案例背景](#_Toc20858)

[五、聚类算法](#_Toc22386)

[1. K-MEANS聚类算法](#_Toc16818)

[2.均值偏移聚类算法](#_Toc17956)

[3.DBSCAN聚类算法](#_Toc2089)

[4.使用高斯混合模型（GMM）的期望最大化（EM）聚类](#_Toc18744)

[5. 层次聚类算法](#_Toc20773)

[6. k-means聚类算法](#_Toc30405)

[6.1 算法简介](#_Toc12700)

[6.2代码实现案例](#_Toc4641)

[六、关联规则](#_Toc16612)

[1.什么是关联规则](#_Toc31061)

[2.概念](#_Toc8387)

[3.实现过程](#_Toc7018)

[4.代码实现案例](#_Toc28428)

[七、离群点挖掘](#_Toc9745)

[1.概述](#_Toc13621)

[2.基于相对密度的方法实现案例](#_Toc19809)

[八、神经网络](#_Toc19846)

[1.概述](#_Toc26314)

[2. 神经网络实现边界决策](#_Toc13009)

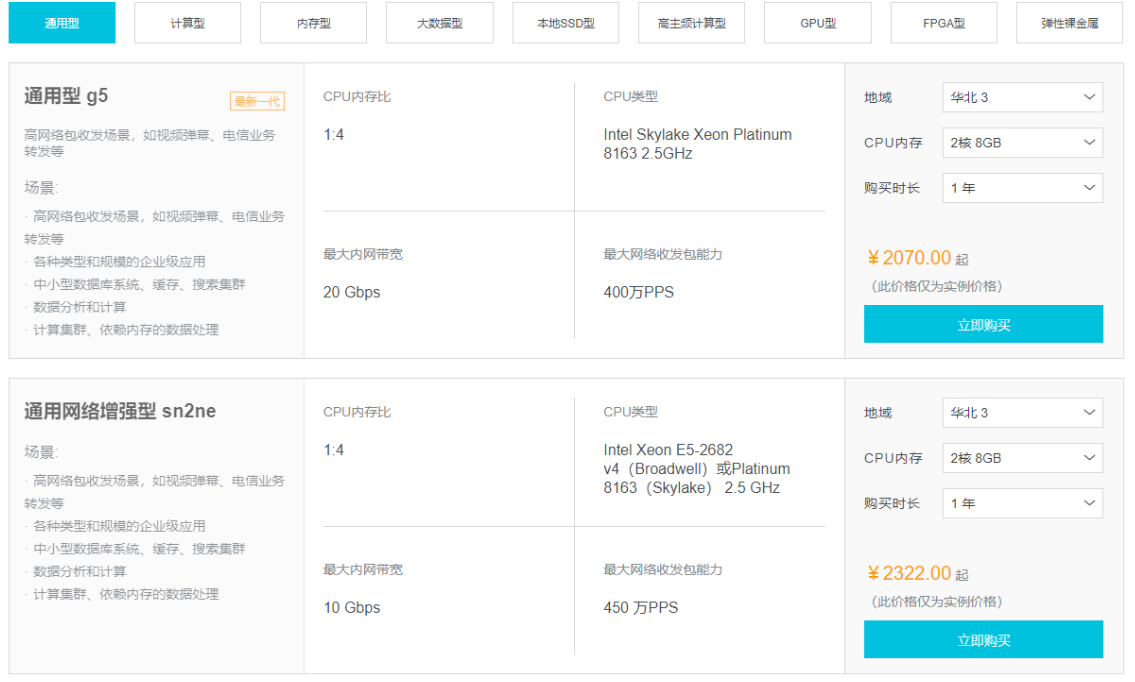
# 一、背景

所谓的云，实质上是服务器计算机和相应的应用服务组成的整体。云计算的意义在于资源的共享。如果企业收集到一组数据，要对数据进行处理，当然可以选择使用企业拥有的电脑来处理。但是，如果这组数据需要处理的内容足够多，需要好几台电脑一天24小时不停运行处理程序，运行好几天仍然得不到最终结果，那么，为了支撑这些不断增长的需求，企业不得不去购买各类硬件设备（计算机，存储，带宽等等）和软件（数据库，中间件等等），另外还需要组建一个完整的运维团队来支持这些设备或软件的正常运作，这些维护工作就包括安装、配置、测试、运行、升级以及保证系统的安全等。便会发现支持这些应用的开销变得非常巨大，而且它们的费用会随着你应用的数量或规模的增加而不断提高。

所以，云计算，应运而生——更大、更快、更强。针对上述问题解决方案便是“云计算”！将应用部署到云端后，可以不必再关注那些令人头疼的硬件和软件问题，它们会由云服务提供商的专业团队去解决。使用的是共享的硬件，这意味着像使用一个工具一样去利用云服务（就像插上插座，你就能使用电一样简单）。

只需要按照你的需要来支付相应的费用，而关于软件的更新，资源的按需扩展都能自动完成。例如，运行某个网络服务，原本需要买100台服务器维护和管理，云的概念出现以后，现在只需要花十分之一的钱直接租用云服务器，企业老总看着财务报表睡觉都能笑醒了。

图为阿里云云服务器截图:



图一 云服务器租用界面

## 1.云计算框架

### 1.1显示层

多数数据中心云计算架构的这层主要是用于以友好的方式展现用户所需的内容和服务体验，并会利用到下面[中间件](https://baike.baidu.com/item/中间件#_blank)层提供的多种服务，主要有五种技术：

HTML：标准的Web页面技术，现在主要以[HTML4](https://baike.baidu.com/item/HTML4#_blank)为主，但是将要推出的HTML5会在很多方面推动Web页面的发展，比如视频[1]和本地存储等方面。

JavaScript：一种用于Web页面的[动态语言](https://baike.baidu.com/item/动态语言#_blank)，通过JavaScript，能够极大地丰富Web页面的功能，并且用以JavaScript为基础的AJAX创建更具交互性的[动态页面](https://baike.baidu.com/item/动态页面#_blank)。

CSS：主要用于控制Web页面的外观，而且能使页面的内容与其表现形式之间进行优雅地分离。

Flash[2]：业界最常用的RIA(Rich Internet Applications)技术，能够在现阶段提供HTML等技术所无法提供的基于Web的富应用，而且在用户体验[3]方面，非常不错。

Silverlight：来自业界巨擎[微软](https://baike.baidu.com/item/微软#_blank)[4]的RIA技术，虽然其现在市场占有率稍逊于Flash，但由于其可以使用C#[5]来进行编程，所以对开发者非常友好。

## 1.2中间层

这层是承上启下的，它在下面的基础设施层所提供资源的基础上提供了多种服务，比如[缓存服务](https://baike.baidu.com/item/缓存服务#_blank)和REST服务等，而且这些服务即可用于支撑显示层，也可以直接让用户调用，并主要有五种技术：

**REST：**通过REST技术，能够非常方便和优雅地将[中间件](https://baike.baidu.com/item/中间件#_blank)层所支撑的部分服务提供给调用者。

**多租户：**就是能让一个单独的应用实例可以为多个组织服务，而且保持良好的隔离性和安全性，并且通过这种技术，能有效地降低应用的购置和维护成本。

[**并行处理**](https://baike.baidu.com/item/并行处理#_blank)**：**为了处理海量的数据，需要利用庞大的X86[集群](https://baike.baidu.com/item/集群#_blank)进行规模巨大的并行处理，Google的MapReduce是这方面的代表之作。

[**应用服务器**](https://baike.baidu.com/item/应用服务器#_blank)**：**在原有的应用服务器的基础上为云计算做了一定程度的优化，比如用于Google App Engine的Jetty应用服务器。

**分布式缓存：**通过分布式缓存技术，不仅能有效地降低对后台服务器的压力，而且还能加快相应的反应速度，最著名的分布式缓存例子莫过于Memcached。

## 1.3基础设施层

这层作用是为给上面的[中间件](https://baike.baidu.com/item/中间件#_blank)层或者用户准备其所需的计算和存储等资源，主要有四种技术：

**虚拟化：**也可以理解它为基础设施层的“多租户”，因为通过[虚拟化技术](https://baike.baidu.com/item/虚拟化技术#_blank)，能够在一个物理服务器上生成多个虚拟 机，并且能在这些虚拟机之间能实现全面的隔离，这样不仅能减低服务器的购置成本，而且还能同时降低服务器的运维成本，成熟的X86虚拟化技术有 VMware的ESX和开源的Xen。

**分布式存储：**为了承载海量的数据，同时也要保证这些数据的可管理性，所以需要一整套分布式的[存储系统](https://baike.baidu.com/item/存储系统#_blank)。

[**关系型数据库**](https://baike.baidu.com/item/关系型数据库#_blank)**：**基本是在原有的关系型数据库的基础上做了扩展和管理等方面的优化，使其在云中更适应。

NoSQL：为了满足一些关系数据库所无法满足的目标，比如支撑海量的数据等，一些公司特地设计一批不是基于关系模型的数据库。

## 1.4管理层

这层是为横向的三层服务的，并给这三层提供多种管理和维护等方面的技术，主要有下面这六个方面：

[帐号](https://baike.baidu.com/item/帐号#_blank)管理：通过良好的帐号管理技术，能够在安全的条件下方便用户地登录，并方便管理员对帐号的管理。

SLA监控：对各个层次运行的[虚拟机](https://baike.baidu.com/item/虚拟机#_blank)，服务和应用等进行性能方面的监控，以使它们都能在满足预先设定的SLA(Service Level Agreement)的情况下运行。

计费管理：也就是对每个用户所消耗的资源等进行统计，来准确地向用户索取费用。

安全管理：对数据，应用和帐号等IT[6]资源采取全面地保护，使其免受犯罪分子和恶意程序的侵害。

[负载均衡](https://baike.baidu.com/item/负载均衡#_blank)：通过将流量分发给一个应用或者服务的多个实例来应对突发情况。

[运维管理](https://baike.baidu.com/item/运维管理#_blank)：主要是使运维操作尽可能地专业和自动化，从而降低云计算中心的运维成本。

云计算架构其中有三层是横向的，分别是显示层、[中间件](https://baike.baidu.com/item/中间件#_blank)层和基础设施层，通过这三层技术能够提供非常丰富的云计算能力和友好的用户界面，云计算架构还有一层是纵向的，称为管理层，是为了更好地管理和维护横向的三层而存在的。

图1.2

# 二、树莓派数据处理

模拟云计算的场景。收集数据并对数据进行分析处理。

课程使用树莓派作为云服务器，电脑使用ssh连接，python语言收集并处理数据。python爬虫收集网址，然后对网址进行分析并存储。

严格意义上讲，这样的一个简单流程离真正的云计算应用场景相去甚远，但是它却反映了云计算的基本过程：收集数据，处理数据，存储数据。

## 配置树莓派

使用putty，ssh连接树莓派

首先需要连接wifi或网线。

如下图，依次打开 菜单(Menu) > 首选项(Preferences) > Raspberry Pi Configuration：

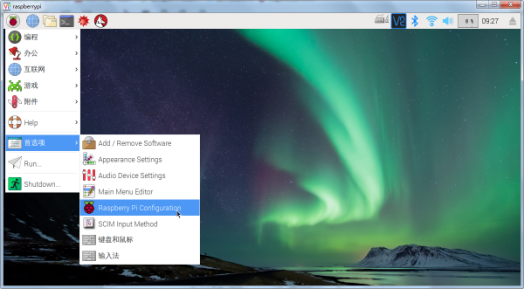


图2.1

点击 Interfaces 栏，选择“enable” SSH服务。如下图：

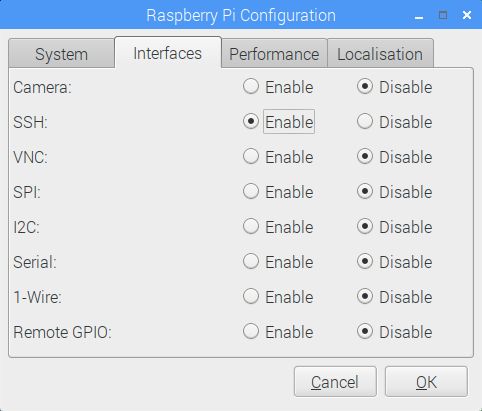


图2.2

使用ifconfig命令查询树莓派ip

如图中ip为192.168.123.18

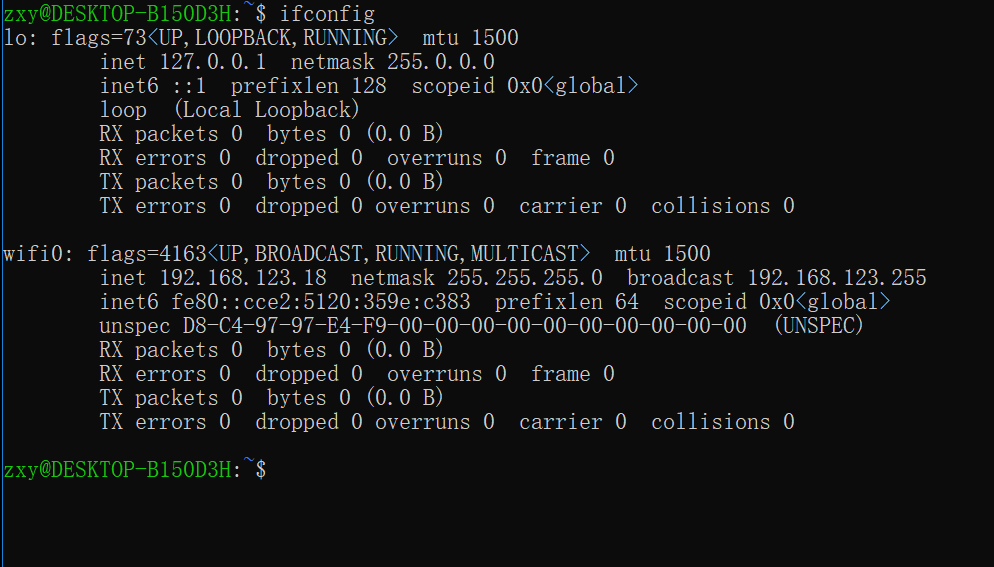


图2.3

使用电脑打开putty，在host输入树莓派的ip地址

然后点击open

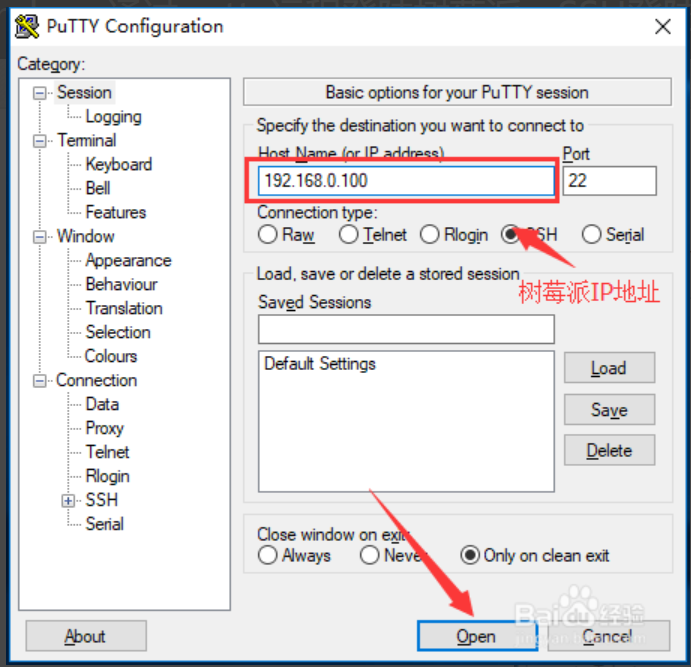


图2.4

输入用户名和密码

用户名：pi

密码：raspberry

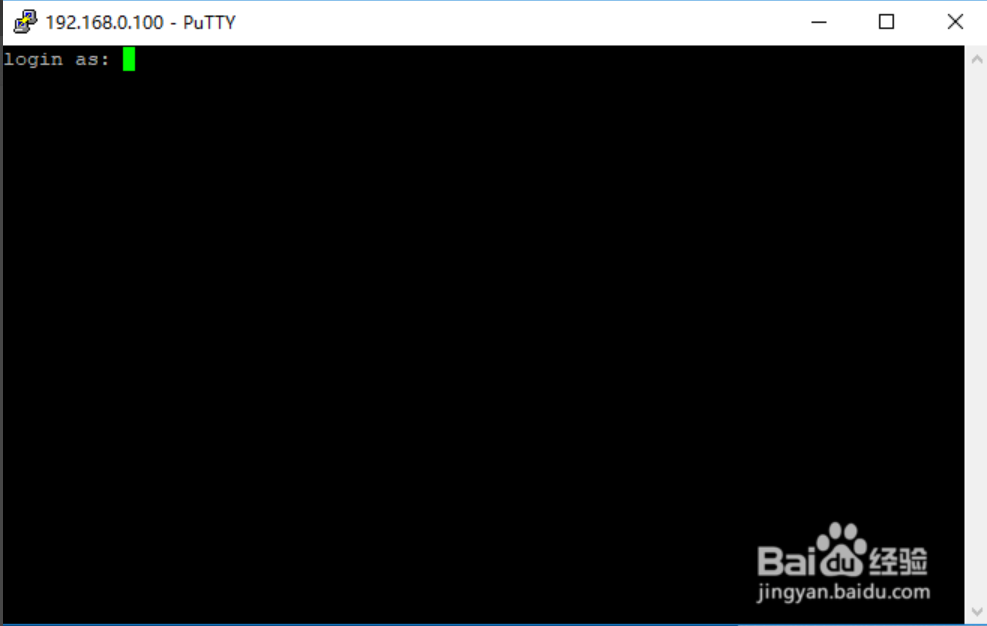


图2.5

ssh登陆成功之后，就可以对树莓派进行命令行进行操作了（ssh只有命令行模式）

需要使用命令行安装组件

sudo apt-get install mysql-server python3 python3-pip whois

pip3 install install mysql-connector

测试mysql：

输入指令mysql -u root -p

得到以下结果：

Enter password: （默认没有密码，回车即可）

Welcome to the MySQL monitor. Commands end with ; or \g.

Your MySQL connection id is 47

Server version: 5.5.41-0+wheezy1 (Debian)

Copyright (c) 2000, 2014, Oracle and/or its affiliates. All rights reserved.

Oracle is a registered trademark of Oracle Corporation and/or its

affiliates. Other names may be trademarks of their respective

owners.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

Python3

python语言常用于数据挖掘和分析。它的语法非常简单，很多人对它评价是“人生苦短，我用python”。同样的事情用python往往只需要写java的一半代码量甚至更少就可以完成。但是它的运行效率和稳健性常常比java要差很多。

基本的语法可以在附录中找到。

收集网址并对网址数据进行分析和处理

## 收集数据

使用python3收集网址,然后进行简单处理。

.py文件放在同一目录下，然后依次运行

python3 getweb.py

python3 guolv.py

python3 guolv2.py

python3 guolv3.py

源代码：

文件命名为getweb.py，内容如下

#解析网页导航并将结果导入txt文档

import urllib.request #调用包

def getweb(argv): #定义函数

request=urllib.request.Request(argv) #

response=urllib.request.urlopen(request) #

html=response.read() #

htm=str(html, encoding = "utf-8") #

open('result-readline.txt', 'w').write(htm) #打开文件并将结果写入txt文档

getweb("[http://www.369.com](http://www.369.com/)")

文件命名为guolv.py，内容如下

import re

f = open('result-readline.txt','r') #以读取模式打开文件

fo = open("result2.txt", "w") #以写入模式打开文件

for line in open('result-readline.txt'): #逐行读取

line = f.readline()

if line == "\n":

n = 1

else:

reg = re.compile('"http.\*?"') #使用正则表达式进行字符匹配

m1 = reg.findall(line)

print(m1)

fo.writelines(m1)

fo.writelines("\n")

f.close()

fo.close()

Python3 正则表达式

正则表达式是一个特殊的字符序列，它能帮助你方便的检查一个字符串是否与某种模式匹配。

以上代码中reg = re.compile('"http.\*?"')就是匹配以”http开头,”结尾的字符串，在这里就是网页地址。

文件命名为guolv2.py，内容如下：

# coding = utf-8

def clearBlankLine():

file1 = open('result2.txt', 'r', encoding='utf-8') # 要去掉空行的文件

file2 = open('result32.txt', 'w', encoding='utf-8') # 生成没有空行的文件

try:

for line in file1.readlines():

if line == '\n':

line = line.strip("\n")

file2.write(line)

finally:

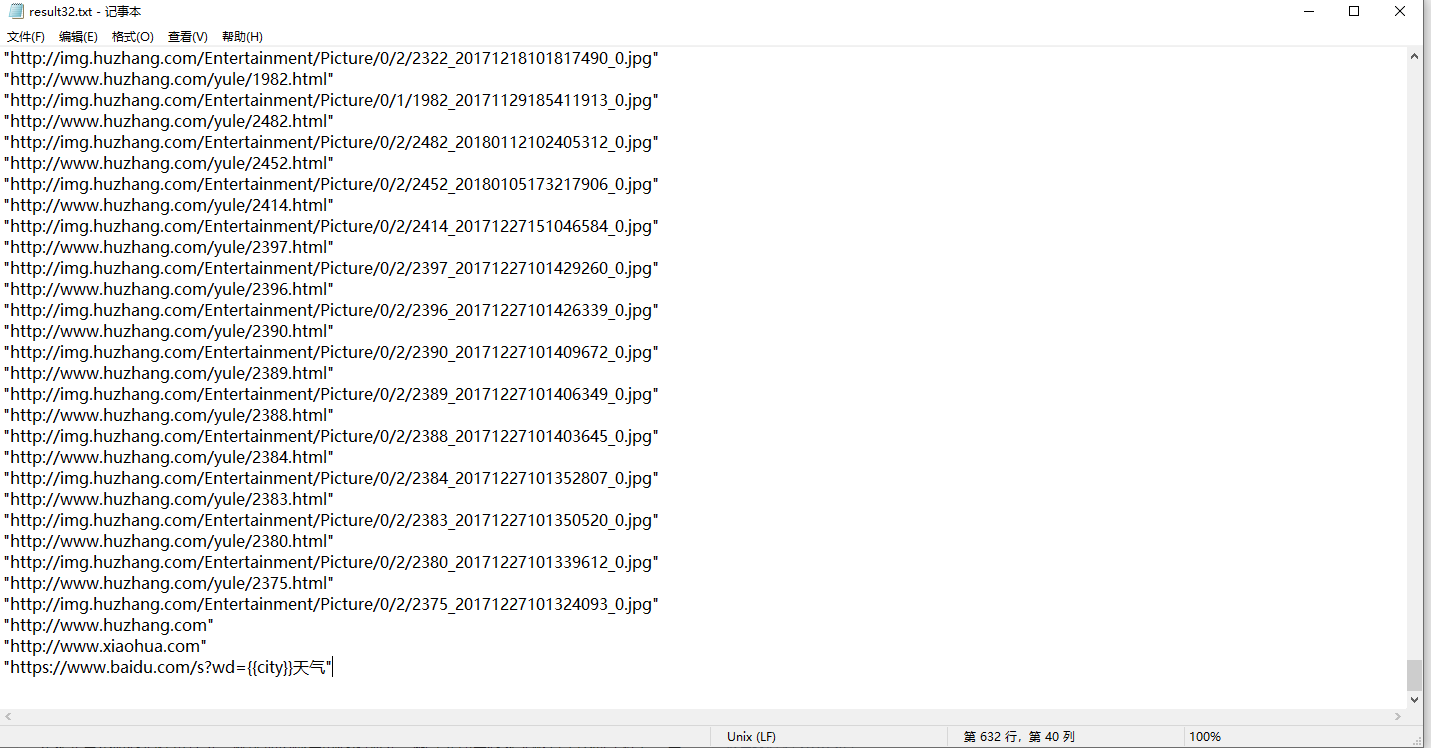
file1.close()

file2.close()

if \_\_name\_\_ == '\_\_main\_\_':

clearBlankLine()

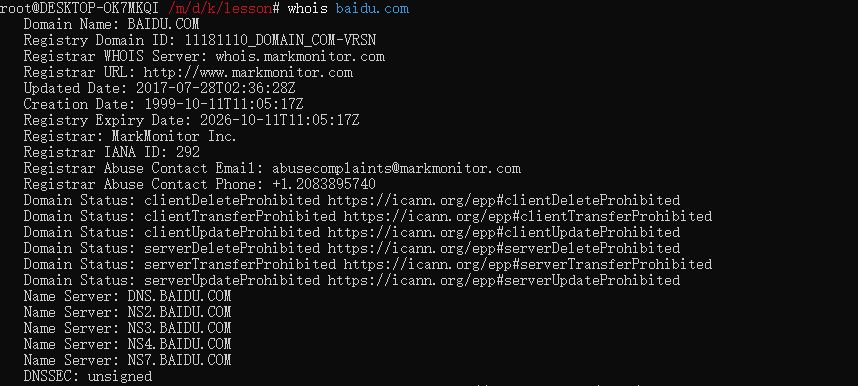
此时打开 result32.txt会得到700行网页地址

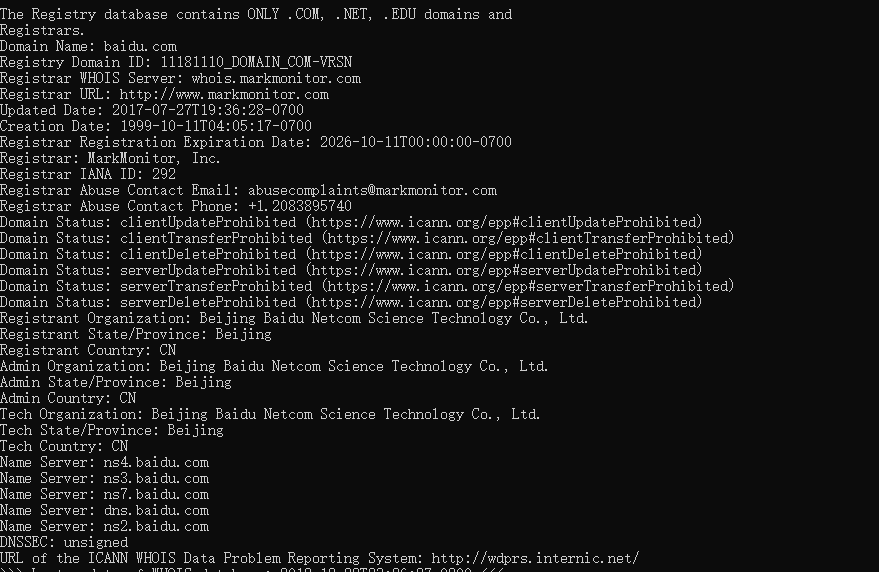
图2.6

从网址能够得到哪些数据？

ip地址段 DNS域名信息 邮件地址 所属公司信息 公开商业信息

子域名收集 指纹识别 等（其他的信息，例如端口扫描，旁站C段查询，安全漏洞信息等涉及到网络安全部分，这里不作展开）





以上内容同样适用于客户端软件app。对客户端进行网络抓包和分析，获得服务器ip后，也可以用以上的方法获取非常类似的数据。

## 数据处理并存储

有了以上这些数据，就可以得到更深一层次数据。这里给出一些例子：

对网站可信度的分析pinggu.zx110.org

对网络服务进行价值评估www.91guzhi.com

服务器安全评估

课程将对指定的网址进行whois处理，对处理结果进行汇总。

由于whois指定服务器并不相同，虽然.cn网址的服务器得到的结果和.com服务器得到的结果可以使用同样方式处理，但却不能采用同样的代码进行处理，所以这里只给出部分.com网址的处理代码，其他类型whois服务器处理同理。

首先，登入mysql数据库然后执行source /文件路径/database.sql 建立数据库

database.sql源代码如下：

-- 创建一个数据库net\_information  
-- 创建一个表network\_information  
-- network为主键且不可为空  
create database net\_information;  
  
use net\_information;  
  
create table network\_information  
(  
 network\_Name VARCHAR(100) not null,  
 network\_big\_Domain\_Name VARCHAR(100) null,  
 network\_little\_Domain\_Name VARCHAR(100) null,  
 network\_Registrant\_Organization VARCHAR(100) null,  
 `network\_Registrant\_State/Province` VARCHAR(100) null,  
 network\_Registrant\_Country varchar(100) null,  
 constraint network\_information\_pk  
 primary key (network\_Name)  
);

查询网页的whois信息，处理后写入数据库

file5.txt

qq.com  
jd.com  
baidu.com  
163.com  
amazon.com

源代码use.py内容

import os

import re

import sys

import mysql.connector

fp3=open("file5.txt","r")

for a in fp3.readlines():

os.system('whois %s > 1.txt'%a)

do(1.txt)

def do(argv):

mydb = mysql.connector.connect(

host="111.111.111.111", # 数据库主机ip,本机为localhost

user="root", # 用户名

passwd="\*\*\*\*\*\*\*", # 密码

database="net\_information" # 使用的数据库名

)

mycursor = mydb.cursor()

resultslist = search(argv[1])

sql = "INSERT INTO network\_information (network\_Name,network\_big\_Domain\_Name,network\_little\_Domain\_Name,network\_Registrant\_Organization,`network\_Registrant\_State/Province`,network\_Registrant\_Country) VALUES (%s,%s,%s,%s,%s,%s)"

val = (argv[1],resultslist[0],resultslist[1],resultslist[2],resultslist[3],resultslist[4])

mycursor.execute(sql, val)

mydb.commit() # 数据表内容有更新，必须使用到该语句

print(mycursor.rowcount, "记录插入成功。")

# 函数名: search

# 参数: 网络名&文件名

def search(netname):

file = open(netname).read()

findword1 = u"Domain Name: .\*"

pattern1 = re.compile(findword1)

findword2 = u"Registrant Organization: .\*"

pattern2 = re.compile(findword2)

findword3 = u"Registrant State/Province: .\*"

pattern3 = re.compile(findword3)

findword4 = u"Registrant Country: .\*"

pattern4 = re.compile(findword4)

results1 = re.findall(pattern1,file)

for index in range(len(results1)):

results1[index] = results1[index][13:]

results2 = re.findall(pattern2,file)

for index in range(len(results2)):

results2[index] = results2[index][25:]

results3 = re.findall(pattern3,file)

for index in range(len(results3)):

results3[index] = results3[index][27:]

results4 = re.findall(pattern4,file)

for index in range(len(results4)):

results4[index] = results4[index][20:]

resultslist = results1+results2+results3+results4

print(resultslist)

return resultslist

此时就可以在mysql数据库中查询到处理后的结果

# 三、数据挖掘

## 1.58同城二手车的信息爬取

爬取的是58同城的二手车中的4个属性（二手车品牌，二手车系列，二手车年份，二手车价格）。

用的是python的beautifulsoup框架和request库爬取的网页的源码里面的内容，把58同城的二手车的内容进行爬取，爬取思路是检索网页源码中指定lass的ul标签里面的li标签下面的h4和h5标签里面的文本内容，存储到Excel文档中。

案例如下（建议连接稳定网络，爬取过程可能较长，1-10分钟不等）：

import requests  
import xlwt  
from bs4 import BeautifulSoup as bs  
  
n = "1"  
a = 0  
b=1  
calc=0 #用于计数的变量  
workbook = xlwt.Workbook()  
sheet = workbook.add\_sheet("sheet1")  
sheet.write(0, 0, '二手车品牌')  
sheet.write(0, 1, '二手车系列')  
sheet.write(0, 2, '二手车年份')  
sheet.write(0, 3, '二手车价格')  
  
url = "https://bj.58.com/ershouche/pn" + n + "/?pane=true&PGTID=0d30001d-0000-1c2d-c765-e266ab6bfc82&ClickID=29" #此处的n是页码  
  
positionList = [] # 存放爬取的所有内容  
position = [] # 存放单次爬取的内容  
primary = [] # 这个是后面用的列表，存取positionList[]里面的多个字符串转化成的多个浮点型的数据  
for i in range(1, 250): # 这里是个循环，循环爬取250页的数据.  
 n = str(i) # 此处的i是循环次数，需要先转化成字符才可以  
 url = "https://bj.58.com/ershouche/pn" + n + "/?pane=true&PGTID=0d30001d-0000-1c2d-c765-e266ab6bfc82&ClickID=29" # 爬取的网页链接按字符串的形式存取到url里面去  
 response = requests.get(url)  
 soup = bs(response.text,"html.parser")

ul = soup.find(attrs={'class': 'car\_pane clearfix ac\_container'})  
 lis = ul.find\_all('li')  
 i=1  
 for li in lis:  
 list=(li.find('h5').text).replace(' ', ' ').replace('\n', '').replace('\r', '').strip().split()  
 list2=li.find('h4').text  
 sheet.write(b, 0, list[0])  
 sheet.write(b, 1, list[1])  
 sheet.write(b, 2, list[2])  
 sheet.write(b, 3, list2)  
 b+=1  
 i+=1  
 workbook.save("58\_old\_car.xls")  
print("10000条数据爬取完成，已经存入与该文件同级目录下的58\_old\_car.xls")

效果如下：

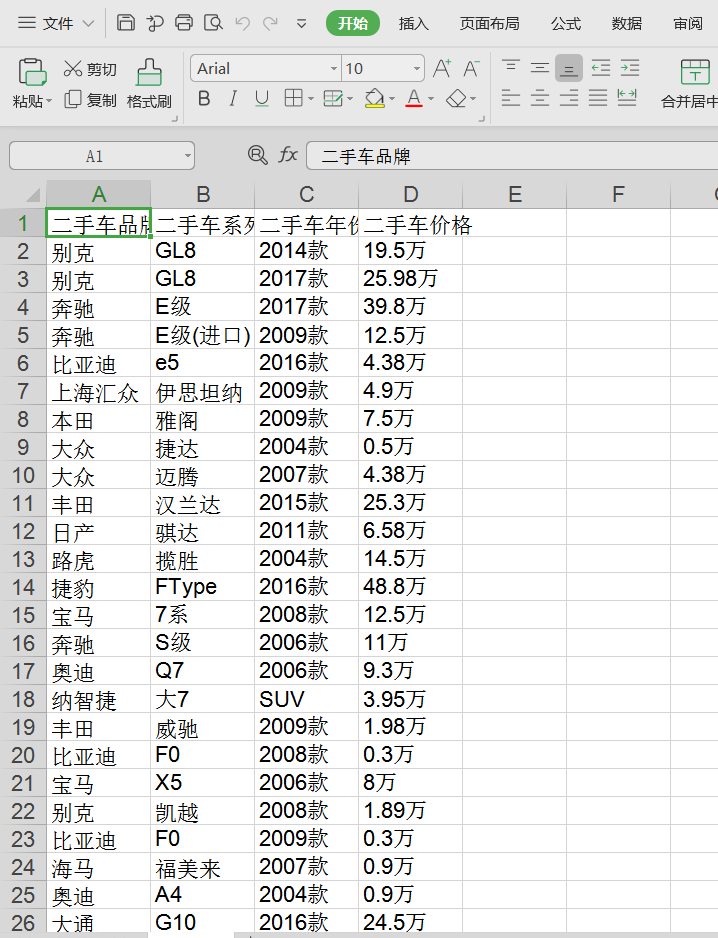


图3.1 运行结果

## 猫眼电影网《风语咒》的影评爬取

一.爬取思路

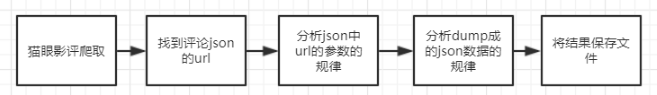


图3.2

图：爬取流程图

爬取过程

1.找到猫眼电影中风语咒影评得json数据:

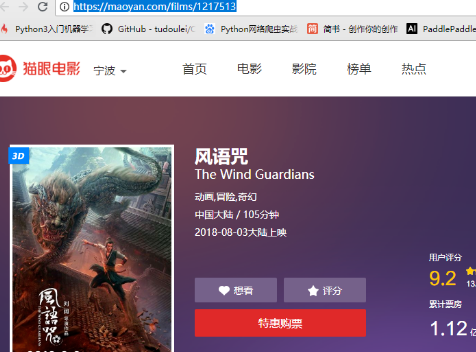


图3.3

找出url后，往下滚动后,发现其并无评论页面得接口，这时通过f12启动手机版得调试模式,通过手机端得界面寻找此电影的评论的接口位置。刷新后如下

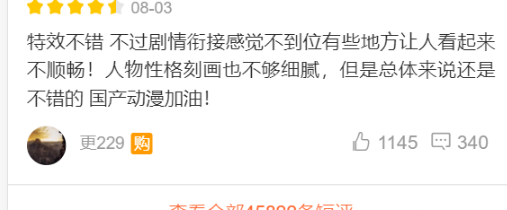


图3.4

找到位置后，点开评论区,因为其中的评论采用的是js的方式加载,故当其向下拉动时,通过Network可以获得其中的json串，通过json串得出其中的评论json的url。

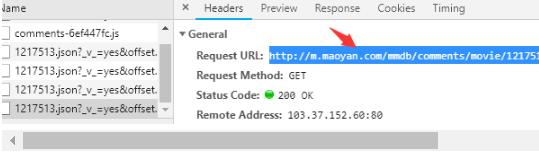


图3.5

通过分析其中的参数并在浏览器上尝试，可以发现其中offset和starttime的参数均为可变的,经实践得知,starttime参数的含义是该条评论的时间,而offset指的是该页评论的index；那么我们可以通过格式化这两个参数的值,来对影评评论数据进行抓取。

  因为暂时不知其中的offset到多少会为空,所以我们可以使用for循环对page进行操作,通过观察每页中的数据,发现每页评论的条数为15条，故在此每次遍历15条，即第一次的offset为0的话，下一次遍历的offset则为15,同样日期也可以用range的方法进行赋值：

二.load其中的json内容,从中筛选提取出自己需要的数据：

   通过将得到的json数据解析后,发现其中评论在cmt内,通过loads的方法将json转化为字典

采用对整体进行cmt采取遍历。

三.保存成excel格式的文件

   将文件保存,由于评论数据中不止有评论内容,还包含评论时间,评论作者,评论地点等一些内容,所以采用了pandas的方法来存储成excel格式，考虑到采集数据过程中可能会出现异常,故使用try,,except的格式来进行，当采集到的数据为空时,就表示采集完毕,保存到excel文件中:

代码案例：

import requests  
import time  
import random  
import json  
import pandas as pd  
  
  
movie = pd.DataFrame(columns=['city','content'])  
for i in range(0, 300):  
 j = random.randint(1,300)  
 print(str(i)+' '+str(j))  
 try:  
 time.sleep(2)  
 url= "http://m.maoyan.com/mmdb/comments/movie/1217513.json?\_v\_=yes&offset=" + str(j)  
 html = requests.get(url=url).content  
 data = json.loads(html.decode('utf-8'))['cmts']  
  
 for item in data:  
 movie = movie.append({'city':item['cityName'],  
 'content':item['content'],'date':item['startTime']},ignore\_index=True)  
 movie.to\_excel('风雨骤.xlsx',index=False)  
 except:  
 continue



# 四、分类与回归算法

## 算法概述

分类是数据挖掘、机器学习和模式识别中一个重要的研究领域。通过对当前数据挖掘中具有代表性的优秀分类算法进行分析和比较，总结出了各种算法的特性，为使用者选择算法或研究者改进算法提供了依据。

## 分类算法种类以及特征

分类模型的构造方法有决策树、统计方法、机器学习方法、神经网络方法等。按大的方向分类主要有：决策树，关联规则，贝叶斯，神经网络，规则学习，k－临近法，遗传算法，粗糙集以及模糊逻辑技术。

### 2.1 决策树（decision tree）分类算法

决策树是以实例为基础的归纳学习算法。它从一组无次序、无规则的元组中推理出决策树表示形式的分类规则。它采用自顶向下的递归方式，在决策树的内部结点进行属性值的比较，并根据不同的属性值从该结点向下分支，叶结点是要学习划分的类。从根到叶结点的一条路径就对应着一条合取规则，整个决策树就对应着一组析取表达式规则。1986年Quinlan提出了著名的ID3算法。在ID3算法的基础上，1993年Quinlan又提出了C4.5算法。为了适应处理大规模数据集的需要，后来又提出了若干改进的算法，其中SLIQ (super-vised learning in quest)和SPRINT (scalable parallelizableinduc[TI](http://bbs.elecfans.com/zhuti_715_1.html)on of decision trees)是比较有代表性的两个算法。

(1) ID3算法

ID3算法的核心是：在决策树各级结点上选择属性时，用信息增益（informa[TI](http://bbs.elecfans.com/zhuti_715_1.html)on gain）作为属性的选择标准，以使得在每一个非叶结点进行测试时，能获得关于被测试记录最大的类别信息。其具体方法是：检测所有的属性，选择信息增益最大的属性产生决策树结点，由该属性的不同取值建立分支，再对各分支的子集递归调用该方法建立决策树结点的分支，直到所有子集仅包含同一类别的数据为止。最后得到一棵决策树，它可以用来对新的样本进行分类。

某属性的信息增益按下列方法计算。通过计算每个属性的信息增益，并比较它们的大小，就不难获得具有最大信息增益的属性。

设S是s个数据样本的集合。假定类标号属性具有m个不同值，定义m个不同类Ci(i=1,…,m)。设si是类Ci中的样本数。对一个给定的样本分类所需的期望信息由下式给出：

其中pi=si/s是任意样本属于Ci的概率。注意，对数函数以2为底，其原因是信息用二进制编码。

设属性A具有v个不同值{a1,a2,……,av}。可以用属性A将S划分为v个子集{S1,S2,……,Sv}，其中Sj中的样本在属性A上具有相同的值aj（j=1,2,……,v）。设sij是子集Sj中类Ci的样本数。由A划分成子集的熵或信息期望由下式给出：

熵值越小，子集划分的纯度越高。对于给定的子集Sj，其信息期望为

其中pij=sij/sj 是Sj中样本属于Ci的概率。在属性A上分枝将获得的信息增益是

Gain(A)= I(s1, s2, …,sm)-E(A)

ID3算法的优点是：算法的理论清晰，方法简单，学习能力较强。其缺点是：只对比较小的数据集有效，且对噪声比较敏感，当训练数据集加大时，决策树可能会随之改变。

(2) C4.5算法

C4.5算法继承了ID3算法的优点，并在以下几方面对ID3算法进行了改进：

1) 用信息增益率来选择属性，克服了用信息增益选择属性时偏向选择取值多的属性的不足；

2) 在树构造过程中进行剪枝；

3) 能够完成对连续属性的离散化处理；

4) 能够对不完整数据进行处理。

C4.5算法与其它分类算法如统计方法、神经网络等比较起来有如下优点：产生的分类规则易于理解，准确率较高。其缺点是：在构造树的过程中，需要对数据集进行多次的顺序扫描和排序，因而导致算法的低效。此外，C4.5只适合于能够驻留于内存的数据集，当训练集大得无法在内存容纳时程序无法运行。

(3) SLIQ算法

SLIQ算法对C4.5决策树分类算法的实现方法进行了改进，在决策树的构造过程中采用了“预排序”和“广度优先策略”两种技术。

1) 预排序。对于连续属性在每个内部结点寻找其最优分裂标准时，都需要对训练集按照该属性的取值进行排序，而排序是很浪费时间的操作。为此，SLIQ算法采用了预排序技术。所谓预排序，就是针对每个属性的取值，把所有的记录按照从小到大的顺序进行排序，以消除在决策树的每个结点对数据集进行的排序。具体实现时，需要为训练数据集的每个属性创建一个属性列表，为类别属性创建一个类别列表。

2) 广度优先策略。在C4.5算法中，树的构造是按照深度优先策略完成的，需要对每个属性列表在每个结点处都进行一遍扫描，费时很多，为此，SLIQ采用广度优先策略构造决策树，即在决策树的每一层只需对每个属性列表扫描一次，就可以为当前决策树中每个叶子结点找到最优分裂标准。

SLIQ算法由于采用了上述两种技术，使得该算法能够处理比C4.5大得多的训练集，在一定范围内具有良好的随记录个数和属性个数增长的可伸缩性。

然而它仍然存在如下缺点：

1)由于需要将类别列表存放于内存,而类别列表的元组数与训练集的元组数是相同的,这就一定程度上限制了可以处理的数据集的大小。

2) 由于采用了预排序技术，而排序算法的复杂度本身并不是与记录个数成线性关系，因此，使得SLIQ算法不可能达到随记录数目增长的线性可伸缩性。

(4) SPRINT算法

为了减少驻留于内存的数据量，SPRINT算法进一步改进了决策树算法的数据结构，去掉了在SLIQ中需要驻留于内存的类别列表，将它的类别列合并到每个属性列表中。这样，在遍历每个属性列表寻找当前结点的最优分裂标准时，不必参照其他信息，将对结点的分裂表现在对属性列表的分裂，即将每个属性列表分成两个，分别存放属于各个结点的记录。

SPRINT算法的优点是在寻找每个结点的最优分裂标准时变得更简单。其缺点是对非分裂属性的属性列表进行分裂变得很困难。解决的办法是对分裂属性进行分裂时用哈希表记录下每个记录属于哪个孩子结点，若内存能够容纳下整个哈希表，其他属性列表的分裂只需参照该哈希表即可。由于哈希表的大小与训练集的大小成正比，当训练集很大时，哈希表可能无法在内存容纳，此时分裂只能分批执行，这使得SPRINT算法的可伸缩性仍然不是很好。

贝叶斯分类是统计学分类方法，它是一类利用概率统计知识进行分类的算法。在许多场合，朴素贝叶斯(Na?ve Bayes，NB)分类算法可以与决策树和神经网络分类算法相媲美，该算法能运用到大型数据库中，且方法简单、分类准确率高、速度快。由于贝叶斯定理假设一个属性值对给定类的影响独立于其它属性的值，而此假设在实际情况中经常是不成立的，因此其分类准确率可能会下降。为此，就出现了许多降低独立性假设的贝叶斯分类算法，如TAN(tree augmented Bayes network)算法。  
3.ID3决策树分类算法的实现

### 3.1案例背景

决策数(Decision Tree)在机器学习中也是比较常见的一种算法，属于监督学习中的一种。看字面意思应该也比较容易理解，相比其他算法比如支持向量机(SVM)或神经网络，似乎决策树感觉“亲切”许多。

①优点：计算复杂度不高，输出结果易于理解，对中间值的缺失值不敏感，可以处理不相关特征数据。

②缺点：可能会产生过度匹配的问题。

③使用数据类型：数值型和标称型。

简单介绍完毕，让我们来通过一个例子让决策树“原形毕露”。

老师问了个问题，只根据头发和声音怎么判断一位同学的性别。   
为了解决这个问题，同学们马上简单的统计了7位同学的相关特征，数据如下：



图4.1

机智的同学A想了想，先根据头发判断，若判断不出，再根据声音判断，于是画了一幅图，如下：

于是，一个简单、直观的决策树就这么出来了。头发长、声音粗就是男生；头发长、声音细就是女生；头发短、声音粗是男生；头发短、声音细是女生。   
原来机器学习中决策树就这玩意，这也太简单了吧。。。   
这时又蹦出个同学B，想先根据声音判断，然后再根据头发来判断，如是大手一挥也画了个决策树：

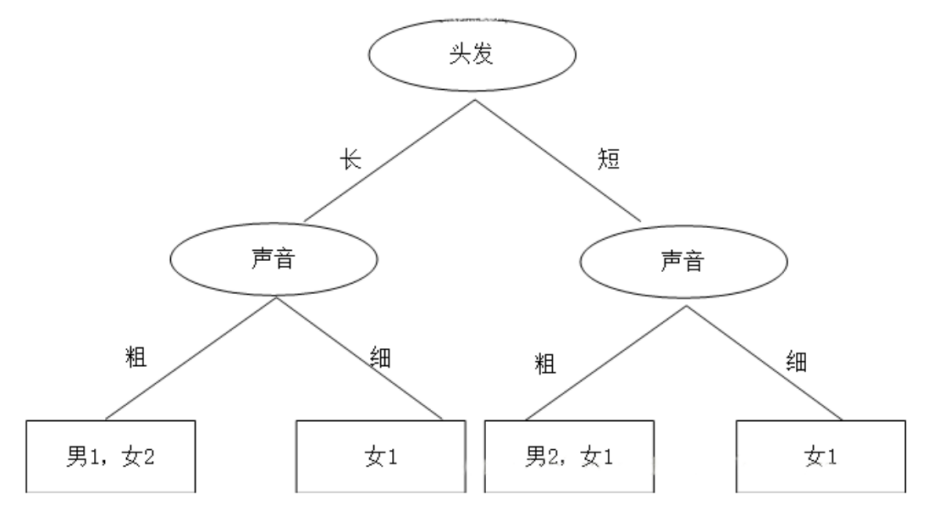


图4.2

于是，一个简单、直观的决策树就这么出来了。头发长、声音粗就是男生；头发长、声音细就是女生；头发短、声音粗是男生；头发短、声音细是女生。   
原来机器学习中决策树就这玩意，这也太简单了吧。。。   
这时又蹦出个同学B，想先根据声音判断，然后再根据头发来判断，如是大手一挥也画了个决策树：

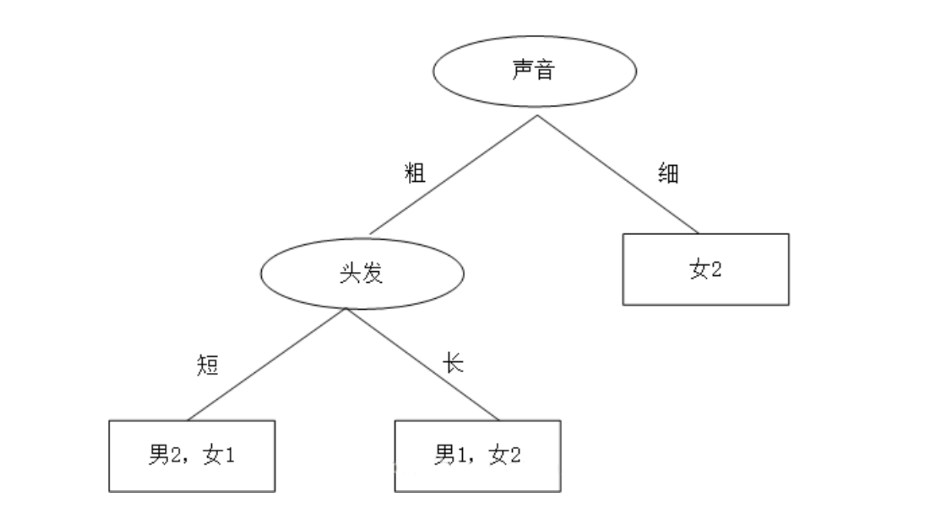


图4.3

同学B的决策树：首先判断声音，声音细，就是女生；声音粗、头发长是男生；声音粗、头发长是女生。

那么问题来了：同学A和同学B谁的决策树好些？计算机做决策树的时候，面对多个特征，该如何选哪个特征为最佳的划分特征？

划分数据集的大原则是：将无序的数据变得更加有序。   
我们可以使用多种方法划分数据集，但是每种方法都有各自的优缺点。于是我们这么想，如果我们能测量数据的复杂度，对比按不同特征分类后的数据复杂度，若按某一特征分类后复杂度减少的更多，那么这个特征即为最佳分类特征。   
Claude Shannon 定义了熵（entropy）和信息增益(information gain)。   
用熵来表示信息的复杂度，熵越大，则信息越复杂。公式如下：

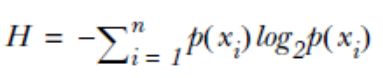


图4.4

息增益(information gain)，表示两个信息熵的差值。   
首先计算未分类前的熵，总共有8位同学，男生3位，女生5位。   
熵（总）=-3/8\*log2(3/8)-5/8\*log2(5/8)=0.9544   
接着分别计算同学A和同学B分类后信息熵。   
同学A首先按头发分类，分类后的结果为：长头发中有1男3女。短头发中有2男2女。   
熵（同学A长发）=-1/4\*log2(1/4)-3/4\*log2(3/4)=0.8113   
熵（同学A短发）=-2/4\*log2(2/4)-2/4\*log2(2/4)=1   
熵（同学A）=4/8\*0.8113+4/8\*1=0.9057   
信息增益（同学A）=熵（总）-熵（同学A）=0.9544-0.9057=0.0487   
同理，按同学B的方法，首先按声音特征来分，分类后的结果为：声音粗中有3男3女。声音细中有0男2女。   
熵（同学B声音粗）=-3/6\*log2(3/6)-3/6\*log2(3/6)=1   
熵（同学B声音粗）=-2/2\*log2(2/2)=0   
熵（同学B）=6/8\*1+2/8\*0=0.75   
信息增益（同学B）=熵（总）-熵（同学A）=0.9544-0.75=0.2087

按同学B的方法，先按声音特征分类，信息增益更大，区分样本的能力更强，更具有代表性。   
以上就是决策树ID3算法的核心思想。   
3.2实现案例

from math import log  
import operator  
  
def calcShannonEnt(dataSet): # 计算数据的熵(entropy)  
 numEntries=len(dataSet) # 数据条数  
 labelCounts={}  
 for featVec in dataSet:  
 currentLabel=featVec[-1] # 每行数据的最后一个字（类别）  
 if currentLabel not in labelCounts.keys():  
 labelCounts[currentLabel]=0  
 labelCounts[currentLabel]+=1 # 统计有多少个类以及每个类的数量  
 shannonEnt=0  
 for key in labelCounts:  
 prob=float(labelCounts[key])/numEntries # 计算单个类的熵值  
 shannonEnt-=prob\*log(prob,2) # 累加每个类的熵值  
 return shannonEnt  
  
def createDataSet1(): # 创造示例数据  
 dataSet = [['长', '粗', '男'],  
 ['短', '粗', '男'],  
 ['短', '粗', '男'],  
 ['长', '细', '女'],  
 ['短', '细', '女'],  
 ['短', '粗', '女'],  
 ['长', '粗', '女'],  
 ['长', '粗', '女']]  
 labels = ['头发','声音'] #两个特征  
 return dataSet,labels  
  
def splitDataSet(dataSet,axis,value): # 按某个特征分类后的数据  
 retDataSet=[]  
 for featVec in dataSet:  
 if featVec[axis]==value:  
 reducedFeatVec =featVec[:axis]  
 reducedFeatVec.extend(featVec[axis+1:])  
 retDataSet.append(reducedFeatVec)  
 return retDataSet  
  
def chooseBestFeatureToSplit(dataSet): # 选择最优的分类特征  
 numFeatures = len(dataSet[0])-1  
 baseEntropy = calcShannonEnt(dataSet) # 原始的熵  
 bestInfoGain = 0  
 bestFeature = -1  
 for i in range(numFeatures):  
 featList = [example[i] for example in dataSet]  
 uniqueVals = set(featList)  
 newEntropy = 0  
 for value in uniqueVals:  
 subDataSet = splitDataSet(dataSet,i,value)  
 prob =len(subDataSet)/float(len(dataSet))  
 newEntropy +=prob\*calcShannonEnt(subDataSet) # 按特征分类后的熵  
 infoGain = baseEntropy - newEntropy # 原始熵与按特征分类后的熵的差值  
 if (infoGain>bestInfoGain): # 若按某特征划分后，熵值减少的最大，则次特征为最优分类特征  
 bestInfoGain=infoGain  
 bestFeature = i  
 return bestFeature  
  
def majorityCnt(classList): #按分类后类别数量排序，比如：最后分类为2男1女，则判定为男；  
 classCount={}  
 for vote in classList:  
 if vote not in classCount.keys():  
 classCount[vote]=0  
 classCount[vote]+=1  
 sortedClassCount = sorted(classCount.items(),key=operator.itemgetter(1),reverse=True)  
 return sortedClassCount[0][0]  
  
def createTree(dataSet,labels):  
 classList=[example[-1] for example in dataSet] # 类别：男或女  
 if classList.count(classList[0])==len(classList):  
 return classList[0]  
 if len(dataSet[0])==1:  
 return majorityCnt(classList)  
 bestFeat=chooseBestFeatureToSplit(dataSet) #选择最优特征  
 bestFeatLabel=labels[bestFeat]  
 myTree={bestFeatLabel:{}} #分类结果以字典形式保存  
 del(labels[bestFeat])  
 featValues=[example[bestFeat] for example in dataSet]  
 uniqueVals=set(featValues)  
 for value in uniqueVals:  
 subLabels=labels[:]  
 myTree[bestFeatLabel][value]=createTree(splitDataSet\  
 (dataSet,bestFeat,value),subLabels)  
 return myTree  
  
  
if \_\_name\_\_=='\_\_main\_\_':  
 dataSet, labels=createDataSet1() # 创造示列数据  
 print(createTree(dataSet, labels)) # 输出决策树模型结果

运行结果：

{'声音': {'细': '女', '粗': {'头发': {'长': '女', '短': '男'}}}}

这个结果的意思是：首先按声音分类，声音细为女生；然后再按头发分类：声音粗，头发短为男生；声音粗，头发长为女生。   
这个结果也正是同学B的结果。   
补充说明：判定分类结束的依据是，若按某特征分类后出现了最终类（男或女），则判定分类结束。使用这种方法，在数据比较大，特征比较多的情况下，很容易造成过拟合，于是需进行决策树枝剪，一般枝剪方法是当按某一特征分类后的熵小于设定值时，停止分类。

ID3算法存在的缺点：   
1. ID3算法在选择根节点和内部节点中的分支属性时，采用信息增益作为评价标准。信息增益的缺点是倾向于选择取值较多是属性，在有些情况下这类属性可能不会提供太多有价值的信息。   
2. ID3算法只能对描述属性为离散型属性的数据集构造决策树 。

4.贝叶斯分类算法

4.1算法概述

贝叶斯分类算法是一大类分类算法的总称

贝叶斯分类算法以样本可能属于某类的概率来作为分类依据

朴素贝叶斯分类算法是贝叶斯分类算法中最简单的一种

注：朴素的意思是条件概率独立性

P(A|x1x2x3x4)=p(A|x1)\*p(A|x2)p(A|x3)p(A|x4)则为条件概率独立

P(xy|z)=p(xyz)/p(z)=p(xz)/p(z)\*p(yz)/p(z)

4.2算法思想

朴素贝叶斯的思想是这样的：

如果一个事物在一些属性条件发生的情况下，事物属于A的概率>属于B的概率，则判定事物属于A

通俗来说比如，你在街上看到一个黑人，我让你猜这哥们哪里来的，你十有八九猜非洲。为什么呢？

在你的脑海中，有这么一个判断流程：

1、这个人的肤色是黑色 <特征>

2、黑色人种是非洲人的概率最高 <条件概率：黑色条件下是非洲人的概率>

3、没有其他辅助信息的情况下，最好的判断就是非洲人

这就是朴素贝叶斯的思想基础。

再扩展一下，假如在街上看到一个黑人讲英语，那我们是怎么去判断他来自于哪里？

提取特征：

肤色： 黑

语言： 英语

黑色人种来自非洲的概率： 80%

黑色人种来自于美国的概率：20%

讲英语的人来自于非洲的概率：10%

讲英语的人来自于美国的概率：90%

在我们的自然思维方式中，就会这样判断：

这个人来自非洲的概率：80% \* 10% = 0.08

这个人来自美国的概率：20% \* 90% =0.18

我们的判断结果就是：此人来自美国！

其蕴含的数学原理如下：

p(A|xy)=p(Axy)/p(xy)=p(Axy)/p(x)p(y)=p(A)/p(x)\*p(A)/p(y)\* p(xy)/p(xy)=p(A|x)p(A|y)

P(类别 | 特征)=P(特征 | 类别)\*P(类别) / P(特征)

4.3实现案例

class NBClassify(object):  
 def \_\_init\_\_(self, fillNa=1):  
 self.fillNa = 1  
 pass  
  
 def train(self, trainSet):  
 # 计算每种类别的概率  
 # 保存所有tag的所有种类，及它们出现的频次  
 dictTag = {}  
 for subTuple in trainSet:  
 dictTag[str(subTuple[1])] = 1 if str(subTuple[1]) not in dictTag.keys() else dictTag[str(subTuple[1])] + 1  
 # 保存每个tag本身的概率  
 tagProbablity = {}  
 totalFreq = sum([value for value in dictTag.values()])  
 for key, value in dictTag.items():  
 tagProbablity[key] = value / totalFreq  
 # print(tagProbablity)  
 self.tagProbablity = tagProbablity  
 ##############################################################################  
 # 计算特征的条件概率  
 # 保存特征属性基本信息{特征1:{值1:出现5次, 值2:出现1次}, 特征2:{值1:出现1次, 值2:出现5次}}  
 dictFeaturesBase = {}  
 for subTuple in trainSet:  
 for key, value in subTuple[0].items():  
 if key not in dictFeaturesBase.keys():  
 dictFeaturesBase[key] = {value: 1}  
 else:  
 if value not in dictFeaturesBase[key].keys():  
 dictFeaturesBase[key][value] = 1  
 else:  
 dictFeaturesBase[key][value] += 1  
 # dictFeaturesBase = {  
 # '职业': {'农夫': 1, '教师': 2, '建筑工人': 2, '护士': 1},  
 # '症状': {'打喷嚏': 3, '头痛': 3}  
 # }  
 dictFeatures = {}.fromkeys([key for key in dictTag])  
 for key in dictFeatures.keys():  
 dictFeatures[key] = {}.fromkeys([key for key in dictFeaturesBase])  
 for key, value in dictFeatures.items():  
 for subkey in value.keys():  
 value[subkey] = {}.fromkeys([x for x in dictFeaturesBase[subkey].keys()])  
 # dictFeatures = {  
 # '感冒 ': {'症状': {'打喷嚏': None, '头痛': None}, '职业': {'护士': None, '农夫': None, '建筑工人': None, '教师': None}},  
 # '脑震荡': {'症状': {'打喷嚏': None, '头痛': None}, '职业': {'护士': None, '农夫': None, '建筑工人': None, '教师': None}},  
 # '过敏 ': {'症状': {'打喷嚏': None, '头痛': None}, '职业': {'护士': None, '农夫': None, '建筑工人': None, '教师': None}}  
 # }  
 # initialise dictFeatures  
 for subTuple in trainSet:  
 for key, value in subTuple[0].items():  
 dictFeatures[subTuple[1]][key][value] = 1 if dictFeatures[subTuple[1]][key][value] == None else \  
 dictFeatures[subTuple[1]][key][value] + 1  
 # print(dictFeatures)  
 # 将驯良样本中没有的项目，由None改为一个非常小的数值，表示其概率极小而并非是零  
 for tag, featuresDict in dictFeatures.items():  
 for featureName, fetureValueDict in featuresDict.items():  
 for featureKey, featureValues in fetureValueDict.items():  
 if featureValues == None:  
 fetureValueDict[featureKey] = 1  
 # 由特征频率计算特征的条件概率P(feature|tag)  
 for tag, featuresDict in dictFeatures.items():  
 for featureName, fetureValueDict in featuresDict.items():  
 totalCount = sum([x for x in fetureValueDict.values() if x != None])  
 for featureKey, featureValues in fetureValueDict.items():  
 fetureValueDict[featureKey] = featureValues / totalCount if featureValues != None else None  
 self.featuresProbablity = dictFeatures  
 ##############################################################################  
  
 def classify(self, featureDict):  
 resultDict = {}  
 # 计算每个tag的条件概率  
 for key, value in self.tagProbablity.items():  
 iNumList = []  
 for f, v in featureDict.items():  
 if self.featuresProbablity[key][f][v]:  
 iNumList.append(self.featuresProbablity[key][f][v])  
 conditionPr = 1  
 for iNum in iNumList:  
 conditionPr \*= iNum  
 resultDict[key] = value \* conditionPr  
 # 对比每个tag的条件概率的大小  
 resultList = sorted(resultDict.items(), key=lambda x: x[1], reverse=True)  
 return resultList[0][0]  
  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 trainSet = [  
 ({"症状": "打喷嚏", "职业": "护士"}, "感冒 "),  
 ({"症状": "打喷嚏", "职业": "农夫"}, "过敏 "),  
 ({"症状": "头痛", "职业": "建筑工人"}, "脑震荡"),  
 ({"症状": "打喷嚏", "职业": "建筑工人"}, "有人在背后指指点点"),  
 ({"症状": "打喷嚏", "职业": "建筑工人"}, "有人在背后指指点点"),  
 ({"症状": "打喷嚏", "职业": "建筑工人"}, "有人在背后指指点点"),  
 ({"症状": "头痛", "职业": "建筑工人"}, "感冒 "),  
 ({"症状": "打喷嚏", "职业": "教师"}, "感冒 "),  
 ({"症状": "头痛", "职业": "教师"}, "脑震荡"),  
  
 ]  
 monitor = NBClassify()  
 # trainSet is something like that [(featureDict, tag), ]  
 monitor.train(trainSet)  
 # 打喷嚏的建筑工人  
 # 请问他患上感冒的概率有多大？  
 result = monitor.classify({"症状": "打喷嚏", "职业": "建筑工人"})  
 print(result)

测试：

{"症状": "打喷嚏", "职业": "建筑工人"}

运行结果：

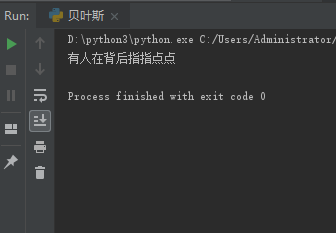


图4.5

# 五、聚类算法

什么是聚类算法？聚类是一种机器学习技术，它涉及到数据点的分组。给定一组数据点，我们可以使用聚类算法将每个数据点划分为一个特定的组。理论上，同一组中的数据点应该具有相似的属性和/或特征，而不同组中的数据点应该具有高度不同的属性和/或特征。聚类是一种无监督学习的方法，是许多领域中常用的统计数据分析技术。

在数据科学中，我们可以使用聚类分析从我们的数据中获得一些有价值的见解。

聚类算法有哪些呢？在这篇文章中，我们将研究5种流行的聚类算法以及它们的优缺点。

## K-MEANS聚类算法

K-Means聚类算法可能是大家最熟悉的聚类算法。它出现在很多介绍性的数据科学和机器学习课程中。在代码中很容易理解和实现！

图片参考网页：

<http://imgcdn.atyun.com/2018/03/1-KrcZK0xYgTa4qFrVr0fO2w.gif>

1.首先，我们选择一些类/组来使用并随机地初始化它们各自的中心点。要想知道要使用的类的数量，最好快速地查看一下数据，并尝试识别任何不同的分组。中心点是与每个数据点向量相同长度的向量，在上面的图形中是“X”。

2.每个数据点通过计算点和每个组中心之间的距离进行分类，然后将这个点分类为最接近它的组。

3.基于这些分类点，我们通过取组中所有向量的均值来重新计算组中心。

4.对一组迭代重复这些步骤。你还可以选择随机初始化组中心几次，然后选择那些看起来对它提供了最好结果的来运行。

K-Means聚类算法的优势在于它的速度非常快，因为我们所做的只是计算点和群中心之间的距离;它有一个线性复杂度O(n)。

另一方面，K-Means也有几个缺点。首先，你必须选择有多少组/类。这并不是不重要的事，理想情况下，我们希望它能帮我们解决这些问题，因为它的关键在于从数据中获得一些启示。K-Means也从随机选择的聚类中心开始，因此在不同的算法运行中可能产生不同的聚类结果。因此，结果可能是不可重复的，并且缺乏一致性。其他聚类方法更加一致。

K-Medians是另一种与K-Means有关的聚类算法，除了使用均值的中间值来重新计算组中心点以外，这种方法对离群值的敏感度较低（因为使用中值），但对于较大的数据集来说，它要慢得多，因为在计算中值向量时，每次迭代都需要进行排序。

## 2.均值偏移聚类算法

均值偏移（Mean shift）聚类算法是一种基于滑动窗口（sliding-window）的算法，它试图找到密集的数据点。而且，它还是一种基于中心的算法，它的目标是定位每一组群/类的中心点，通过更新中心点的候选点来实现滑动窗口中的点的平均值。这些候选窗口在后期处理阶段被过滤，以消除几乎重复的部分，形成最后一组中心点及其对应的组。

图片参考网页：

<http://imgcdn.atyun.com/2018/03/1-bkFlVrrm4HACGfUzeBnErw.gif>

<http://imgcdn.atyun.com/2018/03/1-vyz94J_76dsVToaa4VG1Zg.gif>

1.为了解释这一变化，我们将考虑二维空间中的一组点（就像上面的例子）。我们从一个以点C（随机选择）为中心的圆形滑窗开始，以半径r为内核。均值偏移是一种爬山算法（hill climbing algorithm），它需要在每个步骤中反复地将这个内核移动到一个更高的密度区域，直到收敛。

2.在每一次迭代中，滑动窗口会移向密度较高的区域，将中心点移动到窗口内的点的平均值（因此得名）。滑动窗口中的密度与它内部的点的数量成比例。自然地，通过移向窗口中点的平均值，它将逐渐向更高的点密度方向移动。

3.我们继续根据均值移动滑动窗口，直到没有方向移动可以容纳内核中的更多点。看看上面的图表;我们一直在移动这个圆，直到我们不再增加密度（也就是窗口中的点数）。

4.步骤1到3的过程是用许多滑动窗口完成的，直到所有的点都位于一个窗口内。当多个滑动窗口重叠的时候，包含最多点的窗口会被保留。然后，数据点根据它们所在的滑动窗口聚类。

下面展示了从端到端所有滑动窗口的整个过程的演示。每个黑点代表一个滑动窗口的质心，每个灰色点都是一个数据点。

## 3.DBSCAN聚类算法

DBSCAN(Density-Based Spatial Clustering of Applications with Noise)是一个比较有代表性的基于密度的聚类算法，类似于均值转移聚类算法，但它有几个显著的优点。

图片参考网页：

<http://imgcdn.atyun.com/2018/03/1-tc8UF-h0nQqUfLC8-0uInQ.gif>

1.DBSCAN以一个从未访问过的任意起始数据点开始。这个点的邻域是用距离ε（所有在ε距离的点都是邻点）来提取的。

2.如果在这个邻域中有足够数量的点（根据 minPoints），那么聚类过程就开始了，并且当前的数据点成为新聚类中的第一个点。否则，该点将被标记为噪声（稍后这个噪声点可能会成为聚类的一部分）。在这两种情况下，这一点都被标记为“访问（visited）”。

3.对于新聚类中的第一个点，其ε距离附近的点也会成为同一聚类的一部分。这一过程使在ε邻近的所有点都属于同一个聚类，然后重复所有刚刚添加到聚类组的新点。

4.步骤2和步骤3的过程将重复，直到聚类中的所有点都被确定，就是说在聚类附近的所有点都已被访问和标记。

5.一旦我们完成了当前的聚类，就会检索并处理一个新的未访问点，这将导致进一步的聚类或噪声的发现。这个过程不断地重复，直到所有的点被标记为访问。因为在所有的点都被访问过之后，每一个点都被标记为属于一个聚类或者是噪音。

DBSCAN比其他聚类算法有一些优势。首先，它不需要一个预设定的聚类数量。它还将异常值识别为噪声，而不像均值偏移聚类算法，即使数据点非常不同，它也会将它们放入一个聚类中。此外，它还能很好地找到任意大小和任意形状的聚类。

DBSCAN的主要缺点是，当聚类具有不同的密度时，它的性能不像其他聚类算法那样好。这是因为当密度变化时，距离阈值ε和识别邻近点的minPoints的设置会随着聚类的不同而变化。这种缺点也会出现在非常高维的数据中，因为距离阈值ε变得难以估计。

## 4.使用高斯混合模型（GMM）的期望最大化（EM）聚类

K-Means的一个主要缺点是它对聚类中心的平均值的使用很简单幼稚。我们可以通过图片来了解为什么这不是最好的方法。在左边看起来很明显的是，有两个圆形的聚类，不同的半径以相同的平均值为中心。K-Means无法处理，因为聚类的均值非常接近。在聚类不是循环的情况下，K-Means也会失败，这也是使用均值作为聚类中心的结果。

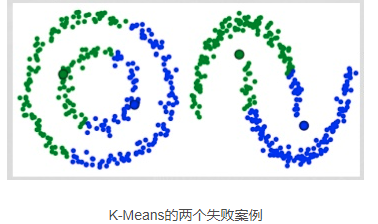


图5.1

高斯混合模型（GMMs）比K-Means更具灵活性。使用高斯混合模型，我们可以假设数据点是高斯分布的;比起说它们是循环的，这是一个不那么严格的假设。这样，我们就有两个参数来描述聚类的形状：平均值和标准差！以二维的例子为例，这意味着聚类可以采用任何形式的椭圆形状（因为在x和y方向上都有标准差）。因此，每个高斯分布可归属于一个单独的聚类。

为了找到每个聚类的高斯分布的参数（例如平均值和标准差）我们将使用一种叫做期望最大化（EM）的优化算法。看看下面的图表，就可以看到高斯混合模型是被拟合到聚类上的。然后，我们可以继续进行期望的过程——使用高斯混合模型实现最大化聚类。

图片参考网页：

<http://imgcdn.atyun.com/2018/03/1-OyXgise21a23D5JCss8Tlg.gif>

1.我们首先选择聚类的数量（如K-Means所做的那样），然后随机初始化每个聚类的高斯分布参数。通过快速查看数据，可以尝试为初始参数提供良好的猜测。注意，在上面的图表中可以看到，这并不是100%的必要，因为高斯开始时的表现非常不好，但是很快就被优化了。

2.给定每个聚类的高斯分布，计算每个数据点属于特定聚类的概率。一个点离高斯中心越近，它就越有可能属于那个聚类。这应该是很直观的，因为有一个高斯分布，我们假设大部分的数据都离聚类中心很近。

3.基于这些概率，我们为高斯分布计算一组新的参数，这样我们就能最大程度地利用聚类中的数据点的概率。我们使用数据点位置的加权和来计算这些新参数，权重是属于该特定聚类的数据点的概率。为了解释这一点，我们可以看一下上面的图，特别是黄色的聚类作为例子。分布在第一次迭代中是随机的，但是我们可以看到大多数的黄色点都在这个分布的右边。当我们计算一个由概率加权的和，即使在中心附近有一些点，它们中的大部分都在右边。因此，自然分布的均值更接近于这些点。我们还可以看到，大多数点都是“从右上角到左下角”。因此，标准差的变化是为了创造一个更符合这些点的椭圆，从而使概率的总和最大化。

步骤2和3被迭代地重复，直到收敛，在那里，分布不会从迭代到迭代这个过程中变化很多。

使用高斯混合模型有两个关键的优势。首先，高斯混合模型在聚类协方差方面比K-Means要灵活得多;根据标准差参数，聚类可以采用任何椭圆形状，而不是局限于圆形。K-Means实际上是高斯混合模型的一个特例，每个聚类在所有维度上的协方差都接近0。其次，根据高斯混合模型的使用概率，每个数据点可以有多个聚类。因此，如果一个数据点位于两个重叠的聚类的中间，通过说X%属于1类，而y%属于2类，我们可以简单地定义它的类。

## 层次聚类算法

层次聚类算法实际上分为两类：自上而下或自下而上。自下而上的算法在一开始就将每个数据点视为一个单一的聚类，然后依次合并（或聚集）类，直到所有类合并成一个包含所有数据点的单一聚类。因此，自下而上的层次聚类称为合成聚类或HAC。聚类的层次结构用一棵树（或树状图）表示。树的根是收集所有样本的唯一聚类，而叶子是只有一个样本的聚类。在继续学习算法步骤之前，先查看下面的图表。

图片参考网页：

<http://imgcdn.atyun.com/2018/03/1-ET8kCcPpr893vNZFs8j4xg.gif>

1.我们首先将每个数据点作为一个单独的聚类进行处理。如果我们的数据集有X个数据点，那么我们就有了X个聚类。然后我们选择一个度量两个聚类之间距离的距离度量。作为一个示例，我们将使用平均连接（average linkage）聚类，它定义了两个聚类之间的距离，即第一个聚类中的数据点和第二个聚类中的数据点之间的平均距离。

2.在每次迭代中，我们将两个聚类合并为一个。将两个聚类合并为具有最小平均连接的组。比如说根据我们选择的距离度量，这两个聚类之间的距离最小，因此是最相似的，应该组合在一起。

3.重复步骤2直到我们到达树的根。我们只有一个包含所有数据点的聚类。通过这种方式，我们可以选择最终需要多少个聚类，只需选择何时停止合并聚类，也就是我们停止建造这棵树的时候！

层次聚类算法不要求我们指定聚类的数量，我们甚至可以选择哪个聚类看起来最好。此外，该算法对距离度量的选择不敏感;  
它们的工作方式都很好，而对于其他聚类算法，距离度量的选择是至关重要的。层次聚类方法的一个特别好的用例是，当底层数据具有层次结构时，你可以恢复层次结构;而其他的聚类算法无法做到这一点。层次聚类的优点是以低效率为代价的，因为它具有O(n³)的时间复杂度，与K-Means和高斯混合模型的线性复杂度不同。

## k-means聚类算法

### 6.1 算法简介

K-means算法是很典型的基于距离的聚类算法，采用距离作为相似性的评价指标，即认为两个对象的距离越近，其相似度就越大。该算法认为簇是由距离靠近的对象组成的，因此把得到紧凑且独立的簇作为最终目标。

算法过程如下:

1)从N个文档随机选取K个文档作为质心

2)对剩余的每个文档测量其到每个质心的距离，并把它归到最近的质心的类

3)重新计算已经得到的各个类的质心

4)迭代2~3步直至新的质心与原质心相等或小于指定阈值，算法结束

具体如下:

输入:k, data[n];

(1) 选择k个初始中心点，例如c[0]=data[0],…c[k-1]=data[k-1];

(2) 对于data[0]….data[n]，分别与c[0]…c[k-1]比较，假定与c[i]差值最少，就标记为i;

(3) 对于所有标记为i点，重新计算c[i]={ 所有标记为i的data[j]之和}/标记为i的个数;

(4) 重复(2)(3)，直到所有c[i]值的变化小于给定阈值。

### 6.2代码实现案例

import numpy  
import random  
import matplotlib.pyplot as plt  
  
  
def findCentroids(data\_get, k): # 随机获取k个质心  
  
 return random.sample(data\_get, k)  
  
  
def calculateDistance(vecA, vecB): # 计算向量vecA和向量vecB之间的欧氏距离  
  
 return numpy.sqrt(numpy.sum(numpy.square(vecA - vecB)))  
  
  
def minDistance(data\_get, centroidList):  
 # 计算data\_get中的元素与centroidList中k个聚类中心的欧式距离，找出距离最小的  
 # 将该元素加入相应的聚类中  
  
 clusterDict = dict() # 用字典存储聚类结果  
 for element in data\_get:  
 vecA = numpy.array(element) # 转换成数组形式  
 flag = 0 # 元素分类标记，记录与相应聚类距离最近的那个类  
 minDis = float("inf") # 初始化为最大值  
  
 for i in range(len(centroidList)):  
 vecB = numpy.array(centroidList[i])  
 distance = calculateDistance(vecA, vecB) # 两向量间的欧式距离  
 if distance < minDis:  
 minDis = distance  
 flag = i # 保存与当前item距离最近的那个聚类的标记  
  
 if flag not in clusterDict.keys(): # 簇标记不存在，进行初始化  
 clusterDict[flag] = list()  
 clusterDict[flag].append(element) # 加入相应的类中  
  
 return clusterDict # 返回新的聚类结果  
  
  
def getCentroids(clusterDict):  
 centroidList = list()  
 for key in clusterDict.keys():  
 centroid = numpy.mean(numpy.array(clusterDict[key]), axis=0) # 求聚类中心即求解每列的均值  
 centroidList.append(centroid)  
  
 return numpy.array(centroidList).tolist()  
  
  
def calculate\_Var(clusterDict, centroidList):  
 # 计算聚类间的均方误差  
 # 将类中各个向量与聚类中心的距离进行累加求和  
  
 sum = 0.0  
 for key in clusterDict.keys():  
 vecA = numpy.array(centroidList[key])  
 distance = 0.0  
 for item in clusterDict[key]:  
 vecB = numpy.array(item)  
 distance += calculateDistance(vecA, vecB)  
 sum += distance  
  
 return sum  
  
  
def showCluster(centroidList, clusterDict):  
 # 画聚类结果  
  
 colorMark = ['or', 'ob', 'og', 'ok', 'oy', 'ow'] # 元素标记  
 centroidMark = ['dr', 'db', 'dg', 'dk', 'dy', 'dw'] # 聚类中心标记  
 for key in clusterDict.keys():  
 plt.plot(centroidList[key][0], centroidList[key][1], centroidMark[key], markersize=12) # 画聚类中心  
 for item in clusterDict[key]:  
 plt.plot(item[0], item[1], colorMark[key]) # 画类下的点  
  
 plt.show()  
  
  
data = [[0.0, 0.0], [3.0, 8.0], [2.0, 2.0], [1.0, 1.0], [5.0, 3.0],  
 [4.0, 8.0], [6.0, 3.0], [5.0, 4.0], [6.0, 4.0], [7.0, 5.0]]  
  
if \_\_name\_\_ == '\_\_main\_\_':  
  
 centroidList = findCentroids(data, 3) # 随机获取3个聚类中心  
 clusterDict = minDistance(data, centroidList) # 第一次聚类迭代  
 newVar = calculate\_Var(clusterDict, centroidList) # 计算均方误差值，通过新旧均方误差来获得迭代终止条件  
 oldVar = -0.0001 # 初始化均方误差  
  
 print('\*\*\*\*\* 第1次迭代 \*\*\*\*\*')  
 for key in clusterDict.keys():  
 print('聚类中心: ', centroidList[key])  
 print('对应聚类: ', clusterDict[key])  
 print('平均均方误差: ', newVar)  
 showCluster(centroidList, clusterDict) # 展示聚类结果  
  
 k = 2  
 while abs(newVar - oldVar) >= 0.0001: # 当连续两次聚类结果差距小于0.0001时，迭代结束  
 centroidList = getCentroids(clusterDict) # 获得新的聚类中心  
 clusterDict = minDistance(data, centroidList) # 新的聚类结果  
 oldVar = newVar  
 newVar = calculate\_Var(clusterDict, centroidList)  
  
 print('\*\*\*\*\* 第%d次迭代 \*\*\*\*\*' % k)  
  
 for key in clusterDict.keys():  
 print('聚类中心: ', centroidList[key])  
 print('对应聚类: ', clusterDict[key])  
 print('平均均方误差: ', newVar)  
 showCluster(centroidList, clusterDict) # 展示聚类结果  
  
 k += 1

运行结果如下：

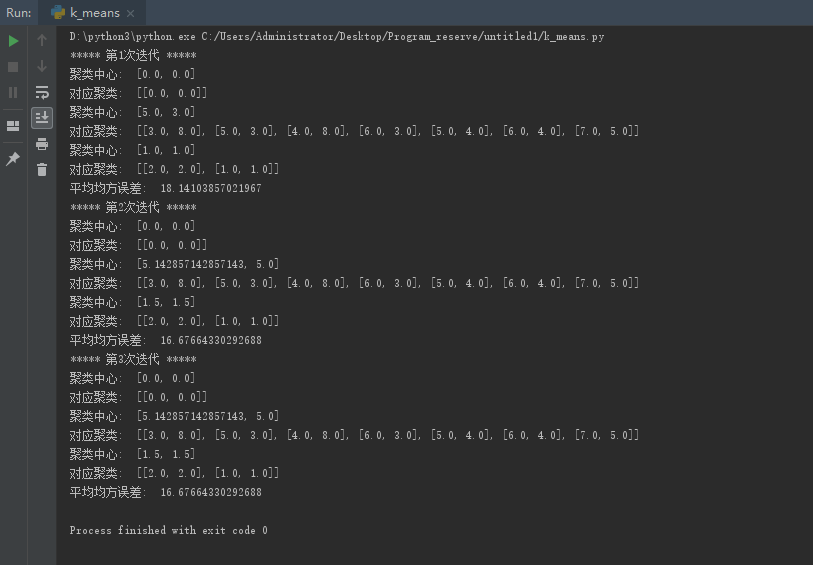


图5.2

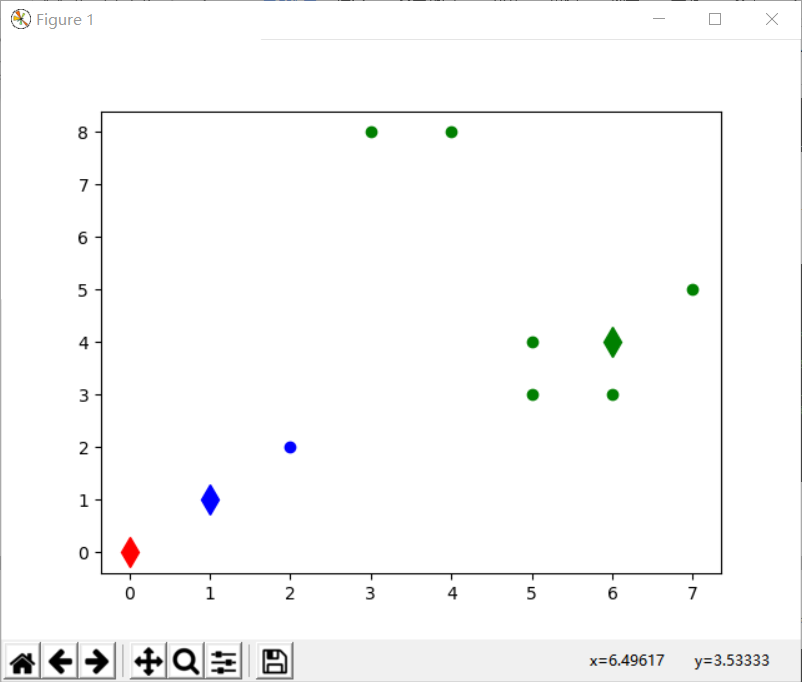


图5.3

# 六、关联规则

## 1.什么是关联规则

所谓数据挖掘就是以某种方式分析源数据，从中发现一些潜在的有用的信息，即数据挖掘又可以称作知识发现。而机器学习算法则是这种“某种方式”，关联规则作为十大经典机器学习算法之一，因此搞懂关联规则（虽然目前使用的不多）自然有着很重要的意义。顾名思义，关联规则就是发现数据背后存在的某种规则或者联系。

　　举个简单的例子（尿布和啤酒太经典）：通过调研超市顾客购买的东西，可以发现30%的顾客会同时购买床单和枕套，而在购买床单的顾客中有80%的人购买了枕套，这就存在一种隐含的关系：床单→枕套，也就是说购买床单的顾客会有很大可能购买枕套，因此商场可以将床单和枕套放在同一个购物区，方便顾客购买。

　　一般，关联规则可以应用的场景有：

优化货架商品摆放或者优化邮寄商品的目录

交叉销售或者捆绑销售

搜索词推荐或者识别异常

## 2.概念

①项目：交易数据库中的一个字段，对超市的交易来说一般是指一次交易中的一个物品，如：牛奶

②事务：某个客户在一次交易中，发生的所有项目的集合：如｛牛奶，面包，啤酒｝

③项集：包含若干个项目的集合（一次事务中的），一般会大于0个

④支持度：项集｛X，Y｝在总项集中出现的概率（见下面的例子）

⑤频繁项集：某个项集的支持度大于设定阈值（人为设定或者根据数据分布和经验来设定），即称这个项集为频繁项集。

⑥置信度：在先决条件X发生的条件下，由关联规则｛X->Y ｝推出Y的概率（见下面的例子）

⑦提升度：表示含有X的条件下同时含有Y的概率，与无论含不含X含有Y的概率之比。

　　支持度和提升度示例：

　　假如有一条规则：牛肉—>鸡肉，那么同时购买牛肉和鸡肉的顾客比例是3/7，而购买牛肉的顾客当中也购买了鸡肉的顾客比例是3/4。这两个比例参数是很重要的衡量指标，它们在关联规则中称作支持度（support）和置信度（confidence）。对于规则：牛肉—>鸡肉，它的支持度为3/7，表示在所有顾客当中有3/7同时购买牛肉和鸡肉，其反应了同时购买牛肉和鸡肉的顾客在所有顾客当中的覆盖范围；它的置信度为3/4，表示在买了牛肉的顾客当中有3/4的人买了鸡肉，其反应了可预测的程度，即顾客买了牛肉的话有多大可能性买鸡肉。其实可以从统计学和集合的角度去看这个问题， 假如看作是概率问题，则可以把“顾客买了牛肉之后又多大可能性买鸡肉”看作是条件概率事件，而从集合的角度去看，可以看下面这幅图：

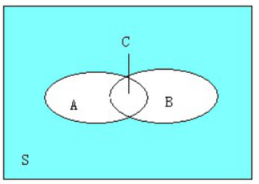


图6.1

    上面这副图可以很好地描述这个问题，S表示所有的顾客，而A表示买了牛肉的顾客，B表示买了鸡肉的顾客，C表示既买了牛肉又买了鸡肉的顾客。那么C.count/S.count=3/7，C.count/A.count=3/4。

　　提升度示例：

1000名顾客，购买年货，A组有500人购买茶叶，有450人购买咖啡；B组有0人购买茶叶，有450人购买咖啡。

|  |  |  |
| --- | --- | --- |
|  | 购买茶叶 | 购买咖啡 |
| A组（500人） | 500 | 450 |
| B组（500人） | 0 | 450 |
|  |  |  |

茶叶->咖啡的支持度＝450/1000=45%

　　茶叶->咖啡的置信度＝450/500=90%

　　茶叶->咖啡的提升度＝90%／90%＝1

　　说明：

　　（1）由于lift(茶叶X->咖啡Y)＝1，所以说明X与Y相互独立，即是否有X对于Y的出现没有影响。虽然支持度和置信度都高，但它们之间没有必然的关联关系。

　　（2）满足最小支持度和最小置信度的关联关系叫做强关联关系

如果lift>1，叫做有效的强关联关系，

如果lift<=1，叫做无效的强关联关系

特别的如果lift（X->Y）＝1，则称X与Y相互独立

## 3.实现过程

从以上的分析可以得知，关联规则是从事务集合中挖掘出这样的关联规则｛X->Y｝：它的支持度和置信度要大于最小阈值（minSup，minConf），当然这个最小阈值是由用户指定的，可以根据数据分布和经验；同时他的提升度最好是大于1的（具体值根据实际情况设定，例如：3、5均可），即是有效强关联规则。

　　使用关联规则的过程主要包含以下三个步骤：

（1）数据筛选，首先对数据进行清洗，清洗掉那些公共的项目，比如：热门词，通用词（此步依据具体项目而定）

（2）根据支持度（support），从事务集合中找出频繁项集（使用算法：Apriori算法，FP-Growth算法）

（3）根据置信度（confidence），从频繁项集中找出强关联规则（置信度阈值需要根据实验或者经验而定）

（4）根据提升度（lift），从强关联规则中筛选出有效的强关联规则（提升度的设定需要经过多次试验确定）

## 4.代码实现案例

import os,itertools  
import numpy as np  
import pandas as pd  
class Apriori(object):  
 def \_\_init\_\_(self, itemSets, minSupport=0.5, minConf=0.7, sort = False):  
 self.itemSets = itemSets  
 self.minSupport = minSupport  
 self.minConf = minConf  
 self.sort = sort  
 self.\_\_Initialize()  
 def \_\_Initialize(self):  
 self.\_\_item()  
 self.\_\_creat\_matrix()  
 self.update(minSupport=self.minSupport, minConf=self.minConf)  
 def \_\_item(self):  
 *''' 获取项目元素列表 '''* self.item = []  
 for itemSet in self.itemSets:  
 for item in itemSet:  
 if item not in self.item:  
 self.item.append(item)  
 self.item.sort()  
 def \_\_creat\_matrix(self):  
 *''' 将项集转为 pandas.DataFrame 数据类型 '''* self.data = pd.DataFrame(columns=self.item)  
 for i in range(len(self.itemSets)):  
 self.data.loc[i, self.itemSets[i]] = 1  
 def \_\_candidate\_itemsets\_l1(self):  
 *''' 创建单项频繁项集及 L1'''* self.L1 = self.data.loc[:, self.data.sum(axis=0) / len(self.itemSets) >= self.minSupport]  
 self.L1\_support\_selects = dict(self.L1.sum(axis=0) / len(self.itemSets)) # 只作为分母，不进行拆分  
 def \_\_candidate\_itemsets\_lk(self):  
 *''' 根据 L1 创建多项频繁项集 Lk ，非频繁项集的任何超集都不是频繁项集 '''* last\_support\_selects = self.L1\_support\_selects.copy() # 初始化  
 while last\_support\_selects:  
 new\_support\_selects = {}  
 for last\_support\_select in last\_support\_selects.keys():  
 for L1\_support\_name in set(self.L1.columns) - set(last\_support\_select.split(',')):  
 columns = sorted([L1\_support\_name] + last\_support\_select.split(',')) # 新的列名：合并后排序  
 count = (self.L1.loc[:, columns].sum(axis=1) == len(columns)).sum()  
 if count / len(self.itemSets) >= self.minSupport:  
 new\_support\_selects[','.join(columns)] = count / len(self.itemSets)  
 self.support\_selects.update(new\_support\_selects)  
 last\_support\_selects = new\_support\_selects.copy() # 作为新的 Lk，进行下一轮更新  
 def \_\_support\_selects(self):  
 *''' 支持度选择 '''* self.\_\_candidate\_itemsets\_l1()  
 self.\_\_candidate\_itemsets\_lk()  
 self.item\_Conf = self.L1\_support\_selects.copy()  
 self.item\_Conf.update(self.support\_selects)  
 def \_\_confidence\_selects(self):  
 *''' 生成关联规则，其中 support\_selects 已经按照长度大小排列 '''* for groups, Supp\_groups in self.support\_selects.items():  
 groups\_list = groups.split(',')  
 for recommend\_len in range(1, len(groups\_list)):  
 for recommend in itertools.combinations(groups\_list, recommend\_len):  
 items = ','.join(sorted(set(groups\_list) - set(recommend)))  
 Conf = Supp\_groups / self.item\_Conf[items]  
 if Conf >= self.minConf:  
 self.confidence\_select.setdefault(items, {})  
 self.confidence\_select[items].setdefault(','.join(recommend),{'Support': Supp\_groups,  
'Confidence': Conf})  
 def show(self,\*\*kwargs):  
 *''' 可视化输出 '''* if kwargs.get('data'):  
 select = kwargs['data']  
 else:  
 select = self.confidence\_select  
 items = []  
 value = []  
 for ks, vs in select.items():  
 items.extend(list(zip([ks] \* vs.\_\_len\_\_(), vs.keys())))  
 for v in vs.values():  
 value.append([v['Support'], v['Confidence']])  
 index = pd.MultiIndex.from\_tuples(items, names=['Items', 'Recommend'])  
 self.rules = pd.DataFrame(value, index=index, columns=['Support', 'Confidence'])  
 if self.sort or kwargs.get('sort'):  
 result = self.rules.sort\_values(by=['Support', 'Confidence'], ascending=False)  
 else:  
 result = self.rules.copy()  
 return result  
 def update(self, \*\*kwargs):  
 *''' 用于更新数据 '''* if kwargs.get('minSupport'):  
 self.minSupport = kwargs['minSupport']  
 self.support\_selects = {} # 用于储存满足支持度的频繁项集  
 self.\_\_support\_selects()  
 if kwargs.get('minConf'):  
 self.minConf = kwargs['minConf']  
 self.confidence\_select = {} # 用于储存满足自信度的关联规则  
 self.\_\_confidence\_selects()  
 print(self.show())  
 if kwargs.get('file\_name'):  
 file\_name = kwargs['file\_name']  
 self.show().to\_excel(f'/../table/{file\_name}.xlsx')  
 self.apriori\_rules = self.rules.copy()  
 def \_\_get\_Recommend\_list(self,itemSet):  
 *''' 输入数据，获取关联规则列表 '''* self.recommend\_selects = {}  
 itemSet = set(itemSet) & set(self.apriori\_rules.index.levels[0])  
 if itemSet:  
 for start\_str in itemSet:  
 for end\_str in self.apriori\_rules.loc[start\_str].index:  
 start\_list = start\_str.split(',')  
 end\_list = end\_str.split(',')  
 self.\_\_creat\_Recommend\_list(start\_list, end\_list, itemSet)  
 def \_\_creat\_Recommend\_list(self,start\_list,end\_list,itemSet):  
 *''' 迭代创建关联规则列表 '''* if set(end\_list).issubset(itemSet):  
 start\_str = ','.join(sorted(start\_list+end\_list))  
 if start\_str in self.apriori\_rules.index.levels[0]:  
 for end\_str in self.apriori\_rules.loc[start\_str].index:  
 start\_list = start\_str.split(',')  
 end\_list = end\_str.split(',')  
 self.\_\_creat\_Recommend\_list(sorted(start\_list),end\_list,itemSet)  
 elif not set(end\_list) & itemSet:  
 start\_str = ','.join(start\_list)  
 end\_str = ','.join(end\_list)  
 self.recommend\_selects.setdefault(start\_str, {})  
 self.recommend\_selects[start\_str].setdefault(end\_str, {'Support': self.apriori\_rules.loc[(start\_str,  
end\_str), 'Support'], 'Confidence': self.apriori\_rules.loc[(start\_str, end\_str), 'Confidence']})  
 def get\_Recommend(self,itemSet,\*\*kwargs):  
 *''' 获取加权关联规则 '''* self.recommend = {}  
 self.\_\_get\_Recommend\_list(itemSet)  
 self.show(data = self.recommend\_selects)  
 items = self.rules.index.levels[0]  
 for item\_str in items:  
 for recommends\_str in self.rules.loc[item\_str].index:  
 recommends\_list = recommends\_str.split(',')  
 for recommend\_str in recommends\_list:  
 self.recommend.setdefault(recommend\_str,0)  
 self.recommend[recommend\_str] += self.rules.loc[(item\_str,recommends\_str),'Support'] \* self.rules.loc[(item\_str,recommends\_str),'Confidence'] \* self.rules.loc[item\_str,'Support'].mean()/(self.rules.loc[item\_str,'Support'].sum()\*len(recommends\_list))  
 result = pd.Series(self.recommend,name='Weight').sort\_values(ascending=False)  
 result.index.name = 'Recommend'  
 result = result/result.sum()  
 result = 1/(1+np.exp(-result))  
 print(result)  
 if kwargs.get('file\_name'):  
 file\_name = kwargs['file\_name']  
 excel\_writer = pd.ExcelWriter(f'{os.getcwd()}/../table/{file\_name}.xlsx')  
 result.to\_excel(excel\_writer,'推荐项目及权重')  
 self.rules.to\_excel(excel\_writer, '关联规则树状表')  
 self.show().to\_excel(excel\_writer, '总关联规则树状表')  
 self.show(sort = True).to\_excel(excel\_writer, '总关联规则排序表')  
 excel\_writer.save()  
 return result  
def str2itemsets(strings, split=','):  
 *''' 将字符串列表转化为对应的集合 '''* itemsets = []  
 for string in strings:  
 itemsets.append(sorted(string.split(split)))  
 return itemsets  
if \_\_name\_\_ == '\_\_main\_\_':  
 # 1.导入数据  
 data = pd.read\_excel(r'apriori算法实现.xlsx', index=False)  
 # 2.关联规则中不考虑多次购买同一件物品，删除重复数据  
 data = data.drop\_duplicates()  
 # 3.初始化列表  
 itemSets = []  
 # 3.按销售单分组，只有 1 件商品的没有意义，需要进行过滤  
 groups = data.groupby(by='销售单明细')  
 for group in groups:  
 if len(group[1]) >= 2:  
 itemSets.append(group[1]['商品编码'].tolist())  
 # 4.训练 Apriori  
 ap = Apriori(itemSets, minSupport=0.03, minConf=0.5)  
 ap.get\_Recommend('2BYP206,2BYW001-,2BYW013,2BYX029'.split(','))

# 七、离群点挖掘

## 1.概述

离群点检测是数据挖掘中重要的一部分，它的任务是发现与大部分其他对象显著不同的对象。大部分数据挖掘方法都将这种差异信息视为噪声而丢弃，然而在一些应用中，罕见的数据可能蕴含着更大的研究价值。

离群点的检测已经被广泛应用于电信和信用卡的诈骗检测、贷款审批、电子商务、网络入侵和天气预报等领域。

（1）离群点的成因

离群点的主要成因有：数据来源于不同的类、自然变异、数据测量和手机误差。

（2）离群点的类型

从数据范围来看，分为全局离群点和局部离群点，整体来看，某些对象没有离群特征，但是从局部来看，却显示了一定的离群性。

从数据类型来看，分为数值型离群点和分类型离群点，这是以数据集的属性类型进行划分的。

从属性的个数来看，分为一维离群点和多维离群点，一个对象可能有一个或多个属性。

（3）离群点的检测方法

基于统计：

大部分的基于统计的离群点检测方法是构建一个概率分布模型，并计算对象符合该模型的概率，把具有低概率的对象视为离群点。基于统计模型的离群点检测方法的前提是必须知道数据集服从什么分布；对于高维数据，检验效果可能很差。

基于邻近度：

通常可以在数据对象之间定义邻近性度量，把原理大部分点的对象视为离群点。二位或三维的数据可以做散点图观察；大数据集不适用；对参数选择敏感；具有全局阈值，不能处理具有不同密度区域的数据集

基于密度：

考虑数据集可能存在不同密度区域这一事实，从基于密度的观点分析，离群点是在低密度区域中的对象。一个对象的离群点得分是该对象周围密度的逆。给出了对象是离群点的定量度量，并且即使数据具有不同的区域也能够很好的处理；大数据集不适用；参数选择是困难的。

基于聚类：

一种利用聚类检测离群点的方法是丢弃远离其他簇的小簇；另一种更系统的方法，首先聚类所有帝乡，然后评估对象属于簇的程度。基于聚类技术来发现离群点可能是高度有效的；聚类算法产生的簇的质量对该算法产生的离群点的质量影响非常大。

基于统计模型的离群点检测方法需要满足统计学原理，如果分布一直，则检验可能非常有效。基于邻近度的离群点检测方法比统计学方法更一般、更容易使用，因为确定数据集有意义的邻近度量比确定他的统计分布更容易。基于密度的离群点检测与基于邻近度的离群点检测密切相关，因为密度常用邻近度定义：一种是定义密度为到K个最邻近的平均距离的倒数，如果该距离小，则密度高；另一种是使用DBSCAN聚类算法，一个对象周围的密度等于该对象指定距离d内对象的个数。

## 2.基于相对密度的方法实现案例

代码实现如下：

import pandas as pd  
import numpy as np  
import matplotlib.pyplot as plt  
import xlrd  
def localoutlierfactor(data, predict, k):  
 from sklearn.neighbors import LocalOutlierFactor  
 clf = LocalOutlierFactor(n\_neighbors=k + 1, algorithm='auto', contamination=0.1, n\_jobs=-1)  
 clf.fit(data)  
# 记录 k 邻域距离  
 predict['k distances'] = clf.kneighbors(predict)[0].max(axis=1)  
# 记录 LOF 离群因子，做相反数处理  
 predict['local outlier factor'] = -clf.\_decision\_function(predict.iloc[:, :-1])  
 return predict  
def plot\_lof(result, method):  
 import matplotlib.pyplot as plt  
 plt.rcParams['font.sans-serif'] = ['SimHei'] # 用来正常显示中文标签  
 plt.rcParams['axes.unicode\_minus'] = False # 用来正常显示负号  
 plt.figure(figsize=(8, 4)).add\_subplot(111)  
 plt.scatter(result[result['local outlier factor'] > method].index,  
 result[result['local outlier factor'] > method]['local outlier factor'], c='red', s=50,marker='.', alpha=None,label='离群点')  
 plt.scatter(result[result['local outlier factor'] <= method].index,result[result['local outlier factor'] <= method]['local outlier factor'], c='black', s=50,  
marker='.', alpha=None, label='正常点')  
 plt.hlines(method, -2, 2 + max(result.index), linestyles='--')  
 plt.xlim(-2, 2 + max(result.index))  
 plt.title('LOF 局部离群点检测', fontsize=13)  
 plt.ylabel('局部离群因子', fontsize=15)  
 plt.legend()  
 plt.show()  
def lof(data, predict=None, k=5, method=1, plot=False):  
 import pandas as pd  
# 判断是否传入测试数据，若没有传入则测试数据赋值为训练数据  
 try:  
 if predict == None:  
 predict = data.copy()  
 except Exception:  
 pass  
 predict = pd.DataFrame(predict)  
# 计算 LOF 离群因子  
 predict = localoutlierfactor(data, predict, k)  
 if plot == True:  
 plot\_lof(predict, method)  
# 根据阈值划分离群点与正常点  
 outliers = predict[predict['local outlier factor'] > method].sort\_values(by='local outlier factor')  
 inliers = predict[predict['local outlier factor'] <= method].sort\_values(by='local outlier factor')  
 return outliers, inliers  
  
# 根据文件位置自行修改  
posi = pd.read\_excel(r'D:\Python\云计算课设\已结束项目任务数据.xls') #注意输入文件的具体地址  
lon = np.array(posi["任务gps经度"][:]) # 注意excel文档中经度一栏的格式，gps与经度之间没有空格  
lat = np.array(posi["任务gps 纬度"][:]) # 纬度  
A = list(zip(lat, lon)) # 按照纬度-经度匹配  
# 获取任务密度，取第5 邻域，阈值为2（LOF 大于2 认为是离群值）  
outliers1, inliers1 = lof(A, k=5, method = 2)  
  
for k in [3,5,10]:  
 plt.figure('k=%d'%k)  
 outliers1, inliers1 = lof(A, k=k, method = 2)  
 plt.scatter(np.array(A)[:,0],np.array(A)[:,1],s = 10,c='b',alpha = 0.5)  
 plt.scatter(outliers1[0],outliers1[1],s = 10+outliers1['local outlier factor']\*100,c='r',alpha = 0.2)  
 plt.title('k=%d' % k)  
 plt.show()

plt.title('k=%d' % k)  
 plt.show()

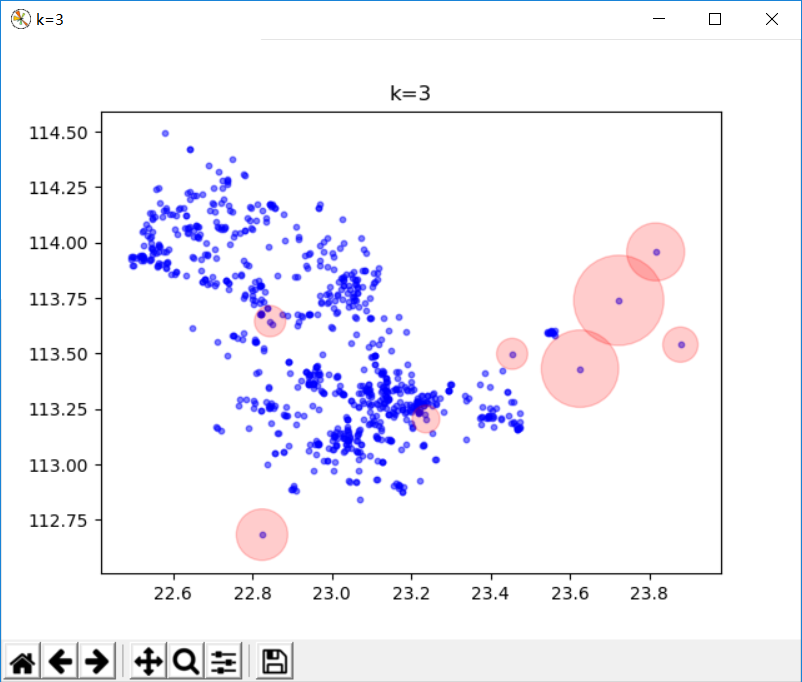


图7.1 k=3运行结果

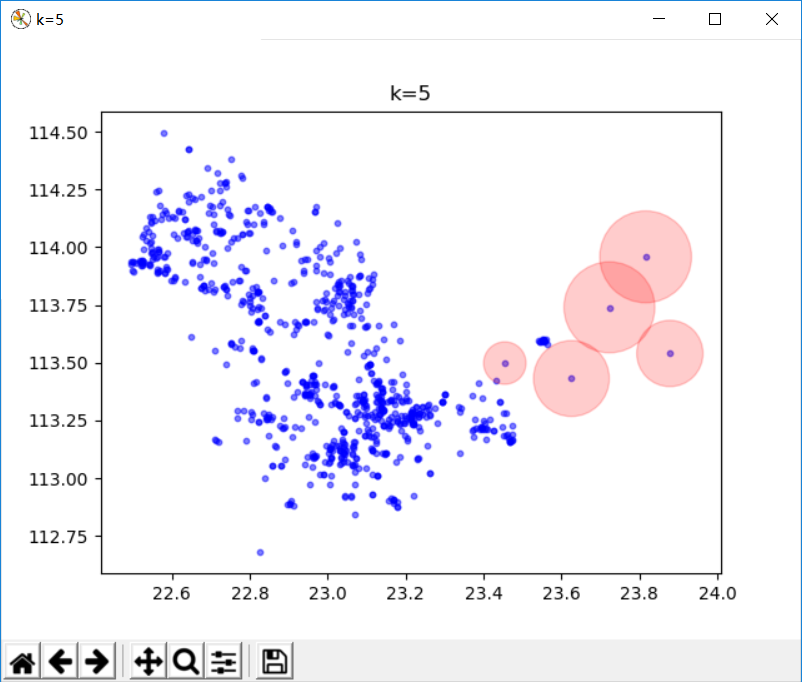


图7.1 k=5运行结果

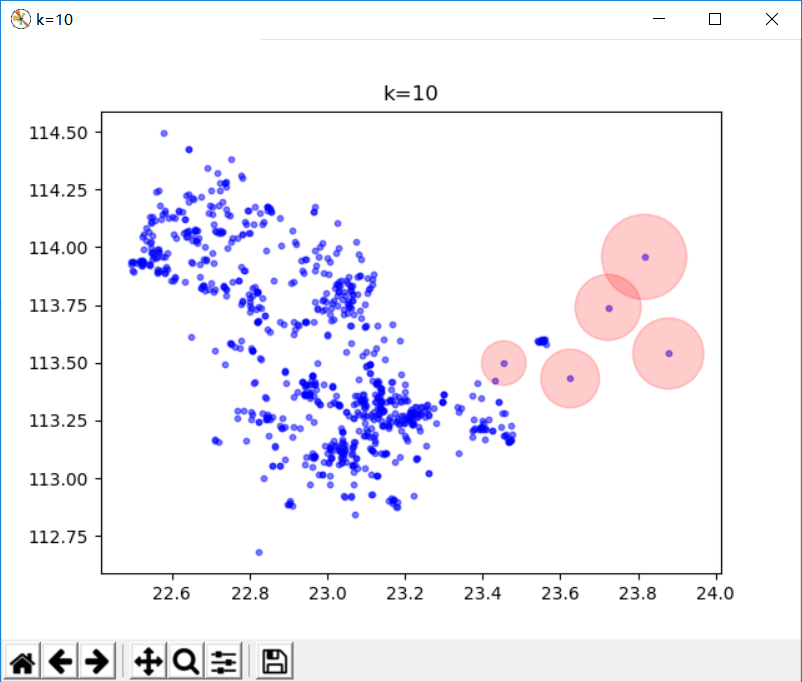


图7.1 k=7运行结果

# 八、神经网络

## 1.概述

所谓人工神经网络就是基于模仿生物大脑的结构和功能而构成的一种信息处理系统（计算机）。需要指出，尽管人工神经网络是对大脑结构的模仿，但这种模仿目前还处于极低的水平。

人工神经网络的过程可以分为两阶段，一是训练学习、二是操作预测：

1. 训练学习

训练时，把要教给神经网络的信息（外部输入）作为网络的输入和要求的输出，使网络按某种规则（称为训练算法）调节各处理单元间的连接权值，直至加上给定输入，网络就能产生给定输出为止。这时，各连接权已调接好，网络的训练就完成了。

2. 正常操作（回忆操作）

对训练好的网络输入一个信号，它就可以正确回忆出相应输出，得到识别结果。

人工神经网络的学习主要分为两类，有指导的训练：将输入样本加到网络输入并给出相应的输出，通过多次训练迭代获得连接权值。好像告诉网络：“当你看到这个图形（比如5）时，请给我指示5”。无指导的训练：网络通过训练自行调节连接加权，从而对输入样本分类。在网络训练时，有时只能给出大量的输入图形，没有指定它们的输出，网络就自行按输入图形的特征对它们进行分类。如小孩通过大量观察可以分辨出哪是狗、哪是猫一样。

## 神经网络实现边界决策

首先我们生成一个可以操作的数据集，幸运的是，[scikit-learn](http://scikit-learn.org/stable/)提供了一些有用的数据集生成器，所以我们不需要自己写代码来生成数据集，只需使用[make\_moons](http://scikit-learn.org/stable/modules/generated/sklearn.datasets.make_moons.html)这个函数就可以。

# 生成数据集并绘制出来  
np.random.seed(0)  
X, y = sklearn.datasets.make\_moons(200, noise=0.20)  
plt.scatter(X[:,0], X[:,1], s=40, c=y, cmap=plt.cm.Spectral)

我们生成的数据集中有两种类型的数据，分别用红点和蓝点标识了出来。你可以将蓝点视为男性患者，将红点视为女性患者，并且将x轴和y轴视为医疗方式。

我们的目标是训练一个机器学习分类器，让它在x、y坐标系下预测正确的类别（男性或者女性）。需要注意的是这些数据不能被线性分割，我们无法画出一条直线将这两种类型的数据分开，这意味着线性分类器（比如Logistic回归）将无法拟合这些数据，除非你手工设计的非线性特征（如多项式），为给定的数据集工作良好。

事实上，这就是神经网络主要的优点之一，你不必担心[特征工程](https://machinelearningmastery.com/discover-feature-engineering-how-to-engineer-features-and-how-to-get-good-at-it/)，神经网络的隐藏层将会为你学习特征。

Logistic回归

为了证明这一点，让我们训练一个Logistic回归分类器。这个分类器的输入是坐标x、y，它的输出是预测的数据类型（0或1）。为了方便，我们使用scikit-learn中的Logistic Regression类。

# Train the logistic rgeression classifier  
clf = sklearn.linear\_model.LogisticRegressionCV()  
clf.fit(X, y)  
  
# Plot the decision boundary  
plot\_decision\_boundary(lambda x: clf.predict(x))  
plt.title("Logistic Regression")

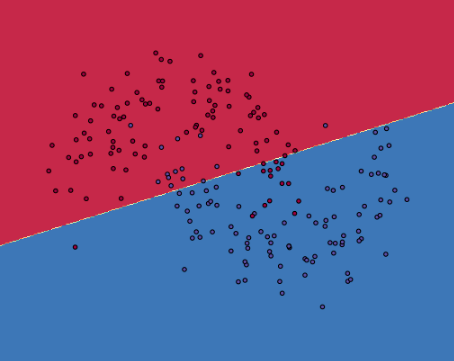


图8.1

上图显示了Logistic回归分类器学习的决策边界，它用直线尽可能地把数据分开了，但是并没有捕获数据的月牙形状。

训练一个神经网络

现在让我们搭建一个3层的神经网络，其中包含1个输入层，1个隐藏层以及1个输出层。输出层的节点数取决于我们的数据的维度，也就是2；类似地，输出层的节点数取决于我们有多少类数据，这里也是2（因为我们只有两类数据，所以实际上可以只有一个输出节点，输出0或1，但是有两个输出节点会使得以后有多类数据的时候神经网络更容易扩展）。神经网络的输入是x、y坐标，而输出是两个概率，一个是类型为0（女性患者）的概率，另一个是类型为1（男性患者）的概率。这个神经网络看起来就像下面这样：

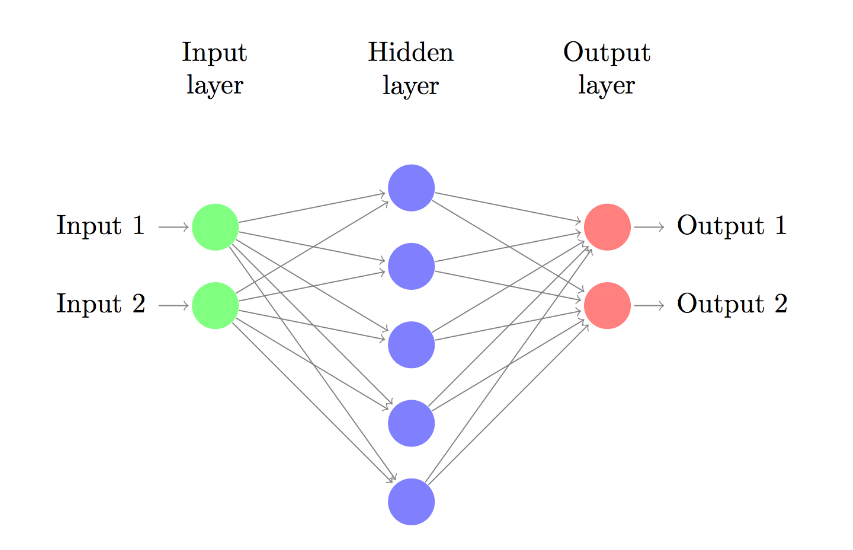


图8.2

我们可以自己选择隐藏层的维度（节点数量），隐藏层节点的数量越多，我们能够适应的功能就越复杂。但是高维度总是伴随着高成本。如果节点数量很多，首先，为了预测结果以及学习神经网络参数，就需要进行大量的计算；其次，更多的参数数量（译者注：隐藏层的参数量是由该层节点数和前一层的节点数决定的）意味着更容易造成对数据的过度拟合。

那么该如何选择隐藏层的节点数呢？尽管有一些通用指南和建议，但是隐藏层的节点数通常取决于你要解决的特定的问题，与其说节点数的选择是一门科学，不如说它是一门艺术。我们稍后会探究隐藏层节点的数量问题，并观察它是如何影响神经网络的输出的。

我们还需要为隐藏层选择一个激活函数，激活函数用来将该层的输入转换成输出。非线性激活函数能够使我们拟合非线性数据。常用的激活函数有[tanh](https://reference.wolfram.com/language/ref/Tanh.html)、[sigmoid](https://en.wikipedia.org/wiki/Sigmoid_function)以及[RELUs](https://en.wikipedia.org/wiki/Rectifier_(neural_networks))，在这里我们使用tanh，因为它在很多场景中都表现得很好。这些函数都有一个很棒的特性，那就是它们的导数都能用它们本身表示，比如，tanh(x)的导数是1−tanh2(x)1−tanh2(x)，这非常有用，因为我们只要计算tanh(x)的值一次，就可以在对tanh(x)求导的时候重复使用它。

因为我们想要让神经网络输出概率，所以输出层的激活函数我们使用[softmax](https://en.wikipedia.org/wiki/Softmax_function)，它可以简单地把原始数值转换成概率。如果你对逻辑函数很熟悉，那么你可以将softmax看作是逻辑函数在多类型中的一般化应用。

预测

我们的神经网络利用正向传播来做出预测，所谓正向传播，就是指一组矩阵的相乘以及我们之前提到的激活函数的应用。假设x是对神经网络的二维输入，那么我们按照如下步骤计算我们的预测结果y^y^（维度也是2）：

z1=xW1+b1a1=tanh(z1)z2=a1W2+b2a2=y^=softmax(z2)z1=xW1+b1a1=tanh(z1)z2=a1W2+b2a2=y^=softmax(z2)

zizi是第ii层的输入，aiai是第ii层使用激活函数计算后的输出，W1,b1,W2,b2W1,b1,W2,b2是我们这个神经网络的参数，是需要通过训练数据来让神经网络学习的。你可以将他们看作是在神经网络的不同层之间转换数据的矩阵。观察上面的矩阵乘法运算，我们可以算出这些矩阵的维度，如果隐藏层有500个节点，那么W1∈R2×500,b1∈R500,W2∈R500×2,b2∈R2W1∈R2×500,b1∈R500,W2∈R500×2,b2∈R2。现在你应该知道为什么如果我们增加隐藏层的节点数量，我们就会有更多的参数了。

参数传递

为神经网络学习参数意味着在训练数据上寻找最佳参数(W1,b1,W2,b2)(W1,b1,W2,b2)，以此达到使错误最小化的目的。但是我们应该如何定义“错误”呢？我们将衡量错误的函数称为损失函数（loss function）。对于softmax输出来说，一个常用的选择是分类交叉熵损失（[categorical cross-entropy loss](https://en.wikipedia.org/wiki/Cross_entropy#Cross-entropy_error_function_and_logistic_regression)，又称为负对数似然 negative log likelihood）。如果我们有N个训练样本，以及C个输出类别，那么我们的预测结果y^y^相对于真值标签yy的损失是这样定义的：

L(y,y^)=−1N∑n∈N∑i∈Cyn,ilogy^n,iL(y,y^)=−1N∑n∈N∑i∈Cyn,ilog⁡y^n,i

这个公式看起来复杂，其实它做的事情就是总结训练样本，如果预测错了类型，就增加损失（译者注：粗体部分的原文如下：sum over our training examples and add to the loss if we predicted the incorrect class. 我不确定粗体部分这样翻译是否合适，如果有网友有更好的翻译欢迎指正）。yy（正确的标签）和y^y^（我们的预测值）的数值相差越大，我们的损失就越大。通过寻找使损失最小化的参数，我们可以最大限度地提高训练数据的似然。

我们可以用[梯度下降法](http://cs231n.github.io/optimization-1/)来寻找损失函数的最小值。我会用固定的学习率实现一个最普通版本的梯度下降法，也称为批量梯度下降法，它的变化版本比如随机梯度下降法和小批量梯度下降法在实践中通常表现得更好，所以如果你对此要求严格，那么你一定会想要使用它们，最好随着时间的推移再配合以[学习率衰减](https://cs231n.github.io/neural-networks-3/#anneal)。

作为输入，梯度下降法需要计算损失函数对于参数的梯度（导数向量）：∂L∂W1,∂L∂b1,∂L∂W2,∂L∂b2∂L∂W1,∂L∂b1,∂L∂W2,∂L∂b2。我们利用著名的反向传播算法来计算这些梯度，从输出开始用反向传播计算梯度会很高效。对于反向传播的工作原理，我不会深入讲解其细节，但是网上有很多出色的解释（[这里](https://colah.github.io/posts/2015-08-Backprop/)或者[这里](https://cs231n.github.io/optimization-2/)）。

反馈

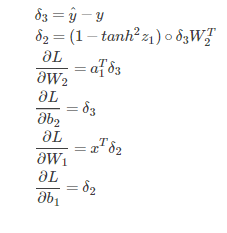


图8.3

代码实现如下：

# -\*- coding:utf-8 -\*-  
#!python3  
\_\_author\_\_ = 'Wsine'  
import numpy as np  
import sklearn  
import sklearn.datasets  
import sklearn.linear\_model  
import matplotlib.pyplot as plt  
import matplotlib  
import operator  
import time  
def createData(dim=200, cnoise=0.20):  
 *"""  
 输出：数据集, 对应的类别标签  
 描述：生成一个数据集和对应的类别标签  
 """* np.random.seed(0)  
 X, y = sklearn.datasets.make\_moons(dim, noise=cnoise)  
 plt.scatter(X[:, 0], X[:, 1], s=40, c=y, cmap=plt.cm.Spectral)  
 #plt.show()  
 return X, y  
def plot\_decision\_boundary(pred\_func, X, y):  
 *"""  
 输入：边界函数, 数据集, 类别标签  
 描述：绘制决策边界(画图用)  
 """* # 设置最小最大值, 加上一点外边界  
 x\_min, x\_max = X[:, 0].min() - .5, X[:, 0].max() + .5  
 y\_min, y\_max = X[:, 1].min() - .5, X[:, 1].max() + .5  
 h = 0.01  
 # 根据最小最大值和一个网格距离生成整个网格  
 xx, yy = np.meshgrid(np.arange(x\_min, x\_max, h), np.arange(y\_min, y\_max, h))  
 # 对整个网格预测边界值  
 Z = pred\_func(np.c\_[xx.ravel(), yy.ravel()])  
 Z = Z.reshape(xx.shape)  
 # 绘制边界和数据集的点  
 plt.contourf(xx, yy, Z, cmap=plt.cm.Spectral)  
 plt.scatter(X[:, 0], X[:, 1], c=y, cmap=plt.cm.Spectral)  
def calculate\_loss(model, X, y):  
 *"""  
 输入：训练模型, 数据集, 类别标签  
 输出：误判的概率  
 描述：计算整个模型的性能  
 """* W1, b1, W2, b2 = model['W1'], model['b1'], model['W2'], model['b2']  
 # 正向传播来计算预测的分类值  
 z1 = X.dot(W1) + b1  
 a1 = np.tanh(z1)  
 z2 = a1.dot(W2) + b2  
 exp\_scores = np.exp(z2)  
 probs = exp\_scores / np.sum(exp\_scores, axis=1, keepdims=True)  
 # 计算误判概率  
 corect\_logprobs = -np.log(probs[range(num\_examples), y])  
 data\_loss = np.sum(corect\_logprobs)  
 # 加入正则项修正错误(可选)  
 data\_loss += reg\_lambda/2 \* (np.sum(np.square(W1)) + np.sum(np.square(W2)))  
 return 1./num\_examples \* data\_loss  
def predict(model, x):  
 *"""  
 输入：训练模型, 预测向量  
 输出：判决类别  
 描述：预测类别属于(0 or 1)  
 """* W1, b1, W2, b2 = model['W1'], model['b1'], model['W2'], model['b2']  
 # 正向传播计算  
 z1 = x.dot(W1) + b1  
 a1 = np.tanh(z1)  
 z2 = a1.dot(W2) + b2  
 exp\_scores = np.exp(z2)  
 probs = exp\_scores / np.sum(exp\_scores, axis=1, keepdims=True)  
 return np.argmax(probs, axis=1)  
def initParameter(X):  
 *"""  
 输入：数据集  
 描述：初始化神经网络算法的参数  
 必须初始化为全局函数！  
 这里需要手动设置！  
 """* global num\_examples  
 num\_examples = len(X) # 训练集的大小  
 global nn\_input\_dim  
 nn\_input\_dim = 2 # 输入层维数  
 global nn\_output\_dim  
 nn\_output\_dim = 2 # 输出层维数  
 # 梯度下降参数  
 global epsilon  
 epsilon = 0.01 # 梯度下降学习步长  
 global reg\_lambda  
 reg\_lambda = 0.01 # 修正的指数  
def build\_model(X, y, nn\_hdim, num\_passes=20000, print\_loss=False):  
 *"""  
 输入：数据集, 类别标签, 隐藏层层数, 迭代次数, 是否输出误判率  
 输出：神经网络模型  
 描述：生成一个指定层数的神经网络模型  
 """* # 根据维度随机初始化参数  
 np.random.seed(0)  
 W1 = np.random.randn(nn\_input\_dim, nn\_hdim) / np.sqrt(nn\_input\_dim)  
 b1 = np.zeros((1, nn\_hdim))  
 W2 = np.random.randn(nn\_hdim, nn\_output\_dim) / np.sqrt(nn\_hdim)  
 b2 = np.zeros((1, nn\_output\_dim))  
 model = {}  
 # 梯度下降  
 for i in range(0, num\_passes):  
 # 正向传播  
 z1 = X.dot(W1) + b1  
 a1 = np.tanh(z1) # 激活函数使用tanh = (exp(x) - exp(-x)) / (exp(x) + exp(-x))  
 z2 = a1.dot(W2) + b2  
 exp\_scores = np.exp(z2) # 原始归一化  
 probs = exp\_scores / np.sum(exp\_scores, axis=1, keepdims=True)  
 # 后向传播  
 delta3 = probs  
 delta3[range(num\_examples), y] -= 1  
 dW2 = (a1.T).dot(delta3)  
 db2 = np.sum(delta3, axis=0, keepdims=True)  
 delta2 = delta3.dot(W2.T) \* (1 - np.power(a1, 2))  
 dW1 = np.dot(X.T, delta2)  
 db1 = np.sum(delta2, axis=0)  
 # 加入修正项  
 dW2 += reg\_lambda \* W2  
 dW1 += reg\_lambda \* W1  
 # 更新梯度下降参数  
 W1 += -epsilon \* dW1  
 b1 += -epsilon \* db1  
 W2 += -epsilon \* dW2  
 b2 += -epsilon \* db2  
 # 更新模型  
 model = { 'W1': W1, 'b1': b1, 'W2': W2, 'b2': b2}  
 # 一定迭代次数后输出当前误判率  
 if print\_loss and i % 1000 == 0:  
 print("Loss after iteration %i: %f" % (i, calculate\_loss(model, X, y)))  
 plot\_decision\_boundary(lambda x: predict(model, x), X, y)  
 plt.title("Decision Boundary for hidden layer size %d" % nn\_hdim)  
 #plt.show()  
 return model  
def main():  
 dataSet, labels = createData(200, 0.20)  
 initParameter(dataSet)  
 nnModel = build\_model(dataSet, labels, 3, print\_loss=False)  
 print("Loss is %f" % calculate\_loss(nnModel, dataSet, labels))  
if \_\_name\_\_ == '\_\_main\_\_':  
 start = time.clock()  
 main()  
 end = time.clock()  
 print('finish all in %s' % str(end - start))  
 plt.show()

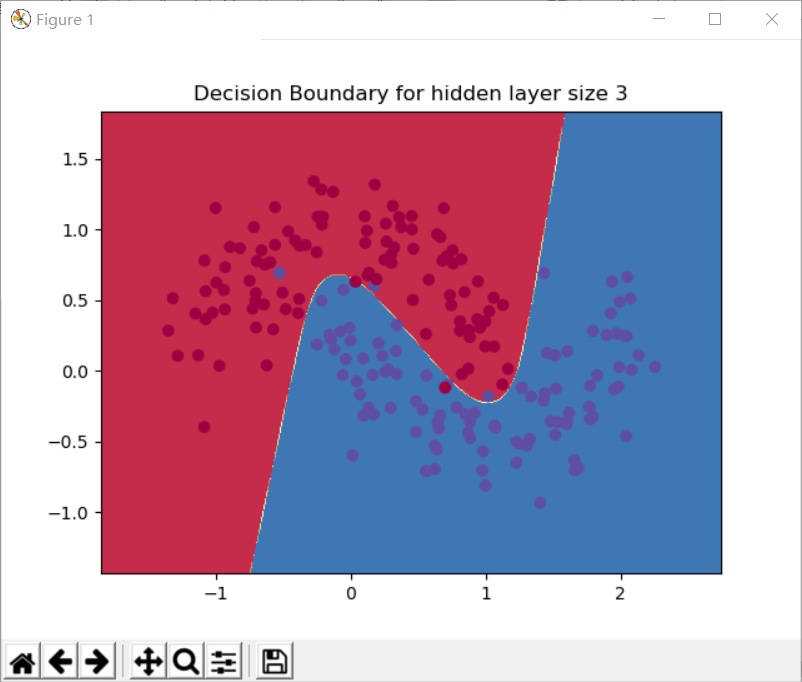


图8.4 运行结果