哈尔滨工业大学

<<大数据分析>>

实验报告之二

(2019 年度春季学期**)**

姓名:	姜思琪
学号:	**1160300814**
学院:	计算机学院
教师:	杨东华、王金宝

实验二 聚类与分类

一、实验目的

掌握对数据使用聚类分析与分类，并理解其在大数据环境下的实现方式。

二、实验环境

Windows、Linux（推荐），伪分布式 Hadoop 环境（推荐），Java、Python 等。

三、实验过程及结果

3.1 聚类分析

3.1.1 K-Means 聚类方法

1. K-Means 算法思想

首先，从原始数据集中随机选取 k 个数据对象，其中每个数据对象都代表一个初始聚类的质心；然后遍历数据集中所有的数据对象，把它们分配到相似度最高的一类簇中，其中相似度计算标准是以该数据对象和质心的距离长短来衡量的，即取数据对象与质心间最小欧式距离：

$$D(x, y) = \sqrt{(\sum_{i=1}^{m}(x_i - y_i)^2)};$$

其后该类簇的新的质心就会被重新计算，用最小误差平方和对其进行定义：

$$Z = \sum_{j=1}^{k}\sum_{i=1}^{Mi}(S_i - C^*_i)^2;$$

该过程继续以迭代的方式进行下去，重复步骤直到标准函数收敛，即当新中心点和旧中心点的距离低于某个选定的阈值时迭代结束。

伪代码如下：

1: Begin 初始化 n ,k ,C_1, C_2, ……, C_k

2: Do 按照最近邻 C_i 分类 n 个样本

3: 重新计算 C_i 得到 C_i’

4: If（C_i’ - C_i <= α, α 为迭代停止阈值）

5：　　Return C_1, C_2, …, C_k

6：　　Else $C_i = C_i'$

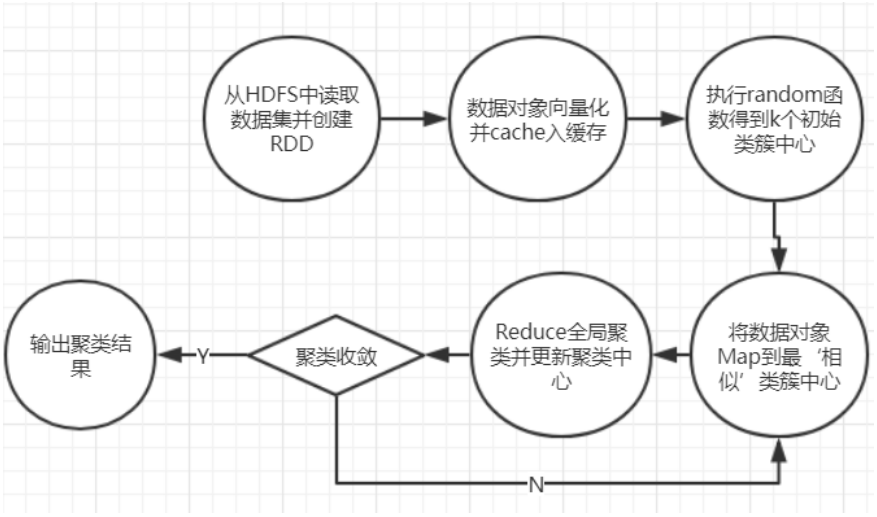
7：　　End If

8：　End Do

9：End

2.基于 Spark 实现 K-means

Kmeans 算法在 Spark 上并行化的流程图：



从 HDFS 读入预处理过的数据向量、输入参数对相应变量进行初始化。将每个数据分区以 RDD 的形式缓存到内存中。随机产生 k 个初始类簇中心。

Kmeans 算法主要代码段，当聚类准则比较变量 temp Dist 比聚类准则值 converge Dist 大，而且当前迭代次数 temp Iteration 小于最大迭代次数 Max Iteration 时，运行循环内容，具体执行流程如下：

求数据对象的局部数据聚类，由 closest Point 函数求得数据向量 p 与哪一个类簇中心的相似度最高，然后标记数据对象的所属类别，每个数据对象 p 可以映射成（id,（point，1））。point Stats 操作是对具有相同 id 的 p 进行合并，可以表示为（数据对象特征值和，数据对象数量和）。new Points 变量是指求得的新类簇中心点，键值对（id，数据对象特征值和/数据对象数量和）。将 temp Dist 重新归零。求新旧类簇中心点的差平方和。更新聚类中心。对当前迭代次数进行更新。

关键代码如下：

```
while(convergeDis < tempDis && tempIteration < 20){
  val pointStats = closest.groupByKey().map(x => {
    var update = Vector[Double]()
    val len = x._2.toArray.take(1)(0)._1.length
    var arrayBuffer = ArrayBuffer[Double]()
    var cnt =0
    for(ix <- 0 until len){
```

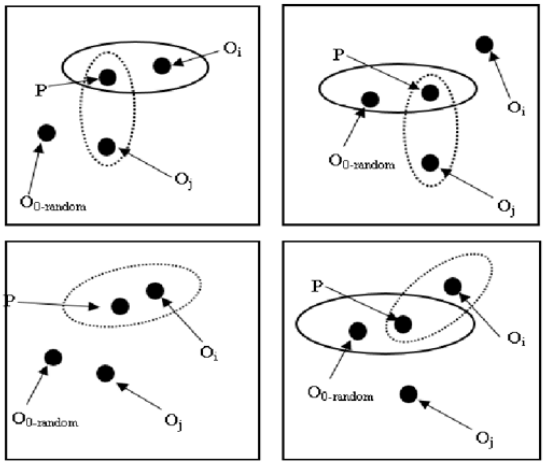
```
          arrayBuffer ++= ArrayBuffer(0.0)
        }
        for(arr <- x._2.toArray){
          val tr = arr._1.toArray
          cnt += arr._2
          for(i <- tr.indices){
            arrayBuffer(i) = arrayBuffer(i) +(tr(i)*arr._2.toDouble)
          }
        }
        for(i <- arrayBuffer.indices){
          arrayBuffer(i) = arrayBuffer(i)/cnt.toDouble
          update ++= Vector(arrayBuffer(i))
        }
        (x._1,update)
      })
      var arrayPointS = Array[Vector[Double]]()
      arrayPointS = pointStats.map(x => x._2).cache().take(K)
      tempDis = 0.0
      for(iii <- 0 until K){
          tempDis +=
squaredDis(kPoints(iii),DenseVector(arrayPointS(iii).toArray))
      }
      for(iii <- 0 until K){
          kPoints(iii) = DenseVector(arrayPointS(iii).toArray)
      }
      closest = data.map(x => ( closestPoint(x, kPoints),(x,1) ) )
      println("Finished iteration ( "+tempIteration+" , "+tempDis+"), center: ")
      kPoints.foreach(println)
      tempIteration = tempIteration + 1
    }
```

3.1.2 K-medoids 算法

1. 算法概述

这是对 K-means 算法的一个改进，K-means 算法计算出来的中心点为某个簇的均值，而 K-medoids 算法采取的簇中某个点到其他点的距离之和最小，作为中心点，这就避免了某些非质心点是离群点，这个点可能偏离整个簇的情况。

有 N 个数据对象的数据集 D，在数据集 D 中选取 k 个不同的对象作为初始的簇中心，计算每个对象到其他的簇中心的距离，并将到的这个距离进行累加求和，将这些对象分配给最近的簇中。依次计算某个簇它的非中心点到该簇中其他点的距离，且累加求和，比较大小，留下距离和小的中心点，依次反复迭代，算法结束的条件是新的簇中心点不再发生变化为止或者到达规定的迭代次数算法结束。K-medoids 算法的聚类代价函数的描述，如图所示：



K-medoids 算法聚类代价函数的 4 种情况

2. 在 Spark 上具体算法步骤描述：

步骤 1:任意选择数据集 D 中 k 个数据对象当做聚类的初始中心点。

```
val K = 5 //fen ji lei
val data = input.map(parseVector _).cache()
var kPoints = data.takeSample(withReplacement = false,K)
```

步骤 2:依次计算数据集 D 中剩余的每个对象到 k 个中心点的距离将他们重新分配到最近的簇中，并将剩余的每个对象到中心点的距离累加求和，记为 Sumi。

```
var closest = data.map(x =>{
  val mediumkeys = closestPoint(x, kPoints)
  (mediumkeys._1,(x,1,mediumkeys._2))
})
for(arr <- x._2.toArray){ //die dai qi, cacalate sumi and random point
      sumi += arr._3
      if(count == num) {
        comparepoint = arr._1
      }
      count += 1
 }
```

步骤 3:在某个簇中任选某个点 Ctest,计算它到簇中其他点的距离，并将这些距离累加求和，记为 Sumtest,如果 Sumtest<Sumi,则 Ctest 替换原来这个簇的中心点 Ci。

```
for(iii <- 0 until K){
    tempDis += squaredDis(kPoints(iii),pointStats(iii)._2)
  }
for(iii <- 0 until K){//uodate center points
    kPoints(iii) = pointStats(iii)._2
  }
```

步骤 4:反复迭代过程第二步和第三步，一直到各个簇的中心点稳定，不再发生变化，或者达到规定的迭代次数，则算法结束。

3.2 数据分类

3.2.1 朴素贝叶斯算法

1.算法概述

对于给出的待分类项，求解在此项出现的条件下各个类别出现的概率，哪个最大，就认为此待分类项属于哪个类别。

朴素贝叶斯分类的正式定义如下：

(1)设 x={a1,a2,……,am}为一个待分类项，而每个 a 为 x 的一个特征属性。

(2)有类别集合 C={y1,y2,……,yn}。

(3)计算 P(y1|x), P(y2|x),……, P(yn|x)

(4) P(yk|x)= max{P(y1|x), P(y2|x),……, P(y2|x)}，则 x 属于 yk。

在实际计算中，为了防止计算出现概率为 0 的情况，在计算先验概率和条件概率时加入一个拉普拉斯平滑指数 λ（λ>= 0)。（贝叶斯估计）

先验概率的贝叶斯估计：

$$P_\lambda(Y=c_k)=\frac{\sum_{k=1}^{N} I(y_i=c_k)+\lambda}{N+k\lambda}$$

条件概率的贝叶斯估计：

$$P_\lambda(X^{(j)}=a_{jl}|Y=c_k)=\frac{\sum_{k=1}^{N} I(X^{(j)}=a_{jl},y_i=c_k)+\lambda}{\sum_{k=1}^{N} I(y_i=c_k)+S_j\lambda}$$

2. 基于 Spark 实现过程

(1)本次实验将样本数据划分，训练样本占 0.8，测试样本占 0.2。关键代码如下：

```
val alltrainData = input.map(line => {
        val items = line.split(",")
        (items(0).toDouble, DenseVector(items.slice(1, items.length).map(_.toDouble)))
    })
val dataParts = alltrainData.randomSplit(Array(0.8, 0.2))
```

(2)实验数据为连续型变量，所以利用高斯分布来估计概率。分布函数如下：

```
def distributiveFunc(mean: Double, variance: Double)(x: Double) : Double = {
```

```
        if (variance == 0.0) {
          if (x == mean) 1.0
          else 0.0
        } else {
          1.0 / sqrt(2 * Pi * variance) * exp(- pow(x - mean, 2.0) / (2 * variance))
        }
  }
```

再通过数据求得每一列属性的平均值和方差。关键代码如下：

```
    val pji = trainData.cache().mapValues(a => {//b.mapValues("x" + _ +
"x").collect
          val aSum = a.reduce((v1 ,v2) => v1 + v2) //求总数
          val aSampleN = a.toArray.length //求总数
          val mean = aSum / (aSampleN * 1.0) //求均值
          val variance = a.map(i => { //求方差(去中心化->求和->求均值)
            ((i - mean)*(i - mean))
          }).reduce((v1 ,v2) => v1 + v2) / (aSampleN * 1.0)
          val paras = mean.toArray.zip(variance.toArray)
          paras.map(p => distributiveFunc(p._1, p._2)_) //返回(类别,[特征 1 的分
布函数, ..., 特征 n 的分布函数])
      })
```

(3)需要加入一个平滑因子来计算各类出现的概率，计算时取对数，避免后期出现连乘，容易精度丢失。关键代码如下：

```
        val allpi = trainData.cache().map{case (c, a) => {
          val p = (a.toList.length * 1.0 + lambda) / (sampleN + lambda * classN)
          (c, log2(p))
    }}
```

(4)最后预测函数就是取所有可能概率中最大的情况，即为预测值，同时计算时仍取对数，避免后期出现连乘，容易精度丢失。关键代码如下：

```
    def predict(features: DenseVector[Double]) = {
```

```
        pji.map{case (label, models) => {
          val score = models.zip(features.toArray).map{case (m, v) => {
            log2(m(v))
          }}.sum + pi(label)
          (score, label)
        }}.max
  }
```

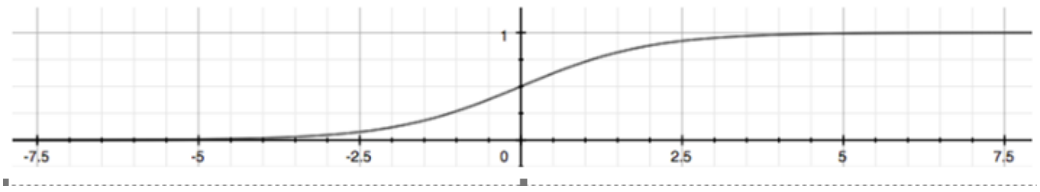
3.准确率：73%

3.2.2 逻辑回归

1.算法概述

Logistic 回归是分类方法，它利用的是 Sigmoid 函数阈值在[0,1]这个特性。Logistic 回归进行分类的主要思想是：根据现有数据对分类边界线建立回归公式，以此进行分类。其实，Logistic 本质上是一个基于条件概率的判别模型(Discriminative Model)。

(1)Sigmoid 函数：



$$h_\theta(x) = g(\theta^T x) = \frac{1}{1 + e^{-\theta^T x}}$$

z 是一个矩阵，θ 是参数列向量(要求解的)，x 是样本列向量(给定的数据集)。θ ^T 表示 θ 的转置。g(z)函数实现了任意实数到[0,1]的映射，这样我们的数据集([x0,x1,...,xn])，不管是大于 1 或者小于 0，都可以映射到[0,1]区间进行分类。

(2)梯度上升算法:

用迭代的方法来做。就像爬坡一样，一点一点逼近极值。这种寻找最佳拟合参数的方法，就是最优化算法。爬坡这个动作用数学公式表达即为：

$$x_{i+1} = x_i + \alpha \frac{\partial f(x_i)}{x_i}$$

其中，α为步长，也就是学习速率，控制更新的幅度。

梯度上升迭代公式:

$$\theta_j := \theta_j + \alpha \sum_{i=1}^{m} (y^{(i)} - h_\theta(x^{(i)}))x_j^{(i)}$$

矢量化：

$$\theta := \theta + \alpha X^T(y - g(X\theta))$$

2.基于 Spark 实现过程

(1)sigmoid 转换函数

```
def sigmoid(x:Double):Double = {
    return 1.0/(1+exp(-x))
}
```

(2)我将每条数据转为向量，方便计算。需要多次尝试设置迭代次数和梯度阈值，此次实验的迭代次数为 200，阈值为 0.001. 梯度的阀值越大梯度上升幅度越大。对样本分类就可以通过公式计算出一个概率，如果这个概率大于 0.5，就可以说样本是正样本，否则样本是负样本。

关键代码如下：

```
//梯度上升求最优参数
while(cyclenum < maxCycles){
    println(" Cycle: "+cyclenum)
    val weights1 = trainData.map(x => {
      var ab = Vector[Double]()
      val h = sigmoid(VMult(x._2,weights))
      val error = x._1 - h
      for(ii <- 0 until x._2.length){
        ab ++= Vector(alpha*error*x._2(ii))
      }
      DenseVector(ab.toArray)
```

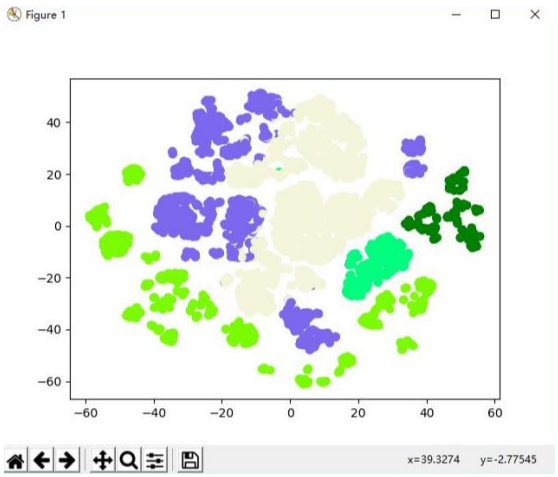
```
        }).reduce((v1 ,v2) => VAdd(v1, v2))
        weights = VAdd(weights1,weights)
        println(weights.toString())
        cyclenum += 1
    }
```

3.准确率：77%

3.3 聚类结果可视化

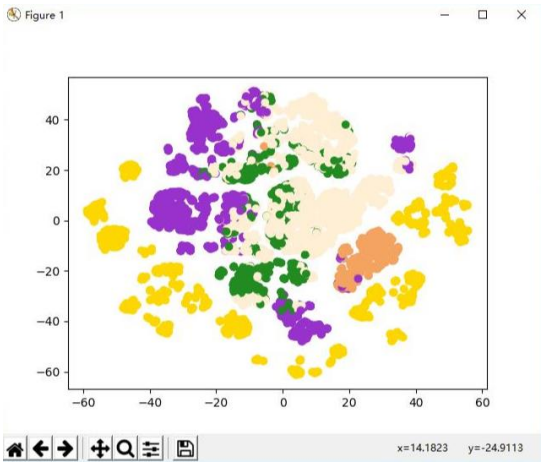
利用 View.python 代码和 usdata.pickle 文件，进行可视化。取前 10000 个点来绘图，结果如下。

K-Means：

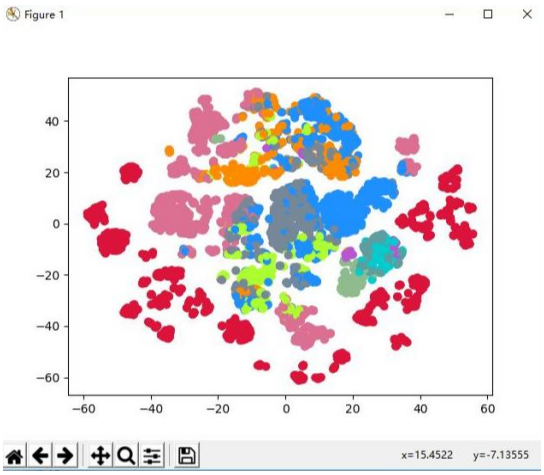


K-Means 结果图（k=5）

KMedoids:



KMedoids 结果图（k=5）



KMedoids 结果图（k=10）

K=5 效果更好一些。

四、实验心得

1. 通过本次实验，掌握了两种分类方法和聚类方法。分类方法中，逻辑回归的准确虑要高于朴素贝叶斯，分别为 77%和 73%。

2. 一开始，逻辑回归准确率只有 66%，通过实验可得，需要多次实验不同的迭代次数和梯度阈值，才能找到最高点，而不是迭代次数越多越好，迭代次数过多会导致准确率下降；也不是阈值越低越好，阈值过低，梯度上升算法中上升的会很慢，浪费时间和空间，降低效率。

3. 在聚类的可视化过程中，Spark 是并行计算的，要注意输出的结果是无序的，这一点很重要，已经踩过两次坑了。要想有序，需添加标记以记录顺序。