6

基于Pregel的图并行处理

许多实际问题可以通过简化为图问题而得以解决。 例如，基于PageRank图算法，如今我们可以搜索近五十亿个网页。 除了网页搜索外，还有其他应用场景，如社交等需要迭代图处理的领域。 在本章中，我们将学习如何使用Pregel，它一种非常适用于此类问题的一种计算模型。 Pregel最初是由Google提出的，目前已被Spark用作迭代图计算的通用编程接口。 在本章中，您将了解Pregel计算模型。并且，我们的学习目标是了解Spark中Pregel接口并能够使用它。 通过完成具体示例，您将能够使用Pregel接口开发自己的算法。

Pregel计算模型

Pregel是一个称为超步（supersteps）的迭代序列，每次迭代中顶点可以接收其邻居在前一次迭代中发送的入站消息（inbound messages），并更新其属性及其边的取值。 另外，每个顶点还在每个superstep结束时向其邻居发送消息。 通过这种专注于顶点的抽象方式，可以使得对图并行处理变得简单。 我们只需要考虑每个顶点应该接收的消息，它对入站消息进行的处理，以及下一超步时需要发送给其邻居的消息。 幸运的是，这种消息传递方法可以灵活的表示一大类图算法。 更重要的是，图算法可以借助Spark可伸缩的架构，批量和并行的方式处理消息。 这种并行计算模型可以很容易地实现大多数图并行算法。

案例-通过迭代实现社会平等

在介绍Pregel API之前，让我们用一个虚构的社交网络例子来说明一些概念，假设网络中的每个人都非常无私，并且每个人都知道他们的朋友在银行里有多少钱。但是，他们需要一种算法让他们的财富相等。这样做好还是不好，每个人看法不一定相同，但这仅仅是一个例子，并且它将有助于弄清Pregel工作机制，事实上，每个人都会将自己的钱与他朋友进行比较，并将其中的一部分钱给那些钱少的人。

通过使用Pregel，他们将通过一系列的迭代来互相送钱，以使彼此的钱相等。在这种情况下，我们可以使用浮点型（Double）作为我们算法的消息的类型。在每次迭代开始时，每个人首先会收到由朋友在前一次迭代中给的一笔钱。通过将自己当前钱与他们朋友当前的钱进行对比。他需要找出比自己钱少的那些人，然后计算应该给这些朋友多少钱。与此同时，他也决定在账户中保留多少钱。正如我们所描述的那样，每个Pregel迭代由三个连续的任务组成，这就是为什么它被称超步（superstep）的原因。

因此，他首先需要一个名为mergeMsg的函数来汇总他从比他较富有的朋友那里收到的钱。

def mergeMsg(fromA： Double, fromB： Double)： Double = fromA + fromB

其次，他还需要一个称为顶点函数来计算他在上一次超步中收到钱后的总金额：

def vprog(id: VertexId, balance： Double, credit： Double) = balance

+ credit

最后，需要一个sendMsg函数，将他的一部分钱分给他朋友：

def sendMsg(triplet： EdgeTriplet[VD, ED])： Iterator[(VertexId, A)]

从前面的函数定义可以看出，sendMsg是将边三元组而不是顶点作为输入，这样我们可以访问源节点和目标节点。 我们在下一节中将给出sendMsg的准确实现。

让我们通过考虑三个朋友之间的三角网络来进一步简化我们的例子：

scala> val nodes: RDD[(Long,Double)] = sc.parallelize(Li

st((1,10.0),(2,3.0),(3,5.0)))

nodes: RDD[(Long, Double)]

scala> val edges = sc.parallelize(List(Edge(1/2/1)/Edge(2/1/1),Edge(1/3/1 ),

Edge(3,1,1),Edge(2,3,1),Edge(3,2,1)))

edges: Edge[Int]]

scala> val graph = Graph(nodes, edges)

graph: Graph[Double,Int]

scala> graph.vertices.foreach(println)

(1/10.0)

(2/3.0)

(3/5.0)

为了简单起见，假设每个人将自己财富的百分之五分配给每个较穷的朋友。 她不需要担心一个朋友会因此收到太多的钱，因为这里不考虑自私和贪婪因素。 所以，这是我们第一次运行sendMsg函数：

def sendMsg(t： EdgeTriplet[Double/ Double])=

if (t.srcAttr <= t.dstAttr)工terator.empty

else Iterator((t.dstId,t.srcAttr \* 0.05))

这似乎是合理的。 如果一个人比她的朋友富有，她会把她百分之五的钱投给那个朋友。 反之则什么都不用给。 经过十次迭代后，我们的新图变成：

val newGraph = graph.pregel(0.0,10)(vprog, sendMsg, mergeMsg)

注意Pregel需要两个参数列表（例如，graph.pregel（list1）（list2））。 第一个参数列表包括算法开始时发送到所有顶点的初始消息以及最大迭代次数。 第二个参数列表包含用于组合，接收和计算消息的三个自定义函数。

scala> newGraph.vertices.foreach(println)

(3/10.951096875000001)

(2/10.246937500000001)

(1/10.512346875)

这有些不对劲。 开始他们总共18美元，最终结束时总和却超过30美元。 这不可能是真的！ 那么，我们做错了什么？ 为了找出错误，让我们看看Pregel的第一次迭代后发生了什么：

val newGraphl = graph.pregel(0.0,1)(vprog, sendMsg, mergeMsg)

让我们看看输出

scala> newGraphl.vertices.foreach(println)

(1,10.0)

(2.3.75)

(3,5.5)

可见，他们的总财富在一次迭代后超过18美元。 这是因为当一个人把钱给他的朋友时，这笔款项不会从他的账户中扣除。 我们可以通过同时发送消息给收到钱的人以及给钱的人来解决这个问题。 所以，如果A把X美元给B，我们应该将X美元发送给B，同时将-X美元发送给A：

def sendMsg(t： EdgeTriplet[Double/ Double])=

if (t.srcAttr <= t.dstAttr)工terator.empty

else Iterator((t.dstId, t.srcAttr \* 0.05),

(t.srcId, - t.srcAttr \* 0.05))

val afterOnelter = graph.pregel(0.0, 1)(vprog, sendMsg, mergeMsg)

让我们看看输出

scala> afterOneIter.vertices.foreach(println)

(1,9.0)

(2.3.75)

(3,5.25)

这回可以看出结果与预期是一致的。

此外，如果我们增加最大迭代次数呢？ 让我们看看会发生什么：

scala> afterTenIters.vertices.foreach(println)

(1,5.999611965064453)

(2,6.37018749852539)

(3,5.630200536410156)

scala> afterHundredIters.vertices.foreach(println)

(1,6.206716647163644)

(2,6.207038273723298)

(3,5.586245079113054)

即使经过100次迭代，我们可以看到每个人帐户余额也不会收敛到6美元的理想值，而是围绕它进行波动。 这对于这个简单化的例子来说，是意料之中的事。

GraphX中的Pregel API

现在，让我们调用Pregel的函数式编程接口。 下面是它的定义：

class GraphOps[VD, ED] {

def pregel[A]

(initialMsg： A,

maxIter: Int = Int.MaxValue,

activeDir: EdgeDirection = EdgeDirection.Out)

(vprog: (VertexId, VD, A) => VD,

sendMsg: EdgeTriplet[VD, ED] => Iterator[(VertexId, A)],

mergeMsg: (A, A) => A)

:Graph[VD, ED]

}

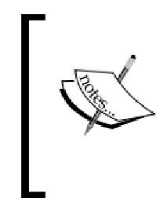
在属性图上调用pregel，会返回一个具有相同类型和结构的新图。 在从一个超集到下一个超集的过程中，虽然边保持不变，但顶点的属性可能会发生改变。 Pregel需要以下两个参数列表。 第一个列表包含：

•自定义类型为A的初始消息，当算法启动时，会将它发送给每个顶点。

•最大迭代次数

•发送消息的方向

****

****Pregel算法在没有消息要发送时，或达到指定的最大迭代次数时终止。 在实际运行算法时，有一点非常重要：始终限制迭代的最大次数，特别是当算法不能保证收敛时。

如果没有指定活跃（active）的边的方向，Pregel假定只沿着每个顶点的输出边发送消息。此外，如果一个顶点在前一个超步中没有收到消息，那么在当前超步结束前将不会有消息沿着它的出边发送

另外，第二个参数列表必须包含三个函数：

vprog: (VertexId, VD, A) => VD

该顶点函数会在前一次迭代后更新所有收到消息的点的属性

mergeMsg: (A, A) => A)

该函数为每个顶点合并它收到全部的消息

sendMsg: EdgeTriplet[VD, ED] => Iterator[(VertexId, A)]

该函数通过EdgeTriplet生成将要发送给起始节点和目标节点的消息

基于标签传播的社区发现算法

在本节中，我们将使用Pregel接口实现社区检测算法。**标签传播算法（LPA）**是用于检测图中社区的一种简单快速的方法。通过标签传播发现社区，要求每个节点在其社区内至少具有与其他每个社区一样多的邻居。

让我们先简要概述下LPA的工作原理。首先，每个节点最初被赋予其顶点ID作为其标签。在随后的迭代中，每个节点根据其邻居的标签确定其社区。具体而言，该节点选择加入其邻居所属的最大数目的社区。如果最大数目的社区并列多个时，则随机选取其中一个社区。当我们在图中以这种方式传播标签时，大多数标签都会消失，而其余标签会定义社区。理想情况下，当网络中没有节点更改其标签时，此迭代算法会收敛。最终，具有相同标签的节点被组合在一起作为一个社区。

通过在Pregel中实现此算法，我们想要获得一个图，其中顶点属性是社区中的标签。 因此，我们首先通过将每个顶点的标签设置为其id来初始化LPA图：

val lpaGraph = graph.mapVertices { case (vid, \_) => vid }

接下来，我们定义消息的类型为Map [Label，Long]，它将社区标签与具有此标签的邻居数关联起来。 将发送给每个节点的初始消息设置为空：

type Label = VertexId

val initialMessage = Map[Label, Long]()

在此之上，我们定义一个sendMsg函数，用它将每个节点的当前的标签，通知其邻居。 对于每个三元组，源节点将接收目标节点的标签，反之亦然：

def sendMsg(e: EdgeTriplet[Label, ED]):工terator[(VertexId,

Map[Label, Long])]=

Iterator((e.srcId, Map(e.dstAttr -> 1L)), (e.dstId,

Map(e.srcAttr -> 1L)))

在收到来自其邻居的消息后，节点将其社区标签改为当前其大多数邻居所属的标签。 因此，使用以下顶点函数来对每个节点执行此操作：

def vprog(vid： VertexId, attr: Long, message： Map[Label, Long])： VertexId = if (message.isEmpty) attr else message.maxBy(\_.\_2).\_1

上一个函数在每次迭代中返回某顶点的大多数邻居当前所属的社区的标签（即Vertexid属性）。

我们还需要一个mergeMsg函数来将节点从其邻居收到的所有消息合并为一个。 如果两条消息都包含相同的标签，我们只需统计该标签对应邻居数量即可：

def mergeMsg(count1： Map[Label, Long],

： Map[VertexId, Long] = {

(count1.keySet ++ count2.keySet).map { i =>

val count1Val = count1.getOrElse(i, 0L)

val count2Val = count2.getOrElse(I, 0L)

i -> (count1Val + count2Val)

}.toMap

}

最后，我们可以运行LPA算法，就像我们之前用pregel方法来平衡社会财富一样：

****lpaGraph.pregel(initialMessage, 50)(vprog, sendMsg, mergeMsg)

C:\Users\ADMINI~1.DES\AppData\Local\Temp\FineReader12.00\media\image29.jpegLPA的主要优点是其简单和时间效率高。 实际上，收敛所需的迭代次数与图的大小无关，并且每次迭代的时间复杂度都是线性的。但时标签传播算法可能不一定会收敛，并且也可能导致无意义的结果，例如每个节点被识别为单个社区。 实际上，该算法可能会出现二分图或几乎二分结构的图形。

基于Pregel的PageRank实现

我们已经知道GraphX有一个PageRank API。 下面，我们看看如何使用Pregel轻松实现这个著名的网页搜索算法。 由于我们在前一章已经解释了PageRank是如何工作的，现在我们将简单地解释它的Pregel实现：

首先，我们需要初始化图，将每个边属性值设置为1除以出度，每个顶点属性设置为1.0：

val rankGraph: Graph[(Double, Double), Double]=

// Associate the degree with each vertex graph.outerJoinVertices(graph.outDegrees) {

(vid, vdata, deg) => deg.getOrElse(O)

}.mapTriplets( e => 1.0 / e.srcAttr )

.mapVertices( (id, attr) => (0.0, 0.0))

根据Pregel抽象要求，我们定义实现PageRank所需的三个函数。 首先，我们定义顶点函数如下：

val resetProb = 0.15

def vProg(id： VertexId, attr： (Double, Double), msgSum： Double)：

(Double, Double) = {

val (oldPR, lastDelta) = attr

val newPR = oldPR + (1.0 - resetProb) \* msgSum (newPR, newPR - oldPR)

}

接下来是创建消息的函数：

val tol = 0.001

def sendMessage(edge: EdgeTriplet[(Double, Double), Double]) = {

if (edge.srcAttr.\_2 > tol) {

Iterator((edge.dstId, edge.srcAttr.\_2 \* edge.attr))

} else {

Iterator.empty

}

}

第三个函数为mergeMsg，用于计算排名：

def mergeMsg(a: Double, b: Double): Double = a + b

然后我们将得到如下的节点排名：

rankGraph.pregel(initialMessage, activeDirection =

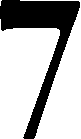
EdgeDirection.Out)

(vProg, sendMsg, mergeMsg)

.mapVertices((vid, attr) => attr.\_1)

总结

简而言之，Pregel提供了在大规模图上编写自定义迭代和并行算法的简化通用接口。 在本章中，我们已经看到了如何使用这个简单的抽象来实现不同的图算法。 在下一章中，我们将看到如何使用Spark的MLlib和GraphX来解决一些图数据相关的机器学习问题。



**学习图结构**

在本章中，我们将向您展示如何利用Spark从图中学习有趣的结构。 通常，一个人选择感兴趣的问题进行研究，那么他将更高效的学习，更能发现数据中的联系。 最常见的学习问题有回归，分类，排序和聚类。 在本书中，我们将重点关注聚类。 具体而言，我们将重点关注图数据，并应用聚类来发现图中的社区。 本章结构如下：首先，我们将介绍谱聚类的概念。 然后，我们将学习一个具体的方法，通过它我们可以用Spark对图进行聚类分析。 最后，我们将利用这些技术对音乐和歌曲播放列表数据集进行分析。 这个例子也可以看作是对我们在前面章节所学知识掌握情况的检验。 本章中我们将综合运用它们。

针对图进行聚类

聚类是根据定义的相似性度量，将给定个体（如对象或人）划分到不同子集的一类问题。 同一类中的个体非常相似，并且与其他类中的所有个体都不相同。 聚类通常使用无监督方法。 换句话说，它对未标记的个体的属性或特征数据进行操作。 此外，聚类方法可以大致分为参数方法和非参数方法。 参数化方法运用概率模型，如**高斯混合模型（GMM）和隐狄利克雷分布模型（LDA）**。 非参数模型根据数据本身来推断聚类的结构，如k均值和谱聚类。 以上所有方法都可以在Spark的MLlib库中找到。

为什么聚类与图处理是紧密相关的呢？ 主要有两个原因。 首先，聚类对于检测图中 “社区”非常有用。 这些社区本质上是拥有相似特征节点群。 虽然两个节点没有明确的连线，但是通过学习属性数据，聚类仍可以发现它们的相似之处。 例如，社交网站会使用这些信息来给你推荐你可能认识的人，或者你可能想认识的人。 此外，聚类可以帮助在一些高度连接的网络中发现有趣的结构。 第二个原因是我们在这里看到的聚类方法是基于图处理的。 特别是，我们将注意力集中在**快速迭代聚类（PIC）**上，它是一种简单快速的谱聚类方法。

谱聚类

如前所述，聚类的目的是将数据点分成几类，使得同一个类中的点非常相似，不同类中的点彼此不相似。 “相似性图”是表示数据点相似性的好方法。每个点成为相似度图中的一个节点，而每个边具的属性表示的连接节点的“相似性度量”。因此，聚类问题可以简化为，对图进行分组，使得不同组之间的边具有非常低的权重，并且组内的边具有高权重。为此，我们使用谱聚类技术，该方法可以将高维相似性图压缩为低维。为了讲解的通俗性，我们不会罗列数学证明过程。但是，可以通过阅读一些很好的教程来了解技术细节，例如http://arxiv.org/abs/0711.0189上提供的教程。

快速迭代聚类

快速迭代聚类（PIC）方法是一种高效且可扩展的谱聚类方法。 它在MLlib库中详细的定义，具体链接为： http://spark.apache.[org/docs/latest/mllib-clustering.html#power-iteration-clustering-pic](http://spark.apache.org/docs/latest/mllib-clustering.html%23power-iteration-clustering-pic). 它的实现用到了在Spark中GraphX的API和缓存优化技术。 以下是PIC聚类方法的API程序：

class PowerIterationClustering {

//运行PIC算法

def run(similarities: RDD[(Long, Long, Double)]):

PowerIterationClusteringModel

//设置初始化模式，是"random" 或者 "degree"

def setInitializationMode(mode: String)： PowerIterationClustering.this.type

//设置聚类的簇数

def setK(k: Int): PowerIterationClustering.this.type

//设置快速迭代聚类的最大迭代次数

def setMaxIterati〇ns(maxIterati〇ns: Int): PowerIterationClustering.this.type

}

若要对图进行PIC聚类，我们需要遵循以下五个步骤：

1.首先，将数据加载到Spark图属性中。

2.其次，提取节点的特征。

3.第三，定义两个节点之间的相似性度量。

4.然后，使用相似性度量**，**创建初始图的关联矩阵。

5.最后，在关联矩阵上运行k-means聚类。

步骤1和2可以使用我们在*第2章构建和探索图*中学习的**图构建方法**完成。 第3步只需要我们定义一个函数来计算两个节点的相似程度。 相似性度量的选择取决于节点的特征和面对的具体问题。 尽管如此，我们可以选择**常规方法**。 例如，如果节点特征是二元向量，我们可以使用Jaccard相似度。 又如果当节点特征是实向量时，可以使用高斯核函数。 这些不是唯一的选择，我们也可以自定义度量方法。

在步骤4中，关联矩阵的相似度应该由元组（i，j，*sim*）的RDD表示。 相似度sim必须是一个非负数。 对于任何非零相似的边（i j），输入中应该是（i，j，sim）或（j，i，sim）。 由于关联矩阵必须是对称的，如果数据中已知（i，j，sim），则假定（j，i，sim）也已知，反之亦然。 并且，对i = j的元组可以直接忽略。

最后一步由两个步骤组成。 首先，我们从相似矩阵中创建PowerlterationClusteringModel，然后对其执行k-means聚类。 在运行聚类模型之前，我们还必须选择两个参数：

•k均值聚类的最大迭代次数

•聚类的最大簇数，K

以下代码展示了PIC应用程序的大致结构

import org.apache.spark.mllib.clustering.PowerlterationClustering

// 基于初始图定义每对节点的相似度

RDD[(Long, Long, Double)]=...

//构建PIC聚类模型

val pic = new PowerIteartionClustering()

.setK(maxClusterNumber)

.setMaxIterations(maxIterations)

//运行PIC聚类模型

val clusteringResult: RDD[Assignment]=

pic.run(similarities).assignments

clusteringResult.collect().foreach { a =>

println(s"${a.id} -> ${a.cluster}")

}

PIC方法返回一个特定的RDD，它是由节点ID（VertexId），以及该节点ID类型Int及其所属群组成的元组。

应用-歌迷社区发现

根据每首歌曲的标签，我们尝试将之前的图聚类方法应用于音乐歌曲的聚类上。 另外，歌曲播放列表数据集也可以用来聚类找出那些在许多列表中经常播放的歌曲。 我们将要使用的数据集可以在<http://www.cs.cornell.edu/~shuochen/lme/data_page.html>下载。数据集由以下文件组成：

•train.txt：该文件通过使用整数ID来表示歌曲，其中包含播放列表数据

•tags.txt：该文件包使用整数ID表示歌曲对应社交标签

•song\_hash.txt：该文件将歌曲ID映射到对应的标题和艺术家

•tag\_hash.txt：这个将标签ID映射到其名称

每个文件都有一个特定的格式，例如：

**•播放列表数据的格式：**数据文件的第一行由用空格分隔的歌曲的ID（不是整数ID，而是用于标识歌曲的其他来源的ID）组成。这里我们不需要第一行，因此可以跳过。第二行包含文件中每首歌曲的出现次数，也由空格分隔。从第三行开始的是播放列表，每个歌曲由该文件中的整数ID表示（从0到歌曲总数减1）。请注意，在播放列表数据文件中，每行以空格结束。

**•标签数据的格式：**标签数据文件与播放列表数据中的歌曲总数具有相同的行数。每行由歌曲标签的ID组成，用整数表示，并用空格分隔。如果一首歌没有标签，用＃代替。请注意，对于标记文件，每行末尾没有空格。

**•歌曲映射文件的格式：**每行对应一首歌曲，格式为Integer\_ID\tTitle\tArtist\n。

**•标签映射文件的格式：**每行对应一首歌曲，格式为Integer\_ID\Name\n。

首先，让我们按照前面的五个步骤，通过标签对歌曲进行聚类。

第一步-将数据导入成Spark属性图

我们定义一个歌曲类。 每首歌的属性都有一个标题，一个歌手名字和一组标签：

scala> case class Song(title: String, artist: String, tags: Set[String])

{

override def toString: String = title + ", " + artist

}

现在，我们将歌曲导入RDD [（VertexId，Song）]，并用一组空白的标签初始化每首歌曲：

scala> var songs: RDD[(VertexId, Song)]=

sc.textFile("./data/song\_hash.txt").

map {line =>

val row = line split '\t'

val vid = row(0).toLong

val song = Song(row(1), row(2), Set.empty)

(vid, song)

}

songs: RDD[(VertexId, Song)]

然后，我们可以创建一个属性图，其节点是歌曲。 开始时不需要将任何边添加到图中，只将空RDD传递给Graph.apply方法：

scala> val graphFromSongs: Graph[Song, Int] = {

val zeroEdge: RDD[Edge[Int]] = sc.parallelize(Nil)

Graph(songs, zeroEdge)

}

graphFromSongs: Graph[Song,Int]

scala> graphFromSongs.vertices.take(5).foreach(println)

(1084,Song(Tequila Sunrise,Fiji,Set()))

(1410,Song(The Sweetest Taboo,Sade,Set()))

(3066,Song(Bow Chicka Wow Wow,Mike Posner,Set()))

(1894,Song(Love Your Love The Most,Eric Church,Set()))

(466,Song(Stupify,Disturbed,Set()))

第二步-抽取每个节点的特征

现在，让我们将数据集tags.txt文件的标签加入节点。 为此，我们首先需要创建RDD [（Vertexid，Set [string]）]，然后我们将加入graphFromSong

scala> val tagIter: Iterator[(VertexId, Set[String])]=

Source.fromFile(H./data/tags.txtH).getLines.zipWithIndex.

map {

x =>

val tags = x.\_1 split ' '

(x.\_2.toLong, tags.toSet)

}

tagIter: Iterator[(VertexId, Set[String])] = non-empty iterator

scala> val tagRDD = sc.parallelize(tagIter.toSeq)

tagRDD: RDD[(VertexId, Set[String])]

现在，我们仅仅通过tagRDD在歌曲ID与标签ID之间建立了映射关系。

scala> tagRDD.take(3).foreach(println)

(0,Set(115, 173))

(1,Set(62, 88, 110, 90, 123, 155, 173, 14, 190, 214, 115, 27))

(2,Set(115, 173))

我们想要的是从给定标签ID的tag\_hash.txt中提取标签名称。 我们现在可以在graphFromSongs上调用joinVertices，并将标签tagRDD的RDD与提取标签的函数一起传递。 请注意，在名为tags.txt文件中，如果歌曲ID旁边有＃标签，意味着没有标签与该歌曲相关联。 在这种情况下，我们只需用用空标签返回给歌曲。 否则，我们将这组标签添加到歌曲中：

scala> val songsNtags = graphFromSongs.joinVertices(tagRDD){

(id, s, ks) => ks.toList match {

case List("#") => s

case \_ => {

val tags: Map[Int, String] =

Source.fromFile("./data/tag\_hash.txt").getLines().

map {

line =>

val row = line split ", "

row(0).toInt -> row(1)

}.toMap

val songTags = ks.map(\_.toInt) flatMap (tags get)

Song(s.title, s.artist, songTags.toSet)

}

}

}

songsNtags: Graph[Song,Int]

scala> songsNtags.vertices.take(3).foreach(println)

(1084,Tequila Sunrise, Fiji)

(1410,The Sweetest Taboo, Sade)

(3066,Bow Chicka Wow Wow, Mike Posner)

第三步-定义两节点之间的相似性度量

由于我们希望通过他们的社交标签对歌曲进行聚类，因此衡量两首歌曲之间相似性的一种自然方式就是Jaccard标距（metric）。 简而言之，它是两首歌曲之间共同标签数量与标签总数之比。 如果没有共同标签，我们假设它们的相似度为零：

def similarity(one： Song, other： Song)：Double = {

val numCommonTags = (one.tags intersect other.tags).size

val numTotalTags = (one.tags union other.tags).size

if (numTotalTags > 0)

numCommonTags.toDouble / numTotalTags.toDouble

else 0.0

}

第4步-构建关联矩阵

现在，我们需要计算数据中每对歌曲之间的相似度。 如果有1000首歌曲，我们将不得不计算和存储100万个相似度值。 如果我们有一百万首歌曲会怎样？ 显然，计算每一对之间的相似性将是低效的。 相反，我们可以优先考虑具有较高相似性分数的歌曲。 在这之后，我们将类似的歌曲聚类在一起。 因此，我们用以下函数来过滤节点：

scala> def quiteSimilar(one: Song, other: Song, threshold: Double):

Boolean = {

val commonTags = one.tags intersect other.tags

val combinedTags = one.tags union other.tags commonTags.size > combinedTags.size \* threshold

}

quiteSimilar: (one: Song, other: Song, threshold: Double)Boolean

下一个函数用来删除图形数据中的重复歌曲：

def differentSong(one: Song, other: Song): Boolean =

one.title != other.title || one.artist != other.artist

有了这两个函数，我们现在可以创建RDD [Edge [Double]]，它将包含较为相似节点之间的相似性度量：

//首先，获取歌曲标签

songs = songsNtags.vertices

//然后，计算每对歌曲之间的相似度

//相似度分数大于0.7

similarConnections： RDD[Edge[Double]] = { val ss = songs cartesian songs val similarSongs = ss filter { p => p.\_1.\_1 != p.\_2.\_1 &&

similarByTags(p.\_1.\_2, p.\_2.\_2, 0.7) &&

differentSong(p.\_1.\_2, p.\_2.\_2)

}

similarSongs map {

p => {

val jacIdx = similarity(p.\_1.\_2, p.\_2.\_2) Edge(p.\_1.\_1, p.\_2.\_1, jacIdx)

}

}

}

通过上述计算，我们只需要存储1,506个相似性分数而不是100万个：

scala> similarConnections.count

res8: Long = 1506

scala> songs.count

res9: Long = 3168

scala> 3168 \* 3168

res10: Int = 10036224

经过上述处理，我们可以创建的相似图：

scala> val similarByTagsGraph = Graph(songs, similarConnections)

有些歌曲只有少数标签，所以我们可以先将他们过滤掉。

val similarHighTagsGraph = similarByTagsGraph.subgraph(vpred =

(id: VertexId, attr: Song) => attr.tags.size > 5)

让我们检查下输出：

scala> similarHighTagsGraph.vertices.count

res12: Long = 2144

scala> similarHighTagsGraph.edges.count

res13: Long = 126

进一步的

scala> similarHighTagsGraph.triplets.take(6).foreach(t => println(t.

srcAttr + "~~~ " + t.dstAttr + " => " + t.attr))

Fancy (w\/ T.I. & Swizz Beatz), Drake ~~~Any Girl (w\/ Lloyd), Lloyd

Banks => 0.8571428571428571

Roll With It, Easton Corbin ~~~ You Lie, The Band Perry =>

0.7142857142857143

Any Girl (w\/ Lloyd), Lloyd Banks ~~~ Fancy (w\/ T.I. & Swizz Beatz),

Drake => 0.8571428571428571

Any Girl (w\/ Lloyd), Lloyd Banks ~~~ I'm Going In (w\/ Young Jeezy & Lil

Wayne), Drake => 0.7142857142857143

Everything Falls, Fee ~~~ Needful Hands, Jars Of Clay =>

0.7142857142857143

Bring The Rain, MercyMe ~~~ Needful Hands, Jars Of Clay => 0.75

我们可以发现Drake 的Fancy 与Lloyd Banks的Any Girl相似度很高。这是必然的，因为他们都是说唱类歌曲（rap songs.）

最后，我们创建一个相关矩阵[（LOOng，Long，Double）]的RDD，这是运行PIC算法所需的：

val similarities: RDD[(Long,Long,Double)]=

similarHighTagsGraph.triplets.map{t => (t.srcId, t.dstId, t.attr)}

第五步-在关联矩阵上运行K-means（聚类）

我们选择聚类的簇数K = 7

scala> val similarities: RDD[(Long,Long,Double)] = similarHighTagsGraph.

triplets.map{t => (t.srcId, t.dstId, t.attr)}

scala> val pic = new PowerIterationClustering().setK(15).

setMaxIterations(20)

pic: org.apache.spark.mllib.clustering.PowerIterationClustering

scala> val clusteringModel = pic.run(similarities) clusteringModel: org.apache.spark.mllib.clustering. PowerIterationClusteringModel

scala> clusteringModel.assignments.foreach { a =>

| println(s”${a.id} -> ${a.cluster}”)

| }

327 -> 0

715 -> 0

3063 -> 2

2879 -> 2

1623 -> 0

3003 -> 0

2539 -> 0

2283 -> 0

2163 -> 0

2979 -> 0

2615 -> 5

2147 -> 1

2667 -> 3

2531 -> 0

2149 -> 4

查看聚类结果，显然，我们无法从这些数字看出任何东西。 因此，我们来探索这些类，看看每个类中歌曲的共同标签。

首先，我们使用聚类结果创建一个图，其节点包含 歌曲以及其类别ID。 为此，我们使用两次VertexRDD集合的innerJoin方法。 首先，我们将聚类列别加入到歌曲图中。 由于innerJoin可以改变顶点的属性类型，所以初始图形顶点属性类型是什么并不重要。 为了简单起见，我们将每个顶点属性值初始化为0.0. innerJoin的第二个作用是将VertexRDD歌曲集合加入到第一个运行的结果中：

val clustering： RDD[(Long, Int)]=

clusteringModel.assignments.map(a => (a.id, a.cluster)) val graph:

VertexRDD[Double]=

Graph.fromEdges[Double/Double](similarities.map(t => Edge(t.\_1,t.\_2,t.\_3)), 0.0).vertices

val clusteredSongs: VertexRDD[(Song, Int)]= graph.innerJoin(clustering){ (id, \_, cluster) => cluster }.innerJoin(songs){ (id, cluster, s) => (s, cluster)}

因此，我们获得了一个新的VertexRDD集合clusteredSongs，其中包含歌曲及其类别ID：

scala> clusteredSongs.first

res25: (VertexId, (Song, Int)) = (2372,(Hold My Heart, Tenth Avenue

North,7))

我们可以将相似性度分数加入属性图中：

scala> val clusterNScoreGraph = Graph(clusteredSongs, similarities.map(t

=> Edge(t.\_1,t.\_2,t.\_3)))

clusterNScoreGraph: Graph[(Song, Int),Double]

scala> clusterNScoreGraph.triplets.first

res37: EdgeTriplet[(Song, Int),Double] = ((38,(Fancy (w\/ T.I.

& Swizz Beatz), Drake,2)),(1976,(Any Girl (w\/ Lloyd), Lloyd Banks,2)),

0.8571428571428571)

由于并非所有相邻歌曲都对应于同一个群集，因此我们可以去掉那些彼此相连但却属于不同类的两首歌曲节点之间的边。 换句话说，这些歌有一些相似之处，但它们并不真正属于一类：

scala> val clusteredSongGraph = clusterNScoreGraph.subgraph(epred = t => t.srcAttr.\_2 == t.dstAttr.\_2)

clusteredSongGraph: Graph[(Song, Int),Double] scala> clusteredSongGraph.edges.count res5: Long = 50

接下来，对于剩余的边，我们将彼此连接歌曲的共同类别标签替换连接它们的边的属性值。 借助mapTriplets方法可以很容易地完成：

val clusteredTagsGraph = clusteredSongGraph.mapTriplets(t => t.srcAttr.\_1.tags intersect t.dstAttr.\_1.tags)

让我们看看我们得到了什么：

scala> clusteredTagsGraph.triplets.take(3).foreach(println)

((482,(Roll With It, Easton Corbin,0)),(2866,(You Lie, The Band Perry,0)),

Set(new country, modern country, country, great song, my favorite))

((1976,(Any Girl (w\/ Lloyd), Lloyd Banks,6)),(2470,(I'm Going In (w\/ Young Jeezy & Lil Wayne), Drake,6)),Set(rap, wdzh-fm, wjlb-fm, whtd-fm, wkqi-fm))

((2364,(While I'm Waiting, John Waller,0)),(2372,(Hold My Heart,

Tenth Avenue North,0)),Set(worship, favorite, christian, contemporary christian, christian rock))

现在，如果我们想要近似地找到每个群集中的共同标签，我们可以使用Pregel接口来做到这一点。 请记住，Spark中的Pregel实现只允许在相邻节点之间传递消息。 但是，我们的聚类标签图存在一些不直接相接，但仍属于同一个集群的节点，（可能通过其他节点连接）。 因此，Pregel算子将无法在每个群集中找到绝对共同标签，但它仍然有助于发现一些模式：

val commonTagsByCluster =

clusteredTagsGraph.pregel[Set[String]](initialMsg = Set.empty, maxIterations = 10){

(id, sc, m) => sc,

t => Iterator((t.srcId, t.srcAttr.\_1.tags intersect t.dstAttr.\_1.tags),

(t.dstId, t.srcAttr.\_1.tags intersect t.dstAttr.\_1.tags)),

(s1, s2) => si intersect s2

}

查看结果，我们可以找到一些聚类。 像类＃1由敬拜诗歌组成：

scala> commonTagsByCluster.triplets.filter(\_.srcAttr.\_2 == 1).foreach(

t => println(t.srcAttr.\_1 + " => " + t.attr))

Lead Me, Sanctus Real => Set(rock, worship, christian, contemporary christian, christian rock, happy)

Needful Hands, Jars Of Clay => Set(rock, worship, christian, contemporary christian, christian rock, favorites)

Revelation, Third Day => Set(rock, worship, christian, contemporary christian, christian rock, favorites)

Give You Glory, Jeremy Camp => Set(rock, worship, christian, contemporary christian, christian rock, favorites)

Revelation, Third Day => Set(rock, worship, christian, contemporary christian, christian rock, favorites)

Cry Holy, Sonicflood => Set(rock, worship, christian, contemporary christian, christian rock)

类#2是乡村音乐和R&B歌曲

scala> commonTagsByCluster.triplets.filter(\_.srcAttr.\_2 == 2).foreach(t => println(t.srcAttr.\_1 + " => " + t.attr))

This Is Country Music, Brad Paisley => Set(beautiful, new country, memories, country, great song, my favorite)

Bring It Back, Travis Porter => Set(wdzh-fm, hip hop, 2010s, wjlb-fm, whtd-fm, energetic)

Anything Like Me, Brad Paisley => Set(beautiful, new country, memories, country, great song, my favorite)

The Boys Of Fall, Kenny Chesney => Set(beautiful, new country, memories, country, great song, my favorite)

Anything Like Me, Brad Paisley => Set(beautiful, new country, memories, country, great song, my favorite)

Grove St. Party (w\/ Kebo Gotti), Waka Flocka Flame => Set(wdzh-fm, hip hop, 2010s, wjlb-fm, whtd-fm, energetic, wkqi-fm)

The Boys Of Fall, Kenny Chesney => Set(beautiful, new country, memories, country, great song, my favorite)

Make It Rain, Travis Porter => Set(wdzh-fm, hip hop, 2010s, wjlb-fm, whtd-fm, energetic)

Grove St. Party (w\/ Kebo Gotti), Waka Flocka Flame => Set(wdzh-fm, hip hop, 2010s, wjlb-fm, whtd-fm, energetic)

Love Faces, Trey Songz => Set(male vocalists, r&b, 2010s, wjlb-fm, rnb, whtd-fm)

Words, Bobby V => Set(male vocalists, r&b, 2010s, wjlb-fm, rnb, whtd-fm) Cupid, Lloyd => Set(male vocalists, r&b, 2010s, wjlb-fm, rnb, whtd-fm)

如果你看看其他类，类#6是嘻哈。 然而，最后一个集群＃0不太明显。 这是不在以上分类中的歌曲的一个组合。 这种分类结果仅仅是因为标签数据的不完善导致，而不是由于PIC聚类方法本身问题。

练习-对播放列表进行协同聚类

通过标签对音乐歌曲进行聚类并不是非常有效。 想象一下你自己必须标记你听的每一首歌。 我们可以使用共享的播放列表来推进行聚类，而不是使用显式特征（如标签）。 播放列表是组织音乐的一种更普遍的方式。 现在，受这个启发，如果两首歌曲在许多列表中重复出现，那么它们的相似度很大可能性会高于那些没有共同出现在一些歌单中的两首歌。 我将把剩下的作为练习。 只要按照我们之前采取的相同的5个步骤即可。

总结

在本章中，我们研究了如何解决大规模图的聚类问题。 为此，我们介绍了快速迭代聚类(Power Iteration Clustering)方法，并展示了如何将它应用到有社会性标签（social tags）的歌曲聚类问题上。 通过歌曲聚类的例子，我们也回顾了我们在本书中学到的主要图构建和处理技术。 您现在已具备了具备了运用Spark的图处理方法来解决更有趣的问题的能力了。