08｜Embedding实战：如何使用Spark生成Item2vec和Graph Embedding？

你好，我是王喆。

前面两节课，我们一起学习了从 Item2vec 到 Graph Embedding 的几种经典 Embedding 方法。在打好了理论基础之后，这节课就让我们从理论走向实践，看看到底**如何基于 Spark 训练得到物品的 Embedding 向量**。

通过特征工程部分的实践，我想你已经对 Spark 这个分布式计算平台有了初步的认识。其实除了一些基本的特征处理方法，在 Spark 的机器学习包 Spark MLlib 中，还包含了大量成熟的机器学习模型，这其中就包括我们讲过的 Word2vec 模型。基于此，这节课我们会在 Spark 平台上，完成 **Item2vec 和基于 Deep Walk 的 Graph Embedding** 的训练。

对其他机器学习平台有所了解的同学可能会问，TensorFlow、PyTorch 都有很强大的深度学习工具包，我们能不能利用这些平台进行 Embedding 训练呢？当然是可以的，我们也会在之后的课程中介绍 TensorFlow 并用它实现很多深度学习推荐模型。但是 Spark 作为一个原生的分布式计算平台，在处理大数据方面还是比 TensorFlow 等深度学习平台更具有优势，而且业界的很多公司仍然在使用 Spark 训练一些结构比较简单的机器学习模型，再加上我们已经用 Spark 进行了特征工程的处理，所以，这节课我们继续使用 Spark 来完成 Embedding 的实践。

首先，我们来看看怎么完成 Item2vec 的训练。

Item2vec：序列数据的处理

我们知道，Item2vec 是基于自然语言处理模型 Word2vec 提出的，所以 Item2vec 要处理的是类似文本句子，观影序列之类的序列数据。那在真正开始 Item2vec 的训练之前，我们还要先为它准备好训练用的序列数据。在 movieLens 数据集中，有一张叫 rating（评分）的数据表，里面包含了用户对看过电影的评分和评分的时间。既然时间和评分历史都有了，我们要用的观影序列自然就可以通过处理 rating 表得到啦。

userId	movieId	rating	timestamp
1	2	3.5	1112486027
1	29	3.5	1112484676
1	32	3.5	1112484819
1	47	3.5	1112484727
1	50	3.5	1112484580
1	112	3.5	1094785740
1	151	4.0	1094785734
1	223	4.0	1112485573
1	253	4.0	1112484940
1	260	4.0	1112484826
1	293	4.0	1112484703
1	296	4.0	1112484767
1	318	4.0	1112484798
1	337	3.5	1094785709
1	367	3.5	1112485980

图1 movieLens数据集中的rating评分表

不过，在使用观影序列编码之前，我们还要再明确两个问题。一是 movieLens 这个 rating 表本质上只是一个评分的表，不是真正的“观影序列”。但对用户来说，当然只有看过这部电影才能够评价它，所以我们几乎可以把评分序列当作是观影序列。二是我们是应该把所有电影都放到序列中，还是只放那些打分比较高的呢？

这里，我是建议对评分做一个过滤，只放用户打分比较高的电影。为什么这么做呢？我们要思考一下 Item2vec 这个模型本质上是要学习什么。我们是希望 item2vec 能够学习到物品之间的近似性。既然这样，我们当然是希望评分好的电影靠近一些，评分差的电影和评分好的电影不要在序列中结对出现。

好，那到这里我们明确了样本处理的思路，就是对一个用户来说，我们先过滤掉他评分低的电影，再把他评论过的电影按照时间戳排序。这样，我们就得到了一个用户的观影序列，所有用户的观影序列就组成了 Item2vec 的训练样本集。

那这个过程究竟该怎么在 Spark 上实现呢？其实很简单，我们只需要明白这 5 个关键步骤就可以实现了：

1. 读取 ratings 原始数据到 Spark 平台。
2. 用 where 语句过滤评分低的评分记录。
3. 用 groupBy userId 操作聚合每个用户的评分记录，DataFrame 中每条记录是一个用户的评分序列。
4. 定义一个自定义操作 sortUdf，用它实现每个用户的评分记录按照时间戳进行排序。
5. 把每个用户的评分记录处理成一个字符串的形式，供后续训练过程使用。

具体的实现过程，我还是建议你来参考我下面给出的代码，重要的地方我也都加上了注释，方便你来理解。

```
def processItemSequence(sparkSession: SparkSession
  //设定rating数据的路径并用spark载入数据
  val ratingsResourcesPath = this.getClass.getReso
  val ratingSamples = sparkSession.read.format("cs
```

```
  //实现一个用户定义的操作函数(UDF)，用于之后的排序
  val sortUdf: UserDefinedFunction = udf((rows: Se
    rows.map { case Row(movieId: String, timestamp
      .sortBy { case (movieId, timestamp) => times
      .map { case (movieId, timestamp) => movieId
  })


  //把原始的rating数据处理成序列数据
  val userSeq = ratingSamples
    .where(col("rating") >= 3.5)  //过滤掉评分在3.5-
    .groupBy("userId")            //按照用户id分组
    .agg(sortUdf(collect_list(struct("movieId", "t
    .withColumn("movieIdStr", array_join(col("movi
                //把所有id连接成一个String，方便后续wo


  //把序列数据筛选出来，丢掉其他过程数据
  userSeq.select("movieIdStr").rdd.map(r => r.getA
```

通过这段代码生成用户的评分序列样本中，每条样本的形式非常简单，它就是电影 ID 组成的序列，比如下面就是 ID 为 11888 用户的观影序列：

```
296 380 344 588 593 231 595 318 480 110 253 288 47
```

Item2vec：模型训练

训练数据准备好了，就该进入我们这堂课的重头戏，模型训练了。手

```
def trainItem2vec(samples : RDD[Seq[String]]): Uni
    //设置模型参数
    val word2vec = new Word2Vec()
```

```
    val word2vec = new Word2vec()
      .setVectorSize(10)
      .setWindowSize(5)
      .setNumIterations(10)


  //训练模型
  val model = word2vec.fit(samples)


  //训练结束，用模型查找与item"592"最相似的20个item
  val synonyms = model.findSynonyms("592", 20)
  for((synonym, cosineSimilarity) <- synonyms) {
    println(s"$synonym $cosineSimilarity")
  }

  //保存模型
  val embFolderPath = this.getClass.getResource("/
  val file = new File(embFolderPath.getPath + "emb
  val bw = new BufferedWriter(new FileWriter(file)
  var id = 0
  //用model.getVectors获取所有Embedding向量
  for (movieId <- model.getVectors.keys){
    id+=1
    bw.write( movieId + ":" + model.getVectors(mov
  }
  bw.close()
```

从上面的代码中我们可以看出，Spark 的 Word2vec 模型训练过程非常简单，只需要四五行代码就可以完成。接下来，我就按照从上到下的顺序，依次给你解析其中 3 个关键的步骤。

首先是创建 Word2vec 模型并设定模型参数。我们要清楚 Word2vec 模型的关键参数有 3 个，分别是 setVectorSize、setWindowSize 和

setNumIterations。其中，setVectorSize 用于设定生成的 Embedding 向量的维度，setWindowSize 用于设定在序列数据上采样的滑动窗口大小，setNumIterations 用于设定训练时的迭代次数。这些超参数的具体选择就要根据实际的训练效果来做调整了。

其次，模型的训练过程非常简单，就是调用模型的 fit 接口。训练完成后，模型会返回一个包含了所有模型参数的对象。

最后一步就是提取和保存 Embedding 向量，我们可以从最后的几行代码中看到，调用 getVectors 接口就可以提取出某个电影 ID 对应的 Embedding 向量，之后就可以把它们保存到文件或者其他数据库中，供其他模块使用了。

在模型训练完成后，我们再来验证一下训练的结果是不是合理。我在代码中求取了 ID 为 592 电影的相似电影。这部电影叫 Batman 蝙蝠侠，我把通过 item2vec 得到相似电影放到了下面，你可以从直观上判断一下这个结果是不是合理。

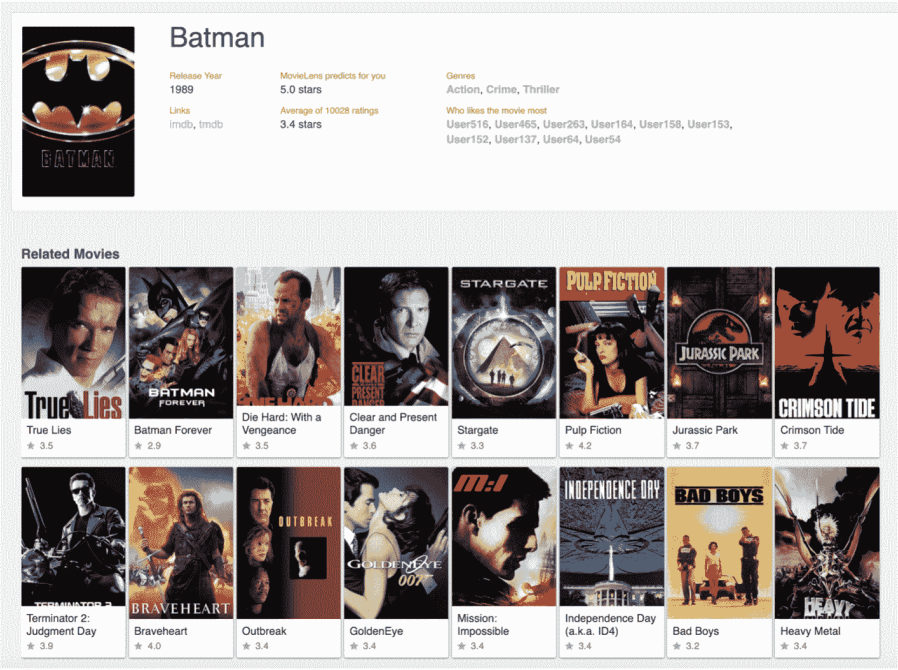
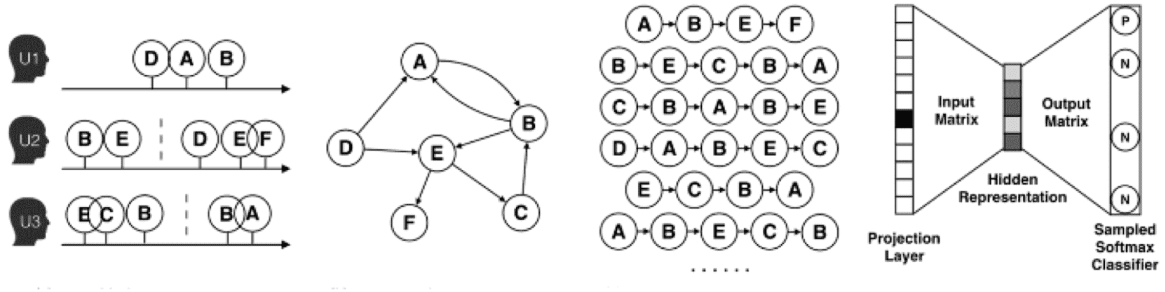


图2 通过Item2vec方法找出的电影Batman的相似电影

当然，因为 SparrowRecsys 在演示过程中仅使用了 1000 部电影和部分用户评论集，所以我们得出的结果不一定非常准确，如果你有兴趣优化这个结果，可以去 movieLens 下载全部样本进行重新训练。

Graph Embedding：数据准备

到这里，我相信你已经熟悉了 Item2vec 方法的实现。接下来，我们再来说说基于随机游走的 Graph Embedding 方法，看看如何利用 Spark 来实现它。这里，我们选择 Deep Walk 方法进行实现。



（a）用户行为序列　（b）构建物品关系图　（c）随机游走　（d）使用Word2vec模型产生Embedding

图3 Deep Walk的算法流程

在 Deep Walk 方法中，我们需要准备的最关键数据是物品之间的转移概率矩阵。图 3 是 Deep Walk 的算法流程图，转移概率矩阵表达了图 3（b）中的物品关系图，它定义了随机游走过程中，从物品 A 到物品

B 的跳转概率。所以我们首先来看一下如何利用 Spark 生成这个转移概率矩阵。

```
//samples 输入的观影序列样本集
def graphEmb(samples : RDD[Seq[String]], sparkSess
  //通过flatMap操作把观影序列打碎成一个个影片对
  val pairSamples = samples.flatMap[String]( sampl
    var pairSeq = Seq[String]()
    var previousItem:String = null
    sample.foreach((element:String) => {
      if(previousItem != null){
        pairSeq = pairSeq :+ (previousItem + ":" +
      }
      previousItem = element
    })
    pairSeq
  })
  //统计影片对的数量
  val pairCount = pairSamples.countByValue()
  //转移概率矩阵的双层Map数据结构
  val transferMatrix = scala.collection.mutable.Ma
  val itemCount = scala.collection.mutable.Map[Str


  //求取转移概率矩阵
  pairCount.foreach( pair => {
    val pairItems = pair._1.split(":")
    val count = pair._2
    lognumber = lognumber + 1
    println(lognumber, pair._1)


    if (pairItems.length == 2){
      val item1 = pairItems.apply(0)
      val item2 = pairItems.apply(1)
      if(!transferMatrix.contains(pairItems.apply(
```

```
      if(!transferMatrix.contains(pairItems.apply(
        transferMatrix(item1) = scala.collection.m
      }


      transferMatrix(item1)(item2) = count
      itemCount(item1) = itemCount.getOrElse[Long]
    }
```

生成转移概率矩阵的函数输入是在训练 Item2vec 时处理好的观影序列数据。输出的是转移概率矩阵，由于转移概率矩阵比较稀疏，因此我没有采用比较浪费内存的二维数组的方法，而是采用了一个双层 map 的结构去实现它。比如说，我们要得到物品 A 到物品 B 的转移概率，那么 transferMatrix(itemA)(itemB) 就是这一转移概率。

在求取转移概率矩阵的过程中，我先利用 Spark 的 flatMap 操作把观影序列“打碎”成一个个影片对，再利用 countByValue 操作统计这些影片对的数量，最后根据这些影片对的数量求取每两个影片之间的转移概率。

在获得了物品之间的转移概率矩阵之后，我们就可以进入图 3（c）的步骤，进行随机游走采样了。

Graph Embedding：随机游走采样过程

随机游走采样的过程是利用转移概率矩阵生成新的序列样本的过程。这怎么理解呢？首先，我们要根据物品出现次数的分布随机选择一个起始物品，之后就进入随机游走的过程。在每次游走时，我们根据转移概率矩阵查找到两个物品之间的转移概率，然后根据这个概率进行跳转。比如当前的物品是 A，从转移概率矩阵中查找到 A 可能跳转到

物品 B 或物品 C，转移概率分别是 0.4 和 0.6，那么我们就按照这个概率来随机游走到 B 或 C，依次进行下去，直到样本的长度达到了我们的要求。

根据上面随机游走的过程，我用 Scala 进行了实现，你可以参考下面的代码，在关键的位置我也给出了注释：

```
//随机游走采样函数
//transferMatrix 转移概率矩阵
//itemCount 物品出现次数的分布
def randomWalk(transferMatrix : scala.collection.m
  //样本的数量
  val sampleCount = 20000
  //每个样本的长度
  val sampleLength = 10
  val samples = scala.collection.mutable.ListBuffe

  //物品出现的总次数
  var itemTotalCount:Long = 0
  for ((k,v) <- itemCount) itemTotalCount += v


  //随机游走sampleCount次，生成sampleCount个序列样本
  for( w <- 1 to sampleCount) {
    samples.append(oneRandomWalk(transferMatrix, i
  }



  Seq(samples.toList : _*)
}


//通过随机游走产生一个样本的过程
//transferMatrix 转移概率矩阵
```

```
//itemCount 物品出现次数的分布
//itemTotalCount 物品出现总次数
//sampleLength 每个样本的长度
def oneRandomWalk(transferMatrix : scala.collectio
  val sample = scala.collection.mutable.ListBuffer


  //决定起始点
  val randomDouble = Random.nextDouble()
  var firstElement = ""
  var culCount:Long = 0
  //根据物品出现的概率，随机决定起始点
  breakable { for ((item, count) <- itemCount) {
    culCount += count
    if (culCount >= randomDouble * itemTotalCount)
      firstElement = item
      break
    }
  }}


  sample.append(firstElement)
  var curElement = firstElement
  //通过随机游走产生长度为sampleLength的样本
  breakable { for( w <- 1 until sampleLength) {
    if (!itemCount.contains(curElement) || !transf
      break
    }
    //从curElement到下一个跳的转移概率向量
    val probDistribution = transferMatrix(curEleme
    val curCount = itemCount(curElement)
    val randomDouble = Random.nextDouble()
    var culCount:Long = 0
    //根据转移概率向量随机决定下一跳的物品
    breakable { for ((item, count) <- probDistribu
      culCount += count
```

```
      if (culCount >= randomDouble * curCount){
        curElement = item
        break
      }
    }}
    sample.append(curElement)
  }}
  Seq(sample.toList : _
```

通过随机游走产生了我们训练所需的 sampleCount 个样本之后，下面的过程就和 Item2vec 的过程完全一致了，就是把这些训练样本输入到 Word2vec 模型中，完成最终 Graph Embedding 的生成。你也可以通过同样的方法去验证一下通过 Graph Embedding 方法生成的 Embedding 的效果。

小结

这节课，我们运用 Spark 实现了经典的 Embedding 方法 Item2vec 和 Deep Walk。它们的理论知识你应该已经在前两节课的学习中掌握了，这里我就总结一下实践中应该注意的几个要点。

关于 Item2vec 的 Spark 实现，你应该注意的是训练 Word2vec 模型的几个参数 VectorSize、WindowSize、NumIterations 等，知道它们各自的作用。它们分别是用来设置 Embedding 向量的维度，在序列数据上采样的滑动窗口大小，以及训练时的迭代次数。

而在 Deep Walk 的实现中，我们应该着重理解的是，生成物品间的转移概率矩阵的方法，以及通过随机游走生成训练样本过程。

最后，我还是把这节课的重点知识总结在了一张表格中，希望能帮助你进一步巩固。

知识点	关键描述
Spark中的自定义函数	用于实现一些Spark自带聚合函数外的复杂操作， 比如在Item2vec中实现的排序函数sortUdf: UserDefinedFunction
Word2vec中的关键参数	VectorSize、WindowSize和NumIterations
flatMap操作和map操作的区别	flatMap操作能够实现单条输入、多条输出， 而map操作只能够进行单条输入、单条输出
Deep Walk中随机游走的过程	是利用转移概率矩阵生成新的序列样本的过程， 其中最关键的两个函数是randomWalk和oneRandomWalk
Deep Walk中 转移概率矩阵的求取	使用到graphEmb函数，输入的是观影序列数据， 输出的是转移概率矩阵，并且采用了双层map的结构

这里，我还想再多说几句。这节课，我们终于看到了深度学习模型的产出，我们用 Embedding 方法计算出了相似电影！对于我们学习这门课来说，它完全可以看作是一个里程碑式的进步。接下来，我希望你能总结实战中的经验，跟我继续同行，一起迎接未来更多的挑战！

课后思考

上节课，我们在讲 Graph Embedding 的时候，还介绍了 Node2vec 方法。你能尝试在 Deep Walk 代码的基础上实现 Node2vec 吗？这其中，我们应该着重改变哪部分的代码呢？

欢迎把你的思考和答案写在留言区，如果你掌握了 Embedding 的实战方法，也不妨把它分享给你的朋友吧，我们下节课见！