8. Spark中RDD基础、键值对操作和MLlib

罗翔宇
中国人民大学统计与大数据研究院

此课件内容完全基于Holden Karau等编著的《Spark快速大数据分析》（O'REILLY，王道远译）

上周复习

- Spark是一个用来实现**快速**而通用的集群计算的平台
- 在Spark中，我们通过对分布式数据集的操作来表达计算意图，这些计算会自动地在集群上并行执行。这样的数据集被称为弹性分布式数据集（RDD）
- 创建RDD：(1)将程序中已有的集合转化为RDD;(2)从外部存储中读取数据
- 转化操作：从一个RDD到另一个RDD
  - 惰性求值的，在被调用行动操作之前，Spark不会开始计算
  - 对于一个RDD：map，filter，flatmap
  - 对于两个RDD：union, intersection, subtract, cartesian

行动操作

- reduce()接收一个函数作为参数，这个函数要操作两个相同元素类型的数据并返回一个同样类型的新元素

```
scala> val nums = sc.parallelize(List(1,2,3,3))
nums: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[1] at parallelize a
t <console>:24

scala> val sum = nums.reduce((x,y) => x+y)
sum: Int = 9

scala> val sum = nums.fold(0)((x,y) => x+y)
sum: Int = 9
```

- fold()和reduce()的功能一样，但是需要提供“初始值”来作为每个分区第一次调用的结果。初始值应是所提供操作的单位元素。

```
scala> val prod = nums.fold(1)((x,y) => x*y)
prod: Int = 18
```

行动操作

- aggregate()函数允许返回值类型与RDD中元素类型不同（比如返回一个二元数组）
- 使用aggregate()时，需要提供返回的类型的初始值
- 通过一个函数把RDD中的元素合并起来放入累加器
- 由于每个节点是在本地进行累加的，还需要提供第二个函数来将累加器两两合并
- 例子：计算RDD中元素的平均数

```
scala> val result = nums.aggregate((0,0))(
     |                    (acc, value) => (acc._1 + value, acc._2 + 1),
     |                    (acc1, acc2) => (acc1._1 + acc2._1, acc1._2 + acc2._2))
result: (Int, Int) = (9,4)

scala> val avg = result._1 / result._2.toDouble
avg: Double = 2.25
```

行动操作

- collect()将数据返回到驱动器程序，其要求所有数据都必须能放入单台机器的内存中
- take(n)返回RDD中的n个元素

```
scala> nums.collect()
```

```
scala> nums.take(2)
```

行动操作

对数据为{1,2,3,3}的RDD进行基本的RDD行动操作

函数名	目的	示例	结果
collect()	返回RDD中的所有元素	rdd.collect()	{1,2,3,3}
count()	RDD中的元素个数	rdd.count()	4
countByValue()	各元素在RDD中出现的次数	rdd.countByValue()	{(1,1),(2,1),(3,2)}
take(num)	从RDD中返回num个元素	rdd.take(2)	{1,2}
reduce(func)	并行整合RDD中所有数据	rdd.reduce((x, y)=>x+y)	9

Table credit: 《Spark快速大数据分析》表3-4

行动操作

对数据为{1,2,3,3}的RDD进行基本的RDD行动操作

函数名	目的	示例	结果
fold(zero)(func)	和reduce()一样，但是需要提供初始值	rdd.fold(0)((x,y)=>x+y)	9
aggregate(zeroValue)(seqOp, combOp)	和reduce()相似，但是通常返回不同类型的函数	rdd.aggregate((0,0))((x,y)=>(x._1+y,x._2+1), (x,y)=>(x._1+y._1,x._2+y._2))	(9,4)
foreach(func)	对RDD中的每个元素使用给定的函数	rdd.foreach(func)	

Table credit: 《Spark快速大数据分析》表3-4

持久化（缓存）

- Spark RDD是惰性求值的，有时希望能多次使用同一个RDD，如果简单地对RDD调用行动操作，Spark每次都会重算RDD以及它的所有依赖
- 为避免多次计算同一个RDD，可以让Spark对数据持久化
- 在Scala中，persist()会把数据以序列化的形式缓存在JVM的堆空间中，且有不同的持久化级别（MEMORY_ONLY, MEMORY_AND_DISK, DISK_ONLY）

```
scala> import org.apache.spark.storage.StorageLevel
import org.apache.spark.storage.StorageLevel

scala> val result = nums.map(x => x*x)
result: org.apache.spark.rdd.RDD[Int] = MapPartitionsRDD[5] at map at <console>
:26

scala> result.persist(StorageLevel.DISK_ONLY)
res4: result.type = MapPartitionsRDD[5] at map at <console>:26
```

- unpersist()，该方法可以手动把持久化RDD从缓存中移除

一种特殊的RDD：键值对RDD

- (key, value)：键值对RDD是Spark中许多操作所需要的常见数据类型（比如MapReduce框架下，求某个文件的单词频数），也叫做pair RDD
- Pair RDD提供了并行操作各个键或跨节点重新进行数据分组的操作接口
- 比如，pair RDD提供reduceByKey()方法，分别归约每个键对应的数据（实际例子：词频统计）
- join()方法，可以把两个RDD中键相同的元素组合到一起，合并为一个RDD（实际例子：通过用户ID的数据融合）

创建Pair RDD

- 1. 很多存储键值对的数据格式会在读取时直接返回由其键值对数据组成的pair RDD
- 2. 通过map()将一个普通RDD转化为pair RDD，传递的函数需要返回键值对
- 例子：以第一个单词为键创造出一个pair RDD

```
scala> val lines = sc.parallelize(List("Hello everyone! Good morning!", "How ar
e you?", "Good! Thank you!"))
lines: org.apache.spark.rdd.RDD[String] = ParallelCollectionRDD[6] at paralleli
ze at <console>:25

scala> val pairs = lines.map(x => (x.split(" ")(0), x))
pairs: org.apache.spark.rdd.RDD[(String, String)] = MapPartitionsRDD[7] at map
at <console>:26
```

Pair RDD的转化操作

- mapValues(func)对pair RDD中的每个值应用一个函数而不改变键
- reduceByKey()合并具有相同键的值，会为数据集中的每个键进行并行的聚合操作。（聚合操作是一种转化操作）

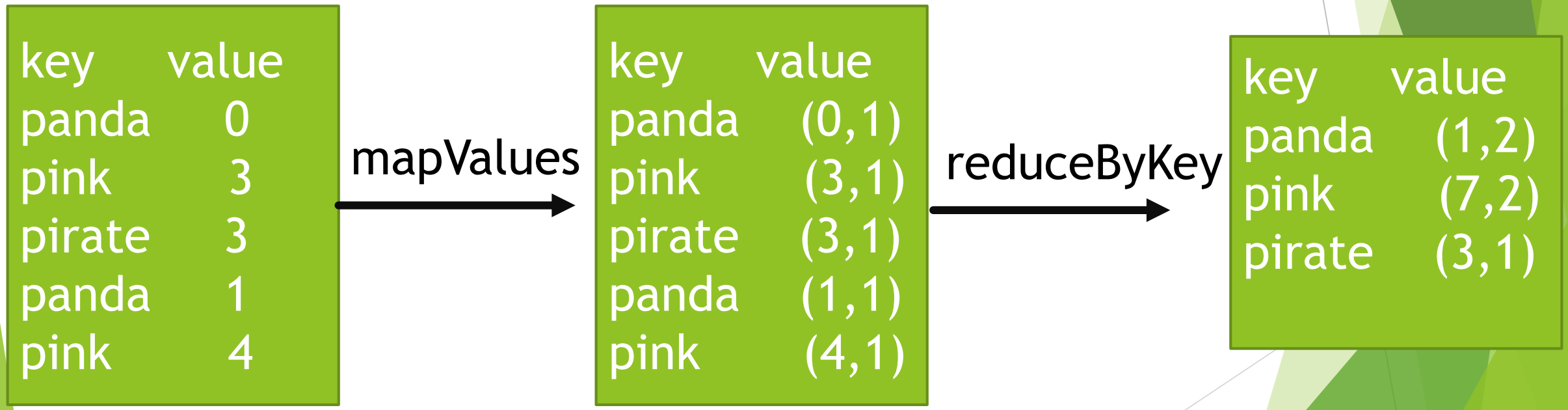

Figure credit: 《Spark快速大数据分析》图4-2

Pair RDD的转化操作

- 代码实现

```
scala> val rdd = sc.parallelize(List(("panda", 0), ("pink", 3), ("pirate", 3),
("panda", 1), ("pink", 4)))
rdd: org.apache.spark.rdd.RDD[(String, Int)] = ParallelCollectionRDD[5] at para
llelize at <console>:24

scala> rdd.mapValues(x => (x,1)).reduceByKey((x,y) => (x._1+y._1, x._2+y._2))
res2: org.apache.spark.rdd.RDD[(String, (Int, Int))] = ShuffledRDD[7] at reduce
ByKey at <console>:26
```

Pair RDD的转化操作

对数据为{(1,2),(3,4),(3,6)}的pair RDD的转化操作

函数名	目的	示例	结果
reduceByKey(func)	合并具有相同键的值	rdd.reduceByKey((x, y)=>x+y)	{(1,2),(3,10)}
groupByKey	对具有相同键的值进行分组	rdd.groupByKey()	{(1,[2]),(3,[4,6])}
mapValues(func)	对pairRDD中的每个值应用一个函数而不改变键	rdd.mapValues(x=>x+1)	{(1,3),(3,5),(3,7)}
flatMapValues(func)	对pairRDD中的每个值应用一个返回迭代器的函数	rdd.mapValues(x=>(x to 5))	{(1,2),(1,3),(1,4),(3,4),(3,5)}

Table credit: 《Spark快速大数据分析》表4-1

Pair RDD的转化操作

对数据为{(1,2),(3,4),(3,6)}的pair RDD的转化操作

函数名	目的	示例	结果
keys()	返回一个仅包含键的RDD	rdd.keys	{1,3,3}
values()	返回一个仅包含值的RDD	rdd.values	{2,4,6}
sortByKey()	返回一个根据键排序的RDD	rdd.sortByKey()	{(1,2),(3,4),(3,6)}

Table credit: 《Spark快速大数据分析》表4-1

Pair RDD的聚合操作

- 利用reduceByKey(),解决分布式单词计数问题

```
scala> val lines = sc.parallelize(List("hello i am a cat", "hello i am a kid",
"hello we are friends"))
lines: org.apache.spark.rdd.RDD[String] = ParallelCollectionRDD[0] at paralleli
ze at <console>:24

scala> val words = lines.flatMap(x => x.split(" "))
words: org.apache.spark.rdd.RDD[String] = MapPartitionsRDD[1] at flatMap at <co
nsole>:25

scala> val pairRDD = words.map(x => (x, 1))
pairRDD: org.apache.spark.rdd.RDD[(String, Int)] = MapPartitionsRDD[2] at map a
t <console>:25

scala> val result = pairRDD.reduceByKey((x,y) => x+y)
result: org.apache.spark.rdd.RDD[(String, Int)] = ShuffledRDD[3] at reduceByKey
 at <console>:25
```

Pair RDD的聚合操作

- combineByKey()有多个参数分别对应聚合操作的各个阶段，非常适合用来解释聚合操作各个阶段的功能划分

```
scala> val rdd = sc.parallelize(List(("panda", 0), ("pink", 3), ("pirate", 3),
("panda", 1), ("pink", 4)))
rdd: org.apache.spark.rdd.RDD[(String, Int)] = ParallelCollectionRDD[11] at par
allelize at <console>:24

scala> val result = rdd.combineByKey(
     |         (v) => (v, 1),
     |         (acc: (Int, Int), v) => (acc._1 + v, acc._2 + 1)
     |         (acc1: (Int, Int), acc2: (Int, Int)) => (acc1._1 + acc2._1, acc1._
2 + acc2._2)).map{ case (key, value) => (key, value._1 / value._2.toFloat)}
```

Pair RDD的聚合操作

- 并行度调优：每个RDD都有固定数目的分区，分区数决定了在RDD上执行操作时的并行度

```
scala> val data = Seq(("a", 3), ("b", 4), ("a", 1))
data: Seq[(String, Int)] = List((a,3), (b,4), (a,1))

scala> sc.parallelize(data).reduceByKey((x,y) => x+y)
res5: org.apache.spark.rdd.RDD[(String, Int)] = ShuffledRDD[15] at reduceByKey
at <console>:27

scala> sc.parallelize(data).reduceByKey((x,y) => x+y, 10)
res6: org.apache.spark.rdd.RDD[(String, Int)] = ShuffledRDD[17] at reduceByKey
at <console>:27
```

- 第一个reduceByKey()使用了默认并行度
- 第二个reduceByKey()使用了自定义并行度

数据分组

- 对于有键的数据，一个常见例子是根据键进行分组，比如查看一个顾客的所有订单
- groupByKey()会使用RDD中的键来对数据进行分组
- 类型K的键和类型V的值，其返回的结果是[K, Iterable[V]]
- 请注意大多数情况下，使用一种根据键进行聚合的函数更为高效
- 比如，rdd.reduceByKey(func)比rdd.groupByKey().mapValues(value => value.reduce(func))等价，但前者更为高效

连接操作

- 连接操作作用于两组有键但具有不同类型值的RDD
- 连接方式有：右外连接，左外连接，交叉连接，内链接
- join表示内连接，只有两个pair RDD中都存在的键才是输出
- leftOuterJoin(other)左外连接，左边的pairRDD的键都存在
- rightOuterJoin(other)右外连接，右边的pairRDD的键都存在

连接操作

```
scala> val storeAddress = sc.parallelize(Seq(("Ritual", "1026 Valencia St"), ("
Philz", "748 Van Ness Ave"), ("Philz", "3101 24th St"), ("Starbucks", "Seattle"
)))
storeAddress: org.apache.spark.rdd.RDD[(String, String)] = ParallelCollectionRD
D[18] at parallelize at <console>:24

scala> val storeRating = sc.parallelize(Seq(("Ritual",4.9),("Philz", 4.8)))
storeRating: org.apache.spark.rdd.RDD[(String, Double)] = ParallelCollectionRDD
[19] at parallelize at <console>:24

scala> storeAddress.join(storeRating)
res7: org.apache.spark.rdd.RDD[(String, (String, Double))] = MapPartitionsRDD[2
2] at join at <console>:28
```

连接操作

```
scala> storeAddress.leftOuterJoin(storeRating)
res8: org.apache.spark.rdd.RDD[(String, (String, Option[Double]))] = MapPartiti
onsRDD[25] at leftOuterJoin at <console>:28

scala> storeAddress.rightOuterJoin(storeRating)
res9: org.apache.spark.rdd.RDD[(String, (Option[String], Double))] = MapPartiti
onsRDD[28] at rightOuterJoin at <console>:28
```

两个Pair RDD的链接操作

针对两个pair RDD的转化操作，rdd={(1,2),(3,4),(3,6)}, other={(3,9)}

函数名	目的	示例	结果
subtractByKey	删掉RDD中与other中的键相同的键	rdd.subtractByKey(other)	{(1,2)}
join	对两个RDD进行内链接	rdd.join(other)	{(3,(4,9)),(3,(6,9))}
rightOuterJoin	对两个RDD进行链接操作，确保右边RDD的键必须存在	rdd.rightOuterJoin(other)	{(3,(Some(4),9)),(3, (Some(6),9))}
leftOuterJoin	对两个RDD进行链接操作，确保左边RDD的键必须存在	rdd.leftOuterJoin(other)	{(1,(2,None)), (3,(4, Some(9))), (3,(6, Some(9)))}

Table credit: 《Spark快速大数据分析》表4-2

数据排序以及Pair RDD的行动操作

- sortByKey对pairRDD的键进行排序，其中的参数ascending表示我们是否让结果升序排序

针对pair RDD的行动操作，rdd = {(1,2), (3,4), (3,6)}

函数名	目的	示例	结果
countByKey	对每个键对应的元素分别计数	rdd.countByKey	{(1,1), (3,2)}
collectAsMap()	将结果以映射的形式返回	rdd.collectAsMap()	Map{(1,2), (3,6)}
Lookup(key)	返回给定键对应的所有值	rdd.lookup(3)	{4，6}

Table credit: 《Spark快速大数据分析》表4-3

数据读取与保存

- 目前所展示的例子均是从本地集合或者普通文件进行数据读取和保存的。但有时，数据量可能大到无法放在一台机器中。
- Spark支持多种输入输出源，三类常见的数据源
- 1.文件格式与文件系统
  - 存储在本地文件系统或分布式文件系统中的数据
  - 可以访问不同的文件格式：文本文件、JSON、SequenceFile, protocol buffer
- 2.Spark SQL中的结构化数据源
  - 包括JSON和Apache Hive在内的结构化数据源
- 3.数据库与键值存储

文件格式

Spark支持的一些常见格式

格式名称	结构化	备注
文本文件	否	普通的文本文件，每行一条记录
JSON	半结构化	常见的基于文本的格式，半结构化；大多数库都要求每行一条记录
CSV	是	非常常见的基于文本的格式，通常在电子表格应用中使用
SequenceFiles	是	一种用于键值对数据的常见Hadoop文件格式
Protocol buffers	是	一种快速、节约空间的跨语言格式

Table credit: 《Spark快速大数据分析》表5-1

文本文件

- 将一个文本文件读取为RDD时，输入的每一行都会成为RDD的一个元素

```
scala> val input = sc.textFile("README.md")
input: org.apache.spark.rdd.RDD[String] = README.md MapPartitionsRDD[1] at text
File at <console>:24
```

- sc.wholeTextFiles()同时处理多个文件，此时返回一个pairRDD,键是输入文件的文件名，值是文件内容
- 保存文本文件，saveAsTextFile()方法接收一个路径，并将RDD中的内容都输入到路径对应的文件中
- 若将路径作为目录对待，会在那个目录下输入多个文件，Spark便能从多个节点上并行输出了

逗号分隔值与制表符分隔值

- CSV每行都有固定数目的字段，字段间用逗号隔开
- TSV文件中用制表符(Tab)隔开
- 先把文件当作普通文本文件来读取数据，再对数据进行处理

```
scala> import java.io.StringReader
import java.io.StringReader

scala> import au.com.bytecode.opencsv.CSVReader
import au.com.bytecode.opencsv.CSVReader

scala> val input = sc.textFile("example.csv")
input: org.apache.spark.rdd.RDD[String] = example.csv MapPartitionsRDD[5] at te
xtFile at <console>:28

scala> val result = input.map{line => val reader = new CSVReader(new StringRead
er(line)); reader.readNext();}
result: org.apache.spark.rdd.RDD[Array[String]] = MapPartitionsRDD[6] at map at
 <console>:29
```

基于MLlib的机器学习

- MLlib是Spark中提供机器学习函数的库，专为在集群上并行运行的情况而设计的
- MLlib设计理念：把数据以RDD的形式表示，然后在分布式数据集上调用各种算法
- 比如用MLlib识别垃圾邮件的任务
  - 用字符串RDD表示每个邮件内容
  - 用MLlib中的一个特征提取算法将文本数据转化为数值数据，该操作返回一个向量RDD
  - 对向量RDD调用分类算法，比如逻辑回归
  - 使用MLlib中评估函数在测试数据集上评估模型
- 注意：MLlib中只包含能够在集群上运行良好的并行算法；只有当在一个大规模分布式数据集上训练模型时，MLlib的优势才能凸显出来

机器学习基础

- 系统要求：MLlib需要机器预装一些线性代数库，安装gfortran运行库。若在Python中使用MLlib，需要安装NumPy
- 几种机器学习问题：分类、回归、聚类
- 分类是基于已经被标记的数据点作为学习对象来识别新的数据属于哪一类别的问题

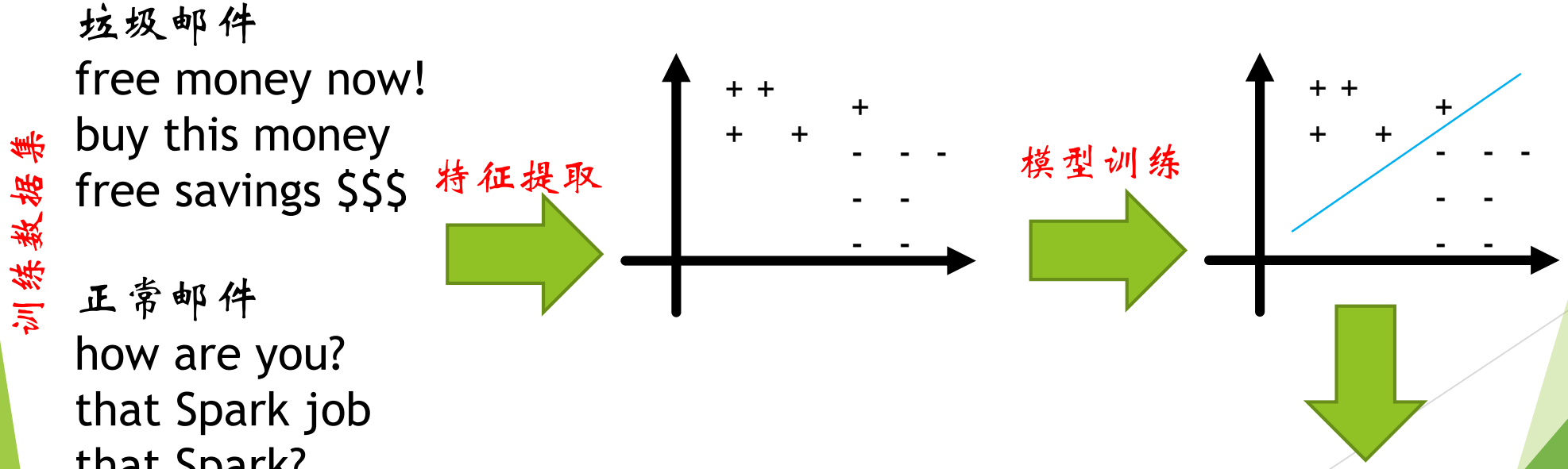

Figure credit: 《Spark快速大数据分析》 图11-1

机器学习基础

- import org.apache.spark.mllib.regression.LabeledPoint
- import org.apache.spark.mllib.feature.HashingTF
- import org.apache.spark.mllib.classification.LogisticRegressionWithSGD

- HashingTF从文本数据构建词频(term frequency)特征向量
- LogisticRegressionWithSGD使用随机梯度下降实现逻辑回归

Code credit: 《Spark快速大数据分析》例11-2

机器学习基础

- val spam = sc.textFile(“spam.txt”)
- val normal = sc.textFile(“normal.txt”)
- //创建一个HashingTF实例来把邮件文本映射为包含10000个特征的向量
- val tf = new HashingTF(numFeatures = 10000)
- //各邮件被切分为单词，每个单词被映射为一个特征
- val spamFeatures = spam.map(email => tf.transform(email.split(“ “)))
- val normalFeatures = normal.map(email => tf.transform(email.split(“ “)))

Code credit: 《Spark快速大数据分析》例11-2

机器学习基础

- //创建LabeledPoint数据集分别存放垃圾邮件(positive)和正常邮件(negative)
- val positiveExamples = spamFeatures.map(features => LabeledPoint(1, features))
- val negativeExamples = normalFeatures.map(features => LabeledPoint(0, features))
- val trainingData = positiveExamples.union(negativeExamples)
- trainingData.cache() //因为逻辑回归是迭代算法，所以缓存训练数据RDD

Code credit: 《Spark快速大数据分析》例11-2

机器学习基础

- //使用SGD算法运行逻辑回归
- val model = new LogisticRegressionWithSGD().run(trainingData)

- //在测试集上进行测试
- val posTest = tf.transform("O M G GET cheap stuff by sending money to …".split(" "))
- val negTest = tf.transform("Hi Dad, I started studying Spark the other …".split(" "))
- println("Prediction: " + model.predict(posTest))
- println("Prediction: " + model.predict(negTest))

Code credit: 《Spark快速大数据分析》例11-2

数据类型

- MLlib包含一些特有的数据类型，位于org.apache.spark.mllib包中
- Vector: 一个数学向量。既支持稠密向量也支持稀疏向量。前者表示向量的每一位都存储下来，后者则只会存储非零元以节约空间
- LabeledPoint: 在分类或回归这样的监督性学习算法中，LabeledPoint用来表示带标签的数据点
- Rating：用户对一个产品的评分，在mllib.recommendation中，用于产品推荐
- 各种Model类：一般有一个predict()方法可以用来对新的数据点进行预测

操作向量

```
scala> import org.apache.spark.mllib.linalg.Vectors
import org.apache.spark.mllib.linalg.Vectors

scala> val denseVec1 = Vectors.dense(1.0, 2.0, 3.0)
denseVec1: org.apache.spark.mllib.linalg.Vector = [1.0,2.0,3.0]

scala> val denseVec2 = Vectors.dense(Array(1.0, 2.0, 3.0))
denseVec2: org.apache.spark.mllib.linalg.Vector = [1.0,2.0,3.0]

scala> //create a vector <1, 0, 2, 0>

scala> val sparseVec1 = Vectors.sparse(4, Array(0, 2), Array(1.0, 2.0))
sparseVec1: org.apache.spark.mllib.linalg.Vector = (4,[0,2],[1.0,2.0])
```

算法-特称提取

- mllib.feature包中包含一些用来进行常见特征转化的类
- TF-IDF：词频-逆文档频率，是一种用来从文本文档中生成特称向量的方法。用两种算法可以进行计算：HashingTF和IDF
- 缩放：去除单位的影响，将所有特征调为平均值为0，标准差为1。可以使用StandardScaler进行。
- 正规化：把向量正规化为长度为1，使用Normalizer.transform(rdd)可以实现
- Word2Vec: 是一个基于神经网络的文本特征化算法，可以将数据传给许多下游算法。通过mllib.feature.Word2Vec实现

算法-统计

- mllib.stat.Statistics中的提供了广泛之用的统计函数
- Statistics.colStats(rdd):计算向量集合中，每列的最小值、最大值、平均值和方差
- Statistics.corr(rdd, method)：计算RDD中列间的相关矩阵，method可以是pearson或spearman中的一种
- Statistics.corr(rdd1, rdd2, method)：计算两个RDD间的相关矩阵
- Statistics.chiSqTest(rdd)：计算由LabeledPoint对象组成的RDD中每个特征与标签的独立性测试
- mean(),stdev(),sum()

算法-分类与回归

- 分类与回归是监督式学习的两种主要形式
- 分类与回归都会使用Mllib中的LabeledPoint类
- 线性回归
  - mllib.regression.LinearRegressionWithSGD
  - LassoWithSGD, RidgeRegressionWithSGD
  - 参数：numIterations, stepSize, intercept, regParam(正规化参数)
- 逻辑回归
  - mllib.classification.LogisticRegressionWithSGD

算法-分类与回归

- 支持向量机
  - SVMWithSGD
- 朴素贝叶斯
  - mllib.classification.NaiveBayes
- 决策树与随机森林
  - mllib.tree.DecisionTree
  - RandomForrest

算法-聚类、降维

- 聚类算法：K-means
  - mllib.clustering.Kmeans
  - 参数：initialzationMode，maxIterations, runs(算法兵法运行的数目)
- 降维
  - 主成分分析
  - mat.computePrincipalComponents
  - 奇异值分解
  - mat.computeSVD

模型评估

- 不同模型或同一模型不同参数设置都会导致不一样的结果
- 在训练数据集上训练模型，在测试数据集上评估模型
- 不能使用训练数据集来评估模型，否则导致模型的过度拟合(overfitting)
- 一些评估函数位于mllib.evaluation包中，比如
  - BinaryClassificationMetrics
  - MulticlassMetrics
  - 可计算sensitivity, specificity, AUC等指标