# Spark MLlib 机器学习

## Spark MLlib 基本数据类型

MLlib支持的数据类型

|  |  |
| --- | --- |
| 类型名称 | 释义 |
| Local Vector 本地向量集 | 向spark 提供一组可操作的数据集合 |
| Labeled point 向量标签 | 让用户分类不同的数据集合 |
| Local Matrix 本地矩阵 | 将数据集合以矩阵的形式存储在本地集合中 |
| Distributed Matrix 分布式矩阵 | 将数据以矩阵形式存储在分布式计算机中 |

### 本地向量集

我们这里主要介绍两类向量集：稀疏型数据集（spares）和密集型数据集（dense）。

我们通过代码来演示。

代码：

|  |
| --- |
| **import** org.apache.spark.mllib.linalg.{Vectors,Vector}  **object** MLlibTest {  **def** main(args: Array[String]): Unit = {   **val** vd:Vector = Vectors.*dense*(2,0,6)  *println*(vd(0))  **val** vs:Vector = Vectors.*sparse*(4,*Array*(0,1,2,3),*Array*(9,5,2,7))  *println*(vs(3))  } } |

dense方法不需要多做解释，可以将其理解为MLlib专用的一种集合形式，它与Array类似，最终显示结果和方法调用也类似。

而spare方法是将给定的数据Array数据（9,5,2,7）分解成4个部分进行处理，其对应值分别属于程序中vs的向量对应值。接下来我们看一下第一个参数4，在这里代表输入数据的大小，一般要求大于等于输入的数据值；而第三个参数Array（9,5,2,7）是输入的数据值，这里一般要求将其作为一个Array类型的数据进行输入；第二个参数Array（0,1,2,3）是数据vs下标的数值。这里严格要求按序增加的方法增加数据。

### 向量标签

向量标签用于对MLlib中机器学习算法的不同值做标记。例如：分类问题中，可以将不同的数据集分成若干份，以整型数0、1、2.......进行标记，即可以根据自己的需要对数据进行标记。

|  |
| --- |
| **import** org.apache.spark.mllib.linalg.{Vector, Vectors} **import** org.apache.spark.mllib.regression.LabeledPoint  **object** MLlibTest {  **def** main(args: Array[String]): Unit = {   **val** vd:Vector = Vectors.*dense*(2,0,6)  **val** pos = *LabeledPoint*(1,vd)  *println*(pos.features)  *println*(pos.label)  **val** vs:Vector = Vectors.*sparse*(4,*Array*(0,1,2,3),*Array*(9,5,2,7))  **val** neg = *LabeledPoint*(2,vs)  *println*(neg.features)  *println*(neg.label)  }  } |

结果：

|  |
| --- |
| [2.0,0.0,6.0]  1.0  (4,[0,1,2,3],[9.0,5.0,2.0,7.0])  2.0 |

我们从结果可以看出，LabeledPoint是建立向量标签的静态类，主要有两个方法，Features用于显示打印标记点所代表的内容，而label用于显示标记数。

除了使用以上两种方法建立向量标签，MLlib还支持直接从数据库中获取固定格式的数据集方法。其数据格式如下：

|  |
| --- |
| Label index:value1 index2:value2 ... //Label是每一行的标签，index索引值，value是数据值 |

**例子:**

数据：

|  |
| --- |
| 1 0:2 1:3 2:4  2 0:5 1:8 2:9  1 0:7 1:6 2:7  1 0:3 1:2 2:1 |

使用方法：将以上数据复制到一个自定义的数据文件中，之后调用MLUtils.loadLibSVMfile方法对数据进行读取。

代码：

|  |
| --- |
| **import** org.apache.spark.mllib.util.MLUtils **import** org.apache.spark.{SparkConf, SparkContext}  **object** TestLabeledPoint2 {   **def** main(args: Array[String]): Unit = {  **val** conf = **new** SparkConf().setMaster(**"local"**).setAppName(**"TestLabeledPoint2"**)  **val** sc = **new** SparkContext(conf)  **val** mu = MLUtils.*loadLibSVMFile*(sc,**"D://test//22.txt"**)  mu.foreach(*println*)  }  } |

提示：MLUtils.*loadLibSVMFile数据集标记的index是从1开始，我们例子中的数据是从0开始的，所以程序在执行时会报错。*

### 本地矩阵

大数据运算中，为了更好的提升计算效率，可以更多地使用矩阵运算进行数据处理。部署在单机中的本地矩阵就是一个很好的存储方法。

举一个简单的例子，例如：一个数组Array（1,2,3,4,5,6）,将其分为2行3列的矩阵，可用如下程序处理。

|  |
| --- |
| **import** org.apache.spark.mllib.linalg.Matrices **import** org.apache.spark.mllib.util.MLUtils **import** org.apache.spark.{SparkConf, SparkContext}  **object** TestLabeledPoint2 {   **def** main(args: Array[String]): Unit = {  **val** mx = Matrices.*dense*(2,3,*Array*(1,2,3,4,5,6))  *println*(mx)  } } |

结果如下：

|  |
| --- |
| 1.0 3.0 5.0  2.0 4.0 6.0 |

从结果看，数组Array(1,2,3,4,5,6)被重组成一个新的2行3列的矩阵。Matrices.dense方法是矩阵重组的调用方法，第一个参数是新矩阵行数，第二个参数是新矩阵的列数，第三个参数为传入的数据值。

### 分布式矩阵

一般来说，采用分布式矩阵进行存储的情况都是数量非常大的，其处理速度和效率与其存储格式息息相关。MLlib提供了四种分布式矩阵存储形式，均由支持长整型的行列数和双精度浮点型的数据内容构成。这四种矩阵分别为：行矩阵、带有行索引的行矩阵、坐标矩阵和块矩阵，这里我们只介绍最常用的前三种。

1. 行矩阵

行矩阵是最基本的一种矩阵类型。行矩阵是以行作为基本方向的矩阵存储格式，列的作用相对较小。可以理解为行矩阵是一个巨大的特征向量的集合。每一行就是一个具有相同格式的向量数据，且每一行的向量内容都可以单独取出来进行操作。行矩阵的具体实例可参看下边的程序：

|  |
| --- |
| **import** org.apache.spark.mllib.linalg.distributed.RowMatrix **import** org.apache.spark.mllib.linalg.{Matrices, Vectors} **import** org.apache.spark.mllib.util.MLUtils **import** org.apache.spark.{SparkConf, SparkContext}  **object** TestLabeledPoint2 {   **def** main(args: Array[String]): Unit = {  **val** conf = **new** SparkConf().setMaster(**"local"**).setAppName(**"TestLabeledPoint2"**)  **val** sc = **new** SparkContext(conf)  **val** rdd = sc.textFile(**"d:\\test\\33.txt"**).map(\_.split(**" "**).map(\_.toDouble))  .map(x=>Vectors.*dense*(x))  **val** rm = **new** RowMatrix(rdd)  *println*(rm.numRows())  *println*(rm.numCols())  rm.rows.foreach(*println*)  } } |

运行结果如下：

|  |
| --- |
| 2  3  [1.0,2.0,3.0]  [4.0,5.0,6.0] |

数据如下：

|  |
| --- |
| 1 2 3  4 5 6 |

这是一个2行3列的矩阵，读者可以将其保存的磁盘上例如：“d:\test\33.txt”

1. **带有行索引的的行矩阵**

从行矩阵程序可以看到，单纯的行矩阵对其内容无法进行直接显示（可以调用其方法显示内部数据）。有时，为了方便在系统调试的过程中队行矩阵的内容进行观察和显示，需要用到有行索引的行矩阵。

|  |
| --- |
| **import** org.apache.spark.mllib.linalg.distributed.{IndexedRow, IndexedRowMatrix, RowMatrix} **import** org.apache.spark.mllib.linalg.{Matrices, Vectors} **import** org.apache.spark.mllib.util.MLUtils **import** org.apache.spark.{SparkConf, SparkContext}  **object** TestLabeledPoint2 {   **def** main(args: Array[String]): Unit = {  **val** conf = **new** SparkConf().setMaster(**"local"**).setAppName(**"TestLabeledPoint2"**)  **val** sc = **new** SparkContext(conf)  **val** rdd = sc.textFile(**"d:\\test\\33.txt"**).map(\_.split(**" "**).map(\_.toDouble))  .map(x=>Vectors.*dense*(x))  .map(x=>**new** IndexedRow(x.toString.hashCode,x))  **val** rm = **new** IndexedRowMatrix(rdd)    *println*(rm.getClass)  *println*(rm.rows.foreach(*println*))   } } |

运行结果如下：

|  |
| --- |
| class org.apache.spark.mllib.linalg.distributed.IndexedRowMatrix  IndexedRow(1811567284,[1.0,2.0,3.0])  IndexedRow(-1162598773,[4.0,5.0,6.0]) |

注：数据还用行矩阵的数据

第一行显示的IndexedRowMatrix实例化后的类型，第二行和第三行显示的是矩阵在计算机中存储的具体内容。

除此之外，IndexedRowMatrix还有转换成其他矩阵的功能，例如：toRowMatrix将其转化成单纯的行矩阵，toCoordinateMatrix将其转化成坐标矩阵，toBlockMatrix将其转化成块矩阵。本节主要介绍第一种坐标矩阵，块矩阵应用较少，这里就不做介绍。

1. **坐标矩阵**

从名称上看，坐标矩阵是一种带有坐标标记的矩阵。事实上也是如此，其中的每一个具体数据都有一组坐标进行标识，其类型格式如下：

|  |
| --- |
| （x:long,y:long,value:Double） |

从格式上看，x和y分别代表标示坐标的坐标轴标号，value是具体内容。X是行坐标，y是列坐标。坐标矩阵一般用于数据比较多且数据较为分散的情形，即矩阵中含0或者某个具体值较多的情况下。

坐标矩阵的用法如下程序：

|  |
| --- |
| **import** org.apache.spark.mllib.linalg.distributed.\_ **import** org.apache.spark.mllib.linalg.{Matrices, Vectors} **import** org.apache.spark.mllib.util.MLUtils **import** org.apache.spark.{SparkConf, SparkContext}  **object** TestLabeledPoint2 {   **def** main(args: Array[String]): Unit = {  **val** conf = **new** SparkConf().setMaster(**"local"**).setAppName(**"TestLabeledPoint2"**)  **val** sc = **new** SparkContext(conf)  **val** rdd = sc.textFile(**"d:\\test\\33.txt"**).map(\_.split(**" "**).map(\_.toDouble))  .map(x=>(x(0).toLong,x(1).toLong,x(2)))  .map(x=>**new** MatrixEntry(x \_1,x \_2,x \_3))  **val** crm = **new** CoordinateMatrix(rdd)   crm.entries.foreach(*println*)   } } |

这里需要提示的RDD语句中最后一句\_1、\_2、\_3是Scala语句中元组参数的序数专用标号。

结果如下：

|  |
| --- |
| MatrixEntry(1,2,3.0)  MatrixEntry(4,5,6.0) |

## **MLlib数理统计基本概念**

### 基本统计量

数理统计中，基本统计量包括数据的平均值、方差，这是一组求数据统计量的基本内容。在MLlib中，统计量的计算主要用到Statistics类库，如下表：

|  |  |
| --- | --- |
| 类型名称 | 释义 |
| colStats | 以列为基础计算统计量的基本数据 |
| chiSqTest | 对数据集内的数据进行皮尔逊距离计算，根据参数的不同，返回值格式有差异 |
| corr | 对两个数据集进行相关系数计算，根据参量的不同，返回值格式有差异 |

从表中可看出，Statistics类中不同的方法代表不同的统计量的求法

### 统计量的基本数据

colstats是Statistics类计算基本统计量的方法，这里需要注意的是，其工作和计算是以列为基础进行计算，调用不同的方法获得不同的统计量值，其方法如下表所示：

|  |  |
| --- | --- |
| 方法名称 | 释义 |
| count | 行内数据个数 |
| Max | 最大数值单位 |
| Mean | 最小数值单位 |
| normL1 | 欧几里德距离 |
| normL2 | 曼哈顿距离 |
| numNonzeros | 不包含0值的个数 |
| variance | 方差 |

在这里需要求数据的均值和方差，首先在硬盘上建立名为testSummary.txt的文件，加入如下数据:

|  |
| --- |
| 1 2  2 3  3 4  4 5  5 6 |

求数据的均值和方差

|  |
| --- |
| **import** org.apache.spark.mllib.linalg.Vectors **import** org.apache.spark.mllib.stat.Statistics **import** org.apache.spark.{SparkConf, SparkContext}  **object** TestSummary {  **def** main(args: Array[String]): Unit = {  **val** conf = **new** SparkConf().setAppName(**"TestSummary"**).setMaster(**"local"**)  **val** sc = **new** SparkContext(conf)  **val** rdd = sc.textFile(**"d:\\test\\testSummary.txt"**).map(\_.split(**" "**).map(\_.toDouble))  .map(line => Vectors.*dense*(line))  **val** summary = Statistics.*colStats*(rdd)  *println*(summary.mean)  *println*(summary.variance)   }  } |

结果如下：

|  |
| --- |
| [3.0,4.0]  [2.5,2.5] |

从结果可以看到summary的实例将列数据的内容计算并存储，供下一步的数据分析使用。

### 距离计算

除了一些基本统计量的计算，colStats方法中还包括两种距离的计算，分别是normL1和normL2，代表欧几里得距离和曼哈顿距离。这两种距离主要用以表达数据集内部长度的常用算法。具体见代码

|  |
| --- |
| **import** org.apache.spark.mllib.linalg.Vectors **import** org.apache.spark.mllib.stat.Statistics **import** org.apache.spark.{SparkConf, SparkContext}  **object** TestSummary {  **def** main(args: Array[String]): Unit = {  **val** conf = **new** SparkConf().setAppName(**"TestSummary"**).setMaster(**"local"**)  **val** sc = **new** SparkContext(conf)  **val** rdd = sc.textFile(**"d:\\test\\testSummary.txt"**).map(\_.split(**" "**).map(\_.toDouble))  .map(line => Vectors.*dense*(line))  **val** summary = Statistics.*colStats*(rdd)  *println*(summary.normL1)  *println*(summary.normL2)   }  } |

结果如下：

|  |
| --- |
| [15.0,20.0]  [7.416198487095663,9.486832980505138] |

### 相关系数计算

相关系数里面包括皮尔逊系数和斯皮尔曼相关系数，在代码时间中我分别用在了两组数据的相关系数计算和单个数据集之间相关系数的计算，具体使用参考代码部分

|  |
| --- |
| **import** org.apache.spark.mllib.linalg.Vectors  **import** org.apache.spark.mllib.stat.Statistics **import** org.apache.spark.{SparkConf, SparkContext}  **object** TestSummary {  **def** main(args: Array[String]): Unit = {  **val** conf = **new** SparkConf().setAppName(**"TestSummary"**).setMaster(**"local"**)  **val** sc = **new** SparkContext(conf)  **val** rddX = sc.textFile(**"D:\\test\\testCorrectX.txt"**).flatMap(\_.split(**' '**).map(\_.toDouble))   **val** rddY = sc.textFile(**"D:\\test\\testCorrectY.txt"**).flatMap(\_.split(**' '**).map(\_.toDouble))  **val** rdd = sc.textFile(**"D:\\test\\testCorrectX.txt"**).flatMap(\_.split(**' '**).map(\_.toDouble))  .map(line=>Vectors.*dense*(line))  **val** correlation:Double = Statistics.*corr*(rddX, rddY)//皮尔逊相关系数  val correlationS:Double = Statistics.corr(rddX,rddY,**"spearman"**)//斯皮尔曼相关系数  **val** correlationf = Statistics.*corr*(rdd,**"spearman"**)  *println*(correlation)  println(correlationS)  Println(correlationf)   }  } |

根据皮尔逊相关系数和斯皮尔曼相关系数计算法，最终计算结果如下：

|  |
| --- |
| 0.9999999999999998  0.9999999999999998 |

### 分层抽样

分层抽样是一种数据提取算法，先将总体的单位按照某种特征分为若干次级总体（层），然后再从每一层内进行单纯的随机取样，组成一个样本的统计学计算方法。这种方法以前常常用于数据量比较大，计算处理非常不方便的情况。

  一般抽样时，将总体分成互不交叉的层，按照一定的比例，从各层次独立的抽取一定数量的个体，将各层次取出的个体合在一起作为样本，这种抽样方法是一种分层抽样。

  在MLlib中，使用Map作为分层抽样的数据标记，一般情况下，Map的构成是[key,value]格式，key作为数据组，而value作为数据标签进行处理。下面建立一个数据集，如下：

|  |
| --- |
| aa  bb  cc  aaa  bbb  ccc |

对数据集的分层抽样代码，如下：

|  |
| --- |
| **import** org.apache.spark.{SparkConf, SparkContext}  **object** TestSummary {  **def** main(args: Array[String]): Unit = {  **val** conf = **new** SparkConf().setAppName(**"TestSummary"**).setMaster(**"local"**)  **val** sc = **new** SparkContext(conf)  **val** data = sc.textFile(**"D:\\test\\44.txt"**).map(row =>{  *//判断字符数* **if** (row.length == 3)  (row, 1)  **else** (row, 2)  })  data.foreach(*println*)  *println*(**"SampleByKey:"**)  *//设定抽样格式* **val** fractions: Map[String, Double] = *Map*(**"aa"**->2)  *//计算抽样样本* **val** approxSample = data.sampleByKey(withReplacement =**false**, fractions, 0)  approxSample.foreach(*println*)    }  } |

结果如下：

|  |
| --- |
| (aa,2)  (bb,2)  (cc,2)  (aaa,1)  (bbb,1)  (ccc,1)  SampleByKey:  (aa,2) |

### 假设检验

常用的假设检验方法是卡方检验。卡方检验是一种常用的假设检验方法，能够较好的对数据集之间的拟合度、相关性和独立性进行验证。MLlib中规定常用的卡方检验使用的数据集一般为向量和矩阵。

                                                     假设检验的常用术语介绍

|  |  |
| --- | --- |
| 自由度 | 总体参数估计量中变量值独立自由变化的数目 |
| 统计量 | 不同方法下的统计量 |
| P值 | 显著性差异指标 |
| 方法 | 卡方检验使用方法 |

卡方检验使用了皮尔逊计算法对数据集进行计算，得到最终结果P值，一般情况下，P<0.05是指数据集不存在显著性差异。

详细操作请看代码部分。

|  |
| --- |
| **package** org.apache.spark.examples.streaming  **import** org.apache.spark.mllib.linalg.{Matrices, Vectors} **import** org.apache.spark.mllib.stat.Statistics  **object** testChiSq {   **def** main(args: Array[String]): Unit = {  **val** vd = Vectors.*dense*(1,2,3,4,5)  **val** vdResult = Statistics.*chiSqTest*(vd)  *println*(vdResult)  *println*(**"-------------------------------------------"**)  **val** mtx = Matrices.*dense*(3,2,*Array*(1,3,5,2,4,6))  **val** mtxResult = Statistics.*chiSqTest*(mtx)  *println*(mtxResult)  }  } |

结果如下：

|  |
| --- |
| Chi squared test summary:  method: pearson  degrees of freedom = 4  statistic = 3.333333333333333  pValue = 0.5036682742334986  No presumption against null hypothesis: observed follows the same distribution as expected..  -------------------------------------------  Chi squared test summary:  method: pearson  degrees of freedom = 2  statistic = 0.14141414141414144  pValue = 0.931734784568187  No presumption against null hypothesis: the occurrence of the outcomes is statistically independent.. |

### 随机数

随机数是统计分析中常用的一些数据文件，一般用来检验随机算法和执行效率等。RandomRDDS类是随机数生成类，详细看代码部分

|  |
| --- |
| **import** org.apache.spark.mllib.linalg.Vectors **import** org.apache.spark.mllib.random.RandomRDDs **import** org.apache.spark.mllib.stat.Statistics **import** org.apache.spark.{SparkConf, SparkContext}  **object** TestSummary {  **def** main(args: Array[String]): Unit = {  **val** conf = **new** SparkConf().setAppName(**"TestSummary"**).setMaster(**"local"**)  **val** sc = **new** SparkContext(conf)  **val** randomNum = RandomRDDs.*normalRDD*(sc, 100)  randomNum.foreach(*println*)     } |