Scala Handbook

王震宇

南 开 大 学
统 计 与 数 据 科 学 学 院

天　　津

前 言

《分布式存储与计算》课程涉及两部分内容，分别为Scala与PySpark，该教程仅包含Scala的相关语法及常用算法。本书为笔者2022年选修此课程时整理的笔记，主要参考下列文献：

- 陈大川老师备课笔记
- 《分布式存储与计算》 冯兴东(上海财经大学出版社)

尽管课程内容十分优质，但笔者学艺不精，对许多内容浅尝辄止，考试也是不尽人意。望读者加勉，真正领悟分布式存储与计算的思想。

王震宇

2023年2月25日

分布式存储与计算

> 王震宇
>
> 统计与数据科学学院

第一次课

- 数据类型：分为结构化数据和非结构化数据。前者占10%左右，主要指存储在关系数据库中的数据。
- 大数据的计算模式：MapReduce，Spark属于批处理计算，指大规模数据的批量处理。
- 大数据处理架构：Hadoop
    - Hadoop的核心是：
        - 分布式文件系统（Hadoop Distributed File System, HDFS）

          为海量的数据提供存储！
        - MapReduce

          为海量的数据提供计算！
- Hadoop将并行计算过程高度的抽象到了两个函数Map和Reduce上，允许用户在不了解分布式系统底层细节的情况下开发并行应用程序。
- MapReduce的核心思想：分而治之。将输入的数据集分为若干独立的数据块，分发给主节点下的各个分节点并行完成，最后整个各个节点的中间节点得到最终结果。
- Spark
    - 目的：改善MapReduce
    - MapReduce是Spark运算系统的严格子集
        - 理解：Spark有first、take等函数；而Hadoop只有Map和Reduce两个函数。所以是"严格子集"
- Hadoop的缺点
    - 表达能力有限：计算只有Map和Reduce
    - 硬盘读写开销大：每次执行都需要从硬盘读取数据，每次计算完成需要将中间结果写入到磁盘中，IO开销大
    - 延迟高：前一个任务完成之前，其余任务无法开始
- Spark与Hadoop的对比：
    - Spark的计算模式属于MapReduce，但不局限于Map和Reduce，还提供了多种数据集操作类型
    - Spark将中间结果直接放到内存中
    - Spark基于DAG任务调度执行机制
- RDD：弹性分布式数据集，是分布式内存的一个抽象概念，提供了一种高度受限的共享内存模型。
    - 高度受限：只可读！
    - 一个RDD就是一个分布式对象的集合，本质上是一个只读的分区记录集合。
        - 理解：每个RDD有多个分区，每个分区就是一个数据集片段，各个分区并行计算。（可以简单理解为一个"数据类型"）
- 由于RDD只可读，所以只能通过在其他RDD上执行确定的转换操作，从而创建得到新的RDD！
- DAG：有向无环图，反映RDD之间的依赖关系
    - 理解：DAG是RDD在经过一系列的转换后，各个RDD之间的内在联系，有了DAG之后，这样才能知道最后的计算该怎么算！
- RDD操作的分类：
    - 行动（Action）：接收RDD但返回一个值或者结果
    - 转换（Transformation）：接收RDD并返回RDD（如map、filter等）
- RDD的执行过程：
    - 每次转换，都会产生新的RDD，供下一个转换使用
    - 最后一个RDD经过"行动"操作进行处理，得到一个值
    - RDD的惰性调用：
        - 真正的计算发生在RDD的"行动"操作，对于转换操作，Spark只记录转换操作应用的数据集以及生成RDD的轨迹，即相互依赖关系（即DAG），不会触发真正的计算！

第二次课

在Scala安装成功后，我们可以直接在命令行运行代码；Windows系统打开命令行的方法：Win+R，输入cmd即可。

同时，也可以下载jupyter后，安装Scala语言的内核，从而更方便地进行代码调试；

只需在命令行输入以下两句：pip install -i https://pypi.tuna.tsinghua.edu.cn/simple spylon-kernel

python -m spylon_kernel install

开始Scala语言编程，输入：

```
spark-shell
```

- **值与变量**
  - 值(value)是不可变的，有对应类型的存储单元；申请一个值使用 `val`
  - 变量(variable)表示一个唯一的标识符，对应一个已分配或保留的内存空间，这一空间的内容是动态的；申请一个变量使用 `var`
  - 申请一个字符值”math”: `val a = “math”` （而且只能是双引号）
  - 注意：数字和字符串之间是不能在同一个变量上转换的，整数型和双精度型的变量也不可以在同一个变量上转换
    - 虽然不能直接赋值转换，但可以用 `.toDouble`, `.toString` 函数进行转换后使用
  - 申请一个双精度的变量=3.1415926: `var b=3.1415926`
  - 将一个整数值转换成双精度类型： `val c=5; c.toDouble`
  - 将一个双精度类型的变量转换成整数值： `b.toInt` (把小数部分**抹去**，不是四舍五入)
  - `Val a =“1234.5”; a.toDouble` 可以将a变成一个双精度的数，但是 `a.toInt` 会报错；（字符型与数字之间的转换实例）
  - 常见的数据类型：Int [-2^31, 2^31-1], Long [-2^63, 2^63-1], Double, Boolean {true, false（注意都是小写）}, String 字符串
  - 如果要进行复杂的数学运算，需要引用相关的包:
    - `import scala.math._`
    - `min(20,4)`
    - `pow(2,0.5)`
- 表达式和条件表达式
  - 表达式：
    - 表达式是一个值的代码单元
    - 多个表达式用大括号即可构造一个表达式块，值得注意的是，表达式有自己的作用域，可以作用于所属表达式块中的局部（或全局）值和变量
    - **表达式块中最后一个表达式即为整个表达式块的返回值**
    - **Example:** `{ val a = 5.0; val b=4; b}`
    - **Example:** `val c = { val a = 5.0; val b=4}`
  - 条件表达式 if-else 语句
    - 语法： `if(<Boolean expression 判断语句>)<expression>`

      ```
      val A_age = 15; val B_age = 20;
      val Older_age = if(A_age>B_age) A_age else B_age
      Older_age: Int = 20
      ```

      (注意：整数/整数结果仍然是整数，如 `15/20==0`，否则改成双精度 `a.toDouble/b.toDouble`，如果一个为双精度，则结果也为双精度！自动转换。）
    - 如果if表达式要执行的有很多条语句，可以用大括号来构建表达式块；

      注意 `println` 为输出（类似于 `printf`）

```
val a = 15; val b = 20
a: Int = 15
b: Int = 20
scala> if(a>b)
     | {println(a+b)
     | println(a-b)
     | }else{
     | println(a*b)
     | println(a/b)}
300
0
```

- 循环
  - For循环: `for(<identifier变量名> <- <iterator循环域>) [yield] [<expression 循环体>]`

    （可以在条件表达式里加入一个“if语句”）

```
scala> for(t <- 0 until 4 if t%4==0)
     | {println(t)}
0
```

```
scala> for(t <- 0 to 4 if t%4==0)
     | {println(t)}
0
4
```

注意: `until` 是 <，而 `to` 是 ≤！ 注意区分。（为什么要创建两个呢？因为 `until` 可以用在向量迭代里——某些从0开始）

- 如果使用 `yield`，输出的是一个向量，如果不适用 `yield`，结果按次序依次输出；
- `%`：取余数
- `0 until 4`：取0,1,2,3
- `0 to 4`：取0,1,2,3,4

双重循环（此时条件语句一定要**用大括号**，且一定要**换行**）：

```
scala> for{x<-0 until 3
     | y<-0 until 3}
     | {println(x)
     | println(y)}
0
0
0
1
0
2
1
0
1
1
1
2
2
0
2
1
2
2
```

（注：`print` 也可以，但是每次迭代输出不

换行，而 `println` 就会自动换行）

- ```
  scala> for( x<- 0 until 3; y <- 0 until 3) println("x="+x+", y="+y+", x*y="+(x*y))
  x=0, y=0, x*y=0
  x=0, y=1, x*y=0
  x=0, y=2, x*y=0
  x=1, y=0, x*y=0
  x=1, y=1, x*y=1
  x=1, y=2, x*y=2
  x=2, y=0, x*y=0
  x=2, y=1, x*y=2
  x=2, y=2, x*y=4
  ```

  - 如果是两边一个是字符串一个是数字, `+` 号是用来连接字符串和数字的，数字也会自动转换成字符串；如果两边都是整数或双精度数， `+` 号是用来相加运算的

- While循环 `while(<Boolean expression 判断语句>) [<expression 循环体>]`

  例如：

  ```
  scala> while(a<20)
       | {println("value of a:"+a)
       | a = a+1}
  value of a:10
  value of a:11
  value of a:12
  value of a:13
  value of a:14
  value of a:15
  value of a:16
  value of a:17
  value of a:18
  value of a:19
  ```

  此时a的值应该为20！

  ```
  scala> a
  res7: Int = 20
  ```

- Do-While循环 `do [<expression 循环体>] while(<Boolean expression 判断语句>)`

  ```
  scala> var a =10
  a: Int = 10

  scala> do{println("value of a:"+a)
       | a = a + 1
       | }while(a<20)
  value of a:10
  value of a:11
  value of a:12
  value of a:13
  value of a:14
  value of a:15
  value of a:16
  value of a:17
  value of a:18
  value of a:19
  ```

  此时a的值也应该为20！

  ```
  scala> a
  res10: Int = 20
  ```

- 函数：
  - 有一个或多个输入参数
  ```

- 只使用输入参数完成计算
- 对于相同的输入总返回相同的值
- 不使用影响函数之外的任何数据
- 只返回**一个值**（但是这个值可能是向量）

  注意：一定是返回一个值，不能返回一个操作！（例如：`println（x+y）`无法返回！）
- 不受函数之外的任何数据的影响
- 语法：`def <identifier 函数名>（<identifier 自变量名>：<type：自变量类型>[，……]）：<type: 输出变量的类型>=<expression 函数体>`

  例如：

  ```
  def power(x:Int,n:Int):Long={函数体语句}
  ```

  ```
  scala> def test(x:Int,y:Int):Long={
       | x+y}
  test: (x: Int, y: Int)Long
  ```

测试：

```
scala> test(1,3)
res11: Long = 4
```

注：可以将函数理解为一种特殊变量，所以用`def`（名字、变量）：类型=，而在（名字、变量）内部也是（名字：类型）

- 递归函数：函数体内会引用自身的函数

  ```
  scala> def power(x:Int,n:Int):Long={
       | if(n>0){
       | power(x,n-1)*x
       | }else 1}
  power: (x: Int, n: Int)Long
  ```

  实例测试：

  ```
  scala> power(2,10)
  res12: Long = 1024
  ```

  - Scala的函数中，输入参数和输出结果都可能是向量
- 创建RDD的方法

  创建向量的方法：

  ```
  scala> var numbers = List(1,2,3,4,5,6,7,8,9,10)
  numbers: List[Int] = List(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
  ```

  此处说明了返回值类型是`List[Int]`，在定义函数时会用到！

  类似的，`List`可以换成`Array`

  - `sc.parallelize()`：用内部数据创建RDD

    ```
    scala> val rddExample = sc.parallelize(numbers)
    rddExample: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[0] at parallelize at
    <console>:24
    ```

    如果此时输入`rddExample`呢？

    ```
    scala> rddExample
    res13: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[0] at parallelize at
    <console>:24
    ```

- `data.first`: 返回RDD的第一个元素

  ```
  scala> rddExample.first
  res14: Int = 1
  ```

- `data.take(n)`: 返回RDD的前n个元素

  ```
  scala> rddExample.take(5)
  res15: Array[Int] = Array(1, 2, 3, 4, 5)
  ```

如果要返回第二个元素呢?

```
scala> rddExample.take(5)(1)
res16: Int = 2
```

注意：array从0开始计数!

- `data.collect()`: 打印整个RDD的所有元素（注意不是`take()`了！而且`collect`后面有括号！）

  如果想知道array的元素个数:

  ```
  scala> rddExample.collect().size
  res17: Int = 10
  ```

  但是要注意：`size`只能对array用，而不能在这里直接对rddExample用!

  - `sc.textFile()`: 从外部读取数据集
    - 将文件按照**字符串**的形式读入，每一行为一个RDD元素

      ```
      scala> val data = sc.textFile("C:/Users/zywang/Desktop/WineData.csv")
      data: org.apache.spark.rdd.RDD[String] = C:/Users/zywang/Desktop/WineData.csv
      MapPartitionsRDD[2] at textFile at <console>:23
      ```

      注意：是/!

      读入csv的第一行:

      ```
      scala> data.first
      res18: String = 7.4,0.7,0,1.9,0.076,11,34,0.9978,3.51,0.56,9.4,5
      ```

`map`和`reduce`的用法介绍:

```
scala> val NewData = data.map(line => line.split(",").map(_.toDouble))
NewData: org.apache.spark.rdd.RDD[Array[Double]] = MapPartitionsRDD[3] at map at <console>:23
```

此处的line为任意一个变量名指代RDD中的任意一行

`=>`指对RDD进行转换，等号后为转换结果，此处得到一个字符串向量

`map`函数此处将字符串里的每一个元素转换为双精度数值

```
scala> val Result = NewData.map(z => (z(0)+z(1)+z(2),z.size))
Result: org.apache.spark.rdd.RDD[(Double, Int)] = MapPartitionsRDD[4] at map at <console>:23
```

此处`(z(0)+z(1)+z(2),z.size)`为一个元组。即将每一行转换为一个元组!

```
scala> val Screening = Result.filter(s => s._1 >4).map(s =>(s._1,s._2*2))
Screening: org.apache.spark.rdd.RDD[(Double, Int)] = MapPartitionsRDD[6] at map at <console>:23
```

调用元组利用`_num`，且元组的序号从1开始!

```
scala> val FinalResult = Screening.reduce((x,y) => (x._1+y._1,x._2+y._2))
FinalResult: (Double, Int) = (14580.375000000004,38376)
```

此处的 `(x,y)` 代表两个元组！

前面的 `map` 操作都**只记录、不执行**！直到最后的 `reduce` 语句才执行！而 `first`， `take` 语句均为action语句！

`sc.textFile`,`map`,`filter` 和 `reduce` 的用法

- `sc.parallelize`
- `sc.textFile`：
  - `sc.textFile` 命令是用来处理字符串(string)对象的

    所以地址要加**双引号**
  - 其对象的每一个记录（或元素）是文本文件的一行
  - 如果文本文件是csv文件，那么它的每一行都是由逗号分隔的若干字符串
- `map(s => s.func)`
  - 对RDD的每一个元素s都施加函数func的作用，将返回值构成新的RDD
  - 因为map函数对RDD中的每个元素都施加了相同的作用，所以它可以被应用在分布式计算中。
  - 用法：
    - `data.map(s => s.func)`
    - `data.map(_.func)`
- `reduce( (x,y) => func(x,y))`
  - 作用：并行整合RDD中所有数据x,y，这里的函数 `func` 必须是**可交换可结合**的

    x,y，还有结果的数据类型应该完全相同！

    可交换可结合：加法和乘法
  - x,y的数据类型是相同， `func` 结果的数据类型与x,y的数据类型也应该是相同的
  - 这里的x或y可以是一个数，也可以是一个向量(DenseVector或Array)，也可以是一个tuple(元组)数据
  - 这里的func函数需要保证生成的结果 `func(x,y)` 和x,y为同样的数据类型，如果是向量或元组，元素的个数也必须相同（一般而言， `func` 函数都是类似加法的运算，因为加法具有交换性）
  - 执行的流程为：从RDD中任意选取两个元素x和y，对他们施加 `func` 的作用，得到 `func(x,y)`，这时因为 `func(x,y)` 与x,y具有相同的数据类型，因此用 `func(x,y)` 替换原来RDD中的x,y，形成新的RDD，再从头执行之前的操作，直到RDD中只剩一个元素为止。
  - 因为 `reduce` 函数在整合RDD时具有任意性，所以它可以被用在分布式计算中。
    - Example 1:

      ```
      scala> val data = Array(Array(1,2), Array(3,4), Array(5,6), Array(7,8))
      data: Array[Array[Int]] = Array(Array(1, 2), Array(3, 4), Array(5, 6), Array(7, 8))

      scala> val rdd = sc.parallelize(data)
      rdd: org.apache.spark.rdd.RDD[Array[Int]] = ParallelCollectionRDD[7] at parallelize at
      <console>:24

      scala> rdd.reduce((x,y)=> Array(x(0)+y(0), x(1)+y(1))) //由于x,y为Array，因此结果也必须是
      Array
      res19: Array[Int] = Array(16, 20)
      ```
    - Example 2:

```
scala> val data = Array((1,2), (3,4), (5,6), (7,8))
data: Array[(Int, Int)] = Array((1,2), (3,4), (5,6), (7,8))

scala> val rdd = sc.parallelize(data)
rdd: org.apache.spark.rdd.RDD[(Int, Int)] = ParallelCollectionRDD[8] at parallelize at
<console>:24

scala> rdd.reduce((x,y)=>(x._1 + y._1, x._2 + y._2))
res20: (Int, Int) = (16,20)
```

- Example 3：计算一个样本的3阶矩：

```
scala> val numbers = List(1,2,3,4,5,6,7,8,9,10)
numbers: List[Int] = List(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)

scala> val rdd = sc.parallelize(numbers)
rdd: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[9] at parallelize at
<console>:24

scala> val rdd3 = rdd.map(s => s*s*s)
rdd3: org.apache.spark.rdd.RDD[Int] = MapPartitionsRDD[10] at map at <console>:23

scala> val rdd3_sum = rdd3.reduce((x,y)=> x+y)
rdd3_sum: Int = 3025

scala> val rdd3_moment = rdd3_sum.toDouble/rdd3.collect().size.toDouble
rdd3_moment: Double = 302.5
```

注意：此处不能直接`rdd.reduce((x,y) => x*x*x + y*y*y)`，因为在reduce的过程中，结果会被逐次立方。

- `rdd3.collect().size`：计算rdd3这个RDD中所有元素的个数，即样本容量

```
scala> rdd3.collect().size
res21: Int = 10
```

- `filter(s => func(s))`
  - 对RDD中的每个元素s施加判断函数`func`的作用，将`func(s)`为true的结果对应的元素s组成新的RDD

    例如：`s_1>4`

第三次课

Breeze 程序包

- 创建向量，矩阵以及简单的计算
  - 调用Breeze程序包：`import breeze.linalg._`

    理解：breeze是线性代数库

    linalg是库中的package，代表linear algebra
  - 创建一个长度为n的双精度类型的零向量：`DenseVector.zeros[Double](n)`

```
scala> val a = DenseVector.zeros[Double](10)
a: breeze.linalg.DenseVector[Double] = DenseVector(0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
0.0, 0.0)
```

    稀疏矩阵将非零元以`(x,y,value)`的形式存储，节省存储空间

    默认生成列向量！
  - 创建一个一般性的DenseVector：`DenseVector(Array)`

- 向量a的转置：`a.t`

  ```
  scala> a.t
  res22: breeze.linalg.Transpose[breeze.linalg.DenseVector[Double]] =
  Transpose(DenseVector(0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0))
  ```

- 利用Breeze包创建的向量都是列向量，因此我们可以通过`a.t`得到行向量的形式；
- **将a的元素从第一个到最后一个分别加上它所处位置的坐标（从0开始！）**：

  ```
  scala>  (1 to a.length) //其常用于DenseVector的索引，Array的构建等
  res2: scala.collection.immutable.Range.Inclusive = Range 1 to 10
  ```

  变成了Range类型，要加`toArray`！

  ```
  scala>  (1 to a.length).toArray
  res3: Array[Int] = Array(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
  ```

  如何将Array的每一个元素转换为双精度类型？不能直接`toDouble`，可以用`map`函数！

  ```
  scala> (1 to a.length).toArray.map(_.toDouble)
  res4: Array[Double] = Array(1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0, 10.0)
  ```

  - 返回向量a的长度：`a.length` 或 `a.size`

    输出结果：

    ```
    scala> a.size
    res5: Int = 10

    scala> a.length
    res6: Int = 10
    ```

    注意：DenseVector也是从0开始计数：

    ```
    scala> a(0)
    res7: Double = 0.0
    ```

  所以答案应该是：

  ```
  scala> DenseVector((0 until a.length).toArray.map(_.toDouble))+a
  res8: breeze.linalg.DenseVector[Double] = DenseVector(0.0, 1.0, 2.0, 3.0, 4.0, 5.0, 6.0,
  7.0, 8.0, 9.0)
  ```

  - 按元素给向量赋值：要使用“:=”；

    > 可以理解为，一个数对向量中的多个位置赋值！

  - 如果需要赋值的元素个数超过一个，就需要使用“:=”；

    可以理解为：一次修改多个元素，例如将DenseVector的1 to 3修改为7，8，9，则应该为`a(1 to 3):=DenseVector(7,8,9)`，而且在用向量修改的时候，也要用DenseVector！

    注意：只有DenseVector可以用(a to b)进行索引，Array不行！

  - 把其他类型的向量转换成Array: `.toArray`

- 创建一个n*m的整数型零矩阵：`DenseMatrix.zeros[Int](n,m)`

  ```
  scala> val b = DenseMatrix.zeros[Int](3,5)
  b: breeze.linalg.DenseMatrix[Int] =
  0  0  0  0  0
  0  0  0  0  0
  0  0  0  0  0
  ```

- 抽取b矩阵中的第三行的所有数据：`b(2,::)`

```
  scala> b(2,::)
res10: breeze.linalg.Transpose[breeze.linalg.DenseVector[Int]] = Transpose(DenseVector(0, 0, 0, 0, 0))
```

注意：是两个冒号！

- 给b矩阵的第一列赋值（1,2,3）：

```
  scala> b(::,0):=DenseVector(1,2,3)
res11: breeze.linalg.DenseVector[Int] = DenseVector(1, 2, 3)
```

思考：b不是值吗，为什么能改变呢？

答案：实际上，对矩阵b的内部进行赋值时，b的地址并没改变。

- 让矩阵b中第2列到第四列和第二行到第三行对应的矩阵块元素全为4：

  - ```
    scala> b(1 to 2, 1 to 3) := DenseMatrix((4,4,4),(4,4,4))
    res12: breeze.linalg.DenseMatrix[Int] =
      4  4  4
    4  4  4
    ```

    注意：这里的行列也是从0开始

    同时注意：索引只有一个括号！

    同样地，创建一个一般性的DenseMartix：`DenseMatrix(Array_1,...Array_n)`，其中，Array_i代表第i行

  - ```
    scala> b(1 to 2, 1 to 3) := 4
    res13: breeze.linalg.DenseMatrix[Int] =
    4  4  4
    4  4  4
    ```

  如果要将b的第三行变为(1,2,3,4,5)，则用：

  ```
    scala> b(2,::):=DenseVector(1,2,3,4,5).t
  res14: breeze.linalg.Transpose[breeze.linalg.DenseVector[Int]] =
  Transpose(DenseVector(1, 2, 3, 4, 5))
  ```

  一定要注意：DenseVector默认生成的是列向量，所以对行进行操作时，务必要加转置！

  也说明：可以直接用DenseVector后括号加数字

- 将矩阵拉直为向量：对DenseMatrix矩阵直接“`.toDenseVector`”即可

  先创建一个矩阵：

  ```
  scala> val a =DenseMatrix((1,2,3),(1,2,3))
  a: breeze.linalg.DenseMatrix[Int] =
  1  2  3
  1  2  3
  ```

  再转换为向量：

  ```
  scala> a.toDenseVector
  res16: breeze.linalg.DenseVector[Int] = DenseVector(1, 1, 2, 2, 3, 3)
  ```

- 创建一个长度为n的双精度类型的元素全为1向量：`DenseVector.ones[Double](n)`
- 创建一个n*m的双精度类型的元素全为1的矩阵：`DenseMatrix.ones[Double](n,m)`

  如果创建一个全为2的矩阵：

```
scala> var b = DenseMatrix.ones[Double](3,5)
b: breeze.linalg.DenseMatrix[Double] =
1.0  1.0  1.0  1.0  1.0
1.0  1.0  1.0  1.0  1.0
1.0  1.0  1.0  1.0  1.0

scala> b:=2.0
res21: breeze.linalg.DenseMatrix[Double] =
2.0  2.0  2.0  2.0  2.0
2.0  2.0  2.0  2.0  2.0
2.0  2.0  2.0  2.0  2.0
```

（注意：是:=，同时Double是2.0）

- 创建一个任意常数向量：`DenseVector.fill(n,a)`

  n为元素个数，a为常数值

```
scala> DenseVector.fill(10,2)
res22: breeze.linalg.DenseVector[Int] = DenseVector(2, 2, 2, 2, 2, 2, 2, 2, 2, 2)
```

- 固定步长向量（整数类型）：`DenseVector.range(start, stop, step)`

```
scala> DenseVector.range(1,11,2)
res23: breeze.linalg.DenseVector[Int] = DenseVector(1, 3, 5, 7, 9)
```

要注意：stop位置是取不到的！

- 固定步长向量（双精度类型）：`Vector.rangeD(start, stop, step)`

```
scala> Vector.rangeD(1.0,9.0,2.0)
res24: breeze.linalg.Vector[Double] = DenseVector(1.0, 3.0, 5.0, 7.0)
```

- 固定长度向量：`linspace(start, stop, num)`（num是一个整数，start,stop为Double类型）

```
scala> linspace(0,1.0,9)
res25: breeze.linalg.DenseVector[Double] = DenseVector(0.0, 0.125, 0.25, 0.375, 0.5, 0.625,
0.75, 0.875, 1.0)
```

注意：一共是num个数字，均分

- 生成单位矩阵：`DenseMatrix.eye[Double](n)`

```
scala> DenseMatrix.eye[Double](5)
res26: breeze.linalg.DenseMatrix[Double] =
1.0  0.0  0.0  0.0  0.0
0.0  1.0  0.0  0.0  0.0
0.0  0.0  1.0  0.0  0.0
0.0  0.0  0.0  1.0  0.0
0.0  0.0  0.0  0.0  1.0
```

- 用一个向量生成一个对角矩阵：`diag(DenseVector(1,2,3,4,5))`

  先生成一个对角元素的向量：

```
scala> val a = DenseVector(1,2,3,4,5)
a: breeze.linalg.DenseVector[Int] = DenseVector(1, 2, 3, 4, 5)
```

再生成以该向量为对角元的对角阵：

```
scala> val A = diag(a)
A: breeze.linalg.DenseMatrix[Int] =
1  0  0  0  0
0  2  0  0  0
0  0  3  0  0
0  0  0  4  0
0  0  0  0  5
```

- 抽取一个矩阵A中的对角线元素：`diag(A)`

```
scala> diag(A)
res29: breeze.linalg.DenseVector[Int] = DenseVector(1, 2, 3, 4, 5)
```

作用在向量上，则生成对角矩阵；作用在矩阵上，则提取对角元素！

- 矩阵的合并：

  横向合并：`horzcat`

```
scala> val a= DenseMatrix((1,2,3),(3,4,5))
a: breeze.linalg.DenseMatrix[Int] =
1  2  3
3  4  5

scala> DenseMatrix.horzcat(a,a)
res32: breeze.linalg.DenseMatrix[Int] =
1  2  3  1  2  3
3  4  5  3  4  5
```

  因为相当于创建一个新矩阵，所以当然得加`DenseMatrix`.

  纵向合并：`vertcat`

```
scala> DenseMatrix.vertcat(a,a)
res33: breeze.linalg.DenseMatrix[Int] =
1  2  3
3  4  5
1  2  3
3  4  5
```

- tabulate函数：

对于向量：

```
scala> DenseVector.tabulate(5){i =>i*i}
res35: breeze.linalg.DenseVector[Int] = DenseVector(0, 1, 4, 9, 16)
```

对于矩阵：

```
scala> DenseMatrix.tabulate(3,4){case(i,j) =>i+j}
res36: breeze.linalg.DenseMatrix[Int] =
0  1  2  3
1  2  3  4
2  3  4  5
```

相当于tabulate对所在位置的索引进行变换，从而构造矩阵！

比较巧妙的用法：

```
scala> val a = Array(1,2,3,4,5)
a: Array[Int] = Array(1, 2, 3, 4, 5)

scala> DenseMatrix.tabulate(3,4){case(i,j) =>a(i)*a(j)}
res76: breeze.linalg.DenseMatrix[Int] =
1  2  3  4
2  4  6  8
3  6  9  12
```

一定要注意：整型DenseVector没有索引！即不能`a(i)`！但Double型DenseVector具有索引。

其中，a是一个向量，这样就可以从只对索引计算→对数值计算

- 向量a和b按元素相乘：`a*:*b`

  先定义a,b两个矩阵：

  ```
  scala> val a= DenseMatrix((1,2,3),(3,4,5))
  a: breeze.linalg.DenseMatrix[Int] =
  1  2  3
  3  4  5

  scala> val b =DenseMatrix((2,3),(1,4),(5,6))
  b: breeze.linalg.DenseMatrix[Int] =
  2  3
  1  4
  5  6
  ```

  按元素相乘：

  ```
  scala> a*:*b.t
  res77: breeze.linalg.DenseMatrix[Int] =
  2  2   15
  9  16  30
  ```

  按元素相除：（也可以直接用`/`，因为没有定义矩阵相除）

  ```
  scala> a/:/b.t
  res78: breeze.linalg.DenseMatrix[Int] =
  0  2  0
  1  1  0
  ```

  - 矩阵a乘以矩阵b：`a*b`（要保证a的列数等于b的行数）

    ```
    scala> a*b
    res79: breeze.linalg.DenseMatrix[Int] =
    19  29
    35  55
    ```

    计算最值：（`max`,`min`,`argmax`,`argmin`）

    ```
    scala> max(a)
    res80: Int = 5
    ```

    ```
    scala> argmax(a)
    res81: (Int, Int) = (1,2)
    ```

  - 矩阵a按元素递乘2.0: `a:*=2.0`

    类似于C++中的`a+=2`

    但是要注意到，等号右侧的数值类型要与a的类型相同（Int, Double）

- **整行或整列的运算**

- 对矩阵a的每一行求和：`sum(a(*,::))`

  对矩阵直接求和：

  ```
  scala> sum(a)
  res82: Int = 18
  ```

  对每一行求和：（注意此时行用`*`代替，而得到的是一个列向量）

  ```
  scala> sum(a(*,::))
  res83: breeze.linalg.DenseVector[Int] = DenseVector(6, 12)
  ```

- 对矩阵a的每一列求和：`sum(a(::,*))`

  此时列用`*`代替，得到的是一个行向量

  ```
  scala> sum(a(::,*))
  res84: breeze.linalg.Transpose[breeze.linalg.DenseVector[Int]] = Transpose(DenseVector(4,
  6, 8))
  ```

- 求矩阵a的列数：`a.cols`

  ```
  scala> a.cols
  res85: Int = 3
  ```

- 求矩阵a的行数：`a.rows`

  ```
  scala> a.rows
  res86: Int = 2
  ```

- 让矩阵a的每一列减去一个固定的列向量b: `a(::,*)-b` （总是用`a(::,*)`代替各列）

  ```
  scala> val b = DenseVector(1,2)
  b: breeze.linalg.DenseVector[Int] = DenseVector(1, 2)

  scala> a(::,*)-b
  res91: breeze.linalg.DenseMatrix[Int] =
  0  1  2
  1  2  3
  ```

- 让矩阵a的每一行减去一个固定的行向量b: `a(*,::) - b`（仍然不用加转置`.t`！，自动匹配）

  ```
  scala> a
  res98: breeze.linalg.DenseMatrix[Int] =
  1  2  3
  3  4  5

  scala> b
  res99: breeze.linalg.DenseVector[Int] = DenseVector(1, 2, 3)

  scala> a(*,::) - b
  res100: breeze.linalg.DenseMatrix[Int] =
  0  0  0
  2  2  2
  ```

- **常用的向量、矩阵函数：**
  - 行列式：`det(A)`
  - 求逆：`inv(A)`
  - 求广义逆：`pinv(A)`
  - 求向量的2-norm：`norm(a)`
- **常用的数学计算：**

    - 调用数学函数包：`import breeze.numerics._`
- `sin`, `cos`, `tan`, `log`, `exp`, `log10`, `sqrt`, `pow`
    - 注意：`exp`的实参必须为Double型！
- **常用分布**
    - 调用程序包：`import breeze.stats.distributions._`
    - 多元正态分布: `new MultivariateGaussian(muVector, sigmaMatrix)`

Breeze补充内容

- 矩阵的谱分解：

```
scala> val a=DenseMatrix((0.17,0.11,0.07,0.10,0.04),(0.11,0.16,0.07,0.09,0.05),
(0.07,0.07,0.09,0.07,0.03),(0.10,0.09,0.07,0.16,0.03),(0.04,0.05,0.03,0.03,0.06))
a: breeze.linalg.DenseMatrix[Double] =
0.17  0.11  0.07  0.1   0.04
0.11  0.16  0.07  0.09  0.05
0.07  0.07  0.09  0.07  0.03
0.1   0.09  0.07  0.16  0.03
0.04  0.05  0.03  0.03  0.06

scala> eig(a)
22/10/11 18:45:10 WARN LAPACK: Failed to load implementation from:
com.github.fommil.netlib.NativeSystemLAPACK
22/10/11 18:45:10 WARN LAPACK: Failed to load implementation from:
com.github.fommil.netlib.NativeRefLAPACK
res0: breeze.linalg.eig.DenseEig =
Eig(DenseVector(0.4253939960429207, 0.07635956684332165, 0.05809247061780503, 0.03675280160898894,
0.04340116488696337),DenseVector(0.0, 0.0, 0.0, 0.0, 0.0),-0.5520216214925445
-0.13525037844191246  -0.8100649503825562   ... (5 total)
-0.5293341343878225   -0.5079438950590737   0.34124724469453555  ...
-0.3486780745241735   0.06424689506354715   0.29271439723550113  ...
-0.5023922052405363   0.7767687401071603    0.19828514347508833  ...
-0.20273918229168844  -0.34088560466902146  0.31991620415073585  ...)
```

此时包含矩阵的特征值和特征向量，难以分辨，故一般**单独调用**矩阵的特征值与特征向量

- 矩阵的特征值（返回值为一个向量）：

```
scala> eig(a).eigenvalues
res1: breeze.linalg.DenseVector[Double] = DenseVector(0.4253939960429207, 0.07635956684332165,
0.05809247061780503, 0.03675280160898894, 0.04340116488696337)
```

- 矩阵的特征向量

```
scala> eig(a).eigenvectors
res2: breeze.linalg.DenseMatrix[Double] =
-0.5520216214925445   -0.13525037844191246  -0.8100649503825562  ... (5 total)
-0.5293341343878225   -0.5079438950590737   0.34124724469453555  ...
-0.3486780745241735   0.06424689506354715   0.29271439723550113  ...
-0.5023922052405363   0.7767687401071603    0.19828514347508833  ...
-0.20273918229168844  -0.34088560466902146  0.31991620415073585  ...
```

此时返回的是一个矩阵，每一列为一个特征向量！即：

$$A = Q\Lambda Q'$$

`eig(a).eigenvectors`返回的是`a`的Q

- 下面对上述原理进行验证：
    - V是一个正交阵

```
scala> val V=eig(a).eigenvectors
```

```
V: breeze.linalg.DenseMatrix[Double] =
-0.5520216214925445   -0.13525037844191246  -0.8100649503825562  ... (5 total)
-0.5293341343878225   -0.5079438950590737   0.34124724469453555  ...
-0.3486780745241735   0.06424689506354715   0.29271439723550113  ...
-0.5023922052405363   0.7767687401071603    0.19828514347508833  ...
-0.20273918229168844  -0.34088560466902146  0.31991620415073585  ...

scala> V*V.t
22/10/11 18:47:46 WARN BLAS: Failed to load implementation from:
com.github.fommil.netlib.NativeSystemBLAS
22/10/11 18:47:46 WARN BLAS: Failed to load implementation from:
com.github.fommil.netlib.NativeRefBLAS
res3: breeze.linalg.DenseMatrix[Double] =
0.9999999999999991      2.7755575615628914E-16   ... (5 total)
2.7755575615628914E-16  1.000000000000001        ...
-9.71445146547012E-17   1.6653345369377348E-16   ...
2.498001805406602E-16   -2.0816681711721685E-16  ...
-6.869504964868156E-16  -1.1657341758564144E-15  ...

scala> diag(V*V.t)
res4: breeze.linalg.DenseVector[Double] = DenseVector(0.9999999999999991, 1.000000000000001,
0.999999999999998, 0.9999999999999992, 1.0000000000000022)
```

- 所做的是矩阵a的特征值分解（谱分解）

```
scala> val error = a - V*diag(W)*V.t
error: breeze.linalg.DenseMatrix[Double] =
1.942890293094024E-16   4.163336342344337E-17   ... (5 total)
4.163336342344337E-17   0.0                     ...
2.7755575615628914E-17  0.0                     ...
1.1102230246251565E-16  5.551115123125783E-17   ...
3.469446951953614E-17   2.7755575615628914E-17  ...

scala> error.toDenseVector
res7: breeze.linalg.DenseVector[Double] = DenseVector(1.942890293094024E-16,
4.163336342344337E-17, 2.7755575615628914E-17, 1.1102230246251565E-16, 3.469446951953614E-
17, 4.163336342344337E-17, 0.0, 0.0, 5.551115123125783E-17, 2.7755575615628914E-17,
2.7755575615628914E-17, -2.7755575615628914E-17, 8.326672684688674E-17,
-2.7755575615628914E-17, -8.326672684688674E-17, 1.1102230246251565E-16,
5.551115123125783E-17, -1.3877787807814457E-17, 1.1102230246251565E-16, 1.734723475976807E-
17, 3.469446951953614E-17, 2.7755575615628914E-17, -8.326672684688674E-17,
1.734723475976807E-17, -1.249000902703301E-16)

scala> norm(error.toDenseVector)
res9: Double = 3.5858039253722845E-16
```

如何验证两个矩阵相等?

两个矩阵分别拉直后相减，计算得到的向量2-范数!

- 奇异值分解

原理:

$$B_{m\times n} = U_{m\times m} D_{m\times n} V_{n\times n}$$

其中，U, V均为正交矩阵

此时，对矩阵a做SVD分解，应该返回包含三个参数的一个向量值。但显然不能直接令u、d、v为返回值，因此要专门用 `svd.SVD(u,d,v)`

```
scala> val svd.SVD(u,d,v)=svd(a)
u: breeze.linalg.DenseMatrix[Double] =
-0.5520216214925445   0.13525037844191135   -0.8100649503825571  ... (5 total)
-0.5293341343878223   0.5079438950590747    0.341247244694535    ...
-0.34867807452417326  -0.06424689506354699  0.29271439723550186  ...
-0.5023922052405363   -0.7767687401071605   0.19828514347508963  ...
-0.2027391822916883   0.34088560466902157   0.3199162041507342   ...
d: breeze.linalg.DenseVector[Double] = DenseVector(0.42539399604292116, 0.07635956684332162,
0.058092470617805064, 0.04340116488696339, 0.036752801608988894)
v: breeze.linalg.DenseMatrix[Double] =
-0.5520216214925446  -0.5293341343878225   -0.3486780745241733   ... (5 total)
0.13525037844191123  0.5079438950590746    -0.06424689506354674  ...
-0.8100649503825568  0.34124724469453505   0.2927...
```

按理说，此时为方阵的奇异值分解，等价于谱分解；故

$$U = V^\top$$

验证该结论：（拉直后取2-范数）

```
scala> u-v.t
res11: breeze.linalg.DenseMatrix[Double] =
1.1102230246251565E-16  1.1102230246251565E-16   ... (5 total)
2.220446049250313E-16   1.1102230246251565E-16   ...
5.551115123125783E-17   -2.498001805406602E-16   ...
2.220446049250313E-16   -3.3306690738754696E-16  ...
0.0                     -1.1102230246251565E-16  ...

scala> norm((u-v.t).toDenseVector)
res12: Double = 2.4148945880937642E-15
```

构造一个非方阵，研究奇异值分解：

```
scala> val b =a(::,0 to 2)
b: breeze.linalg.DenseMatrix[Double] =
0.17  0.11  0.07
0.11  0.16  0.07
0.07  0.07  0.09
0.1   0.09  0.07
0.04  0.05  0.03

scala> val svd.SVD(u1,d1,v1)=svd(b)
u1: breeze.linalg.DenseMatrix[Double] =
-0.5847262587563399   -0.7382281133635373   0.19487889487021706   ... (5 total)
-0.5628568936204275   0.6277735843788151    0.4680712643942523    ...
-0.3542214853702293   0.19094969166815096   -0.8394758650702095   ...
-0.42142651654118635  -0.04424618459401045  -0.19515527057586574  ...
-0.1954843985663476   0.15000023265514775   0.011233882392693618  ...
d1: breeze.linalg.DenseVector[Double] = DenseVector(0.35914100293769985, 0.05608150264978233,
0.04767184776621826)
v1: breeze.linalg.DenseMatrix[Double] =
-0.6573330584045433  -0.631715955515004   -0.4109114282619732
-0.7400268139341191  0.6441163799099541   0.19358306689948956
0.14238526955779499  0.4313340244861992   -0.8908857358457891
```

此处的d1为一个向量，但实际分解中

$$D = \begin{pmatrix} \lambda_1 & 0 & 0 \\ 0 & \lambda_2 & 0 \\ 0 & 0 & \lambda_3 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}$$

即：会在下方补充零行，以满足5行3列的要求！

构造矩阵D，验证上述结论：

```
scala> diag(d1)
res16: breeze.linalg.DenseMatrix[Double] =
0.35914100293769985  0.0                  0.0
0.0                  0.05608150264978233  0.0
0.0                  0.0                  0.04767184776621826

scala> val d0 = DenseMatrix.zeros[Double](2,3)
d0: breeze.linalg.DenseMatrix[Double] =
0.0  0.0  0.0
0.0  0.0  0.0

scala> val D = DenseMatrix.vertcat(diag(d1),d0)
D: breeze.linalg.DenseMatrix[Double] =
0.35914100293769985  0.0                  0.0
0.0                  0.05608150264978233  0.0
0.0                  0.0                  0.04767184776621826
0.0                  0.0                  0.0
0.0                  0.0                  0.0

scala> norm((b-u1*D*v1).toDenseVector)
res22: Double = 6.691626947686432E-17
```

- 矩阵的秩：

```
scala> rank(a)
res17: Int = 5

scala> rank(b)
res18: Int = 3
```

- 按行相除：

```
scala> a(*,::)/DenseVector(1.0,2.0,3.0,4.0,5.0)
res30: breeze.linalg.DenseMatrix[Double] =
0.17  0.055  0.023333333333333334  0.025   0.008
0.11  0.08   0.023333333333333334  0.0225  0.01
0.07  0.035  0.03                  0.0175  0.006
0.1   0.045  0.023333333333333334  0.04    0.006
0.04  0.025  0.01                  0.0075  0.012
```

- 按行相乘：（但必须是列向量）

```
scala> a(*,::)*DenseVector(1.0,2.0,3.0,4.0,5.0)
res31: breeze.linalg.DenseMatrix[Double] =
0.17  0.22  0.21000000000000002  0.4   0.2
0.11  0.32  0.21000000000000002  0.36  0.25
0.07  0.14  0.27                 0.28  0.15
0.1   0.18  0.21000000000000002  0.64  0.15
0.04  0.1   0.09                 0.12  0.3
```

- 关于DenseVector的索引

DenseVector如果要用索引，就只能是Double类型！如果DenseVector是Int类型，则会报错！

```
scala> val a = DenseVector(1.0,2.0,3.0)
a: breeze.linalg.DenseVector[Double] = DenseVector(1.0, 2.0, 3.0)

scala> a(0)
res32: Double = 1.0
```

HW1

1. **请利用for循环计算100以内所有奇数的总和**

```
scala> var sum = 0
sum: Int = 0

scala> for(t<-1 to 100 if t%2==1)
     | {sum = sum + t}

scala> sum
res2: Int = 2500
```

2. **请利用for循环计算当n=100时1+1/2+1/3+1/4+...+1/100的值**

```
scala> var sum1 =0.0
sum1: Double = 0.0

scala> for(i<-1 to 100){
     | sum1 = sum1 + 1.0/i}

scala> sum1
res4: Double = 5.187377517639621
```

思考：运算中不能自动转换为Double吗？

答案：能，但问题在于sum1的类型无法进行转换！

3. **使用递归函数的方法实现阶乘函数n!= 1*2*...*n。并计算15!**

```
scala> def fac(n:Int):Long = {
     | if(n == 0) 1
     | else n * fac(n - 1)}
fac: (n: Int)Long

scala> fac(15)
res0: Long = 1307674368000
```

4. **使用递归函数的方法计算斐波那契数列的第40项的值。在这里我们假设斐波那契数列的第0项为0，第一项为1**

```
scala> def fib(n:Int):Long = {
     | if(n == 0) 0
     | else{
     |     if(n == 1) 1
     |     else fib(n - 2) + fib(n - 1)}}
fib: (n: Int)Long

scala> fib(40)
res1: Long = 102334155
```

5. **使用RDD和MapReduce的方式为数据集{1,3,5,7,9,11,13,15,17,19}求样本方差**

```
scala> val a = (1 to 10).toArray.map(s => 2*s - 1)
a: Array[Int] = Array(1, 3, 5, 7, 9, 11, 13, 15, 17, 19)

scala> val a_rdd = sc.parallelize(a)
a_rdd: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[0] at parallelize at <console>:24
```

```
scala> val mu = a_rdd.reduce((x,y) => x + y)/a.size.toDouble
mu: Double = 10.0

scala> val mom = a_rdd.map(s => (s - mu)*(s - mu))
mom: org.apache.spark.rdd.RDD[Double] = MapPartitionsRDD[1] at map at <console>:24

scala> val sigma = mom.reduce((x,y) => x + y)/(a.size - 1)
sigma: Double = 36.666666666666664
```

6. **假设**$X = (X_1, X_2, \cdots, X_n)$**，是一个**$n$**元向量，我们定义**$X$**的**$k$**阶差分**$X^{(k)}$**为：**

$$X^{(k)} \triangleq \left(X_1^{(k)}, X_2^{(k)}, \cdots, X_{n-k}^{(k)}\right)$$

其中对任意$1 \leq i \leq n-k$**，**

$$X_i^{(k)} \triangleq X_{i+1}^{(k-1)} - X_i^{(k-1)}$$

并且对$k = 0$**, 我们假设**$X^{(0)} = X$**. 请编写一个scala的递归函数，使得这个函数的输入参数为某个向量**X**和某个正整数**k**，输出结果为向量**X**的**k**阶差分**$X^{(k)}$**。最后，运行这个函数，计算向量(1,5,3,7,4,2,7,8,2,4,6,9,3,3,76,8)的5阶差分**

```
scala> def fun(x:breeze.linalg.DenseVector[Int],k:Int):breeze.linalg.DenseVector[Int] = {
     | if(k == 0) x
     | else{
     |     var a = DenseVector.zeros[Int](x.size - 1)
     |     a = x(1 until x.size) - x(0 until (x.size - 1))
     |     fun(a,k - 1)}}
fun: (x: breeze.linalg.DenseVector[Int], k: Int)breeze.linalg.DenseVector[Int]

scala> val a = DenseVector(1,5,3,7,4,2,7,8,2,4,6,9,3,3,76,8)
a: breeze.linalg.DenseVector[Int] = DenseVector(1, 5, 3, 7, 4, 2, 7, 8, 2, 4, 6, 9, 3, 3, 76, 8)

scala> fun(a,5)
res7: breeze.linalg.DenseVector[Int] = DenseVector(46, -23, -15, 25, 10, -41, 32, -20, 36, 27,
-333)
```

注：该方法采取了向量化操作，避免循环影响程序效率！

第四次课

- 常用数学计算：

  做数学计算需要导入`breeze.numerics._`包

```
scala> import breeze.numerics._
import breeze.numerics._
```

  但是package中的函数大部分只针对**双精度**的数据类型！（如`exp`）

```
scala> val tmpVec = DenseVector(0.0,1.1,2.2,3.3)
tmpVec: breeze.linalg.DenseVector[Double] = DenseVector(0.0, 1.1, 2.2, 3.3)

scala> exp(tmpVec)
res34: breeze.linalg.DenseVector[Double] = DenseVector(1.0, 3.0041660239464334, 9.025013499434122,
27.112638920657883)
```

  说明这些函数适用于向量化操作；

  如果对单精度类型使用这些函数，则会报错：

```
scala> val e = DenseVector(1,2,3)
e: breeze.linalg.DenseVector[Int] = DenseVector(1, 2, 3)

scala> exp(e)
<console>:34: error: could not find implicit value for parameter impl:
breeze.numerics.exp.Impl[breeze.linalg.DenseVector[Int],VR]
       exp(e)
```

- 常用分布:

导入有关概率分布的`breeze.stats.distributions._`包:

```
scala> import breeze.stats.distributions._
import breeze.stats.distributions._
```

首先需要**构造一个分布**，此处为参数为2.0的Poisson分布

```
scala> val MyPoisson = new Poisson(2.0)
Mypoisson: breeze.stats.distributions.Poisson = Poisson(2.0)
```

即`MyPoisson`为一个分布

构造由分布产生的随机数，利用`<distribution>.sample(number)`函数

```
scala> val RandNumbers = MyPoisson.sample(100)
RandNumbers: IndexedSeq[Int] = Vector(2, 2, 2, 2, 3, 0, 1, 0, 4, 1, 4, 3, 3, 0, 4, 0, 1, 2, 2, 0,
0, 3, 3, 3, 1, 3, 1, 3, 3, 0, 1, 2, 1, 0, 1, 4, 2, 2, 2, 5, 0, 0, 5, 4, 1, 2, 2, 1, 6, 2, 0, 1, 3,
2, 3, 2, 1, 0, 4, 3, 2, 1, 1, 2, 0, 0, 1, 1, 1, 1, 2, 3, 6, 1, 1, 0, 2, 3, 2, 0, 5, 1, 2, 2, 2, 0,
0, 2, 1, 2, 4, 1, 0, 2, 2, 4, 2, 4, 1, 0)
```

注意到此处返回值类型为IndexedSeq[Int]，与后面的充分统计量求取有关；

由于生成的随机数本质上是一个向量，故可以用向量的函数进行操作:

```
scala> RandNumbers.size
res35: Int = 100

scala> sum(RandNumbers)
res36: Int = 185

scala> sum(RandNumbers)/RandNumbers.size.toDouble
res38: Double = 1.85
```

注意：虽然Scala语言较“鸡肋”，许多函数都没有，但对Vector、Array与DenseVector都有sum函数，不必reduce计算；

根据大数定律，均值应趋于2，取更大的样本量进行验证:

```
scala> val RandNumbers = MyPoisson.sample(100)
RandNumbers: IndexedSeq[Int] = Vector(2, 4, 1, 1, 2, 1, 3, 1, 1, 3, 6, 1, 3, 1, 3, 1, 0, 2, 2, 3,
2, 3, 4, 0, 3, 1, 1, 3, 2, 4, 1, 1, 4, 1, 0, 0, 1, 2, 2, 1, 1, 2, 2, 3, 3, 6, 1, 2, 1, 3, 2, 3, 3,
0, 3, 2, 3, 1, 1, 3, 2, 2, 4, 2, 3, 0, 1, 1, 2, 2, 2, 0, 1, 3, 1, 7, 1, 1, 4, 2, 2, 0, 4, 3, 4, 1,
2, 2, 1, 1, 2, 2, 2, 2, 3, 1, 1, 0, 2, 2)

scala> val RandNumbers = MyPoisson.sample(100000)
RandNumbers: IndexedSeq[Int] = Vector(0, 1, 2, 3, 5, 2, 2, 4, 0, 3, 2, 2, 2, 4, 3, 4, 2, 5, 1, 4,
4, 0, 1, 2, 2, 2, 1, 1, 0, 1, 2, 3, 3, 0, 5, 0, 2, 3, 2, 1, 0, 1, 0, 5, 1, 3, 2, 2, 2, 1, 4, 1, 2,
1, 1, 1, 3, 0, 3, 1, 2, 4, 1, 1, 2, 0, 1, 2, 2, 1, 1, 1, 2, 3, 6, 6, 4, 3, 4, 2, 3, 1, 3, 2, 4, 2,
1, 0, 4, 3, 1, 1, 5, 2, 0, 3, 2, 5, 3, 4, 1, 4, 1, 2, 3, 1, 5, 1, 1, 3, 0, 3, 2, 2, 3, 3, 2, 2, 2,
1, 3, 4, 2, 1, 2, 1, 1, 1, 1, 0, 2, 2, 1, 1, 2, 0, 2, 0, 7, 1, 1, 2, 1, 4, 1, 0, 4, 4, 0, 4, 3, 1,
0, 3, 6, 1, 3, 4, 0, 2, 5, 1, 1, 0, 2, 0, 1, 4, 1, 1, 2, 1, 0, 2, 4, 1, 3, 2, 1, 4, 5, 3, 2, 2, 0,
1, 1, 2, 2, 0, 1, 1, 2, 1, 2, 1, 0, 4, 3, 3, 3, 1, 4, 4, 1, 2, 4, 3, 2, 1, 1, 1, 3, 1, 0, 5, 2, 2,
1, 1, 1, 0, 2, 0, 4, 2, 3, 1, 3, 2, 1, 1, 0, 2, 4, 0, 1, 0, 1, 2, 1, 3, 1, 2, 1, 1, 3, 2, 3, 2, 4,
2, 0, ...

scala> sum(RandNumbers)/RandNumbers.size.toDouble
res65: Double = 1.99949
```

- 计算特定点的概率

  直接利用`<distribution>.probabilityOf(x)`

```
scala> MyPoisson.probabilityOf(0)
res43: Double = 0.1353352832366127

scala> MyPoisson.probabilityOf(1)
res44: Double = 0.2706705664732254
```

若是要计算一个向量中各个值的PMF，**不能直接用**`<distribution>.probabilityOf(vector)`，否则会报错：

```
scala> MyPoisson.probabilityOf(RandNumbers)
<console>:35: error: type mismatch;
 found   : IndexedSeq[Int]
 required: Int
       MyPoisson.probabilityOf(RandNumbers)
```

即：`<distribution>.probabilityOf(Int)`必须传入**整型值**！

因此可以考虑用`map`函数：

```
scala> RandNumbers.map(i => MyPoisson.probabilityOf(i))
res42: IndexedSeq[Double] = Vector(0.2706705664732254, 0.2706705664732254, 0.2706705664732254,
0.2706705664732254, 0.18044704431548356, 0.1353352832366127, 0.2706705664732254,
0.1353352832366127, 0.09022352215774178, 0.2706705664732254, 0.09022352215774178,
0.18044704431548356, 0.18044704431548356, 0.1353352832366127, 0.09022352215774178,
0.1353352832366127, 0.2706705664732254, 0.2706705664732254, 0.2706705664732254, 0.1353352832366127,
0.1353352832366127, 0.18044704431548356, 0.18044704431548356, 0.18044704431548356,
0.2706705664732254, 0.18044704431548356, 0.2706705664732254, 0.18044704431548356,
0.18044704431548356, 0.1353352832366127, 0.2706705664732254, 0.2706705664732254,
0.2706705664732254, 0.1353352832366127, 0.2706705664732254, 0.09022352215774178,
0.2706705664732254, 0.2706705...
```

在这里，`map`函数可以简化：（注意此处没有`.`，而是**空格**！）

```
scala> RandNumbers map{MyPoisson.probabilityOf(_)}
res47: IndexedSeq[Double] = Vector(0.2706705664732254, 0.2706705664732254, 0.2706705664732254,
0.2706705664732254, 0.18044704431548356, 0.1353352832366127, 0.2706705664732254,
0.1353352832366127, 0.09022352215774178, 0.2706705664732254, 0.09022352215774178,
0.18044704431548356, 0.18044704431548356, 0.1353352832366127, 0.09022352215774178,
0.1353352832366127, 0.2706705664732254, 0.2706705664732254, 0.2706705664732254, 0.1353352832366127,
0.1353352832366127, 0.18044704431548356, 0.18044704431548356, 0.18044704431548356,
0.2706705664732254, 0.18044704431548356, 0.2706705664732254, 0.18044704431548356,
0.18044704431548356, 0.1353352832366127, 0.2706705664732254, 0.2706705664732254,
0.2706705664732254, 0.1353352832366127, 0.2706705664732254, 0.09022352215774178,
0.2706705664732254, 0.2706705...
```

- 计算各点的累计分布函数（CDF）值：

```
scala> MyPoisson.cdf(2)
res45: Double = 0.6766764161830635
//验算CDF与PMF之间的关系
scala> MyPoisson.probabilityOf(0)+Mypoisson.probabilityOf(1)+Mypoisson.probabilityOf(2)
res46: Double = 0.6766764161830635
```

- 此处Poisson分布的**自变量只能取值为整数**！

  ```
  scala> MyPoisson.cdf(2)
  res4: Double = 0.6766764161830635

  scala> MyPoisson.cdf(2.0)
  <console>:36: error: type mismatch;
   found   : Double(2.0)
   required: Int
         MyPoisson.cdf(2.0)
  ```

  下面实现：如何使Poisson分布自变量取值为浮点数

- **for循环中yield的用法**

首先回顾for循环语法 `for(<identifier> <- <iterator>) [yield] [<expression>]`，此处`yield`的用法为：

其中identifier为迭代变量，iterator为迭代器（取值范围）

对于for循环的每次迭代，yield都会生成一个将“被记住”的值。就像有一个看不见的缓冲区，for循环的每一次迭代都会将另一个新的值添加到该缓冲区。当for循环结束运行时，它将依次返回该“缓冲区”值的集合。返回的集合的类型与迭代产生的类型相同，因此Map会生成Map，List将生成List，等等。

注意：最初的集合没有改变，`for-yield`根据指定的算法创建一个新的集合！

```
scala> val a = Array(1, 2, 3, 4, 5)
a: Array[Int] = Array(1, 2, 3, 4, 5)

scala> for (e <- a) yield e
res5: Array[Int] = Array(1, 2, 3, 4, 5)

scala> for (e <- a) yield e * 2
res6: Array[Int] = Array(2, 4, 6, 8, 10)

scala> for (e <- a) yield e % 2
res7: Array[Int] = Array(1, 0, 1, 0, 1)

scala> val b = List(1, 2, 3, 4, 5)
b: List[Int] = List(1, 2, 3, 4, 5)

scala> for (e <- b) yield e
res8: List[Int] = List(1, 2, 3, 4, 5)
```

目的：将Poisson分布产生的整数型随机数转变成双精度型

原因：只有产生的随机数为**双精度型**时，才能计算样本的**均值与方差**！（因为均值和方差均为双精度型！）

```
scala> val DoublePoisson = for(x <- MyPoisson) yield x.toDouble
DoublePoisson: breeze.stats.distributions.Rand[Double] =
MappedRand(Poisson(2.0),$Lambda$2057/173462595@79b3937a)
```

注意：此时生成的DoublePoisson本质上**不是一个概率分布**，而是一个专门用来**生成随机数**的分布！

解释：

```
scala> MyPoisson
res11: breeze.stats.distributions.Poisson = Poisson(2.0)

scala> DoublePoisson
res12: breeze.stats.distributions.Rand[Double] =
MappedRand(Poisson(2.0),$Lambda$2057/173462595@79b3937a)
```

由此，可以利用DoublePoisson分布进行"双精度"抽样：

```
scala> DoublePoisson.sample(10)
res48: IndexedSeq[Double] = Vector(3.0, 0.0, 0.0, 2.0, 2.0, 2.0, 3.0, 2.0, 2.0, 2.0)
```

- 样本均值与方差：

```
scala> breeze.stats.meanAndVariance(DoublePoisson.sample(10))
res49: breeze.stats.meanAndVariance.MeanAndVariance = MeanAndVariance(1.5,1.1666666666666667,10)

scala> breeze.stats.meanAndVariance(DoublePoisson.sample(10000))
res50: breeze.stats.meanAndVariance.MeanAndVariance =
MeanAndVariance(1.9933999999999974,2.0135577957795863,10000)
```

注：（1）后者说明了大数定律

（2）当然也可以`import breeze.stats._`后直接用`meanAndVariance`函数

```
scala> import breeze.stats._
import breeze.stats._

scala> meanAndVariance(DoublePoisson.sample(10))
res14: breeze.stats.meanAndVariance.MeanAndVariance = MeanAndVariance(1.7,1.3444444444444443,10)
```

- 总体均值与方差：

```
scala> MyPoisson.mean
res15: Double = 2.0

scala> MyPoisson.variance
res17: Double = 2.0
```

注：对样本求均值方差时，是Variance；而对总体求均值方差时，是variance

- Beta分布

  与创建Poisson分布类似，创建Beta分布并抽取样本：

```
scala> val MyBeta = new Beta(1.0,2.0)
MyBeta: breeze.stats.distributions.Beta = Beta(1.0,2.0)

scala> val RandNumbers = MyBeta.sample(10)
RandNumbers: IndexedSeq[Double] = Vector(0.40883125384638774, 0.2350534364309944,
0.6094560213471603, 0.042365414904149926, 0.160774612018955, 0.7891093493976181,
0.25202685409475334, 0.34830538306979364, 0.05270738685076458, 0.15923792203377157)
```

- 连续型分布看PDF，离散型分布看PMF：即连续型为`<distribution>.pdf`，离散型为`<distribution.probabilityOf>`

```
scala> MyBeta.pdf(0.05)
res53: Double = 1.9
```

与Poisson分布各样本值的CDF、PMF类似，不能直接传入向量；而是利用Map函数！

```
scala> RandNumbers map{MyBeta.pdf(_)}
res54: IndexedSeq[Double] = Vector(1.1823374923072245, 1.5298931271380112, 0.7810879573056795,
1.9152691701917002, 1.67845077596209, 0.4217813012047637, 1.4959462918104933, 1.3033892338604127,
1.8945852262984708, 1.6815241559324567)

scala> RandNumbers map{MyBeta.cdf(_)}
res55: IndexedSeq[Double] = Vector(0.6505195135711661, 0.4148567548839691, 0.8474754007380103,
0.08293600142829909, 0.2957007481680644, 0.9555251334885041, 0.4405361730046085,
0.5752941262641917, 0.10263670507289299, 0.2931191282539096)
```

求解极大似然估计

以Dirichlet分布为例：

Dirichlet分布：

对独立同分布的连续随机变量$\boldsymbol{X} \in \mathbb{R}_d$和支撑集$\boldsymbol{X} \in (\mathbf{0}, \mathbf{1}), \|\boldsymbol{X}\| = 1$，若$\boldsymbol{X}$服从狄利克雷分布，则其概率密度函数$\mathrm{Dir}(\boldsymbol{X} \mid \boldsymbol{\alpha})$有如下定义

$$\mathrm{Dir}(\boldsymbol{X} \mid \boldsymbol{\alpha}) = \frac{1}{\mathrm{B}(\boldsymbol{\alpha})} \prod_{i=1}^{d} X_i^{\alpha_i - 1}$$

$$\mathrm{B}(\boldsymbol{\alpha}) = \frac{\prod_{i=1}^{d} \Gamma(\alpha_i)}{\Gamma(\alpha_0)}, \quad \alpha_0 = \sum_{i=1}^{d} \alpha_i, \quad d \geq 3$$

创建一个Dirichlet分布：

```
scala> val MyDir = new Dirichlet(DenseVector(3.0,2.0,6.0))
MyDir: breeze.stats.distributions.Dirichlet[breeze.linalg.DenseVector[Double],Int] =
Dirichlet(DenseVector(3.0, 2.0, 6.0))
```

注意：Dirichlet分布的参数有多个，即$d \geq 3$，同时d是一个不定的数，因此不能像Beta分布一样，直接输入两个数；而是应该输入一个DenseVector！

```
scala> val MyDir = new Dirichlet(3.0,2.0,6.0)
<console>:32: error: could not find implicit value for parameter space:
breeze.math.EnumeratedCoordinateField[(Double, Double, Double),I,Double]
       val MyDir = new Dirichlet(3.0,2.0,6.0)
```

由于Dirichlet分布是d元分布，因此抽样结果应该是一个d元向量：

```
scala> val data = MyDir.sample(10)
data: IndexedSeq[breeze.linalg.DenseVector[Double]] = Vector(DenseVector(0.11537559192900902,
0.36771439954143814, 0.5169100085295529), DenseVector(0.1682227972088889, 0.28117291314971343,
0.5506042896413977), DenseVector(0.43293293948629014, 0.11483659222997701, 0.45223046828373287),
DenseVector(0.5687528579853531, 0.14065410014479687, 0.29059304186984997),
DenseVector(0.49397846522812494, 0.16216260306876684, 0.34385893170310816),
DenseVector(0.5305610579923182, 0.12443792474524379, 0.3450010172624378),
DenseVector(0.4850846785330612, 0.18110350532277336, 0.33381181614416544),
DenseVector(0.22462178340837 3, 0.11712601258379933, 0.658522040078278),
DenseVector(0.42964345843904367, 0.22119418833825083, 0.3491623532227055),
DenseVector(0.42658254133194934, 0.18628698588337858, 0.3871304...
```

为做极大似然估计，必须要将数据类型转换为各个分布所需要的格式，例如在Dirichlet分布中，数据需要作为**IndexedSeq**类型！

```
scala> val data0 = data.toIndexedSeq
data0: scala.collection.immutable.IndexedSeq[breeze.linalg.DenseVector[Double]] =
Vector(DenseVector(0.29088723557577995, 0.23599961357316732, 0.47311315085105277),
DenseVector(0.5419060416084837, 0.054476627154263074, 0.4036173312372534),
DenseVector(0.42555304241776554, 0.1844063772463482, 0.3900405803358863),
DenseVector(0.4130982997507517, 0.16528440020236806, 0.42161730004688036),
DenseVector(0.26681051050340315, 0.31559449615515534, 0.41759499334144146),
DenseVector(0.1782126137786933, 0.32824824154582866, 0.4935391446754781),
DenseVector(0.2516363818187393, 0.07006406582686814, 0.6782995523543925),
DenseVector(0.2767449448881314, 0.020108432546342148, 0.7031466225655264),
DenseVector(0.13711390701534773, 0.22277583644255125, 0.6401102565421011),
DenseVector(0.5990856954584832, 0....
```

目标：根据样本对关心的参数做极大似然估计；

回忆数理统计的方法：一个样本可以做MLE，n个样本也可以做MLE。即：每新增一个样本，都可以在原有的基础上继续进行极大似然估计！而在没有样本时，我们当然无法进行MLE，故相当于从“0”开始逐次累加更迭。

因此，我们首先需要初始化该Dirichlet分布的对象：

```
scala> val expFam = new Dirichlet.ExpFam(DenseVector.zeros[Double](3))
expFam: breeze.stats.distributions.Dirichlet.ExpFam[breeze.linalg.DenseVector[Double],Int] =
breeze.stats.distributions.Dirichlet$ExpFam@370e81f8
```

此时该分布族的充分统计量均为0（因为没有任何样本）：

```
scala> expFam.emptySufficientStatistic
res0: expFam.SufficientStatistic = SufficientStatistic(0.0,DenseVector(0.0, 0.0, 0.0))
```

由因子分解定理，再根据样本数据计算充分统计量：

```
scala> val SuffStat = data0.foldLeft(expFam.emptySufficientStatistic){(x,y) => x
+expFam.sufficientStatisticFor(y)}
SuffStat: expFam.SufficientStatistic = SufficientStatistic(10.0,DenseVector(-16.231732995964492,
-19.802596017519576, -5.579123407607025))
```

理解：

（1）首先研究 `<distribution>.sufficientStatisticFor(<data>)` 的意义：

data0(1)的数据结构：

```
scala> data0(1)
res3: breeze.linalg.DenseVector[Double] = DenseVector(0.2083471454841435, 0.2658170172653276,
0.5258358372505291)
```

`<distribution>.sufficientStatisticFor(<data>)` 的作用：将该样本 `data` 的数据变换为充分统计量需要的形式，加到 `expFam` 上；例如此处的充分统计量为

$$(\sum_{j=1}^{n}\log X_{1j},\sum_{j=1}^{n}\log X_{2j},\sum_{j=1}^{n}\log X_{3j})$$

故按理说会将data0(1)的数据取对数后加到 `expFam` 的初始值上：

```
scala> log(data0(1))
res4: breeze.linalg.DenseVector[Double] = DenseVector(-1.5685496217875932, -1.3249471119332246,
-0.6427662114385536)

scala> expFam.sufficientStatisticFor(data0(1))
res6: expFam.SufficientStatistic = SufficientStatistic(1.0,DenseVector(-1.5685496217875932,
-1.3249471119332246, -0.6427662114385536))
```

（2）再研究 `foldLeft` 的意义：

`<data>.foldLeft(初始值)((x,y) => fun(x,y))` 将data的每一个元素按照从左到右的顺序加到初始值之上：

```
scala> List(1,2,3).foldLeft(0)((x,y)=>x+y)
res23: Int = 6

scala> List(1,2,3).foldLeft(0)((x,y)=>x+y*y)
res24: Int = 14
```

尤其要注意上面第二个例子！

> 大川：在foldLeft里面，左边的那个数x已经是充分统计量了，所以只需要计算y的充分统计量就行
>
> 震宇：意思是foldLeft和map函数意义不一样，map是x和y都取自data，所以都得变；但是foldLeft里面规定其中一个已经变好了，只用变一个吗
>
> 大川：对的，可以这么理解

最后，再根据计算的充分统计量求解MLE：

直接调用 `<distribution>.mle(充分统计量)` 即可

```
scala> val EstimatedParameter = expFam.mle(SuffStat)
EstimatedParameter: breeze.linalg.DenseVector[Double] = DenseVector(2.5081651320561233,
1.8911365322914648, 6.378015503847785)
```

- 用MapReduce方法求解Dirichlet分布的MLE

```
scala> val data = MyDir.sample(100).toIndexedSeq
data: scala.collection.immutable.IndexedSeq[breeze.linalg.DenseVector[Double]] =
Vector(DenseVector(0.3583757255177561, 0.3564377911594644, 0.2851864833227795),
DenseVector(0.11250859683649364, 0.18467237123705074, 0.7028190319264556),
DenseVector(0.20031616562357168, 0.22229553487568177, 0.5773882995007467),
DenseVector(0.3341164112128445, 0.32348642088959517, 0.3423971678975603),
DenseVector(0.29107408564668724, 0.07707233448989609, 0.6318535798634166),
DenseVector(0.20873061705899987, 0.2996867854442573, 0.4915825974967428),
DenseVector(0.41393893478867716, 0.1799304309962169, 0.4061306342151059),
DenseVector(0.5104357143795478, 0.005132694261481613, 0.4844315913589705),
DenseVector(0.27373498001436997, 0.06248221295296667, 0.6637828070326632),
DenseVector(0.14225494182369777, 0.1168...

scala> val rdd = sc.parallelize(data)
rdd: org.apache.spark.rdd.RDD[breeze.linalg.DenseVector[Double]] = ParallelCollectionRDD[0] at
parallelize at <console>:33

scala> val rdd1 = rdd.map(s => log(s))
rdd1: org.apache.spark.rdd.RDD[breeze.linalg.DenseVector[Double]] = MapPartitionsRDD[1] at map at
<console>:32

scala> val result = rdd1.reduce((x,y) => x+y)
result: breeze.linalg.DenseVector[Double] = DenseVector(-141.8007846117827, -190.6482138216638,
-67.95617017759261)

scala> val expFam = new Dirichlet.ExpFam(DenseVector.zeros[Double](3))
expFam: breeze.stats.distributions.Dirichlet.ExpFam[breeze.linalg.DenseVector[Double],Int] =
breeze.stats.distributions.Dirichlet$ExpFam@68be702f

scala> val SuffStat = expFam.SufficientStatistic(100.0,result)
SuffStat: expFam.SufficientStatistic = SufficientStatistic(100.0,DenseVector(-141.8007846117827,
-190.6482138216638, -67.95617017759261))

scala> val EstimatedParameter = expFam.mle(SuffStat)
EstimatedParameter: breeze.linalg.DenseVector[Double] = DenseVector(2.726410750850821,
1.848387648086095, 5.188400727637554)
```

要注意：在用MapReduce函数时，首先要**转换为分布式存储**，即用 `sc.parallelize` 函数进行转换！

HW2

1. 假设向量$a = (1.0, 3.0, 5.0, 7.0, 9.0, 7.0, 5.0, 3.0, 1.0)$. 试使用Scala中的矩阵运算新建一个矩阵$A$使得$A(i, j) = a(i) + a(j)$. 注意请勿使用循环语句完成本题。

```
scala> import breeze.linalg._
import breeze.linalg._

scala> val a = DenseVector(1.0,3.0,5.0,7.0,9.0,7.0,5.0,3.0,1.0)
a: breeze.linalg.DenseVector[Double] = DenseVector(1.0, 3.0, 5.0, 7.0, 9.0, 7.0, 5.0, 3.0, 1.0)

scala> val A = DenseMatrix.tabulate(9,9){case(i,j) => a(i)+a(j)}
A: breeze.linalg.DenseMatrix[Double] =
2.0   4.0   6.0   8.0   10.0  8.0   6.0   4.0   2.0
4.0   6.0   8.0   10.0  12.0  10.0  8.0   6.0   4.0
6.0   8.0   10.0  12.0  14.0  12.0  10.0  8.0   6.0
8.0   10.0  12.0  14.0  16.0  14.0  12.0  10.0  8.0
10.0  12.0  14.0  16.0  18.0  16.0  14.0  12.0  10.0
8.0   10.0  12.0  14.0  16.0  14.0  12.0  10.0  8.0
6.0   8.0   10.0  12.0  14.0  12.0  10.0  8.0   6.0
4.0   6.0   8.0   10.0  12.0  10.0  8.0   6.0   4.0
2.0   4.0   6.0   8.0   10.0  8.0   6.0   4.0   2.0
```

2. 假设向量$a = (1.0, 3.0, 5.0, 7.0, 9.0, 7.0, 5.0, 3.0, 1.0)$. 试使用Scala中的矩阵运算新建一个矩阵$B$使得$B(i, j) = a(i) \times a(j)$. 注意请勿使用循环语句完成本题。

```
scala> val B = DenseMatrix.tabulate(9,9){case(i,j) => a(i)*a(j)}
B: breeze.linalg.DenseMatrix[Double] =
1.0  3.0   5.0   7.0   9.0   7.0   5.0   3.0   1.0
3.0  9.0   15.0  21.0  27.0  21.0  15.0  9.0   3.0
5.0  15.0  25.0  35.0  45.0  35.0  25.0  15.0  5.0
7.0  21.0  35.0  49.0  63.0  49.0  35.0  21.0  7.0
9.0  27.0  45.0  63.0  81.0  63.0  45.0  27.0  9.0
7.0  21.0  35.0  49.0  63.0  49.0  35.0  21.0  7.0
5.0  15.0  25.0  35.0  45.0  35.0  25.0  15.0  5.0
3.0  9.0   15.0  21.0  27.0  21.0  15.0  9.0   3.0
1.0  3.0   5.0   7.0   9.0   7.0   5.0   3.0   1.0
```

3. 假设矩阵C有如下的形式：

$$C = \begin{bmatrix} -1.0 & -2.0 & -3.0 \\ 4.0 & 5.0 & 6.0 \\ 7.0 & 8.0 & 9.0 \\ 6.0 & 5.0 & 4.0 \\ -3.0 & -2.0 & -1.0 \end{bmatrix}$$

试使用Scala中的矩阵运算语言计算矩阵C每列的均值，然后将每一行的数据减去各自对应列的均值，输出得到的新矩阵。

```
scala> val C = DenseMatrix((-1.0,-2.0,-3.0),(4.0,5.0,6.0),(7.0,8.0,9.0),(6.0,5.0,4.0),
(-3.0,-2.0,-1.0))
C: breeze.linalg.DenseMatrix[Double] =
-1.0  -2.0  -3.0
4.0   5.0   6.0
7.0   8.0   9.0
6.0   5.0   4.0
-3.0  -2.0  -1.0

scala> val C_colmean = sum(C(::,*))/C.rows.toDouble
```

```
C_colmean: breeze.linalg.Transpose[breeze.linalg.DenseVector[Double]] = Transpose(DenseVector(2.6, 
2.8, 3.0))

scala> val C_center = C(*,::) - C_colmean.t
C_center: breeze.linalg.DenseMatrix[Double] =
-3.6  -4.8  -6.0
1.4   2.2   3.0
4.4   5.2   6.0
3.4   2.2   1.0
-5.6  -4.8  -4.0
```

4. **试编写Scala函数计算某方阵X的平方根矩阵S, 使得$S * S = X$。注意，这里的*指矩阵相乘运算，不是按元素相乘。**

```
scala> def fun(X:DenseMatrix[Double]):DenseMatrix[Double] = {
     | val Q = eig(X).eigenvectors
     | val lambda = eig(X).eigenvalues
     | val lambda_new = sqrt(lambda)
     | val A = diag(lambda_new)
     | Q * A * Q.t}
fun: (X: breeze.linalg.DenseMatrix[Double])breeze.linalg.DenseMatrix[Double]

scala> val test = C.t*C
test: breeze.linalg.DenseMatrix[Double] =
111.0  114.0  117.0
114.0  122.0  130.0
117.0  130.0  143.0

scala> val test_sqrt = matrix_sqrt(test)
test_sqrt: breeze.linalg.DenseMatrix[Double] =
7.165742311442667  6.0089889035545    4.852235513962529
6.008988903554499  6.370217885792754  6.731446831438616
4.852235513962529  6.731446831438616  8.610658167210904

scala> norm((test - test_sqrt * test_sqrt).toDenseVector)
res26: Double = 5.859285502108464E-14
```

5. **定义如下的一个三元正态分布：**

$$N\left(\begin{pmatrix}1.0\\2.0\\3.0\end{pmatrix},\begin{pmatrix}5.0 & 2.0 & 3.0\\2.0 & 5.0 & 4.0\\3.0 & 4.0 & 5.0\end{pmatrix}\right)$$

试使用Scala语言从上述正态分布中抽出100个样本；并计算向量$(1.0, 2.0, 3.0)$的概率密度函数值。

```
scala> import breeze.stats.distributions._
import breeze.stats.distributions._

scala> val normal3D = new MultivariateGaussian(DenseVector(1.0,2.0,3.0),DenseMatrix((5.0,2.0,3.0),
(2.0,5.0,4.0),(3.0,4.0,5.0)))
normal3D: breeze.stats.distributions.MultivariateGaussian =
MultivariateGaussian(DenseVector(1.0, 2.0, 3.0),5.0  2.0  3.0
2.0  5.0  4.0
3.0  4.0  5.0  )

scala> val RandomNumbers = normal3D.sample(100)
```

```
RandomNumbers: IndexedSeq[breeze.linalg.DenseVector[Double]] =
Vector(DenseVector(2.036073971930197, -0.766279146809214, 0.42546331533246207),
DenseVector(1.4300283001798677, 0.8349276468086608, 3.5568674939619056),
DenseVector(0.5385490552486616, 3.3588147962046158, 4.747870960696333),
DenseVector(1.2647264905335556, 1.164873491384196, 3.4734296712278208),
DenseVector(0.026933139702533748, -2.8529292972928566, -1.3678993744742618),
DenseVector(0.8896637220375545, 3.506426618345599, 2.9849602277914373),
DenseVector(-0.26096183624970837, 2.1239029892451136, 3.122787516662388),
DenseVector(2.4270667575823346, 3.6336441034218088, 4.309451989991169),
DenseVector(2.089528015073965, 1.3612350109602356, 3.1076706085830246),
DenseVector(2.6938105019613547, 2.573049990076095, 3.695850270123837),...

scala> normal3D.pdf(DenseVector(1.0,2.0,3.0))
res27: Double = 0.011999169322662555
```

第五次课

- 创建一个多元正态分布

```
scala> val normal2D = new MultivariateGaussian(DenseVector(1.0,2.0),DenseMatrix((1.0,0.5),
(0.5,1.0)))
22/10/18 19:00:01 WARN LAPACK: Failed to load implementation from:
com.github.fommil.netlib.NativeSystemLAPACK
22/10/18 19:00:01 WARN LAPACK: Failed to load implementation from:
com.github.fommil.netlib.NativeRefLAPACK
normal2D: breeze.stats.distributions.MultivariateGaussian =
MultivariateGaussian(DenseVector(1.0, 2.0),1.0  0.5
0.5  1.0  )
```

注意：均值向量与协方差矩阵要用**DenseVector与DenseMatrix**

- 对该多元正态分布进行抽样

```
scala> val RandomNumbers = normal2D.sample(5)
RandomNumbers: IndexedSeq[breeze.linalg.DenseVector[Double]] =
Vector(DenseVector(0.550564622350382, 1.0391688391676168), DenseVector(1.965332728293348,
3.681269911033965), DenseVector(0.7384144577843279, 1.8458251536061105),
DenseVector(0.7882660702181268, 2.090109483273836), DenseVector(0.7267555916048041,
2.009090587593719))
```

- 多元正态分布的均值与协方差矩阵

```
scala> normal2D.mean
res2: breeze.linalg.DenseVector[Double] = DenseVector(1.0, 2.0)

scala> normal2D.variance
res3: breeze.linalg.DenseMatrix[Double] =
1.0  0.5
0.5  1.0
```

- 求样本各点的PDF

```
scala> RandomNumbers map(normal2D.pdf(_))
res6: IndexedSeq[Double] = Vector(0.11575560791970581, 0.04425722502681081, 0.17753029490752792,
0.1751598714673228, 0.17455383143088854)

scala> val pdf_sample = normal2D.pdf(DenseVector(1.0,2.0))
pdf_sample: Double = 0.18377629847393073
```

注意：在求取单个样本的pdf时，由于是多元分布，参数**数据类型也应为DenseVector**

- 基于Breeze包的分布式计算

```
val data =
sc.parallelize(Array(DenseVector(1.0,2.0,3.0),DenseVector(4.0,5.0,6.0),DenseVector(7.0,8.0,9.0)))
data: org.apache.spark.rdd.RDD[breeze.linalg.DenseVector[Double]] = ParallelCollectionRDD[2] at
parallelize at <console>:32

val computation = data.map({s => s(1)*=2.0
     | s})      //此处返回值为s，同时由于内部的赋值，data自身也发生了改变！
computation: org.apache.spark.rdd.RDD[breeze.linalg.DenseVector[Double]] = MapPartitionsRDD[3] at
map at <console>:32

computation.reduce((x,y)=>x+y)
res7: breeze.linalg.DenseVector[Double] = DenseVector(12.0, 30.0, 18.0)
```

- 计算向量的内积之和

```
scala> val computation = data.map({s => s.t*s})
computation: org.apache.spark.rdd.RDD[Double] = MapPartitionsRDD[4] at map at <console>:32

scala> computation.collect
res8: Array[Double] = Array(14.0, 77.0, 194.0)

scala> computation.reduce((x,y)=>x+y)
res9: Double = 285.0
```

随机模拟与统计推断

- 计算机生成随机数的方法：线性同余法
- 逆累积分布函数法

定理：如果$U \sim Unif(0,1)$，令$X = F^{-1}(U)$，那么$X \sim F.$

首先导入用于生成随机数的`java.util.concurrent.ThreadLocalRandom`包

```
scala> import java.util.concurrent.ThreadLocalRandom
import java.util.concurrent.ThreadLocalRandom
```

再生成可以用于产生服从**标准正态分布**和**均匀分布**的随机变量的**函数**

```
scala> val r = ThreadLocalRandom.current
r: java.util.concurrent.ThreadLocalRandom = java.util.concurrent.ThreadLocalRandom@7af8c65a
```

- 生成$U(0,1)$分布的随机数

```
scala> r.nextDouble
res10: Double = 0.03603863620100467
```

每运行一次，则生成一个随机数

- 生成标准正态分布的随机数

```
scala> r.nextGaussian
res11: Double = 0.5731264081037789
```

- 生成1000个U(0,1)分布的随机数

```
scala> val u_1000 = (1 to 1000).map(x => r.nextDouble)
u_1000: scala.collection.immutable.IndexedSeq[Double] = Vector(0.7176295002623009,
0.1388356659495702, 0.8927162475344624, 0.15279070853936305, 0.6355366007804669,
0.04929675129144029, 0.9691212419110531, 0.9875982717806496, 0.019623070390331843,
0.358496561039783, 0.587763424400087, 0.6067608208679872, 0.13010362709094914, 0.25549313127635476,
0.8933628081582926, 0.1397866887584578, 0.7030474527969683, 0.8441247307452482, 0.5936623284316513,
0.7204066056658219, 0.5006820265805735, 0.31447684643202733, 0.36841929334569157,
0.9777313216375878, 0.7081758660764719, 0.9798457675742667, 0.4232061655928716,
0.22247382456772102, 0.14652090852423816, 0.950656228393939, 0.42101934406521324,
0.6842622975378057, 0.7204837026674152, 0.4290302529459147, 0.8920047577497079, 0.5868026925603608,
0.8031...
```

- 生成1000个服从指数分布$exp(1)$的随机数

  根据逆累积分布函数法：

$$F(x) = 1 - e^{-x}$$
$$F^{-1}(x) = -\log(1 - x)$$

  又根据对称性，我们不妨直接取：

$$F^{-1}(x) = -\log(x)$$

```
scala> val ExpRV = u_1000.map(x => -math.log(x))
ExpRV: scala.collection.immutable.IndexedSeq[Double] = Vector(0.3318018594639369,
1.9744643046269654, 0.11348650052057101, 1.8786862120870593, 0.4532855963360734, 3.009897096830126,
0.03136555426706506, 0.012479271430981483, 3.931049344207316, 1.0258362116250601,
0.5314307514993797, 0.4996206003886148, 2.039424014598268, 1.3645597531582387, 0.1127625004825973,
1.9676376700358331, 0.3523308890270931, 0.16945501006215816, 0.5214445919139814,
0.3279394962806456, 0.6917840568741717, 1.156844825799671, 0.9985336053270799, 0.02252036894234552,
0.3450628177210993, 0.02036009972931379, 0.8598958295629084, 1.5029458266885163,
1.920587140406621, 0.05060276612661479, 0.8650764984569592, 0.3794139589352258,
0.32783248327456044, 0.8462278428530695, 0.11428381261623138, 0.5330666441751657, 0.2192412...
```

  注：如果前面导入了`scala.math._`包，则可以直接`val ExpRV = u_1000.map(x => -log(x))`

Array的相关操作

- 求取Array的最值：

```
scala> ExpRV.min
res18: Double = 3.0230426234195424E-4

scala> ExpRV.max
res19: Double = 7.84998611671577
```

- 对Array进行由小到大的排序：

```
scala> ExpRV.sorted
res20: scala.collection.immutable.IndexedSeq[Double] = Vector(3.0230426234195424E-4,
0.001058763860977339, 0.002184224943962582, 0.002364997444830723, 0.0036575037316174826,
0.004575854926881684, 0.004948372909653278, 0.005262046685485586, 0.00582553725416576,
0.007101677364392974, 0.00954318765528734, 0.009747867695226469, 0.012479271430981483,
0.016513660222819235, 0.01685584933380696, 0.01715099119814802, 0.018098743234722006,
0.019842303606753833, 0.02003202307972079, 0.020070064490599147, 0.02036009972931379,
0.020364513957441047, 0.022114121993369318, 0.022295032033687007, 0.02252036894234552,
0.02366383721721807, 0.025183078495665253, 0.025452552099893572, 0.025634996810167496,
0.02597727721160488, 0.026291916079424268, 0.027415022434996404, 0.029154958306070577,
0.02932531031812...
```

- 对Array进行逆序排列：

```
scala> ExpRV.reverse
res21: scala.collection.immutable.IndexedSeq[Double] = Vector(0.3496765151669919,
0.9953710705442609, 0.2683135909784291, 2.27305433275989, 2.34161459102668, 0.002184224943962582,
0.5422439733851839, 2.3608059970299826, 0.24530247591969365, 0.10639394594106409,
0.37435082458837504, 0.5385208521528316, 0.6181892635974144, 1.639478152605218, 0.1113811062328256,
2.448263237199481, 3.530337738988229, 0.4340857564762933, 5.731928373533258, 0.3496275195979622,
0.10971423604795934, 0.3616666841003597, 2.269811979626463, 0.18697558410619275,
1.5320001339287135, 0.392276558887071, 1.3113885137386363, 0.046381131109386485, 2.058084287021981,
0.6736763763962125, 1.2241007064818015, 0.3192940885309286, 0.574989486757794, 2.2445622271844528,
0.2926645360084932, 0.9329906744234083, 0.0045758549268816...
```

- 对Array进行由大到小的排序

```
scala> ExpRV.sorted.reverse
res22: scala.collection.immutable.IndexedSeq[Double] = Vector(7.84998611671577, 5.815702168827306,
5.731928373533258, 5.56426478383095, 4.940006469081154, 4.930926800179207, 4.805090222510961,
4.757240748613276, 4.510088164279332, 4.404728434889375, 4.216739673779885, 4.123278837477409,
3.9994944629025317, 3.931049344207316, 3.904357894389848, 3.8882632447209544, 3.7873458381407055,
3.750752924394127, 3.6951866553510233, 3.6917439995842143, 3.631730087119045, 3.6056422057297355,
3.530337738988229, 3.4952137915584265, 3.439237950714319, 3.4301985853854258, 3.429879298169976,
3.362170012039244, 3.337363907160389, 3.279268763242235, 3.2664683325167423, 3.2529129828528585,
3.226661867776936, 3.210194138357579, 3.207159929232365, 3.1833758192750317, 3.1790987935626966,
3.1777632032367586, 3....
```

- 对Array进行求和与计算均值

```
scala> ExpRV.sum
res23: Double = 964.2721371243331

scala> ExpRV.sum/1000
res28: Double = 0.9642721371243331
```

- **拼接**两个Array，构成一个新Array

```
scala> Array(1,2,3).union(Array(4,5,6))
res29: Array[Int] = Array(1, 2, 3, 4, 5, 6)
```

从回归模型中模拟数据

我们考虑如下线性回归模型：

$$y_i = x_i^\top \beta + \epsilon_i, i = 1, \cdots, 400$$

其中β是一个长度为5000的稀疏向量，该向量的前10个数为2，其余为0。模型误差ϵ_i和解释变量x_i独立地产生于标准正态分布。所产生的数据集被等分为两个部分，前200数据以HDFS文件格式分别存储在不同的计算节点之上。

- `broadcast`函数：

```
scala> val RowSize = sc.broadcast(200)
RowSize: org.apache.spark.broadcast.Broadcast[Int] = Broadcast(3)
```

广播变量的作用：广播变量允许编程人员在每台机器上保持1个只读的缓存变量，而不是传送变量的副本给各个任务，即各个节点都可以读取该值。

- 收取广播变量的值

```
scala> RowSize.value
res32: Int = 200
```

- 设置基本参数

```
scala> val RowSize = sc.broadcast(200)
RowSize: org.apache.spark.broadcast.Broadcast[Int] = Broadcast(4)
```

```
scala> RowSize.value
res21: Int = 200

scala> val ColumnSize = sc.broadcast(5)
ColumnSize: org.apache.spark.broadcast.Broadcast[Int] = Broadcast(5)

scala> val RowLength = sc.broadcast(2)
RowLength: org.apache.spark.broadcast.Broadcast[Int] = Broadcast(6)

scala> val ColumnLength = sc.broadcast(1000)
ColumnLength: org.apache.spark.broadcast.Broadcast[Int] = Broadcast(7)

scala> val NonZeroLength = 10
NonZeroLength: Int = 10

scala> val p = ColumnSize.value*ColumnLength.value
p: Int = 5000
```

- 生成β:

```
scala> val beta = (1 to p).map(_.toDouble).toArray[Double].map(i => {if(i<=NonZeroLength) 2.0 else
0.0})
beta: Array[Double] = Array(2.0, 2.0, 2.0, 2.0, 2.0, 2.0, 2.0, 2.0, 2.0, 2.0, 0.0, 0.0, 0.0, 0.0,
0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
0.0,...
```

> 注1：`(a to b).map()`一般连用，所以`.map(_.toDouble)`是必须的；
>
> 注2：`(1 to p)`得到的是一个迭代器，所以需要转换为Array

```
scala> (1 to p)
res33: scala.collection.immutable.Range.Inclusive = Range 1 to 5000
```

> 注3：可以利用if语句生成β

- 再将β广播出去，从而使得每个节点都可以读取该值：

```
scala> val MyBeta = sc.broadcast(beta)
MyBeta: org.apache.spark.broadcast.Broadcast[Array[Double]] = Broadcast(8)
```

- 定义误差项ϵ_i的标准差，并将其广播，使得每个节点都可以读取该值：

```
scala> val sigma = 1
sigma: Int = 1

scala> val Sigma = sc.broadcast(sigma)
Sigma: org.apache.spark.broadcast.Broadcast[Int] = Broadcast(10)
```

- 构建RDD（但根据题意“前200数据以HDFS文件格式分别存储在不同的计算节点之上”，只需要构建两个节点）

Step1：构建代表两个节点的（0，1）向量

```
scala> var indices = 0 until RowLength.value
indices: scala.collection.immutable.Range = Range 0 until 2
```

Step2：根据该向量构建RDD

```
scala> var ParallelIndices = sc.parallelize(indices, indices.length)
ParallelIndices: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[4] at parallelize at
<console>:34
```

分析该RDD的成分：

```
scala> ParallelIndices.collect()
res38: Array[Int] = Array(0, 1)
```

- 定义**产生标准正态分布样本**的函数

```
scala> def rn(n:Int)=(0 until n).map(x => r.nextGaussian).toArray[Double]
rn: (n: Int)Array[Double]
```

注意：产生样本后，要将样本数据变为Array

例如，生成容量为5的标准正态分布样本：

```
scala> rn(5)
res40: Array[Double] = Array(-0.44652385440817566, -0.07031891905526692, 0.21969961812934452,
0.33206048266539523, -0.7315344655965659)
```

- 开始构建线性模型

```
scala> val lines = ParallelIndices.map(s => {      //对两个节点分别操作
     | val beta = MyBeta.value                     //将各个广播变量的值收回，以便在各个节点上使用
     | val sigma = Sigma.value
     | val rowsize = RowSize.value
     | val columnsize = ColumnSize.value
     | val columnlength = ColumnLength.value
     | var lines = new Array[String](rowsize)       //最后结果以字符串形式存储，故将每次生成的x以字符串保存
     | val p = columnsize*columnlength
     | for(i <- 0 until rowsize)                    //首先对每一行迭代，共rowsize(200)行
     | {
     |     var line=""                              //先将每一行定义为一个空字符串
     |     var y=0.0                                //初始化响应变量yi的值为0
     |     for(j <- 0 until columnlength)           //再对第i行中的各个块进行迭代，共columnlength(500)块
     |     {
     |         var x = rn(columnsize)               //其中第j块为容量为columnsize(5)的标准正态分布样本
     |         for(k <- 0 until columnsize)         //将第j块中五个协变量的贡献值加到y上
     |         {
     |             y = y + beta(j*columnsize+k)*x(k)
     |         }
     |         var segment = x.map("%.4f" format _).reduce(_+" "+_)
                                                    //将第j块中的五个协变量各保留4位小数，并进行合并
     |         line = line + "," + segment          //将各块进行合并
     |     }
     |     y = y + sigma*r.nextGaussian             //给y加上误差项
     |     lines(i)="%.4f".format(y)+line+"\n"      //合并响应变量y和各个协变量x的值，并存入向量lines中
     | }
     | lines.reduce(_+_)                            //合并lines中rowsize(200)个字符型变量为一个元素
     | })
lines: org.apache.spark.rdd.RDD[String] = MapPartitionsRDD[6] at map at <console>:43
```

- 分析lines的成分
    - lines本质为字符型变量

```
scala> lines.collect()
res43: Array[String] =
Array("-1.4888,-0.0619 -0.8042 0.8398 0.1259 0.0728,-0.0663 0.4882 -0.9305 -0.0537
-0.0046,0.6067 0.0206 -2.7905 0.2679 -0.1165,-0.4167 0.9803 -1.0587 -0.1742 -0.1471,0.9865
1.1706 -0.0390 0.8237 -1.2946,0.5654 0.7944 -0.0450 -1.2950 -1.0164,-0.5628 -0.6562 0.7866
-0.3881 0.2339,0.7519 -0.2437 1.5286 0.8851 -1.2688,-0.2268 1.4386 0.8253 -0.1302
-0.2733,1.0492 0.2491 -0.0163 1.1585 -0.2293,1.2609 -0.1895 -0.3550 0.8161 -0.0231,1.6633
1.1343 -1.8831 -2.0853 -0.5691,-0.9909 -0.2893 0.0091 -1.4993 -0.3618,-0.1081 -1.5002 0.8231
0.4146 -0.3335,-0.4292 0.6517 -0.6424 0.9721 -0.4412,-1.0004 0.8991 0.4253 -1.2089
-0.1976,1.1045 -0.8534 1.0936 0.5794 -2.6023,-1.1417 -0.4207 -3.0374 1.5857 -0.0566,1.8025
-0.2368 -1.2296 -1.9228 -0.0511,-2.1225 0.4223 0.7563 -0.3821 -1.0444...
```

- lines仅有两个元素

```
scala> lines.collect().size
res44: Int = 2
```

- 分析lines各个元素的特点

```
scala> lines.collect()(0)
res45: String =
"-2.1403,-0.1348 0.7145 0.0415 -1.0462 -0.0775,0.4207 -0.7381 -1.2420 0.1985 1.2539,-0.2596 1.7011
1.8210 -1.0155 -0.3425,-0.8982 0.3485 -0.6411 -2.7227 -1.3576,-0.9021 0.7076 0.0172 0.4650
0.3450,0.8586 -2.4288 2.8228 0.0882 -1.2571,2.1775 0.5468 0.3027 1.3107 -1.2389,0.2140 1.8136
0.3944 0.9345 2.1870,-1.0786 -0.1858 0.9750 0.2642 -1.3535,1.3487 0.3747 -0.2455 0.3145
1.4811,-0.6987 0.8446 0.0952 1.9783 -0.2888,0.6453 -0.6181 -0.3373 0.2051 0.5279,-3.4822 0.5233
0.6482 -0.0370 -0.0720,0.7273 1.5815 -0.0096 -0.1136 2.0157,0.6001 -0.5844 -0.6785 -1.4875
-1.2317,-0.3168 -1.8176 0.9053 -0.7445 0.2577,-0.4031 1.1872 1.0479 -0.5599 1.1339,1.2897 -0.4951
-0.5672 -0.6463 -0.3408,1.3436 1.6401 -0.0700 -0.6974 0.2588,-0.6889 0.7088 1.8751 -0.0177
-1.4088,-0.9804 1.8617 -2.8548 0...
```

注意到，`lines.collect()(0)`与`lines.collect()`的结果不同，本质原因为：**惰性调用**，每一次运行会生成新的随机数

- 观察lines各个元素的长度

```
scala> lines.collect()(0).size
res46: Int = 7501336
```

注意到：并不是5001或其余数值，原因在于字符串的长度为其中的字符个数：

```
scala> "string".size
res48: Int = 6
```

下面考虑在线性模型例题中几个特殊的函数：

- **声明一个新的Array**

```
val a = new Array[String](n)
```

其中`String`可以换成其余的数据类型，结果会返回n个null组成的Array

```
scala> new Array[String](5)
res42: Array[String] = Array(null, null, null, null, null)
```

- 在map函数中**给一个Double型向量各元素均保留四位小数**

利用：`a.map("%.4f" format _)`函数

```
scala> val a = Array(1.23456,2.51185,8.22513,9.12648)
a: Array[Double] = Array(1.23456, 2.51185, 8.22513, 9.12648)

scala> a.map("%.4f" format _)
res52: Array[String] = Array(1.2346, 2.5119, 8.2251, 9.1265)
```

注意：在map里，format前后必须有**空格**，且返回结果为Array[String]

其中：`%.4`代表保留4位小数，`f`代表为浮点型数据，保留方式为四舍五入

- 给一个**Double型变量**保留四位小数

  利用`"%.4f".format(y)`函数

```
scala> val y = 1.234823
y: Double = 1.234823

scala> "%.4f".format(y)
res53: String = 1.2348
```

注意到，在map函数之外，需要用format的函数形式

- 关于线性模型例题的两个问题
  - *Question1*

> 震宇：这里在构建RDD的时候，直接sc.parallelize(indices)不就可以生成两个节点了吗？加indices.length的意义是什么呢？
>
> ```
> scala> var ParallelIndices = sc.parallelize(indices, indices.length)
> ParallelIndices: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[4] at parallelize
> at <console>:34
> ```
>
> 大川：如果sc.parallelize里面是由一个参数，意思是把这个参数代表的数据集分散到现有的多个节点上去，但是你决定不了到底分配到具体几个节点，如果你加了第二个参数，表示specify了你要分配的节点的个数
>
> 震宇：明白了，虽然有两个数据，但还是有可能分到一个节点上，所以人为指定一下
>
> 大川：可以这么理解，或者说，如果你有10个节点，但你现在只想用5个节点，你可以在第二个参数上写个5

  - *Question2*

> 震宇：在这里我们两次用到reduce(+)的语法，但是分布式运算中，难道不是"乱着加"的吗？怎么能保证这里是按"x1 x2 x3 x4 x5"的顺序合并呢？
>
> ```
> var segment = x.map("%.4f" format _).reduce(_+" "+_)
> lines.reduce(_+_)
> ```
>
> 大川：这也是个好问题，这里的这个reduce里面，你要是自己分析上下文的话，你会发现，它是用来串行使用的，在整个代码里面，只有最外面的那个map是并行的
>
> 大川：串行程序里也是可以用map和reduce的
>
> 震宇：是因为现在在同一个节点上，所以是个串行的，就类似于foldLeft的原理吗
>
> 大川：嗯嗯，对的

- 存储字符串数据集

  首先导入用于输出的`java.io._`包：

```
scala> import java.io._
import java.io._
```

文件创建：

```
scala> val pw = new PrintWriter(new File("scala_output.txt"))
pw: java.io.PrintWriter = java.io.PrintWriter@783c7d01
```

文件写入：

```
scala> pw.write(lines.collect()(0))
```

最后要关闭文件：

```
scala> pw.close
```

第六次课

文件读写与输出

- 将*第五次课*中的lines变量输出到不同文件中

首先，重新生成以分布式存储的变量lines，并对其做一些改变：

```
scala> import java.util.concurrent.ThreadLocalRandom
import java.util.concurrent.ThreadLocalRandom

scala> val RowSize = sc.broadcast(200)
RowSize: org.apache.spark.broadcast.Broadcast[Int] = Broadcast(0)

scala> val ColumnSize = sc.broadcast(5)
ColumnSize: org.apache.spark.broadcast.Broadcast[Int] = Broadcast(1)

scala> val RowLength = sc.broadcast(4)                          //(1)
RowLength: org.apache.spark.broadcast.Broadcast[Int] = Broadcast(2)

scala> val ColumnLength = sc.broadcast(1000)
ColumnLength: org.apache.spark.broadcast.Broadcast[Int] = Broadcast(3)

scala> val NonZeroLength = 10
NonZeroLength: Int = 10

scala> val p = ColumnSize.value * ColumnLength.value
p: Int = 5000

scala> val beta = (1 to p).map(_.toDouble).toArray[Double].map(i => {if(i<NonZeroLength+1) 2.0 else
0.0})
beta: Array[Double] = Array(2.0, 2.0, 2.0, 2.0, 2.0, 2.0, 2.0, 2.0, 2.0, 2.0, 0.0, 0.0, 0.0, 0.0,
0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
0.0,...

scala> val MyBeta = sc.broadcast(beta)
MyBeta: org.apache.spark.broadcast.Broadcast[Array[Double]] = Broadcast(4)

scala> val sigma = 1.0
sigma: Double = 1.0

scala> val Sigma = sc.broadcast(sigma)
Sigma: org.apache.spark.broadcast.Broadcast[Double] = Broadcast(5)

scala> var indices = 0 until RowLength.value
indices: scala.collection.immutable.Range = Range 0 until 4
```

```
scala> var ParallelIndices = sc.parallelize(indices, indices.length)
ParallelIndices: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[0] at parallelize at
<console>:34

scala> var lines = ParallelIndices.map(s => {
     |     val r = ThreadLocalRandom.current
     |     def rn(n: Int) = (0 until n).map(x => r.nextGaussian).toArray[Double]
     |     val beta = MyBeta.value
     |     val sigma = Sigma.value
     |     val rowsize = RowSize.value
     |     val columnsize = ColumnSize.value
     |     val columnlength = ColumnLength.value
     |     val lines = new Array[String](rowsize)
     |     val p = columnsize * columnlength
     |     for(i <- 0 until rowsize)
     |     {
     |         var line = "";
     |         var y = 0.0;
     |         for(j <- 0 until columnlength)
     |         {
     |             var x = rn(columnsize)
     |             for(k <- 0 until columnsize) y+=beta(j*columnsize + k)*x(k)
     |             var segment = x.map("%.4f" format _).reduce(_+" "+_)
     |             line = line+","+segment
     |         }
     |         y+= sigma*r.nextGaussian
     |         lines(i) = "%.4f".format(y) + line + "\n"
     |     }
     |     (s,lines.reduce(_+_))                                    //(2)
     | })
lines: org.apache.spark.rdd.RDD[(Int, String)] = MapPartitionsRDD[1] at map at <console>:40
```

注意到，相较于第五次课lines的定义，此处做了两个改变：

1. `val RowLength = sc.broadcast(4)`，即：原本RowLength为2，此处改为4；

```
scala> lines.collect().size
res0: Int = 4
```

2. 在输出lines时，此处改为以元组的格式输出，即：`(s,lines.reduce(_+_))`。

思考1：为什么此处“组合输出”?

解答：因为我们希望在生成四个文件时，分别加以对应的序号，即`LinearModel_1.txt`，`LinearModel_2.txt`，`LinearModel_3.txt`，`LinearModel_4.txt`，因此在输出时，同序号“组合输出”。

思考2：为什么此处用元组而非列表、向量?

解答：因为元组的各个元素可以是不同类型的！

思考3：`reduce`在并行使用时（对RDD使用）乱序执行，串行使用时按顺序执行

```
scala> val aaa = Array("1","2","3","4")
aaa: Array[String] = Array(1, 2, 3, 4)

scala> val rdd = sc.parallelize(aaa)
rdd: org.apache.spark.rdd.RDD[String] = ParallelCollectionRDD[2] at parallelize at <console>:31

scala> rdd.reduce(_+" "+_)
res3: String = 1 2 4 3

scala> rdd.reduce(_+" "+_)
res4: String = 3 2 4 1
```

下面开始输出文件：

Step1: 引入输出文件的`java.io._`包:

```
scala> import java.io._
import java.io._
```

Step2: 将分布式存储的变量输出到多个文件中:

```
scala> val output = lines.collect().map(s =>{
     | val path = "LinearModel_"+s._1+".txt"
     | val pw = new PrintWriter(new File(path))
     | pw.write(s._2)
     | pw.close
     | path
     | })
output: Array[String] = Array(LinearModel_0.txt, LinearModel_1.txt, LinearModel_2.txt,
LinearModel_3.txt)
```

思考1: 本质上可以直接利用:

```
scala> lines.collect().map(s => {
     | val path = "LinearModel_"+s._1+".txt";
     | val pw = new PrintWriter(new File(path));
     | pw.write(s._2);
     | pw.close;
     | })
res4: Array[Unit] = Array((), (), (), ())
```

解答: 因为`lines.collect()`本身是一个`Array[(Int, String)]`，因此`map`函数中的`s`代表各个`Array[(Int, String)]`，故经过上述变换（仅有生成文件的操作）后，不会返回任何"值"，即返回"空"，故得到的是`Array((), (), (), ())`。因此，我们可以在最后加一句`path`，即返回文件名；再定义一个`output`变量，表示由所有文件名组成的Array。

注意1: 这里在使用`map`函数时，有必要先使用`collect()`，否则无法输出。

注意2: 元组索引从1开始!

- 读入文件

```
scala> val lines = sc.textFile("C:\\Users\\zywang\\LinearModel_0.txt")
lines: org.apache.spark.rdd.RDD[String] = C:\Users\zywang\LinearModel_0.txt MapPartitionsRDD[10] at
textFile at <console>:30
```

一次只能读入一个文件，且读入文件的格式为RDD；在读入文件时，注意Windows下"\"是转义符，所以地址分隔符应该用"\\"。

分析文件内容:

```
scala> lines.collect()
res17: Array[String] = Array(2.7601,-1.1386 0.4835 1.3637 1.4924 0.0893,1.0956 -0.7946 -1.1573
-0.2077 0.5193,-0.1034 -1.1892 0.9378 -0.6069 0.5018,-0.4095 -1.2792 -0.6688 1.3650 -0.0193,0.1869
-0.7279 -1.1468 0.1423 0.5688,0.7074 -0.3836 -0.0880 -0.8235 0.0690,-0.7025 -0.0846 -1.2633 -1.4699
-0.3945,0.5877 -0.7004 0.4383 -0.4180 0.0509,-0.3405 -0.7271 -1.2959 1.8048 -1.1041,-0.4107 0.6790
-2.3475 0.5713 0.5163,0.2697 0.9991 1.3726 0.9523 0.3381,-0.8825 0.0005 -0.1102 0.0188
1.5751,1.7053 0.9707 0.6304 1.6821 1.4498,0.9811 1.5589 -0.1066 -0.0939 -0.6646,-0.1712 1.4133
-0.7175 1.3502 0.4262,-1.4992 -1.6341 0.7580 -0.0490 0.0098,-0.3205 1.4855 -0.4139 0.4842
-1.3736,-0.4473 -0.8911 1.0252 1.1189 -0.0654,1.3792 -0.7258 -1.0709 0.2250 -1.0008,0.2312 0.0254
0.3661 0.8850 1.1391,-2.3878 0.855...

scala> lines.collect().size
res18: Int = 200
```

- 舍弃Array的前n个数: `drop(n)`

```
scala> Array(0,1).drop(1)
res58: Array[Int] = Array(1)

scala> Array(0,1).drop(2)
res60: Array[Int] = Array()

scala> Array(0,1,2,3,4,5,6).drop(2)
res61: Array[Int] = Array(2, 3, 4, 5, 6)
```

注意：`drop(n)`不是舍弃第n个数，而是舍弃前n个数，所以自然从1而非0开始计数。

- **将上述字符串转换为Array[Double]**（以第一行为例）

首先以"，"为分隔符，将lines的第一行分割为1001个**字符串**：

```
scala> lines.collect()(0).split(",")
res19: Array[String] = Array(2.7601, -1.1386 0.4835 1.3637 1.4924 0.0893, 1.0956 -0.7946 -1.1573
-0.2077 0.5193, -0.1034 -1.1892 0.9378 -0.6069 0.5018, -0.4095 -1.2792 -0.6688 1.3650 -0.0193,
0.1869 -0.7279 -1.1468 0.1423 0.5688, 0.7074 -0.3836 -0.0880 -0.8235 0.0690, -0.7025 -0.0846
-1.2633 -1.4699 -0.3945, 0.5877 -0.7004 0.4383 -0.4180 0.0509, -0.3405 -0.7271 -1.2959 1.8048
-1.1041, -0.4107 0.6790 -2.3475 0.5713 0.5163, 0.2697 0.9991 1.3726 0.9523 0.3381, -0.8825 0.0005
-0.1102 0.0188 1.5751, 1.7053 0.9707 0.6304 1.6821 1.4498, 0.9811 1.5589 -0.1066 -0.0939 -0.6646,
-0.1712 1.4133 -0.7175 1.3502 0.4262, -1.4992 -1.6341 0.7580 -0.0490 0.0098, -0.3205 1.4855 -0.4139
0.4842 -1.3736, -0.4473 -0.8911 1.0252 1.1189 -0.0654, 1.3792 -0.7258 -1.0709 0.2250 -1.0008,
0.2312 0.0254 0.3661 0.8850 ...

scala> lines.collect()(0).split(",").size
res21: Int = 1001
```

再将响应变量单独存储，由于此时元素为字符串，因此还需要转换为Double型变量：

```
scala> val Y = lines.collect()(0).split(",")(0).toDouble
Y: Double = 2.7601
```

再将剩下1000个字符型变量存储到`X`中：

```
scala> val X = lines.collect()(0).split(",").drop(1)
X: Array[String] = Array(-1.1386 0.4835 1.3637 1.4924 0.0893, 1.0956 -0.7946 -1.1573 -0.2077
0.5193, -0.1034 -1.1892 0.9378 -0.6069 0.5018, -0.4095 -1.2792 -0.6688 1.3650 -0.0193, 0.1869
-0.7279 -1.1468 0.1423 0.5688, 0.7074 -0.3836 -0.0880 -0.8235 0.0690, -0.7025 -0.0846 -1.2633
-1.4699 -0.3945, 0.5877 -0.7004 0.4383 -0.4180 0.0509, -0.3405 -0.7271 -1.2959 1.8048 -1.1041,
-0.4107 0.6790 -2.3475 0.5713 0.5163, 0.2697 0.9991 1.3726 0.9523 0.3381, -0.8825 0.0005 -0.1102
0.0188 1.5751, 1.7053 0.9707 0.6304 1.6821 1.4498, 0.9811 1.5589 -0.1066 -0.0939 -0.6646, -0.1712
1.4133 -0.7175 1.3502 0.4262, -1.4992 -1.6341 0.7580 -0.0490 0.0098, -0.3205 1.4855 -0.4139 0.4842
-1.3736, -0.4473 -0.8911 1.0252 1.1189 -0.0654, 1.3792 -0.7258 -1.0709 0.2250 -1.0008, 0.2312
0.0254 0.3661 0.8850 1.1391, -2.3...
```

即利用`drop`函数舍弃第一个响应变量的值；

再以空格为分隔符，分割各个变量中的五个值：

```
scala> X.map(_.split(" "))
res24: Array[Array[String]] = Array(Array(-1.1386, 0.4835, 1.3637, 1.4924, 0.0893), Array(1.0956,
-0.7946, -1.1573, -0.2077, 0.5193), Array(-0.1034, -1.1892, 0.9378, -0.6069, 0.5018),
Array(-0.4095, -1.2792, -0.6688, 1.3650, -0.0193), Array(0.1869, -0.7279, -1.1468, 0.1423, 0.5688),
Array(0.7074, -0.3836, -0.0880, -0.8235, 0.0690), Array(-0.7025, -0.0846, -1.2633, -1.4699,
-0.3945), Array(0.5877, -0.7004, 0.4383, -0.4180, 0.0509), Array(-0.3405, -0.7271, -1.2959, 1.8048,
-1.1041), Array(-0.4107, 0.6790, -2.3475, 0.5713, 0.5163), Array(0.2697, 0.9991, 1.3726, 0.9523,
0.3381), Array(-0.8825, 0.0005, -0.1102, 0.0188, 1.5751), Array(1.7053, 0.9707, 0.6304, 1.6821,
1.4498), Array(0.9811, 1.5589, -0.1066, -0.0939, -0.6646), Array(-0.1712, 1.4133, -0.7175, 1.3502,
0.4262), Array(-1.4992, -1.63...
```

思考：为什么不能直接用`X.split`？

解答：考虑`split(" a ")`函数的用法：将"**一个字符型变量**"以a为分隔符进行分割。而在此处，X的数据形式为`Array[String]`，即由1000个字符串组成的向量，不满足"一个"的条件，因此用`map`函数。

但是此时得到的数据类型为`Array[Array[String]]`，而我们需要得到的是一个`Array[Double]`。因此，首先将各个元素转换为Double型变量：

方法1：

```
scala> X.map(_.split(" ")).map(_.map(_.toDouble))
res26: Array[Array[Double]] = Array(Array(-1.1386, 0.4835, 1.3637, 1.4924, 0.0893), Array(1.0956,
-0.7946, -1.1573, -0.2077, 0.5193), Array(-0.1034, -1.1892, 0.9378, -0.6069, 0.5018),
Array(-0.4095, -1.2792, -0.6688, 1.365, -0.0193), Array(0.1869, -0.7279, -1.1468, 0.1423, 0.5688),
Array(0.7074, -0.3836, -0.088, -0.8235, 0.069), Array(-0.7025, -0.0846, -1.2633, -1.4699, -0.3945),
Array(0.5877, -0.7004, 0.4383, -0.418, 0.0509), Array(-0.3405, -0.7271, -1.2959, 1.8048, -1.1041),
Array(-0.4107, 0.679, -2.3475, 0.5713, 0.5163), Array(0.2697, 0.9991, 1.3726, 0.9523, 0.3381),
Array(-0.8825, 5.0E-4, -0.1102, 0.0188, 1.5751), Array(1.7053, 0.9707, 0.6304, 1.6821, 1.4498),
Array(0.9811, 1.5589, -0.1066, -0.0939, -0.6646), Array(-0.1712, 1.4133, -0.7175, 1.3502, 0.4262),
Array(-1.4992, -1.6341, 0...
```

分析：第一个`map`指对Array[Array[String]]进行操作；但此时仍然是一个字符型向量，因此用第二层`map`对Array[Array[String]]进行操作。

方法2：

```
scala> X.map(_.split(" ").map(_.toDouble))
res38: Array[Array[Double]] = Array(Array(-1.1386, 0.4835, 1.3637, 1.4924, 0.0893), Array(1.0956,
-0.7946, -1.1573, -0.2077, 0.5193), Array(-0.1034, -1.1892, 0.9378, -0.6069, 0.5018),
Array(-0.4095, -1.2792, -0.6688, 1.365, -0.0193), Array(0.1869, -0.7279, -1.1468, 0.1423, 0.5688),
Array(0.7074, -0.3836, -0.088, -0.8235, 0.069), Array(-0.7025, -0.0846, -1.2633, -1.4699, -0.3945),
Array(0.5877, -0.7004, 0.4383, -0.418, 0.0509), Array(-0.3405, -0.7271, -1.2959, 1.8048, -1.1041),
Array(-0.4107, 0.679, -2.3475, 0.5713, 0.5163), Array(0.2697, 0.9991, 1.3726, 0.9523, 0.3381),
Array(-0.8825, 5.0E-4, -0.1102, 0.0188, 1.5751), Array(1.7053, 0.9707, 0.6304, 1.6821, 1.4498),
Array(0.9811, 1.5589, -0.1066, -0.0939, -0.6646), Array(-0.1712, 1.4133, -0.7175, 1.3502, 0.4262),
Array(-1.4992, -1.6341, 0...
```

分析：在第一层`map`里，执行对象为Array[Array[String]]，因此再直接套用第二层`map`即可对Array[Array[String]]进行操作。

最后，将1000个`Array[Double]`合并：

方法1：

```
scala> X.map(_.split(" ")).map(_.map(_.toDouble)).reduce((x,y) => x.union(y))
res27: Array[Double] = Array(-1.1386, 0.4835, 1.3637, 1.4924, 0.0893, 1.0956, -0.7946, -1.1573,
-0.2077, 0.5193, -0.1034, -1.1892, 0.9378, -0.6069, 0.5018, -0.4095, -1.2792, -0.6688, 1.365,
-0.0193, 0.1869, -0.7279, -1.1468, 0.1423, 0.5688, 0.7074, -0.3836, -0.088, -0.8235, 0.069,
-0.7025, -0.0846, -1.2633, -1.4699, -0.3945, 0.5877, -0.7004, 0.4383, -0.418, 0.0509, -0.3405,
-0.7271, -1.2959, 1.8048, -1.1041, -0.4107, 0.679, -2.3475, 0.5713, 0.5163, 0.2697, 0.9991, 1.3726,
0.9523, 0.3381, -0.8825, 5.0E-4, -0.1102, 0.0188, 1.5751, 1.7053, 0.9707, 0.6304, 1.6821, 1.4498,
0.9811, 1.5589, -0.1066, -0.0939, -0.6646, -0.1712, 1.4133, -0.7175, 1.3502, 0.4262, -1.4992,
-1.6341, 0.758, -0.049, 0.0098, -0.3205, 1.4855, -0.4139, 0.4842, -1.3736, -0.4473, -0.8911,
1.0252, 1.1189, -0.0654, 1.3792, -0...

scala> X.map(_.split(" ")).map(_.map(_.toDouble)).reduce((x,y) => x.union(y)).size
res34: Int = 5000
```

方法2：

```
scala> x.map(_.split(" ").map(_.toDouble)).foldLeft(Array(0.0)){(x,y) => x.union(y)}
res36: Array[Double] = Array(0.0, -1.1386, 0.4835, 1.3637, 1.4924, 0.0893, 1.0956, -0.7946,
-1.1573, -0.2077, 0.5193, -0.1034, -1.1892, 0.9378, -0.6069, 0.5018, -0.4095, -1.2792, -0.6688,
1.365, -0.0193, 0.1869, -0.7279, -1.1468, 0.1423, 0.5688, 0.7074, -0.3836, -0.088, -0.8235, 0.069,
-0.7025, -0.0846, -1.2633, -1.4699, -0.3945, 0.5877, -0.7004, 0.4383, -0.418, 0.0509, -0.3405,
-0.7271, -1.2959, 1.8048, -1.1041, -0.4107, 0.679, -2.3475, 0.5713, 0.5163, 0.2697, 0.9991, 1.3726,
0.9523, 0.3381, -0.8825, 5.0E-4, -0.1102, 0.0188, 1.5751, 1.7053, 0.9707, 0.6304, 1.6821, 1.4498,
0.9811, 1.5589, -0.1066, -0.0939, -0.6646, -0.1712, 1.4133, -0.7175, 1.3502, 0.4262, -1.4992,
-1.6341, 0.758, -0.049, 0.0098, -0.3205, 1.4855, -0.4139, 0.4842, -1.3736, -0.4473, -0.8911,
1.0252, 1.1189, -0.0654, 1.379...
```

注意：此处若用 `foldLeft` 函数，初始值一定要和后面 `(x,y)` 中的 x 保持一致，即 `Array[Double]`。

再将初始值0舍弃即可：

```
scala> x.map(_.split(" ").map(_.toDouble)).foldLeft(Array(0.0)){(x,y) => x.union(y)}.drop(1)
res37: Array[Double] = Array(-1.1386, 0.4835, 1.3637, 1.4924, 0.0893, 1.0956, -0.7946, -1.1573,
-0.2077, 0.5193, -0.1034, -1.1892, 0.9378, -0.6069, 0.5018, -0.4095, -1.2792, -0.6688, 1.365,
-0.0193, 0.1869, -0.7279, -1.1468, 0.1423, 0.5688, 0.7074, -0.3836, -0.088, -0.8235, 0.069,
-0.7025, -0.0846, -1.2633, -1.4699, -0.3945, 0.5877, -0.7004, 0.4383, -0.418, 0.0509, -0.3405,
-0.7271, -1.2959, 1.8048, -1.1041, -0.4107, 0.679, -2.3475, 0.5713, 0.5163, 0.2697, 0.9991, 1.3726,
0.9523, 0.3381, -0.8825, 5.0E-4, -0.1102, 0.0188, 1.5751, 1.7053, 0.9707, 0.6304, 1.6821, 1.4498,
0.9811, 1.5589, -0.1066, -0.0939, -0.6646, -0.1712, 1.4133, -0.7175, 1.3502, 0.4262, -1.4992,
-1.6341, 0.758, -0.049, 0.0098, -0.3205, 1.4855, -0.4139, 0.4842, -1.3736, -0.4473, -0.8911,
1.0252, 1.1189, -0.0654, 1.3792, -0...
```

- 生成随机数的第三种方法：

  先调用两个机器学习相关的包：

```
scala> import org.apache.spark.SparkContext
import org.apache.spark.SparkContext

scala> import org.apache.spark.mllib.random.RandomRDDs._
import org.apache.spark.mllib.random.RandomRDDs._
```

  优点：生成的**以随机数构成的向量**自动以RDD形式存储。

  例如：从Poisson(1)分布中生成100万个随机数，并将其十等分存储：

```
scala> val u = poissonRDD(sc, 1, 1000000L, 10)
u: org.apache.spark.rdd.RDD[Double] = RandomRDD[10] at RDD at RandomRDD.scala:42

scala> u.collect()
res62: Array[Double] = Array(0.0, 1.0, 0.0, 3.0, 2.0, 2.0, 2.0, 0.0, 1.0, 3.0, 1.0, 0.0, 0.0, 0.0,
0.0, 0.0, 1.0, 0.0, 2.0, 2.0, 2.0, 2.0, 1.0, 0.0, 1.0, 1.0, 2.0, 1.0, 0.0, 3.0, 3.0, 1.0, 0.0, 0.0,
1.0, 3.0, 0.0, 2.0, 0.0, 1.0, 1.0, 1.0, 1.0, 2.0, 1.0, 2.0, 1.0, 2.0, 1.0, 1.0, 0.0, 0.0, 0.0, 2.0,
1.0, 1.0, 0.0, 1.0, 1.0, 0.0, 0.0, 3.0, 0.0, 2.0, 2.0, 0.0, 1.0, 3.0, 1.0, 0.0, 2.0, 1.0, 0.0, 3.0,
0.0, 0.0, 3.0, 1.0, 1.0, 0.0, 0.0, 1.0, 0.0, 0.0, 2.0, 2.0, 1.0, 2.0, 0.0, 2.0, 1.0, 2.0, 1.0, 2.0,
3.0, 2.0, 2.0, 2.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 2.0, 0.0, 1.0, 0.0, 0.0, 2.0,
1.0, 1.0, 4.0, 0.0, 2.0, 0.0, 1.0, 0.0, 2.0, 1.0, 2.0, 3.0, 0.0, 1.0, 0.0, 1.0, 0.0, 2.0, 0.0, 1.0,
1.0, 0.0, 0.0, 0.0, 3.0, 0.0, 3.0, 1.0, 1.0, 0.0, 1.0, 0.0, 0.0, 1.0, 1.0, 1.0, 1.0, 1.0, 3.0,
1.0...

scala> u.collect().size
res63: Int = 1000000
```

- 用于生成服从特定分布的随机数的函数：

```
LogNormal: logNormalRDD(sc, mean, std, 1000, 10)
Exponential: exponentialRDD(sc, mean, 1000, 10)
Standard Normal: normalRDD(sc, 1000, 10)
Uniform: uniformRDD(sc, 1000, 10)
Uniform Vectors: uniformVectorRDD(sc, num, dim, 100, 10)
```

注：上述`uniformVectorRDD`函数中参数100代表生成随机数的种子。

EM算法

- EM算法：

  假设全部数据Y由两部分组成，即$Y=(X,Z)$，其中X是完整观察数据，而Z是缺失数据。

  让$f_X(x\mid\theta)$和$f_Y(x\mid\theta)$分别表示完整观察数据和全部数据的概率密度函数。那么缺失数据给定观察数据的条件概率密度函数是：

$$f_{Z|X}(z\mid x,\theta)=\frac{f_Y(y\mid\theta)}{f_X(x\mid\theta)}$$

  其中$y=(x,z)$.

  EM算法的主要思想是希望找到一个θ的估计从而最大化似然函数的条件期望$E_{Z|X}\{L(\theta\mid x)\mid X\}$.

  我们让$\theta^{(k)}$表示在第k个循环得到的解，然后我们定义：

$$\begin{aligned}Q\left(\theta\mid\theta^{(k)}\right)&=E_{Z|X}\left\{\log f_Y(y\mid\theta)\mid x,\theta^{(k)}\right\}\\&=\int\{\log f_Y(y\mid\theta)\}f_{Z|X}\left(z\mid x,\theta^{(k)}\right)dz\end{aligned}$$

  那么*EM(Expectation-Maximization)*算法可以简单的表示为：

  (1) E-step: 计算$Q(\theta\mid\theta^{(t)})$；

  (2) M-step: 最大化$Q(\theta\mid\theta^{(t)})$，记其解为$Q(\theta^{(t+1)})$；

  (3) 重复以上步骤直到某些停止标准被满足。（例如：$|\theta^{(t+1)}-\theta^{(t)}|<\epsilon$）

- 算法原理：

  注意到

$$\log f_X(x\mid\theta)=\log f_Y(y\mid\theta)-\log f_{Z|X}(z\mid x,\theta)$$

  那么我们可以得到下面的条件期望：

$$\begin{aligned}&E_{Z|X}\left\{\log f_X(x\mid\theta)\mid x,\theta^{(t)}\right\}\\=&\log f_X(x\mid\theta)\\=&Q\left(\theta\mid\theta^{(t)}\right)-H\left(\theta\mid\theta^{(t)}\right)\end{aligned}$$

  其中

$$\begin{aligned}Q\left(\theta\mid\theta^{(t)}\right)&=E_{Z|X}\left\{\log f_Y(y\mid\theta)\mid x,\theta^{(t)}\right\}\\H\left(\theta\mid\theta^{(t)}\right)&=E_{Z|X}\left\{\log f_{Z|X}(z\mid x,\theta)\mid x,\theta^{(t)}\right\}\end{aligned}$$

  我们可以进一步得到：

$$\begin{aligned}&H\left(\theta^{(t)}\mid\theta^{(t)}\right)-H\left(\theta\mid\theta^{(t)}\right)\\=&E\left\{\log f_{Z|X}\left(z\mid x,\theta^{(t)}\right)-\log f_{Z|X}(z\mid x,\theta)\right\}\\=&\int-\log\left\{\frac{f_{Z|X}(z\mid x,\theta)}{f_{Z|X}\left(z\mid x,\theta^{(t)}\right)}\right\}f_{Z|X}\left(z\mid x,\theta^{(t)}\right)dz\\\geq&-\log\int f_{Z|X}(z\mid x,\theta)dz\\=&0,\end{aligned}$$

其中不等式的结果来自Jensen's 不等式；

这样的话，如果我们选择$\theta^{(t+1)}$使得 $Q(\theta \mid \theta^{(t)})$达到最大，那么我们必然有：

$$H\left(\theta^{(t)} \mid \theta^{(t)}\right) \geq H\left(\theta^{(t+1)} \mid \theta^{(t)}\right)$$

因而我们得到：

$$\log f_X\left(x \mid \theta^{(t+1)}\right) - \log f_X\left(x \mid \theta^{(t)}\right) \geq 0$$

- **分布式EM算法**

由于EM算法依赖于似然函数，而似然函数又表达成和的形式，因此在很多情况下我们可以考虑分布式计算。

假设观察值数据集被分割成s块$\{D_1, D_2, \cdots, D_s\}$

如果我们考虑的分布属于指数分布族的话，那么EM算法中的M步就会是求解若干包含该分布下**期望充分统计量**的等式。

因此我们可以考虑在E步计算其期望充分统计量，即

$$t = E(T \mid X, \theta)$$

其中T是充分统计量。

给定每个数据块D_i来计算期望充分统计量$E(T_i \mid D_i, \theta)$与给定其他数据块所计算的期望统计量独立，且这部分期望只是这部分数据贡献的期望。

实际上，对于服从指数分布族的数据，我们有：

$$t = \sum_{i=1}^{s} E\left(T_i \mid D_i, \theta\right)$$

于是我们就可以在s个计算节点上算出各自的期望统计量，然后通过节点间的通信得到最后的期望统计量。

在得到汇总后的期望统计量之后，我们就可以考虑M步，更新参数估计。

第七次课

高斯混合模型

混合正态模型，我们一般用如下的记号表示：

$$X_i \sim \sum_{k=1}^{K} \pi_k N(\mu_k, \sigma_k^2)$$

其中$\sum_{k=1}^{K} \pi_k = 1$.这个记号表示的意思是：X_i有π_k的概率来自于分布$N(\mu_k, \sigma_k^2)$

一般在学习过程中，同学会写出以下两种变体：

1. 令$Y_k \sim N(\mu_k, \sigma_k^2)$，得到一个新的随机变量：$Y = \sum_{k=1}^{K} \pi_k Y_k$.
2. 令$f_k(x)$为$N(\mu_k, \sigma_k^2)$的概率密度函数，得到一个新的随机变量Z使得其概率密度函数$f(x)$满足$f(x) = \sum_{k=1}^{K} \pi_k f_k(x)$.

现需要回答下面两个问题：

1. 上面的两种描述有区别吗？如果有，请举出反例；如果没有，请给出证明
2. 上面的两种描述中，哪一种符合混合正态模型的定义？

解答：

设X有π_k的概率来自于分布$N(\mu_k, \sigma_k^2)$

$$\mathbb{P}\{X < v\} = \mathbb{E}[\mathbb{1}_{\{X<v\}}] = \mathbb{E}[\mathbb{E}[\mathbb{1}_{\{X<v\}} \mid X \sim N(\mu_k, \sigma_k^2)]] = \sum_{k=1}^{K} \pi_k f_k(x)$$

因此，第二种描述符合混合正态模型的定义。

- 考虑$N(\mu_k, \Sigma_k)$的高斯混合模型

参数(π, μ, Σ)的充分统计量为：

$$
\begin{aligned}
N_k &= \sum_{i=1}^{n} \gamma_{ik} \\
\mu_k &= \frac{1}{N_k} \sum_{i=1}^{n} \gamma_{ik} x_i \\
\Sigma_k &= \frac{1}{N_k} \sum_{i=1}^{n} \gamma_{ik} x_i x_i^T, k = 1, \ldots, K
\end{aligned}
$$

其中$\gamma_{ik} = \{x_i\text{来自第}k\text{个正态分布} \mid \text{数据}\}$.

基本的EM算法可以描述为：

***E-step*：**

$$
\gamma_{ik}^{(t+1)} = P\left(x_i \text{ 来自于第 } k \text{ 个正态分布 } \mid \text{data}\right) = \frac{\pi_k^{(t+1)} N\left(x_i \mid \mu_k^{(t)}, \boldsymbol{\Sigma}_k^{(t)}\right)}{\sum_{j=1}^{K} \pi_j^{(t+1)} N\left(x_i \mid \mu_j^t, \Sigma_j^{(t)}\right)}
$$

其中

$$
\pi_k^{(t+1)} = \frac{N_k^{(t)}}{\sum_{j=1}^{K} N_j^{(t)}}
$$

***M-step*：**

$$
\begin{aligned}
N_k^{(t+1)} &= \sum_{i=1}^{n} \gamma_{ik}^{(t+1)} \\
\mu_k^{(t+1)} &= \frac{1}{N_k^{(t+1)}} \sum_{i=1}^{n} \gamma_{ik}^{(t+1)} x_i \\
\Sigma_k^{(t+1)} &= \frac{1}{N_k^{(t+1)}} \sum_{i=1}^{n} \gamma_{ik}^{(t+1)} \left(x_i - \mu_k^{(t+1)}\right)\left(x_i - \mu_k^{(t+1)}\right)^T
\end{aligned}
$$

$$
Cov(X) = E(XX') - (EX)(EX')
$$

- 从混合正态分布$0.2N(5,4) + 0.8N(0,1)$中产生10000个随机数，然后利用分布式EM算法（使用5个计算节点）来估计相关参数：
  - 首先引入生成随机数的函数

```
scala> import java.util.concurrent.ThreadLocalRandom
import java.util.concurrent.ThreadLocalRandom

scala> val random = ThreadLocalRandom.current
random: java.util.concurrent.ThreadLocalRandom =
java.util.concurrent.ThreadLocalRandom@550a78ed

scala> random.nextDouble
res0: Double = 0.5909702094956042

scala> random.nextGaussian
res1: Double = 1.1486026409814427
```

  - 设置基本参数

```
scala> val N = 10000                //随机数的数目
N: Int = 10000

scala> val MU1 = 5.0                //两个正态分布的均值与方差
MU1: Double = 5.0

scala> val Sig1 = 2.0
Sig1: Double = 2.0

scala> val MU2 = 0.0
```

```
MU2: Double = 0.0

scala> val Sig2 = 1.0
Sig2: Double = 1.0

scala> val P = 0.2
P: Double = 0.2

scala> val NumOfSlaves = 5          //节点数
NumOfSlaves: Int = 5
```

- 用 `Array.ofDim[]()` 函数，生成用于存放10000个随机数的Array

```
scala> val data = Array.ofDim[Double](N)
data: Array[Double] = Array(0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,...
```

探究 `Array.ofDim[]()` 函数

```
scala> val d2 = Array.ofDim[Double](10,2)
d2: Array[Array[Double]] = Array(Array(0.0, 0.0), Array(0.0, 0.0), Array(0.0, 0.0), Array(0.0,
0.0), Array(0.0, 0.0), Array(0.0, 0.0), Array(0.0, 0.0), Array(0.0, 0.0), Array(0.0, 0.0),
Array(0.0, 0.0))
```

说明：`Array.ofDim[Double](10,2)` 生成10个2元Double型数组，并且这十个数组整体组成一个新的数组；

```
scala> val d3 = Array.ofDim[Double](4,3,2)
d3: Array[Array[Array[Double]]] = Array(Array(Array(0.0, 0.0), Array(0.0, 0.0), Array(0.0,
0.0)), Array(Array(0.0, 0.0), Array(0.0, 0.0), Array(0.0, 0.0)), Array(Array(0.0, 0.0),
Array(0.0, 0.0), Array(0.0, 0.0)), Array(Array(0.0, 0.0), Array(0.0, 0.0), Array(0.0, 0.0)))
```

说明：`Array.ofDim[Double](4,3,2)` 生成4个Array，其中每个Array由3个2元Double型Array组成，而这四个Array整体构成一个新的Array。

总结： 对于 `Array.ofDim[Double](a1,...,an)` 可以"倒着理解"，即最小的Array包含a_n个0.0，而a_{n-1}个这样的Array组成一个新的Array，以此类推，且最后的结果为一个大Array。

- 生成10000个来自混合正态分布$0.2N(5,4)+0.8N(0,1)$的随机数，并存入上述Array

```
scala> for(i <- 0 until N)
     | {
     |     val db = random.nextDouble
     |     if(db < P)
     |     {
     |         data(i) = MU1 + Sig1*random.nextGaussian
     |     }else
     |     {
     |         data(i) = MU2 + Sig2*random.nextGaussian
     |     }
     | }

scala> data
```

```
res5: Array[Double] = Array(-2.7132844808173724, 0.30510291482221424, 0.069554789755275536,
1.4442903127781825, 3.538969083047303, -0.8314246175847418, 4.782946753933859,
0.16360240110546329, 4.429613610706143, 0.6528553410434296, -0.08907029668751042,
-1.3290115507472073, 0.5799824535505411, 4.52421397263275, -0.7029061338656183,
4.198876886131852, -0.6174082213389915, 0.5203361664936965, -0.12029571017418926,
0.8605344964754867, 0.1342587699405833, 0.094355656543043, 7.208743591410735,
-2.239603374202545, 2.711734923994691, 1.019499901891506, -0.9614881697004042,
8.19277320800127, -0.0867942127635768, -0.4284606374978425, -2.8909446668217242,
0.3792389670469632, 4.250161594148422, -1.264061270753557, -1.1723445616532817,
1.5565131556540177, -0.2139491474412162, 1.192102474852409, 0.733...
```

原理：`random.nextGaussian`生成服从$N(0,1)$的随机数，因而`MU + Sig*random.nextGaussian`可以产生服从$N(\mu,\sigma^2)$的随机数。

- 由于需要设置五个计算节点来估计相关参数，故先将随机数组进行分布式存储：

```
scala> var ParData = sc.parallelize(data,NumOfSlaves)
ParData: org.apache.spark.rdd.RDD[Double] = ParallelCollectionRDD[1] at parallelize at
<console>:38
```

- 为了防止**直接产生的随机数无法序列化**，因此先将生成的RDD转换为文本格式，再转换为Double型：

*Step1：*先转换为文本格式，并保留四位小数

```
scala> val ParDataStr = ParData.map("%.4f" format _)
ParDataStr: org.apache.spark.rdd.RDD[String] = MapPartitionsRDD[2] at map at <console>:36

scala> ParDataStr.collect
res7: Array[String] = Array(-2.7133, 0.3051, 0.0696, 1.4443, 3.5390, -0.8314, 4.7829, 0.1636,
4.4296, 0.6529, -0.0891, -1.3290, 0.5800, 4.5242, -0.7029, 4.1989, -0.6174, 0.5203, -0.1203,
0.8605, 0.1343, 0.0944, 7.2087, -2.2396, 2.7117, 1.0195, -0.9615, 8.1928, -0.0868, -0.4285,
-2.8909, 0.3792, 4.2502, -1.2641, -1.1723, 1.5565, -0.2139, 1.1921, 0.7332, -0.0057, 8.5789,
-0.0470, -1.0285, 1.2773, -1.3827, -0.0947, 3.8473, -1.7656, 0.1267, 1.0834, 6.5781, -1.2062,
-1.0344, -0.0683, -0.8044, -1.0885, -2.2203, 3.0056, 2.4436, 0.2740, 0.0077, 0.7107, 0.7032,
0.9914, -0.3173, -0.2264, -1.0308, 0.2603, 8.7240, 4.6375, 0.9243, -0.2358, -0.1497, 0.3581,
0.1943, -0.3730, -0.9031, -1.7075, -0.1335, 0.6900, 0.3935, -0.8810, -0.8849, 1.0367, 7.0638,
-0.0933, 2.0308, 0.3771, 4.3590, 1.5392, 6.6383, -1...
```

注意1：`map("%.4f" format _)`的`format`前后均有**空格**！

注意2：数值经过格式化输出后，得到的是一个**字符串**！

*Step2：*再转换为Double型：

```
scala> ParData = ParDataStr.map(_.toDouble)
ParData: org.apache.spark.rdd.RDD[Double] = MapPartitionsRDD[3] at map at <console>:37

scala> ParData.collect()
res8: Array[Double] = Array(-2.7133, 0.3051, 0.0696, 1.4443, 3.539, -0.8314, 4.7829, 0.1636,
4.4296, 0.6529, -0.0891, -1.329, 0.58, 4.5242, -0.7029, 4.1989, -0.6174, 0.5203, -0.1203,
0.8605, 0.1343, 0.0944, 7.2087, -2.2396, 2.7117, 1.0195, -0.9615, 8.1928, -0.0868, -0.4285,
-2.8909, 0.3792, 4.2502, -1.2641, -1.1723, 1.5565, -0.2139, 1.1921, 0.7332, -0.0057, 8.5789,
-0.047, -1.0285, 1.2773, -1.3827, -0.0947, 3.8473, -1.7656, 0.1267, 1.0834, 6.5781, -1.2062,
-1.0344, -0.0683, -0.8044, -1.0885, -2.2203, 3.0056, 2.4436, 0.274, 0.0077, 0.7107, 0.7032,
0.9914, -0.3173, -0.2264, -1.0308, 0.2603, 8.724, 4.6375, 0.9243, -0.2358, -0.1497, 0.3581,
0.1943, -0.373, -0.9031, -1.7075, -0.1335, 0.69, 0.3935, -0.881, -0.8849, 1.0367, 7.0638,
-0.0933, 2.0308, 0.3771, 4.359, 1.5392, 6.6383, -1.7977, 2.005...
```

- 设置估计的初始值（即迭代初始值）：

```
scala> val InitialP = 0.5                  //初始p值
InitialP: Double = 0.5

scala> var Nk = N.toDouble*InitialP        //初始情况下，第一个正态分布所占数目
Nk: Double = 5000.0
```

```
scala> var EstMu1=ParData.reduce((x,y) => x+y)/N.toDouble
EstMu1: Double = 1.0327776300000007         //对MU1的初始估计：即所有数值的平均（任意给即可）

scala> var EstSig1 = math.sqrt(ParData.map(x => x*x).reduce((x,y) => x+y)/N.toDouble -
EstMu1*EstMu1)
EstSig1: Double = 2.394075206324687         //对Sig1的初始估计，即假设全部来自第一个正态分布
                                            //利用方差=平方的均值-均值的平方

scala> var EstMu2 = EstMu1 - 1.0
EstMu2: Double = 0.03277763000000067

scala> var EstSig2 = EstSig1
EstSig2: Double = 2.394075206324687

scala> var Diff = 0.0                       //Diff代表两次迭代的差值
Diff: Double = 0.0

scala> var OldEstMu1 = 0.0                  //Old用于存放上一次各统计量的估计值
OldEstMu1: Double = 0.0

scala> var OldEstMu2 = 0.0
OldEstMu2: Double = 0.0

scala> var OldEstSig1 = 0.0
OldEstSig1: Double = 0.0

scala> var OldEstSig2 = 0.0
OldEstSig2: Double = 0.0

scala> var ii = 0                           //ii为迭代次数
ii: Int = 0

scala> var eps = 0.00001                    //eps为迭代停止条件值
eps: Double = 1.0E-5
```

- 进行EM算法迭代

  原理：

$$\gamma_{i1}^{(t+1)} = P\left(x_i \text{ 来自于第 } 1 \text{ 个正态分布 } \mid \text{data}\right) = \frac{p^{(t+1)} N\left(x_i \mid \mu_1^{(t)}, \boldsymbol{\Sigma}_1^{(t)}\right)}{p^{(t+1)} N\left(x_i \mid \mu_1^{(t)}, \boldsymbol{\Sigma}_1^{(t)}\right) + (1 - p^{(t+1)}) N\left(x_i \mid \mu_2^{(t)}, \boldsymbol{\Sigma}_2^{(t)}\right)}$$

  因而：

$$\gamma_{i1}^{(t+1)} = P\left(x_i \text{ 来自于第 } 1 \text{ 个正态分布 } \mid \text{data}\right) = \frac{p^{(t+1)}}{p^{(t+1)} + (1 - p^{(t+1)}) f\left(x_i \mid \mu_2^{(t)}, \sigma_2^{(t)}\right) / f\left(x_i \mid \mu_1^{(t)}, \sigma_1^{(t)}\right)}$$

  再化简可得：

$$\gamma_{i1}^{(t+1)} = P\left(x_i \text{ 来自于第 } 1 \text{ 个正态分布 } \mid \text{data}\right) = \frac{N_1^{(t+1)}}{N_1^{(t+1)} + (N - N_1^{(t+1)}) \frac{\sigma_1}{\sigma_2} \exp\{-\frac{1}{2\sigma_2^2}(x-\mu_2)^2 + \frac{1}{2\sigma_1^2}(x-\mu_2)^2\}}$$

  在本代码中，令$x_1 = -\frac{1}{2\sigma_2^2}(x-\mu_2)^2, x2 = -\frac{1}{2\sigma_1^2}(x-\mu_1)^2$

```
scala> do
     | {
     |       ii = ii + 1
     |       OldEstMu1 = EstMu1
     |       OldEstMu2 = EstMu2
     |       OldEstSig1 = EstSig1
     |       OldEstSig2 = EstSig2
     |       var SufficientStatistics = ParData.map(line => {
     |           val x1 = -math.pow((line - EstMu2)/EstSig2,2)/2.0
     |           val x2 = -math.pow((line - EstMu1)/EstSig1,2)/2.0
```

```
     |            val gamma = Nk * EstSig2/(Nk*EstSig2 + (N- Nk)*EstSig1*math.exp(x1-x2))
     |            (line, gamma, line*gamma, line*line*gamma, 1-gamma, line*(1-gamma), line*line*
(1-gamma))
     |        })
     |        val Results = SufficientStatistics.reduce((x,y)=> (x._1+y._1, x._2+y._2, x._3 +
y._3, x._4 + y._4,x._5 + y._5, x._6 + y._6, x._7 + y._7))
     |        Nk = Results._2
     |        EstMu1 = Results._3/Nk
     |        EstSig1 = math.sqrt(Results._4/Nk - EstMu1*EstMu1)
     |        EstMu2 = Results._6/Results._5
     |        EstSig2 = math.sqrt(Results._7/Results._5 - EstMu2*EstMu2)
     |        Diff = math.abs(EstMu1 - OldEstMu1) + math.abs(EstMu2 - OldEstMu2)
     |        Diff = Diff + math.abs(EstSig1 - OldEstSig1) + math.abs(EstSig2 - OldEstSig2)
     | }while(Diff>eps)
```

- 查看迭代结果

```
scala> ii
res11: Int = 68

scala> EstMu1
res12: Double = 5.132389067049064

scala> EstSig1
res13: Double = 1.8961105525090098

scala> EstMu2
res14: Double = 0.0076927833072746345

scala> EstSig2
res15: Double = 1.0062468095557497
```

思考：为何估计值与设定值有一定的差距？

解答：因为在生成混合正态分布时，有一定的随机性。

二分法求解函数的零点

首先不妨定义函数为$F(x)-U$：

```
scala> val myNormal = new Gaussian(0.0,1.0)
myNormal: breeze.stats.distributions.Gaussian = Gaussian(0.0, 1.0)

scala> def f(x: Double, U: Double):Double={myNormal.cdf(x)-U}
f: (x: Double, U: Double)Double

scala> val U = 0.234
U: Double = 0.234
```

构造二分法：

```
scala> def BiSec(a:Double,b:Double,U:Double,eps:Double):Double={
     | var a0 = a
     | var b0 = b
     | var mid = (a0 + b0)/2
     | if(b0 - a0 < eps)
     | {
     |     mid
     | }else
     | {
     |     while(b0 - a0 >= eps)
     |     {
     |         if(f(a0,U)*f(mid,U)<0)
     |         {
     |             b0 = mid
```

```
     |          }else
     |          {
     |              a0 = mid
     |          }
     |          mid = (a0 + b0)/2.0
     |      }
     |      mid
     | }
     | }
BiSec: (a: Double, b: Double, U: Double, eps: Double)Double
```

求解上述函数的零点：

```
scala> BiSec(-100.0,100.0,U,0.00001)
res16: Double = -0.7257372140884399

scala> myNormal.cdf(-0.7257372140884399)
res17: Double = 0.23399994175412514
```

HW3

1. **使用逆累积分布函数法产生密度函数为$f(x) = \frac{3x^2}{2}, -1 \leq x \leq 1$的随机数。试用Scala编写程序产生10000个上述分布的随机数。**

解答：

$$F(x) = \frac{1}{2}x^3 + \frac{1}{2}$$
$$F^{-1}(x) = (2x-1)^{\frac{1}{3}}$$

根据逆累积分布法：

```
scala> import java.util.concurrent.ThreadLocalRandom
import java.util.concurrent.ThreadLocalRandom

scala> val r = ThreadLocalRandom.current
r: java.util.concurrent.ThreadLocalRandom = java.util.concurrent.ThreadLocalRandom@1b36c5aa

scala> var u = (1 to 10000).map(s => r.nextDouble)
u: scala.collection.immutable.IndexedSeq[Double] = Vector(0.7747174584284546, 0.26292269644471056,
0.07091937150652206, 0.966829181563792, 0.32212256354442115, 0.7477773555700113,
0.20695994896581194, 0.4788257633701669, 0.870409177765118, 0.13958878451276613,
0.05081129725383915, 0.19936338735779802, 0.14001946881144345, 0.7021833413410115,
0.7947794514858071, 0.9799711374282833, 0.6353623095860543, 0.7016850227249449, 0.9937864015103489,
0.719758820335911, 0.6544151727337932, 0.5171914232066632, 0.6956185638402718, 0.7823498605793139,
0.572527434417522, 0.3690437660348209, 0.979198061821772, 0.6662879019551995, 0.5817927521284562,
0.7596845663758951, 0.48313694594243606, 0.707224834101742, 0.9416649976903332, 0.9988996765134891,
0.8342978992992494, 0.31159601782759017, 0.6205511613655...

scala> val rand = u.map(s => {
     |    if(s > 0.5) pow(2*s-1,1.0/3)
     |    else -pow(1-2*s,1.0/3)})
rand: scala.collection.immutable.IndexedSeq[Double] = Vector(0.8190405776977047,
-0.7797822135286503, -0.9502903109005704, 0.977378235673682, -0.7085714003280594,
0.7913413686146487, -0.8368590665573759, -0.3485613689657877, 0.9048374729374069,
-0.8966220829812561, -0.9649088141682809, -0.8440288436813819, -0.896264792052758,
0.7394777680620233, 0.8385115851827156, 0.9864650567739578, 0.6469080922588814, 0.7388697412206489,
0.9958403220780099, 0.7603124527080184, 0.6759376784128099, 0.3251725991715108, 0.7313860746750686,
0.826556459887959, 0.525425045156067, -0.6398115232193519, 0.9859351486062984, 0.6928356343955375,
0.5469088347600433, 0.8038198221446292, -0.32308892845808407, 0.745573732307823,
0.9594911928256422, 0.9992659122560398, 0.8744222792568249, -0.7222818324739444, 0.622396...
```

2. **假设随机向量$(X,Y)^\top$来自于二元正态分布：**

$$N\left(\begin{pmatrix}10.0\\-10.0\end{pmatrix},\begin{pmatrix}1.0 & 0.5\\0.5 & 1.0\end{pmatrix}\right)$$

随机产生10^5个这样的向量，请使用MapReduce的方法计算下列期望所对应的样本值：

(a) $\mathbb{E}\left(X^2Y^2\right)$

(b) $\mathbb{E}\begin{pmatrix}X^2 & XY\\XY & Y^2\end{pmatrix}$

(c) $\mathbb{E}\begin{pmatrix}\sin\left(X^2\right) & \cos(XY)\\\cos(XY) & \sin\left(Y^2\right)\end{pmatrix}$

```
scala> val multi = new MultivariateGaussian(DenseVector(10.0,-10.0),DenseMatrix((1.0,0.5),
(0.5,1.0)))
multi: breeze.stats.distributions.MultivariateGaussian =
MultivariateGaussian(DenseVector(10.0, -10.0),1.0  0.5
0.5  1.0  )

scala> val sample = multi.sample(100000)
22/11/04 16:39:29 WARN BLAS: Failed to load implementation from:
com.github.fommil.netlib.NativeSystemBLAS
22/11/04 16:39:29 WARN BLAS: Failed to load implementation from:
com.github.fommil.netlib.NativeRefBLAS
sample: IndexedSeq[breeze.linalg.DenseVector[Double]] = Vector(DenseVector(10.03421475800303,
-10.946653399934359), DenseVector(9.506592279014118, -8.956606581914116),
DenseVector(10.178296977856402, -9.007716222371986), DenseVector(10.373124393631828,
-9.775895474153053), DenseVector(9.963082876780032, -10.3316599470694),
DenseVector(10.433792646803214, -9.65562815355985), DenseVector(9.22679218390505,
-10.412973707441113), DenseVector(11.24962197682054, -9.280148983530937),
DenseVector(10.034192351208471, -9.011202727224035), DenseVector(10.288915837072881,
-9.233151818673763), DenseVector(10.523300216615013, -9.467738879544871),
DenseVector(8.886682330866634, -11.188656116830238), DenseVector(10.302129486942706,
-11.056854530457287), DenseVector(12.294866816642028, -8.426247795743885...

scala> val a = sample.map(x => x(0)*x(0)*x(1)*x(1)).reduce((x,y) => x+y)/100000.0
a: Double = 10004.876579661783

scala> val b = sample.map(x => DenseMatrix((x(0)*x(0),x(0)*x(1)),
(x(0)*x(1),x(1)*x(1)))).reduce((x,y) => x+y)/100000.0
b: breeze.linalg.DenseMatrix[Double] =
101.04416587516124  -99.5138549187069
-99.5138549187069   100.99036785217537

scala> val c = sample.map(x => DenseMatrix((sin(x(0)*x(0)),cos(x(0)*x(1))),
(cos(x(0)*x(1)),sin(x(1)*x(1))))).reduce((x,y) =>
 x+y)/100000.0
c: breeze.linalg.DenseMatrix[Double] =
7.29444243579557E-5    6.630187660699625E-4
6.630187660699625E-4  -2.507662882824005E-4
```

3. **首先从混合二元正态分布**

$$0.4\times N\left(\begin{pmatrix}1.0\\2.0\end{pmatrix},\begin{pmatrix}2.0 & 1.0\\1.0 & 2.0\end{pmatrix}\right)+0.6\times N\left(\begin{pmatrix}5.0\\6.0\end{pmatrix},\begin{pmatrix}4.0 & 1.0\\1.0 & 4.0\end{pmatrix}\right)$$

中产生10000个随机向量。

```
scala> val multi1 = new MultivariateGaussian(DenseVector(1.0,2.0),DenseMatrix((2.0,1.0),(1.0,2.0)))
multi1: breeze.stats.distributions.MultivariateGaussian =
MultivariateGaussian(DenseVector(1.0, 2.0),2.0  1.0
1.0  2.0  )
```

```
scala> val multi2 = new MultivariateGaussian(DenseVector(5.0,6.0),DenseMatrix((4.0,1.0),(1.0,4.0)))
multi2: breeze.stats.distributions.MultivariateGaussian =
MultivariateGaussian(DenseVector(5.0, 6.0),4.0  1.0
1.0  4.0  )

scala> var data = Array.ofDim[Double](10000).map(x => DenseVector(0.0,0.0))
data: breeze.linalg.DenseVector[breeze.linalg.DenseVector[Double]] = DenseVector(DenseVector(0.0,
0.0), DenseVector(0.0, 0.0), DenseVector(0.0, 0.0), DenseVector(0.0, 0.0), DenseVector(0.0, 0.0),
DenseVector(0.0, 0.0), DenseVector(0.0, 0.0), DenseVector(0.0, 0.0), DenseVector(0.0, 0.0),
DenseVector(0.0, 0.0), DenseVector(0.0, 0.0), DenseVector(0.0, 0.0), DenseVector(0.0, 0.0),
DenseVector(0.0, 0.0), DenseVector(0.0, 0.0), DenseVector(0.0, 0.0), DenseVector(0.0, 0.0),
DenseVector(0.0, 0.0), DenseVector(0.0, 0.0), DenseVector(0.0, 0.0), DenseVector(0.0, 0.0),
DenseVector(0.0, 0.0), DenseVector(0.0, 0.0), DenseVector(0.0, 0.0), DenseVector(0.0, 0.0),
DenseVector(0.0, 0.0), DenseVector(0.0, 0.0), DenseVector(0.0, 0.0), DenseVector(0.0, 0.0),
DenseVector(0.0, 0.0), DenseVector(0.0, 0.0), Den...

scala> val p = 0.4
p: Double = 0.4

scala> for(i <- 0 until 10000)
     | {
     |     val db = r.nextDouble
     |     if(db < p)
     |     {
     |         data(i) = multi1.sample(1)(0)
     |     }
     |     else
     |     {
     |         data(i) = multi2.sample(1)(0)
     |     }
     | }

scala> data
res1: Array[breeze.linalg.DenseVector[Double]] = Array(DenseVector(4.987968035330491,
4.1990004462536135), DenseVector(5.542443960091046, 5.460091579450139),
DenseVector(7.558530418358064, 11.282553464955173), DenseVector(0.2807549309724787,
1.8756729724239556), DenseVector(4.323249993269325, 7.829753342276425),
DenseVector(4.919225992159352, 4.64516902216495), DenseVector(1.6110749984513428,
5.3704002430530675), DenseVector(4.259750344608874, 3.878041708263233),
DenseVector(3.991835777135728, 6.489085098532391), DenseVector(5.419220701309479,
5.702661720626656), DenseVector(0.4688701298925827, 1.9915790525748605),
DenseVector(0.4266643463809965, 1.4240016567808809), DenseVector(0.06635572302441894,
2.885293448635386), DenseVector(3.6994833695082603, 5.317586401889226), DenseVector(2.92...
```

4. **使用上题中产生的数据，编写分布式EM算法（使用4个计算节点）来估计相关参数。**

```
val N = 10000
var ParData = sc.parallelize(data,4)
var ParDataStr = ParData.map(_.map("%.4f" format _))
ParDataStr.collect()
ParData = ParDataStr.map(_.map(_.toDouble))
ParData.collect()
val InitialP =0.5
var Nk = N.toDouble * InitialP
var EstMu1 = ParData.reduce((x,y) => x+y)/N.toDouble
var EstMu2 = EstMu1 - 1.0
var EstSig1 = DenseMatrix((1.8,1.2),(1.2,1.8))
var EstSig2 = DenseMatrix((3.7,0.9),(0.9,3.7))
var Diff = 0.0
var OldEstMu1 = DenseVector(0.0,0.0)
var OldEstMu2 = DenseVector(0.0,0.0)
var OldEstSig1 = DenseMatrix((0.0,0.0),(0.0,0.0))
var OldEstSig2 = DenseMatrix((0.0,0.0),(0.0,0.0))
var ii = 0
```

```
val eps = 0.00001
do
{
    ii = ii + 1
    OldEstMu1 = EstMu1
    OldEstMu2 = EstMu2
    OldEstSig1 = EstSig1
    OldEstSig2 = EstSig2
    var SufficientStatistics = ParData.map(line =>{
    var x1 = 0.5*(line-EstMu1).t*inv(EstSig1)*(line-EstMu1)
    var x2 = 0.5*(line-EstMu2).t*inv(EstSig2)*(line-EstMu2)
    var gamma = Nk/(Nk+(N-Nk)*pow(det(EstSig1),0.5)/pow(det(EstSig2),0.5)*exp(x1(0)-x2(0)))
    (line,gamma,line*gamma,line*line.t*gamma,1-gamma,line*(1-gamma),line*line.t*(1-gamma))
    })
    val Results = SufficientStatistics.reduce((x,y) => (x._1+y._1, x._2+y._2, x._3 + y._3, x._4 +
y._4,x._5 + y._5, x._6 + y._6, x._7 + y._7))
    Nk = Results._2
    EstMu1 = Results._3/Nk
    EstSig1 = Results._4/Nk - EstMu1*EstMu1.t
    EstMu2 = Results._6/Results._5
    EstSig2 = Results._7/Results._5 - EstMu2*EstMu2.t
    Diff = norm(EstMu1 - OldEstMu1) + norm(EstMu2 - OldEstMu2)
    Diff = Diff + norm((EstSig1-OldEstSig1).toDenseVector) + norm((EstSig2-
OldEstSig2).toDenseVector)
}while(Diff > eps)
```

其中，计算SufficientStatistics的步骤可以利用PDF绕过多元统计分析理论，从而简化代码：

```
val SufficientStatistics = ParData.map(line => {
    val n1 = new MultivariateGaussian(OldEstMu1,OldEstSig1)
    val n2 = new MultivariateGaussian(OldEstMu2,OldEstSig2)
    val gamma = (Nk*n1.pdf(line)/(Nk*n1.pdf(line)+(N - Nk)*n2.pdf(line)))
    (line,gamma,line*gamma,line*line.t*gamma,1-gamma,line*(1-gamma),line*line.t*(1-gamma))
  })
```

结果：

```
scala> ii
res20: Int = 102

scala> EstMu1
res21: breeze.linalg.DenseVector[Double] = DenseVector(1.0252855520502246, 1.973327806380762)

scala> EstSig1
res22: breeze.linalg.DenseMatrix[Double] =
2.043149719223137   0.9895476231115192
0.9895476231115192  1.8743835114032037

scala> EstMu2
res23: breeze.linalg.DenseVector[Double] = DenseVector(5.008478596310377, 6.011061036791729)

scala> EstSig2
res24: breeze.linalg.DenseMatrix[Double] =
3.962392604820238   0.9463950727114749
0.9463950727114749  3.999194851413577

scala> Diff
res25: Double = 9.072809152038718E-6
```

第八次课—优化方法

矩阵运算的分布式实现

- **案例：分位数回归分布式参数估计**
- 考虑线性分位数回归模型

$$Q_\tau\left(y_i \mid x_i\right)=x_i^T \beta_0$$

  其中x和y分别是解释变量和响应变量，$Q_\tau\left(y_i \mid x_i\right)$表示响应变量在给定解释变量下的条件$\tau$分位数，而$\beta_0$则是对应的真实系数.
- 对于参数β_0的估计，我们可以通过最小化如下目标函数来得到：

$$L(\beta)=\frac{1}{n} \sum_{i=1}^n \rho_\tau\left(y_i-x_i^T \beta\right)$$

  其中$\rho_\tau(u)=u\{\tau-I(u<0)\}$.
- 应用ADMM方法，最小化$L(\beta)$就等同于解决如下问题：

$$\begin{aligned} & \operatorname{minimize} \sum_{i=1}^n \rho_\tau\left(z_i\right) \\ & \text {s.t. } \quad z=y-X \beta\end{aligned}$$

  其中$z=\left(z_1, \ldots, z_n\right)^T, y=\left(y_1, \ldots, y_n\right)^T, X=\left(x_1, \ldots, x_n\right)^T$
- 这就相当于求解

$$L_{r_1}(\alpha, \beta, z)=\sum_{i=1}^n \rho_\tau\left(z_i\right)+\frac{r_1}{2}\|(y-X \beta)-z\|^2+\langle y-X \beta-z, \alpha\rangle$$

  其中$<\cdot,\cdot>$表示内积.
- 具体的更新步骤如下：
  - (1)更新β

$$\beta^{k+1}=\left(X^T X\right)^{-1}\left\{\frac{1}{r_1} X^T \alpha+X^T\left(y-z^k\right)\right\}$$

  - (2)更新z

    令$a_i^k=y_i-x_i^T \beta^{k+1}+\frac{\alpha_i^k}{r_1}$可得：

$$z_i^{k+1}=\left\{\begin{array}{cc}a_i^k-\frac{\tau}{r_1} & \frac{\tau}{r_1}<a_i^k \\ 0 & \frac{\tau-1}{r_1} \leq a_i^k \leq \frac{\tau}{r_1} \\ a_i^k-\frac{\tau-1}{r_1} & a_i^k<\frac{\tau-1}{r_1}\end{array}\right.$$

  - (3)更新α：

$$\alpha^{k+1}=\alpha^k+r_1\left(y-X \beta^{k+1}-z^{k+1}\right)$$

- 在这个案例中，我们尝试解决样本量n很大的情况。
- 我们从下面的线性回归模型中产生随机实验数据，用于分位数回归的估计：

$$y_i=x_{1, i}+x_{2, i}+x_{3, i}+\cdots+x_{100, i}+x_{1, i} \epsilon_i$$

- Array的相关语法复习：
  - `A.union(B)`：连接Array向量A和Array向量B

```
scala> val a = Array(1.0,2.0,3.0)
a: Array[Double] = Array(1.0, 2.0, 3.0)

scala> val b = Array(4.0,5.0)
b: Array[Double] = Array(4.0, 5.0)

scala> a.union(b)
res1: Array[Double] = Array(1.0, 2.0, 3.0, 4.0, 5.0)
```

- `new Array[Double](p)`：生成长度为p的Array

```
scala> new Array[Double](10)
res2: Array[Double] = Array(0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0)
```

注意： p只能为一维数值

- `Array.fill(N)(a)`：生成长度为N的Array, 其中的元素全为a

```
scala> Array.fill(10)(2.0)
res3: Array[Double] = Array(2.0, 2.0, 2.0, 2.0, 2.0, 2.0, 2.0, 2.0, 2.0, 2.0)
```

- **Spark线性运算类：**
  - 首先导入Spark线性运算类的包：`import org.apache.spark.mllib.linalg._`

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

导入`org.apache.spark.mllib.linalg`中矩阵运算的函数；由于部分函数与Scala自带函数重名，因此进行重命名

```
scala> import org.apache.spark.mllib.linalg.{Vectors, Matrix => sparkMatrix, DenseMatrix =>
sparkDenseMatrix}
import org.apache.spark.mllib.linalg.{Vectors, Matrix=>sparkMatrix,
DenseMatrix=>sparkDenseMatrix}
```

注意： 本质上不用单独导入`Vectors`函数，因为在`import org.apache.spark.mllib.linalg._`时，已经导入了该函数，即在未运行上述代码时，仍有：

```
scala> Vectors.dense(Array(1.0,2.0,3.0))
res8: org.apache.spark.mllib.linalg.Vector = [1.0,2.0,3.0]
```

因此，此命令的主要目的在于函数的重命名。

再来考虑上述向量函数与矩阵函数的用法：

- 通过`new sparkDenseMatrix(m,n,Array)`创建一个m行n列的矩阵，再通过一个Array向量传入所有的元素值，并**按列的方式进行填充。**

  例如：创建一个3行2列的矩阵

```
scala> new sparkDenseMatrix(3,2,Array(1.0,2.0,3.0,4.0,5.0,6.0))
res9: org.apache.spark.mllib.linalg.DenseMatrix =
1.0  4.0
2.0  5.0
3.0  6.0
```

回顾：DenseMatrix是按行进行填充的！

```
scala> DenseMatrix((1.0,2.0,3.0),(4.0,5.0,6.0))
res11: breeze.linalg.DenseMatrix[Double] =
1.0  2.0  3.0
4.0  5.0  6.0
```

- 利用`Vectors.dense(A)`将Array A转换成Spark DenseVector

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

注意：此处Array的值必须是Double类型的，不能是Int类型

- 上述生成的矩阵与向量均为**Local Matrix**与**Local Vector**：

        - Local Matrix：只存在**主节点**上的矩阵，不是分布式矩阵

    因此，下面我们考虑RDD中线性运算类。

- **RDD中的线性运算类：**
    - IndexedRow：
        - IndexedRow(Index, Vectors)：包含两部分，即"Index"和"Vectors"
            - 原因：在分布式矩阵的计算中，本质上将各行作为元素，存储到不同的节点；因此需要添加一个Index进行排序，从而将**不同Index的向量分到不同的节点上。**

        ```
        scala> IndexedRow(10,Vectors.dense(Array(1.0,2.0,3.0)))
        res42: org.apache.spark.mllib.linalg.distributed.IndexedRow = IndexedRow(10,[1.0,2.0,3.0])
        ```

        注意： Index必须是一个Long型数据！
    - IndexedRowMatrix：将IndexedRow组成一个矩阵，即为IndexedRowMatrix，且同样是分布式存储。
        - 生成方法：`new IndexedRowMatrix(X)`，其中X为一个各元素均为IndexedRow的RDD
        - IndexedRowMatrix的运算功能有限，此处只有两种，即乘法运算与分块运算：
            - 矩阵乘法运算：

                `A.multiply(B)`：A与一个Local Matrix B相乘，即与上述`sparkDenseMatrix`函数生成的矩阵相乘。
            - 矩阵的分块操作：
                - `A.toBlockMatrix(rowsPerBlock,colsPerBlock)`：将矩阵A转化为一个分块矩阵；其中rowsPerBlock为每一块的行数，colsPerBlock为每一块的列数。

            流程归纳：
        - *Step1*：Spark中的一个Local Vector（由`Vectors.dense()`生成）添加Index，转换为IndexedRow（RDD）；
        - *Step2*：将IndexedRow（RDD）集合，从而生成一个IndexedRowMatrix（RDD）；
        - *Step3*：将IndexedRowMatrix转换为分块矩阵，从而才能进行分布式的运算。
        - 即：**主节点→分布式→分块**
        - 转换原因：正因为IndexedRowMatrix的运算功能有限，因此要转换为BlockMatrix进行运算！
    - BlockMatrix的运算：
        - `A.add(B)`：将两个BlockMatrix A和B相加
        - `A.multiply(B)`：将两个BlockMatrix A和B相乘
        - `A.subtract(B)`：从BlockMatrix A中减去BlockMatrix B
        - `A.transpose`：将BlockMatrix A进行转置
        - `A.toLocalMatrix`：将分布式的BlockMatrix A收取到主节点上，形成一个**sparkDenseMatrix**
        - `A.toIndexedRowMatrix()`：将BlockMatrix A转换成一个**IndexedRowMatrix**

        注意： 矩阵求逆无法分块计算！

- 从线性回归模型中产生随机实验数据，并用于分位数回归的估计：

$$y_i = x_{1,i} + x_{2,i} + x_{3,i} + \cdots + x_{100,i} + x_{1,i}\epsilon_i$$

- 首先导入相关的包：

```
scala> import org.apache.spark.SparkContext
import org.apache.spark.SparkContext

scala> import org.apache.spark.SparkConf
import org.apache.spark.SparkConf

scala> import org.apache.spark.mllib.linalg.distributed._
import org.apache.spark.mllib.linalg.distributed._

scala> import org.apache.spark.mllib.linalg.{Vectors,Matrix => sparkMatrix,DenseMatrix =>
sparkDenseMatrix}
```

```
import org.apache.spark.mllib.linalg.{Vectors, Matrix=>sparkMatrix, DenseMatrix=>sparkDenseMatrix}

scala> import breeze.linalg._
import breeze.linalg._

scala> import breeze.stats.distributions._
import breeze.stats.distributions._

scala> import scala.math._
import scala.math._
```

- 由于在求取矩阵的逆时，需要转换为`breeze`的`DenseMatrix`进行运算，又因没有直接进行转换的函数，故需要声明一个转换函数，将sparkMatrix转换为DenseMatrix：
    - 思路：将sparkMatrix的各个元素逐一“复制”到DenseMatrix中即可
    - 运用函数：对于sparkMatrix A，可以用`A.numRows`返回A的行数，`A.numCols`返回A的列数
        - 回顾：对于breeze的DenseMatrix，其行数与列数用`A.rows`与`A.cols`返回

```
scala> DenseMatrix((1.0,2.0,3.0),(4.0,5.0,6.0)).cols
res13: Int = 3
```

声明函数`tobreezematrix`：

```
scala> def tobreezematrix(tmp1: sparkMatrix): DenseMatrix[Double] = {
     |     var tmp2:DenseMatrix[Double] = DenseMatrix.zeros[Double](tmp1.numRows, tmp1.numCols)
     |     for(i <- 0 until tmp1.numRows){
     |           for(j <- 0 until tmp1.numCols){
     |               tmp2(i,j) = tmp1(i,j)
     |           }
     |     }
     |     tmp2
     | }
tobreezematrix: (tmp1: org.apache.spark.mllib.linalg.Matrix)breeze.linalg.DenseMatrix[Double]
```

注意：此处在生成变量时，用了`var 变量名:变量类型=值`，其中的“变量类型”可加可不加；

- 初始化数据模拟的模型设置

```
scala> val P = sc.broadcast(100)                              //P为协变量的数目
P: org.apache.spark.broadcast.Broadcast[Int] = Broadcast(0)

scala> val N = sc.broadcast(10000)                            //N为观测数据的数目(行数)
N: org.apache.spark.broadcast.Broadcast[Int] = Broadcast(1)   //综上：设计矩阵为10000×100维

scala> val r1 = sc.broadcast(2)                               //r1与tau为模型默认参数
r1: org.apache.spark.broadcast.Broadcast[Int] = Broadcast(26)

scala> val tau = sc.broadcast(0.5)
tau: org.apache.spark.broadcast.Broadcast[Double] = Broadcast(3)

scala> val iter = sc.broadcast(20)                            //iter为最大迭代次数
iter: org.apache.spark.broadcast.Broadcast[Int] = Broadcast(4)

scala> var indeces = 0 until N.value                          //indeces为IndexedRow的标签
Index(10000行)
indeces: scala.collection.immutable.Range = Range 0 until 10000

scala> val Beta = sc.broadcast(DenseVector.ones[Double](P.value))    //Beta为参数向量
Beta: org.apache.spark.broadcast.Broadcast[breeze.linalg.DenseVector[Double]] = Broadcast(5)

scala> val PallalelIndeces = sc.parallelize(indeces)          //将行标签进行分布式存储
PallalelIndeces: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[0] at parallelize at
<console>:43
```

思考： 为何要将行标签进行分布式存储?

解答： 因为在构造设计矩阵时，需要利用 `map` 函数，将每一个标签转换为一个IndexedRow。

- 构造设计矩阵:
  - 补充知识:
    - 1. `sample()` 函数得到的是一个Vector，而不是Array，所以需要进行转换，即 `toArray`

```
scala> val test = new Gaussian(0,1)
test: breeze.stats.distributions.Gaussian = Gaussian(0.0, 1.0)

scala> test.sample(10)
res17: IndexedSeq[Double] = Vector(1.0662118602747506, 1.9351797437531548,
-1.0625496927168014, 2.1597022224710916, 0.4200699080548156, -0.3116637463914656,
-0.2595464792920875, -0.5514885681117637, 0.9115839226592733, 1.0464810877228836)

scala> test.sample(10).toArray
res18: Array[Double] = Array(1.6520614663206845, -0.3472462815959844,
-0.7421792686102668, 0.9113765181988956, -1.5584019791317885, -0.9534380744366391,
0.7362123752571006, -1.1343189286303623, 1.5443346294951814, -0.5614933309528332)
```

2. 抽取一个样本时，可以用 `sample(1)` ，也可以用 `draw()` 函数:

```
scala> test.draw()
res19: Double = 0.15031961588178544
```

3. 生成DenseMatrix时，除了可以直接 `DenseMatrix((Array_1),...,(Array_n))`，还可以 `new DenseMatrix(rows=m,cols=n,Array)` 从而创建一个m行n列的矩阵，并且Array以列的方式进行填充:

```
scala> new DenseMatrix(rows=1,cols=3,Array(1.0,2.0,3.0))
res20: breeze.linalg.DenseMatrix[Double] = 1.0  2.0  3.0

scala> new DenseMatrix(rows=2,cols=2,Array(1.0,2.0,3.0,4.0))
res21: breeze.linalg.DenseMatrix[Double] =
1.0  3.0
2.0  4.0
```

注意： 必须传入一个Array，不能是数组或Vector；

4. DenseMatrix的行列索引均从0开始，且用数组进行索引:

```
scala> val x = new DenseMatrix(rows=2,cols=2,Array(1.0,2.0,3.0,4.0))
x: breeze.linalg.DenseMatrix[Double] =
1.0  3.0
2.0  4.0

scala> x(0,0)
res25: Double = 1.0
```

5. 矩阵与矩阵运算的返回值为**矩阵**，而矩阵与向量的运算返回值为**向量**:

```
scala> val x = new DenseMatrix(rows=1,cols=3,Array(1.0,2.0,3.0))
x: breeze.linalg.DenseMatrix[Double] = 1.0  2.0  3.0

scala> val y = DenseMatrix(1.0,2.0,3.0)
y: breeze.linalg.DenseMatrix[Double] =
1.0
2.0
3.0

scala> x*y
res30: breeze.linalg.DenseMatrix[Double] = 14.0
```

```
scala> val z = DenseVector(1.0,2.0,3.0)
z: breeze.linalg.DenseVector[Double] = DenseVector(1.0, 2.0, 3.0)

scala> x*z
res31: breeze.linalg.DenseVector[Double] = DenseVector(14.0)
```

因此，若要提取内积，需要进行索引：

```
scala> val t = x*z
t: breeze.linalg.DenseVector[Double] = DenseVector(14.0)

scala> t(0)
res35: Double = 14.0
```

6. 此处y的生成参考公式：

$$y_i = x_{1,i} + x_{2,i} + x_{3,i} + \cdots + x_{100,i} + x_{1,i}\epsilon_i$$

7. `union`函数的作用对象一定是Array，因此若要连接一个数字，则需要在外面嵌套`Array()`.

生成设计矩阵：

```
scala> var sample_matrix = PallallelIndeces.map(s => {
     |   var p = 100
     |   var norm_list = new Gaussian(0,1)
     |   var x = new DenseMatrix(rows=1,cols=p,norm_list.sample(p).toArray)
     |   var y = x*Beta.value + x(0,0)*norm_list.draw()
     |   Array(s.toDouble).union(x.toArray).union(Array(y(0)))
     | })       //得到的矩阵第0列为Index，1到100列为协变量的观测值，101列为相应的y值
sample_matrix: org.apache.spark.rdd.RDD[Array[Double]] = MapPartitionsRDD[1] at map at <console>:43

scala> sample_matrix.persist()
res43: org.apache.spark.rdd.RDD[Array[Double]] = MapPartitionsRDD[1] at map at <console>:57

scala> sample_matrix.collect()
res6: Array[Array[Double]] = Array(Array(0.0, -1.2447918623783025, -1.5311671843408896,
0.8395263456932032, -0.2958391805890554, 1.270555781899143, 1.9610747733355283, 0.0370897332854251,
-1.4220863916401978, -1.032707306838984, 0.10061298369608421, 0.7133117263853416,
1.2403226730945365, 2.1219758601616148, 0.030604871827160242, -0.12837285662381978,
-0.29541900260312226, -1.776796599655421, -0.41525380170742787, -1.2593007081863221,
-0.8240465824575977, -0.47658901527811365, -0.6777729176425852, -0.1504678656060219,
1.2053218330967967, 0.35975445459563765, 0.7745407986459804, 0.5067941861463653,
-0.8734385719829749, 0.40656438872674944, 1.1461809278990902, -1.9530309986993386,
-0.2633405863247586, 0.9625658247571406, 0.46645447840053844, 0.10051461060040094,
-0.1115587520340581, -0.35...
```

- 将设计矩阵进行拆分，分别为[Index, X]与[Index, y]
    - 先拆分[Index, X]

```
scala> var sample_x = sample_matrix.map(f => IndexedRow(f.take(1)
(0).toLong,Vectors.dense(f.drop(1).dropRight(1))))
sample_x: org.apache.spark.rdd.RDD[org.apache.spark.mllib.linalg.distributed.IndexedRow] =
MapPartitionsRDD[2] at map at <console>:42
```

注意：

1. 本质上此处`f.take(1)(0)`与`f(0)`没有区别；

```
scala> sample_matrix.collect()(0).take(1)
res11: Array[Double] = Array(0.0)

scala> sample_matrix.collect()(0)(0)
res12: Double = 0.0
```

2. 向量只去除第一个元素时，可以用`drop(1)`，而在去除倒数第一个元素时，可以用`dropRight(1)`。
    - 再拆分[Index, y]

```
scala> var sample_y = sample_matrix.map(f => IndexedRow(f.take(1)
(0).toLong,Vectors.dense(f.drop(P.value+1))))
sample_y: org.apache.spark.rdd.RDD[org.apache.spark.mllib.linalg.distributed.IndexedRow] =
MapPartitionsRDD[3] at map at <console>:43
```

注意：

1. 在去除y的值时，可以用`f.drop(P.value+1)`，即舍弃前101个值；这么做的好处是：保持Array的结构，以符合`Vectors.dense`函数的参数类型；

```
scala> P.value+1
res13: Int = 101

scala> sample_matrix.collect()(0).drop(P.value+1)
res14: Array[Double] = Array(3.4109893415114536)

scala> sample_matrix.collect()(0).size
res17: Int = 102
```

2. 也可以直接取出y值，再套用`Array()`：

```
scala> Array(sample_matrix.collect()(0)(101))
res49: Array[Double] = Array(3.4109893415114536)
```

- 将IndexedRow集合生成一个IndexedRowMatrix（目的：生成分块矩阵）：

```
scala> var sample_x_indexedrowmatrix = new IndexedRowMatrix(sample_x)
sample_x_indexedrowmatrix: org.apache.spark.mllib.linalg.distributed.IndexedRowMatrix =
org.apache.spark.mllib.linalg.distributed.IndexedRowMatrix@2c499df0

scala> var sample_y_indexedrowmatrix = new IndexedRowMatrix(sample_y)
sample_y_indexedrowmatrix: org.apache.spark.mllib.linalg.distributed.IndexedRowMatrix =
org.apache.spark.mllib.linalg.distributed.IndexedRowMatrix@5017c959
```

- 将上述矩阵进行分块，即转换为BlockMatrix：

```
scala> var x = sample_x_indexedrowmatrix.toBlockMatrix(rowsPerBlock = N.value/10,colsPerBlock =
P.value/10)
x: org.apache.spark.mllib.linalg.distributed.BlockMatrix =
org.apache.spark.mllib.linalg.distributed.BlockMatrix@25eca20b

scala> var y = sample_y_indexedrowmatrix.toBlockMatrix(rowsPerBlock = N.value/10,colsPerBlock =
P.value)
y: org.apache.spark.mllib.linalg.distributed.BlockMatrix =
org.apache.spark.mllib.linalg.distributed.BlockMatrix@60ad1ce2
```

注意： 此处将X分块为10×10的矩阵，将y分块为10×1的矩阵

- 生成β的初值：

```
scala> var beta_hat, beta_old: DenseMatrix[Double] = DenseMatrix.zeros[Double](P.value,1)
beta_hat: breeze.linalg.DenseMatrix[Double] =
0.0
0.0
0.0
0.0
0.0
... (100 total)
beta_old: breeze.linalg.DenseMatrix[Double] =
```

```
0.0
0.0
0.0
0.0
0...
```

注意： 当两个变量取相同值的时候，可以直接用`var 变量1,变量2 = 值`进行定义；

- 根据模型需要，定义z与α的初值：

```
scala> var z: DenseMatrix[Double] = DenseMatrix.zeros[Double](N.value,1)
z: breeze.linalg.DenseMatrix[Double] =
0.0
0.0
0.0
0.0
... (10000 total)

scala> var alpha: DenseMatrix[Double] = DenseMatrix.zeros[Double](N.value,1)
alpha: breeze.linalg.DenseMatrix[Double] =
0.0
0.0
0.0
... (10000 total)
```

注意： 本质上α与z均为10000维的向量，但为了后续矩阵运算的方便，将其定义矩阵；

- 计算$(X^\top X)^{-1}$：

  补充： 对于一个IndexedRowMatrix，无法直接显示其元素，我们需要将其转换为LocalMatrix才能看到矩阵的实际形式：

```
scala> x
res51: org.apache.spark.mllib.linalg.distributed.BlockMatrix =
org.apache.spark.mllib.linalg.distributed.BlockMatrix@5bc979

scala> x.toLocalMatrix()
res54: org.apache.spark.mllib.linalg.Matrix =
-0.16972629332252995   -1.5685148414520471    ... (100 total)
0.5002388426025064     -1.2032832407555762    ...
1.2975016726342612     0.8275157762502359     ...
2.035869129768978      -0.25690774312253206   ...
-0.5868408134001343    0.3115969482793508     ...
0.010709849669210453   0.5983383184170089     ...
0.20336005115257338    -0.04608346895970109   ...
-1.592638654840383     0.08999438527275608    ...
1.0261065373807807     -2.0956908406796098    ...
-2.267502485560876     -0.2977272213627126    ...
1.8916833796187063     -0.8780447616633527    ...
-0.005483008991321771  1.0397649048384892     ...
1.5956697763407988     0.7678857981939323     ...
0.3947280365403289     -0.047932520498053924  ...
-0.6986812295305993    0...
```

注意： `toLocalMatrix()`函数的返回值是Matrix类型！

  - 首先计算$X^\top X$：

```
scala> val tmp1 = x.transpose.multiply(x).toLocalMatrix()
tmp1: org.apache.spark.mllib.linalg.Matrix =
10223.878562833468   10.366436573583648   -258.88860729929115  ... (100 total)
10.366436573583648   9924.899008694267    122.93515942366577   ...
-258.88860729929115  122.93515942366577   9957.451305882982    ...
-125.44204776071028  -39.692231869449536  29.847578619555176   ...
-130.5622523672725   -148.17795035937905  75.66455399541454    ...
-121.84860313190929  50.041777075416746   -80.00290664609417   ...
-7.144345468586522   -109.03738832048532  -93.8820738172258    ...
140.67779784036253   -1.778532141184391   -140.5413957930944   ...
-28.05563025882016   93.92173831022602    -199.36618795174599  ...
195.76063495052705   13.625265262247629   -72.2573728668649    ...
-1.8002579345063872  -156.5326277915792   -110.233871628312...
```

- 由于分块矩阵无法直接计算逆，故需要将其转换为`breeze`的DenseMatrix：

```
scala> var tmp2 = tobreezematrix(tmp1)
tmp2: breeze.linalg.DenseMatrix[Double] =
10223.878562833468   10.366436573583648   -258.88860729929115  ... (100 total)
10.366436573583648   9924.899008694267    122.93515942366577   ...
-258.88860729929115  122.93515942366577   9957.451305882982    ...
-125.44204776071028  -39.692231869449536  29.847578619555176   ...
-130.5622523672725   -148.17795035937905  75.66455399541454    ...
-121.84860313190929  50.041777075416746   -80.00290664609417   ...
-7.144345468586522   -109.03738832048532  -93.8820738172258    ...
140.67779784036253   -1.778532141184391   -140.5413957930944   ...
-28.05563025882016   93.92173831022602    -199.36618795174599  ...
195.76063495052705   13.625265262247629   -72.2573728668649    ...
-1.8002579345063872  -156.5326277915792   -110.2338716283123  ...
```

- 最后计算$X^\top X$的逆矩阵：

```
scala> val beta_fixed = inv(tmp2)
beta_fixed: breeze.linalg.DenseMatrix[Double] =
9.890091317233849E-5    -1.854679058026976E-7   ... (100 total)
-1.8546790580269776E-7  1.0152745852443541E-4   ...
2.375843494881291E-6    -1.336760712647236E-6   ...
1.189470705658601E-6    4.666799099435082E-7    ...
1.2037991605042058E-6   1.544649889691965E-6    ...
1.4850560326458429E-6   -5.470193630094287E-7   ...
1.828483064424705E-7    1.00918722622145E-6     ...
-1.3518162194988243E-6  6.156993889392175E-8    ...
4.0551331840963535E-7   -9.405770504435701E-7   ...
-1.9422854347285156E-6  -1.929190844647729E-7   ...
-1.5763102286300564E-8  1.4531644002583156E-6   ...
-6.584668842477257E-8   -1.1092117652169183E-6  ...
2.0977589465666735E-6   -2.377631622695257E-8   ...
-4.1884748422608294E-7  5.128075091819994E-7   ...
```

- 将y转换为LocalMatrix：

```
scala> var y_local = y.toLocalMatrix()
y_local: org.apache.spark.mllib.linalg.Matrix =
-3.4993671719440007
3.459974093170319
9.592430322324...
```

目的：便于后续转换为DenseMatrix，从而与z相减；

- 初始化迭代器与最大迭代次数

```
scala> var i: Int = 0
i: Int = 0

scala> var diff: Double = 1.0
diff: Double = 1.0
```

- 进行迭代

```
scala> while(i < iter.value && diff > pow(10 , -6)){
     | beta_old = beta_hat
     | var tmp3 = (tobreezematrix(y_local) - z).toArray
     | var beta_right = x.transpose.toIndexedRowMatrix.multiply(new
sparkDenseMatrix(N.value,1,alpha.map(s => s/r1.value).toArray)).toBlockMatrix(rowsPerBlock =
P.value/10,colsPerBlock = 1).add(x.transpose.toIndexedRowMatrix.multiply(new
sparkDenseMatrix(N.value,1,tmp3)).toBlockMatrix(rowsPerBlock = P.value/10,colsPerBlock =
1)).toLocalMatrix()
     | beta_hat = beta_fixed * tobreezematrix(beta_right)
     | var tmp4 = y.subtract(x.toIndexedRowMatrix.multiply(new
sparkDenseMatrix(P.value,1,beta_hat.toArray)).toBlockMatrix(rowsPerBlock = N.value/10,colsPerBlock
= 1)).toLocalMatrix
     | z = (tobreezematrix(tmp4) + alpha.map(s => s/r1.value)).map(s =>{
     | if((tau.value/r1.value)<s)
     | { s-tau.value/r1.value
     | } else if(s<((tau.value - 1)/r1.value)){
     | s -(tau.value - 1)/r1.value} else 0 })
     | alpha = alpha + (tobreezematrix(tmp4)-z).map(s => s*r1.value)
     | diff = sum((beta_old - beta_hat).map(s => abs(s)))
     | i = i + 1
     | }
```

注意：

1. 在 `(i<iter.value) && (diff>pow(10, -6))` 中， `&` 与 `&&` 等价：

```
scala> (2 < 3) & (5 < 6)
res60: Boolean = true

scala> (2 < 3) && (5 < 6)
res61: Boolean = true
```

2. 由于z为breeze的DenseMatrix，因此相减时，需要将y也化为breeze的DenseMatrix，否则无法计算；
3. 为何需要将x转换为IndexedRowMatrix？因为在与α相乘时，此时α并不是一个分块矩阵，因此分别将x转换为IndexedRowMatrix，再将alpha转换为LocalMatrix，从而便于计算，即：

```
new sparkDenseMatrix(N.value, 1, alpha.map(s => s/r1.value).toArray)
```

4. 又由于需要进行分块计算，因此将得到的$\frac{1}{r_1}X^T\alpha$转换为分块矩阵，即：

```
var beta_right = x.transpose.toIndexedRowMatrix.multiply(new sparkDenseMatrix(N.value, 1,
alpha.map(s => s/r1.value).toArray)).toBlockMatrix(rowsPerBlock = P.value/10, colsPerBlock=1)
```

5. 再加上$X^T\left(y-z^k\right)$，与上面类似的步骤，先将X转置并转换为IndexedRowMatrix，再将y-z转换为LocalMatrix：

```
add(x.transpose.toIndexedRowMatrix.multiply(new sparkDenseMatrix(N.value, 1,
tmp3)).toBlockMatrix(rowsPerBlock = P.value/10, colsPerBlock=1)).toLocalMatrix()
```

6. 最后，为了能与$(X^\top X)^{-1}$相乘，因此先将beta转换为LocalMatrix，如上所示，再将其转换为breeze中的DenseMatrix：

```
beta_hat = beta_fixed*tobreezematrix(beta_right)
```

从而对于β的更新结束，再开始更新z；

7. 再计算$y_i - x_i^T\beta^{k+1}$，根据y为BlockMatrix，因此需要将X*beta转换为BlockMatrix；而X为BlockMatrix，beta为breeze的DenseMatrix，因此相乘时需要将X转换为IndexedRowMatrix，beta转换为LocalMatrix，才能进行计算：

```
var tmp4 = y.subtract(x.toIndexedRowMatrix.multiply(new
sparkDenseMatrix(P.value,1,beta_hat.toArray)).toBlockMatrix(rowsPerBlock = N.value/10, colsPerBlock
= 1)).toLocalMatrix
```

由于最终还需要加上$\frac{\alpha_i^k}{r_1}$，相当于向量加上$\frac{\alpha^k}{r_1}$，而alpha为DenseMatrix，因此需要将上述结果转换为LocalMatrix，进一步转换为breeze的DenseMatrix；

8. 再根据分段函数，进行z的更新：

$$z_i^{k+1} = \begin{cases} a_i^k - \frac{\tau}{r_1} & \frac{\tau}{r_1} < a_i^k \\ 0 & \frac{\tau-1}{r_1} \le a_i^k \le \frac{\tau}{r_1} \\ a_i^k - \frac{\tau-1}{r_1} & a_i^k < \frac{\tau-1}{r_1} \end{cases}$$

```
z = (tobreezematrix(tmp4)+alpha.map(s=> s/r1.value)).map(s=> {
    if((tau.value/r1.value)<s) s-tau.value/r1.value else if(s < ((tau.value - 1)/r1.value)) s-
(tau.value-1)/r1.value else 0
})
```

注意到此处的`tau.value/r1.value`均为常数值，且进行了广播；

9. 最后进行α的更新：

$$\alpha^{k+1} = \alpha^k + r_1\left(y - X\beta^{k+1} - z^{k+1}\right)$$

由于$y - X\beta^{k+1}$已经在前面计算出来，即为tmp4，且为LocalMatrix；而z为DenseMatrix，因此需要先将tmp4转换为DenseMatrix，再减去z的值；

```
alpha = alpha + (tobreezematrix(tmp4)-z).map(s => s*r1.value)
```

注意：对于LocalMatrix，不能直接与一个常数进行运算，所以需要利用map函数，逐个运算；

10. 更新当前的diff与迭代次数i，判断是否进入下一次循环：

```
diff = sum((beta_old - beta_hat).map(s => abs(s)))
i+=1
```

- β的最终迭代值

```
scala> beta_hat
res79: breeze.linalg.DenseMatrix[Double] =
1.0196353237447766
0.9961435388753934
1.0093461665187138
1.0013362097713594
1.003323044754393
1.0064740732444215
0.9978957670267075
1.0060348505793064
0.999581067410646
1.0029016874089982
0.998423309656059
0.9878396771201597
0.9987065928926906
1.000170793868079
1.0005463646140509
1.0026718272947472
0.999873900536493
1.0006162833784957
```

```
1.0025994811659642
1.003101929548613
1.0032358927233296
1.0030810473143719
1.0086717529194251
0.9976047634509991
0.9973812235294028
1.0051177405172145
1.0036785822573004
0.9995300039078066
1.011907651568113
0.9958897761973915
0.9993211399659778
1.006625380062999
1.0022127605487394
0.9954044008195644
0.9997192593271021
1.0028038657547012
1.000662131709638
0.999627878936693
0....

scala> sum(beta_hat)
res80: Double = 100.04906898817536
```

Lasso算法

For the problem of LASSO, we have:

$$f(\beta)=\frac{1}{2}\|y-X\beta\|_2^2+\lambda\|\beta\|_1$$

and the KKT condition can be expressed as:

$$X^T(y-X\beta)=\lambda\times\partial\|\beta\|_1$$

where we have:

$$\partial\|\beta\|_1=\left(\frac{\partial|\beta_1|}{\partial\beta_1},\frac{\partial|\beta_2|}{\partial\beta_2},\ldots,\frac{\partial|\beta_p|}{\partial\beta_p}\right)^T$$

More specifically, we have:

$$\frac{\partial\left|\beta_i\right|}{\partial\beta_i}=\begin{cases}1 & \text{if } \beta_i>0\\ -1 & \text{if } \beta_i<0\\ [-1,1] & \text{if } \beta_i=0\end{cases}$$

Therefore, denote the i-th column of X by $X_{(i)}$, then we know that:

- If $|X_{(i)}^T(y-X\beta)|<\lambda$,then $\beta_i=0$.
- If $\beta_i\neq 0$, then we have:$X_{(i)}^T(y-X\beta)=\lambda\frac{\beta_i}{|\beta_i|}$, which implies that

$$X_{(i)}^T\left(y-\sum_{j\neq i}X_{(j)}\beta_j-X_{(i)}\beta_i\right)=\lambda\frac{\beta_i}{|\beta_i|}$$

and

$$X_{(i)}^T\left(y-\sum_{j\neq i}X_{(j)}\beta_j\right)=\left(\frac{\lambda}{|\beta_i|}+X_{(i)}^TX_{(i)}\right)\beta_i$$

and finally, we have:

$$\beta_i=X_{(i)}^T\left(y-\sum_{j\neq i}X_{(j)}\beta_j\right)/\left(\frac{\lambda}{|\beta_i|}+X_{(i)}^TX_{(i)}\right)$$

Consequently, we have the iteration algorithm as follows:

Denote the estimator of β_i in the m-th iteration by $\beta_i[m]$. Assume we have the initial value of β_i as $\beta_i[0]$.

For the m-th iteration, given the estimators $\beta[m-1]=(\beta_1[m-1],\beta_2[m-1],\ldots,\beta_p[m-1])$ obtained in the previous iteration, we have:

If $\left|X_{(i)}^T(y-X\beta[m-1])\right|<\lambda$, then $\beta_i[m]=0$.

Otherwise:

$$\beta_i[m]=X_{(i)}^T\left(y-\sum_{j\neq i}X_{(j)}\beta_j[m-1]\right)/\left(\frac{\lambda}{|\beta_i[m-1]|}+X_{(i)}^TX_{(i)}\right)$$

HW4

考虑如下的线性回归模型:

$$y_i=\sum_{j=1}^{p}\beta_jx_{ij}+\epsilon_i,\quad i=1,\ldots,N$$

其中$x_{ij}\sim N(0,1)$和y分别是解释变量和响应变量，而$\epsilon\sim N(0,0.04)$是模型误差。当$j=1,2,3$时，$\beta_j=2$；当$j>3$时，$\beta_j=0,N=1000,p=200$.

1. **试编写Scala程序求上述模型的LASSO估计量:**

$$\hat\beta(\lambda)=\arg\min_{\beta}\frac{1}{2}\|y-X\beta\|_2^2+\lambda\|\beta\|_1$$

其中λ是调试参数，$\|\cdot\|$表示L_1模。

在本次作业中，我们取$\lambda=0.85\times10^{-12}$，$\beta=(\beta_1,\beta_2,\cdots,\beta_p)$的初值取它的最小二乘估计量。最大迭代次数为6次。

```
import breeze.linalg._
import java.util.concurrent.ThreadLocalRandom
import scala.math._
val r = ThreadLocalRandom.current

val N = 1000
val p = 200
val lambda = 0.85e-12
val X = DenseMatrix.tabulate(N,p){case(i,j) => r.nextGaussian}
val epsilon = DenseVector.tabulate(N){i => r.nextGaussian * 0.2}
val beta = DenseVector.tabulate(p){i => {if(i < 3) 2.0 else 0.0}}
val y = X * beta + epsilon
val kmax = 6
var Estbeta = inv(X.t*X)*X.t*y
var OldEstbeta = DenseVector.zeros[Double](p)
var k = 0

while(k < kmax)
{
    OldEstbeta(0 until p) := Estbeta(0 until p)
    for(i <- 0 until p){
        if(abs(X(::,i).t*(y - X*OldEstbeta)) < lambda){
            Estbeta(i) = 0
        }else{
            Estbeta(i) = X(::,i).t*(y - X*OldEstbeta +
X(::,i)*OldEstbeta(i))/(lambda/abs(OldEstbeta(i))+X(::,i).t*X(::,i))
        }
    }
    k = k + 1
}
```

2. **将问题1中的代码使用MapReduce方法进行分布式改进。**

```
import breeze.linalg._
```

```scala
import java.util.concurrent.ThreadLocalRandom
import scala.math._
val r = ThreadLocalRandom.current

val N = 1000
val p = 200
val lambda = 0.85e-12
val X = DenseMatrix.tabulate(N,p){case(i,j) => r.nextGaussian}
val epsilon = DenseVector.tabulate(N){i => r.nextGaussian * 0.2}
val beta = DenseVector.tabulate(p){i => {if(i < 3) 2.0 else 0.0}}
val y = X * beta + epsilon
val kmax = 6
var Estbeta = inv(X.t*X)*X.t*y
var OldEstbeta = DenseVector.zeros[Double](p)
var k = 0

while(k < kmax)
{
    OldEstbeta(0 until p) := Estbeta(0 until p)
    val IndexedOldEstbeta = DenseVector.tabulate(p)(i => (i,OldEstbeta(i))).toArray
    val ParIndexed = sc.parallelize(IndexedOldEstbeta)
    val Results = ParIndexed.map(s => {
        val i = s._1
        if(abs(X(::,i).t*(y - X*OldEstbeta)) < lambda){
            0
        }else{
            X(::,i).t*(y - X*OldEstbeta + X(::,i) * OldEstbeta(i)) /
            (lambda / abs(OldEstbeta(i)) + X(::,i).t * X(::,i))
        }
    })
    Estbeta = DenseVector(Results.collect())
    k = k + 1
}
Estbeta
```

注意：

1. 用MapReduce方法指的是转换为RDD进行分布式运算，而不是单纯地利用Map函数与Reduce函数！
2. 无法对DenseVector进行分布式，一般是对Array分布式运算。

第九次课

随机梯度下降算法

- 在统计参数估计问题中，我们需要通过数据来估计参数，因此我们经常需要考虑最小化某一个目标函数：

$$\rho_n(\theta)=N^{-1}\sum_{i=1}^{N}d\left(m\left(x_i;\theta\right),y_i\right)$$

其中x和y分别是解释变量和响应变量，d是某种损失函数，而m是某种依赖于参数θ的函数，比如$m(x;\theta)=x^T\theta$.

我们记$l\left(z_i;\theta\right)=d\left(m\left(x_i;\theta\right),y_i\right)$，其中$z_i=\left(x_i^T,y_i\right)^T$，因此有

$$\rho_n(\theta)=N^{-1}\sum_{i=1}^{N}l\left(z_i;\theta\right)$$

由于在梯度的反方向上，函数值下降最快，因此由梯度下降法：

$$\theta^{(t+1)}=\theta^{(t)}-\gamma\nabla\rho(\theta^{(t)})$$

利用梯度下降法，我们得到迭代公式如下：

$$\theta^{(t+1)}=\theta^{(t)}-\gamma N^{-1}\sum_{i=1}^{N}\nabla l\left(z_i;\theta^{(t)}\right)$$

- 随机梯度下降法的基本思想是用一个随机选择的观察值$z^{(t)}$的梯度来替代计算所有数据梯度的平均值，即迭代公式为：

$$\theta^{(t+1)} = \theta^{(t)} - \gamma^{(t)} \nabla l \left(z^{(t)}; \theta^{(t)} \right)$$

优化方法与自举法

- 调用不同的目标函数，并引入相关的包：

```
scala> import org.apache.spark.mllib.optimization.{GradientDescent, LogisticGradient, SquaredL2Updater}
import org.apache.spark.mllib.optimization.{GradientDescent, LogisticGradient, SquaredL2Updater}
```

其中导入的函数分为三块，分别为`GradientDescent`,`LogisticGradient`与`SquaredL2Updater`：

`GradientDescent`：表示梯度下降函数

`LogisticGradient`类型的函数由变量的类型决定：（相当于使用不同的梯度类型）

当响应变量是一个二元变量的时候，使用：`LogisticGradient`

当响应变量是一个连续变量的时候，使用：`LeastSquaresGradient`

`SimpleUpdater`类型的函数由目标函数的类型决定：

当使用最小二乘估计的目标函数，使用`SimpleUpdater`

当使用岭回归的目标函数，使用`SquaredL2Updater`

当使用Lasso的目标函数，使用`L1Updater`

注意：当使用最小二乘估计的目标函数，应使用`SimpleUpdater`，而不是`LeastSquareUpdater`！

- 第四种生成随机数的方法，利用`scala.util.Random`包：

```
scala> import scala.util.Random
import scala.util.Random

scala> val random = new Random()
random: scala.util.Random = scala.util.Random@5b859845

scala> val u = random.nextDouble
u: Double = 0.8310613738794326

scala> random.nextGaussian
res0: Double = 0.05049307144140205
```

随机梯度下降算法

模型：样本量为40的数据集，在数据集中的解释变量的个数为2000，响应变量的取值为0或1，对该数据拟合logistic回归，并通过随机梯度下降法来估计参数，即考虑如下的线性回归模型：

$$y_i = x_i^T \beta_0 + \epsilon_i, \quad i = 1, \ldots, N$$

其中x和y分别是解释变量和响应变量，而ϵ是模型误差。

- 首先引入相关的包

```
scala> import scala.collection.JavaConverters._
import scala.collection.JavaConverters._

scala> import scala.util.Random          //用于生成随机数
import scala.util.Random

scala> import org.apache.spark.mllib.linalg.Vectors   //目的：在随机梯度下降算法中的数据需要为(Double,Vector)格式
import org.apache.spark.mllib.linalg.Vectors

scala> import org.apache.spark.mllib.regression._
import org.apache.spark.mllib.regression._
```

```
scala> import org.apache.spark.mllib.optimization.{GradientDescent, LogisticGradient,
SquaredL2Updater}
import org.apache.spark.mllib.optimization.{GradientDescent, LogisticGradient, SquaredL2Updater}
```

解释：

引入的最后一个包说明，此模型应用随机梯度下降法(`GradientDescent`)、logistic回归(`LogisticGradient`)和岭回归目标函数(`SquaredL2Updater`)。

- 定义gradient变量与updater变量

```
scala> val gradient = new LogisticGradient()
gradient: org.apache.spark.mllib.optimization.LogisticGradient =
org.apache.spark.mllib.optimization.LogisticGradient@17f856a3

scala> val updater = new SquaredL2Updater()
updater: org.apache.spark.mllib.optimization.SquaredL2Updater =
org.apache.spark.mllib.optimization.SquaredL2Updater@d3e9b94
```

定义此变量的目的：代入随后的随机梯度下降算法中。

- 定义其他需要的变量

```
scala> val stepSize = 0.1
stepSize: Double = 0.1

scala> val numIterations = 10
numIterations: Int = 10

scala> val regParam = 1.0
regParam: Double = 1.0

scala> val miniBatchFrac = 1.0
miniBatchFrac: Double = 1.0
```

变量解释：

stepSize：每次迭代的步长

numIterations：最大迭代次数

regParam：岭回归中的惩罚项系数

miniBatchFrac = 1.0：指利用所有的数据对模型进行训练

- 构造数据集

其中数据集points为一个**RDD格式**的数据，且数据中的每个记录需要为**tuple格式**`(Double,Vector)`，其中Vector为MLlib程序库里对应的Vector数据，2000个解释变量数据均为服从$U(0,1)$的随机数。

```
scala> val n = 40       //样本数目
n: Int = 40

scala> val p = 2000     //解释变量数目
p: Int = 2000

scala> val points = sc.parallelize(0 until n, 2).map{iter =>        //n个样本需要为RDD格式
     | val random = new Random()                                    //用于生成随机数的函数
     | val u = random.nextDouble()                                  //生成U(0,1)的随机数，从而生成y
     | val y = if(u > 0.5) 1.0 else 0.0
     | (y, Vectors.dense(Array.fill(p)(random.nextDouble())))
     | }
points: org.apache.spark.rdd.RDD[(Double, org.apache.spark.mllib.linalg.Vector)] =
MapPartitionsRDD[1] at map at <console>:34
```

- 回顾`Array.fill()()`函数的用法：

```
scala> Array.fill(3)(0.3)
res0: Array[Double] = Array(0.3, 0.3, 0.3)

scala> Array.fill(3)(random.nextDouble)
res2: Array[Double] = Array(0.3100338297950017, 0.3827139916390052, 0.09707942411907067)
```

注意：若第二个参数为`random.nextDouble`，则每一个数都是**单独生成的随机数**，而**不是相同的值！**

- 进行随机梯度下降算法

```
scala> val(weight,loss) = GradientDescent.runMiniBatchSGD(
     | points,
     | gradient,
     | updater,
     | stepSize,
     | numIterations,
     | regParam,
     | miniBatchFrac,
     | Vectors.dense(new Array[Double](p)))
weight: org.apache.spark.mllib.linalg.Vector =
[0.003101589123746108,-0.015840247107918745,-0.007774317781555606,-0.003806113094931952,-0.00863613
7281604877,-0.005148049107586421,-0.001594902075648122,-0.001884423335325916,-0.008962148944901091,
0.0013868444641252052,-0.005665572822255591,-0.0020799698556476184,0.005031525183761886,-0.00453622
2522621226,-0.004337097687316992,-0.002002418778051359б,0.003675628031430061,0.004099360893456437,7
.055644300952508E-
4,-0.010469016691143758,0.0022251802674151363,-0.007341138259945909,-0.005298714983047668,-0.004316
530769012353,-0.003352373386427165 5,-0.005486685811971845,-0.006852253632018 54,-0.0051111552941361 8
3,-0.005135443328541 06,0.001443188377580537,-0.008205120527875304,-0.0012501454147246576,0.00421169
6292260818,-0.006491113727497695,-0.009...
```

- 随机梯度下降算法函数解释：
    - 随机梯度下降算法的函数名为`GradientDescent.runMiniBatchSGD`
    - 随机梯度下降算法函数返回两个值：β的估计值weight、模型在迭代中各次拟合的loss（含初始值拟合的loss）
    - 第一个参数为RDD类型的数据集
    - 第二个参数为梯度类型（即引入包时的第二个参数生成的变量）
    - 第三个参数为目标函数的类型（即引入包时的第三个参数生成的变量）
    - 第四个参数为每次迭代的步长
    - 第五个参数为最大迭代次数
    - 第六个参数为惩罚项系数
    - 第七个参数为进行训练的数据比例（即前面所定义的miniBatchFrac）
    - 第八个参数初值（即估计量β的初值）
- 结果分析：

```
scala> weight
res4: org.apache.spark.mllib.linalg.Vector =
[0.003101589123746108,-0.015840247107918745,-0.007774317781555606,-0.003806113094931952,-0.00863613
7281604877,-0.005148049107586421,-0.001594902075648122,-0.001884423335325916,-0.008962148944901091,
0.0013868444641252052,-0.005665572822255591,-0.0020799698556476184,0.005031525183761886,-0.00453622
2522621226,-0.004337097687316992,-0.0020024187780513596,0.003675628031430061,0.004099360893456437,7
.055644300952508E-
4,-0.010469016691143758,0.0022251802674151363,-0.007341138259945909,-0.005298714983047668,-0.004316
530769012353,-0.0033523733864271655,-0.005486685811971845,-0.00685225363201854,-0.0051111552941361 8
3,-0.00513544332854106,0.001443188377580537,-0.008205120527875304,-0.0012501454147246576,0.00421169
6292260818,-0.006491113727497695,-0.00937...
```

注意：weight是一个spark的Vector，而不是DenseVector

```
scala> weight.size
res5: Int = 2000
```

此处β的估计值长度为2000

```
scala> loss
res7: Array[Double] = Array(0.6931471805599453, 2.699711306506718, 5.139530894501279,
4.229240188847298, 3.391763675455557, 1.329373330787051, 2.587739351139891, 2.7339045331757297,
1.7159432033134563, 1.1074258559954735, 1.6613880715959737)

scala> loss.size
res8: Int = 11
```

一共进行了10次的迭代，包含初值的拟合，因此共有11个loss，组成一个Array。

有限内存的BFGS算法

模型：研究Scala内置的**sample_libsvm_data**数据，并将数据分成训练集和测试集，利用训练集估计Logistic回归模型中的参数，然后在测试集上估计事件发生的概率。

- 首先导入相关的包

```
scala> import org.apache.spark.mllib.classification.LogisticRegressionModel
import org.apache.spark.mllib.classification.LogisticRegressionModel

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

scala> import org.apache.spark.mllib.optimization.{LBFGS, LogisticGradient, SquaredL2Updater}
import org.apache.spark.mllib.optimization.{LBFGS, LogisticGradient, SquaredL2Updater}

scala> import org.apache.spark.mllib.util.MLUtils       //目的：用于数据的导入与处理
import org.apache.spark.mllib.util.MLUtils
```

- 导入相关数据：

```
scala> val data = MLUtils.loadLibSVMFile(sc,"C:\\spark-3.3.0-bin-
hadoop3\\data\\mllib\\sample_libsvm_data.txt")
data: org.apache.spark.rdd.RDD[org.apache.spark.mllib.regression.LabeledPoint] =
MapPartitionsRDD[6] at map at MLUtils.scala:86
```

- 数据分析

```
scala> data.collect()
res0: Array[org.apache.spark.mllib.regression.LabeledPoint] = Array((0.0,(692,
[127,128,129,130,131,154,155,156,157,158,159,181,182,183,184,185,186,187,188,189,207,208,209,210,21
1,212,213,214,215,216,217,235,236,237,238,239,240,241,242,243,244,245,262,263,264,265,266,267,268,2
69,270,271,272,273,289,290,291,292,293,294,295,296,297,300,301,302,316,317,318,319,320,321,328,329,
330,343,344,345,346,347,348,349,356,357,358,371,372,373,374,384,385,386,399,400,401,412,413,414,426
,427,428,429,440,441,442,454,455,456,457,466,467,468,469,470,482,483,484,493,494,495,496,497,510,51
1,512,520,521,522,523,538,539,540,547,548,549,550,566,567,568,569,570,571,572,573,574,575,576,577,5
78,594,595,596,597,598,599,600,601,602,603,604,622,623,624,625,626,627,628,629,630,651,652,653,654,
655,656,657],[51.0,159.0,2...
```

分析：

1. `LabeledPoint`说明数据包含Label(因变量值)与Point(自变量值)
2. 最前面的0.0代表该样本的Label值
3. 692代表该样本解释变量的数目（原因：该数据本质上为稀疏向量）
4. 第一个数组代表非零值的位置
5. 第二个数组代表第一个数组对应位置的具体数值

```
scala> data.take(1)(0).label
res6: Double = 0.0
```

即第一个样本的响应变量值为0.0；

```
scala> val numFeatures = data.take(1)(0).features.size
numFeatures: Int = 692
```

即解释变量的数目为692：

先取出第一个样本：

```
scala> data.take(1)(0)
res2: org.apache.spark.mllib.regression.LabeledPoint = (0.0,(692,
[127,128,129,130,131,154,155,156,157,158,159,181,182,183,184,185,186,187,188,189,207,208,209,210,21
1,212,213,214,215,216,217,235,236,237,238,239,240,241,242,243,244,245,262,263,264,265,266,267,268,2
69,270,271,272,273,289,290,291,292,293,294,295,296,297,300,301,302,316,317,318,319,320,321,328,329,
330,343,344,345,346,347,348,349,356,357,358,371,372,373,374,384,385,386,399,400,401,412,413,414,426
,427,428,429,440,441,442,454,455,456,457,466,467,468,469,470,482,483,484,493,494,495,496,497,510,51
1,512,520,521,522,523,538,539,540,547,548,549,550,566,567,568,569,570,571,572,573,574,575,576,577,5
78,594,595,596,597,598,599,600,601,602,603,604,622,623,624,625,626,627,628,629,630,651,652,653,654,
655,656,657],[51.0,159.0,253.0,159.0,50...
```

再取出第一个样本的解释变量：

```
scala> data.take(1)(0).features
res3: org.apache.spark.mllib.linalg.Vector = (692,
[127,128,129,130,131,154,155,156,157,158,159,181,182,183,184,185,186,187,188,189,207,208,209,210,21
1,212,213,214,215,216,217,235,236,237,238,239,240,241,242,243,244,245,262,263,264,265,266,267,268,2
69,270,271,272,273,289,290,291,292,293,294,295,296,297,300,301,302,316,317,318,319,320,321,328,329,
330,343,344,345,346,347,348,349,356,357,358,371,372,373,374,384,385,386,399,400,401,412,413,414,426
,427,428,429,440,441,442,454,455,456,457,466,467,468,469,470,482,483,484,493,494,495,496,497,510,51
1,512,520,521,522,523,538,539,540,547,548,549,550,566,567,568,569,570,571,572,573,574,575,576,577,5
78,594,595,596,597,598,599,600,601,602,603,604,622,623,624,625,626,627,628,629,630,651,652,653,654,
655,656,657],[51.0,159.0,253.0,159.0,50.0,48.0,238.0,2...
```

因此可以得到解释变量的数目：

```
scala> data.take(1)(0).features.size
res4: Int = 692
```

注意：此处由于是稀疏向量，因此表示形式有所不同，但总数目仍为692。

- 进行训练集与测试集的划分
    - 利用`randomSplit`函数将数据集data按0.6-0.4的比例随机地分成两部分：

```
scala> val splits = data.randomSplit(Array(0.6, 0.4), seed=11L)
splits: Array[org.apache.spark.rdd.RDD[org.apache.spark.mllib.regression.LabeledPoint]] =
Array(MapPartitionsRDD[7] at randomSplit at <console>:27, MapPartitionsRDD[8] at randomSplit at
<console>:27)
```

- 提取测试集

```
scala> val test = splits(1)
test: org.apache.spark.rdd.RDD[org.apache.spark.mllib.regression.LabeledPoint] =
MapPartitionsRDD[8] at randomSplit at <console>:27

scala> test.collect().size
res7: Int = 39

scala> data.collect().size
res8: Int = 100
```

注意到测试集的比例接近40%

- 提取训练集

由于需要增加一个截距项，因此需要在各Vector向量的最后加一个元素1；同时要保证数据**仍然为LabeledPoint**的格式。

方法：利用`MLUtils.appendBias(Vector)`函数，在Vector向量的最后加一个元素1：

```
scala> val training = splits(0).map(x => (x.label,MLUtils.appendBias(x.features))).cache()
training: org.apache.spark.rdd.RDD[(Double, org.apache.spark.mllib.linalg.Vector)] =
MapPartitionsRDD[10] at map at <console>:27
```

分析训练集的数据：

```
scala> training.take(1)(0)
res11: (Double, org.apache.spark.mllib.linalg.Vector) = (0.0,(693,
[127,128,129,130,131,154,155,156,157,158,159,181,182,183,184,185,186,187,188,189,207,208,209,210,21
1,212,213,214,215,216,217,235,236,237,238,239,240,241,242,243,244,245,262,263,264,265,266,267,268,2
69,270,271,272,273,289,290,291,292,293,294,295,296,297,300,301,302,316,317,318,319,320,321,328,329,
330,343,344,345,346,347,348,349,356,357,358,371,372,373,374,384,385,386,399,400,401,412,413,414,426
,427,428,429,440,441,442,454,455,456,457,466,467,468,469,470,482,483,484,493,494,495,496,497,510,51
1,512,520,521,522,523,538,539,540,547,548,549,550,566,567,568,569,570,571,572,573,574,575,576,577,5
78,594,595,596,597,598,599,600,601,602,603,604,622,623,624,625,626,627,628,629,630,651,652,653,654,
655,656,657,692],[51.0,159.0,253.0,159...
```

注意：693说明成功地将“1”补充到了Vector的末尾；

- 定义算法所需要的变量值

```
scala> val numCorrections = 10          //定义修正个数
numCorrections: Int = 10

scala> val convergenceTol = 1e-4        //定义收敛判断条件的值
convergenceTol: Double = 1.0E-4

scala> val maxNumIterations = 20        //定义最大迭代次数
maxNumIterations: Int = 20

scala> val regParam = 0.1               //定义惩罚项系数
regParam: Double = 0.1

scala> val initialWeightsWithIntercept = Vectors.dense(new Array[Double](numFeatures+1))
initialWeightsWithIntercept: org.apache.spark.mllib.linalg.Vector =
[0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.
0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0
.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,
0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0
,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.
0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0
.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,
0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,...
//定义beta的初始值
```

- 进行有限BFGS算法

```
scala> val (weightsWithIntercept,loss) = LBFGS.runLBFGS(
     | training,
     | new LogisticGradient(),
     | new SquaredL2Updater(),
     | numCorrections,
     | convergenceTol,
     | maxNumIterations,
     | regParam,
     | initialWeightsWithIntercept)
weightsWithIntercept: org.apache.spark.mllib.linalg.Vector =
[0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.
0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0
.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,
0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,-1.625812669862
3678E-5,-7.170995168857331E-5,5.792527923605993E-7,1.0656731417382428E-4,1.9777915336420613E-
4,1.65701498337392E-5,-1.9090094533724542E-4,9.601736436052946E-
6,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,-2.047405388873459E-
5,-8.835834793715243E-6,1.9909912194126047E-4,3.0085646193983226E-4,1.030043478387045...
```

注意：这里`(weightsWithIntercept,loss)`的变量名不能改变，是固定的！

LBFGS算法函数的分析：

1. 函数的返回值有两个：β的估计值、各次迭代估计值拟合模型的loss（包含初值拟合的loss）
2. 第一个参数为数据集
3. 第二个参数为所用的梯度类型
4. 第三个参数为目标函数的类型
5. 第四个参数为修正个数
6. 第五个参数为收敛判断条件的值
7. 第六个参数为最大迭代次数
8. 第七个参数为目标函数惩罚项的系数
9. 第八个参数为β的迭代初值

- 根据估计出来的参数建立模型
    - 提取Array元素的方法：

      `A.slice(from-index, until-index)`： 提取Array A中从from-index开始，到until-index为止（不包含until-index）的元素；

      ```
      scala> weightsWithIntercept.toArray.slice(1,5)
      res17: Array[Double] = Array(0.0, 0.0, 0.0, 0.0)

      scala> weightsWithIntercept.toArray.slice(0,numFeatures).size       //提取了除截距项以外的估计值
      res21: Int = 692
      ```

    - 提取最后一个元素（即截距项的系数估计值）：

      ```
      scala> weightsWithIntercept(numFeatures)
      res23: Double = 3.855481364680967E-6
      ```

下面开始构建模型：

```
scala> val model = new LogisticRegressionModel(
     | Vectors.dense(weightsWithIntercept.toArray.slice(0,weightsWithIntercept.size - 1)),
     | weightsWithIntercept(weightsWithIntercept.size - 1))
model: org.apache.spark.mllib.classification.LogisticRegressionModel =
org.apache.spark.mllib.classification.LogisticRegressionModel: intercept = 3.855481364680967E-6,
numFeatures = 692, numClasses = 2, threshold = 0.5
```

分析：

1. `LogisticRegressionModel`函数需要两个参数：去除截距项的β估计值、截距项的估计值

2. 模型的返回结果：截距项为3.855481364680967E-6，解释变量的数目为692个，`numClasses = 2`代表响应变量的种类数为2个（此处即0和1），`threshold = 0.5`代表通过概率判断所属类的阈值，即概率值大于0.5则响应值为1，小于0.5则概率值为0；

- 清除上述模型的threshold值

```
scala> model.clearThreshold()
res24: model.type = org.apache.spark.mllib.classification.LogisticRegressionModel: intercept =
3.855481364680967E-6, numFeatures = 692, numClasses = 2, threshold = None
```

从而此时的阈值为None；

- 在测试集上计算各样本的概率值

```
scala> val ProbabilityAndLabels = test.map{point => val probability = model.predict(point.features)
     | (probability,point.label)
     | }
ProbabilityAndLabels: org.apache.spark.rdd.RDD[(Double, Double)] = MapPartitionsRDD[45] at map at
<console>:28
```

分析：

1. 利用`model.predict`函数对测试集数据进行概率预测
2. 返回值为（预测的概率值，真实的响应变量值）

- 预测数据的分析

```
scala> ProbabilityAndLabels.collect()
res26: Array[(Double, Double)] = Array((2.8524920092879822E-5,0.0), (0.9999999971488216,1.0),
(5.772614252612441E-7,0.0), (1.5947589454375812E-8,0.0), (2.738161786779754E-10,0.0),
(0.9999999891542356,1.0), (1.033896639683206E-5,0.0), (1.9073133561740993E-6,0.0),
(9.22616697551085E-9,0.0), (5.1386081747186324E-5,0.0), (0.9999999927889052,1.0),
(0.9999997269928688,1.0), (0.9999999997705833,1.0), (0.9999999998991687,1.0), (2.51830122800747E-
6,0.0), (0.9999990580781305,1.0), (0.9999999999930136,1.0), (0.9962526416288039,1.0),
(0.9999999924134505,1.0), (1.4512536629770197E-6,0.0), (0.9999999906406031,1.0),
(1.0180580897593368E-9,0.0), (0.9999999785636804,1.0), (0.9999999999991813,1.0),
(9.427474653897982E-10,0.0), (5.1940238929564844E-8,0.0), (0.9999999869198901,1.0),
(0.9999999786184832,1.0...
```

- 若以0.5为概率的阈值（即大于0.5则响应变量预测为1，否则响应变量预测为0），预测结果为：

```
scala> val error = test.map{point =>
     | val probability = model.predict(point.features)
     | var predictY = 0.0
     | if(probability > 0.5) predictY = 1.0 else predictY = 0.0
     | math.abs(point.label - predictY)}            //即每个元素为真实值与预测值差值的绝对值
error: org.apache.spark.rdd.RDD[Double] = MapPartitionsRDD[46] at map at <console>:28

scala> error.collect()
res29: Array[Double] = Array(0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
0.0, 0.0, 0.0, 0.0, 0.0)

scala> error.reduce((x,y) => x + y)
res30: Double = 0.0
```

注意到全部预测正确！

第十次课

- **附加题：**
  - 用MapReduce与DenseMatrix实现分块矩阵的运算；

自由自举法(Bootstrap)

- 考虑如下的线性回归模型：

$$y_i = x_i^T \beta_0 + \epsilon_i, \quad i = 1, \ldots, N$$

其中x和y分别是解释变量和响应变量，而ϵ是模型误差。

- 考虑以下的**自由自举法**的步骤：

1. 首先得到β_0的估计量$\hat{\beta}_0$，然后计算出残差项$\hat{\epsilon}_i = y_i - x_i^T \hat{\beta}_0$
2. 从均值为0，方差为1的标准正态分布中产生随机数$\omega_1, \cdots, \omega_N$
3. 产生伪响应变量观察值$y_i^* = x_i^T \hat{\beta}_0 + \omega_i \hat{\epsilon}_i, i = 1, \ldots, N$（即相当于对残差进行一个正态的扰动）
4. 对数据y_i^*, x_i再次估计参数，得到自举法估计量$\hat{\beta}^*$
5. 重复2-4步若干次，得到若干个自举法估计量，从而利用这些估计量来对原来估计量$\hat{\beta}_0$实现统计推断

- 首先导入相关的包

```
scala> import org.apache.spark.mllib.regression.LabeledPoint
import org.apache.spark.mllib.regression.LabeledPoint    //拟合回归模型，用的数据格式为(Label,Vectors)

scala> import org.apache.spark.mllib.optimization.{LBFGS, LeastSquaresGradient, SquaredL2Updater}
import org.apache.spark.mllib.optimization.{LBFGS, LeastSquaresGradient, SquaredL2Updater}
                                                         //在该线性模型中，响应变量为连续型，且利用岭回归模型
scala> import org.apache.spark.mllib.util.MLUtils        //读入数据
import org.apache.spark.mllib.util.MLUtils

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

scala> import breeze.linalg._
import breeze.linalg._

scala> import java.util.concurrent.ThreadLocalRandom
import java.util.concurrent.ThreadLocalRandom
```

- 读入数据

```
scala> val data = MLUtils.loadLibSVMFile(sc, "C:\\spark-3.3.0-bin-
hadoop3\\data\\mllib\\sample_linear_regression_data.txt")
data: org.apache.spark.rdd.RDD[org.apache.spark.mllib.regression.LabeledPoint] =
MapPartitionsRDD[6] at map at MLUtils.scala:86

scala> data.collect()
res0: Array[org.apache.spark.mllib.regression.LabeledPoint] = Array((-9.490009878824548,(10,
[0,1,2,3,4,5,6,7,8,9],
[0.4551273600657362,0.36644694351969087,-0.38256108933468047,-0.4458430198517267,0.3310979035891472
6,0.8067445293443565,-0.2624341731773887,-0.44850386111659524,-0.07269284838169332,0.56580355758007
15])), (0.2577820163584905,(10,[0,1,2,3,4,5,6,7,8,9],
[0.8386555657374337,-0.1270180511534269,0.499812362510895,-0.22686625128130267,-0.6452430441812433,
0.18869982177936828,-0.5804648622673358,0.651931743775642,-0.6555641246242951,0.17485476357259122])
), (-4.438869807456516,(10,[0,1,2,3,4,5,6,7,8,9],
[0.5025608135349202,0.14208069682973434,0.16004976900412138,0.505019897181302,-0.9371635223468384,-
0.2841601610457427,0.6355938616712786,-0.1646249064941625,0.9480713629917628,0.4268125...

scala> val numFeatures = data.take(1)(0).features.size
numFeatures: Int = 10
```

注意：读入的数据格式为LabeledPoint，响应变量为连续值，共10个解释变量；

- 在数据的末尾加一个截距项

```
scala> val training = data.map(x => (x.label, MLUtils.appendBias(x.features))).cache()
training: org.apache.spark.rdd.RDD[(Double, org.apache.spark.mllib.linalg.Vector)] =
MapPartitionsRDD[7] at map at <console>:37

scala> training.collect()
res6: Array[(Double, org.apache.spark.mllib.linalg.Vector)] = Array((-9.490009878824548,(11,
[0,1,2,3,4,5,6,7,8,9,10],
[0.4551273600657362,0.36644694351969087,-0.38256108933468047,-0.4458430198517267,0.3310979035891472
6,0.8067445293443565,-0.2624341731773887,-0.44850386111659524,-0.07269284838169332,0.56580355758007
15,1.0])), (0.2577820163584905,(11,[0,1,2,3,4,5,6,7,8,9,10],
[0.8386555657374337,-0.1270180511534269,0.499812362510895,-0.22686625128130267,-0.6452430441812433,
0.18869982177936828,-0.5804648622673358,0.651931743775642,-0.6555641246242951,0.17485476357259122,1
.0])), (-4.438869807456516,(11,[0,1,2,3,4,5,6,7,8,9,10],
[0.5025608135349202,0.14208069682973434,0.16004976900412138,0.505019897181302,-0.9371635223468384,-
0.2841601610457427,0.6355938616712786,-0.1646249064941625,0.948071362...
```

- 进行LBFGS算法基本参数的设定

```
scala> val numCorrections = 10          //修正数
numCorrections: Int = 10

scala> val convergenceTol = 1e-4        //收敛条件的阈值
convergenceTol: Double = 1.0E-4

scala> val maxNumIterations = 20        //最大迭代次数
maxNumIterations: Int = 20

scala> val regParam = 1.0               //惩罚项系数
regParam: Double = 1.0

scala> val initialWeightsWithIntercept = Vectors.dense(new Array[Double](numFeatures+1))
initialWeightsWithIntercept: org.apache.spark.mllib.linalg.Vector =
[0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0]       //beta的迭代初值
```

- 根据Bootstrap的方法，首先利用LBFGS算法计算β_0的估计值

```
scala> val (weightsWithIntercept0, loss) = LBFGS.runLBFGS(
     | training,
     | new LeastSquaresGradient(),
     | new SquaredL2Updater(),
     | numCorrections,
     | convergenceTol,
     | maxNumIterations,
     | regParam,
     | initialWeightsWithIntercept)
weightsWithIntercept0: org.apache.spark.mllib.linalg.Vector =
[0.022917619144397203,0.16223861427526343,-0.1839044620018147,0.5454078186017856,0.1094197493846255
8,0.2811178300566573,-0.09271590011136743,-0.12330860139641511,-0.15505995633811184,0.1615654581472
7047,0.11614360033051634]
loss: Array[Double] = Array(53.156115128976595, 52.86141046393933, 52.801233318674036,
52.80046016202687, 52.8004497573181)

scala> weightsWithIntercept0                          //beta的估计值
res7: org.apache.spark.mllib.linalg.Vector =
[0.022917619144397203,0.16223861427526343,-0.1839044620018147,0.5454078186017856,0.1094197493846255
8,0.2811178300566573,-0.09271590011136743,-0.12330860139641511,-0.15505995633811184,0.1615654581472
7047,0.11614360033051634]

scala> weightsWithIntercept0.size
res8: Int = 11
```

- 分别提取β的截距项与非截距项

```
scala> val coefficients = DenseVector(weightsWithIntercept0.toArray.slice(0,
weightsWithIntercept0.size - 1))          //提取非截距项：由于weightsWithIntercept0为Spark的
Vector，因此需要转换为Array用slice函数
coefficients: breeze.linalg.DenseVector[Double] = DenseVector(0.022917619144397203,
0.16223861427526343, -0.1839044620018147, 0.5454078186017856, 0.10941974938462558,
0.2811178300566573, -0.09271590011136743, -0.12330860139641511, -0.15505995633811184,
0.16156545814727047)

scala> val Intercept = weightsWithIntercept0(weightsWithIntercept0.size - 1)
Intercept: Double = 0.11614360033051634     //提取截距项系数
```

- 计算响应变量的估计值（即$\hat{y}_i$）与残差
  - 回顾：可以用Array生成DenseVector：

```
scala> DenseVector(Array(1.0,2.0))
res11: breeze.linalg.DenseVector[Double] = DenseVector(1.0, 2.0)
```

  - 计算$\hat{y}_i$与残差

```
scala> val lines = data.map(line => {
     | val fitted = DenseVector(line.features.toArray).t*coefficients + Intercept
     | val residual = line.label - fitted
     | LabeledPoint(residual,line.features)
     | })
lines: org.apache.spark.rdd.RDD[org.apache.spark.mllib.regression.LabeledPoint] =
MapPartitionsRDD[15] at map at <console>:39
```

注意：现在的数据格式为LabeledPoint，其中Label值为残差，Features为原来的解释变量值

```
scala> lines.take(1)(0)
res16: org.apache.spark.mllib.regression.LabeledPoint = (-9.948565367849152,(10,
[0,1,2,3,4,5,6,7,8,9],
[0.4551273600657362,0.36644694351969087,-0.38256108933468047,-0.4458430198517267,0.331097903589
14726,0.8067445293443565,-0.2624341731773887,-0.44850386111659524,-0.07269284838169332,0.565803
5575800715]))
```

- 根据Bootstrap的方法，计算y_i^*

```
scala> val transit = lines.map(line => {
     | val r  = ThreadLocalRandom.current
     | val fitted = DenseVector(line.features.toArray).t*coefficients + Intercept
     | LabeledPoint(fitted + r.nextGaussian * line.label, line.features)
     | })
transit: org.apache.spark.rdd.RDD[org.apache.spark.mllib.regression.LabeledPoint] =
MapPartitionsRDD[16] at map at <console>:39
```

注意：

1. 此时做Map的数据是上一步得到的lines，其首位为残差
2. 最后我们仍然保存为LabeledPoint格式

- 计算β_0^*，与上述计算β_0的方法相同
  - 首先在数据集的末尾添加截距项

```
scala> val training = transit.map(x => (x.label, MLUtils.appendBias(x.features))).cache()
training: org.apache.spark.rdd.RDD[(Double, org.apache.spark.mllib.linalg.Vector)] =
MapPartitionsRDD[17] at map at <console>:37
```

  - 利用LBFGS算法估计β_0^*

```
scala> val (weightsWithIntercept, loss) = LBFGS.runLBFGS(
     | training,
     | new LeastSquaresGradient(),
     | new SquaredL2Updater(),
     | numCorrections,
     | convergenceTol,
     | maxNumIterations,
     | regParam,
     | initialWeightsWithIntercept)
weightsWithIntercept: org.apache.spark.mllib.linalg.Vector =
[-0.09798941791744982,-0.04060246109617967,-0.15394856931545242,-0.047303231130858416,0.0192578
79510612606,0.4343199246286681,0.4442458820833316,0.1463362913371062,0.03847674483054382,-0.040
019669119610746,-0.16495442262968002]
loss: Array[Double] = Array(47.841251055356764, 47.59050410167165, 47.50631404345194,
47.50589776762564)
```

- 分析两次估计值的差异

```
scala> weightsWithIntercept
res18: org.apache.spark.mllib.linalg.Vector =
[-0.09798941791744982,-0.04060246109617967,-0.15394856931545242,-0.047303231130858416,0.01925787951
0612606,0.4343199246286681,0.4442458820833316,0.1463362913371062,0.03847674483054382,-0.04001966911
9610746,-0.16495442262968002]

scala> coefficients
res20: breeze.linalg.DenseVector[Double] = DenseVector(0.022917619144397203, 0.16223861427526343,
-0.1839044620018147, 0.5454078186017856, 0.10941974938462558, 0.2811178300566573,
-0.09271590011136743, -0.12330860139641511, -0.15505995633811184, 0.16156545814727047)

scala> Intercept
res21: Double = 0.11614360033051634
```

子集合自举法(Bag of Little Bootstraps, BLB)

- 子集合自举法：

1. 首先，我们从原来的数据中无放回的抽取K个互不相交的子集，每个子集的样本大小为b;
2. 其次，我们在每个子集上实现自举法，并计算我们感兴趣的统计量的值，记为$\hat{\xi}_k^*, k = 1, \dots, K$
3. 最后，将K个子集上计算出来的统计量加以平均作为BLB估计量

- **Key-Value pairs: (key, value)** (可以翻译为键-值对) 的相关包与函数
    - 使用Key-Value pairs需要导入包 `org.apache.spark.rdd.PairRDDFunctions` 与 `org.apache.spark.HashPartitioner`
    - `mapValues(s => Func(s))`：其中s为一个Key-Value pairs，只是在普通的Map函数基础上，新增了“以Key为类别进行操作”，类似于R中的groupby函数；
    - `reduceByKey((x,y) => Func(x,y))`：与上面的 `mapValue` 函数类似，只是在普通的Reduce函数基础上，以Key为类别进行操作，例如：取键值对为 `Array((1,1),(1,2),(1,3),(2,4),(2,5),(2,6))` (RDD格式)，Func取 `x+y`，则结果为 `Array((1,6),(2,15))`；
    - `sampleByKey(withReplacement, fraction)`：指对键值对进行有放回的抽样，其中 `fraction` 为如下的类似形式：
        - `fraction = List((1, 0.4), (2, 0.3)).toMap`：此指对key = 1的键值对抽取比例为0.4，对key = 2的键值对抽取比例为0.3，且各次抽样单独生成一个均匀分布随机数，故实际抽样数目与总数目之比可能与抽样比例不等；
    - `sampleByKeyExact(withReplacement, fraction)`：使得用户可以准确的从子集k中抽出$f_k n_k$个数据，其中f_k是用户需要的比例大小，n_k是子集k的样本量
    - `KVP1.join(KVP2)`：指键值对KVP1与键值对KVP2按key为连接变量进行连接，例如

      KVP1：

Key	Value
1	X1
2	X2

KVP2：

Key	Value
1	Y1
2	Y2

则`KVP1.join(KVP2)`返回值为：

Key	Value
1	(X1,Y1)
2	(X2,Y2)

  - `KVP.values`：将Key-Value pairs KVP里面的所有value都提取出来，组成一个新的Array，例如将`Array((1,1),(1,2),(1,3),(2,4),(2,5),(2,6))`(RDD格式)返回为`(1,2,3,4,5,6)`

- 模型

  从标准正态分布产生两百万个随机数，然后从中取样约8000个数据，用于得到该分布二阶中心矩的自举估计；

- 首先导入相关的包

```
scala> import scala.collection.JavaConverters._
import scala.collection.JavaConverters._

scala> import scala.util.Random
import scala.util.Random

scala> import org.apache.spark.rdd.PairRDDFunctions
import org.apache.spark.rdd.PairRDDFunctions

scala> import org.apache.spark.HashPartitioner
import org.apache.spark.HashPartitioner
```

- 定义模型的相关参数

```
scala> val n = 2000000          //表示随机数的数目
n: Int = 2000000
```

- 生成2000000个随机数，并添加键

```
scala> val points = sc.parallelize(0 until n).map{iter =>
     | val random = new Random()
     | val u = random.nextDouble()
     | val mykey = if(u < 0.5) 1 else 2          //先生成键
     | (mykey,random.nextGaussian())             //以(键，值)的tuple形式存储
     | }.partitionBy(new HashPartitioner(2)).persist()  //目的：利用哈希余数法进行节点分配（非重点内容）
points: org.apache.spark.rdd.RDD[(Int, Double)] = ShuffledRDD[24] at partitionBy at <console>:51

scala> points.collect()
res24: Array[(Int, Double)] = Array((2,1.317409979326181), (2,0.7519570647579242),
(2,0.17774860740532372), (2,-0.6557549298811552), (2,-1.5044517860311668),
(2,-0.39283923722532643), (2,1.4432420357658895), (2,-1.555360321368837), (2,1.164147383177984),
(2,-0.007553985527088342), (2,-0.17887308225416387), (2,-0.23897910085436203),
(2,0.6124628996341784), (2,0.7190414267248113), (2,1.5282861261961012), (2,-0.7362958942928843),
(2,-0.8914867786354898), (2,-0.550281275451298), (2,-0.4153911229538217), (2,0.15933784211268467),
(2,0.35896283976455307), (2,0.46231243978906594), (2,-0.3172357210636544), (2,-0.7404207663372514),
(2,1.4496602539532824), (2,-0.04685942423783813), (2,-0.6680281442842231),
(2,-0.13069274837846534), (2,-1.1046673760367947), (2,-0.7439197610329105), (2,-0.5711271090...
```

- 定义节点数与抽样比例

```
scala> val K = points.partitions.size       //partitions.size返回节点数目
K: Int = 2

scala> val SampleFractions = List((1, 0.004),(2, 0.004)).toMap
SampleFractions: scala.collection.immutable.Map[Int,Double] = Map(1 -> 0.004, 2 -> 0.004)
```

- 分别对具有相同Key值的数据进行分层再取样

```
scala> val SampledPoints = points.sampleByKey(false,SampleFractions)
SampledPoints: org.apache.spark.rdd.RDD[(Int, Double)] = MapPartitionsRDD[25] at sampleByKey at
<console>:46

scala> SampledPoints.collect().size
res26: Int = 7919
```

注意：`sampledByKey`的第一个参数为`withReplecement`，此处为不放回抽样，因此为false

- 分别计算Key为1、2时的样本量

```
scala> val N = SampledPoints.mapValues(x => 1).reduceByKey((x,y) => x + y)
N: org.apache.spark.rdd.RDD[(Int, Int)] = MapPartitionsRDD[27] at reduceByKey at <console>:45

scala> N.collect()
res31: Array[(Int, Int)] = Array((2,4023), (1,3896))
```

- 针对取样之后的子集分别进行自举法并计算中心二阶矩
    - 首先进行抽样

    ```
    scala> val BootstrapFractions = List((1,1.0),(2,1.0)).toMap          //定义抽样的比例
    BootstrapFractions: scala.collection.immutable.Map[Int,Double] = Map(1 -> 1.0, 2 -> 1.0)

    scala> val BootstrappedPoints = SampledPoints.sampleByKeyExact(true,BootstrapFractions)
    BootstrappedPoints: org.apache.spark.rdd.RDD[(Int, Double)] = MapPartitionsRDD[31] at
    sampleByKeyExact at <console>:46              //进行有放回的抽样

    scala> BootstrappedPoints.collect().size
    res32: Int = 7919            //说明确实是100%的重抽样
    ```

    - 由于需要计算各个子集的二阶中心矩，因此首先需要各样本总值

```
scala> val EstBootSum = BootstrappedPoints.reduceByKey((x,y) => x + y)
EstBootSum: org.apache.spark.rdd.RDD[(Int, Double)] = MapPartitionsRDD[32] at reduceByKey at
<console>:45

scala> EstBootSum.collect()
res33: Array[(Int, Double)] = Array((2,-45.90099637930528), (1,95.47773992578918))
```

- 再对总值与子集样本数进行连接

```
scala> EstBootSum.join(N).collect()
res35: Array[(Int, (Double, Int))] = Array((2,(-45.90099637930528,4023)), (1,
(95.47773992578918,3896)))
```

- 从而可以利用 `mapValues` 函数计算各样本子集的均值

```
scala> val EstBootMu = EstBootSum.join(N).mapValues(x => x._1/x._2)
EstBootMu: org.apache.spark.rdd.RDD[(Int, Double)] = MapPartitionsRDD[42] at mapValues at
<console>:46

scala> EstBootMu.collect()
res36: Array[(Int, Double)] = Array((2,-0.011409643643874045), (1,0.024506606757132746))
```

- 为了计算各个子集的中心二阶矩，不妨将各子集中样本的值与该子集的均值进行连接

```
scala> val UpdatedPoints = BootstrappedPoints.join(EstBootMu)
UpdatedPoints: org.apache.spark.rdd.RDD[(Int, (Double, Double))] = MapPartitionsRDD[48] at join
at <console>:46

scala> UpdatedPoints.take(3)
res39: Array[(Int, (Double, Double))] = Array((2,(1.3404020563827164,-0.011409643643874045)),
(2,(1.3404020563827164,-0.011409643643874045)), (2,
(-0.23910078795637518,-0.011409643643874045)))
```

- 计算各个子集的二阶中心矩

```
scala> val Est2thMomSum = UpdatedPoints.mapValues(x => (x._1 - x._2)*(x._1 -
x._2)).reduceByKey((x,y) => x + y)
Est2thMomSum: org.apache.spark.rdd.RDD[(Int, Double)] = MapPartitionsRDD[50] at reduceByKey at
<console>:45

scala> Est2thMomSum.collect()
res40: Array[(Int, Double)] = Array((2,3940.852922430612), (1,3868.493597745298))

scala> val Est2thMom = Est2thMomSum.join(N).mapValues(x => x._1/x._2)
Est2thMom: org.apache.spark.rdd.RDD[(Int, Double)] = MapPartitionsRDD[54] at mapValues at
<console>:46

scala> Est2thMom.collect()
res41: Array[(Int, Double)] = Array((2,0.9795806419166324), (1,0.9929398351502304))
```

- 由子集合自举法，最后利用将各个子集估计量的平均
  - 方法一：直接利用Reduce函数

```
scala> Est2thMom.reduce((x,y) => ((x._1 + y._1),(x._2 + y._2)))._2/K
res44: Double = 0.9862602385334314
```

注意：**reduce函数得到的数据类型应保持与原来相同！**

  - 方法二：利用 `KVP.values` 函数

```
scala> val MetaEst2thMom = Est2thMom.values.reduce((x,y) => x + y)/K
MetaEst2thMom: Double = 0.9862602385334314
```

HW5

1. **考虑如下的线性回归模型:**

$$y_i = x_i^T\beta_0 + \epsilon_i, \quad i = 1, \ldots, N$$

其中x和y分别是解释变量和响应变量，而ϵ是模型误差。

考虑以下的自由自举法的步骤:

***Step1*:** 首先得到β_0的估计量$\hat{\beta}_0$，然后计算出残差项$\hat{\epsilon}_i = y_i - x_i^T\hat{\beta}_0$

***Step2*:** 从均值为0，方差为1的标准正态分布中产生随机数$\omega_1, \cdots, \omega_N$

***Step3*:** 产生伪响应变量观察值$y_i^* = x_i^T\hat{\beta}_0 + \omega_i\hat{\epsilon}_i, i = 1, \ldots, N$

***Step4*:** 对数据y_i^*, x_i再次估计参数，得到自举法估计量$\hat{\beta}^*$

***Step5*:** 重复2-4步200次，得到200个自举法估计量$\hat{\beta}^*$

***Step6*:** 求第5步中的自举法估计量的均值和标准差.

作业：请利用课堂上没有写完的代码完成上述步骤中的第5步和第6步。

- 先导入相关的包

```
import org.apache.spark.mllib.regression.LabeledPoint
import org.apache.spark.mllib.optimization.{LBFGS, LeastSquaresGradient, SquaredL2Updater}
import org.apache.spark.mllib.util.MLUtils
import org.apache.spark.mllib.linalg.Vectors
import breeze.linalg._
import java.util.concurrent.ThreadLocalRandom
```

- 读入数据

```
val data = MLUtils.loadLibSVMFile(sc, "C:\\spark-3.3.0-bin-
hadoop3\\data\\mllib\\sample_linear_regression_data.txt")
data.collect()
```

- 定义所需变量

```
val numFeatures = data.take(1)(0).features.size
val training = data.map(x => (x.label, MLUtils.appendBias(x.features))).cache()
training.collect()
```

- 进行LBFGS算法基本参数的设定

```
val numCorrections = 10
val convergenceTol = 1e-4
val maxNumIterations = 20
val regParam = 1.0
val initialWeightsWithIntercept = Vectors.dense(new Array[Double](numFeatures+1))
```

- 根据Bootstrap的方法，首先利用LBFGS算法计算β_0的估计值

```
val (weightsWithIntercept0, loss) = LBFGS.runLBFGS(
    training,
    new LeastSquaresGradient(),
    new SquaredL2Updater(),
    numCorrections,
    convergenceTol,
    maxNumIterations,
    regParam,
    initialWeightsWithIntercept)
```

- 分别提取β的截距项与非截距项

```
val coefficients = DenseVector(weightsWithIntercept0.toArray.slice(0, weightsWithIntercept0.size -
1))
val Intercept = weightsWithIntercept0(weightsWithIntercept0.size - 1)
```

- 循环

```
val size = weightsWithIntercept0.size
var beta_all = DenseVector((0 until 200).map(x => DenseVector.zeros[Double](size)).toArray)
val lines = data.map(line => {
    val fitted = DenseVector(line.features.toArray).t*coefficients + Intercept
    val residual = line.label - fitted
    LabeledPoint(residual,line.features)
})
for(i <- 0 until 200){
    val transit = lines.map(line => {
        val r  = ThreadLocalRandom.current
        val fitted = DenseVector(line.features.toArray).t*coefficients + Intercept
        LabeledPoint(fitted + r.nextGaussian * line.label, line.features)
    })
    val training = transit.map(x => (x.label, MLUtils.appendBias(x.features))).cache()
    val (weightsWithIntercept, loss) = LBFGS.runLBFGS(
        training,
        new LeastSquaresGradient(),
        new SquaredL2Updater(),
        numCorrections,
        convergenceTol,
        maxNumIterations,
        regParam,
        initialWeightsWithIntercept)
    beta_all(i) = DenseVector(weightsWithIntercept.toArray)
}
```

- 自由自举法估计量及其统计特征

```
scala> beta_all
res3: breeze.linalg.DenseVector[breeze.linalg.DenseVector[Double]] =
DenseVector(DenseVector(-0.07847217404597306, -0.061199651867988085, 0.2516086874675261,
0.2696764752897606, -0.1681855485035505, -0.24891562105808804, -0.06581205733193848,
-0.4335280759460743, 0.12065120929769006, 0.5426335125736291, -0.35719767103020733),
DenseVector(-0.027659111674427802, 0.11396736439404112, 0.20623910893624545, -0.03579264697682845,
0.011621415778861076, -0.12214450418255351, -0.19381199873002894, -0.012479093293079082,
0.009936842129639443, -0.16227021971604733, -0.03468632896860682),
DenseVector(-0.12090219876978689, 0.1397338436484088, 0.3301060670035595, -0.04183086572065614,
0.08778507329279764, 0.1447771708568299, -0.18132678714944134, -0.3782439165669319,
-0.09795856235598317, -0.174411370...
```

```
scala> val beta_mean = beta_all.reduce((x,y) => x + y)/200.0
beta_mean: breeze.linalg.DenseVector[Double] = DenseVector(0.016412326063708526,
0.03655494730961871, -0.024577395562262218, 0.12168211942422756, 0.01926933973318555,
0.051306235466737286, -0.044433992053851255, -0.019928113792221702, -0.05083467633123586,
0.02092604054149301, 0.06821308441329413)

scala> val beta_single_std = sqrt(beta_all.map(x => (x-beta_mean)*(x-beta_mean)).reduce((x,y) => x
+ y)/199.0)
beta_single_std: breeze.linalg.DenseVector[Double] = DenseVector(0.2053284390927121,
0.19714228002042417, 0.19267133932708883, 0.19898369568732505, 0.2132191285080567,
0.2094902367527184, 0.17795061333359097, 0.18960931277013449, 0.20259739797136553,
0.21021050941026345, 0.20694659601862775)
```

归纳：Scala常用package及其函数

- 自带Array函数：
  - 声明一个Array：new Array[Double](num)
  - Array的最值、求和：array.min，array.max，array.sum
  - Array的由小到大排序：array.sorted
  - Array的逆序：array.reverse
  - Array的由大到小排序：array.sorted.reverse
  - 两个Array的拼接：array1.union(array2)
  - 舍弃Array的前n个数：array.drop(n)
  - 舍弃Array的后n个数：array.dropRight(n)
  - 生成由a1个数构成的Array，进而a2个Array构成的Array...：Array.ofDim[Double](an,...,a2,a1)
  - 生成由n个a组成的Array：Array.fill(n)(a)
  - 生成由n个随机数组成的Array：Array.fill(n)(r.nextDouble)，其中r.nextDouble为生成随机数的java/scala函数
  - Array的切片：array.slice(from,until)
- 保留小数的指定位数：
  - a.map("%.4f" format _)
  - "%.4f".format(a)
- 数学运算：scala.math._
  - 函数：pow(底数，指数)，min，max
- 向量、矩阵运算：breeze.linalg._
  - 创建长度为n的双精度零向量：DenseVector.zeros[Double](n)
  - 创建n*m的整型零矩阵：DenseMatrix.zeros[Double](n)
  - 创建任意常数向量：DenseVector.fill(n,a)
  - 创建m行n列的矩阵：new DenseMatrix(rows = m, cols = n, Array) 注意：Array按列填充
  - 固定步长向量：DenseVector.range**(D)**(start,stop,step)
  - 固定长度向量：linspace(start,stop,size)
  - 创建单位阵：DenseMatrix[double](n)
  - 生成对角阵、提取对角元：diag(DV/A)
  - 矩阵横向合并：**DenseMatrix**.horzcat(A,B)
  - 矩阵纵向合并：**DenseMatrix**.vertcat(A,B)
  - 索引变换函数：
    - 向量：DenseVector.tabulate(n){i => f(i)} 注意：都是从0开始！
    - 矩阵：DenseMatrix.tabulate**(n,m)**{**case**(i,j) => f(i,j)} （实则case加否均可）
  - 矩阵按元素相乘、除：a*:*b，/
  - 向量、矩阵的求和：
    - 整体求和：sum(A)
    - 对矩阵每一行求和：sum(A(*,::))
    - 对矩阵每一列求和：sum(A(::,*))
  - 矩阵的行、列数：A.rows，A.cols

- 矩阵的每一行、列减去固定的向量：
  - 列：A(::,*) - u
  - 行：A(*,::) - v （**不用**加转置，自动匹配）
- 矩阵的每一行、列乘除固定的向量：
  - 列：A(::,*) * u
  - 行：A(*,::) * v （也**不用**加转置，自动匹配）
- 矩阵的行列式、逆，向量的二范数：det(A),inv(A),norm(a)
- 矩阵的特征值、特征向量：eig(A).eigenvalues,eig(A).eigenvectors
- 矩阵的奇异值分解：svd.SVD(u,d,v) = svd(A)
- 矩阵的秩：rank(A)

- 数学函数包：breeze.numerics._
  - 函数：sin,cos,tan,log,exp,log10,sqrt,pow
- 统计分布包：breeze.stat**s**.distributions._
  - 创建分布：
    - Poisson分布：new Poisson(parameter)
    - Beta分布：new Beta(para_1,para_2)
    - 正态分布：new Gaussian(mean,std)
    - Dirichlet分布：new Dirichlet(DenseVector(para_1,...,para_n))
      - 不定参数数目的分布需要以DenseVector的形式传入参数
    - 多元正态分布：new MultivariateGaussian(DenseVector,DenseMatrix)
  - 计算总体均值、方差：distribution.mean，distribution.variance
  - 抽取特定分布的样本：distribution.sample(num)
  - 计算特定点的概率：distribution.probability**O**f(x) （对于向量则使用map）
  - 计算特定点的PDF，CDF值：distribution.pdf(x)，distribution.cdf(x)
  - 计算根据分布所抽取样本的均值与方差：**breeze.stats.**mean**A**nd**V**ariance(sample) （必须是Double型，否则需要转换）
    - breeze.stats原因：没有import该package
    - 转换方式：
      - for循环yield改变分布：for(x <- distribution) yield x.toDouble
      - map改变样本：sample.map(_.toDouble)
  - 指数分布族参数的MLE
    - 创建样本：
      - 构造所需分布：val MyDir = new Dirichlet(DenseVector(3.0,2.0,6.0))
      - 获取样本数据：val data = MyDir.sample(100)
      - 转换数据结构：val data0 = data.toIndexedSeq
    - MLE：
      - 构建指数族：val ExpFam = new Dirichlet.ExpFam(DenseVector.zeros[Double](3))
      - 获取充分统计量：val SuffStat = data0.foldLeft(ExpFam.emptySufficientStatistic){(x,y) => x + ExpFam.sufficientStatisticFor(y)}
      - 根据充分统计量得到MLE：ExpFam.mle(SuffStat)
  - MapReduce方法求解MLE：
    - 获取充分统计量：
      - 分布式存储：val rdd = sc.parallelize(data0)
      - 计算充分统计量：val result = rdd.map(s => log(s)).reduce((x,y) => x + y)
      - 获取MLE函数所需的充分统计量：val SuffStat = ExpFam.**S**ufficientStatistic(100,result)
    - MLE：
      - 根据充分统计量得到MLE：ExpFam.mle(SuffStat)
- 四个随机数包：
  - 利用概率密度函数抽样生成随机数
  - java.util.concurrent.ThreadLocalRandom

        - 定义随机数函数：val r = ThreadLocalRandom.current
        - 生成(0,1)均匀分布随机数：r.nextDouble
        - 生成标准正态分布随机数：r.nextGaussian
        - 生成num个随机数：(1 to num).map(s => r.nextGaussian)
    - 机器学习包：org.apache.spark.**SparkContext**，org.apache.spark.mllib.random.RandomRDDs**._**
        - 生成RDD的随机数向量：
            - 100个Poisson(a)存储在4个节点上：poissonRDD(**sc**,a,100,4)
            - (0,1)均匀分布：uniformRDD(sc,100,4)
            - 指数分布Exp(1/a)：exponentialRDD(sc,a,100,4)
            - 标准正态分布：normalRDD(sc,100,4)
            - 对数正态分布：logNormalRDD(sc,mean,std,100,4)
            - 多元(0,1)均匀分布：uniformVectorRDD(sc,100,dim,4)
    - 效果类似于第二个java包的scala.util.Random
        - 定义随机数函数：val r = **new** Random()
        - 生成(0,1)均匀分布的随机数：r.nextDouble
        - 生成标准正态分布的随机数：r.nextGaussian
- 文件的输出包：java.io._
    - 打开/创建文件：val pw = new PrintWriter(Direction / new file(filename))
    - 写入文件：pw.write(value) 注意：只能写入Int/String，不能写入Array！
    - 关闭文件：pw.close
    - 读入文本文件：sc.textFile(direction) 注意：读入的即为RDD格式
- 线性运算类包：org.apache.spark.mllib.linalg._
    - 导入矩阵运算函数：org.apache.spark.mllib.linalg.{Vectos,Matrix => sparkMatrix,DenseMatrix => sparkDenseMatrix}
    - 创建一个m行n列的矩阵，array为元素且按列填充：new sparkDenseMatrix(m,n,array)
    - 利用**Double型**array创建一个spark DenseVector：Vectors.dense(array)
- RDD的线性运算类：(只能用Double数据类型)
    - Local：
        - 生成Vector：Vectors.dense(Array)
        - 生成sparkDenseMatrix：new sparkDenseMatrix(m,n,Array)   注意：Array按列填充
    - Indexed：
        - IndexedRow(Index,Vectors)   注意：Index必须为Long型数据
        - 生成IndexedRowMatrix：本质将Array各元素map为IndexedRow
            - 运算：
                - A.multiply(spark DenseMatrix)
                - A.toBlockMatrix(rowsPerBlock, colsPerBlock)
    - BlockMatrix：
        - 基本运算:
            - 四则运算：A.add/multiply/subtract(B)， A.transpose
            - 转换为上述两种类型：
                - A.toLocalMatrix
                - A.toIndexedRowMatrix
    - 转换步骤：
        - Local Vector →IndexedRow→IndexedRowMatrix→BlockMatrix→（显示、转换）Local Matrix→DenseMatrix
- 优化方法包：
    - 需添加：scala.collection.JavaConverters._
    - 回归模型包：org.apache.spark.mllib.regression._
    - 迭代数据格式：org.apache.spark.mllib.linalg.Vectors
    - 优化算法包：org.apache.spark.mllib.optimization.{GradientDescent, LogisticGradient/LeastSquare**s**Gradient, SimpleUpdater/SquaredL2Updater,L1Updater}
    - 常规步骤：

        - 定义梯度、更新函数：val gradient = **new** LogisticGradient(), val updater = **new** SquaredL2Updater
        - 定义所需的步长、迭代次数、惩罚项系数、训练集比例
    - 逻辑回归模型随机梯度下降算法：
        - 函数：GradientDescent.runMiniBatch**SGD**(数据，gradient，updater，步长，迭代次数，惩罚项系数，训练集比例，初值)
        - 结果：(weight, loss)
    - LBFGS：
        - 步骤梗概：
            - 数据导入与预处理：定义解释变量数目，训练集与测试集的划分，训练集添加截距项；
            - 定义算法初值：修正数目，收敛条件，迭代次数，惩罚项系数，初值；
            - 通过LBFGS算法求解回归系数；
            - 构建逻辑回归模型；
            - 清除模型阈值，计算测试集各点（概率值，拟合值）；
            - 以0.5为模型阈值，评价模型预测结果。
        - 相关函数：
            - packages：
                - 导入数据：org.apache.spark.mllib.util.MLUtils
                - 构建模型：org.apache.spark.mllib.classification.LogisticRegressionModel
                - LBFGS算法：org.apache.spark.mllib.optimization.{LBFGS,LogisticGradient,SquaredL2Updater}
                - 算法所需数据：org.apache.spark.mllib.linalg.Vectors
            - 数据导入与预处理：
                - 定义解释变量数目：val numFeatures = data.take(1)(0).features.size
                - 划分数据集：val splits = data.randomSplit(Array(0.6,0.4),seed = 11L)

                  val test = splits(1)

                  val training = splits(0).map(s => (s.label,MLUtils.appendBias(s.features)))
                - 定义初值：numCorrections = 10, convergenceTol = 1e-4, numIterations = 20,regParam = 0.1, initial**Weights**WithIntercept = Vectors.dense(new Array[Double](numFeatures + 1))
                - LBFGS算法：

```
val (weightsWithIntercept,loss) = LBFGS.runLBFGS(
  training,
  new LogisticGradient(),
  new SquaredL2Updater(),
  numCorrections,
  convergenceTol,
  numIterations,
  regParam,
  initialWeightsWithIntercept)
```

                - 构建Logistic回归模型：val model = new LogisticRegressionModel(Vectors.dense(weightsWithIntercept.toArray.slice(0,numFeatures)), weightsWithIntercept(numFeatures))
                - 定义阈值：model.clearThreshold()，计算各点概率值：

```
val Probability = test.map(s => {
    val pro = model.predict(s.features)
    (s.label,pro)})
```

                - 评价模型预测结果：

```
val Pro = test.map(s => {var pre = 0.0
                         val pro = model.predict(s.features)
                         if(pro > 0.5) pre = 1.0
                         math.abs(pre - s.label)})
Pro.reduce((x,y) => x + y)
```

- 自由自举法

    - 线性模型回归参数：
        - LBFGS，LeastSquaresGradient，SquaredL2Updater
        - 所用变量为LabeledPoint
    - 思路梗概：
        - 对数据进行LBFGS算法（只有训练集）
        - 分离回归系数与**截距项**，计算残差
        - 计算残差扰动后的值，构建新模型数据（LabeledPoint）
        - 循环上述过程
- 子集合自举法
    - 常用函数：
        - mapValues(s => fun(s))　　注意：KVP必须是RDD形式
        - reduceByKey((x,y) => fun(x,y))　注意：reduceByKey之后需要collect()才会显示结果
        - sampleByKey(replacement,frac)
            - frac为List((1,抽样比例1),(2,抽样比例2))**.toMap**
        - sampleByKeyExact(replacement,frac)
        - KVP1.join(KVP2)
        - KVP.values
    - 思路梗概：
        - 产生N个RDD格式的键值对
        - 对具有相同键值的数据进行抽样，记录各组样本数目
        - 自举法：对分组样本进行100%重抽样
        - 反复利用join，mapValues，reduceByKey函数进行计算