第一章 Apache Spark简介
- 理论部分
- 代码部分
  - 基础语句
    - 值与变量
    - 数值转换
    - 复杂的数学运算
    - 表达式块和条件表达式
    - 循环语句和迭代对象
    - 函数
  - RDD语句
    - 内部数据定义RDD
    - 从读取外部数据并生成RDD
    - map函数
    - filter函数
    - reduce函数

Breeze程序包
- 理论部分
- 代码部分
  - 需要记忆的程序包
  - linalg中的DenseVector
    - 特殊向量的创建
    - 转置和求长度
    - 从range创建坐标向量
    - 切片和广播
  - linalg中的DenseMatrix
    - 特殊矩阵的创建
    - tabulate创建向量或矩阵及二维坐标
    - 切片和索引及赋值
    - 获取尺寸
    - 拉直与合并
  - linalg中的运算
    - 基本运算
    - 最值、和与范数
    - 按行或列的运算
    - 行列式和逆
    - 谱分解
    - 自定义函数
  - numerics包：元素类型应为Double
  - stats.distributions包
    - 创建随机数对象
    - 生成随机数与查看pdf等值
    - 将pois对象改为双精度类型的生成器
    - 均值和方差
    - foldLeft和foldRight
    - 向量内积

随机模拟和统计计算
- 理论部分
  - 逆累积分布法生成随机数
  - EM算法求高斯混合模型
- 代码部分
  - 要记的程序包
  - 使用java包生成随机数
    - 调用java的随机数包
    - 生成随机数种子
    - 均匀分布随机数（一个）
    - 标准正态随机数（一个）
    - 逆累积分布生成1000个随机数
  - mllib包生成RDD随机数
  - 回归模拟的小代码
    - 线性回归模型
    - 广播
    - 定义一个Array向量
    - 给每个Double变量保留四位小数
    - Array的重要函数
    - 用java.io._进行输出

读取
EM算法小代码
生成全零Array
生成混合正态分布随机数
RDD->文本->Double
优化方法
理论部分
ADMM算法
随机梯度下降法
最小二乘、岭回归和LASSO
代码部分
生成随机数的方法
breeze.stats.distributions
java.util.concurrent.ThreadLocalRandom
org.apache.spark.mllib.random.RandomRDDs._
scala.util.Random
矩阵的分布式运算
要导入的包
Spark线性运算类
RDD中的线性运算类
优化算法
自举法
理论部分
自由自举法
子集合自举法
代码部分

第一章 Apache Spark简介

「 理论部分」

无

「 代码部分」

基础语句

值与变量

```
var a=3
val b=4
```

- val是不可变的，有对应类型的存储单元
- var是可变的，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，其内容是动态的
- 对于Array类型的val，其中的元素是可变的

数值转换

- 常用的类型：Int[-2^31, 2^31-1], Long [-2^63, 2^63-1], Double, Boolean {True, False}, String
- Int -> Double : `val c=5; c.toDouble`
- Double -> Int : `val b=3.6; b.toInt` ，直接去掉小数部分
- String -> Double : `val s="123.45"; s.toDouble`
- String -> Int : `val s="123";s.toInt`
- `var a = 3.0; a = 2` ：合法，会自动将2转换为2.0
- Int和Double在四则运算后得Double

- 常见报错：
  - `var a = 2; a = 3.0`：报错，将Double 赋给了Int
  - `val s="123.45"; s.toInt`：报错，跳步转换了

复杂的数学运算

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

- `min(20,4)`：二元函数，返回较小的数
- `pow(2,0.3)`：2的0.3次方。
  - 当幂次是无限小数时，pow函数的第一个参数不能是负数，否则返回nan

表达式块和条件表达式

- 表达式块由多个表达式用大括号构成，有自己的作用域，里面可有全局和局部变量
- 表达式块最后一个表达式的值是整个表达式块的返回值
  - 赋值语句的值是Unit，即空
- ```
  if(a>b){
      println(a)
  }else if(a<b){
      println(b)
  }else println(0)
  ```
  - 注意：`if` 和 `if` 之间必须有 `else`
  - 语法：`if(<Boolean expression 判断语句>)<expression>`

循环语句和迭代对象

- for循环：

  ```
  for(t <- 0 until 4 if t%4==0) println(t)
  ```

  - 语法：`for(<identifier变量名>  <- <iterator循环域>) [yield] [<expression 循环体>]`
  - 使用 `yield` 会输出一个向量
  - ```
    for(i <- 0 until 4) yield i
    res2: scala.collection.immutable.IndexedSeq[Int] = Vector(0, 1, 2, 3)
    ```

    不能写println(i)，因为这个语句不会返回值
  - %：取余数
  - `if` 是迭代对象的守卫，表示筛选条件

- `until` 和 `to`：
  - `0 until 2`：代表0,1
  - `0 to 2`：代表0,1,2
  - `until` 左闭右开，`to` 左闭右闭
- 双循环变量：

```
for (x <- 0 until 2; y <- 0 until 2){
    println("x="+x+",y="+y+",x*y="+(x*y))
}
x=0,y=0,x*y=0
x=0,y=1,x*y=0
x=1,y=0,x*y=0
x=1,y=1,x*y=1
```

  - 字符串加法：用 `+号` 表示，两端都是字符串；也可以一端是字符串一端是数字，则会将数字自动转换为字符串
  - 输出逻辑见上

- while循环：

```
var a=0
while(a<20){
    println(a)
    a=a+1
}
```

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

- do-while循环：

```
var a=0
do{
    println(a)
    a=a+1
}while(a<20)
```

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

函数

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

  - 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>`
  - 输入参数和输出结果都可以是向量

    ```
    def prod(arr:Array[Double]):Double={
        var temp=1.0
        for (i <- 0 until arr.length){
            temp=temp*arr(i)
        }
        return temp
    }
    ```

    注意声明中向量必须指定元素类型
    ```

RDD语句

内部数据定义RDD

- 向量：List()， `Array()`
  - 调出Array中的元素：用圆括号，索引从0开始
  - 查看向量的长度： `.size` 或者 `.length` ，返回一个Int
  - List一旦创建，不能改变元素值，只能增减元素; Array则可以修改
- ```
  val numbers=Array(1,2,3,4)
  val rddEx=sc.parallelize(numbers)
  rddEx.take(2)
  res17: Array[Int] = Array(1, 2)
  ```
  - 如上用 `sc.parallelize()` 来将一个向量并行化生成rdd。
  - 查看第一个数： `rddEx.first` ，返回一个数
  - 查看前若干个数： `rddEx.take(2)` ，返回一个Array
  - 返回整个内容： `rddEx.collect()` ，返回一个Array

从读取外部数据并生成RDD

- ```
  val rddEx=sc.textFile(<路径名>)
  ```
  - 返回一个rdd
  - 按行读取，rdd里面装的是一个 `Array[String]` ，每个元素是一行的数据且是字符串形式，对每个元素：若是csv文件则分隔符是逗号。
  - 用mapreduce将数据清洗出来

    ```
    val data=rddEx.map(line => line.split(",")).map(_.toDouble)
    ```
    - `.split` 是将一个字符串切割、转换成一个 `Array[String]`

map函数

- ```
  val res=data.map(z => (z(0)+z(1)+z(2),z.size))
  ```
  - 每个元素的类型可以自由变换，这里是把Array[Int]变成了tuple(Int,Int)
- 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
- 语法： `data.map(s => s.func)` ， `data.map(_.func)`

filter函数

- ```
  val screening=res.filter(s => s._1>4).map(s => (s._1,s._2*2))
  ```
- 语法： `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

reduce函数

- ```
  val fin=screening.reduce((x,y)=> (x._1+y._1,x._2+y._2))
  ```
  - 元组第一个元素相加，第二个元素相加，返回由这两个和组成的元组
- 由于RDD的惰性调用，前面各步都是只记录不执行，只在 `reduce、collect、first、take` 处进行了action
- 语法： `reduce( (x,y) => func(x,y))`
  - 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
  - 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。

Breeze程序包

「 理论部分」

无

「 代码部分」

需要记忆的程序包

```
import breeze.linalg._
import breeze.numerics._
import breeze.stats.distributions._
```

linalg中的DenseVector

特殊向量的创建

- `DenseVector.zeros[Double](N)` ， `DenseVector.ones[Double](N)` ：元素全为Double的全0或1向量
- `DenseVector(a,b,...)` ：直接创建一个DenseVector，有Double则为Double型
- `DenseVector.fill(n,a)` ：用a填充组成一个n长的向量，元素类型同a
- `DenseVector.range(b,e,s)` ：三个参数均为Int，begin, end, step，是左闭右开的
- `Vector.rangeD(b,e,s)` ：三个均可以为Double，输出的结果的元素类型必为Double
- `linspace(b,e,n)` ：前两个参数均可以为Double，n必为Int，输出的结果的元素类型必为Double

转置和求长度

- `a.t` : breeze包产生的向量均为列向量，经过转置变为行向量
- `a.length` ， `a.size` ：求向量长度，返回一个Int
```

从range创建坐标向量

```
val a=DenseVector.zeros[Double](10)
a:=DenseVector((0 until a.length).toArray.map(_.toDouble))+a
```

- `(0 until 10).toArray.map(_.toDouble)`：用(.toArray)把其他类型的向量转换成Array，再把元素变成Double
- Array上作用 `DenseVector` 可转变为DenseVector
- 同长度向量直接用 + 号做加法
- `:=`：按元素给向量赋值。如果需要赋值的元素个数超过一个，就需要使用“**:=**”
  - 左右两边的元素类型必须一致

切片和广播

```
a(1 to 3):= -1.0    //负号前一定要空格
a(0)=100.0
```

- 在圆括号中直接放入一个**until**或**to**的range即可切片
- 用**:=**可以对单元素实现广播，即**:=**要不为单元素，要不为对应长度的DenseVector
- 支持上面两种直接改值

linalg中的DenseMatrix

特殊矩阵的创建

- `DenseMatrix.zeros[Double](n,m)`：n*m的Double型全零矩阵
- `DenseMatrix.ones[Double](n,m)`：n*m的Double型全1矩阵
- `DenseMatrix.eye[Double](N)`：N阶Double型单位阵
- `diag(<DenseVector>)`：生成以传入向量为对角向量的对角阵
- `diag(<DenseMatrix>)`：抽取传入矩阵的对角向量
- `DenseMatrix((a1,b1,...),(a2,b2,...),...)`：直接创建一个DenseMatrix，有Double则为Double型
- `new DenseMatrix(rows=n,cols=p,Array[XX])`

tabulate创建向量或矩阵及二维坐标

```
val tavec=DenseVector.tabulate(5){i=>i*i}
val tamat=DenseMatrix.tabulate(3,4){case(i,j)=>i+j}
val a=DenseVector(1.0,2.0,3.0,4.0,5.0)
val denseta=DenseMatrix.tabulate(5,5){case(i,j)=>a(i)*a(j)}
val rotation=DenseMatrix.tabulate(3,4){case(i,j)=>(i,j)}
(0 until 3).toArray.map(i => rotation(i,::).t.toArray)

tavec: breeze.linalg.DenseVector[Int] = DenseVector(0, 1, 4, 9, 16)
tamat: breeze.linalg.DenseMatrix[Int] =
0  1  2  3
1  2  3  4
2  3  4  5
denseta: breeze.linalg.DenseMatrix[Double] =
1.0  2.0  3.0
2.0  4.0  6.0
3.0  6.0  9.0
rotation: breeze.linalg.DenseMatrix[(Int, Int)] =
(0,0)  (0,1)  (0,2)  (0,3)
```

```
(1,0)  (1,1)  (1,2)  (1,3)
(2,0)  (2,1)  (2,2)  (2,3)
res7: Array[Array[(Int, Int)]] = Array(Array((0,0), (0,1), (0,2), (0,3)),
Array((1,0), (1,1), (1,2), (1,3)), Array((2,0), (2,1), (2,2), (2,3)))
```

- `DenseVector.tabulate(n){func}` ：将**0 until n**经过变换func生成DenseVector
- `DenseVector.tabulate(m,n){func}` ：将**0 until m,0 until n**经过变换func生成DenseMatrix

切片和索引及赋值

- `mat(nr,::)` ：mat的第**nr+1**行
- `mat(::,nc)` ：mat的第**nc+1**列
- `mat(a1 to/until a2,b1 to/until b2)` ：切出一个子矩阵，注意索引是从0开始的
- 可以用**:=**进行单元素广播或者对应尺寸矩阵的替换，**=**可用

获取尺寸

- `mat.rows` ：行数
- `mat.cols` ：列数

拉直与合并

- `mat.toDenseVector` ：将矩阵(按列)拉直
- `DenseMatrix.horzcat(mat1,mat2)` ：将两个矩阵水平合并
- `DenseMatrix.vertcat(mat1,mat2)` ：将两个矩阵垂直合并

linalg中的运算

基本运算

- `mv1*mv2` ：向量矩阵乘法
- `mat1*:*mat2` ： 按位相乘
- `mat1/:/mat2` ：按位相除
- `mv:*=2.0` ：按位递乘2.0，没有冒号结果也对
- `sqrt(num)` ：对一个数开根号

最值、和与范数

- `min(mv)` ：返回所有元素中的最小值。最大值即为max
- `argmax(mv)` ：返回最大值的坐标或位置。最小值即为argmin
- `sum(mv)` ：求出所有元素的和
- `norm(vec)` ：求出元素二范数，即平方和开根号

按行或列的运算

- `mat(*,::)` ：相当于(axis=1)，运算方向为columns
- `mat(::,*)` ：相当于(axis=0)，运算方向为index
- 对上面各函数使用则变为了对各行或列分别求
- 应用：中心标准化

```
val mean=sum(a(*,::))/a.cols.toDouble  //出来一个列向量
a(::,*)-mean  //按列相减，此为中心化的步骤
//或者：下面的结果与上面的结果互为转置
val meancol=sum(a(::,*))/a.rows.toDouble //出来一个行向量
a(*,::)-meancol.t   //减去必须是列向量
a(::,*)/DenseVector(1.0,2.0,3.0)
//按行相除，被除的必须为列向量，可以此进行标准化
```

  - 减去的必须为列向量，否则报错
  - 被除的必须为列向量，否则报错

行列式和逆

- `det(mat)`：行列式
- `inv(mat)`：逆
- `pinv(mat)`：广义逆

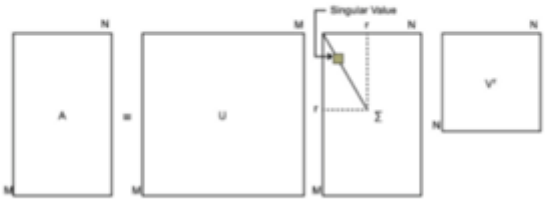
谱分解

- 以下函数都只对 `DenseMatrix[Double]` 能用
- `eig(mat)`：返回特征值组成的向量和特征向量组成的矩阵
  - `eig(mat).eigenvalues`：特征值组成的向量
  - `eig(mat).eigenvectors`：特征向量组成的矩阵
- `val svd.SVD(u,d,v)=svd(mat)`：
  - d是Σ的对角阵部分的对角向量: 先将d变为对角阵，再在下面拼一个零矩阵跟v的维数变成一样（维数参考下图）
  - mat=u*DenseMatrix.vertcat(diag(d),DenseMatrix.zeros[Double](1,2))*v

$$A = U\Sigma V^T$$

其中 U 是一个 $m \times m$ 的矩阵，Σ 是一个 $m \times n$ 的矩阵，除了主对角线上的元素以外全为0，主对角线上的每个元素都称为奇异值，V 是一个 $n \times n$ 的矩阵，U 和 V 都是酉矩阵，即满足

$U^TU = I, V^TV = I$。下图可以很形象的看出上面SVD的定义：



  - 那么我们如何求出SVD分解后的U,Σ,V这三个矩阵呢?
- `rank(mat)`：求矩阵的秩
- 验证矩阵相等：用 `norm((mat1-mat2).toDenseVector)`

自定义函数

- 注意函数声明中一定要写中括号和元素类型
- 传入的DenseVector其实是传入引用：注意，函数传入的变量默认是val，可以改值而不能整体更换

```
import breeze.linalg._
var x0=DenseVector(Array(1,5,3,7,4,2,7,8,2,4,6,9,3,3,76,8))
def cf(v:DenseVector[Int],k:Int):DenseVector[Int]={  //DenseVector一定要写中括号及元素类型!
    if(k==0) v
    else {
```

```
        val x1=cf(v,k-1)
        v(0)=4 //改着试试
        x1(1 to x1.length-1)-x1(0 to x1.length-2)
    }
}
cf(x0,5)

import breeze.linalg._
x0: breeze.linalg.DenseVector[Int] = DenseVector(4, 5, 3, 7, 4, 2, 7, 8, 2, 4,
6, 9, 3, 3, 76, 8)
cf: (v: breeze.linalg.DenseVector[Int], k: Int)breeze.linalg.DenseVector[Int]
res6: breeze.linalg.DenseVector[Int] = DenseVector(43, -23, -15, 25, 10, -41,
32, -20, 36, 27, -333)
```

numerics包：元素类型应为Double

- 调用数学函数包：import breeze.numerics._
- `sin, cos, tan, log, exp, log10, sqrt, pow`
- 可对向量和矩阵逐元素作用，区别于 `scala.math._` 中的函数，只能对单个数字作用
- `pow` 的分数次方时，传入负数一律返回NaN，解决方法是传入整数再取负号

stats.distributions包

- 导入包： `import breeze.stats.distributions._`

创建随机数对象

```
val pois=new Poisson(2.0)  //整数类型
val gau=new MultivariateGaussian(muVector, sigmaMatrix)
```

生成随机数与查看pdf等值

- ```
  pois.probabilityOf(2)
  pois.cdf(2)
  ```
- ```
  val beta=new Beta(1.0,2.0)
  val randb=beta.sample(10)
  randb.map{beta.pdf(_)}
  //或：randb.map{i => beta.pdf(i)}
  beta.cdf(0.05)
  //pdf可以直接对Array作用，但是其他的函数不行
  beta.pdf(randb.toArray)
  ```
- 生成向量用 `.sample(N)`
- 生成一个数用 `.draw()`

将pois对象改为双精度类型的生成器

```
val doublepois=for(x<- pois) yield x.toDouble
doublepois.sample(10)//产生了双精度样本

res12: IndexedSeq[Double] = Vector(3.0, 2.0, 2.0, 0.0, 4.0, 2.0, 2.0, 4.0, 2.0,
0.0)
```

均值和方差

- 样本均值方差和容量（只对元素类型为Double的样本）
  - ```
    breeze.stats.meanAndVariance(doublepois.sample(10000))
    //三个返回值为：均值、方差和样本容量

    res9: breeze.stats.meanAndVariance.MeanAndVariance =
    MeanAndVariance(2.006100000000007,2.0898717771777244,10000
    ```
- 总体均值和方差：

```
pois.mean   //总体均值
pois.variance   //总体方差
```

foldLeft和foldRight

- foldLeft: `List(1,2,3).foldLeft(0)(f)=f(f(f(0,1),2),3)`
- foldRight: `List(1,2,3).foldRight(0)(f) = f(1, f(2, f(3,0)))`
- 例子：
- List(1,2,3).foldLeft(0){(x,y)=>x+2*y}

  List(1,2,3).foldRight(0){(x,y)=>x+2*y}

  前面是从左开始，0+1*2=2；2+2*2=6,6+3*2=12

  后者是从右开始，3+2*0=3；2+2*3=8；1+2*8=17

  第一个括号里的0是初始值，第二个括号里是函数
- f必为二元函数，第一个参数必为上一次迭代的结果，第二个参数为从List（或其他向量类型）中依次抽出的值：
  - ```
    data0.foldLeft(expFam.emptySufficientStatistic)
    {(x,y)=>x+expFam.sufficientStatisticFor(y)}
    ```

向量内积

- **vec1.t * vec2** 或 **vec1 dot vec2**
- data.map(s=>s*s.t).reduce((x,y)=>x+y)：这是变成矩阵再reduce，不是内积
- 补充知识点：

  问有一个任务若用一个节点的时候要用T时间去完成，现在如果我们有M个节点：

  若所用函数为map，时间会减少为：$\frac{T}{M}$

  若所用函数为reduce，时间会减少为：$\frac{T}{M}$

  map好理解，就这么多任务，用M个节点去做肯定减少为上面的时间

  看reduce：

  n个数求和最少要加(n-1)次，就算是reduce的加的方法也是(n-1)次。

  M个节点即把n个数分成了M堆，每一堆上有$[\frac{n}{M}]$个数，要做$[\frac{n}{M}]-1$次加法，再把这M个和加起来需要M-1次加法，最终需要的时间就是$\frac{T}{M}$

随机模拟和统计计算

「理论部分」

逆累积分布法生成随机数

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

（1）计算过程

$$F(x)=\int f(x)=\frac{x^3}{2}+C=\frac{x^3}{2}+\frac{1}{2}(x\in[-1,1])$$

反函数：$y=F(x)\Leftrightarrow x=g(y)=(2y-1)^{\frac{1}{3}}(y\in[0,1])$

均值：$\mu=\int_{-1}^{1}x*f(x)dx=\int_{-1}^{1}\frac{3x^3}{2}\,dx=0$

方差：$\sigma^2=E(x^2)-\mu^2=\int_{-1}^{1}x^2*f(x)dx=\int_{-1}^{1}\frac{3x^4}{2}\,dx=\frac{3}{2*5}x^5\Big|_{-1}^{1}=\frac{3}{5}$

- 则 `g(U(0,1))` 即为所求

EM算法求高斯混合模型

基本的 EM 算法可以描述为：

- E-step：

$$\gamma_{ik}^{(t+1)}=P(\mathbf{x}_i\text{来自于第}k\text{个正态分布}|\text{data})=\frac{\pi_k^{(t+1)}N(\mathbf{x}_i|\mu_k^{(t)},\Sigma_k^{(t)})}{\sum_{j=1}^{K}\pi_j^{(t+1)}N(\mathbf{x}_i|\mu_j^{(t)},\Sigma_j^{(t)})},$$

where $\pi_k^{(t+1)}=\frac{N_k^{(t+1)}}{N}$.

- M-step:

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

不用记迭代公式，会给的

「代码部分」

要记的程序包

- `java.util.concurrent.ThreadLocalRandom`
- `org.apache.spark.mllib.random.RandomRDDs._`

使用java包生成随机数

调用java的随机数包

```
import java.util.concurrent.ThreadLocalRandom
```

生成随机数种子

```
val r=ThreadLocalRandom.current
```

均匀分布随机数（一个）

```
r.nextDouble
```

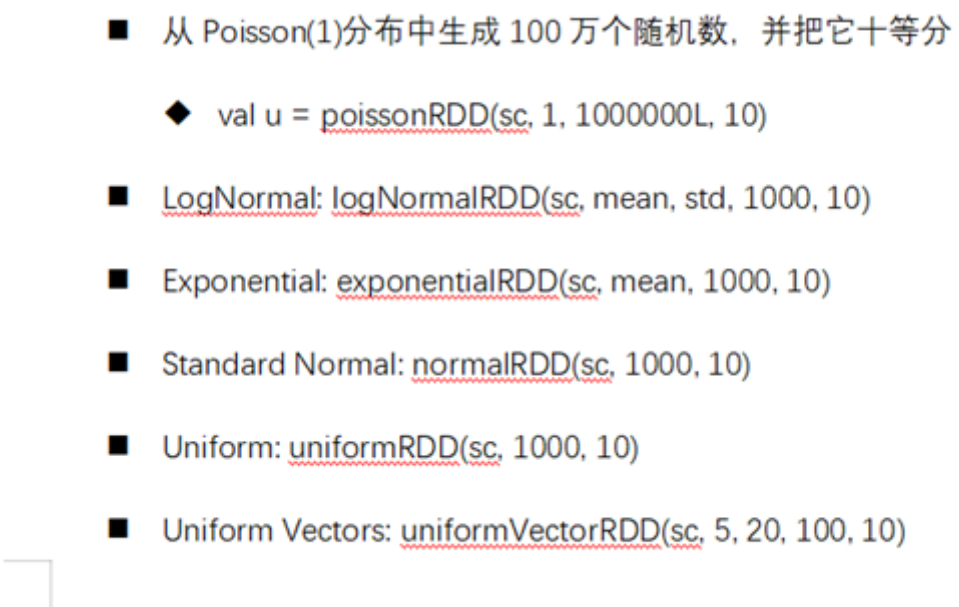
标准正态随机数（一个）

```
r.nextGaussian
```

逆累积分布生成1000个随机数

- `(1 to 1000).map(x=>r.nextDouble).map(x=> -math.log(x)).toArray[Double]`
  - 注意**=>**与**-**之间有个空格

mllib包生成RDD随机数

- `import org.apache.spark.mllib.random.RandomRDDs._`
- `val u = poissonRDD(sc, 1, 1000000L, 10)`
  - 100万个泊松分布的随机数，那个L要不要无所谓
  - 分布式计算的时候要用到sc，这个对象是在初始化spark的时候会自动生成的(Spark context available as 'sc')
  - **十等分**指分在十个**NumOfSlaves**上，即十个计算节点。即上面的第三个参数。
- 

回归模拟的小代码

线性回归模型

- 线性回归模型$y_i = x_i^T\beta + \epsilon_i, i = 1, \ldots, 400$
  - 常数向量β长度为 5000，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
  - 前 200 个数据和后 200 个数据分别存储在不同的文件里

广播

- `val A=sc.broadcast(2.0)`
  - 在整个Spark系统中广播a系数，从而使得每个计算节点上都可以读取这个变量的值
- `val a=A.value`
  - 在此计算节点上读取变量A的值

定义一个Array向量

`val a=new Array[String](n)`：全零向量

给每个Double变量保留四位小数

- `x.map("%.4f" format _)`：注意format前后各一个空格
- `"%.4f".format(y)`：对单个变量y保留四位小数

Array的重要函数

- `<Array>.drop(n)`：删去前n个数后的Array
- `<Array>.take(n)`：取出前n个数组成的Array
- `<Array>.union(<Array>)`：两个Array拼接在一起组成的Array
- `<String>.split(",")`：以,为分隔符将字符串切割成Array[String]
- `Array.fill(N)(a)`：生成元素全为a的N长向量
  - 当a是随机数的时候，相当于是用a生成了N个随机数组成一个Array，结果各异的

用java.io._进行输出

```
val output=lines.collect().map(s => {  //这得用collect回到串型的环境中才能这样写，RDD环境中可能不支持下面这些操作
    import java.io._
    val path="LinearModel_"+s._1+".txt"; //元组的第一个元素
    val pw=new PrintWriter(new File(path));
    pw.write(s._2);
    pw.close;
    path
})
```

读取

- `val lines=sc.textFile("LinearModel_0.txt")`：自动以行为单位划分为数组（划分依据：\n）

EM算法小代码

生成全零Array

- `val data=Array.ofDim[Double](N)`：中括号为元素类型，圆括号为元素个数
- `val d2=Array.ofDim[Double](m,n)`：二维，三维以此类推

生成混合正态分布随机数

```
for(i<- 0 until N){
    val db=random.nextDouble //均匀分布，来决定是生成哪个正态的随机数
    if(db<P){
        data(i)=MU1+Sig1*random.nextGaussian //来自第一个正态分布
    }else{
        data(i)=MU2+Sig2*random.nextGaussian //来自第二个正态分布
    }
}
```

RDD->文本->Double

```
var ParData=sc.parallelize(data,NumOfSlaves)
val ParDataStr=ParData.map("%.4f" format _) //保留四位小数，Array[String]
val ParData=ParDataStr.map(_.toDouble)
```

防止直接产生的随机数无法序列化带来的问题

优化方法

「理论部分」

ADMM算法

- 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：

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

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：

$$minimize \sum_{i=1}^{n}\rho_\tau(z_i)\ subject\ to\ z = y - X\beta$$

其中$z = (z_1, \dots, z_n)^T, y = (y_1, \dots, y_n)^T, X = (x_1, \dots, x_n)^T$。

- 这就相当于求解

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

其中$\langle\cdot,\cdot\rangle$表示内积。

- τ是要回归的分位数，r1项是取定的常惩罚项，内积对应的是拉格朗日乘子补的东西
  - 具体的更新步骤如下：
    - (1)更新β

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

    - (2)更新 z

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

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

    - (3)更新α

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

随机梯度下降法

- 我们记$l(z_i;\theta) = d(m(x_i;\theta), y_i)$，其中$z_i = (x_i^T, y_i)^T$。
- 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：

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

最小二乘、岭回归和LASSO

- **最小二乘估计**：

$$\hat{\beta} = \arg\min_{\beta} \frac{1}{2} \sum_{i=1}^{n} \left(y_i - \sum_{j=1}^{p} \beta_j x_{ij} \right)^2$$

- **岭回归估计**：

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

其中λ是调试参数，$\|\cdot\|_2$表示L_2模。

- **Lasso 估计**：

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

其中λ是调试参数，$\|\cdot\|_1$表示L_1模。

- 随机梯度下降算法都可以为上面的三种估计量制定迭代公式。

不重要

「 代码部分」

生成随机数的方法

breeze.stats.distributions

java.util.concurrent.ThreadLocalRandom

org.apache.spark.mllib.random.RandomRDDs._

scala.util.Random

```
import scala.util.Random
val random=new Random()
val u=random.nextDouble //抽取一个
val OurExpRVs=(1 to 100).map(x=>random.nextDouble)  //抽取一百个
```

矩阵的分布式运算

要导入的包

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

Spark线性运算类

- `new sparkDenseMatrix(3,2,Array(1.0,2.0,3.0,1.0,2.0,3.0))`
  - 3行2列，按列填充
- `Vectors.dense(A)`：A是一个Array[Double]
  - A不能是Array[Int]

RDD中的线性运算类

- BlockMatrix
  - 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: BlockMatrix 的转置
  - A.toLocalMatrix: 将分布式的 BlockMatrix A 收取到主节点上，形成一个 DenseMatrix
  - A.toIndexedRowMatrix() 将 BlockMatrix A 转换成一个 IndexedRowMatrix

优化算法

- 调用不同的目标函数：↵
  - import org.apache.spark.mllib.optimization.{GradientDescent, LogisticGradient, SquaredL2Updater}↵
    - 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↵
    - 当使用最小二乘估计的目标函数，使用 SimpleUpdater↵
    - 如果使用岭回归的目标函数，使用 SquaredL2Updater↵
    - 如果使用 Lasso 的目标函数，使用 L1Updater↵

- 将数据集data按0.6-0.4的比例随机的分成两部分:
  - `val splits = data.randomSplit(Array(0.6, 0.4), seed=11L)`
- `MLUtils.appendBias(Vector)` : 在Vector向量的最后加一个元素1
- `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
- `Labeledpoint.features` ： 解释变量
- `LabeledPoint.label` ： 响应变量
- 着重理解参数的意义

自举法

「理论部分」

自由自举法

自由自举法

考虑如下的线性回归模型：

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

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$\omega_1,\ldots, \omega_N$.
3. 产生伪响应变量观察值$y_i^* = x_i^T \hat{\beta}_0 + \omega_i \hat{\epsilon}_i, i = 1,\ldots, N$.
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。

子集合自举法

子集合自举法(Bag of little bootstraps, BLB)

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$\hat{\xi}_k^*, k = 1,\ldots, K$
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。

「 代码部分」

Key-value pairs: (key, value)可以翻译为键-值对

- `import org.apache.spark.rdd.PairRDDFunctions`
- `import org.apache.spark.HashPartitioner`
- `mapValue(s => Func(s))` ： 仅处理value部分
- `reduceByKey((x,y)=>x+y)` ： 按照key部分合并（相同的key进行reduce）
- `sampleByKey(withReplacement, fraction)` ： 按照key进行抽样，第一个参数决定是否放回
  - `fraction = List((1, 0.4), (2, 0.3)).toMap` ： 给定了不同key下的抽样概率（用U(0,1)<P这一事件来决定某一个体被分到哪一边）
  - 与下同理：

```
val a=Array.ofDim[Double](10000)
import java.util.concurrent.ThreadLocalRandom
val random=ThreadLocalRandom.current
for(i<-0 until 10000)
{  if(random.nextDouble<0.4){a(i)=1.0}else{a(i)=0.0}}
sum(a)
```

结果未必精确等于4000

- `sampleByKeyExact(withReplacement, fraction)`
  - 使得用户可以准确的从子集k中抽出[fk*nk]个数据，fk是用户需要的比例大小，nk是子集k的样本量
- `KVP1.join(KVP2)`：将KVP1和KVP2进行连接，以key值为连接变量，是内连接(**how="inner"**)
- `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