# 迪利克雷分布的充分统计量

## 「理论」

理论：充分统计量

$$f(\alpha,x)=(\frac{1}{B(\alpha)})^n\Pi_{j=1}^n\Pi_{i=1}^3x_{ij}^{\alpha_i-1}=(\frac{1}{B(\alpha)})^ne^{\Sigma_{i=1}^3\alpha_i\Sigma_{j=1}^nlnx_{ij}}e^{-\Sigma_{i=1}^3\Sigma_{j=1}^nlnx_{ij}}$$

由指数族的充分统计量，可知充分统计量就是取ln再加和。
由mle是充分统计量的一个函数，故我们先算充分统计量，再用它算mle

## 「代码」

- ```
  val mydir=new Dirichlet(DenseVector(3.0,2.0,6.0))
  val dat=mydir.sample(10)  //多元分布，10个DenseVector
  val data0=dat.toIndexedSeq
  //为了做下面的mle必须要这么改
  ```
  - 生成了要用的样本
  - `.toIndexedSeq`
  ```

- ```
  val expFam=new Dirichlet.ExpFam(DenseVector.zeros[Double](3)) //指数族，全零
  //定义充分统计量初值为(0,0,0)
  val SuffStat=data0.foldLeft(expFam.emptySufficientStatistic)
  {(x,y)=>x+expFam.sufficientStatisticFor(y)}

  SuffStat: expFam.SufficientStatistic =
  SufficientStatistic(10.0,DenseVector(-19.22369973869366,
  -13.545735118154479, -6.597899982729782))
  ```
  - 指数族对象：`new Dirichlet.ExpFam()`
  - `expFam.emptySufficientStatistic`：是一个全零的sufficient类型
  - `sufficient类型`
    - 包括两部分，第一个参数用来统计的是样本的个数（即foldLeft中迭代的次数）：初始时都是1，每两个相加会使它也加1.
    - 第二个元素即为取log之后的结果（迪利克雷分布的充分统计量就是取log）
  - 第一个括号中的值为初值
  - x指代上一次迭代的结果，就是sufficient类型的
  - y是从样本data0中顺序抽的，转化成sufficient类型和x相加
- ```
  val mle=expFam.mle(SuffStat)

  mle: DenseVector[Double]
  ```

# 从回归模型中模拟数据

## 「理论部分」

- 线性回归模型$y_i = x_i^T\beta + \epsilon_i, i = 1, \ldots, 400$
  - 常数向量$\beta$长度为 5000，前 10 个数为 2，其余为 0
  - 模型误差$\epsilon_i$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

## 「代码部分」

### 生成数据

- ```
  import java.util.concurrent.ThreadLocalRandom
  val RowSize=sc.broadcast(200)
  val ColumnSize=sc.broadcast(5)
  val RowLength=sc.broadcast(2)//将要生成的400个数据分成两拨，同时用两个节点生成
  val ColumnLength=sc.broadcast(1000)//列向量分块方法：5个一组，共1000块
  val NonZeroLength=sc.broadcast(10)//非零变量个数
  val p=ColumnSize.value*ColumnLength.value
  val betta=(1 to p).map(_.toDouble).toArray[Double].map(i => { if(i <=
  NonZeroLength.value ) 2.0 else 0.0 })//前10个数为2，其他数都是0
  val MyBeta=sc.broadcast(betta)
  var sigma=1.0
  val Sigma=sc.broadcast(sigma)
  var indices=0 until RowLength.value //两个节点，Array(0,1)
  var ParallelIndices=sc.parallelize(indices,indices.length)
  //投射到RDD里面去，后一个参数不加也可（是指明节点个数的，显然这里是两个节点，因为默认是传
  入的Array多长就多少个节点）
  ```
  ```

```
def rn(n:Int)=(0 until n).map(x => r.nextGaussian).toArray[Double] //生成长度为5的标准正态样本
```

  - 用两个节点同时生成，每个节点上生成200个数据
  - `sc.parallelize` 默认是传入的Array多长就多少个节点，也可以用后一个参数来指定节点数

- 生成线性模型
- ```
val lines=ParallelIndices.map(s => {  //lines两个元素，每个元素化身一个巨长的字符串，分别是两批数据
    val r=ThreadLocalRandom.current
    val beta=MyBeta.value
    val sigma=Sigma.value
    val rowsize=RowSize.value
    val columnsize=ColumnSize.value
    val columnlength=ColumnLength.value
    var lines=new Array[String](rowsize)
    val p=columnsize*columnlength
    //下面生成(y|X)阵
    for(i <- 0 until rowsize){ //rowsize=200
        //每一块五列，共1000块，这是对每一行进行循环
        var line=""; //每一行是一个字符串
        var y=0.0;  //每一行求一个y出来
        for(j <- 0 until columnlength){  //columnlength=1000
            //这是对这1000块进行循环
            var x=rn(columnsize) //5个正态随机数，作为第i行第j块的那5个
            for(k <- 0 until columnsize){ //columnsize=5
                //这是处理每一块内部的加和
                y+= beta(j*columnsize+k)*x(k)
                //对每一小块都这样加，最终得到这一行的y值，即yi=beta.t*xi
            }
            var segment=x.map("%.4f" format _).reduce(_+" "+_)
//数与数之间用空格隔开
            line=line+","+segment //将每一块用逗号隔开
        }//完成了每一块的操作，每一块的数都加到了y上去，现在是yi=beta.t*xi
        y += sigma*r.nextGaussian //对各行y值加上误差项
        lines(i)="%.4f".format(y)+line+"\n" //将y拼到5000个x前面
        //生成文件的话一定要加上换行符，一个yi一个换行符（一个yi带着它的5000个xi的分量占一行）
    }
    //得到lines是一个200长的Array[String]，每个元素以\n结尾
    (s,lines.reduce(_+_)) //此为返回值，把这两百个元素合并起来；元组的第一个数s为节点的标签。（元组的元素可为不同的类型）
}
)
//这是200*2个，在两个节点上同时跑，每个节点分别输出200个
```
- 得到lines后，每运行一次 `lines.collect()` 都会得到不一样的结果，是由于**mapredece**的惰性调用，导致每次执行这个action操作之后，都会将map的结果重新运行一遍

## 输出数据

- 非分布式的输出：

```
import java.io._
val pw=new PrintWriter(new File("aa.txt"))
pw.write(lines.collect()(0))
pw.close
```
```

- 使用分布式的输出：两个节点同时输出

```
import java.io._
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 //返回路径
})

output: Array[String] = Array(LinearModel_0.txt, LinearModel_1.txt)
```

  - rdd的map和reduce无序，但是串型结构上的有序，见下面的测试

```
val test=Array("1","2","3")//字符串可以直接+
test.reduce(_+" "+_)   //有序的

test: Array[String] = Array(1, 2, 3)
res35: String = 1 2 3

val rdd=sc.parallelize(test)
rdd.reduce(_+" "+_)  //可能无序了

rdd: org.apache.spark.rdd.RDD[String] = ParallelCollectionRDD[14] at parallelize at <console>:38
res34: String = 2 1 3
```

  - 注意：union**不是可交换**函数，不可在rdd的**reduce**里面用，但是可以在串型的reduce用

## 读取数据并使用它

- ```
  import breeze.linalg._
  val lines=sc.textFile("LinearModel_0.txt")
  val transLines0 = lines.map( s => {
      val Y = DenseVector(s.split("\n").map(_.split(",")(0).toDouble))
      val X = s.split("\n").map(_.split(",").drop(1).map(_.split(" ").map(_.toDouble)))
      (Y,X)
  }
  )
  val transLines1 = lines.map( s => {
      val Y = DenseVector(s.split("\n").map(_.split(",")(0).toDouble))
      val X = s.split("\n").map(_.split(",").drop(1).map(_.split(" ").map(_.toDouble)).foldLeft(Array(0.0)){(x,y) => x.union(y)})
      (Y,X(0).drop(1))
  }
  )

  res86: RDD[(breeze.linalg.DenseVector[Double], Array[Array[Array[Double]]])]
  res87: RDD[(breeze.linalg.DenseVector[Double], Array[Array[Double]])]
  ```

  - `sc.textFile`：自动以**\n**为依据将长字符串划分为**Array[String]**，于是**lines**为两百长的**Array[String]**，每个元素为一行的数据（1个y+1000块x）
  ```

- 对lines再调用 `.split("\n")` ，就把Array[String]变成了两百长的**Array[Array[String]]**。其实\n是随便写的一个字符，lines里面根本没有\n。事实是把\n换成任一个非空格非数字非逗号字符都一样。总能达到目的。

```
Array("2342,324,","342,432").map(_.split("s"))

res40: Array[Array[String]] = Array(Array(2342,324,), Array(342,432))
```

- 调用Array[Array[String]]的时候要在用了take和drop之后才能用圆括号索引

- 或者如下也一样：

```
import breeze.linalg._
val lines=sc.textFile("LinearModel_0.txt")
val transLines0 = lines.map( s => {
    val Y = DenseVector(s.split(",")(0).toDouble)
    val X = s.split(",").drop(1).map(_.split(" ").map(_.toDouble))
    (Y,X)
}
)
val transLines1 = lines.map( s => {
    val Y = DenseVector(s.split(",")(0).toDouble)
    val X = s.split(",").drop(1).map(_.split("
").map(_.toDouble)).foldLeft(Array(0.0)){(x,y) => x.union(y)}
    (Y,X.drop(1))
}
)

res85: RDD[(breeze.linalg.DenseVector[Double], Array[Array[Double]])]
res86: RDD[(breeze.linalg.DenseVector[Double], Array[Double])]
```

- 与上面一种除了少一层没用的Array之外没什么差别（上面就是在下面的X上套了一层Array）
- 在第一个map中对X进行逐元素分析：
  - `s.split(",").drop(1)` ：传入String，出来Array[String]，每个元素是x的一块，即用空格分隔的5个x值
  - `.map(_.split(" ").map(_.toDouble))` ：传入String。**对上述每个x块进行split(" ")**，变为Array[String]，再Array[String].map(_.toDouble)，变成Array[Double]。这一步是String->Array[Double]
  - 最终得到String->Array[String]->Array[Array[Double]]
- 在第二个map中对X进行逐元素分析：（遇到map则拿元素出来分析）
  - `s.split(",").drop(1).map(_.split(" ").map(_.toDouble)` ：同上，String->Array[String]->Array[Array[Double]]
  - `.foldLeft(Array(0.0)){(x,y) => x.union(y)})` ：从前面的Array[Array[Double]]中**取出Array[Double]进行foldLeft拼接**，最终拼成了一个Array[Double]。如下：

    ```
    Array(Array(1.0,1.0),Array(2.0,2.0)).foldLeft(Array(0.0)){_.union(_)}

    res107: Array[Double] = Array(0.0, 1.0, 1.0, 2.0, 2.0)
    ```

    再把初值0.0drop掉就好
  - 最终是String->Array[String]->Array[Array[Double]]->Array[Double]

# EM算法

## 「 理论部分」

见平板笔记：LectureNotes_6的最后两面！迭代公式的推导要会啊！
上面有关概率论的推导也要会！
$\gamma$是来自第一个正态分布的概率，则$1-\gamma$是来自第二个正态分布的概率

## 「 代码部分」

### 生成高斯混合模型随机数

```
import java.util.concurrent.ThreadLocalRandom
val random=ThreadLocalRandom.current
val N=10000
//自然参数真实值
val MU1=5.0
val Sig1=2.0  //第一个正态分布
val MU2=0.0
val Sig2=1.0  //第二个正态分布
val P=0.2 //x来自第一个分布的概率
val data=Array.ofDim[Double](N)
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
    }
}
```

### 对该高斯混合模型的自然参数进行估计

- 防止直接产生的随机数无法序列化带来的问题：

```
val NumOfSlaves=5
var ParData=sc.parallelize(data,NumOfSlaves)
val ParDataStr=ParData.map("%.4f" format _)
val ParData=ParDataStr.map(_.toDouble)
```

- 取初值：初值只要大小关系准确就行，不要求接近真实值

```
val InitialP=0.5
var Nk=N.toDouble*InitialP //第一个样本的初始数
var EstMu1=ParData.reduce((x,y) => (x+y))/N.toDouble //第一个分布mu的初始值
var EstSig1=math.sqrt(ParData.map(x => x*x).reduce((x,y)=> x+y)/N.toDouble-
EstMu1*EstMu1) //Ex^2-mu^2

//稍微调整一下，使得初值大小关系准确就行！
var EstMu2=EstMu1-1.0
var EstSig2=EstSig1

var Diff=1.0//这里用的1-范数
val eps=0.001
var OldEstMu1=0.0
var OldEstMu2=0.0
```

```
var OldEstSig1=0.0
var OldEstSig2=0.0
//循环到第几次的指示变量
var ii=0
```

- 主要的循环：

```
do{
    ii+=1
    OldEstMu1=EstMu1
    OldEstMu2=EstMu2
    OldEstSig1=EstSig1
    OldEstSig2=EstSig2 //将上一轮的参数都存下来，待会要用
    //计算充分统计量是唯一可以并行的步骤
    var SufficientStatistics=ParData.map(line => {  //line是来自混合正态分布的
一个随机数
        val x2= -math.pow((line-EstMu1)/EstSig1,2)/2.0
        val x1= -math.pow((line-EstMu2)/EstSig2,2)/2.0  //它这里取名怪，是反的
        val gamma=Nk*EstSig2/(Nk*EstSig2+(N-Nk)*EstSig1*math.exp(x1-x2)) //
笔记最后一面推导了这个公式
        (line,gamma,line*gamma,line*line*gamma,1-gamma,line*(1-
gamma),line*line*(1-gamma))
        //这里是返回值，是一个元组，是mapreduce中常用的传递信息的方式
    })//返回值的含义分别为：样本x，来自第一个分布的gamma，sum(gamma_i1*x_i)，
sum(gamma_i1*x_i^2)，后面都是对分布2的
    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))//得
到了充分统计量（元组没法直接做加法，所以必须这样写）
    //用迭代公式更新参数
    Nk=Results._2
    EstMu1=Results._3/Nk
    EstSig1=math.sqrt(Results._4/Nk-EstMu1*EstMu1)//笔记里有推导
    EstMu2=Results._6/Results._5
    EstSig2=math.sqrt(Results._7/Results._5-EstMu2*EstMu2)
    //各参数的变化绝对值之和
    Diff=math.abs(EstMu1 - OldEstMu1)+math.abs(EstMu2-OldEstMu2)
    Diff+=math.abs(EstSig1 - OldEstSig1)+math.abs(EstSig2-OldEstSig2)
}while(Diff>eps)  //大于eps时继续运行
```

  - **SufficientStatistics**的map代码块的返回值的意义分别为：
    - $x, \gamma, \gamma x, \gamma x^2, \ldots$ (即对向量的每个分量都进行了map变换)
  - **Results**就是把上面各向量的分量都加了起来，即为：
    - $\sum x_i, \sum \gamma_i, \sum \gamma_i x_i, \sum \gamma_i x_i^2, \ldots$

# 「借助二分法运用逆累计分布法」

## 理论

若分布函数F已知，但是反函数不好求，则只需解方程$F(X) = U(0,1)$则可，解出的X则为服从分布F的随机数。解方程的过程运用二分法。
下面用F等于标准正态分布为例。

## 代码

```
import breeze.stats.distributions._
val myNormal = new Gaussian(0.0, 1.0)  //以标准正态分布为例
def f(x: Double, U: Double): Double={myNormal.cdf(x)-U} //要求这个方程的根，即
F(X)-U(0,1)=0的根

//下面是二分法解方程的函数
def BiSec(a: Double, b: Double, U: Double, eps: Double): Double={

var a0 = a;
var b0 = b;
var mid = (a0+b0)/2.0;
var result = 0.0;

if((b0-a0)<eps)
{
    result = mid;
}else
{
       while((b0-a0)>=eps)
       {
                if(f(a0,U)*f(mid,U)<0)
                {
                          b0 = mid;
                }else
                {
                          a0 = mid;
                }

                mid = (a0+b0)/2.0;
       }

       result = mid;
}

result

}

val test=0.234
val X = BiSec(-1000.0, 1000.0, test, 0.0000001)
myNormal.cdf(X) //发现结果极其接近
```

再生成随机数：

```
import java.util.concurrent.ThreadLocalRandom
var r=ThreadLocalRandom.current

(0 until 100).map(s => r.nextDouble).map(s=> BiSec(-1000.0, 1000.0, s,
0.0000001))
```

# ADMM算法

## 「理论部分」

平板笔记：LectureNotes_7

- 具体的更新步骤如下：
  - (1)更新$\beta$

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

  - (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$\alpha$

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

# 「代码部分」

- 准备工作

```
import org.apache.spark.mllib.linalg.distributed._
//这个包含了IndexedRow
import org.apache.spark.mllib.linalg.{Vectors,Matrix =>
sparkMatrix,DenseMatrix => sparkDenseMatrix}
import breeze.linalg._  //为了给矩阵求逆
import breeze.stats.distributions._
import scala.math._

//要写一个转化函数，把sparkmatrix变成DenseMatrix
def tobreezematrix(tmp1: sparkMatrix):DenseMatrix[Double]={
    var tmp2:DenseMatrix[Double]=DenseMatrix.zeros[Double]
(tmp1.numRows,tmp1.numCols) //调用行数列数，DenseMatrix里用的是.rows, .cols
    for (i <- 0 until tmp1.numRows){
        for (j <- 0 until tmp1.numCols){
            tmp2(i,j)=tmp1(i,j)
        }
    }
    tmp2
}
```

- 参数初始化

```
val N=sc.broadcast(10000) //X行数
val r1=sc.broadcast(2)  //给定了惩罚项的值
val P=sc.broadcast(100)  //X列数
val tau=sc.broadcast(0.5)
val iter=sc.broadcast(20)  //控制迭代次数
var indeces=0 until N.value  //行的标签，不能1 to N.value，这样X的维数会多1
var Beta=sc.broadcast(DenseVector.ones[Double](P.value))//待估参数
var PallallelIndeces=sc.parallelize(indeces)
```

- **生成样本**且**各行编号**：

```
var sample_matrix=PallallelIndeces.map(s => {
    var p=100
    var norm_dist=new Gaussian(0,1)
    var x=new DenseMatrix(rows=1,cols=p,norm_dist.sample(p).toArray) //生成一个行向量（一个样本），即用随机数（记得toArray）去填充一个1*p的向量
    //按课件上的公式生成y
    var y=x*Beta.value+x(0,0)*norm_dist.draw()  //draw直接出来一个数，sample出来一个Vector，要得到数还要把它提出来。现在y是一个1*1的DenseVector
    //下面组装我们要的矩阵
    Array(s.toDouble).union(x.toArray).union(Array(y(0))) //x左一个，右一个
})
sample_matrix.persist()//老师说不考这个
```

  - 用union组装各行：x的左边是行的编号，右边是y值

- X、Y分家

```
var sample_x=sample_matrix.map(f => IndexedRow(f.take(1)
(0).toLong,Vectors.dense(f.drop(1).dropRight(1))))
var sample_y=sample_matrix.map(f => IndexedRow(f.take(1)
(0).toLong,Vectors.dense(f.drop(P.value+1))))

var sample_x_indexedrowmatrix=new IndexedRowMatrix(sample_x)
var sample_y_indexedrowmatrix=new IndexedRowMatrix(sample_y)

var
x=sample_x_indexedrowmatrix.toBlockMatrix(rowsPerBlock=N.value/10,colsPerBl
ock=P.value/10)
var
y=sample_y_indexedrowmatrix.toBlockMatrix(rowsPerBlock=N.value/10,colsPerBl
ock=1)
```

  - sample_matrix是RDD[Array[Double]]，map里面为Array[Double]，**take(1)之后仍为Array[Double]**，再索引一下才是Double。（此处不要.take(1)也可以啊）

- 查看（LocalMatrix的维数）的方法：`locmat.numRows`，`locmat.numCols`

- 初始化ADMM算法：

```
//初始化ADMM算法
var i:Int=0
var diff:Double=1.0
val y_local=y.toLocalMatrix()  //block to local
var beta_hat,beta_old:DenseMatrix[Double]=DenseMatrix.zeros[Double]
(P.value,1) //这里一口气定义了两个同形的变量!
var z:DenseMatrix[Double]=DenseMatrix.zeros[Double](N.value,1)
var alpha: DenseMatrix[Double]=DenseMatrix.zeros[Double](N.value,1)
//下面是一个求逆的过程：(X'X)^{-1}
val tmp1=x.transpose.multiply(x).toLocalMatrix() //block to local
var tmp2=tobreezematrix(tmp1) //local to breeze
val beta_fixed=inv(tmp2) //用breezematrix求逆
```

- 主要的循环：

```
while(i < iter.value && diff > pow(10,-6)){
    //按照题目给的更新顺序一个个写下去
    //更新beta
    beta_old=beta_hat
    //算出beta更新式中花括号里面的项
    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,col
sPerBlock=1))
    .toLocalMatrix() //变成local方便tobreeze
    //乘起来得到beta
    beta_hat=beta_fixed*tobreezematrix(beta_right)
    //更新z
    var tmp4=y
    .subtract(x.toIndexedRowMatrix.multiply(new
sparkDenseMatrix(P.value,1,beta_hat.toArray)).toBlockMatrix(rowsPerBlock=N.
value/10,colsPerBlock=1))
    .toLocalMatrix()  //变成local方便tobreeze

    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
    })
    //更新alpha
    alpha=alpha+(tobreezematrix(tmp4)-z).map(s => s*r1.value)
    diff=sum((beta_old-beta_hat).map(s => abs(s)))//scala.math包里的abs只能对
单个数做运算
    i += 1
}

sum(beta_hat)
```

  - 三个参数在更新完之后都是**DenseVector**
  - beta_right：前一块、后一块都是Local相乘出来的，再都转成Block来做加法
  - tmp4：x乘beta是在local中做的，再转成Block跟y去做减法。可用在z和$\alpha$的更新中。
  - z：tmp4+$\frac{\alpha_i^k}{r_1}$（乘常数用map来做，其余都是向量的按位运算），再 在map中分类讨论 （因为此处要对各分量进行判断，map中的s就是题目中那个看似多余的$\alpha_i^k$）
  - 注意理清迭代公式中各矩阵、向量的维数！
  - 关于**beta_old=beta_hat**的说明：
    - **var**的整体换值不会相互牵连，如上例。又见下例

```
var Oldbeta=DenseVector.zeros[Double](3)
var betahat=DenseVector.zeros[Double](3)
Oldbeta=betahat
betahat=DenseVector(1.0,2.0,3.0)
Oldbeta

res40: breeze.linalg.DenseVector[Double] = DenseVector(0.0, 0.0, 0.0)
```

    - 但是对变量的改值会相互牵连，如下：

```
var Oldbeta=DenseVector.zeros[Double](3)
var betahat=DenseVector.zeros[Double](3)
Oldbeta=betahat
betahat(0)=3.0
Oldbeta

res41: breeze.linalg.DenseVector[Double] = DenseVector(3.0, 0.0, 0.0)
```

补救方法为：

```
var Oldbeta=DenseVector.zeros[Double](3)
var betahat=DenseVector.zeros[Double](3)
Oldbeta(0 until Oldbeta.size):=betahat(0 until betahat.size)
betahat(0)=3.0
Oldbeta

res42: breeze.linalg.DenseVector[Double] = DenseVector(0.0, 0.0, 0.0)
```

# 数值计算方法

## 「随机梯度下降法」

### 理论

下面的程序产生了一个样本量为40的数据集，在数据集中的解释变量的个数是2000，响应变量取值为0或者1。对该数据集进行logit回归，并通过随机梯度下降法（用所有的数据，即设置miniBatchFrac=1.0）来估计参数。在调用runMiniBatchSGD()时，输入的数据points需要是**RDD格式**的，且数据中的每个记录需要为tuple格式**(Double,Vector)**；而输入变量Vectors.dense(new Array[Double(p)])是估计参数的初始值（随机生成即可）

### 代码

- 生成points：

```
import scala.util.Random
import scala.collection.JavaConverters._
import org.apache.spark.mllib.linalg.Vectors
import org.apache.spark.mllib.regression._
import org.apache.spark.mllib.optimization.
{GradientDescent,LogisticGradient,SquaredL2Updater}

val gradient=new LogisticGradient()
val updater=new SquaredL2Updater() //squared

val stepSize=0.1
val numIterations=10
val regParam=1.0
val miniBatchFrac=1.0
val n=40
val p=2000  //2000个解释变量
//下面生成n个样本，每个样本是1个响应+p个解释变量
//这里是完全随机生成的，是杂乱无章的
val points=sc.parallelize(0 until n,2).map{iter =>
    val random=new Random()
    val u=random.nextDouble()
    val y=if(u>0.5)1.0 else 0.0 //生成响应变量
```

```
    (y,Vectors.dense(Array.fill(p)(random.nextDouble())))
    //Array.fill里面填充随机数，里面的数是各不相同的（即nextDouble运行p次）。
}
```

- 主要的程序

```
val (weights,loss)=GradientDescent.runMiniBatchSGD( //核心函数
points,  //输入的数据
gradient,  //用来算损失函数的梯度（Loss func）
updater,  //变量选择方法
stepSize,  //步长
numIterations, //最大迭代次数
regParam,  //用来设置惩罚项的调和参数lambda
miniBatchFrac,  //每次循环用的样本占总体的比例
Vectors.dense(new Array[Double](p)) //权重beta的初始值,均为0
)
```

  - 得到**weight**是每个feature的权重的估计，loss是每次循环的**Stochastic Loss**
  - **numIterations**记录了总的循环次数，会发现比**loss.size**小1，因为会在尚未迭代的时候就算一次loss

# 「有限内存BFGS算法」

## 理论

将数据集分成训练集集和测试集，并利用训练集估计logit回归模型中的参数，然后在测试集上估计事件发生的概率。
同样地，输入的数据training需要是**RDD格式**的，且数据中的每个元素需要为tuple格式**(Double,Vector)**

## 代码

- 读数据与整理数据

```
import org.apache.spark.mllib.linalg.Vectors
import org.apache.spark.mllib.classification.LogisticRegressionModel
import org.apache.spark.mllib.optimization.
{LBFGS,LogisticGradient,SquaredL2Updater}
import org.apache.spark.mllib.util.MLUtils //读数据用的
val data=MLUtils.loadLibSVMFile(sc,"sample_libsvm_data.txt")
//文件中的数据：冒号左边是数据位置，冒号右边是数据的值
//data是RDD[LabeledPoint]
val numFeatures=data.take(1)(0).features.size
//为692
```

  - 对(RDD)也可以用**take**，出来一个**Array**
  - `LabeledPoint`,此处每一个features都代表一个样本，展示如下：

```
data.take(1)(0)//第一个labeledpoint

res1: 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
,188,189,207,208,209,210,211,212,213,214,215,216,217,235,236,237,238,239
,240,241,242,243,244,245,262,263,264,265,266,267,268,269,270,271,272,273
,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
,457,466,467,468,469,470,482,483,484,493,494,495,496,497,510,511,512,520
,521,522,523,538,539,540,547,548,549,550,566,567,568,569,570,571,572,573
,574,575,576,577,578,594,595,596,597,598,599,600,601,602,603,604,622,623
,624,625,626,627,628,629,630,651,652,653,654,655,656,657],
[51.0,159.0,253.0,159.0,50,...]))
```

    - 第一个数0.0是响应变量
    - 第二部分是解释变量，这里是一个Vector。
        - 这个特殊的元组的含义是：692是解释变量的个数，第一个中括号表示对应的非零数字的位置，第二个中括号表示对应的非零数字的值。
        - 对本例的 `.features` 用 `.toArray` ，会输出响应的稀疏向量（系数矩阵）。如 `data.take(5)(3).features.toArray` 。
  - `.features` 返回解释变量。
  - `.label` 返回响应变量
- 分测试集和训练集，并给训练集点加截距

```
val splits=data.randomSplit(Array(0.6,0.4),seed=11L)
val test=splits(1)
val training=splits(0).map(x =>
(x.label,MLUtils.appendBias(x.features))).cache() //输入的数据training需要是
RDD格式的，且数据中的每个元素需要为tuple格式(Double,Vector)
//给每个labeledpoint的解释变量加截距
//.cache()不考
```

  - `splits(0)` 是训练集， `splits(1)` 是测试集。
  - 分的原理是：遇到一个样本，生成u(0,1)随机数，若大于0.6归于test，小于0.6归于train。所以两个集合的样本规模比未必是6:4。
  - `MLUtils.appendBias(x.features)` 会给x.features的最后添上一个全1列，使得解释变量的数目+1（对本例而言，从692变成693了）
- 初始化参数

```
val numCorrections=10
val convergenceTol=1e-4
val maxNumIterations=20
val regParam=0.1
val initialWerightsWithIntercept=Vectors.dense(new Array[Double]
(numFeatures+1))
```

- 主要的程序

```
val (weightsWithIntercept,loss)=LBFGS.runLBFGS(
training,
new LogisticGradient(),
new SquaredL2Updater(),
numCorrections,
convergenceTol,
maxNumIterations,
regParam,
initialWerightsWithIntercept)
```

- 参数的意义和顺序：
  - 输入的数据
  - 用来算损失函数的梯度
  - 自变量选择方法
  - 步长
  - 最大迭代次数
  - 用来设置惩罚项的调和参数lambda
  - 每次循环用的样本占总体的比例
  - 权重beta的初始值,均为0

- 建立模型并测试

```
weightsWithIntercept.toArray.slice(0,numFeatures)  //变量的权重
weightsWithIntercept(numFeatures) //截距项
//用估计值建立模型（必须这样分开再合并来传入）
val model=new LogisticRegressionModel(
Vectors.dense(weightsWithIntercept.toArray.slice(0,numFeatures)),weightsWit
hIntercept(numFeatures))
//默认的是threshold=0.5
model.clearThreshold()//清除默认的阈值，现在是threshold=None
//测试
val ProbabilityAndLabels=test.map{point =>
val probability=model.predict(point.features)
(probability,point.label)}
//计算错误率（用threshold那一套）
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)}

error.reduce((x,y)=>x+y)
```

- 当threshold有值的时候，预测的概率大于threshold时认为事件发生（Y=1），否则认为事件没发生（Y=0）
- 当threshold=None的时候，会直接输出事件发生的预测概率，即P{Y=1}
- 用 `model.predict(features)` 来得到预测的概率
- LogisticRegressionModel：用于响应变量是离散变量的时候

# 自举法

## 「自由自举法」

## 理论部分

### 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$\epsilon$是模型误差。

考虑以下的**自由自举法**的步骤:

1. 首先得到$\beta_0$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。

## 代码部分

- BFGS估计得到权重的岭估计量$\hat{\beta}_0$

```
import breeze.linalg._
import org.apache.spark.mllib.regression.LabeledPoint
import org.apache.spark.mllib.regression.LinearRegressionWithSGD
import org.apache.spark.mllib.regression.LinearRegressionModel
import org.apache.spark.mllib.optimization.
{LBFGS,LeastSquaresGradient,SquaredL2Updater}
import org.apache.spark.mllib.linalg.Vectors
import breeze.numerics._
import breeze.stats.distributions._
import java.util.concurrent.ThreadLocalRandom
import org.apache.spark.mllib.util.MLUtils

val data=MLUtils.loadLibSVMFile(sc,"sample_linear_regression_data.txt")
//这回的label是一个连续的变量了，第二个括号中的10是不为零的变量个数，第一个中括号为位置，
第二个中括号为值

val training=data.map(x=> (x.label,MLUtils.appendBias(x.features))).cache()

val numCorrections = 10
val convergenceTol = 1e-4
val maxNumIterations = 20
val regParam=1.0
val initialWeightsWithIntercept=Vectors.dense(new Array[Double]
(numFeatures+1))

val (weightsWithIntercept0, loss) = LBFGS.runLBFGS(
```

```
training,
new LeastSquaresGradient(),
new SquaredL2Updater(),
numCorrections,
convergenceTol,
maxNumIterations,
regParam,
initialWeightsWithIntercept)

//求得岭估计量
val coefficients = DenseVector(weightsWithInterceptO.toArray.slice(0,
weightsWithInterceptO.size - 1))
val Intercept = weightsWithInterceptO(weightsWithInterceptO.size - 1)
```

- **计算残差**

```
var lines = data.map(line => {
    val fitted = DenseVector(line.features.toArray).t*coefficients +
Intercept
    val residual = line.label - fitted  //epshat=y-yhat
    LabeledPoint(residual, line.features)
})
```

  - 关键：把残差放到**LabeledPoint.label**的位置上传出去

- **生成伪响应变量观测值**

```
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) //继续
这样传
})
```

  - 简言之，即是 `fitted+U(0,1)*(label-fitted)`

- 用伪响应变量观测值对权重进行估计

```
val training = transit.map(x => (x.label,
MLUtils.appendBias(x.features))).cache() //加个截距

val (weightsWithIntercept, loss) = LBFGS.runLBFGS(
training,
new LeastSquaresGradient(),
new SquaredL2Updater(),
numCorrections,
convergenceTol,
maxNumIterations,
regParam,
initialWeightsWithIntercept)
```

- 对2~4步重复若干次，见第五次作业

# 「子集合自举法」

## 理论部分

### 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$\hat{\xi}_k^*, k = 1,...,K$
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。

本例中，我们从标准正态分布产生两百万个随机数，然后从中取样约8000个数据出来用于得到该分布二阶中心矩的自举估计

## 代码部分

- 产生数据并用key将其划分成两个不相交的部分

```
import scala.collection.JavaConverters._
import scala.util.Random
import org.apache.spark.rdd.PairRDDFunctions
import org.apache.spark.HashPartitioner

val n=2000000
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  //将数据分成不相交的两份（两个key）
    (mykey,random.nextGaussian()) //键值对
}).partitionBy(new HashPartitioner(2)).persist()
```

  - 后面的partitionBy那些是将不同key的数据分别放到两个节点上去

- 在不相交的两份中分别抽出一个大小约为4000的子集

```
val K=points.partitions.size  //数据被分成了不相交的K份（K种key）
val SampleFractions=List((1,0.004),(2,0.004)).toMap
val SampledPoints=points.sampleByKey(false,SampleFractions)//约为8000个

val N=SampledPoints.mapValues(x => 1).reduceByKey((x,y)=>x+y)
```

  - **sampleByKey**并非精确抽样
  - N是每个子集的样本量，是**RDD[(Int, Int)]**，都约为4000

- 对每个子集用自举法

```
val BootstrapFractions=List((1,1.0),(2,1.0)).toMap//抽100%
val
BootstrappedPoints=SampledPoints.sampleByKeyExact(true,BootstrapFractions)
```

  - 分别进行有放回的100%抽样

- 在每个子集上分别对参数进行估计

```
val EstBootSum=BootstrappedPoints.reduceByKey((x,y)=>x+y)
val EstBootMu=EstBootSum.join(N).mapValues(x=>x._1/x._2)//两个估计量分别除以对应的N，分别得到两子集均值的估计量

val UpdatedPoints=BootstrappedPoints.join(EstBootMu) //每个Values的后面缀一个自己的组均值
val Est2thMomSum=UpdatedPoints.mapValues(x => (x._1-x._2)*(x._1-x._2)).reduceByKey((x,y)=> x+y)
//减组均值再平方，再分别加起来
val Est2thMom=Est2thMomSum.join(N).mapValues(x => x._1/x._2)
//分别除以组规模，K个子集得到K=2的估计量
```

- `join` 函数可以将两个 `成对RDD` 数据（Paired RDD，其每行的数据为(Key, Value)的tuple类型）按照相同的Key值进行匹配，匹配之后每个元素的Values部分都会变成元组（即(Key, Values_ori )-> (Key, (Values_1, Values_2))
- `mapValues` 是忽略Key的存在的，只对Values进行map且不考虑Key
- EstBootMu这个量的Values部分是两子集均值的估计值

- 对k个估计再平均

```
val MetaEst2thMom=Est2thMom.values.reduce((x,y)=>x+y)/K
```

- `.values` 用来取出Key值之外的数据信息，即：RDD[(Key, Values)] -> RDD[Values]
- 或者这样 `Est2thMom.reduce((x,y) => (0,x._2+y._2))._2/K`，是一样的。