sparkMlib学习笔记

# 数据类型

## Local vector

* 稠密向量dense vector
* 稀疏向量 sparse vector

举例：稠密向量是显示出每个值，而稀疏向量或者矩阵只存储非零值及其位置

稠密向量[1.0, 0.0, 3.0]

稀疏向量(3, [0, 2], [1.0, 3.0])

以上是同一个向量的两种存储形式，是等价的

sparkMlib中提供的包是Vector, 有DenseVector 和SparseVector

**Demo：**

**import** **org.apache.spark.mllib.linalg.**{**Vector**, **Vectors**}

*// Create a dense vector (1.0, 0.0, 3.0).*

**val** dv**:** Vector = **Vectors**.dense(1.0, 0.0, 3.0)

*// Create a sparse vector (1.0, 0.0, 3.0) by specifying its indices and values corresponding to nonzero entries.*

**val** sv1**:** Vector = **Vectors**.sparse(3, **Array**(0, 2), **Array**(1.0, 3.0))

*// Create a sparse vector (1.0, 0.0, 3.0) by specifying its nonzero entries.*

**val** sv2**:** Vector = **Vectors**.sparse(3, **Seq**((0, 1.0), (2, 3.0))

## Labeled point

带标签数据类型是sparkMlib中监督学习最基本的数据类型。数据存储格式为

LabeledPoint(label:double,vector:Vector)

对于二分类来说，label为0或者1，对于多分类来讲，label从0开始，这里label为double类型，因此既可以做分类，也可以用来做回归。

**Demo:**

**import** **org.apache.spark.mllib.linalg.Vectors**

**import** **org.apache.spark.mllib.regression.LabeledPoint**

*// Create a labeled point with a positive label and a dense feature vector.*

**val** pos **=** **LabeledPoint**(1.0, **Vectors**.dense(1.0, 0.0, 3.0))

*// Create a labeled point with a negative label and a sparse feature vector.*

**val** neg **=** **LabeledPoint**(0.0, **Vectors**.sparse(3, **Array**(0, 2), **Array**(1.0, 3.0)))

此外，sparkMlib支持读取LIBSVM格式数据，该格式如下：

label index1:value1 index2:value2 ...

在存储稀疏矩阵时非常有效。[MLUtils.loadLibSVMFile](http://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.mllib.util.MLUtils$)用来读取LIBSVM格式数据

**import** **org.apache.spark.mllib.regression.LabeledPoint**

**import** **org.apache.spark.mllib.util.MLUtils**

**import** **org.apache.spark.rdd.RDD**

**val** examples**:** RDD[LabeledPoint] **=** **MLUtils**.loadLibSVMFile(sc, "data/mllib/sample\_libsvm\_data.txt")

## Local matrix

包含系数矩阵和稠密矩阵，存储都是列优先，按列存储

Dense matrix基本格式：

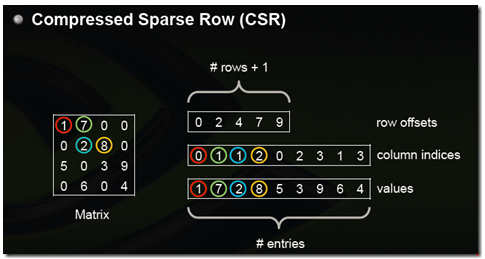
**Matrices**.dense(row:Int, colume:Int,**array:Array[double]**))

Sparse matrix基本格式：

**Matrices**.sparse(3, 2, **Array**(0, 1, 3), **Array**(0, 2, 1), **Array**(9, 6, 8))

稀疏矩阵用Compressed Sparse Column (CSC) 格式存储

Compressed Sparse Row(CSR)存储格式如下：



需要三类数据来表达：数值，列号，以及行偏移

类似地，CSC按列压缩，以上述矩阵为例：

Values：        [1 5 7 2 6 8 3 9 4]

Row Indices：[0 2 0 1 3 1 2 2 3]

Column Offsets：[0 2 5 7 9]

**Demo:**

**import** **org.apache.spark.mllib.linalg.**{**Matrix**, **Matrices**}

*// Create a dense matrix ((1.0, 2.0), (3.0, 4.0), (5.0, 6.0))*

**val** dm**:** Matrix = **Matrices**.dense(3, 2, **Array**(1.0, 3.0, 5.0, 2.0, 4.0, 6.0))

*// Create a sparse matrix ((9.0, 0.0), (0.0, 8.0), (0.0, 6.0))*

**val** sm**:** Matrix = **Matrices**.sparse(3, 2, **Array**(0, 1, 3), **Array**(0, 2, 1), **Array**(9, 6, 8))

## Distributed matrix

分布式矩阵格式（row:LongType,column:Longtype,value:double），分布式存储在一个或者多个RDD上。主要有以下几种格式

### RowMatrix

RowMatrix是又RDD[Vector]生成的，可以计算列概括性统计量或者矩阵分解

**Demo: 矩阵QR分解**

**import** **org.apache.spark.mllib.linalg.Vector**

**import** **org.apache.spark.mllib.linalg.distributed.RowMatrix**

**val** rows**:** RDD[Vector] **=** ... *// an RDD of local vectors*

*// Create a RowMatrix from an RDD[Vector].*

**val** mat**:** RowMatrix = **new** **RowMatrix**(rows)

*// Get its size.*

**val** m **=** mat.numRows()

**val** n **=** mat.numCols()

*//QR decomposition*

**val** qrResult **=** mat.tallSkinnyQR(**true**)

### IndexedRowMatrix

IndexRowMatrix由RDD[IndexedRow]实例生成, IndexRowMatrix可以通过去掉行标签转换为RowMatrix.

**Demo：**

**import** **org.apache.spark.mllib.linalg.distributed.**{**IndexedRow**, **IndexedRowMatrix**, **RowMatrix**}

**val** rows**:** RDD[IndexedRow] **=** ... *// an RDD of indexed rows*

*// Create an IndexedRowMatrix from an RDD[IndexedRow].*

**val** mat**:** IndexedRowMatrix = **new** **IndexedRowMatrix**(rows)

*// Get its size.*

**val** m **=** mat.numRows()

**val** n **=** mat.numCols()

*// Drop its row indices.*

**val** rowMat**:** RowMatrix = mat.toRowMatrix()

### CoordinateMatrix

CoordinateMatrix的每一个entry由一个三元组构成

（row:Long,colume:Long,value:Double）

当数据量比较大有比较稀疏的时候，考虑用该种方式存储数据。

CoordinateMatrix由RDD[MatrixEntry]实例生成，并且可以通过toIndexedRowMatrix转换为IndexedRowMatrix。目前，sparkMlib还不支持其他计算。

**Demo:**

**import** **org.apache.spark.mllib.linalg.distributed.**{**CoordinateMatrix**, **MatrixEntry**}

**val** entries**:** RDD[MatrixEntry] **=** ... *// an RDD of matrix entries*

*// Create a CoordinateMatrix from an RDD[MatrixEntry].*

**val** mat**:** CoordinateMatrix = **new** **CoordinateMatrix**(entries)

*// Get its size.*

**val** m **=** mat.numRows()

**val** n **=** mat.numCols()

*// Convert it to an IndexRowMatrix whose rows are sparse vectors.*

**val** indexedRowMatrix **=** mat.toIndexedRowMatrix()

### BlockMatrix

BlockMatrix通常是由IndexedRowMatrix 或者 CoordinateMatrix通过 toBlockMatrix函数转换为RDD[MatrixEntry]来实现

# 基本统计知识

## 概括统计量、相关性

SparkMlib通过函数colStats为一个RDD[Vector]生成列概括统计量。[colStats()](http://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.mllib.stat.Statistics$) 函数返回一个[MultivariateStatisticalSummary](http://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.mllib.stat.MultivariateStatisticalSummary)实例, 可以计算列最大值、最小值、均值、方差、非零个数或者总记录条数。

**Demo:**

**import** **org.apache.spark.mllib.linalg.Vector**

**import** **org.apache.spark.mllib.stat.**{**MultivariateStatisticalSummary**, **Statistics**}

**val** observations**:** RDD[Vector] **=** ... *// an RDD of Vectors*

*// Compute column summary statistics.*

**val** summary**:** MultivariateStatisticalSummary = **Statistics**.colStats(observations)

println(summary.mean) *// a dense vector containing the mean value for each col*

println(summary.variance) *// column-wise variance*

println(summary.numNonzeros) *// number of nonzeros in each column*

Correlations 是两个数组之间的关系的计算，Spark 目前支持 Pearson 和 Spearman 两类相关性分析。Spark 通过 Statistics 类提供了相关性分析的大部分算法。支持两类数据类型的输入，分别是 RDD[Double] 和 RDD[Vector] 相应的输出分别是 Double 和 Matrix

Pearson 相关系数用来衡量两个数据（集）是否在同一个层面上。统计学依据数据的计量尺度将数据划分为四大类，即定距型数据（Interval Scale）、定序型数据（Ordinal Scale）、定类型数据（Nominal Scale）和定比型数据 (Ratio Scale)。Pearson 用来计算定距变量，相关系数的绝对值越大，相关性越强：相关系数越接近于1或-1，[相关度](http://baike.baidu.com/view/2483063.htm)越强，相关系数越接近于0，相关度越弱。

* 0.8-1.0 极强相关
* 0.6-0.8 强相关
* 0.4-0.6 中等程度相关
* 0.2-0.4 弱相关
* 0.0-0.2 极弱相关或无相关

**Demo:**

**import** **org.apache.spark.SparkContext**

**import** **org.apache.spark.mllib.linalg.\_**

**import** **org.apache.spark.mllib.stat.Statistics**

**val** sc**:** SparkContext = ...

**val** seriesX**:** RDD[Double] **=** ... *// a series*

**val** seriesY**:** RDD[Double] **=** ... *// must have the same number of partitions and cardinality as seriesX*

*// compute the correlation using Pearson's method. Enter "spearman" for Spearman's method. If a*

*// method is not specified, Pearson's method will be used by default.*

**val** correlation**:** Double = **Statistics**.corr(seriesX, seriesY, "pearson")

**al** data**:** RDD[Vector] **=** ... *// note that each Vector is a row and not a column*

*// calculate the correlation matrix using Pearson's method. Use "spearman" for Spearman's method.*

*// If a method is not specified, Pearson's method will be used by default.*

**val** correlMatrix**:** Matrix = **Statistics**.corr(data, "pearson")

## 分层抽样

sparkMlib提供了在RDD键值对上的两种分层抽样的方法，sampleByKey 和

sampleByKeyExact。在分层抽样中，key代表一个label,value值代表一个具体的属性值，例如key可是一个人或者一个文件的id,value值是一个人的具体年龄、性别或者一个文件的有多少个词等等。

sampleByKey方法需要预先给定一个期望的样本数量，然后以掷硬币的方式来决定哪些观测值被选中，而sampleByKeyExact是在置信度为99.99%下，对sampleByKey抽样结果随机抽样。

Stratified sampling 算法是直接集成到键值对类型 RDD[(K, V)] 的 sampleByKey 和 sampleByKeyExact 方法提供支持，无需通过额外的  spark.mllib 库来支持。

**Demo:**

**import** **org.apache.spark.SparkContext**

**import** **org.apache.spark.SparkContext.\_**

**import** **org.apache.spark.rdd.PairRDDFunctions**

**val** sc**:** SparkContext = ...

**val** data **=** ... *// an RDD[(K, V)] of any key value pairs*

**val** fractions**:** Map[K, Double] **=** ... *// specify the exact fraction desired from each key*

*// Get an exact sample from each stratum*

**val** approxSample **=** data.sampleByKey(withReplacement **=** **false**, fractions)

**val** exactSample **=** data.sampleByKeyExact(withReplacement **=** **false**, fractions)

// 假设存在以下数组，第一个数字是userId, 后面的数据是某种业务数据

val list = List(

(2147481832,23355149,1)，

(2147481832,97301062,1)，

(2147481832,21348702,1)，

(2147481832,54102337,1)，

(2147481832,16822060,1)，

(2147481832,11382119,1)，

(2147481832,85220256,1)，

(2147481832,20137598,1)，

(2147481832,48653889,1)，

(2147481832,91918798,1)，

...)

// 我们将它转成Map, 以 userId 为 key，其他数据为 value

val data = sc.parallelize(list.toSeq).map(x => (x.\_1,(x.\_2,x.\_3)))

// 然后抽取userId 的唯一值做为 fractions的 key，并且假设每个用户的被采样几率都是 80%.

valfractions = data.map(\_.\_1).distinct.map(x => (x,0.8)).collectAsMap

// 获得采样数据

val sampleData = data.sampleByKey(withReplacement = false,fractions)

sampleByKey 和 sampleByKeyExact 的区别在于 sampleByKey 并不对过滤全量数据，因此只得到近似值，而 sampleByKeyExtra 会对全量数据做采样计算，因此耗费大量的计算资源，但是结果会更准确。

## 假设检验

sparkMlib通过 Statistics 类来支持 Pearson's chi-squared （卡方检测），主要是比较两个及两个以上样本率( 构成比）以及两个分类变量的关联性分析。其根本思想就是在于比较理论频数和实际频数的吻合程度或拟合优度问题。卡方检测有两种用途，分别是拟合优度检验（Goodness of fit）以及“独立性检定”（independence）。

拟合优度检验需要输入一个Vector，而独立性检验需要输入一个矩阵。此外，SparkMlib还支持数据数据为RDD[LabeledPoint]格式，从而通过卡方检验进行特征选择。

**import** **org.apache.spark.SparkContext**

**import** **org.apache.spark.mllib.linalg.\_**

**import** **org.apache.spark.mllib.regression.LabeledPoint**

**import** **org.apache.spark.mllib.stat.Statistics.\_**

**val** sc**:** SparkContext = ...

**val** vec**:** Vector = ... *// a vector composed of the frequencies of events*

*// compute the goodness of fit. If a second vector to test against is not supplied as a parameter,*

*// the test runs against a uniform distribution.*

**val** goodnessOfFitTestResult **=** **Statistics**.chiSqTest(vec)

println(goodnessOfFitTestResult) *// summary of the test including the p-value, degrees of freedo// test statistic, the method used, and the null hypothesis.*

**val** mat**:** Matrix = ... *// a contingency matrix*

*// conduct Pearson's independence test on the input contingency matrix*

**val** independenceTestResult **=** **Statistics**.chiSqTest(mat)

println(independenceTestResult) *// summary of the test including the p-value, degrees of freedom...*

**val** obs**:** RDD[LabeledPoint] **=** ... *// (feature, label) pairs.*

*// The contingency table is constructed from the raw (feature, label) pairs and used to conduct*

*// the independence test. Returns an array containing the ChiSquaredTestResult for every feature*

*// against the label.*

**val** featureTestResults**:** Array[ChiSqTestResult] **=** **Statistics**.chiSqTest(obs)

**var** i **=** 1

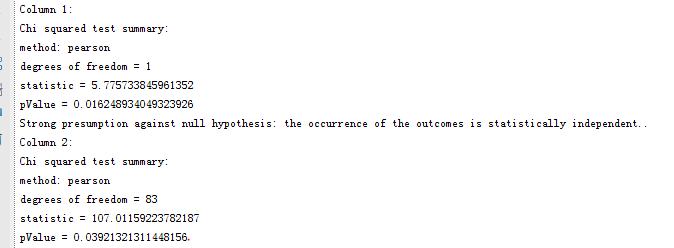
featureTestResults.foreach { result **=>**

println(s"Column $i:\n$result")

i += 1

} *// summary of the test*

打印结果：



假定检测的基本思路是，首先我们假定一个结论，然后为这个结论设置期望值，用实际观察值来与这个值做对比，并设定一个阀值，如果计算结果大于阀值，则假定不成立，否则成立。

根据以上表述，我们需要确定四个值：

1) 结论：结论一般是建立在零假设( Null Hypothesis)的基础上的。零假设即认为观测值与理论值的差异是由于随机误差所致。比如：“掷色子得到的各种结果概率相同”——这个结论显然我们认定的前提是即便不同也是随机因素导致。

2) 期望值：期望值也就是理论值，理论值可以是某种平均数，比如我们投掷120次色子，要维护结论正确，那么每个数字的出现理论值就应该都是20

3) 观测值：也就是实际得到的值

4) 阀值：阀值是根据自由度和显著性水平计算出来的（excel 中的 chiinv() 函数）。自由度=(结果选项数-1)x(对比组数-1)，比如我们将两组掷色子值做比较，那么自由度就是(6-1)x(2-1)=5。显著性水平(a)是原假设为正确的，而我们确把原假设当做错误加以拒绝，犯这种错误的概率，依据拒绝区间所可能承担的风险来决定，一般选择0.05或0.01。

最后就是计算卡方值：卡方值是各组 （观测值－理论值）^2/理论值  的总和。最后就是比较方差值和阀值。如果小于阀值则接受结论，否则拒绝结论。或者根据卡方值反算概率p值(excel 中的 chidist() 函数)，将它和显著性水平比较，小于则拒绝，大于则接受。

除了 Pearson's chi-squared 外 Statistics 还提供了 1-sample, 2-sided Kolmogorov-Smirnov 检测。

**Demo:**

**import** **org.apache.spark.mllib.stat.Statistics**

**val** data**:** RDD[Double] **=** ... *// an RDD of sample data*

*// run a KS test for the sample versus a standard normal distribution*

**val** testResult **=** **Statistics**.kolmogorovSmirnovTest(data, "norm", 0, 1)

println(testResult) *// summary of the test including the p-value, test statistic,*

*// and null hypothesis*

*// if our p-value indicates significance, we can reject the null hypothesis*

*// perform a KS test using a cumulative distribution function of our making*

**val** myCDF**:** Double => **Double** **=** ...

**val** testResult2 **=** **Statistics**.kolmogorovSmirnovTest(data, myCDF)

## 流显著性检验、随机数生成

显著性检验就是事先对总体形式做出一个假设，然后用样本信息来判断这个假设（原假设）是否合理，即判断真实情况与原假设是否显著地有差异。或者说，显著性检验要判断样本与我们对总体所做的假设之间的差异是否纯属偶然，还是由我们所做的假设与总体真实情况不一致所引起的。spark.mllib 实现了一个在线测试用以支持类似A/B测试这样的用例。

Spark 通过 StreamingTest 来提供支持，它接受两个设置参数：

* peacePeriod: 用于设置一段冷静期来忽略stream中开头的一段脏数据。
* windowSize: 采集窗口，如果给0，则不分批次一次处理全量数据

StreamingTest 接受的数据参数是 DStreaming[(Boolean, Double)] 类型，第一个 Boolean 表明该组数据是 control 组（false）还是 treatment 组（true），第二个 Double 是检测数据，所谓 control 和 treatment 组就是分别存放参照数据的组和修正数据的组

**Demo:**

**val** data **=** ssc.textFileStream(dataDir).map(line **=>** line.split(",") **match** {

**case** **Array**(label, value) **=>** **BinarySample**(label.toBoolean, value.toDouble)

})

**val** streamingTest **=** **new** **StreamingTest**()

.setPeacePeriod(0)

.setWindowSize(0)

.setTestMethod("welch")

**val** out **=** streamingTest.registerStream(data)

out.print()

RandomRDDs 是一个工具集，用来生成含有随机数的RDD，可以按各种给定的分布模式生成数据集，例如按正态分布生成随机数：

**Demo:**

**import** **org.apache.spark.SparkContext**

**import** **org.apache.spark.mllib.random.RandomRDDs.\_**  
**val** sc**:** SparkContext = ...

*// 生成100个服从正态分配N(0,1)的RDD〔Double〕，并且分布在 10 个分区中*

**val** u **=** normalRDD(sc, 100L, 10)

*// 转换为 N(1,4) 正态分布.*

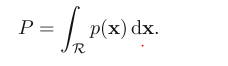
**val** v **=** u.map(x **=>** 1.0 + 2.0 \* x)

## 核密度估计

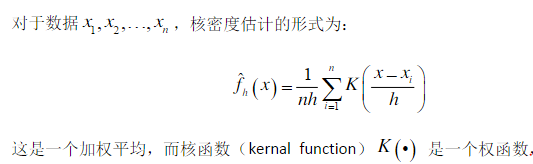
由给定样本点集合求解随机变量的分布密度函数问题是概率统计学的基本问题之一。解决这一问题的方法包括[参数估计](http://baike.baidu.com/view/123231.htm)和非参数估计。参数估计又可分为参数回归分析和参数[判别分析](http://baike.baidu.com/view/1394284.htm)。在参数回归分析中，人们假定数据分布符合某种特定的性态，如线性、可化线性或指数性态等，然后在[目标函数](http://baike.baidu.com/view/633433.htm)族中寻找特定的解，即确定[回归模型](http://baike.baidu.com/view/962884.htm)中的未知参数。在参数判别分析中，人们需要假定作为判别依据的、随机取值的数据样本在各个可能的类别中都服从特定的分布。经验和理论说明，参数模型的这种基本假定与实际的物理模型之间常常存在较大的差距，这些方法并非总能取得令人满意的结果。由于上述缺陷，Rosenblatt和Parzen提出了非参数估计方法，即核密度估计方法．由于核密度估计方法不利用有关数据分布的先验知识，对数据分布不附加任何假定，是一种从数据样本本身出发研究数据分布特征的方法

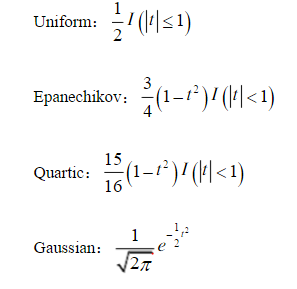
核密度估计（kernel density estimation）是在[概率论](http://zh.wikipedia.org/wiki/%E6%A6%82%E7%8E%87%E8%AE%BA)中用来估计未知的[密度函数](http://zh.wikipedia.org/wiki/%E5%AF%86%E5%BA%A6%E5%87%BD%E6%95%B8)，属於[非参数检验方法](http://zh.wikipedia.org/w/index.php?title=%E9%9D%9E%E5%8F%82%E6%95%B0%E6%A3%80%E9%AA%8C%E6%96%B9%E6%B3%95&act%3Cwbr%3Eion=edit&redlink=1)之一。核密度估计和直方图密度估计类似，根据观测数据对其假定的概率密度函数进行估计。但是直方图估计密度函数有很多局限性（1. 密度函数不光滑2、密度函数受子区间影响大3、数据是一维和两维使可以做，多维时受局限），核密度估计可以克服3的局限，并且当采用光滑的核函数时，就可以得到光滑的核密度函数。

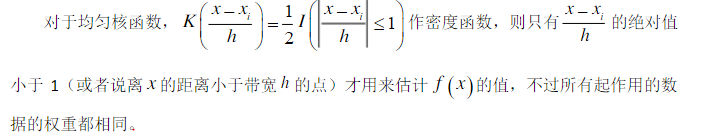
假设样本数据值在D维空间服从一个未知的概率密度函数，那么在区域R内的概率为：

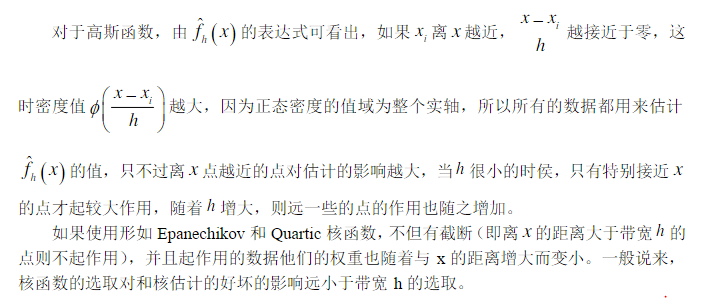


概率P的含义就为每个样本数据点落入区域R的概率为P.



我们通常考虑的核函数为关于原点对称的且积分为1的，一般采用下面四种核函数： 





**Demo:**

**import** **org.apache.spark.mllib.stat.KernelDensity**

**import** **org.apache.spark.rdd.RDD**  
**val** data**:** RDD[Double] **=** ... *// an RDD of sample data*

*// Construct the density estimator with the sample data and a standard deviation for the Gaussian*

*// kernels*

**val** kd **=** **new** **KernelDensity**()

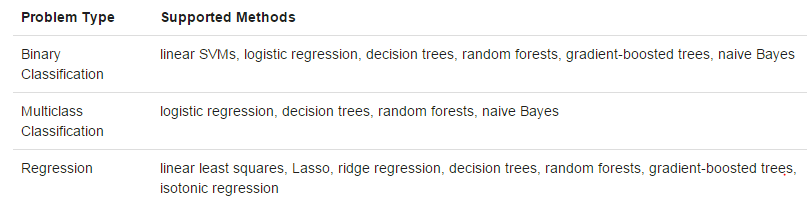
  .setSample(data)  
  .setBandwidth(3.0)

*// Find density estimates for the given values*

**val** densities **=** kd.estimate(**Array**(-1.0, 2.0, 5.0))

# 分类和回归

SparkMlib提供了以下算法：



## 线性模型

### 数学方程

很多机器学习算法都可以被解释为一个凸优化问题，目标函数为

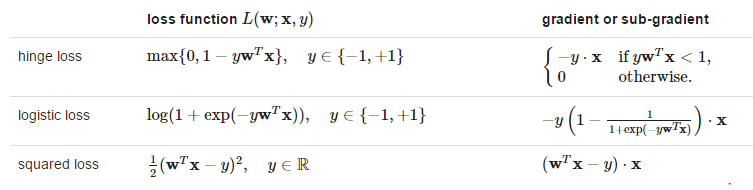


Xi是训练集，yi是对应的label, 如果是线性的，也即可以写为w和x/y的线性组合，我们称之为线性模型。

#### 损失函数

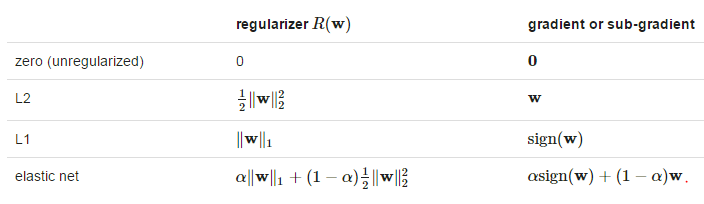
目标函数由两部分组成，一部分是正则化项，一部分是损失函数，在线性模型中，损失函数是关于w的凸函数。

下面列举出sparkMlib支持的损失函数及其响应梯度



#### 正则化项

正则化项是为了避免过拟合，sparkMlib支持以下正则化函数：



L2正则化通常比L1正则化更容易计算，L1正则化通常用于系数的矩阵，比如特征选择之类。 [Elastic net](http://en.wikipedia.org/wiki/Elastic_net_regularization)是L1正则化和L2正则化的组合。

#### 优化

在sparkMlib中，提供了两种凸优化求解最优目标函数的算法：SGD and L-BFGS，大多数算法的API支持Stochastic Gradient Descent (SGD)，一些同时支持 L-BFGS。

解决一个凸优化问题最基本的方法就是梯度下降。基本思想就是用最速下降法不断迭代，求得一个局部最优解。如果一个函数在每个点都是不可微，但仍然是凸函数，就可以用次梯度下降（sub-gradient）算法。但是，无论是梯度下降还是次梯度下降，都需要求出所有数据点的梯度值，计算偏导hession矩阵的计算代价很大。所以通常采用近似的算法，来避免直接计算hession矩阵。随机迭代stochastic gradient descent (SGD)和限制内存的SGD算法可以不用求解hession矩阵，从而减少计算成本。

在sparkMlib中，例如LogisticRegressionWithLBFGS，SVMWithSGD等等。具体算法参见11章节

### 分类

#### 逻辑回归

逻辑回归的损失函数：



对于一个给定的数据集x, 通过



来计算，其中，如果，则被认为是正例，否则为反例。这里0.5要根据具体的业务具体分析。可见，逻辑回归的输出结果不是0或者1，而是一个概率。这是与SVM等其他分类不同的。

在sparkMlib中，逻辑回归可以支持二分类和多分类问题，二分类中，输出为0和1，多分类为0、1、2、3等。0表示第一类。同时，在逻辑回归基础上，支持两种优化算法：mini-batch gradient descent and L-BFGS。

**Demo:**

**import** **org.apache.spark.SparkContext**

**import** **org.apache.spark.mllib.classification.**{**LogisticRegressionWithLBFGS**, **LogisticRegressionModel**}

**import** **org.apache.spark.mllib.evaluation.MulticlassMetrics**

**import** **org.apache.spark.mllib.regression.LabeledPoint**

**import** **org.apache.spark.mllib.linalg.Vectors**

**import** **org.apache.spark.mllib.util.MLUtils**

*// Load training data in LIBSVM format.*

**val** data **=** **MLUtils**.loadLibSVMFile(sc, "data/mllib/sample\_libsvm\_data.txt")

*// Split data into training (60%) and test (40%).*

**val** splits **=** data.randomSplit(**Array**(0.6, 0.4), seed **=** 11L)

**val** training **=** splits(0).cache()

**val** test **=** splits(1)

*// Run training algorithm to build the model*

**val** model **=** **new** **LogisticRegressionWithLBFGS**()

.setNumClasses(10)

.run(training)

*// Compute raw scores on the test set.*

**val** predictionAndLabels **=** test.map { **case** **LabeledPoint**(label, features) **=>**

**val** prediction **=** model.predict(features)

(prediction, label)

}

*// Get evaluation metrics.*

**val** metrics **=** **new** **MulticlassMetrics**(predictionAndLabels)

**val** precision **=** metrics.precision

println("Precision = " + precision)

*// Save and load model*

model.save(sc, "myModelPath")

**val** sameModel **=** **LogisticRegressionModel**.load(sc, "myModelPath")

#### SVM

SVM算法的损失函数为：



通常情况下，SVM采用L2正则化项。

**Demo:**

**import** **org.apache.spark.mllib.classification.**{**SVMModel**, **SVMWithSGD**}

**import** **org.apache.spark.mllib.evaluation.BinaryClassificationMetrics**

**import** **org.apache.spark.mllib.util.MLUtils**

*// Load training data in LIBSVM format.*

**val** data **=** **MLUtils**.loadLibSVMFile(sc, "data/mllib/sample\_libsvm\_data.txt")

*// Split data into training (60%) and test (40%).*

**val** splits **=** data.randomSplit(**Array**(0.6, 0.4), seed **=** 11L)

**val** training **=** splits(0).cache()

**val** test **=** splits(1)

*// Run training algorithm to build the model*

**val** numIterations **=** 100

**val** model **=** **SVMWithSGD**.train(training, numIterations)

*// Clear the default threshold.*

model.clearThreshold()

*// Compute raw scores on the test set.*

**val** scoreAndLabels **=** test.map { point **=>**

**val** score **=** model.predict(point.features)

(score, point.label)

}

*// Get evaluation metrics.*

**val** metrics **=** **new** **BinaryClassificationMetrics**(scoreAndLabels)

**val** auROC **=** metrics.areaUnderROC()

println("Area under ROC = " + auROC)

*// Save and load model*

model.save(sc, "myModelPath")

**val** sameModel **=** **SVMModel**.load(sc, "myModelPath")

SVMWithSGD.train()方法默认采用的是L2正则化，如果想使用L1正则化，可通过以下参数修正：

**import** **org.apache.spark.mllib.optimization.L1Updater**

**val** svmAlg **=** **new** **SVMWithSGD**()

svmAlg.optimizer.

setNumIterations(200).

setRegParam(0.1).

setUpdater(**new** L1Updater)

**val** modelL1 **=** svmAlg.run(training)

### 回归

#### 线性最小二乘/Lasso/岭回归

线性回归采用的损失函数为：



对于普通的线性最小二乘，就是正则化项为0，Lasso回归采用L1正则化，岭回归使用的是L2正则化。对所有这些线性方法，都采用的数均方误差



**Demo:**

**import** **org.apache.spark.mllib.regression.LabeledPoint**

**import** **org.apache.spark.mllib.regression.LinearRegressionModel**

**import** **org.apache.spark.mllib.regression.LinearRegressionWithSGD**

**import** **org.apache.spark.mllib.linalg.Vectors**

*// Load and parse the data*

**val** data **=** sc.textFile("data/mllib/ridge-data/lpsa.data")

**val** parsedData **=** data.map { line **=>**

**val** parts **=** line.split(',')

**LabeledPoint**(parts(0).toDouble, **Vectors**.dense(parts(1).split(' ').map(**\_**.toDouble)))

}.cache()

*// Building the model*

**val** numIterations **=** 100

**val** stepSize **=** 0.00000001

**val** model **=** **LinearRegressionWithSGD**.train(parsedData, numIterations, stepSize)

*// Evaluate model on training examples and compute training error*

**val** valuesAndPreds **=** parsedData.map { point **=>**

**val** prediction **=** model.predict(point.features)

(point.label, prediction)

}

**val** **MSE** **=** valuesAndPreds.map{**case**(v, p) **=>** math.pow((v - p), 2)}.mean()

println("training Mean Squared Error = " + **MSE**)

*// Save and load model*

model.save(sc, "myModelPath")

**val** sameModel **=** **LinearRegressionModel**.load(sc, "myModelPath")

岭回归和Lasso回归一样，默认参数输入可以查阅文档。例如

RidgeRegression {stepSize: 1.0, numIterations: 100, regParam: 0.01, miniBatchFraction: 1.0}.

#### 流回归

当数据以流的方式倒入的话，随着新数据的到达，不断更新参数。拟合回归模型是非常有用的。spark.mllib当前支持普通最小二乘（无正则化）的线性化回归模型。拟合方式和线下的一致，不同的是随着数据的注入，源源不断更新模型。

下面的Demo体现了如何从两个不同的txt文件流中，加载测试集合训练集。将流作为标签，通过第一个流在线建模，然后取预测第二个流。具体怎么使用流处理，参阅文档http://spark.apache.org/docs/latest/streaming-programming-guide.html#initializing

首先需要导入必须的包

**import** **org.apache.spark.mllib.linalg.Vectors**

**import** **org.apache.spark.mllib.regression.LabeledPoint**

**import** **org.apache.spark.mllib.regression.StreamingLinearRegressionWithSGD**

//创建流实例

**val** ssc **=** **new** **StreamingContext**(conf, **Seconds**(1))

//导入数据

**val** trainingData **=** ssc.textFileStream("/training/data/dir").map(**LabeledPoint**.parse).cache()

**val** testData **=** ssc.textFileStream("/testing/data/dir").map(**LabeledPoint**.parse)

//初始化模型

**val** numFeatures **=** 3

**val** model **=** **new** **StreamingLinearRegressionWithSGD**()

.setInitialWeights(**Vectors**.zeros(numFeatures))

//建立一个模型，并对测试集进行预测，并打印出来

model.trainOn(trainingData)

model.predictOnValues(testData.map(lp **=>** (lp.label, lp.features))).print()

ssc.start()

ssc.awaitTermination()

**注：**流的数据格式应该保存为(y,[x1,x2,x3])格式，y是label,x1.x2,x3是特征。流的好处就在于不断在线注入数据，不断更新模型，不断改进预测结果。所以注入的训练数据越多，预测越准确。

SparkMlib支持的线性模型：



## 朴素贝叶斯

朴素贝叶斯是一种多分类算法，设定的前提条件是特征之间是相互独立的。贝叶斯算法的基本思想是利用给定标签特征的条件概率分布，根据贝叶斯定理计算观测变量的的特征分布，然后用来预测。

sparkMlib提供了多项朴素贝叶斯和伯努利朴素贝叶斯（[multinomial naive Bayes](http://en.wikipedia.org/wiki/Naive_Bayes_classifier#Multinomial_naive_Bayes) and [Bernoulli naive Bayes](http://nlp.stanford.edu/IR-book/html/htmledition/the-bernoulli-model-1.html)），这种模型在文本分类中非常实用。每一个观测变量是一个文本，每个词代表一个特征，值为词频（多项朴素贝叶斯）或者该词是否在文本中出现0/1（伯努利朴素贝叶斯）。这就决定了朴素贝叶斯算法的值都必须是非负的。模型中可以调优的参数有：1.多项 2.拉普拉斯光滑因子lamada. 对于文本分来来讲，输入的特征向量通常是稀疏的，应该充分利用稀疏性，将稀疏向量作为输入，因为训练集只用一次，不需要cache.

在sparkMlib中，多项朴素贝叶斯的输入为：1.训练集RDD[ [LabeledPoint](http://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.mllib.regression.LabeledPoint)] 2.拉普拉斯光滑因子lambda, 输出就是贝叶斯模型，用来预测。

**Demo:**

**import** **org.apache.spark.mllib.classification.**{**NaiveBayes**, **NaiveBayesModel**}

**import** **org.apache.spark.mllib.linalg.Vectors**

**import** **org.apache.spark.mllib.regression.LabeledPoint**

**val** data **=** sc.textFile("data/mllib/sample\_naive\_bayes\_data.txt")

**val** parsedData **=** data.map { line **=>**

**val** parts **=** line.split(',')

**LabeledPoint**(parts(0).toDouble, **Vectors**.dense(parts(1).split(' ').map(**\_**.toDouble)))

}

**val** splits **=** parsedData.randomSplit(**Array**(0.6, 0.4), seed **=** 11L)

**val** training **=** splits(0)

**val** test **=** splits(1)

**val** model **=** **NaiveBayes**.train(training, lambda **=** 1.0, modelType **=** "multinomial")

**val** predictionAndLabel **=** test.map(p **=>** (model.predict(p.features), p.label))

**val** accuracy **=** 1.0 \* predictionAndLabel.filter(x **=>** x.\_1 == x.\_2).count() / test.count()

*// Save and load model*

model.save(sc, "target/tmp/myNaiveBayesModel")

**val** sameModel **=** **NaiveBayesModel**.load(sc, "target/tmp/myNaiveBayesModel")

## 决策树

在分类和回归中，决策树算法是非常流行的，可以从二分类过渡到多分类，无需升维。同时，目前比较流行的随机森林和Boosting都是基于决策树演变而来的。

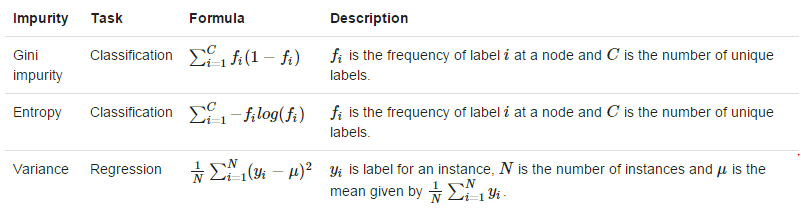
SparkMlib提供的决策树算法可以用来对连续特征和离散特征进行分类（二分类和多分类）和回归。

### 基本算法

决策树算法是一种贪心算法，每次切分使得数据变得最为有序。

#### 节点 impurity 和信息增益

节点impurity是在一个节点，使标签齐次性的一种方法，对于分类，当前的实现提供了两种impurity的方法：基尼impurity和熵，对于回归，用方差Variance来表示impurity程度，方差越大，表示数据间的差异越大。



**信息增益**用于表示，由父节点划分后得到子节点，所带来的impurity的下降，即有序性的增益



#### 切分candidates

1. 连续特征

对于连续的feature，其实就是进行范围划分，而划分的点就是split，划分出的区间就是bin。 理论上划分点是无数的，但是出于效率我们总要选取合适的划分点，有个比较常用的方法是取出训练集中该feature出现过的值作为划分点， 但对于分布式数据，取出所有的值进行排序也比较费资源，所以可以采取sample的方式。

1. 离散特征

对于离散的feature，比较简单，如果有m个值，最多[image](http://images.cnitblog.com/blog/312753/201412/081431454467953.png) 个划分，如果值是有序的，那么就最多m-1个划分   
比如年龄feature，有老，中，少3个值，如果无序有[image](http://images.cnitblog.com/blog/312753/201412/081431454467953.png)个，即3种划分，老|中，少；老，中|少；老，少|中   
但如果是有序的，即按老，中，少的序，那么只有m-1个，即2种划分，老|中，少；老，中|少

#### 停机准则

决策树算法的停机准则有以下几种情况：

1. 节点的深度等于训练模型中给定的最大深度maxDepth
2. 没有candidate可以获得比minInfoGain更大的信息增益
3. split candidate不能再切分孩子节点

#### 模型参数设置

* **algo：**做分类还是回归
* **numClasses：**分多少类（分类问题）
* **categoricalFeaturesInfo：**具体指哪些是分类变量，分为几类，在sparkMlib中，通常以Map格式呈现，eg:

  Map(0 -> 2, 4 -> 10)

指特征0是分类变量，对应的分类类型是2分类（0、1），特征4是分类变量，对应的分类类型是10分类（0-9）

* **maxDepth：**树的最大深度，树越深，精确度越高，但是代价越大，同时容易出现过拟合现象。
* **minInstancesPerNode：**节点可以继续split的前提是其孩子节点的训练样本数要大于等于该参数值
* **minInfoGain：**节点继续划分的前提是信息增益不能低于该参数设置
* **useNodeIdCache**
* **checkpointDir**
* **checkpointInterval**

#### 举例实现

**Demo(分类):**

**import** **org.apache.spark.mllib.tree.DecisionTree**

**import** **org.apache.spark.mllib.tree.model.DecisionTreeModel**

**import** **org.apache.spark.mllib.util.MLUtils**

*// Load and parse the data file.*

**val** data **=** **MLUtils**.loadLibSVMFile(sc, "data/mllib/sample\_libsvm\_data.txt")

*// Split the data into training and test sets (30% held out for testing)*

**val** splits **=** data.randomSplit(**Array**(0.7, 0.3))

**val** (trainingData, testData) **=** (splits(0), splits(1))

*//Empty categoricalFeaturesInfo indicates all features are continuous.*

**val** numClasses **=** 2

**val** categoricalFeaturesInfo **=** **Map**[Int, Int]()

**val** impurity **=** "gini"

**val** maxDepth **=** 5

**val** maxBins **=** 32

**val** model **=** **DecisionTree**.trainClassifier(trainingData, numClasses, categoricalFeaturesInfo, impurity, maxDepth, maxBins)

*// Evaluate model on test instances and compute test error*

**val** labelAndPreds **=** testData.map { point **=>**

**val** prediction **=** model.predict(point.features)

(point.label, prediction)

}

**val** testErr **=** labelAndPreds.filter(r **=>** r.\_1 != r.\_2).count().toDouble / testData.count()

println("Test Error = " + testErr)

println("Learned classification tree model:\n" + model.toDebugString)

**Demo(回归)**

**import** **org.apache.spark.mllib.tree.DecisionTree**

**import** **org.apache.spark.mllib.tree.model.DecisionTreeModel**

**import** **org.apache.spark.mllib.util.MLUtils**

*// Load and parse the data file.*

**val** data **=** **MLUtils**.loadLibSVMFile(sc, "data/mllib/sample\_libsvm\_data.txt")

*// Split the data into training and test sets (30% held out for testing)*

**val** splits **=** data.randomSplit(**Array**(0.7, 0.3))

**val** (trainingData, testData) **=** (splits(0), splits(1))

*// Train a DecisionTree model.*

*// Empty categoricalFeaturesInfo indicates all features are continuous.*

**val** categoricalFeaturesInfo **=** **Map**[Int, Int]()

**val** impurity **=** "variance"

**val** maxDepth **=** 5

**val** maxBins **=** 32

**val** model **=** **DecisionTree**.trainRegressor(trainingData, categoricalFeaturesInfo, impurity,

maxDepth, maxBins)

*// Evaluate model on test instances and compute test error*

**val** labelsAndPredictions **=** testData.map { point **=>**

**val** prediction **=** model.predict(point.features)

(point.label, prediction)

}

**val** testMSE **=** labelsAndPredictions.map{ **case** (v, p) **=>** math.pow(v - p, 2) }.mean()

println("Test Mean Squared Error = " + testMSE)

println("Learned regression tree model:\n" + model.toDebugString)

## 随机森林

随机森林是决策树系的一种，在分类和回归中应用非常广泛。为了解决过拟合的问题，随机森林结合多个决策树训练。和决策树一样，无论是连续型特征还是离散特征，都可以做分类（二分类、多分类）和回归。

### 基本思想

随机森林并行地训练一系列决策树，该算法在训练过程中引入随机性以便每个决策树都略有不同，结合每棵树的预测，从而减少总体预测的方差，提高测试集精度。

随机森林是一个组装（ensemble model）模型，内部的模型使用[决策树](http://www.cnblogs.com/bourneli/archive/2013/03/15/2961568.html)。基本思想是生成很多很多决策树（构成**森林**），最后由这些决策数一起投票决定最终结果。生成树的过程中，从行和列两个方向添加**随机**过程。行方向，在构建每棵树前，使用有放回抽样（称为Bootstrapping），得到训练数据。列方向，每次选择切分点时，对feature进行无放回随机抽样，得到一个feature子集，在当前节点上，只使用这些子集对应的数据计算最优切分点。这也是为什么此算法称为随机森林，是不是很直观。相比于单一决策树，随机森林有以下一些优点：

1.结果比较稳定，不容易出现过拟合；

2.Out-Of-Bag error评估模型效果，无需交叉检验；

3.可得到feature重要性。

随机森林是用随机的方式建立一个森林，森林里面有很多的决策树组成，随机森林的每一棵决策树之间是没有关联的。在得到森林之后，当有一个新的输 入样本进入的时候，就让森林中的每一棵决策树分别进行一下判断，看看这个样本应该属于哪一类（对于分类算法），然后看看哪一类被选择最多，就预测这个样本 为哪一类。

在随机森林的训练过程中，加入以下随机性（其实就是行抽样和列抽样）：

* 每次迭代随机抽样原始数据集，每课树可以有不同的训练集。
* 在树的每个节点上，考虑采用不同的特征split

除了这些随机性之外，随机森林训练多棵树和训练一棵树几乎是一样的。

按这种算法得到的随机森林中的每一棵都是很弱的，但是大家组合起来就很厉害了。我觉得可以这样比喻随机森林算法：每一棵决策树就是一个精通于某一个窄领域 的专家（因为我们从M个feature中选择m让每一棵决策树进行学习），这样在随机森林中就有了很多个精通不同领域的专家，对一个新的问题（新的输入数据），可以用不同的角度去看待它，最终由各个专家，投票得到结果。

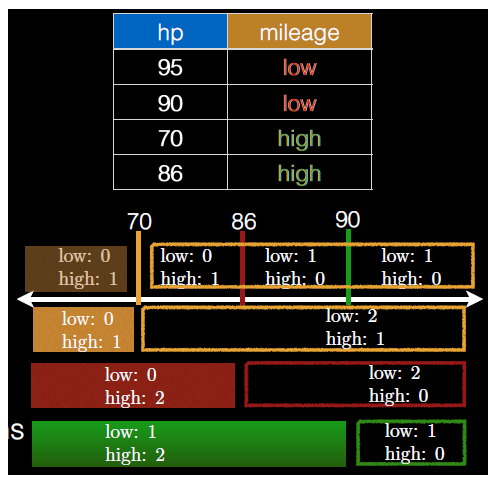
### 优化方案

1. 切分点抽样

此优化主要针对连续变量。一般的决策树是如何对连续变量进行切分点选择的。一般是先对feature进行排序，然后选取相邻两个数据之间的点作为切分点。如果在RDD上执行这个操作，不可避免会使用shuffle过程，此过程会带来大量的网络通讯。而且，一般RDD上的数据都很大，少则几百万，多则几亿到几十亿，甚至更多。在这样的数量级上进行排序操作，想想也是醉了。所以，为了避免排序操作，mllib通过抽样的方法，在样本上进行排序，并且根据样本，获取切分点。据spark团队反馈，使用此策略虽然牺牲了部分精度，但是在实际运用过程中，并没有带来过多的影响，模型效果可以接受。

1. feature装箱

根据抽样，得到切分点后，接下来是对feature进行装箱操作，箱子就是由相邻的样本切分点构成。箱子的个数是非常小的，一般实际中采用30个左右。计算每个箱子中不同种类的占比，可以很快计算出最优切分点



1. 分区统计

RDD分区中装箱数据统计后，可以通过reduce将每个分区的数据合并，得到总体的装箱数据，正是由于装箱数据可以合并，所以很好适应分布式环境。最后的合并也是一些统计数据，不会带来很大的网络开销

1. 逐层计算

### 模型参数

SparkMlib中，随机森林的参数除了需要决策树所需要配的参数外，还需要以下参数：

* **numTrees：森林中树的个数**
* 树越多，预测方差越小，测试精度越高
* 训练的时间和树的个数基本是线性相关的
* 所有从以上两个因素中找到一个平衡点
* **maxDepth：森林中每棵树的最大深度**
* 增加树的深度可以提高精度，但是增加训练时间，同时容易过拟合。
* 通常情况下，训练比较深的树，随机森林比单一决策树更好一些
* **subsamplingRate：训练每棵树随机抽取样本的比率**
* 默认情况下是1.0
* **featureSubsetStrategy：特征采样策略，auto表示算法自选**

### 实现

**Demo(分列)：**

**import** **org.apache.spark.mllib.tree.RandomForest**

**import** **org.apache.spark.mllib.tree.model.RandomForestModel**

**import** **org.apache.spark.mllib.util.MLUtils**

**val** data **=** **MLUtils**.loadLibSVMFile(sc, "data/mllib/sample\_libsvm\_data.txt")

**val** splits **=** data.randomSplit(**Array**(0.7, 0.3))

**val** (trainingData, testData) **=** (splits(0), splits(1))

**val** numClasses **=** 2

**val** categoricalFeaturesInfo **=** **Map**[Int, Int]()

**val** numTrees **=** 3 *// Use more in practice.*

**val** featureSubsetStrategy **=** "auto" *// Let the algorithm choose.*

**val** impurity **=** "gini"

**val** maxDepth **=** 4

**val** maxBins **=** 32

**val** model **=** **RandomForest**.trainClassifier(trainingData, numClasses, categoricalFeaturesInfo,

numTrees, featureSubsetStrategy, impurity, maxDepth, maxBins)

*// Evaluate model on test instances and compute test error*

**val** labelAndPreds **=** testData.map { point **=>**

**val** prediction **=** model.predict(point.features)

(point.label, prediction)}

**val** testErr **=** labelAndPreds.filter(r **=>** r.\_1 != r.\_2).count.toDouble / testData.count()

println("Test Error = " + testErr)

println("Learned classification forest model:\n" + model.toDebugString)

**Demo(回归)：**

**import** **org.apache.spark.mllib.tree.RandomForest**

**import** **org.apache.spark.mllib.tree.model.RandomForestModel**

**import** **org.apache.spark.mllib.util.MLUtils**

*// Load and parse the data file.*

**val** data **=** **MLUtils**.loadLibSVMFile(sc, "data/mllib/sample\_libsvm\_data.txt")

*// Split the data into training and test sets (30% held out for testing)*

**val** splits **=** data.randomSplit(**Array**(0.7, 0.3))

**val** (trainingData, testData) **=** (splits(0), splits(1))

**val** numClasses **=** 2

**val** categoricalFeaturesInfo **=** **Map**[Int, Int]()

**val** numTrees **=** 3 *// Use more in practice.*

**val** featureSubsetStrategy **=** "auto" *// Let the algorithm choose.*

**val** impurity **=** "variance"

**val** maxDepth **=** 4

**val** maxBins **=** 32

**val** model **=** **RandomForest**.trainRegressor(trainingData, categoricalFeaturesInfo,numTrees, featureSubsetStrategy, impurity, maxDepth, maxBins)

*/ Evaluate model on test instances and compute test error*

**val** labelsAndPredictions **=** testData.map { point **=>**

**val** prediction **=** model.predict(point.features)

(point.label, prediction)}

**val** testMSE **=** labelsAndPredictions.map{ **case**(v, p) **=>** math.pow((v - p), 2)}.mean()

println("Test Mean Squared Error = " + testMSE)

println("Learned regression forest model:\n" + model.toDebugString

## Gradient-Boosted树

决策树的一个缺点是容易出现过拟合，可以把利用融合的方式把各个弱模型集成起来，解决过拟合，提高模型的泛化能力。决策树和bagging 、boosting的思想结合在一起，诸如随机森林、GBDT，在数据挖掘中的预测分类、推荐广告以及搜索中的排序算法模型、搜索关键词的扩展推荐等等应用的非常广泛。

### bagging和boosting

**bagging思想：**

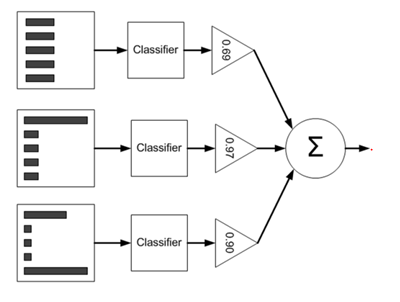
训练多个模型，利用每个模型进行投票，每个模型的权重是一样的，对于分类问题，取总票数最多的作为最终的类别，对于回归问题，取各模型的均值。利用多个弱分类器，集成一个比较强的分类器，随机森林就是采用的这种方式。

**Boosting思想：**

Boosting方式也是训练多个决策树模型，是一种迭代的算法模型。训练中，关注错分的样本，对于越是容易错分的样本，后续模型就要提高错分样本的权重，越要去重点关注这些数据。集成融合时，不同的模型有不同的权重，最终通过加权的方式融合为最终的模型。

Adaboost、GBDT采用的都是这种的思想

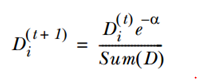
### Adaboost算法思想



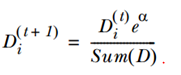
1. 初始的时候，样本的训练权重是一样。 通过一个弱分类器，得到样本的标签，与原始数据对比，就可能出现误差，如果某个样本预测错误，则它对应的错误值为该样本的权重，如果分类正确，则错误值为0. 最后累加5个样本的错误率之和，记为ε
2. 通过ε来计算该弱分类器的权重α



1. 通过α来计算训练下一个弱分类器样本的权重D，如果对应样本分类正确，则减小该样本的权重



如果样本分类错误，则增加该样本的权重，公式为



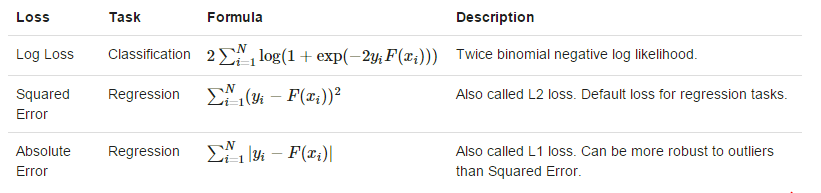
1. 循环1-3步骤，训练多个分类器

输入一个样本到训练好的每个弱分类中，则每个弱分类都对应一个输出标签，然后该标签乘以对应的α，最后求和得到值的符号即为预测标签值

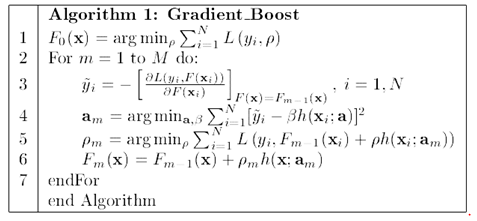
### Gradient Boosting（GBDT）

目前，sparkMlib支持的是GBDT算法，并且暂时只支持二分类和回归，不支持多分类。该算法的思想是，每一次建立模型是在之前建立模型损失函数的梯度下降方向。损失函数(loss function)描述的是模型的不靠谱程度，损失函数越大，则说明模型越容易出错（其实这里有一个[方差、偏差均衡](http://www.cnblogs.com/LeftNotEasy/archive/2010/12/19/mathmatic_in_machine_learning_2_regression_and_bias_variance_trade_off.html)的问题，但是这里就假设损失函数越大，模型越容易出错）。如果我们的模型能够让损失函数持续的下降，则说明我们的模型在不停的改进，而最好的方式就是让损失函数在其梯度（Gradient)的方向上下降。算法的每一步沿着损失函数下降最快的方向建立新的模型，这样使得算法在每一步均沿着下降最快的方向收敛。直到满足要求，建立满足要求的若干组合加权子模型。

损失函数：



算法伪代码：



### 模型参数

除了决策树所需要的参数外

* loss:
* **numIterations:迭代次数**
* **learningRate：通常不需要修改，如果模型不稳定，可以相应减小该参数**
* **algo：分类还是回归**

### 实现

**Demo(分类)：**

**import** **org.apache.spark.mllib.tree.GradientBoostedTrees**

**import** **org.apache.spark.mllib.tree.configuration.BoostingStrategy**

**import** **org.apache.spark.mllib.tree.model.GradientBoostedTreesModel**

**import** **org.apache.spark.mllib.util.MLUtils**

*// Load and parse the data file.*

**val** data **=** **MLUtils**.loadLibSVMFile(sc, "data/mllib/sample\_libsvm\_data.txt")

**val** splits **=** data.randomSplit(**Array**(0.7, 0.3))

**val** (trainingData, testData) **=** (splits(0), splits(1))

*// The defaultParams for Classification use LogLoss by default.*

**val** boostingStrategy **=** **BoostingStrategy**.defaultParams("Classification")

boostingStrategy.numIterations **=** 3 *// Note: Use more iterations in practice.*

boostingStrategy.treeStrategy.numClasses **=** 2

boostingStrategy.treeStrategy.maxDepth **=** 5

*// Empty categoricalFeaturesInfo indicates all features are continuous.*

boostingStrategy.treeStrategy.categoricalFeaturesInfo **=** **Map**[Int, Int]()

**val** model **=** **GradientBoostedTrees**.train(trainingData, boostingStrategy)

*// Evaluate model on test instances and compute test error*

**val** labelAndPreds **=** testData.map { point **=>**

**val** prediction **=** model.predict(point.features)

(point.label, prediction)}

**val** testErr **=** labelAndPreds.filter(r **=>** r.\_1 != r.\_2).count.toDouble / testData.count()

println("Test Error = " + testErr)

println("Learned classification GBT model:\n" + model.toDebugString)

**Demo(回归)：**

**import** **org.apache.spark.mllib.tree.GradientBoostedTrees**

**import** **org.apache.spark.mllib.tree.configuration.BoostingStrategy**

**import** **org.apache.spark.mllib.tree.model.GradientBoostedTreesModel**

**import** **org.apache.spark.mllib.util.MLUtils**

**val** data **=** **MLUtils**.loadLibSVMFile(sc, "data/mllib/sample\_libsvm\_data.txt")

**val** splits **=** data.randomSplit(**Array**(0.7, 0.3))

**val** (trainingData, testData) **=** (splits(0), splits(1))

*// Train a GradientBoostedTrees model.*

*// The defaultParams for Regression use SquaredError by default.*

**val** boostingStrategy **=** **BoostingStrategy**.defaultParams("Regression")

boostingStrategy.numIterations **=** 3

boostingStrategy.treeStrategy.maxDepth **=** 5

boostingStrategy.treeStrategy.categoricalFeaturesInfo **=** **Map**[Int, Int]()

**val** model **=** **GradientBoostedTrees**.train(trainingData, boostingStrategy)

**val** labelsAndPredictions **=** testData.map { point **=>**

**val** prediction **=** model.predict(point.features)

(point.label, prediction)}

**val** testMSE **=** labelsAndPredictions.map{ **case**(v, p) **=>** math.pow((v - p), 2)}.mean()

println("Test Mean Squared Error = " + testMSE)

println("Learned regression GBT model:\n" + model.toDebugString)

## 保序回归

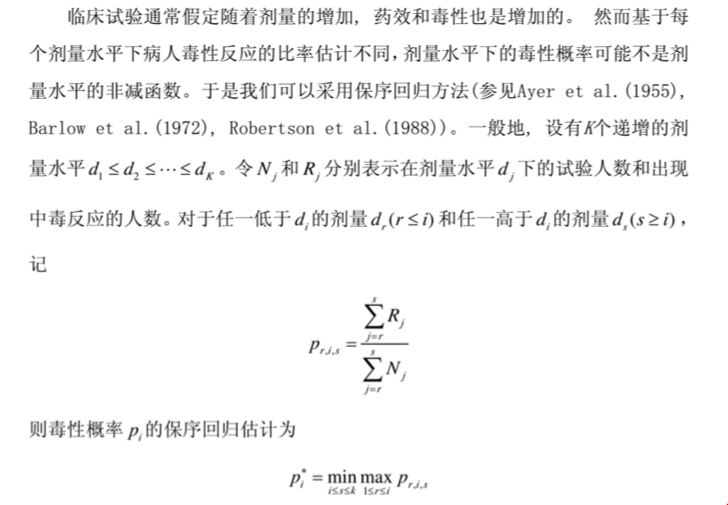
保序回归属于回归的一种，通常情况下，就是拟合一个向量，使得



这里

### 保序回归的背景

在药物剂量反应中，随着药剂量的增加，疗效或者副作用通常呈现一定的趋势。临床实验研究的主要目标之一就是评估药物在不同剂量下的毒性大小，并且建议一个病人最合适的剂量，即最大耐受剂量（MTD）。

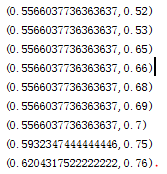


保序回归有PAVA算法、最大最小公式算法、MLS算法等等。spark.mllib支持的[pool adjacent violators algorithm](http://doi.org/10.1198/TECH.2010.10111)（PAVA）算法。了解算法详情，参阅

http://wenku.baidu.com/view/ade7744eee06eff9aef8079f.html?from=search

sparkMlib中，报序回归的输入是一个三元组RDD[tuple],并且tuple的每个元素都是Double类型。另外，算法中有一个参数isotonic，默认为true, 表明报序回归是单调递增还是单调递减的。

报序回归的模型其实是一个分段线性函数，当x等于【集合】，y取某值



**Demo：**

**import** **org.apache.spark.mllib.regression.**{**IsotonicRegression**, **IsotonicRegressionModel**}

**val** data **=** sc.textFile("data/mllib/sample\_isotonic\_regression\_data.txt")

*// Create label, feature, weight tuples from input data with weight set to default value 1.0.*

**val** parsedData **=** data.map { line **=>**

**val** parts **=** line.split(',').map(**\_**.toDouble)

(parts(0), parts(1), 1.0)

}

*// Split data into training (60%) and test (40%) sets.*

**val** splits **=** parsedData.randomSplit(**Array**(0.6, 0.4), seed **=** 11L)

**val** training **=** splits(0)

**val** test **=** splits(1)

*// Create isotonic regression model from training data.*

*// Isotonic parameter defaults to true so it is only shown for demonstration*

**val** model **=** **new** **IsotonicRegression**().setIsotonic(**true**).run(training)

*// Create tuples of predicted and real labels.*

**val** predictionAndLabel **=** test.map { point **=>**

**val** predictedLabel **=** model.predict(point.\_2)

(predictedLabel, point.\_1)

}

*// Calculate mean squared error between predicted and real labels.*

**val** meanSquaredError **=** predictionAndLabel.map { **case** (p, l) **=>** math.pow((p - l), 2) }.mean()

println("Mean Squared Error = " + meanSquaredError)

*// Save and load model*

model.save(sc, "target/tmp/myIsotonicRegressionModel")

**val** sameModel **=** **IsotonicRegressionModel**.load(sc, "target/tmp/myIsotonicRegressionModel")

# 协同过滤

协同过滤算法主要应用于推荐系统，按照数据使用，可以分为：

* 基于用户
* 基于商品
* 基于模型

按照模型来分，可分为：

* 最近邻模型：基于距离的协同过滤
* SVD分解：基于矩阵分解的协同过滤
* Graph: 基本社交网络的模型

## 协同过滤算法概述

### 基于用户（UserCF）---基于用户相似性

基于用户的协同过滤算法，是根据用户对不同商品的评分，评测用户间的相似性，基于用户间的相似性做出推荐。说白了，就是推荐给用户和他相似的用户喜欢的商品。举个例子，通过浏览及其购买商品的种类，推测A和B是相似的，那么我们就可以向A推荐B可能喜欢但是还没有购买的商品。

举例：



基于UserCF的基本思想：基于用户对物品的偏好，找到相邻邻居用户，然后将邻居用户喜欢的商品推荐给当前用户。

计算上，将一个用户对所有物品的偏好作为一个向量来计算用户之间的相似度，找到K邻居后，根据邻居的相似度权重以及他们对物品的偏好，预测当前用户没有偏好的未涉及物品，计算得到一个排序的物品列表作为推荐。

### 基于商品（ItemCF）---基于商品相似性

基于商品的协同过滤，通过用户对不同item的评分来评测item之间的相似性，基于item之间的相似性做出推荐。简单来将，就是给用户推荐和他之前喜欢的物品相似的物品。超市中的捆绑商品就是基于这种理论。

举例：



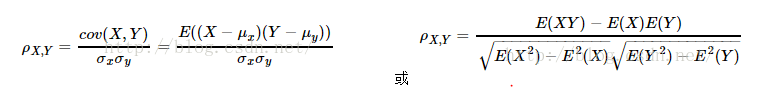
如图，有三个用户A、B、C和三件物品A、B、C，需要向用户C推荐物品。这里，由于用户A买过物品A和C，用户B买过物品A、B、C，用户C买过物品A，从用户A和B可以看出，这两个用户都买过物品A和C，说明物品A和C非常相似，同时，用户C又买过物品A，所以，将物品C推荐给用户C。

基于ItemCF的原理和基于UserCF类似，只是在计算邻居时采用物品本身，而不是从用户的角度，即基于用户对物品的偏好找到相似的物品，然后根据用户的历史偏好，推荐相似的物品给他。

       从计算角度，即将所有用户对某个物品的偏好作为一个向量来计算物品之间的相似度，得到物品的相似物品后，根据用户历史的偏好预测当前用户还没有表示偏好的物品，计算得到一个排序的物品列表作为推荐。

度量向量之间的相似度：距离的倒数、向量夹角、相关系数等。

皮尔森Pearson相关系数：



### 基于模型（ModelCF）

基于模型的协同过滤推荐就是基于样本的用户喜好信息，训练一个推荐模型，然后根据实时的用户喜好的信息进行预测，计算推荐。比如淘宝购物，当你搜索某一关键词时，会有很多店铺推荐，推荐机制就是根据你历史购买记录，推荐用户的偏好，然后根据实时查询条件，做相应推荐。

Spark MLlib当前支持基于模型的协同过滤，其中用户和商品通过一小组隐性因子进行表达，并且这些因子也用于预测缺失的元素。MLlib使用交替最小二乘法（ALS）来学习这些隐性因子。

#### 模型参数

* rank  模型中隐藏因子数
* iterations  算法迭代次数
* lambda    ALS中的正则化参数
* numBlocks  并行计算的block数
* implicitRrefs  使用显式反馈ALS变量或隐式反馈
* alpha    ALS隐式反馈变化率

数据格式： **Rating**（user:Int,product:Int,rate:Double）

#### 举例实现

**Demo1:**

**import** **org.apache.spark.mllib.recommendation.ALS**

**import** **org.apache.spark.mllib.recommendation.MatrixFactorizationModel**

**import** **org.apache.spark.mllib.recommendation.Rating**

**val** data **=** sc.textFile("data/mllib/als/test.data")

**val** ratings **=** data.map(**\_**.split(',') **match** { **case** **Array**(user, item, rate) **=>**

**Rating**(user.toInt, item.toInt, rate.toDouble)

})

*// Build the recommendation model using ALS*

**val** rank **=** 10

**val** numIterations **=** 10

**val** model **=** **ALS**.train(ratings, rank, numIterations, 0.01)//默认是显性

*// Evaluate the model on rating data*

**val** usersProducts **=** ratings.map { **case** **Rating**(user, product, rate) **=>**

(user, product)}

**val** predictions **=**

model.predict(usersProducts).map { **case** **Rating**(user, product, rate) **=>**

((user, product), rate)}

**val** ratesAndPreds **=** ratings.map { **case** **Rating**(user, product, rate) **=>**

((user, product), rate)

}.join(predictions)//设置成key-value，按照key值进行join操作

**val** **MSE** **=** ratesAndPreds.map { **case** ((user, product), (r1, r2)) **=>**

**val** err **=** (r1 - r2)

err \* err

}.mean()

println("Mean Squared Error = " + **MSE**)

如果更换数据源，想用隐式模型，可以用trainImplicit

**val** alpha **=** 0.01

**val** lambda **=** 0.01

**val** model **=** **ALS**.trainImplicit(ratings, rank, numIterations, lambda, alpha)

## 基于模型的协同过滤算法（电影推荐）

**电影推荐**

原始数据：

ratings.dat，数据格式：

UserID::MovieID::Rating::Timestamp

movies.dat，数据格式：

MovieID::Title::Genres(流派)

数据文件包含72000个用户对10000个电影的10 million 个打分。

## 交替最小二乘法alternating least squares（ALS）

ALS假设：打分矩阵是近似低秩的，即



可以把打分矩阵A看成是用户喜好矩阵U和产品特征矩阵V的乘积。

量化的目标就是通过U，V重构A所产生的误差。使用Frobenius范数



就是每个元素的重构误差的平方和。所以问题转换为了目标函数的优化问题。

ALS的目标函数不是凸函数，而是变量互相耦合在一起。我们把用户特征矩阵U和产品特这矩阵V固定其一，问题就变成了凸优化问题。交替固定U/V，求解该优化问题，就可以得到收敛解。并且，ALS对初始点不是很敏感，对最后结果影响不大

# 聚类

## K-mean算法

### 基本概念、模型参数

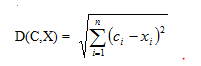
K-mean算法是聚类算法中最常用的一个。spark.mllib同时支持一种k-meanII算法，该算法是k-mean算法的变种。Spark.mllib使用算法模型参数：

* k: 聚类个数
* maxIterations 单次运行最大迭代次数
* initializationMode 表示初始聚类中心点的选择方式, 目前支持随机选择或者 K-means||方式。默认是 K-means||
* runs k-mean表示算法被运行的次数。K-means 算法不保证能返回全局最优的聚类结果，所以在目标数据集上多次跑 K-means 算法，有助于返回最佳聚类结果
* initializationSteps 决定k-means|| 算法的执行步数
* epsilon 代表聚类迭代收敛的阈值
* initialModel  初始化时，可选聚类中心的集合

聚类分析是一个无监督学习 (Unsupervised Learning) 过程, 一般是用来对数据对象按照其特征属性进行分组，经常被应用在客户分群，欺诈检测，图像分析等领域

K-means 算法也是一个迭代式的算法，其主要步骤如下:

* 第一步，选择 K 个点作为初始聚类中心
* 第二步，计算其余所有点到聚类中心的距离，并把每个点划分到离它最近的聚类中心所在的聚类中去。在这里，衡量距离一般有多个函数可以选择，最常用的是欧几里得距离 (Euclidean Distance), 也叫欧式距离。公式如下：



其中 C 代表中心点，X 代表任意一个非中心点

* 第三步，重新计算每个聚类中所有点的平均值，并将其作为新的聚类中心点
* 最后，重复 (二)，(三) 步的过程，直至聚类中心不再发生改变，或者算法达到预定的迭代次数，又或聚类中心的改变小于预先设定的阀值。

在实际应用中，K-means 算法有两个重要问题需要解决 ，并且对聚类的结果有比较大的影响。

* 聚类个数 K 的选择。

Mllib提供了选择k的工具。

Spark MLlib 在 KMeansModel 类里提供了 computeCost 方法，该方法通过计算所有数据点到其最近的中心点的平方和来评估聚类的效果。一般来说，同样的迭代次数和算法跑的次数，这个值越小代表聚类的效果越好。但是在实际情况下，我们还要考虑到聚类结果的可解释性，不能一味的选择使 computeCost 结果值最小的那个 K。

之后实例演示时，演示该方法的使用。

* 初始聚类中心点的选择。

k-mean算法对初始聚类中心的选择比较敏感。选择不同的聚类中心可能导致聚类结果的差异。

Spark MLlib K-means 算法的实现在初始聚类点的选择上，借鉴了一个叫 K-means||的类 K-means++ 实现。K-means++ 算法在初始点选择上遵循一个基本原则: 初始聚类中心点相互之间的距离应该尽可能的远。基本步骤如下:

* 第一步，从数据集 X 中随机选择一个点作为第一个初始点
* 第二步，计算数据集中所有点与最新选择的中心点的距离 D(x)
* 第三步，选择下一个中心点，使得



最大。

* 第四部，重复 (二),(三) 步过程，直到 K 个初始点选择完成。

sparkMlib在通常应用时，先调用 KMeans.train 方法对数据集进行聚类训练，这个方法会返回 KMeansModel 类实例，然后我们也可以使用 KMeansModel.predict 方法对新的数据点进行所属聚类的预测，这是非常实用的功能。

### 实现

**Demo:**

**import** **org.apache.spark.mllib.clustering.**{**KMeans**, **KMeansModel**}

**import** **org.apache.spark.mllib.linalg.Vectors**

**val** data **=** sc.textFile("data/mllib/kmeans\_data.txt")

**val** parsedData **=** data.map(s **=>** **Vectors**.dense(s.split(' ').map(**\_**.toDouble))).cache()

**val** numClusters **=** 2

**val** numIterations **=** 20

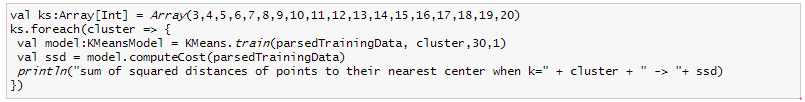
**val** clusters **=** **KMeans**.train(parsedData, numClusters, numIterations)

*// Evaluate clustering by computing Within Set Sum of Squared Errors*

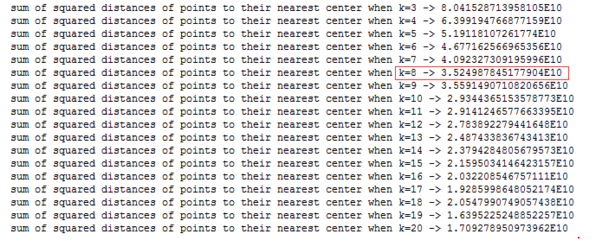
**val** **WSSSE** **=** clusters.computeCost(parsedData)

println("Within Set Sum of Squared Errors = " + **WSSSE**)

在实际情况中，我们可以通过computeCost方法选择一个最有的k



运行结果如下：



从运行结果可以看到，当 K=9 时，cost 值有波动，但是后面又逐渐减小了，所以我们选择 8 这个临界点作为 K 的个数。当然可以多跑几次，找一个稳定的 K 值。理论上 K 的值越大，聚类的 cost 越小，极限情况下，每个点都是一个聚类，这时候 cost 是 0，但是显然这不是一个具有实际意义的聚类结果。

## 高斯混合Gaussian mixture

高斯混合模型是将数据分成k个高斯分布，然后加权和。每个高斯分布都有一个概率，其实就是概率密度函数。SparkMlib使用期望最大值算法（EM）求解高斯密度。

模型参数：

* K： 期望聚类个数
* convergenceTol：
* maxIterations：最大迭代次数
* initialModel、；EM算法初始点

Demo:

**import** **org.apache.spark.mllib.clustering.**{**GaussianMixture**, **GaussianMixtureModel**}

**import** **org.apache.spark.mllib.linalg.Vectors**

*// Load and parse the data*

**val** data **=** sc.textFile("data/mllib/gmm\_data.txt")

**val** parsedData **=** data.map(s **=>** **Vectors**.dense(s.trim.split(' ').map(**\_**.toDouble))).cache()

*// Cluster the data into two classes using GaussianMixture*

**val** gmm **=** **new** **GaussianMixture**().setK(2).run(parsedData)

*// Save and load model*

gmm.save(sc, "target/org/apache/spark/GaussianMixtureExample/GaussianMixtureModel")

**val** sameModel **=** **GaussianMixtureModel**.load(sc,

"target/org/apache/spark/GaussianMixtureExample/GaussianMixtureModel")

*// output parameters of max-likelihood model*

**for** (i **<-** 0 until gmm.k) {

println("weight=%f\nmu=%s\nsigma=\n%s\n" format

(gmm.weights(i), gmm.gaussians(i).mu, gmm.gaussians(i).sigma))

}

## 幂迭代聚类power iteration clustering（PIC）

模型参数：

* K
* maxIterations
* initializationMode

输入：RDD of (srcId, dstId, similarity) tuples

Demo:

**import** **org.apache.spark.mllib.clustering.PowerIterationClustering**

**val** circlesRdd **=** generateCirclesRdd(sc, params.k, params.numPoints)

**val** model **=** **new** **PowerIterationClustering**()

.setK(params.k)

.setMaxIterations(params.maxIterations)

.setInitializationMode("degree")

.run(circlesRdd)

**val** clusters **=** model.assignments.collect().groupBy(**\_**.cluster).mapValues(**\_**.map(**\_**.id))

**val** assignments **=** clusters.toList.sortBy { **case** (k, v) **=>** v.length }

**val** assignmentsStr **=** assignments

.map { **case** (k, v) **=>**

s"$k -> ${v.sorted.mkString("[", ",", "]")}"

}.mkString(", ")

**val** sizesStr **=** assignments.map {

**\_**.\_2.length

}.sorted.mkString("(", ",", ")")

println(s"Cluster assignments: $assignmentsStr\ncluster sizes: $sizesStr")

## 隐含狄利克雷分布latent Dirichlet allocation

LDA是一种文档主题生成模型，也称为一个三层[贝叶斯](http://baike.baidu.com/view/77778.htm)概率模型，包含词、主题和文档三层结构。所谓生成模型，就是说，我们认为一篇文章的每个词都是通过“以一定概率选择了某个主题，并从这个主题中以一定概率选择某个词语”这样一个过程得到。文档到主题服从多项式分布，主题到词服从多项式分布。

LDA是一种非监督机器学习技术，可以用来识别大规模文档集（document collection）或语料库（corpus）中潜藏的主题信息。它采用了词袋（bag of words）的方法，这种方法将每一篇文档视为一个词频向量，从而将文本信息转化为了易于建模的数字信息。但是词袋方法没有考虑词与词之间的顺序，这简化了问题的复杂性，同时也为模型的改进提供了契机。每一篇文档代表了一些主题所构成的一个概率分布，而每一个主题又代表了很多单词所构成的一个概率分布。

LDA的核心公式：

p(w|d)=p(w|t)\*p(t|d)

模型参数：

* K: toptic个数或者聚类中心个数
* Optimizer： EMLDAOptimizer / OnlineLDAOptimizer 学习LDA模型
* docConcentration：狄利克雷参数for prior over documents’ distributions over topics
* topicConcentration：狄利克雷参数for prior over topics’ distributions over terms
* maxIterations
* checkpointInterval

Demo:

**import** **org.apache.spark.mllib.clustering.**{**DistributedLDAModel**, **LDA**}

**import** **org.apache.spark.mllib.linalg.Vectors**

**val** data **=** sc.textFile("data/mllib/sample\_lda\_data.txt")

**val** parsedData **=** data.map(s **=>** **Vectors**.dense(s.trim.split(' ').map(**\_**.toDouble)))

*// Index documents with unique IDs*

**val** corpus **=** parsedData.zipWithIndex.map(**\_**.swap).cache()

*// Cluster the documents into three topics using LDA*

**val** ldaModel **=** **new** **LDA**().setK(3).run(corpus)

*// Output topics. Each is a distribution over words (matching word count vectors)*

println("Learned topics (as distributions over vocab of " + ldaModel.vocabSize + " words):")

**val** topics **=** ldaModel.topicsMatrix

**for** (topic **<-** **Range**(0, 3)) {

print("Topic " + topic + ":")

**for** (word **<-** **Range**(0, ldaModel.vocabSize)) { print(" " + topics(word, topic)); }

println()

}

## 二等分K-mean算法Bisecting k-mean

## 流k-mean算法streaming k-mean

# 降维

## 奇异值分解 SVD

SVD分解就是把矩阵分解为：



其中，U/V是正交矩阵，是对角矩阵，对角线元素是矩阵A的特征值。那么，我们在求解方程组的时候，就将对角线上值非常小的记录去掉，达到降维的目的。同时，也大大提高了计算的效率。

**Demo:**

**import** **org.apache.spark.mllib.linalg.Matrix**

**import** **org.apache.spark.mllib.linalg.SingularValueDecomposition**

**import** **org.apache.spark.mllib.linalg.Vector**

**import** **org.apache.spark.mllib.linalg.Vectors**

**import** **org.apache.spark.mllib.linalg.distributed.RowMatrix**

**val** data **=** **Array**(

**Vectors**.sparse(5, **Seq**((1, 1.0), (3, 7.0))),

**Vectors**.dense(2.0, 0.0, 3.0, 4.0, 5.0),

**Vectors**.dense(4.0, 0.0, 0.0, 6.0, 7.0))

**val** dataRDD **=** sc.parallelize(data, 2)

**val** mat**:** RowMatrix = **new** **RowMatrix**(dataRDD)

*// Compute the top 5 singular values and corresponding singular vectors.*

**val** svd**:** SingularValueDecomposition[RowMatrix, Matrix] **=** mat.computeSVD(5, computeU **=** **true**)

**val** U**:** RowMatrix = svd.U *// The U factor is a RowMatrix.*

**val** s**:** Vector = svd.s *// The singular values are stored in a local dense veo*

**val** V**:** Matrix = svd.V *// The V factor is a local dense matrix.*

## 主成分分析PCA

主成分分析（Principal Component Analysis，PCA）， 是一种统计方法。通过正交变换将一组可能存在相关性的变量转换为一组线性不相关的变量，转换后的这组变量叫主成分。

信息的大小通常用[离差平方和](http://baike.baidu.com/view/2850784.htm)或[方差](http://baike.baidu.com/view/172036.htm)来衡量。在降维中，广泛使用。

**Demo1(**RowMatrix**):**

**import** **org.apache.spark.mllib.linalg.Matrix**

**import** **org.apache.spark.mllib.linalg.Vectors**

**import** **org.apache.spark.mllib.linalg.distributed.RowMatrix**

**val** data **=** **Array**(

**Vectors**.sparse(5, **Seq**((1, 1.0), (3, 7.0))),

**Vectors**.dense(2.0, 0.0, 3.0, 4.0, 5.0),

**Vectors**.dense(4.0, 0.0, 0.0, 6.0, 7.0))

**val** dataRDD **=** sc.parallelize(data, 2)

**val** mat**:** RowMatrix = **new** **RowMatrix**(dataRDD)

*/ Compute the top 4 principal components.*

*// Principal components are stored in a local dense matrix.*

**val** pc**:** Matrix = mat.computePrincipalComponents(4)

*// Project the rows to the linear space spanned by the top 4 principal components.*

**val** projected**:** RowMatrix = mat.multiply(pc)

**Demo2(Vector):**

**import** **org.apache.spark.mllib.feature.PCA**

**import** **org.apache.spark.mllib.linalg.Vectors**

**import** **org.apache.spark.mllib.regression.LabeledPoint**

**import** **org.apache.spark.rdd.RDD**

**val** data**:** RDD[LabeledPoint] **=** sc.parallelize(**Seq**(

**new** **LabeledPoint**(0, **Vectors**.dense(1, 0, 0, 0, 1)),

**new** **LabeledPoint**(1, **Vectors**.dense(1, 1, 0, 1, 0)),

**new** **LabeledPoint**(1, **Vectors**.dense(1, 1, 0, 0, 0)),

**new** **LabeledPoint**(0, **Vectors**.dense(1, 0, 0, 0, 0)),

**new** **LabeledPoint**(1, **Vectors**.dense(1, 1, 0, 0, 0))))

*// Compute the top 5 principal components.*

**val** pca **=** **new** **PCA**(5).fit(data.map(**\_**.features))

*// Project vectors to the linear space spanned by the top 5 principal*

*// components, keeping the label*

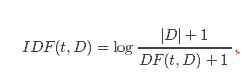
**val** projected **=** data.map(p **=>** p.copy(features **=** pca.transform(p.features))

# 特征提取和转换

## TF-IDF（文本挖掘）

[TF-IDF(Term frequency-inverse document frequency )](http://en.wikipedia.org/wiki/Tf%E2%80%93idf) 是文本挖掘中一种广泛使用的特征向量化方法。TF-IDF反映了语料中单词对文档的重要程度。

假设单词用t表示，文档用d表示，语料用D表示，那么文档频度DF(t, D)是包含单词t的文档数。如果我们只是使用词频度量重要性，就会很容易过分强调重负次数多但携带信息少的单词，例如：”a”, “the”以及”of”。如果某个单词在整个语料库中高频出现，意味着它没有携带专门针对某特殊文档的信息。逆文档频度(IDF)是单词携带信息量的数值度量。



其中 |D|是语料中的文档总数。

由于使用了log计算，如果单词在所有文档中出现，那么IDF就等于0. 注意这里做了平滑处理（+1操作），防止单词没有在语料中出现时IDF计算中除0。

TF-IDF度量是TF和IDF的简单相乘：



在MLlib中，为了灵活性我们将TF和IDF分开处理。

MLlib中词频统计的实现使用了[hashing trick（散列技巧）](http://en.wikipedia.org/wiki/Feature_hashing)，也就是使用哈希函数将原始特征映射到一个数字索引。然后基于这个索引来计算词频。这个方法避免了全局的单词到索引的映射，全局映射对于大量语料有非常昂贵的计算/存储开销；但是该方法也带来了潜在哈希冲突的问题，不同原始特征可能会被映射到相同的索引。为了减少冲突率，我们可以提升目标特征的维度，例如，哈希表中桶的数量。默认特征维度是220 = 1048576。

TF和IDF分别在类[HashingTF](http://spark.apache.org/docs/latest/api/python/pyspark.mllib.html#pyspark.mllib.feature.HashingTF) and [IDF](http://spark.apache.org/docs/latest/api/python/pyspark.mllib.html#pyspark.mllib.feature.IDF)中实现。HashTF以RDD[list]为输入。链表中的每个元素是可遍历的字符串或者其他类型。

**Demo:**

**import** **org.apache.spark.mllib.feature.**{**HashingTF**, **IDF**}

**import** **org.apache.spark.mllib.linalg.Vector**

**import** **org.apache.spark.rdd.RDD**

*// Load documents (one per line).*

**val** documents**:** RDD[Seq[String]] **=** sc.textFile("data/mllib/kmeans\_data.txt") .map(**\_**.split(" ").toSeq)

**val** hashingTF **=** **new** **HashingTF**()

**val** tf**:** RDD[Vector] **=** hashingTF.transform(documents)

*// While applying HashingTF only needs a single pass to the data, applying IDF needs two passes:*

*// First to compute the IDF vector and second to scale the term frequencies by IDF.*

tf.cache()

**val** idf **=** **new** **IDF**().fit(tf)

**val** tfidf**:** RDD[Vector] **=** idf.transform(tf)

*// spark.mllib IDF implementation provides an option for ignoring terms which occur in less than*

*// a minimum number of documents. In such cases, the IDF for these terms is set to 0.*

*// This feature can be used by passing the minDocFreq value to the IDF constructor.*

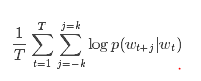
**val** idfIgnore **=** **new** **IDF**(minDocFreq **=** 2).fit(tf)

**val** tfidfIgnore**:** RDD[Vector] **=** idfIgnore.transform(tf)

## Word2Vec

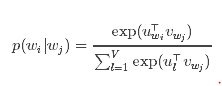
[Word2Vec](https://code.google.com/p/word2vec/) 计算单词的向量表示。这种表示的主要优点是相似的词在向量空间中离得近，这使得向新模式的泛化更容易并且模型估计更鲁棒。向量表示在诸如命名实体识别、歧义消除、句子解析、打标签以及机器翻译等自然语言处理程序中比较有用.

MLlib中的Word2Vec实现，使用的是skip-gram模型。skip-gram的目标函数是学习擅长预测同一个句子中词的上下文的词向量表示。用数学语言表达就是，给定一个训练单词序列：w1, w2, …, wT, skip-gram模型的目标是最大化平均log似然函数(log-likelihood):



其中k是训练窗口的大小，也就是给定一个词，需要分别查看前后k个词.

在skip-gram模型中，每个词w跟两个向量uw和vw关联：uw是w的词向量表示，是vw上下文。给定单词wj，正确预测单词wi的概率取决于softmax模型:

**

其中V是单词总数.

使用softmax的skip-gram模型开销很大，因为log p(wi|wj)的计算量跟V成比例，而V很可能在百万量级。为了加速Word2Vec的训练，我们引入了层次softmax，该方法将计算log p(wi|wj)时间复杂度降低到了O(log(V)).

Demo:

**import** **org.apache.spark.mllib.feature.**{**Word2Vec**, **Word2VecModel**}

**val** input **=** sc.textFile("data/mllib/sample\_lda\_data.txt").map(line **=>** line.split(" ").toSeq)

**val** word2vec **=** **new** **Word2Vec**()

**val** model **=** word2vec.fit(input)

**val** synonyms **=** model.findSynonyms("1", 5)

**for**((synonym, cosineSimilarity) **<-** synonyms) {

println(s"$synonym $cosineSimilarity")

}

## StandardScaler标准化

标准化是指：对于训练集中的样本，基于列统计信息将数据除以方差或（且）者将数据减去其均值（结果是方差等于1，数据在0附近）。这是很常用的预处理步骤。例如，当所有的特征具有值为1的方差且/或值为0的均值时，SVM的径向基函数（RBF）核或者L1和L2正则化线性模型通常有更好的效果。

标准化可以提升模型优化阶段的收敛速度，还可以避免方差很大的特征对模型训练产生过大的影响。

模型参数：

* withMean 默认值False. 在尺度变换（除方差）之前使用均值做居中处理（减去均值）。这会导致密集型输出，所以在稀疏数据上无效
* withStd 默认值True. 将数据缩放（尺度变换）到单位标准差

StandardScaler.fit()方法以RDD[Vector]为输入，计算汇总统计信息，然后返回一个模型，该模型可以根据StandardScaler配置将输入数据转换为标准差为1，均值为0的特征。

模型中还实现了[VectorTransformer](http://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.mllib.feature.VectorTransformer)，这个类可以对Vector和RDD[Vector]做转化。

Demo:

**import** **org.apache.spark.mllib.feature.**{**StandardScaler**, **StandardScalerModel**}

**import** **org.apache.spark.mllib.linalg.Vectors**

**import** **org.apache.spark.mllib.util.MLUtils**

**val** data **=** **MLUtils**.loadLibSVMFile(sc, "data/mllib/sample\_libsvm\_data.txt")

**val** scaler1 **=** **new** **StandardScaler**().fit(data.map(x **=>** x.features))

**val** scaler2 **=** **new** **StandardScaler**(withMean **=** **true**, withStd **=** **true**).fit(data.map(x **=>** x.features))

*// scaler3 is an identical model to scaler2, and will produce identical transformations*

**val** scaler3 **=** **new** **StandardScalerModel**(scaler2.std, scaler2.mean)

*// data1 will be unit variance.*

**val** data1 **=** data.map(x **=>** (x.label, scaler1.transform(x.features)))

*// Without converting the features into dense vectors, transformation with zero mean will raise*

*// exception on sparse vector.*

*// data2 will be unit variance and zero mean.*

**val** data2 **=** data.map(x **=>** (x.label, scaler2.transform(**Vectors**.dense(x.features.toArray))))

## Normalizer归一化

归一化是指将每个独立样本做尺度变换从而是该样本具有单位Lp范数。这是文本分类和聚类中的常用操作。例如，两个做了L2归一化的TF-IDF向量的点积是这两个向量的cosine（余弦）相似度。

[Normalizer](http://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.mllib.feature.Normalizer) 的构造函数有以下参数：

* 在Lp空间的p范数, 默认p=2

Normlizer实现了[VectorTransformer](http://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.mllib.feature.VectorTransformer) ，这个类可以对Vector和RDD[Vector]做归一化。

**import** **org.apache.spark.mllib.feature.Normalizer**

**import** **org.apache.spark.mllib.util.MLUtils**

**val** data **=** **MLUtils**.loadLibSVMFile(sc, "data/mllib/sample\_libsvm\_data.txt")

**val** normalizer1 **=** **new** **Normalizer**()

**val** normalizer2 **=** **new** **Normalizer**(p **=** **Double**.**PositiveInfinity**)

*// Each sample in data1 will be normalized using $L^2$ norm.*

**val** data1 **=** data.map(x **=>** (x.label, normalizer1.transform(x.features)))

*// Each sample in data2 will be normalized using $L^\infty$ norm.*

**val** data2 **=** data.map(x **=>** (x.label, normalizer2.transform(x.features)))

## 特征选择

### 卡方选择ChiSqSelector

[ChiSqSelector](http://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.mllib.feature.ChiSqSelector)是指使用卡方（Chi-Squared）做特征选择。该方法操作的是有标签的类别型数据。ChiSqSelector基于卡方检验来排序数据，然后选出卡方值较大(也就是跟标签最相关)的特征（topk)。

模型参数：

* numTopFeatures 保留的卡方较大的特征的数量

ChiSqSelector.fit() 方法以具有类别特征的RDD[LabeledPoint]为输入，计算汇总统计信息，然后返回ChiSqSelectorModel，这个类将输入数据转化到降维的特征空间

Demo：**import** **org.apache.spark.mllib.feature.ChiSqSelector**

**import** **org.apache.spark.mllib.linalg.Vectors**

**import** **org.apache.spark.mllib.regression.LabeledPoint**

**import** **org.apache.spark.mllib.util.MLUtils**

*// Load some data in libsvm format*

**val** data **=** **MLUtils**.loadLibSVMFile(sc, "data/mllib/sample\_libsvm\_data.txt")

*// Discretize data in 16 equal bins since ChiSqSelector requires categorical features*

*// Even though features are doubles, the ChiSqSelector treats each unique value as a category*

**val** discretizedData **=** data.map { lp **=>**

**LabeledPoint**(lp.label, **Vectors**.dense(lp.features.toArray.map { x **=>** (x / 16).floor }))}

*// Create ChiSqSelector that will select top 50 of 692 features*

**val** selector **=** **new** **ChiSqSelector**(50)

*// Create ChiSqSelector model (selecting features)*

**val** transformer **=** selector.fit(discretizedData)

*// Filter the top 50 features from each feature vector*

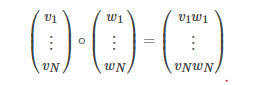
**val** filteredData **=** discretizedData.map { lp **=>**

**LabeledPoint**(lp.label, transformer.transform(lp.features))

}

## ElementwiseProduct:点积

点乘内积的计算方法：



模型参数：

* scalingVec: the transforming vector

Demo:

**import** **org.apache.spark.mllib.feature.ElementwiseProduct**

**import** **org.apache.spark.mllib.linalg.Vectors**

*// Create some vector data; also works for sparse vectors*

**val** data **=** sc.parallelize(**Array**(**Vectors**.dense(1.0, 2.0, 3.0), **Vectors**.dense(4.0, 5.0, 6.0)))

**val** transformingVector **=** **Vectors**.dense(0.0, 1.0, 2.0)

**val** transformer **=** **new** **ElementwiseProduct**(transformingVector)

*// Batch transform and per-row transform give the same results:*

**val** transformedData **=** transformer.transform(data)

**val** transformedData2 **=** data.map(x **=>** transformer.transform(x))

## PCA

Demo:

**import** **org.apache.spark.mllib.feature.PCA**

**import** **org.apache.spark.mllib.linalg.Vectors**

**import** **org.apache.spark.mllib.regression.**{**LabeledPoint**, **LinearRegressionWithSGD**}

**val** data **=** sc.textFile("data/mllib/ridge-data/lpsa.data").map { line **=>**

**val** parts **=** line.split(',')

**LabeledPoint**(parts(0).toDouble, **Vectors**.dense(parts(1).split(' ').map(**\_**.toDouble)))}.cache()

**val** splits **=** data.randomSplit(**Array**(0.6, 0.4), seed **=** 11L)

**val** training **=** splits(0).cache()

**val** test **=** splits(1)

**val** pca **=** **new** **PCA**(training.first().features.size / 2).fit(data.map(**\_**.features))

**val** training\_pca **=** training.map(p **=>** p.copy(features **=** pca.transform(p.features)))

**val** test\_pca **=** test.map(p **=>** p.copy(features **=** pca.transform(p.features)))

**val** numIterations **=** 100

**val** model **=** **LinearRegressionWithSGD**.train(training, numIterations)

**val** model\_pca **=** **LinearRegressionWithSGD**.train(training\_pca, numIterations)

**val** valuesAndPreds **=** test.map { point **=>**

**val** score **=** model.predict(point.features)

(score, point.label)}

**val** valuesAndPreds\_pca **=** test\_pca.map { point **=>**

**val** score **=** model\_pca.predict(point.features)

(score, point.label)}

**val** **MSE** **=** valuesAndPreds.map { **case** (v, p) **=>** math.pow((v - p), 2) }.mean()

**val** **MSE\_pca** **=** valuesAndPreds\_pca.map { **case** (v, p) **=>** math.pow((v - p), 2) }.mean()

println("Mean Squared Error = " + **MSE**)

println("PCA Mean Squared Error = " + **MSE\_pca**)

# 频繁模式挖掘Frequent pattern mining

## 频繁模式增长FP-growth

## 关联规则

## PrefixSpan算法

# 模型评价Evaluation Matrics

## 分类模型评估

### 二分类

以垃圾文件分类为例：

混淆矩阵：

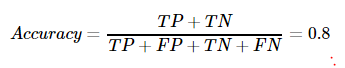


其中True和False是对于评价预测结果而言，也就是评价预测结果是正确的(True)还是错误的(False)。而Positive和Negative则是对分类器的预测结果而言，Positive为预测是垃圾，Negative是预测为非垃圾。为了表达方便，一般会以他们的缩写TP,FP,FN,TN代替。那么上面提到的邮件分类四种情形对应一下就变成如下：

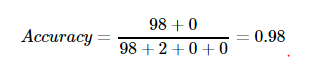
1. 邮件为垃圾，系统正确识别为垃圾(TP正确地预测为垃圾)
2. 邮件不是垃圾，系统错误识别为垃圾(FP错误的预测为垃圾)
3. 邮件不是垃圾，系统正确识别为非垃圾(TN正确地预测为非垃圾)
4. 邮件为垃圾，系统错误识别为非垃圾(FN错误地预测为非垃圾)

#### Accuracy

介绍了分类器的所有可能的分类情形，对于为什么不能只是用Accuracy作为唯一的判断标准还不是很明朗，我们接着分析。   
上面的Confusion Matrix只是对于一封邮件而言，那么对于多封邮件的预测结果情形，TP,FP,FN,TN则是各种情形的计数结果，假如我们有100封测试邮件，其中50封为垃圾50封为非垃圾，就可能会预测出TP=40封，TN=40封，FP = 10封，FN = 10封。那么用Accuracy来衡量的话，就会得到：



上述情形是数据样本比较均衡的情况下，但是如果数据样本非常不均衡的时候，用精确度就会出现问题了。假设100封邮件中，有90封都是垃圾，10封是非垃圾，这种样本数据成为skewd data, 如果有一个trival分类器，永远只输出垃圾邮件，假设TP=98,TN=0,Fn=0,FP=2, 那么准确率为



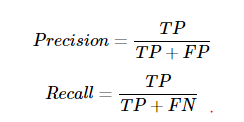
如果只用精确率去评估，结果出乎意料的好，这样最终就选择了一个所谓的Accuracy Paradox现象。可看出，Accuracy并不能很好体现TN=FN=0这种异常现象，所以我们有必要寻求其他Metric来更好的刻画Confusion Matrix所表达的内容。

#### Precision-Recall

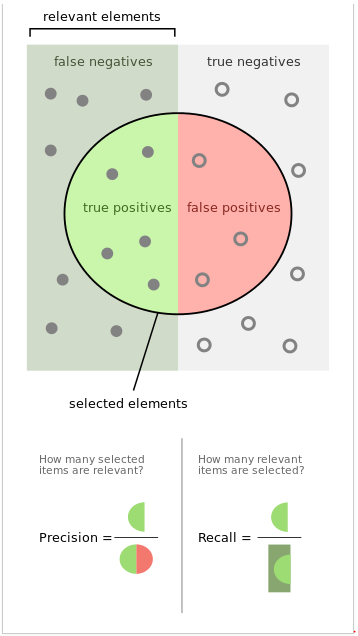
对于信息检索问题，我们更关注搜索到的内容与用户想要搜索的内容是否相关。如果有测试数据的话，我们是希望搜索引擎返回的内容与用户要找的越相关越好，另一方面我们想让搜索引擎返回越多的相关内容越好，所以就有了Precision和Recall这两个指标。

* Precision是返回的相关文献(TP)占总返回文献(TP+FP)的比例，所以Precision越大，返回的结果质量越高
* Recall是返回的相关文献(TP)占总相关文献(TP+FN)的比例，所以Recall越大，返回相关的文献就越多

一个是强调质量(返回结果只要给我相关的就好)，一个是强调数量(返回相关的结果要尽可能越多越好)。按照上面的定义，我们根据Confusion Matrix可以给出Precision和Recall的数学形式：



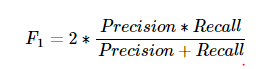
wiki上面有一幅插图很形象地表达了Precision和Recall的含义：



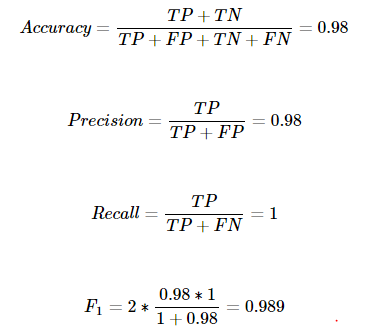
这两个指标哪个更好呢？这个要根据业务需求设定。举个例子，对于识别是否是超市会员与非会员的系统，那么Recall越大越好，因为虽然是混进去一些非会员，但是整体并非损失，只是打个折扣，还是赚钱。但是对于门禁系统来讲，Precision越大越好，Recall可以不用那么严格，虽然让你多刷几次卡，但是总比把外人放进来好。所以，根据不同业务，权衡这两个指标还是很重要的，那么有没有一种指标可以综合这两个因素呢？yes! F-measure

#### F-measure

最简单的合并Precision和Recall的方式是F1\_Measure



这样的其实是均匀地加权平均，利用这个指标，我们回归到上面提到的Accuracy缺陷的例子，假如分类器一直输出为垃圾，对于有100封测试邮件其中垃圾为98封的测试集：TP=98，TN=0，FN=0，FP=2，那么



看来利用Precision-Recall或者是它们的组合F1都无法解决上面提到的skewed distribution问题(网上大量文章都是以Accuracy Paradox的问题引出Precision-Recall到F measure，让人以为F measure可以应对那种问题，其实不然)。但是我们通过拆分Confusion Matrix，再进行组合，可以使得我们更好得把握质和量，所以种种迹象都表面Precision-Recall和F measure比简单的Accuracy具有更细粒度的解释效果。

#### MCC(Matthews Correlation Coefficient)

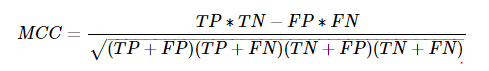
系数表示：

C=1分类器是完美的

C=0 分类器和随机分类器没差

C=-1分类器是最差的，所有预测结果和实际相反

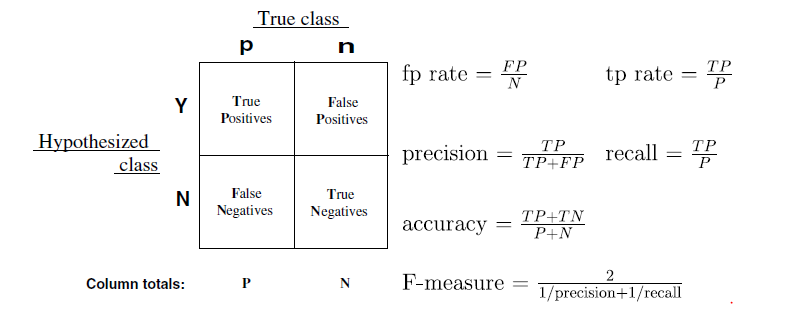
数学表达式：



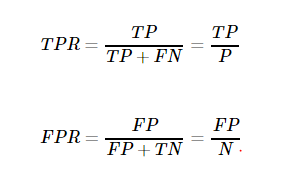
分母中任意一对括号相加之和如果为0，那么整个MCC的值就为0。回归到上面提到的问题，对于TP=98，TN=0，FN=0，FP=2，由于TN，FN同时为0，那么MCC则为0，说明我们简单粗暴的方式和随机分类器没有异同。

#### ROC/AUC

首先ROC是一个二维曲线，那么对于二维曲线我们必须建立一个坐标系，对于ROC，他的纵坐标是TPR(True Positive Rate)，横坐标是FPR(False Positive Rate)。

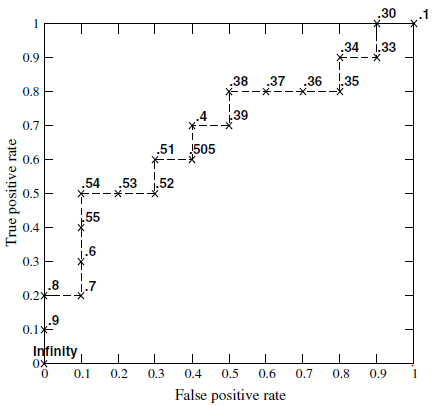


纵横坐标：



TPR和FPR是在Confusion Matrix上两列互相独立的统计量，和Precision-Recall有着明显的不同。

分类器对于每一个测试样例是计算出一个概率，而分类的判定需要给定一个threshold θ，如果概率大于θ就输出一个正结果，如果小于就输出负结果。每一个θ对于固定的测试集，都可以计算出一对(FPR,TPR)，当θ的取值为(−∞,+∞),就有无数对(FPR,TPR)与之对于，将这些点画在坐标系上，ROC便产生了。实际情况下，我们可以将分类器给出的所有分数或概率进行排序(排序只是为了能够高效求解TPR和FPR)，并从小到大用每一个分数作为θ(为什么呢？如果取排序好的两个θi,θj之间的任意值都会得到相同的一对(TPR,FPR))进行作图，我们画出来的ROC曲线如下图折线所示：



如果θ取得越小，测试样本越多，折线就会越趋向平滑。一般来说，对于好的分类器，ROC曲线会越偏向左上角(对应于(0,1)点，此时FPR=0,TPR=1代表完美的分类器)，而越靠近对角线，分类器越趋向于随机分类器。

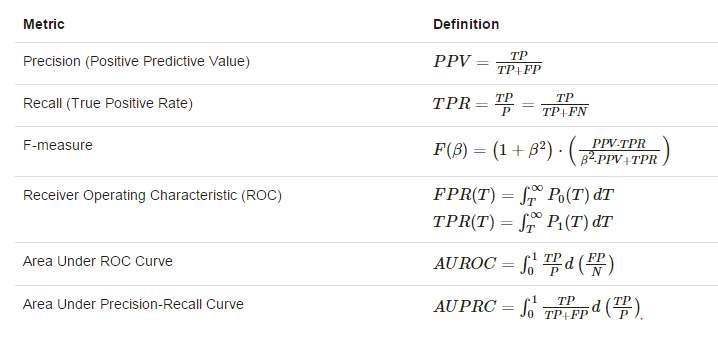
#### sparkMlib模型评估

基于上面的理论，sparkMlib提供了几种评估分类模型的方法，PR/AOC/F-mearure

P-R curve plots (precision, recall) points for different threshold values,

ROC curve plots (recall, false positive rate) points.

具体评估指标有以下几种：



**Demo:**

**import** **org.apache.spark.mllib.classification.LogisticRegressionWithLBFGS**

**import** **org.apache.spark.mllib.evaluation.BinaryClassificationMetrics**

**import** **org.apache.spark.mllib.regression.LabeledPoint**

**import** **org.apache.spark.mllib.util.MLUtils**

*// Load training data in LIBSVM format*

**val** data **=** **MLUtils**.loadLibSVMFile(sc, "data/mllib/sample\_binary\_classification\_data.txt")

*// Split data into training (60%) and test (40%)*

**val** **Array**(training, test) **=** data.randomSplit(**Array**(0.6, 0.4), seed **=** 11L)

training.cache()

*// Run training algorithm to build the model*

**val** model **=** **new** **LogisticRegressionWithLBFGS**()

.setNumClasses(2)

.run(training)

*// Clear the prediction threshold so the model will return probabilities*

model.clearThreshold

*// Compute raw scores on the test set*

**val** predictionAndLabels **=** test.map { **case** **LabeledPoint**(label, features) **=>**

**val** prediction **=** model.predict(features)

(prediction, label)

}

*// Instantiate metrics object*

**val** metrics **=** **new** **BinaryClassificationMetrics**(predictionAndLabels)

*// Precision by threshold*

**val** precision **=** metrics.precisionByThreshold

precision.foreach { **case** (t, p) **=>**

println(s"Threshold: $t, Precision: $p")

}

*// Recall by threshold*

**val** recall **=** metrics.recallByThreshold

recall.foreach { **case** (t, r) **=>**

println(s"Threshold: $t, Recall: $r")

}

*// Precision-Recall Curve*

**val** **PRC** **=** metrics.pr

*// F-measure*

**val** f1Score **=** metrics.fMeasureByThreshold

f1Score.foreach { **case** (t, f) **=>**

println(s"Threshold: $t, F-score: $f, Beta = 1")

}

**val** beta **=** 0.5

**val** fScore **=** metrics.fMeasureByThreshold(beta)

f1Score.foreach { **case** (t, f) **=>**

println(s"Threshold: $t, F-score: $f, Beta = 0.5")

}

*// AUPRC*

**val** auPRC **=** metrics.areaUnderPR

println("Area under precision-recall curve = " + auPRC)

*// Compute thresholds used in ROC and PR curves*

**val** thresholds **=** precision.map(**\_**.\_1)

*// ROC Curve*

**val** roc **=** metrics.roc

*// AUROC*

**val** auROC **=** metrics.areaUnderROC

println("Area under ROC = " + auROC)

### 多分类

#### 定义

定义一个标签集合：



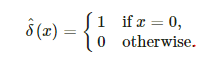
N个数据集的真正输出结果Y:

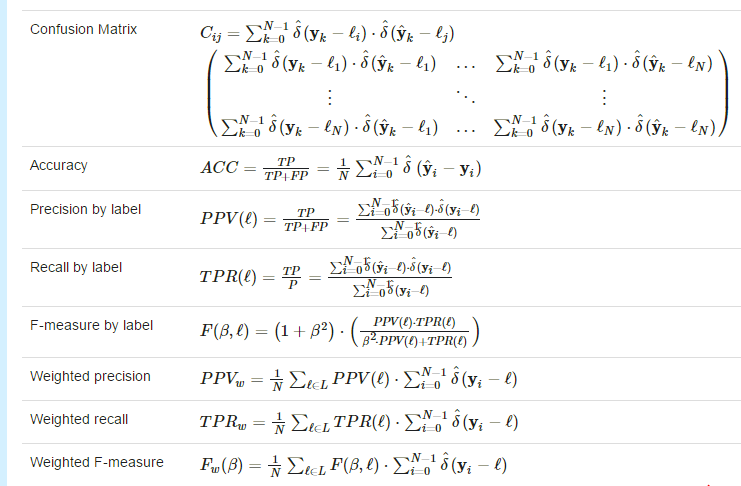
捕获.PNG

多分类模型预测结果：



定义函数：





#### 实现

Demo:

**import** **org.apache.spark.mllib.classification.LogisticRegressionWithLBFGS**

**import** **org.apache.spark.mllib.evaluation.MulticlassMetrics**

**import** **org.apache.spark.mllib.regression.LabeledPoint**

**import** **org.apache.spark.mllib.util.MLUtils**

**val** data **=** **MLUtils**.loadLibSVMFile(sc, "data/mllib/sample\_multiclass\_classification\_data.txt")

**val** **Array**(training, test) **=** data.randomSplit(**Array**(0.6, 0.4), seed **=** 11L)

training.cache()

*// Run training algorithm to build the model*

**val** model **=** **new** **LogisticRegressionWithLBFGS**()

.setNumClasses(3)

.run(training)

*// Compute raw scores on the test set*

**val** predictionAndLabels **=** test.map { **case** **LabeledPoint**(label, features) **=>**

**val** prediction **=** model.predict(features)

(prediction, label)}

*// Instantiate metrics object*

**val** metrics **=** **new** **MulticlassMetrics**(predictionAndLabels)

*// Confusion matrix*

println("Confusion matrix:")

println(metrics.confusionMatrix)

*// Overall Statistics*

**val** accuracy **=** metrics.accuracy

println("Summary Statistics")

println(s"Accuracy = $accuracy")

*// Precision by label*

**val** labels **=** metrics.labels

labels.foreach { l **=>**

println(s"Precision($l) = " + metrics.precision(l))

}*// Recall by label*

labels.foreach { l **=>**

println(s"Recall($l) = " + metrics.recall(l))

}*// False positive rate by label*

labels.foreach { l **=>**

println(s"FPR($l) = " + metrics.falsePositiveRate(l))

}*// F-measure by label*

labels.foreach { l **=>**

println(s"F1-Score($l) = " + metrics.fMeasure(l))

}*// Weighted stats*

println(s"Weighted precision: ${metrics.weightedPrecision}")

println(s"Weighted recall: ${metrics.weightedRecall}")

println(s"Weighted F1 score: ${metrics.weightedFMeasure}")

println(s"Weighted false positive rate: ${metrics.weightedFalsePositiveRate}")

### 多标签分类

#### 基本概念

多标签分类是每一个记录映射到一个集合里面，就比如一篇文章，属于人文/社科，某一个电影属于动作/悬疑。

定义集合D



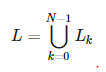
某个记录di从属的标签集合：

捕获.PNG

该记录di预测出的标签集合：



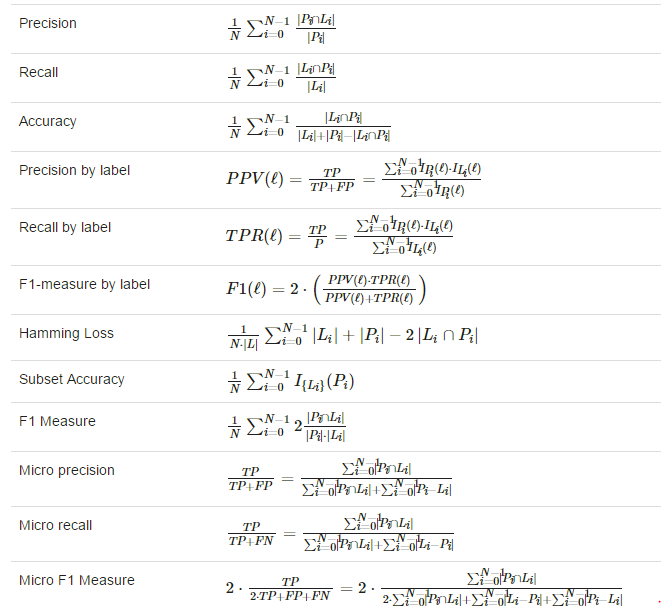
那么我们给记录di打的标签为：



定义一个函数



那么，基于上面的集合表示，多标签分类模型的评价指标有以下几种:



#### 实现

Demo: 假设有以下预测结果

Document predictions:

* doc 0 - predict 0, 1 - class 0, 2
* doc 1 - predict 0, 2 - class 0, 1
* doc 2 - predict none - class 0
* doc 3 - predict 2 - class 2
* doc 4 - predict 2, 0 - class 2, 0
* doc 5 - predict 0, 1, 2 - class 0, 1
* doc 6 - predict 1 - class 1, 2

Predicted classes:

* class 0 - doc 0, 1, 4, 5 (total 4)
* class 1 - doc 0, 5, 6 (total 3)
* class 2 - doc 1, 3, 4, 5 (total 4)

True classes:

* class 0 - doc 0, 1, 2, 4, 5 (total 5)
* class 1 - doc 1, 5, 6 (total 3)
* class 2 - doc 0, 3, 4, 6 (total 4)

**import** **org.apache.spark.mllib.evaluation.MultilabelMetrics**

**import** **org.apache.spark.rdd.RDD**

**val** scoreAndLabels**:** RDD[(Array[Double], Array[Double])] **=** sc.parallelize(

**Seq**((**Array**(0.0, 1.0), **Array**(0.0, 2.0)),

(**Array**(0.0, 2.0), **Array**(0.0, 1.0)),

(**Array**.empty[Double], **Array**(0.0)),

(**Array**(2.0), **Array**(2.0)),

(**Array**(2.0, 0.0), **Array**(2.0, 0.0)),

(**Array**(0.0, 1.0, 2.0), **Array**(0.0, 1.0)),

(**Array**(1.0), **Array**(1.0, 2.0))), 2)

*// Instantiate metrics object*

**val** metrics **=** **new** **MultilabelMetrics**(scoreAndLabels)

*// Summary stats*

println(s"Recall = ${metrics.recall}")

println(s"Precision = ${metrics.precision}")

println(s"F1 measure = ${metrics.f1Measure}")

println(s"Accuracy = ${metrics.accuracy}")

*// Individual label stats*

metrics.labels.foreach(label **=>**

println(s"Class $label precision = ${metrics.precision(label)}"))

metrics.labels.foreach(label **=>** println(s"Class $label recall = ${metrics.recall(label)}"))

metrics.labels.foreach(label **=>** println(s"Class $label F1-score = ${metrics.f1Measure(label)}"))*// Micro stats*

println(s"Micro recall = ${metrics.microRecall}")

println(s"Micro precision = ${metrics.microPrecision}")

println(s"Micro F1 measure = ${metrics.microF1Measure}")

*// Hamming loss*

println(s"Hamming loss = ${metrics.hammingLoss}")

*// Subset accuracy*

println(s"Subset accuracy = ${metrics.subsetAccuracy}")

## 推荐系统(Ranking System)模型评估

### 基本定义

定义M个用户：



每个用户有N个相关记录：



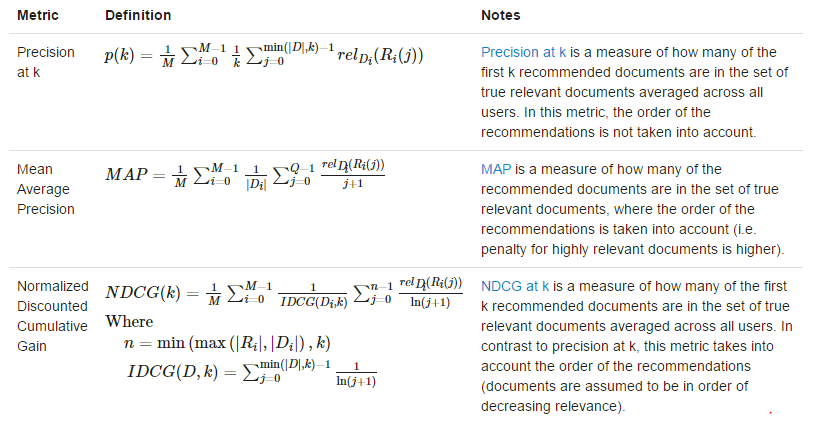
定义Q个预测出相关度比较大的label, 排序后取前Q个



定义一个函数：

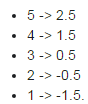


那么可以通过以下方法去评估模型：



### 实现

**Demo:** 假设用户给电影的评分为1-5分，5：一定回去看，4：喜欢，3：还可以2：差 1：相当差。把用户给电影的评分映射为分数：



那么我们就该向用户推荐分数大于0的电影。

**import** **org.apache.spark.mllib.evaluation.**{**RankingMetrics**, **RegressionMetrics**}

**import** **org.apache.spark.mllib.recommendation.**{**ALS**, **Rating**}

**val** ratings **=** spark.read.textFile("data/mllib/sample\_movielens\_data.txt").rdd.map { line **=>**

**val** fields **=** line.split("::")

**Rating**(fields(0).toInt, fields(1).toInt, fields(2).toDouble - 2.5)

}.cache()

*// Map ratings to 1 or 0, 1 indicating a movie that should be recommended*

**val** binarizedRatings **=** ratings.map(r **=>** **Rating**(r.user, r.product,

**if** (r.rating > 0) 1.0 **else** 0.0)).cache()

*// Summarize ratings*

**val** numRatings **=** ratings.count()

**val** numUsers **=** ratings.map(**\_**.user).distinct().count()

**val** numMovies **=** ratings.map(**\_**.product).distinct().count()

println(s"Got $numRatings ratings from $numUsers users on $numMovies movies.")

*// Build the model*

**val** numIterations **=** 10

**val** rank **=** 10

**val** lambda **=** 0.01

**val** model **=** **ALS**.train(ratings, rank, numIterations, lambda)

*// Define a function to scale ratings from 0 to 1*

**def** scaledRating(r**:** Rating)**:** Rating = {

**val** scaledRating **=** math.max(math.min(r.rating, 1.0), 0.0)

**Rating**(r.user, r.product, scaledRating)

}

*// Get sorted top ten predictions for each user and then scale from [0, 1]*

**val** userRecommended **=** model.recommendProductsForUsers(10).map { **case** (user, recs) **=>**

(user, recs.map(scaledRating))

}

*// Assume that any movie a user rated 3 or higher (which maps to a 1) is a relevant document*

*// Compare with top ten most relevant documents*

**val** userMovies **=** binarizedRatings.groupBy(**\_**.user)

**val** relevantDocuments **=** userMovies.join(userRecommended).map { **case** (user, (actual,

predictions)) **=>**

(predictions.map(**\_**.product), actual.filter(**\_**.rating > 0.0).map(**\_**.product).toArray)

}

*// Instantiate metrics object*

**val** metrics **=** **new** **RankingMetrics**(relevantDocuments)

*// Precision at K*

**Array**(1, 3, 5).foreach { k **=>**

println(s"Precision at $k = ${metrics.precisionAt(k)}")

}

*// Mean average precision*

println(s"Mean average precision = ${metrics.meanAveragePrecision}")

*// Normalized discounted cumulative gain*

**Array**(1, 3, 5).foreach { k **=>**

println(s"NDCG at $k = ${metrics.ndcgAt(k)}")

}

*// Get predictions for each data point*

**val** allPredictions **=** model.predict(ratings.map(r **=>** (r.user, r.product))).map(r **=>** ((r.user,

r.product), r.rating))

**val** allRatings **=** ratings.map(r **=>** ((r.user, r.product), r.rating))

**val** predictionsAndLabels **=** allPredictions.join(allRatings).map { **case** ((user, product),

(predicted, actual)) **=>**

(predicted, actual)

}

*// Get the RMSE using regression metrics*

**val** regressionMetrics **=** **new** **RegressionMetrics**(predictionsAndLabels)

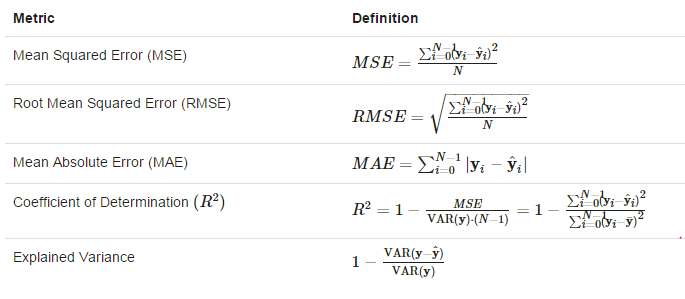
println(s"RMSE = ${regressionMetrics.rootMeanSquaredError}")

*// R-squared*

println(s"R-squared = ${regressionMetrics.r2}")

## 回归模型的评估

### 常用评估算法



### 实现

Demo:

**import** **org.apache.spark.mllib.evaluation.RegressionMetrics**

**import** **org.apache.spark.mllib.linalg.Vector**

**import** **org.apache.spark.mllib.regression.**{**LabeledPoint**, **LinearRegressionWithSGD**}

*// Load the data*

**val** data **=** spark

.read.format("libsvm").load("data/mllib/sample\_linear\_regression\_data.txt")

.rdd.map(row **=>** **LabeledPoint**(row.getDouble(0), row.get(1).asInstanceOf[Vector])).cache()

*// Build the model*

**val** numIterations **=** 100

**val** model **=** **LinearRegressionWithSGD**.train(data, numIterations)

*// Get predictions*

**val** valuesAndPreds **=** data.map{ point **=>**

**val** prediction **=** model.predict(point.features)

(prediction, point.label)}

*// Instantiate metrics object*

**val** metrics **=** **new** **RegressionMetrics**(valuesAndPreds)

*// Squared error*

println(s"MSE = ${metrics.meanSquaredError}")

println(s"RMSE = ${metrics.rootMeanSquaredError}")

*// R-squared*

println(s"R-squared = ${metrics.r2}")

*// Mean absolute error*

println(s"MAE = ${metrics.meanAbsoluteError}")

*// Explained variance*

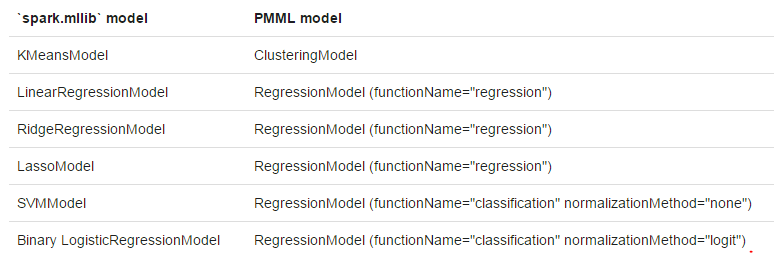
println(s"Explained variance = ${metrics.explainedVariance}")

# 预测模型标记语言PMML模型导入

## 基本概念

预测模型标记语言（Predictive Model Markup Language，PMML）是一种可以呈现预测分析模型的事实标准语言。PMML 是一种事实标准语言，用于呈现数据挖掘模型。PMML 允许您在不同的应用程序之间轻松共享预测分析模型。因此，您可以在一个系统中定型一个模型，在 PMML 中对其进行表达，然后将其移动到另一个系统中，并在该系统中使用上述模型预测机器失效的可能性等。

目前，sparkMlib支持的能够导出为PMML的模型有以下几个:



## 实现

**Demo:**

**import** **org.apache.spark.mllib.clustering.KMeans**

**import** **org.apache.spark.mllib.linalg.Vectors**

*// Load and parse the data*

**val** data **=** sc.textFile("data/mllib/kmeans\_data.txt")

**val** parsedData **=** data.map(s **=>** **Vectors**.dense(s.split(' ').map(**\_**.toDouble))).cache()

*// Cluster the data into two classes using KMeans*

**val** numClusters **=** 2

**val** numIterations **=** 20

**val** clusters **=** **KMeans**.train(parsedData, numClusters, numIterations)

*// Export to PMML to a String in PMML format*

println("PMML Model:\n" + clusters.toPMML)

*// Export the model to a local file in PMML format*

clusters.toPMML("/tmp/kmeans.xml")

*// Export the model to a directory on a distributed file system in PMML format*

clusters.toPMML(sc, "/tmp/kmeans")

*// Export the model to the OutputStream in PMML format*

clusters.toPMML(**System**.out)

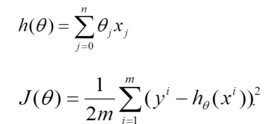
# 优化

对于所有的分类和回归问题，其本质都是求解损失函数，然后通过optimization算法（SGD或LBFGS）求使得loss function最小的凸优化问题，最后得出的解是一个weights向量。各种算法的不同之处都是指定的损失函数和正则化项不同，其求解损失函数最优解方法的过程是相同的，因此将Optimization模板分开提取出来一个章节。目前，sparkMLib支持两种优化算法SGD和LBFGS。

同时，为了避免过拟合，加入正则化项，对权重向量施加一个惩罚。MLlib目前支持两种正则化方法L1和L2。 L2正则化假设模型参数服从高斯分布，L2正则化函数比L1更光滑，所以更容易计算；L1假设模型参数服从拉普拉斯分布，L1正则化具备产生稀疏解的功能，从而具备feature selection的能力。

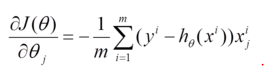
## 随机梯度下降法stochastic gradient descent

下面的h(x)是要拟合的函数，J(theta)损失函数，theta是参数，要迭代求解的值，theta求解出来了那最终要拟合的函数h(theta)就出来了。其中m是训练集的记录条数，j是参数的个数。

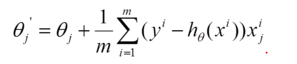


**批量梯度下降的思路：**

1. 将J(theta)对theta求偏导，得到每个theta对应的的梯度



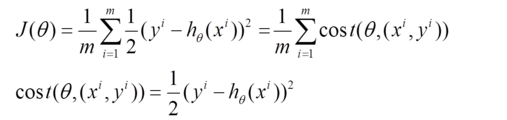
1. 由于是要最小化风险函数，所以按每个参数theta的梯度负方向，来更新每个theta



1. 从上面可以看到，批量梯度得到的是一个最优解，但是每一个迭代，都要用到训练集中的所有数据，如果数据量很大，那么速度就可想而知了。

**随机梯度下降的思路：**

（1）上面的风险函数可以写成如下这种形式，损失函数对应的是训练集中每个样本的粒度，而上面批量梯度下降对应的是所有的训练样本：



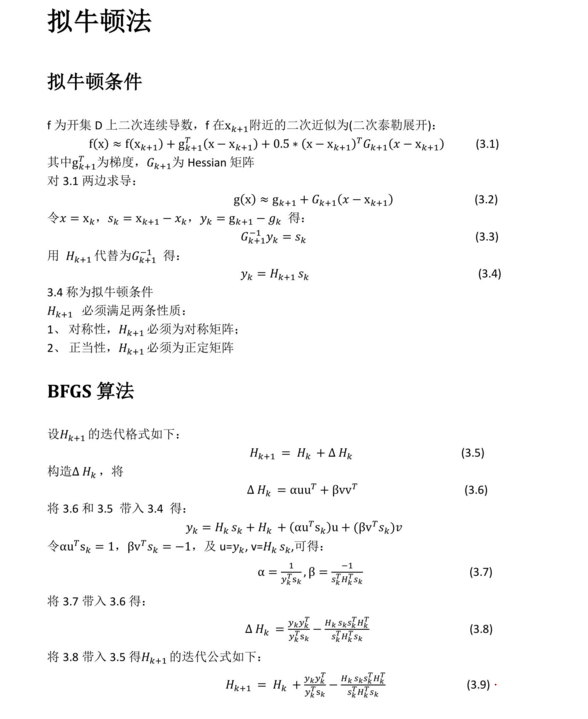
（2）每个样本的损失函数，对theta求偏导得到对应梯度，来更新theta

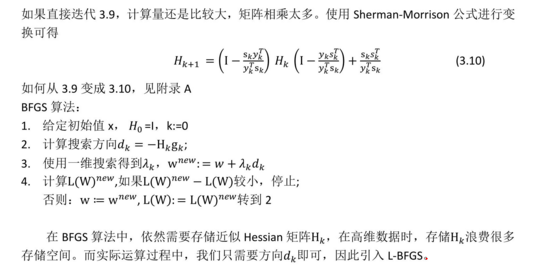
捕获.PNG

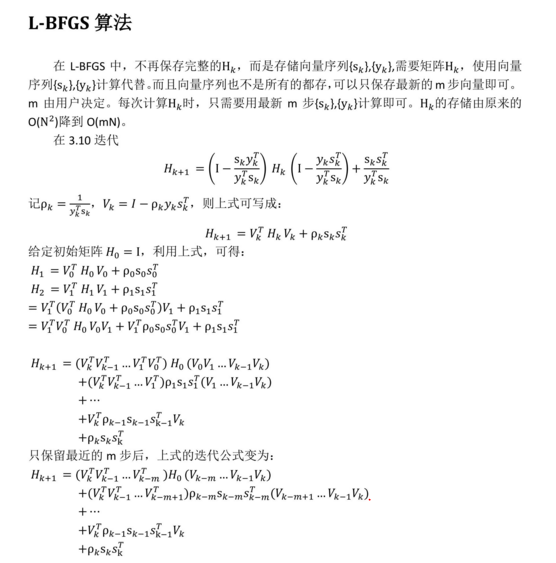
（3）随机梯度下降是每个样本迭代更新一次，如果样本很大，可能只需要迭代几千或者几万就可以收敛了。相比批量迭代，速率大大提升，但是随机梯度无法保证每次都能朝整体最优方向迭代，所以噪声相对较大。

## L-BFGS算法(短时记忆的BFGS)

L-BFGS算法其实是拟牛顿算法的改进，具体算法实现参看下面算法伪代码

http://blog.csdn.net/zhirom/article/details/38332111





# Spark学习记录

1. 进入spark-shell

/home/mr/spark/bin/spark-shell --master spark://$(echo `hostname`):7077  --executor-memory 5G --total-executor-cores 3

1. 运行example

进入到bin目录,运行 run-example.cmd SparkPi 10

# 