Mllib 中的机器学习算法

胡嘉蔚*

银基富力信息技术有限公司

2016/12/22

目录

*E-mail: jiawei.hu@datageek.com.cn

1 简介

Apache Spark 是一个开源框架，作为计算引擎，它把程序分发到集群中的许多机器，同时提供了一个优雅的编程模型。Spark 源自加州大学伯克利分校的 AMPLab，现属于 Apache 软件基金会。

Spark 的前辈有很多，从 MPI 到 MapReduce。利用这些计算框架，我们的程序可以充分利用大量资源，但不需要关心分布式系统的实现细节。数据处理的需求促进了这些技术框架的发展。同样，大数据领域也和这些框架关系密切，这些框架界定了大数据的范围。Spark 能大大提升 ETL 流水作业的性能，并且带来了支持迭代式计算和交互式的探索模式。[14]

MLlib 是 Spark 对常用的机器学习算法的实现库，同时提供了一系列的测试和随机数据生成等相关功能。利用 Spark 的分布式系统的框架，这些机器学习算法的速度可以得到很大的提升。而且我们不再需要手动地将这些平行化算法手动地分配到各个机器中，大大地节约了机器学习算法实现的效率。

由于这些算法函数已经被 Mllib 实现，我们所需要的也许仅仅是调整几个参数，但是我们也依然很有必要熟悉了解这些算法的基本原理。本文将从算法原理解读 MLlib 里的各个函数，从数学理论的角度带领大家明白机器学习算法具体对数据做了哪些操作。也可以更好的理解 MLlib 函数中的各个参数：这些参数在数学上的原理，以及调整这些参数对结果的影响。

本文**第二章**介绍了对数据进行**初步统计分析**的方法以及 Mllib 中的函数应用，包括期望方差的计算，相关系数的计算，分层抽样方法，统计假设的验证，以及随机数据的生成。**第三章**介绍了多种机器学习上经典的分类和回归算法，用于已知标签数据的**监督学习**，比如线性回归，逻辑回归，决策树，随机森林等等。**第四章**介绍了 MLlib 中基于协同过滤算法的**推荐系统**的构建。**第五章**则介绍了几种聚类算法用于实现对于无标签数据的**非监督学习**，包括 k 均值算法，混合高斯模型等等。

本文中的所有 MBlib 函数以 Spark 2.0.2 版本为标准，使用 Python 2.7.12 进行测试。函数范例均由 Apache Spark 官网 [3] 提供。

2 Basic statistics - 基础统计概念

统计学是机器学习算法的基础，机器学习的基本思想是通过已知的大量数据，来发掘其中存在的一些潜在的关系。利用这些信息，我们建立一些模型来拟合数据之间的关系，从而完成学习的目的。这一思想其实就是统计学的思想，利用归纳演绎的方法来解决问题。

我们对数据进行分类肯定需要先对这些数据进行统计学上的一些分析整理。初步的

统计整理可以帮助我们确定数据特征之间的一些相关性，剔除一些异常的数据。

2.1 Summary statistics - 统计概要

期望/方差 常见的统计概述中我们会提供一个数列的期望，方差等信息：对于数列 $(X_i)_{i=1,\dots,n} \in \mathbb{R}^n$（在概率统计上，我们把这个数列称为随机变量（*random variable*））, 我们统计学上定义了一些关于该数列的一些基础信息：

- **期望（mean）**：$\mathbb{E}[X] = \bar{X} = \frac{1}{n}\sum_{1\le i\le n} X_i$
- **方差（variance）**：$Var[X] = \mathbb{E}[(X-\bar{X})^2] = \frac{1}{n}\sum_{1\le i\le n}(X_i-\bar{X})^2$

期望表示了随机变量 X **平均取值**的大小，方差用来度量随机变量和其数学期望（即均值）之间的**偏离程度**。方差越小，随机变量的变化范围也就越小。

Mllib 函数 Mllib 的 stat 包中提供了函数*colStats*用于比较方便的显示矩阵中每一列数据的统计信息 (目前仅支持 scala 语言，Python 中可以利用 Pandas 或其他包来实现这一点)：

```scala
import org.apache.spark.mllib.linalg.Vectors
import org.apache.spark.mllib.stat.{MultivariateStatisticalSummary, Statistics}

val observations = sc.parallelize(
    Seq(
        Vectors.dense(1.0, 10.0, 100.0),
        Vectors.dense(2.0, 20.0, 200.0),
        Vectors.dense(3.0, 30.0, 300.0)
    )
)

// Compute column summary statistics.
val summary: MultivariateStatisticalSummary = Statistics.colStats(observations)
println(summary.mean) // a dense vector containing the mean value for each column
println(summary.variance) // column-wise variance
println(summary.numNonzeros) // number of nonzeros in each column
```

代码 1: Scala 代码：colStat 函数使用实例

我们可以看到 summary.mean 输出的是每一列数据的期望，而 summary.variance 计算了每一列的方差，同时 MLlib 也提供了 summary.numNonzeros 来输出该列中非零值的数量；这些函数方便我们对矩阵中各列值有初步的认识。

2.2 Correlations - 相关系数 [4]

相关系数是用以反映两个变量之间相关关系密切程度的统计指标。

2.2.1 Covariance - 协方差

在介绍相关系数之前，我们先引入另一个统计分析量 - 协方差（*Covariance*），用于衡量两个变量 (X, Y) 的总体误差。

定义 1 两个随机变量 X 和 Y 之间的**协方差（*covariance*）**定义为

$$Cov(X, Y) = \mathbb{E}[(X - \mathbb{E}[X])(Y - \mathbb{E}[Y])] = \mathbb{E}[XY] - \mathbb{E}[X]\mathbb{E}[Y] \tag{2.2.1}$$

当 $X = Y$ 时，我们可以看到 X 本身的协方差就是他的方差。如果 X 与 Y 是独立 *(independent)*的，那么二者之间的协方差就是 0，因为两个独立的随机变量满足 $\mathbb{E}[XY] = \mathbb{E}[X]\mathbb{E}[Y]$。但是，反过来并不成立；即如果 X 与 Y 的协方差为 0，二者并**不一定**是独立的。我们只能说这两个随机变量是不相关的。

2.2.2 皮尔逊相关系数

皮尔逊相关系数（*Pearson Correlation Coefficient*）是按两变量 (X, Y) 的协方差指数除以他们的标准差来反映两变量之间相关程度；着重研究线性的单相关系数。皮尔逊相关系数并不是唯一的相关系数，但是最常见的相关系数。

定义 2 两个随机变量之间的**皮尔逊相关系数（*Pearson Correlation Coefficient*）**定义为：

$$\rho_{X,Y} = \frac{Cov(X, Y)}{\sqrt{Var[X]Var[Y]}} \tag{2.2.2}$$

相关系数 $\rho_{X,Y}$ 取值在 -1 到 1 之间，当 $\rho_{X,Y} = 0$ 时，称 X, Y 不相关；$|\rho_{X,Y}| < 1$ 时，X 的变动引起 Y 的部分变动，$\rho_{X,Y}$ 的绝对值越大，X 的变动引起 Y 的变动就越大，他们之间的相关性也就越强。

2.2.3 斯皮尔曼相关系数 [7]

斯皮尔曼相关系数 (Spearman correlation coefficient)是另一种用来衡量变量之间相关度的指标；不同于皮尔逊系数，Spearman 系数并不表示变量之间的线性相关性，而表示了变量之间单调性联系。

Spearman 系数也是继承了 Pearson 的想法来表示其相关性的，但是它并不是直接计算变量 X 和 Y 的相关度，而是通过计算他们的秩序列（*ranked variables*）的 Pearson 系数来判断两者之间的关系。

定义 3 我们将变量 $X = (X_i)_{1 \le i \le n}$ 按数值大小重新排列（顺序或者倒序），$rg(X_i)$ 表示 X_i 在新的序列中的位置。$rg(X)$ 称为变量 X 的**秩序列**。

定义 4 两个随机变量 X, Y 之间的**斯皮尔曼相关系数** ***(Spearman correlation coefficient)*** 表示：

$$r_s(X,Y) = \rho_{rg(X),rg(Y)} = \frac{Cov(rg(X), rg(Y))}{\sigma_{rg(X)}\sigma_{rg(Y)}} \tag{2.2.3}$$

其中 $rg(X), rg(Y)$ 表示关于变量 X 的秩序列；σ 表示了一个变量的**标准差**（***standard deviation***）。

注意这里无论是顺序排列或者倒序排列 X 的值并不会影响他们的 Spearman 相关系数值。

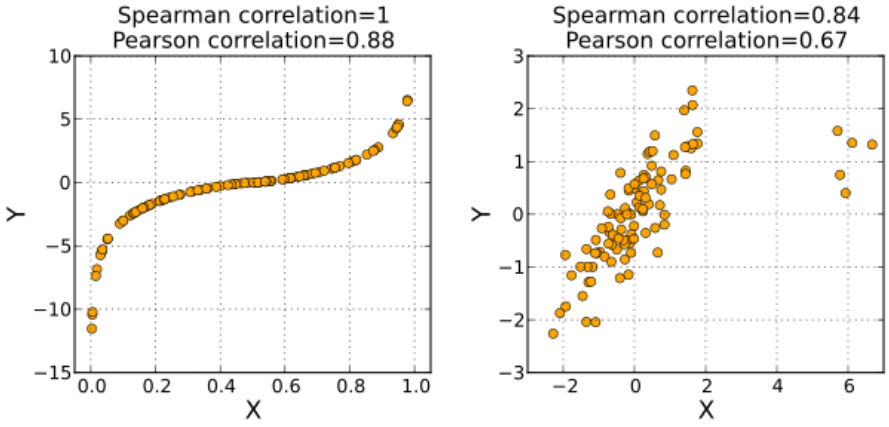


图 1: Spearman vs. Pearson

Spearson 系数更多的反应了两个变量之间的**单调**相关性。换句话说，如果 Y 关于 X 单调递增，那么 $r_s(X,Y) = 1$；如果 Y 关于 X 单调递减，那么 $r_s(X,Y) = -1$。同 Pearson 系数，它的值也在 -1 到 1 之间变化，其绝对值越高，变量之间的相关性越强。相比于 Pearson 系数，他可以更有效的表示非线性相关的递增关系（如平方，开方），异常点对其值的影响较小（如图 1）。

MLlib 函数 在 MLlib 的 Statistics 包中提供了函数corr来计算向量之间的相关性。它的输入有两种方式 (最后一个参数可以选择是计算 Pearson/Spearman 系数)：

1. 计算两个向量之间的相关系数（如代码 2 中的 11-12 行），输出是一个数值；
2. 计算矩阵的各列之间的两两相关系数（第 20 行），输出的是一个**对称**的矩阵，其第 i 行第 j 列表示原矩阵第 i 列和第 j 列之间的相关系数。

```
import numpy as np
from pyspark.mllib.stat import Statistics

seriesX = sc.parallelize([1.0, 2.0, 3.0, 3.0, 5.0]) # a series
# seriesY must have the same number of partitions and cardinality as seriesX
seriesY = sc.parallelize([11.0, 22.0, 33.0, 33.0, 555.0])

# 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.
print("Pearson Correlation is: " + str(Statistics.corr(seriesX, seriesY, method="pearson")))
print("Spearman Correlation is: " + str(Statistics.corr(seriesX, seriesY, method="spearman")))

data = sc.parallelize(
    [np.array([1.0, 10.0, 100.0]),
    np.array([2.0, 20.0, 200.0]),
    np.array([5.0, 33.0, 366.0])]
) # an RDD of Vectors

# calculate the correlation matrix using Pearson's method.
# If a method is not specified, Pearson's method will be used by default.
print(Statistics.corr(data, method="pearson"))
```

代码 2: 相关系数计算使用实例

代码 2 的输出结果如下，可以看到我们所需要的相关关系:

```
Pearson Correlation is: 0.850028676877
Spearman Correlation is: 1.0

[[ 1. 0.97888347 0.99038957]
 [ 0.97888347 1. 0.99774832]
 [ 0.99038957 0.99774832 1. ]]
```

2.3 Stratified sampling - 分层抽样

分层抽样（*stratified sampling*）是将数据集划分为相同标签的子集，然后再在每个子集中进行独立的抽样，组成一个新的样本的统计学计算方法。一般地，在抽样时，将总体分成互不交叉的层，然后按一定的比例，从各层次独立地抽取一定数量的个体，将各层次取出的个体合在一起作为样本，这种抽样方法是一种分层抽样。

分层抽样的特点是将科学分组法与抽样法结合在一起，分组减小了各抽样层变异性

的影响，抽样保证了所抽取的样本具有足够的代表性。

MLlib 函数 分层抽样函数sampleByKey 和sampleByKeyExact可以对 RDD 的**键值对（key-value pairs）**执行，其中 key 用于分类，value 可以是任意数。然后我们通过 fractions 参数（代码 3 行 5）来定义分类条件和采样几率，因此 fractions 参数被定义成一个 Map[K,Double] 类型，Key 是键值的分层条件，Double 是满足该 Key 值条件的采样比例。sampleByKey 和 sampleByKeyExact 的区别在于 sampleByKey 并不对过滤全部数据，因此只得到近似值，而 sampleByKeyExact 会对全局数据做采样计算，因此耗费大量的计算资源，但将提供具有 99.99%置信度的精确抽样大小。

```
1  # an RDD of any key value pairs
2  data = sc.parallelize([(1, 'a'), (1, 'b'), (2, 'c'), (2, 'd'), (2, 'e'), (3, 'f')])
3
4  # specify the exact fraction desired from each key as a dictionary
5  fractions = {1: 0.1, 2: 0.6, 3: 0.3}
6
7  approxSample = data.sampleByKey(withReplacement = False, fractions=fractions)
8  approxSample2 = data.sampleByKey(withReplacement = True, fractions=fractions)
9
10 print(approxSample.collect())
11 print(approxSample2.collect())
```

代码 3: 分层抽样函数 samplebyKey 使用实例

Note: 目前 Spark Python 中不支持 sampleByKeyExact 函数。

其中 withReplacement 参数表示能否重复采样（True 表示可以重复采样，False 不可以重复采样），见输出结果（由于是随机取样，所以每次结果都会有不同）。

```
[(2, 'd'), (2, 'e')]
[(2, 'd'), (2, 'd')]
```

2.4 Hypothesis testing - 假设检验

假设检验 *(Hypothesis Testing)*是统计学中在一定假设条件下由样本推断总体的一种方法。常用的假设检验方法有 μ-检验法、t 检验法、χ^2 检验法 (卡方检验)、F—检验法，秩和检验等。[1]

假设检验的一般步骤：

1. 根据问题的需要对所研究的总体作某种假设，记作 H_0（**Null Hypothesis**），H_1 定义为它的反命题;

2. 计算 H_0 猜想成立之下的统计模型和现实数据之间的差别，利用不同的检测指标来判断两者之间的“距离”（p 值，p-value）；

3. 定义一个阀值 α，如果 $p \leq \alpha$，我们接受假设 H_0; 反之，我们放弃 H_0。

2.4.1 卡方检测 [9]

卡方检测 *(Chi Square Test)*有两种用途，分别是**“拟合优度检定”(Goodness of fit)** 以及**“独立性检定”（independence）**。卡方检验中最常见的就是 Pearson 卡方检测：

利用卡方值检测拟合度

1. 提出假设 H_0：随机变量 X 总体的概率分布函数为 $F(X)$；相对应的，H_1 应该表示为：X 总体的分布函数**不是**$F(X)$。假设 H_0 成立；

2. 若果 X 的取值离散的 k 个值可以直接进行操作；若 X 的值为连续实数，将 X 的取值范围分为 k 个区间：

$$I_1 = (a_0, a_1], I_2 = (a_1, a_2], ..., I_k = (a_{k-1}, a_k)$$

（如果有必要，可以令 $a_0 = -\infty, a_k = +\infty$）；

3. 对于 $\forall 1 \leq j \leq k$，$f_j = card(\{X_i \in I_j / 1 \leq i \leq N\})$ 表示了实际样本中，值落在区间 I_j 的样本个数。而 $p_j = F^{-1}(a_j) - F^{-1}(a_{j-1})$ 表示的是在 H_0 假设下，X 值属于 I_j 的概率。所以在该假设下，I_j 内应该有 $N.p_j$ 的值；

4. 变量 $T = \sum\limits_{1 \leq j \leq k} \frac{(f_j - Np_j)^2}{Np_j}$ 应该满足了 (k-1) 阶的 χ^2 分布。其中 $F_{\chi^2_{k-1}}$ 表示 (k-1) 阶的 χ^2 的分布函数；

5. 定义 p 值（*p-value*）为:$p = \mathbb{P}[X > T] = 1 - F_{\chi^2_{k-1}}(T)$。

6. 给定 $\alpha (= 0.05)$;

7. 如果 $T \geq F^{-1}_{\chi^2_{k-1}}(1 - \alpha) \Leftrightarrow p < \alpha$，我们放弃 H_0；反之我们接受猜想 H_0。

其中变量 X 的概率分布函数（*probability distribution function*）F 在某一点 x 的值表示了变量 X 取值小于等于该点概率，即 $F(x) = \mathbb{P}[X \leq x]$。比如我们通过查表得到对于 5 阶的 χ^2 分布，$F^{-1}_{\chi^2_5}(0.95) = 11.07$。对于 T 这个随机变量，若他符合 5 阶的 χ^2 分布 (即变量 X 分布函数为 F)，那么 $\mathbb{P}[T < 11.07] = 0.95$，当它超过 11.07 时，我们得到了一个小概率事件（概率小于 5%），我们判断 H_0 不成立。

卡方分布表

自由度 \ P	0.995	0.99	0.975	0.95	0.05	0.025	0.01	0.005
1	0.000039	0.0002	0.0010	0.0039	3.8415	5.0239	6.6349	7.8794
2	0.0100	0.0201	0.0506	0.1026	5.9915	7.3778	9.2104	10.5965
3	0.0717	0.1148	0.2158	0.3518	7.8147	9.3484	11.3449	12.8381
4	0.2070	0.2971	0.4844	0.7107	9.4877	11.1433	13.2767	14.8602
5	0.4118	0.5543	0.8312	1.1455	11.0705	12.8325	15.0863	16.7496
6	0.6757	0.8721	1.2373	1.6354	12.5916	14.4494	16.8119	18.5475
7	0.9893	1.2390	1.6899	2.1673	14.0671	16.0128	18.4753	20.2777
8	1.3444	1.6465	2.1797	2.7326	15.5073	17.5345	20.0902	21.9549
9	1.7349	2.0879	2.7004	3.3251	16.9190	19.0228	21.6660	23.5893
10	2.1558	2.5582	3.2470	3.9403	18.3070	20.4832	23.2093	25.1881

利用卡方值检测独立性 在这里我们需要判断两个随机变量 X 和 Y 是否独立：

1. 建立假设 H_0：X 和 Y 独立；则 H_1:X 和 Y 不独立。假设 H_0 成立；
2. 将 X 的值分为 p 个区间（$I_1, ..., I_p$），Y 的值分为 q 个区间（$J_1, ..., J_q$）。记

$$O_{i,j} = Card\{X_n \in I_i, Y_n \in J_j | 1 \le n \le N\}$$

$$O_{i,+} = Card\{X_n \in I_i | 1 \le n \le N\} = \sum_{1 \le j \le q} O_{i,j}$$

$$O_{+,j} = Card\{Y_n \in J_j | 1 \le n \le N\} = \sum_{1 \le i \le p} O_{i,j}$$

$$E_{i,j} = \frac{O_{i,+} \times O_{+,j}}{N}$$

3. T 值定义为：$T = \sum_{i,j} \frac{(O_{i,j} - E_{i,j})^2}{E_{i,j}}$，应该符合 $(p-1) \times (q-1)$ 阶 χ^2 分布。
4. 给定 $\alpha (= 0.05)$;
5. 用 $F_{\chi^2_{(p-1)(q-1)}}$ 表示 (p-1)(q-1) 阶的 χ^2 的分布函数（其值可以通过查表得到），如果 $T \ge F^{-1}_{\chi^2_{(p-1)(q-1)}}(1-\alpha)$，我们放弃 H_0; 反之我们接受 H_0 假设，即 X，Y 独立。

MLlib 函数 spark 通过 Statistics 类来支持 Pearson's chi-squared（卡方检测）。如果chiSqTest函数仅有**一个向量**，那么 MLlib 将利用卡方测试来检测该向量与**正态分布**的拟合程度（如代码 4）：

```
from pyspark.mllib.linalg import Matrices, Vectors
from pyspark.mllib.regression import LabeledPoint
from pyspark.mllib.stat import Statistics

# a vector composed of the frequencies of events
vec = Vectors.dense(0.1, 0.15, 0.2, 0.3, 0.25)
```

```

# 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.
goodnessOfFitTestResult = Statistics.chiSqTest(vec)

# summary of the test including the p-value, degrees of freedom,
# test statistic, the method used, and the null hypothesis.
print("%s\n" % goodnessOfFitTestResult)
```

代码 4: 卡方检测拟合优度实例

```
Chi squared test summary:
method: pearson
degrees of freedom = 4
statistic = 0.12499999999999999
pValue = 0.998126379239318
No presumption against null hypothesis: observed follows the same distribution as expected..
```

从它的输出结果中我们可以看出该测试的自由度（4），卡方值即 T（0.125），p 值（0.9981）以及结论。当然我们也可以稍微改变一下输入的向量，得到一个比较不理想的结果，这个时候 MLlib 就会提示我们应该放弃 H_0，如以下输出：

```
Chi squared test summary:
method: pearson
degrees of freedom = 4
statistic = 9.78125
pValue = 0.044278196743904985
Strong presumption against null hypothesis: observed follows the same distribution as expected..
```

如果chiSqTest函数的输入是一个**两列矩阵**或者两个向量，那么它将测试这两个变量之间的独立性，如代码 5 所示：

```

mat = Matrices.dense(3, 2, [1.0, 3.0, 5.0, 2.0, 4.0, 6.0]) # a contingency matrix

# conduct Pearson's independence test on the input contingency matrix
independenceTestResult = Statistics.chiSqTest(mat)

# summary of the test including the p-value, degrees of freedom,
# test statistic, the method used, and the null hypothesis.
print("%s\n" % independenceTestResult)
```

代码 5: 卡方测试独立性实例

```
Chi squared test summary:
method: pearson
degrees of freedom = 2
statistic = 0.14141414141414144
pValue = 0.931734784568187
No presumption against null hypothesis: the occurrence of the outcomes is statistically independent..
```

如果该函数的输入是一个贴完标签的矩阵（在 MLlib 中表示为一个由 LabeledPoint 类型组成的 RDD），那么它将测试各个特征列与标签列之间的独立性：

```
obs = sc.parallelize([LabeledPoint(1.0, [1.0, 0.0, 2.0]),
    LabeledPoint(1.0, [1.0, 2.0, 2.0]),
    LabeledPoint(0.0, [-1.0, 0.0, 10.0])]
    ) # LabeledPoint(label, feature)

# The contingency table is constructed from an RDD of LabeledPoint and used to conduct
# the independence test. Returns an array containing the ChiSquaredTestResult for every feature
#against the label.
featureTestResults = Statistics.chiSqTest(obs)

for i, result in enumerate(featureTestResults):
    print("Column %d:\n%s" % (i + 1, result))
```

代码 6: 卡方测试标签与特征之间的独立性实例

```
Column 1:
Chi squared test summary:
method: pearson
degrees of freedom = 1
statistic = 3.0000000000000004
pValue = 0.08326451666354884
Low presumption against null hypothesis: the occurrence of the outcomes is statistically independent..
Column 2:
...
Column 3:
Chi squared test summary:
method: pearson
degrees of freedom = 1
statistic = 3.0000000000000004
pValue = 0.08326451666354884
Low presumption against null hypothesis: the occurrence of the outcomes is statistically independent..
```

2.5 Streaming significance testing - 流式显著性检验

显著性检验（significance testing）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。流式检验实现了对数据的实时检验。

该函数在 PySpark 中尚未实现。

2.6 Random data generation - 随机数据生成

MLlib 支持生成各种分布的随机数组来帮助完成测试等功能。MLlib 支持的几种比较常见的概率分布模型（其密度函数图像在下一页）：

名称	类型	取值范围	参数	概率密度函数	期望	方差
均匀分布	离散	$[a,b]$ 间的整数	$a,b\in\mathbb{Z},\ a<b$	$\mathbb{P}[X=k]=\frac{1}{b-a+1}$	$\frac{b+a}{2}$	$\frac{(b-a+1)^2-1}{12}$
泊松分布	离散	$\mathbb{N}$	$\lambda\in\mathbb{R}_+^*$	$\mathbb{P}[X=k]=\frac{\lambda^k}{k!}e^{-k}$	λ	λ
均匀分布	连续	$[a,b]$	$a,b\in\mathbb{R},\ a<b$	$f(x)=\frac{1}{b-a}$	$\frac{b+a}{2}$	$\frac{(b-a)^2}{12}$
指数分布	连续	$\mathbb{R}_+$	$\lambda>0$	$f(x)=\lambda e^{-\lambda x}$	$\frac{1}{\lambda}$	$\frac{1}{\lambda^2}$
正态分布	连续	$\mathbb{R}$	$\mu\in\mathbb{R},\sigma>0$	$f(x)=\frac{1}{\sigma\sqrt{2\pi}}exp(-\frac{(x-\mu)^2}{2\sigma^2})$	μ	σ^2
对数正态分布	连续	$\mathbb{R}_+$	$\mu\in\mathbb{R},\sigma>0$	$f(x)=\frac{1}{x\sigma\sqrt{2\pi}}exp(-\frac{(\ln(x)-\mu)^2}{2\sigma^2})$	$e^{\mu+\sigma^2/2}$	$(e^{\sigma^2}-1)e^{2\mu+\sigma^2}$
伽马分布	连续	$\mathbb{R}_+$	$k>0,\theta>0$	$f(x)=\frac{x^{k-1}e^{-x/\theta}}{\Gamma(k)\theta^k}$	$k\theta$	$k\theta^2$

表 1: 常见概率分布

MLlib 函数 我们最常见的概率模型当属正态分布模型（*Normal distribution/Gaussian distribution*）。RandomRDDs 类中的normalRDD函数可以帮助我们生成一组符合 $\mathcal{N}(0,1)$ 的随机数据，size 参数表示数据的大小；而 numPartitions=10 表示生成了 10 组 $\mathcal{N}(0,1)$ 随机数据。通过 map 函数来自定义参数（表 4 中的 μ,σ）。

```
from pyspark.mllib.random import RandomRDDs

# Generate a random double RDD that contains 1 million i.i.d. values drawn from the
# standard normal distribution ‘N(0, 1)‘, evenly distributed in 10 partitions.
norm = RandomRDDs.normalRDD(sc, size=10000L, numPartitions=10)

# Apply a transform to get a random double RDD following ‘N(1, 4)‘.
v = norm.map(lambda x: 1.0 + 2.0 * x)
```

代码 7: 生成正态分布随机数据实例

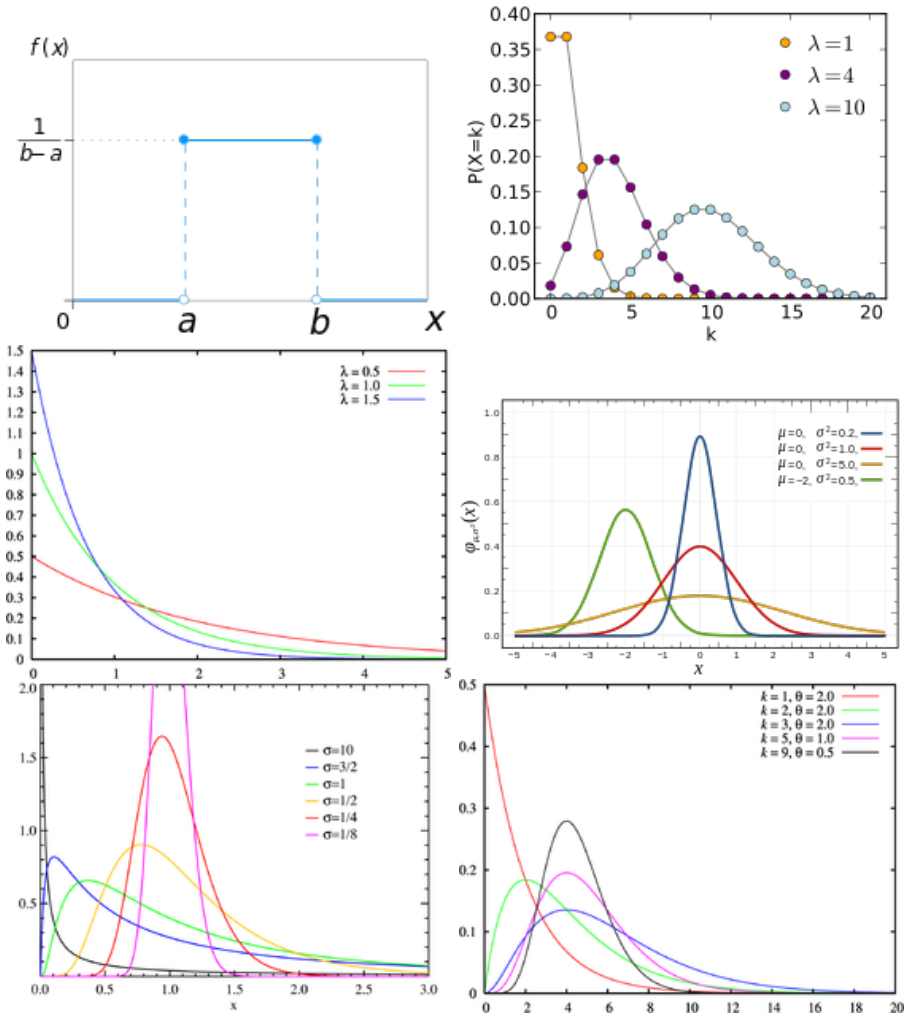


图 2: 常见概率分布的密度函数图像

同时 MLlib 也提供了其他函数来生成符合多种其他概率分布的数据（代码 6）

```
#Other distributions
uniform = RandomRDDs.uniformRDD(sc, size=100L)  #by default U(0.0, 1.0)
uniformGene = uniform.map(lambda v: 10.0 + (20.0 - 10.0) * v)
poisson = RandomRDDs.poissonRDD(sc, mean=1, size=100L)
exp = RandomRDDs.exponentialRDD(sc, mean=1, size=100L)
gamma = RandomRDDs.gammaRDD(sc, shape=1, scale=1, size=100L) #shape=k,scale=theta
logNorm = RandomRDDs.logNormalRDD(sc, mean=0, std=2, size=100L)
```

代码 8: 其他概率分布随机数据实例

3 Classification and regression - 分类和回归算法

分类（*classification*）和回归算法（*regression*）是机器学习中两种最常见的**监督学习（supervised learn）**类型。两者之间相似但也存在不同，一般来说，对待这两种问题，我们首先需要一定量的已知数据：$(x^{(i)}, y^{(i)})_{i=1,2,...,N}$。其中 $x^{(i)} \in \mathbb{R}^m$ 称

为数据的**特征量（features/attributes）**，$y^{(i)}$ 称为数据的**标签（labels）**。无论分类还是回归，算法的目的都是寻找 $y^{(i)}$ 和 $x^{(i)}$ 之间的关系，寻找一个函数 f^* 满足 $\forall 1 \leq i \leq N, y^{(i)} = f^*(x^{(i)})$，并且对于之后新的 x'，可以做一个比较好的预测。针对不同性质的标签类，我们将问题分为两类：

- 如果 $y^{(i)}$ 的值是**离散**的，$y^{(i)} \in \{1, ...K\}$ 比如对性别/颜色的预测，我们将这类问题称为**分类（Classification）**问题。分类问题中比较典型的是二分类问题（binary classification），因为他可以快速转化为多分类的一般问题。
- 如果 $y^{(i)}$ 的值是**连续**的，$y^{(i)} \in \mathbb{R}$ 比如对年龄/股价的预测，我们将这类问题称为**回归（Regression）**问题。

当然我们可以找到一个非常复杂的函数使得对于训练集的所有数据都满足 $y^{(i)} = f^*(x^{(i)})$，但这样的 f^* 不一定能对后续的数据做出很好的预测。所以我们会放宽要求使得 $\forall 1 \leq i \leq N, y^{(i)} \approx f(x^{(i)})$，如何寻找合适的函数 f 是我们研究的主要目的。

数据分组 利用已有数据寻找函数 f 的过程，我们称为**训练（Train）**。为了方便我们对找出的 f 函数的性能做一个初步评估，我们会将已知数据分为两大类：训练集（*Training set*）和测试集（*Testing set*）。由于我们在训练过程中需要对一些参数做预估，经常会用到**交叉验证（cross validation）**的方法，因此需要另一个数据集来对这些参数做评估，称为验证集（*validation set*）。当然我们可以独立的分出一个验证集来测试参数，也可以直接拿测试集当做验证集使用，这在结果上并没有什么特别大的不同。只是根据各人习惯不同，我们将数据集分为几个部分：

- 分为两个部分：训练集 (80%) 和测试集 (20%)。划分比例可以调整。
- 分为三个部分：训练集 (70%) 和验证集 (10%) 和测试集 (20%)。比例也可以根据个人或者数据大小调整。在一些比较复杂的函数模型中（比如多层神经网络），分为 3 个数据集会比较有必要。
- 分为 k 个部分：**k 折交叉验证（K-fold cross validation）**的基本思想：我们利用（k-1）个数据集做训练，剩下的一个做测试；重复 k 次，取其错误率的平均值，得到一个比较平滑的结果。当然这种方法消耗较大的计算成本。

做数据分析时，可以根据自己的习惯和需求去制定分组方案，这里并没有绝对正确的方法，对结果的影响也不是特别大，可以选择适合自己的方法，但分组是必不可少的。

多分类问题 二分类问题（*binary classification*）是分类的基础，很多基础算法（svm，LR）都会拿来解决二分类问题。那么如何利用二分类的结果来处理现实中广泛存在的多分类问题（multiple classification）呢？对于一个 K 分类的问题，训练集（$x^{(i)}, y^{(i)}$）我们选择了一种二分类的算法 B。一般来说我们有两种思想可以实现这一步骤：

- one vs. all 算法: 针对某一类计算数据属于则一类的可能性大小，选出可能性最大的一类（需要运行 K 次二分类算法）

Algorithm 1 one vs. all 算法

for $k \in \{1, ..., K\}$ **do**

建立一个新的标签列，$z^{(i)} = \mathbb{I}_{\{y_i=k\}}$;

利用算法 B 对数据 $(x^{(i)}, z^{(i)})$ 训练出模型 f_k;

计算针对测试数据 x 属于类别 k 的 score $s_k = f_k(x)$;

end for

测试数据的类别预测为 $\hat{y} = argmax_k(f_k(x))$

- one vs. one 算法：类似上一算法，将所有类别两两作比较，判断测试数据属于哪一类的可能性较大，从而选择最优的类。但是该算法需要运行 $K(K-1)/$ 次二分类算法。

3.1 Linear models - 线性模型

线性模型（*Linear Model*）是解决机器学习问题算法中最简单的模型，它可以很简便的解决一些线性规划/线性可分的问题，但是它的局限性也在于只能用于解决一些线性的问题，在面对一些非线性的问题时精确度较低。

线性模型的预测函数一般可以表示为 $y^{(i)} = f(x^{(i)}) = \theta^T x^{(i)} + b$（在二维空间中是一条直线，多维空间中表示一个超平面（*hyper-plane*）），其中 θ 是个 m 维向量，b 是个常数。但为了表述方便，我们通常对 $x^{(i)}$ 和 θ 做一下变形：在 $x^{(i)}$ 首项加一个 1，在 θ 首项前加入 b。于是我们就可以将模型简单的表示成：

$$\forall 1 \leq i \leq n,\ f(x^{(i)}) = \theta^T x^{(i)} \tag{3.1.1}$$

此时 θ 是一个 m+1 维的向量。这样我们既简化了函数的形式，也减少了参数的数量。在后文的描述中，若没有特殊说明，我们都将默认采取这种变形。我们也会将这个训练

集写成矩阵的形式，方便书写：$Y = X.\theta$，其中

$$Y = \begin{pmatrix} y^{(1)} \\ y^{(2)} \\ \dots \\ y^{(n)} \end{pmatrix} \in \mathbb{R}^n \quad X = \begin{pmatrix} x^{(1)T} \\ x^{(2)T} \\ \dots \\ x^{(n)T} \end{pmatrix} = \begin{pmatrix} 1 & x_1^{(1)} & \dots & x_m^{(1)} \\ 1 & x_1^{(2)} & \dots & x_m^{(2)} \\ \dots & \dots & \dots & \dots \\ 1 & x_1^{(n)} & \dots & x_m^{(n)} \end{pmatrix} \in \mathbb{R}^{n \times (m+1)}$$

代价函数 代价函数（*loss function*）J 是统计学中一种衡量损失和错误程度的函数。之前我们提过回归和分类函数的目的是寻找一个合适的函数 f 使得 $\forall 1 \leq i \leq N, y^{(i)} \approx f(x^{(i)})$；那么代价函数就是用来判断函数和训练集数据的拟合程度的，J 的函数值越小，则说明了拟合程度越好，f 也就越“合适”。于是很多时候，我们将工作重心转向寻找代价函数的最小值（一个**凸优化**问题）。

那么怎么定义一个模型的代价函数呢？在线性模型中几种常见的代价函数类型：

名称	适用类型	代价函数	梯度/次梯度
squared loss	回归问题	$J(\theta) = \frac{1}{2n}\|X\theta - Y\|_2^2$	$X^T(\theta X - Y)$
hinge loss	二分类问题	$J(\theta) = \sum_i max(0, 1 - y^{(i)}\theta^T x^{(i)})$	$-\sum_i y^{(i)} x^{(i)} \mathbb{I}_{\{y^{(i)}\theta^T x^{(i)} < 1\}}$
logistic loss	二分类问题	$J(\theta) = \sum_i log_2[1 + exp(y^{(i)}\theta^T x^{(i)})]$	$\sum_i y^{(i)}(1 - \frac{1}{1+exp(y^{(i)}\theta^T x^{(i)})})x^{(i)}$

表 2: 几种常见的代价函数

上表中的二分类问题主要针对标签 $y^{(i)} \in \pm 1$, 而我们所做的预测值就可以表示为 $y^{(i)} = f(x^{(i)}) = sign(\theta^T x^{(i)})$。一般来说，如果 $y^{(i)}\theta^T x^{(i)} \geq 0$ 我们的分类就是正确的，而如果 $y^{(i)}\theta^T x^{(i)} < 0$，我们的预测错误。所以从关于 $y^{(i)}\theta^T x^{(i)}$ 的图像 [6] 上来看：

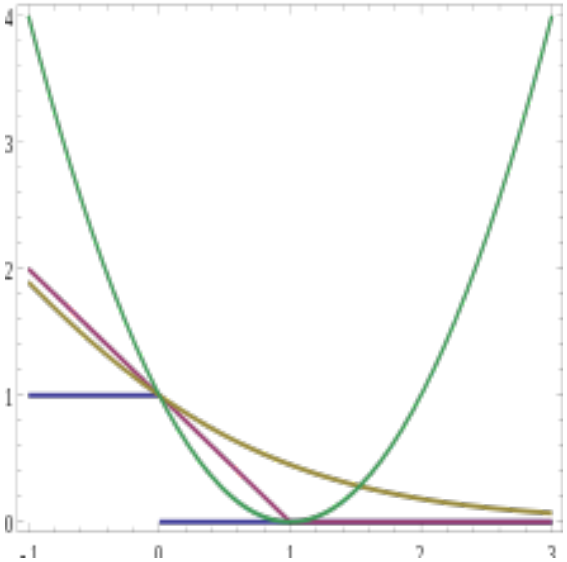


图 3: 二分类问题的代价函数

- 蓝线表示我们理想的代价函数 (以 0 为分界线，负的代价为 1，正的代价为 0)；
- 绿线为 squared loss 的值，很明显它在这里不是很适用；
- 黄线为 logistic loss 的值，比较贴近目标函数并且是一个连续的凸函数；
- 紫色为 hinge loss 的值，类似于 logistic loss，比较符合二分类的代价函数。

正则化 在分类回归问题中，我们经常会碰到模型的“过拟合”（*overfitting*）的情况。什么是 overfitting？overfitting 是指我们的训练过程在数据上进行的太过“完美”了，将训练数据的所有特征全部包含在模型函数 f 中。过于“完美”的模型在测试集上的精度会非常糟糕。图 4 中，左图表示我们的模型还不够好导致分类错误过多，属于**欠拟合**；中图是一个比较好的分类器；而右图也是一个糟糕的分类器，虽然它能将数据很好的分类但他并不能很好的做出预测，就是一个典型的**过拟合**问题。

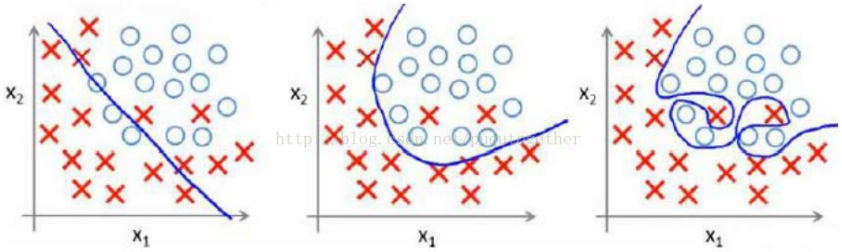


图 4: 各种类型的分类器

其实在训练数据的所有特征中，可以分为**全局特征**和**局部特征**两种。其中全局特征是我们解决分类/回归问题时所需要的，而局部特征是训练集特有的，并无法表现在测试集上。然而机器没有很好的办法区别这两种特征，所以有时候生成模型的时候会选择过多的局部特征使得模型过于复杂。这就是过拟化情形产生的原因。

例 1 我们根据利用的一些特征利用机器学习来区分熊和其他动物，得到了特征：哺乳动物，头大，尾巴短，四肢短而粗，脚掌大，趾端有带钩的爪等等。到了这一步我们已经能比较好的区别一头熊了。

但很不巧的是我们选取的训练样本中只有黑熊，棕熊，灰熊，而没有北极熊。于是我们的机器就会增加一条规则来定义熊：颜色只能为黑/棕/灰。这个时候北极熊就被“残忍”地排除在了熊这一类中。这个时候其实颜色就属于一个**局部特征**，我们得到的模型就产生了**过拟化**的问题。

那么如何防止过拟化的产生呢？这个时候我们就引入了正则化（*Regularization*）的方法。通过对一些特征量的“约束”，我们只选择哪些比较重要的特征（一般都是全

局特征），限制那些局部特征，从而达到简化模型的目的。具体的做法其实很简单，我们在算法的代价函数上增加一个正则项：

$$J(\theta) = J_{old}(\theta) + \lambda r(\theta)$$

其中 λ 称为正则化系数，$r(\theta)$ 则是正则项。λ 越大，我们对特征的“约束”也就越大；反之，这个“约束”也就比较小。而正则项一般是 θ 的模函数。

	$r(\theta)$	梯度/次梯度
L_2	$\frac{1}{2}\|\theta\|_2^2$	θ
L_1	$\|\theta\|_1$	$sign(\theta)$
elastic net	$\alpha\|\theta\|_1 + (1-\alpha)\frac{1}{2}\|\theta\|_2^2$	$\alpha sign(\theta) + (1-\alpha)\theta$

表 3: 几种常见的正则化类型

我们从线性模型的角度上来理解一下这个正则项：

- 首先需要注意，如果使用正则化，那么对原数据采取标准化操作（比如使各列都在 -1 到 1 间取值，但不丢失其原特性）比较重要。
- 对于特征 x_k, 如果 $|\theta_k|$ 的值越大，那么该特征在表示 y 的时候就越重要；
- 如果有过多的特征都显得比较重要，那么 $r(\theta)$ 的值就会变大从而影响新的 $J_(\theta)$ 值;
- 因此最小化新的代价函数的同时，我们会考虑到模型的复杂度。

由于 θ_0 不指向任意特征，因此在正则项中一般不包含 θ_0。

一般来说，L_1 正则比 L_2 正则的结果更加“稀疏”，即提取的特征更少；而在计算上 L_2 正则更加简洁，可以节约大量额计算成本。而 elastic net 是两者的结合。

3.1.1 支持向量机

支持向量机（*Support Vector Machine*, *svm*）是一个经久不衰的算法，最初由 Vladimir N. Vapnik 和 Alexey Ya. Chervonenkis 于 1962 年提出，经过多年的发展，在今天的机器学习领域仍然保持了十足的活力。从直观的角度来看，SVM 从众多的符合条件的线性分类器（*linear classifier)*选择出最适合的一个：下图中绿色的分割线并不能很好地将不同的点分离开来，而红线和蓝线都可以达到分离的效果，但是从直观上来说，红线的分离效果要优于蓝线。

经典 SVM 就是通过**最大化决策边界（maximum margin）**的边缘来找到最适合的分类器。**决策边缘**是指距离分割器最近的点到分割器的距离。如果我们将线性模型记

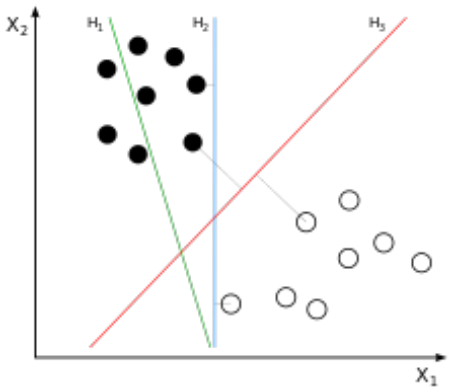


图 5: 线性分类器

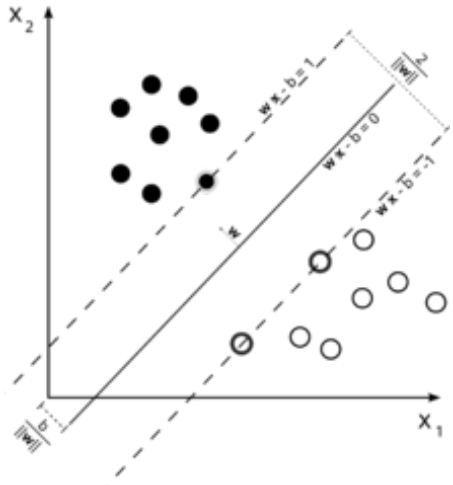


图 6: 决策边缘

为 $y = \theta^T x + \theta_0$（注意：这里我们更倾向于保留参数 θ_0）。我们知道一个点 (x, y) 到该超平面 $\theta^T x + \theta_0 = 0$ 的距离为表示为 $\frac{y(\theta^T x + \theta_0)}{\|\theta\|}$。那么我们的目的最大化决策边缘在数学上表示为：

$$\underset{\theta,\theta_0}{argmax}(min\{\forall 1 \le i \le n, \frac{1}{\|\theta\|} y^{(i)}(\theta^T x^{(i)} + \theta_0)\})$$

由于我们只需要确定分类器的方向，所以可以改变 θ 的模的大小，令它的决策边缘大小为 $\frac{2}{\|\theta\|}$；此时，在决策边缘上的点正好符合条件，$\theta^T x^{(i)} + \theta_0 = \pm 1$，这些点就被叫做支持向量（support vector）。于是 SVM 的数学问题转化为了

$$\max_{\theta} \frac{1}{\| \theta \|} \quad s.t. \quad \forall 1 \le i \le n,\ y^{(i)}(\theta^T x^{(i)} + \theta_0) \ge 1 \tag{3.1.2}$$

在这个优化问题中，前一部分表示将决策边缘最大化，后一部分的条件是为了保证决策边缘内并没有样本存在 *(硬间隔，hard margin)*。但是在很多情况下，我们可以容许决策间隔区域中存在一定数量的点（软间隔 *svm, soft margin svm)*：

$$\min_{\theta} \| \theta \| \quad s.t. \quad \forall 1 \le i \le n, y^{(i)}(\theta^T x^{(i)} + \theta_0) \ge 1 - \xi_i$$

关于 ξ_i（**slack variables**）的取值，如果 $y^{(i)}(\theta^T x^{(i)} + \theta_0) \geq 1$，那么 $\xi_i = 0$；否则 ξ_i 取可能的最小值即 $1 - y^{(i)}(\theta^T x^{(i)} + \theta_0)$。综上，$\xi_i = \max\{0, 1 - y^{(i)}(\theta^T x^{(i)} + \theta_0)\}$。

但是要注意在后面的优化计算中我们需要“惩罚”这些点来保证分类器的精确度。于是我们将优化问题转化为了

$$\min_{\theta} \frac{1}{2} \parallel \theta \parallel^2 + C\frac{1}{n}\Sigma_i \xi_i \quad s.t. \quad \forall 1 \leq i \leq n, y^{(i)}(\theta^T x^{(i)} + \theta_0) \geq 1 - \xi_i$$

在这个问题中，我们也许可以注意到 $\frac{1}{2}\|\theta\|^2$ 其实就是我们之前介绍的 L_2 正则项（当然也可以用 L_1 正则），而 $\frac{1}{n}\Sigma_i \xi_i$ 是一个 hinge loss（（ξ_i 的定义完全符合）。因此如果我们将这个问题等价于一个比较熟悉的来表达：

$$\min_{\theta} J(\theta) = \frac{1}{n}\Sigma_i \xi_i + \frac{1}{2}\lambda \parallel \theta \parallel^2 \quad s.t. \quad \forall 1 \leq i \leq n, y^{(i)}(\theta^T x^{(i)} + \theta_0) \geq 1 - \xi_i \qquad (3.1.3)$$

这里，我们得到了一个凸优化问题，可以使用对应的最优化算法求得近似解。

PS：目前很多 svm 的 package 都采用 SMO（Sequential Minimization Optimization）算法来实现优化，该算法可以同时优化 2 个参数，大大提高优化效率。

核函数 在 svm 问题中引进**核函数（kernel function）**的概念可以帮助 svm 解决那些线性不可分的问题，对于 svm 的推广有着重要意义。

可惜 MLlib 中的 svm 算法还没有引入这一功能。

MLlib 函数 在 Mllib 中定义了SVMWithSGD函数帮助我们解决二分类的问题，其中 SGD 表示该函数中应用了 Sub-Gradient Descents 的优化方法来求解凸优化问题。

```
from pyspark.mllib.classification import SVMWithSGD, SVMModel
from pyspark.mllib.regression import LabeledPoint

# Load and parse the data
def parsePoint(line):
    values = [float(x) for x in line.split(' ')]
    return LabeledPoint(values[0], values[1:])

data = sc.textFile("D:/spark/data/mllib/sample_svm_data.txt")
parsedData = data.map(parsePoint)

# Build the model
model = SVMWithSGD.train(parsedData, iterations=100)
```

代码 9: SVM 分类器应用实例

首先，对于一个分类问题，MLlib 中的函数接受的数据集必须是一个 LabeledPoint 类的向量。所以代码 9 中的 5-7 行 parsePoint 函数就实现了从一行数字到一个 LabeledPoint 类的转换。对于 MLlib 提供的数据 sample_svm_data.txt，其中有 322 行（无列名）和 17 列数据。其中第一列是二分（0/1）的标签列，其余 16 列都是值为实数的特征列。所以建立的 LabeledPoint 的格式为 LabeledPoint(row[0], row[1:])。

最后我们得到的 svm 模型应该是一个 16 维向量 θ 和常数 θ_0。

我们具体来看一下 SVMWithSGD.train 函数定义时的各个参数

```
class SVMWithSGD(object):
    def train(cls, data, iterations=100, step=1.0, regParam=0.01,
        miniBatchFraction=1.0, initialWeights=None, regType="l2",
        intercept=False, validateData=True, convergenceTol=0.001):
```

1. data：即训练集数据，LabeledPoint 类的 RDD；
2. iterations：SGD 算法中的最大迭代次数，默认值为 100；
3. step：SGD 算法中的步长值，即学习率 α，默认值为 1；
4. regParam：正则参数 λ，默认为 0.01；
5. miniBatchFraction：SGD 算法每次迭代中使用的训练数据的比例（用于计算梯度），默认值为 1 即全部数据；（影响较小）
6. initialWeights：θ 的初始值，默认为 None，随机初始值；（影响较小）
7. regType：正则类型，可选项：‘None’，‘l1’，‘l2 ‘（默认）；
8. intercept：是否计算 θ_0 的值，默认为 False，即 θ_0 始终为 0；
9. validateData：逻辑参数表示是否需要在训练之前判断数据是否符合标准，默认值为 True；（影响较小）
10. convergenceTol：判断收敛条件的参数，默认为 0.001。

当然 MLlib 也提供了预测函数帮助我们对新的数据进行预测。

```
# Evaluating the model on training data
labelsAndPreds = parsedData.map(lambda p: (p.label, model.predict(p.features)))

trainErr = labelsAndPreds.filter(lambda (v, p): v != p).count() / float(parsedData.count())

print("Training Error = " + str(trainErr))
```

代码 10: 测试 SVM 模型

```
hjw ===> Testing Error with 100 iterations = 0.38198757764
hjw ===> Testing Error with reg param 0.25 = 0.378881987578
hjw ===> Testing Error with L1 regularization = 0.39751552795
```

MLlib 中的 svm 训练方法并不是很成熟，首先它没有引入**核函数**的概念，导致它只能处理线性的问题。其次 SGD 算法在处理高维大样本数据时效率和效果都不理想，远不如 **smo 优化算法**。其实 MLlib 中的 svm 训练算法比常用的如 libsvm 效果会差很多。

3.1.2 逻辑回归

首先要注意的一点，这里的*逻辑回归*（*logistic regression*）并**不是**一种回归算法，而是一种二分类算法中。

逻辑回归的目的也是寻找一个线性的函数模型。但在二分类算法中，我们的预测值不应再是一个实数，而更多是样本属于正类的概率。所以需要一个函数 g 从实数映射到 $[0,1]$ 的区间，$g:\mathbb{R}\mapsto[0,1]$，在这里我们定义了一个符合该条件的 *S 型函数*（*sigmoid function/logistic function*）：

$$g(t)=\frac{1}{1+e^{-t}}$$

所以逻辑回归模型给我提供了一个在 0 和 1 之间的连续实数，这也是为什么它被取名为回归的原因。

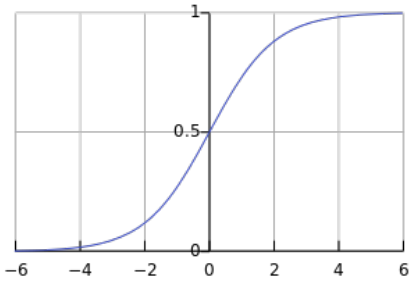


图 7: S 型函数

我们利用线性模型，套用上 S 型函数来获得样本属于正类的概率预测，即

$$g_\theta(x)=\frac{1}{1+e^{-\theta^T x}}=\mathbb{P}[y=1|x;\theta] \tag{3.1.4}$$

我们需要找到一个合适的 θ^* 使得代价函数最小。在这里我们对每个样本的代价函数 (即 **logistic loss**) 定义为

$$Cost(g_\theta(x),y)=\begin{cases}-log(g_\theta(x)) & if \quad y=+1\\ -log(1-g_\theta(x)) & if \quad y=-1\end{cases}$$

对于全局的代价函数应该是

$$
\begin{aligned}
J(\theta) &= \frac{1}{n}\Sigma_{i=1}^{n} Cost(g_\theta(x^{(i)}), y^{(i)}) \\
&= -\frac{1}{2n}\Sigma_{i=1}^{n}[(1+y^{(i)})log(g_\theta(x^{(i)})) + (1-y^{(i)})log(1-g_\theta(x^{(i)}))]
\end{aligned}
$$

再加上我们所需要的正则项和正则系数，利用梯度下降法可以求出最优的 θ^*，如果有新的样本 $\hat{x}$ 需要利用模型进行预测，我们定义预测值：

$$
\hat{y} = \begin{cases} -1 & if \quad g_\theta(\hat{x}) < 0.5 \\ +1 & if \quad g_\theta(\hat{x}) \geq 0.5 \end{cases}
$$

MLlib 函数 MLlib 中的 classification 类中提供了LogisticRegressionWithLBFGS.train函数来训练逻辑回归模型。在这个类中我们使用了 LBFGS 算法来寻求最优解。我们也有类 LogisticRegressionWithSGD 使用了 SGD 算法来求解。

```
from pyspark.mllib.classification import LogisticRegressionWithLBFGS
from pyspark.mllib.regression import LabeledPoint

def parsePoint(line):
    values = [float(x) for x in line.split(' ')]
    return LabeledPoint(values[0], values[1:])

data = sc.textFile("D:/spark/data/mllib/sample_svm_data.txt")
parsedData = data.map(parsePoint)

# Build the model
model = LogisticRegressionWithLBFGS.train(parsedData)
```

代码 11: 逻辑回归分类应用实例

同上一节中的 svm，我们需要将一般数据转换为 LabeledPoint 后在使用对应函数进行训练。而原函数的所有参数

```
class LogisticRegressionWithLBFGS(object):
    def train(cls, data, iterations=100, initialWeights=None, regParam=0.0, regType="l2",
        intercept=False, corrections=10, tolerance=1e-6, validateData=True, numClasses=2):
```

1. data：即训练集数据，类型为 LabeledPoint 的向量；
2. iterations：LBFGS 算法中的最大迭代次数，默认值为 100；
3. initialWeights：θ 的初始值，默认为 None，即不提供初始值；（影响较小）
4. regParam：正则参数 λ，默认为 0.01；

5. regType：正则类型，可选项：‘None’，‘l1’，‘l2 ‘（默认）；
6. intercept：是否计算 θ_0 的值，默认为 False 即 θ_0 始终为 0；
7. corrections: L-BFGS 算法中存储中间值的步数，即算法中的 m 值，默认值为 10；（影响较小）
8. tolerance: L-BFGS 的终止条件判断，默认值为 10^{-6};
9. validateData：boolean 类表示是否需要在训练之前判断数据是否符合标准，默认值为 True；（影响较小）
10. numClasses：分类个数，可支持多分类，默认为 2。

```
Training Error with LBFGS= 0.366459627329
Training Error with SGD= 0.363354037267
```

可以看到在这里，使用 SGD 还是 L-BFGS 算法对精确度的影响不是很大。但一般来说 L-BFGS 算法性能要优于 SGD，特别是在针对大型数据集的计算上，可以提升收敛速度，并节省内存。

总的来说，MLlib 中的逻辑回归函数已经比较成熟；支持 L-BFGS 优化算法，而且能够对**多分类**问题进行处理。

3.1.3 线性回归

线性回归是（Linear Regression）最常见也是最容易实现的回归算法。其思想是针对 m 个样本 $(x^{(i)}, y^{(i)}) \in \mathbb{R}^n \times \mathbb{R}$ 寻找 $y^{(i)}$ 和 $x^{(i)}$ 之间的线性关系，用向量的形式来表达：

$$Y = X.\theta^T \tag{3.1.5}$$

所以我们希望找到一个 θ^* 尽量契合源数据，使得代价函数（squared loss）$J(\theta) = \frac{1}{2} \parallel Y - X\theta \parallel^2 = \frac{1}{2m}\Sigma_{i=1}^{n}(y(i) - \theta^T x^{(i)})^2$ 最小。

在此基础上我们需要加入正则项，避免我们的模型出现过拟合的问题（以 L_2 正则项为例）：

$$J(\theta) = \frac{1}{2m}\Sigma_{i=1}^{n}(y(i) - \theta^T x^{(i)})^2 + \frac{\lambda}{2}\|\theta\|_2^2$$

数学上，对加入 L_2 正则项的回归叫做 **ridge 回归**；而使用 L_1 正则的回归类型叫做 **lasso 回归**。

对于这样一个经典的凸优化问题，我们可以用 SGD 优化算法得到一个比较满意的结果。

MLlib 函数 MLlib 中的 regression 包中提供了LinearRegressionWithSGD类以供我们来训练并预测一个线性规划的模型。

```
from pyspark.mllib.regression import LinearRegressionModel, LinearRegressionWithSGD
from pyspark.mllib.classification import LabeledPoint

# Load and parse the data
def parsePoint(line):
    values = [float(x) for x in line.replace(',', ' ').split(' ')]
    return LabeledPoint(values[0], values[1:])

data = sc.textFile("data/mllib/ridge-data/lpsa.data")
parsedData = data.map(parsePoint)

# Build the model
model = LinearRegressionWithSGD.train(parsedData, iterations=100, step=0.00000001)
```

代码 12: 线性回归模型使用实例

对于回归函数，我们的输入数据同样要求是 LabeledPoint 类型的向量，当然在这里，这个 label 可以是一个实数。看一下原函数定义时的各个参数：

```
class LinearRegressionWithSGD(object):
    def train(cls, data, iterations=100, step=1.0, miniBatchFraction=1.0,
    initialWeights=None, regParam=0.0, regType=None, intercept=False,
    validateData=True, convergenceTol=0.001):
```

1. data：即训练集数据，类型为 LabeledPoint 的向量；
2. iterations：SGD 算法中的最大迭代次数，默认值为 100；
3. step：SGD 算法中的步长值，即学习率 α，默认值为 1；
4. miniBatchFraction：SGD 算法每次迭代中使用的训练数据的比例（用于计算梯度），默认值为 1 即全部数据；（影响较小）
5. initialWeights：θ 的初始值，默认为 None，即不提供初始值；（影响较小）
6. regParam：正则参数 λ，默认为 0.0；
7. regType：正则类型，可选项：‘None’（默认），‘l1’，‘l2 ‘；
8. intercept：是否计算 θ_0 的值，默认为 False 即 θ_0 始终为 0；
9. validateData：boolean 类表示是否需要在训练之前判断数据是否符合标准，默认值为 True；（影响较小）

10. convergenceTol：判断收敛条件的参数，默认为 0.001。

```
# Evaluate the model on training data
valuesAndPreds = parsedData.map(lambda p: (p.label, model.predict(p.features)))
MSE = valuesAndPreds \
    .map(lambda (v, p): (v - p) ** 2) \
    .reduce(lambda x, y: x + y) / valuesAndPreds.count()

print("Mean Squared Error without Regularization = " + str(MSE))
```

代码 13: MLlib 测试线性回归模型

同其他函数，我们可以利用结果来计算该模型在原数据上的方差。当然我们同样可以利用训练出来的模型来测试新的数据：

```
Mean Squared Error without Regularization = 6.83324753751
Mean Squared Error with Regularization = 6.83550847274
```

3.1.4 流式线性规划

当数据是以流的方式进来时，那么选择合适的回归模型是很有用的，MLlib 目前支持使用普通最小二乘法实现流式回归。如果加载训练数据和测试数据从两个不同的输入流进入，将流解析为标签点，对第一个流使用线性回归模型，并预测第二个流。

如果有新的文件放入/training/data/dir 中模型就会自动更新，如果有新的文件放入/testing/data/dir 中就会看到预测结果，如果测试目录中的数据越多，那么预测就会更准确。

该功能目前只支持 scala 语言。

3.2 Naive Bayes - 朴素贝叶斯分类器

一些概率学概念

朴素贝叶斯（*Naive Bayes*）模型是一个广泛应用于**自然语言处理（NLP）**且非常容易理解的**概率模型**。为了更好的认识这一模型，我们需从概率学上的一些基本定义讲起。

定义 5 *对于两个事件 A,B; 事件 A 在事件 B 成立的条件下发生的概率，称为 A 在 B 下的***条件概率（*conditional probability*）***，其值定义为*

$$\mathbb{P}[A|B] = \frac{\mathbb{P}[AB]}{\mathbb{P}[B]} \tag{3.2.1}$$

其中 $\mathbb{P}[AB]$ 表示事件 A 和 B 同时发生的概率。

条件概率也是概率的一种，对于事件 A 的所有可能性，它们关于事件 B 的条件概率总和值也是 1。

定义 6 事件 A 与事件 B **相互独立**（***independent***）定义为 $\mathbb{P}[AB] = \mathbb{P}(A)\mathbb{P}(B)$。

根据条件概率的定义我们不难得出

$$A, B \text{ 相互独立} \quad \Leftrightarrow \quad \mathbb{P}[A|B] = \mathbb{P}[A]$$

了解了这两个概率学上的基础定义之后，我们可以来介绍一下，大名鼎鼎的贝叶斯定理，这也是朴素贝叶斯算法的精髓所在：

定理 3.1 对于两个事件 A, B，我们有**贝叶斯定理**（***Baye's Theorem)***:

$$\mathbb{P}[A|B] = \frac{\mathbb{P}[B|A].\mathbb{P}[A]}{\mathbb{P}[B]} \tag{3.2.2}$$

这个公式从定义上不难推导，它的意义在于当我们的 $\mathbb{P}[A|B]$ 很难直接求得而 $\mathbb{P}[B|A]$ 比较容易得到时，可以实现这两者之间的一个计算转换。

算法原理

现在回到我们的机器学习之中，来讲一下朴素贝叶斯的分类思想。这里我们最好是先从该算法的预测方法说起，假设我们已经对训练数据集完成了训练过程，得到了一个贝叶斯模型。

现在我们有了一个新的数据 x 的各个特征 $(x_1, x_2, ..., x_m)$，我们想将它分到 $y \in \{1, 2, ..., K\}$ 中的一类，那么我们想预测的就是在已知 x 的条件下，这一数据属于类别 k 的概率，即 $\mathbb{P}[y = k|x]$。根据这个概率我们可以从中找出最有可能的所属类别。

根据贝叶斯定理（3.2.2），我们可以将这个概率写成：

$$\mathbb{P}[y = k|x] = \frac{\mathbb{P}[x|y = k]\mathbb{P}[y = k]}{\mathbb{P}[x]}$$

其中 $\mathbb{P}[x|y = k]$，关于 x 这个值我们可以将它看成是 $x_1, .., x_m$ 这些事件同时发生的一种情况。所以这一项我们可以写成：

$$\mathbb{P}[x|y = k] = \prod_{1 \le i \le m} \mathbb{P}[x_i|y = k]$$

这个等式成立需要一个很重要的前提：**所有特征相互独立**。由此我们可以计算 x 属于类别 k 的概率：

$$\mathbb{P}[y = k|x] = \frac{\prod_{1 \le i \le m} \mathbb{P}[x_i|y = k]\mathbb{P}[y = k]}{\mathbb{P}[x]} \tag{3.2.3}$$

上式 3.2.3 中，为了计算这一概率，等式右边的项应该都由我们的训练模型提供。

- $\mathbb{P}[x_i|y=k]$ 项：分为两种类型

  1. 如果 x_i 的值是**离散**的，那么我们可以根据训练集中 x_i 值在各类中所占的比例给出概率；
  2. 如果 x_i 的值是**连续**的，那么我们对训练集中的数据建立概率分布模型（一般是正态分布模型/混合正态分布）来计算概率。

- $\mathbb{P}[y=k]$ 项：直接统计训练集中各类所占比例给出概率；

- $\mathbb{P}[x]$ 项：在比较概率大小时，由于这一分母是公共项，所以其值大小不是关键。只是为了将左边的概率值**归一化**。

但在这些项的计算时，我们会发现一个问题：对于离散型的特征 x_i，如果训练集中所有的属于类别 k 的数据中没有 x_i 这个值，那么也就是说我们的模型中 $\mathbb{P}[x_i|y=k]=0$。这个 0 的存在导致了很多“意外”，假如 x_i 并不是一个很重要的特征，也会直接导致我们的预测结果 $\mathbb{P}[y=k|x_i]=0$，大大影响模型的准确率。

因此我们会“人为”地给 x_i 加入一项来避免 0 概率事件的产生。这样当我们的训练集足够大时，也不会对我们的结果产生很大影响。这一过程，我们叫做 *Laplace* 校准。

朴素贝叶斯算法被广泛的应用在自然语言处理领域上（比如垃圾邮件的分类）。我们可以定义一些代表性单词在语言中出现的频率来作为文章/邮件的特征，根据这些数据对文章进行分类。

MLlib 函数 MLlib 中的 Classfication/NaiveBayes 类提供了建立在朴素贝叶斯算法基础上的模型预测。但比较局限的是他仅仅能对一些离散的数据作分析，并且要求数据严格非负，因此用处可能比较局限，多用于文档的归类。

```
from pyspark.mllib.classification import NaiveBayes, NaiveBayesModel
from pyspark.mllib.util import MLUtils

# Load and parse the data file.
data = MLUtils.loadLibSVMFile(sc, "D:/spark/data/mllib/sample_libsvm_data.txt")

# Split data approximately into training (60%) and test (40%)
training, test = data.randomSplit([0.6, 0.4])

# Train a naive Bayes model.
model = NaiveBayes.train(training, 1.0)
```

代码 14: 朴素贝叶斯算法使用实例

而输入的参数也相对来说比较简洁，只有训练集（LabeledPoint 类 RDD）和 lambda（即 Laplace 优化中校准中的个数，默认为 1）。

```
class NaiveBayes(object):
    def train(cls, data, lambda_=1.0):
```

3.3 Decision trees - 决策树

决策树（Decision Tree）也是一种比较常见的分类模型，同时它可以被用来解决回归问题，其思想类似于分类算法。决策树所建立的分类模型在在形式上比较直观，理解起来比较方便（但它的建立并不容易）。我们从分类树开始说起。

3.3.1 分类树

例 2 一个女生判断约会男生的标准，她肯定会通过一些男生的特征来将男生分类：值得见面（正类）和不想见面的（负类）。于是这里我们就碰到了一个典型的二分类问题，那么这个女孩是如何来判断的呢？如下图，首先是判断年龄，然后外貌...

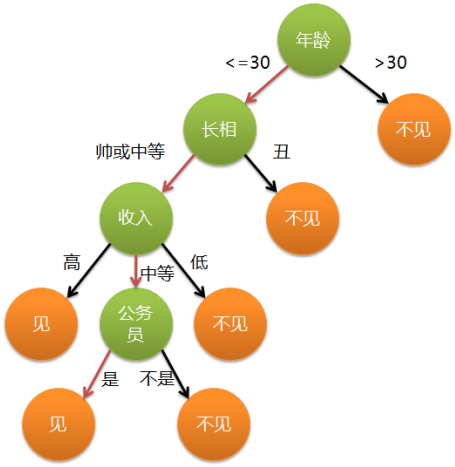


图 8: 决策树 -例

这里我们就看到了一个很典型的决策树，不同于以上几种线性分类器。决策树（*decision tree*）的输出结果是一个**树结构**（可以是二叉树或非二叉树）。其每个非叶节点表示一个特征属性上的测试，每个分支代表这个特征属性在某个值域上的输出，而每个叶节点存放一个类别。使用决策树进行决策的过程就是从根节点开始，测试待分类项中相应的特征属性，并按照其值选择输出分支，直到到达叶子节点，将叶子节点存放的类别作为决策结果。

构造决策树的基本步骤是

1. 开始，所有样本看作一个节点

2. 遍历每个特征的每一种分割方式，找到最好的分割点

3. 分割成两个节点 $N_1^{(1)}$ 和 $N_2^{(1)}$（高级的决策树算法中，也可以同时分成多个结点。）

4. 对 $N_1^{(1)}$ 和 $N_2^{(1)}$ 分别继续重复执行数步，直到每个节点足够“纯”为止

那么到了这一步就会有一个问题，如何找到一个好的分割点，又如何判断一个分类的“纯度”呢？数学上引进了几种信息量来判断分类的好坏。对于一个总数据集 S，其中数据一共有 k 类，现在我们将它分成 p 个部分。

定义 7 对于一个 k 分类数据集 S，我们定义它的**基尼指数（*Gini Index*）**为：

$$Gini(S) = 1 - \sum_{1 \le i \le k} (\frac{n_i}{|S|})^2$$

其中 n_i 表示 S 中属于类别 i 的数据量。如果将 S 分成 p 个部分 $(S_1, ..., S_p)$，那么划分后的基尼指数定义为：

$$Gini(S_1, ..., S_p) = \sum_{1 \le j \le p} \frac{|S_j|}{|S|} Gini(S_j)$$

即加权后的每个子集平均基尼指数。

定义 8 对于一个 k 分类数据集 S，我们定义它的**信息熵（*Entropy*）**为：

$$Q(S) = -\sum_{1 \le i \le k} p_i log_2(p_i)$$

其中 $p_i = \frac{n_i}{|S|}$ 表示 S 中属于类别 i 的样本所占的比例。如果将 S 分成 p 个部分 $(S_1, ..., S_p)$，那么划分后的信息熵定义为：

$$Q(S_1, ..., S_p) = \sum_{1 \le j \le p} \frac{|S_j|}{|S|} Q(S_j)$$

即加权后的每个子集平均信息熵。

一般来说，对于一个划分的判断，我们会用**增益率（ratio gain）**来衡量，可以表示为 $Gini(S) - Gini(S_1, ..., S_p)$ 或者 $Q(S) - Q(S_1, ..., S_p)$。如果增益率越高则说明数据划分的效果越好。而决策树算法本质上属于一种**“贪心算法”**，换句话说，它会在每一个节点思考所有可能的划分情况，选出最优解（增益率最大的划分）。贪心算法每次迭代寻找最优值，但并不一定趋向于全局最优值。

基尼指数和信息熵是两种不同的判断分类信息的标准，其本身的数值很难去解释是否有什么现实意义。完全是为了比较几种划分时才建立了这些量值。根据函数的不同，我们可以建立几种不同类型的决策树：其中基于基尼指数的决策树叫 **CART**（Classification And Regression Tree）；基于信息熵的决策树称为 **ID3**。还有一种常用的 C4.5 决策树，是在 ID3 基础上对信息熵增益率作了改进。

MLlib 函数 MLlib 中的 Classification/DecisionTree 类可以用来解决多分类的问题：

```
from pyspark.mllib.tree import DecisionTree, DecisionTreeModel
from pyspark.mllib.util import MLUtils

# Load and parse the data file into an RDD of LabeledPoint.
data = MLUtils.loadLibSVMFile(sc, 'data/mllib/sample_libsvm_data.txt')
# Split the data into training and test sets (30% held out for testing)
(trainingData, testData) = data.randomSplit([0.7, 0.3])
# Train a DecisionTree model.
# Empty categoricalFeaturesInfo indicates all features are continuous.
model = DecisionTree.trainClassifier(trainingData, numClasses=2, categoricalFeaturesInfo={},
    impurity='gini', maxDepth=5, maxBins=32)
```

代码 15: 决策树分类器使用实例

按照惯例我们找到 trainClassifer 函数的定义，观察其中的各个参数：

```
class DecisionTree(object):
    def trainClassifier(cls, data, numClasses, categoricalFeaturesInfo,
        impurity="gini", maxDepth=5, maxBins=32, minInstancesPerNode=1,
        minInfoGain=0.0):
```

1. data：训练数据，Labeledpoint 类的 RDD；
2. numClasses：类的总量；
3. categoricalFeaturesInfo：特征列的类型。所有特征列分为两种：连续值/类别列。连续值会自动识别，类别列必须在该参数中插入项 (列数，K= 类别总数)，这一列的值应为 0,1,..,K-1；
4. impurity：纯度函数种类，"gini"（默认）/"entropy" 可选；
5. maxDepth：决策树的最大深度，默认 =5；
6. maxBins：数据的最大分裂子集数，即最大叶节点数；默认 =32；
7. minInstancesPerNode：每次分裂子集中包含的最小数据量，默认 =1；
8. minInfoGain：采纳划分的最小信息增益率，默认 =0.0。

```
predictions = model.predict(testData.map(lambda x: x.features))
labelsAndPredictions = testData.map(lambda lp: lp.label).zip(predictions)
testErr = labelsAndPredictions.filter(lambda (v, p): v != p).count() / float(testData.count())
print('Test Error = ' + str(testErr))
```

代码 16: 测试决策树分类器

```
Learned classification tree model with entropy function:
DecisionTreeModel classifier of depth 1 with 3 nodes
    If (feature 406 <= 72.0)
        If (feature 100 <= 165.0)
            Predict: 0.0
        Else (feature 100 > 165.0)
            Predict: 1.0
    Else (feature 406 > 72.0)
        Predict: 1.0

accuracy on Test set with gini indice= 0.941176470588
```

3.3.2 回归树

决策树同样也可以用来解决回归的问题。回归树（*Regression Tree*）的基本思想类似于分类树，显示将数据根据标签的大小分成几类。如下图，利用决策树算法将平面划分成了 7 块：在建立了一个决策树之后，对于一个新的数据 x，我们将其放入到预测的

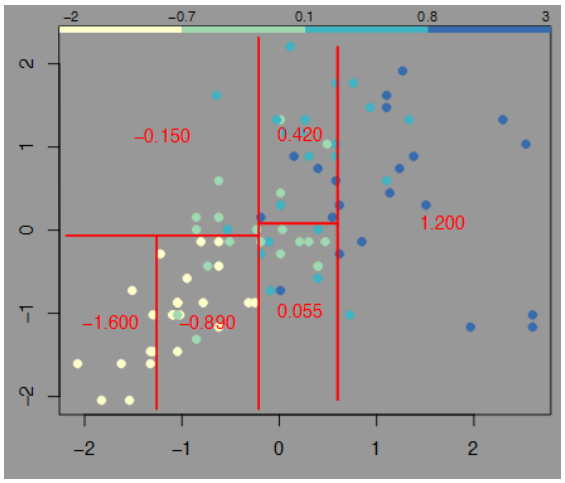


图 9: 利用决策树解决回归问题

类中，而每个类中都有一个对其标签的预测值（等于预测数据中所属的平均值）。

其算法流程类同于决策树分类器，只是我们在判断划分好坏的时候不再使用信息熵/基尼指数，而是用了方差函数来代替。这里的**方差增量**就是：

$$\Delta Var(S_1, ...S_p) = Var(S) - \sum_{1 \le i \le p} \frac{|S_i|}{|S|} Var(S_i)$$

MLlib 函数 MLlib 中的 tree/DecisionTree 类中也提供了trainRegressor，它的具体用法类似 trainClassifier 函数，在这里就不举具体的使用案例了，就稍微来看一下它的函

数定义和参数：

```
class DecisionTree(object):
    def trainRegressor(cls, data, categoricalFeaturesInfo,
        impurity="variance", maxDepth=5, maxBins=32, minInstancesPerNode=1,
        minInfoGain=0.0):
```

1. data：训练数据，Labeledpoint 类的 RDD；
2. categoricalFeaturesInfo：特征列的类型。所有特征列分为两种：连续值/类别列。连续值会自动识别，类别列必须在该参数中插入项 (列数，K= 类别总数)，这一列的值应为 0,1,..,K-1；
3. impurity：纯度函数种类，只能为"variance"；
4. maxDepth：决策树的最大深度，默认 =5；
5. maxBins：数据的最大分裂子集数，即最大叶节点数；默认 =32；
6. minInstancesPerNode：每次分裂子集中包含的最小数据量，默认 =1；
7. minInfoGain：采纳划分的最小信息增益率，默认 =0.0。

3.4 Ensembles of trees (Random Forests and Gradient-Boosted Trees)

- 随机森林和 GBDT 算法

3.4.1 Bagging/Boosting 算法

严格意义上来说，bagging 和 boosting 并不能算是分类算法，它们都是集成算法，是对其他分类算法的一种优化方式。

袋装算法（Bagging）

装袋的概念在于将全局样本分成几个部分，也就是把数据样本放入几个袋中分别进行处理。

用数学的语言来说，我们一共有 N 个数据 $(x_i, y_i)_{i=1,...,N}$，从中**随机**取出 n 个样本，利用机器学习的分类算法得到一个分类器 h_1（这里可以使用 svm，决策树等等）。重复以上操作 p 次，得到 p 个分类器 $h_1, h_2, ..., h_p$，然后利用这 p 个分类器（*弱分类器，weak learner)*得到一个较好的分类器（*强分类器，strong learner*）。其实我们可以把 Bagging 的思想理解为使用弱分类器来进行投票，通过投票的方法来决定最后的结果。这里我们可以选择一个在全局数据上性能最好的分类器，也可以对新数据的类别做一个“投票”，选择“票数”最多的一类。

著名的**随机森林（Random Forest）**算法就是利用 Bagging 思想加以优化的决策树算法，在后面会具体提到。

Boosting 算法

Boosting 算法其实也是对一个简单模型不断提升过程，这个过程通过不断的训练，可以提高模型对数据的分类能力。与 bagging 不同的是，boosting 会通过之前的分类结果对不同样本加权从而得到更好的结果。整个过程如下所示：

1. 先通过对其中 n 个训练样本的学习得到第一个弱分类器 h_1;
2. 将分错的样本和其他的新数据一起构成一个新的 n 个的训练样本，通过对这个样本的学习得到第二个弱分类器 h_2;
3. 将 h_1 和 h_2 都分错了的样本加上其他的新样本构成另一个新的 N 个的训练样本，通过对这个样本的学习得到第三个弱分类器 h_3；
4. 重复数次之后，得到最终经过提升的强分类器。

其中比较著名的 boosting 算法有 Adaboost（Adaptive Boosting）。Adaboost 是一种加和模型，每个模型都是基于上一次模型的错误率来建立的，过分关注分错的样本，而对正确分类的样本减少关注度，逐次迭代之后，可以得到一个相对较好的模型。

Bagging vs. Boosting 二者的主要区别是取样方式不同。bagging 采用**均匀取样**，而 Boosting 根据错误率来取样，因此 boosting 的分类精度要优于 Bagging。bagging 的训练集的选择是随机的，各轮训练集之间相互独立，而 boosting 的各轮训练集的选择与前面各轮的学习结果有关；bagging 的各个预测函数没有权重，而 boosting 是有权重的；bagging 的各个预测函数可以**并行**生成，而 boosting 的各个预测函数只能顺序生成。对于象神经网络这样极为耗时的学习方法。bagging 可通过并行训练节省大量时间开销。

3.4.2 随机森林（Random Forest）

上文已经提到，随机森林其实是一种 Bagging 对决策树算法的优化应用。

什么叫随机森林？当我们利用 bagging 算法得到诸多决策树的时候，我们就得到了一片森林。那么随机的概念体现在什么地方呢？我们重新看决策树的算法

1. **随机**取出 n 个样本（Bagging 思想）；
2. **随机**选出 k 个特征，遍历这些特征的每一种分割方式，找到最好的分割点；

3. 分割成两个节点 $N_1^{(1)}$ 和 $N_2^{(1)}$；

4. 对 $N_1^{(1)}$ 和 $N_2^{(1)}$ 分别继续重复执行 1,2,3 步，直到每个节点足够“纯”为止。

通过以上步骤，我们就建立了一颗所谓的“随机树”。在每次建立决策树的时候，这个随机体现在两个方面。第一，样本随机选择；第二，决策树的节点特征随机选择。当我们有大量的“随机树”时，就构成了一片“随机森林”。最后通过“投票”的方式从森林中找到或者新造一颗合适的“树”。

需要注意的是，通过计算机上的实践表明，随机森林在处理数据时有较高的精确度（而且比传统的 bagging+decision tree 更快），但是目前还没有充分的数学理论可以支持这一结果。

MLlib 函数 由于随机森林 tree/RandomForest 类中的trainClassifier是一种优化了的决策树算法，因此它的许多参数都和决策树相同，在这基础上，他还需要提供更多的参数，因为它的模型更加复杂。

```
from pyspark.mllib.tree import RandomForest, RandomForestModel
from pyspark.mllib.util import MLUtils

# Load and parse the data file into an RDD of LabeledPoint.
data = MLUtils.loadLibSVMFile(sc, 'data/mllib/sample_libsvm_data.txt')
# Split the data into training and test sets (30% held out for testing)
(trainingData, testData) = data.randomSplit([0.7, 0.3])

# Train a RandomForest model.
# Empty categoricalFeaturesInfo indicates all features are continuous.
# Setting featureSubsetStrategy="auto" lets the algorithm choose.
model = RandomForest.trainClassifier(trainingData, numClasses=2, categoricalFeaturesInfo={},
    numTrees=3, featureSubsetStrategy="auto",
    impurity='gini', maxDepth=4, maxBins=32)
```

代码 17: 随机森林分类器例

```
class RandomForest(object):
    def trainClassifier(cls, data, numClasses, categoricalFeaturesInfo, numTrees,
    featureSubsetStrategy="auto", impurity="gini", maxDepth=4, maxBins=32,
    seed=None):
```

1. data：训练数据，Labeledpoint 类的 RDD；

2. numClasses：类的总量；

3. categoricalFeaturesInfo：特征列的类型。所有特征列分为两种：连续值/类别列。连续值会自动识别，类别列必须在该参数中插入项 (列数，K= 类别总数)，这一列的值应为 0,1,..,K-1；

4. numTrees: 训练树的数量；

5. featureSubsetStrategy：随机森林在随机选择特征时的策略，几种备选策略（假设一共有 n 个特征）：

   - ”auto”(默认)，根据树的数量自动选择；
   - ”all”，选择全部特征；
   - ”sqrt”，每次随机选择 $\sqrt{n}$ 个特征；
   - ”log2”，每次随机选择 $\log_2 n$ 个特征；
   - ”onethird”，每次随机选择三分之一的特征。

6. impurity：纯度函数种类，”gini”（默认）/”entropy” 可选；

7. maxDepth：决策树的最大深度，默认 =4；

8. maxBins：数据的最大分裂子集数，即最大叶节点数；默认 =32；

9. seed：提供分袋和特征选择的一个随机种子；默认为 None，将根据系统时间随机生成。

```
TreeEnsembleModel classifier with 5 trees

Tree 0:
    If (feature 580 <= 0.0)
        If (feature 302 <= 0.0)
            If (feature 403 <= 37.0)
                Predict: 1.0
            Else (feature 403 > 37.0)
                Predict: 0.0
        Else (feature 302 > 0.0)
            Predict: 0.0
    Else (feature 580 > 0.0)
        If (feature 381 <= 0.0)
            Predict: 0.0
        Else (feature 381 > 0.0)
        Predict: 1.0
Tree 1:
```

```
    If (feature 517 <= 0.0)
        If (feature 182 <= 0.0)
            If (feature 269 <= 0.0)
                Predict: 1.0
            Else (feature 269 > 0.0)
                Predict: 0.0
        Else (feature 182 > 0.0)
            Predict: 0.0
    Else (feature 517 > 0.0)
        If (feature 540 <= 65.0)
            Predict: 1.0
        Else (feature 540 > 65.0)
            Predict: 0.0
Tree 2:
......
Tree 3:
......
Tree 4:
    If (feature 435 <= 0.0)
        If (feature 459 <= 38.0)
            Predict: 0.0
        Else (feature 459 > 38.0)
            If (feature 183 <= 0.0)
        Predict: 1.0
            Else (feature 183 > 0.0)
                Predict: 0.0
    Else (feature 435 > 0.0)
        Predict: 1.0

Accuracy is = 1.0
```

类似于决策树，随机森林也可以用来解决回归的问题，它的参数定义和上述的几种函数一致：

```
1  def trainRegressor(cls, data, categoricalFeaturesInfo, numTrees, featureSubsetStrategy="auto",
2                     impurity="variance", maxDepth=4, maxBins=32, seed=None):
```

3.4.3 Gradient-Boosted Trees (GBTs)

*Gradient-Boosted Trees (GBTs)*是一种基于 Boosting 想法的的决策树模型，GBTs 的目标在于建立 M 个决策树 $d_m(x)$，在 boosting 算法中这些决策树被称为**弱分类器（weak learner）**，然后对这 M 个决策树做加权平均，得到一个**强分类器（strong**

learner），也就是最终的函数模型：

$$D(x) = \sum_{1 \le m \le M} \gamma_m d_m(x) + \gamma_0$$

其中 γ_0 是一个常数。

而我们如何选择这些弱分类和它们的权值呢？GBTs 算法给我们指明了一条明路，但是首先我们需要确定了一个代价函数 J，类似于之前的线性模型我们的强分类器应该使得代价函数值 $J(D)$ 最小。由于 boosting 算法是一个迭代算法，GBTs 的思想就是我们在每次迭代中使函数 D 进步一点点，最终得到一个使得代价函数较小的结果。关于代价函数的类型我们稍后在看，先来看一下具体的算法：

Algorithm 2 GTBs 算法

1: $F_0(x) = \gamma_0 = argmin_\gamma L(\gamma)$;
2: **for** m = 1:M **do**
3: 计算 pseudo residuals $\forall i, r_m^{(i)} = -[\frac{\partial J(\gamma)}{\partial \gamma}]_{\gamma = D_{m-1}(x^{(i)})} \approx y^{(i)} - F_{m-1}(x^{(i)})$;
4: 对数据集 $(x^{(i)}, r_m^{(i)})$ 训练得到新的决策树分类器 d_m;
5: 计算 d_m 的权值 $\gamma_m = argmin_\gamma \Sigma(J(D_{m-1}(x^{(i)}) + \gamma d_m(x)))$;
6: 新的 $D_m(x) = D_{m-1}(x) + \gamma_m d_m(x)$
7: **end for**

这个算法对分类树和回归树都适用，只是对于两者的代价函数定义稍有区别：

	适用类型	计算公式
log loss	二分类	$J(D) = 2.log(1 + exp(-yD(x)))$
L_2 loss	回归	$J(D) = (y - D(x))^2$
L_1 loss	回归	$J(D) = \|y - D(x)\|$

表 4: 常见代价函数类型

MLlib 函数 利用这些代价函数可以针对分类树和决策树实现 gradient boosting 的算法优化，得到一个比普通决策树更好的模型。在 MLlib 中也提供了多种不同的代价函数以供选择，但是目前 MLlib 中的 GBTs 算法**只能处理二分类的问题**，没法实现多分类问题。

```
1 from pyspark.mllib.tree import GradientBoostedTrees, GradientBoostedTreesModel
2 from pyspark.mllib.util import MLUtils
3
4 # Load and parse the data file.
5 data = MLUtils.loadLibSVMFile(sc, "data/mllib/sample_libsvm_data.txt")
6 # Split the data into training and test sets (30% held out for testing)
```

```
(trainingData, testData) = data.randomSplit([0.7, 0.3])

# Train a GradientBoostedTrees model.
# Notes: (a) Empty categoricalFeaturesInfo indicates all features are continuous.
# (b) Use more iterations in practice.
model = GradientBoostedTrees.trainClassifier(trainingData,
    categoricalFeaturesInfo={}, numIterations=3)
```

代码 18: GBTs 分类器使用实例

其中的参数类型也都类似于随机森林：

```
class GradientBoostedTrees(object):
    def trainClassifier(cls, data, categoricalFeaturesInfo,
        loss="logLoss", numIterations=100, learningRate=0.1, maxDepth=3,
        maxBins=32):
    ...

    def trainRegressor(cls, data, categoricalFeaturesInfo,
    loss="leastSquaresError", numIterations=100, learningRate=0.1, maxDepth=3,
    maxBins=32):
```

1. data：训练数据，Labeledpoint 类的 RDD；
2. categoricalFeaturesInfo：特征列的类型。所有特征列分为两种：连续值/类别列。连续值会自动识别，类别列必须在该参数中插入项 (列数，K= 类别总数)，这一列的值应为 0,1,..,K-1；
3. loss: 代价函数的类型（注意适用
   - "logLoss"(默认)：适用于分类树；
   - "leastSquaresError"（默认），L_2 loss 适用于回归树；
   - "leastAbsoluteError"，L_1 loss 适用于回归树。
4. numIterations：迭代次数，即建立的弱分类器个数，默认 =100；
5. learningRate：学习率，计算每个弱分类器的最优权值时使用 SGD 算法用到，取值在 0 到 1 之间，默认 =0.1；
6. maxDepth：每棵树的最大深度，默认 =3；
7. maxBins：数据的最大分裂子集数，即最大叶节点数，默认 =32。

最后的模型中我们可以看到每棵决策树的结构和权值：

```
# Evaluate model on test instances and compute test error
predictions = model.predict(testData.map(lambda x: x.features))
labelsAndPredictions = testData.map(lambda lp: lp.label).zip(predictions)
testErr = labelsAndPredictions.filter(lambda (v, p): v != p).count() / float(testData.count())
print('Test Error = ' + str(testErr))
print('Learned classification GBT model:')
print(model.toDebugString())
```

由于是对数据做 ±1 二分类划分，因此我们在结果中看到的 Tree2 的权值应为 0.43819

```
Test Error = 0.0769230769231
Learned classification GBT model:
TreeEnsembleModel classifier with 3 trees
    ...
    Tree 2:
    If (feature 434 <= 0.0)
        Predict: -0.4381935810427206
    Else (feature 434 > 0.0)
        Predict: 0.4381935810427206
```

3.5 Isotonic regression - 保序回归

二次保序回归（*isotonic regression*）是一种非线性的回归方式，它的主要思想是建立一个递增的数列来拟合原数据。但它只能对一维的特征数列进行回归分析，因此应用会比较有限。对于一个数列 (x_i, y_i)，根据 x_i 的值从小到大重新排序，得到的新的数列，(x'_i, y'_i)，同时我们给每一个数据一个权重 w_i（一般没有权重，可以使得所有 $w_i = 1$），我们希望找到一个**递增数列**f_i 可以使得代价函数最小：

$$J = \sum_i w_i (y'_i - f_i)^2$$

那么 $f_i = f(x'_i)$。实际计算中，可以使用平均值来替代 y'_i 中的一些降序子数列。

例 3 真实数列 y: $(14, 9, 10, 15)$ ⇒ 预测数列 f: $(11, 11, 11, 15)$

分析：从第一项 *14* 往后看，发现到 *9* 时发生乱序（*14>9*），停止该轮观察转入处理环节，子序列 *(9,14)* 的平均值为 *11.5*，大于下个元素 *10*，所以同时处理 *10*：对子序列 *(14, 9, 10)* 求得平均值 *11*，由于 *11* 小于下个元素 *15*，所以停止操作，用数列 $(11, 11, 11)$ 替代 $(14, 9, 10)$。

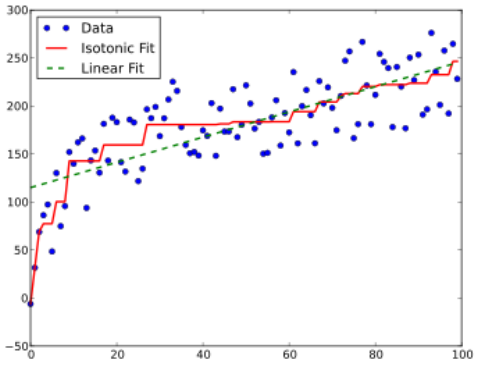


图 10: 保序回归 vs. 线性回归

保序回归对于 Spearman 相关系数高的数据拟合特性较好，但它只能处理**一维特征**的数据，其应用比较有限，多用于医学上的一些模型预测。

MLlib 函数 在 Spark 中可以利用 MLlib 实现保序回归的平行算法:

```
import math
from pyspark.mllib.regression import LabeledPoint, IsotonicRegression, IsotonicRegressionModel
from pyspark.mllib.util import MLUtils

# Load and parse the data
def parsePoint(labeledData):
    return (labeledData.label, labeledData.features[0], 1.0)

data = MLUtils.loadLibSVMFile(sc, "data/mllib/sample_isotonic_regression_libsvm_data.txt")

# Create label, feature, weight tuples from input data with weight set to default value 1.0.
parsedData = data.map(parsePoint)

# Split data into training (60%) and test (40%) sets.
training, test = parsedData.randomSplit([0.6, 0.4], 11)

# Create isotonic regression model from training data.
# Isotonic parameter defaults to true so it is only shown for demonstration
model = IsotonicRegression.train(training)
```

代码 19: 保序回归的 Mllib 使用实例

最为一种比较简单的回归算法，这个 train 函数中只有两个参数：

```
class IsotonicRegression(object):
    def train(cls, data, isotonic=True):
```

1. data：(标签, 特征, 权重) 构成的 RDD；

2. isotonic: 逻辑参数，默认为 True，表示保序回归；若设为 False，表示 y 和 x 负相关，采用**逆序**回归。

利用生成的模型可以快速的对新的数据做预测：

```
predictionAndLabel = test.map(lambda p: (model.predict(p[1]), p[0]))

# Calculate mean squared error between predicted and real labels.
meanSquaredError = predictionAndLabel.map(lambda pl: math.pow((pl[0] - pl[1]), 2)).mean()
print("Mean Squared Error = " + str(meanSquaredError))
```

代码 20: 测试保序回归模型

```
Mean Squared Error = 0.00889887044692
```

4 Collaborative filtering - 协同过滤算法

协同过滤推荐（*Collaborative Filtering recommendation*）是在信息过滤和信息系统中正迅速成为一项很受欢迎的技术。与传统的基于内容过滤直接分析内容进行推荐不同，协同过滤分析用户兴趣，在用户群中找到指定用户的相似（兴趣）用户，综合这些相似用户对某一信息的评价，形成系统对该指定用户对此信息的喜好程度预测。

与传统文本过滤相比，协同过滤有下列优点:

1. 能够过滤难以自动内容分析的信息。如艺术品、音乐;

2. 能够基于一些复杂的，难以表达的概念（信息质量、品位) 进行过滤;

3. 推荐的新颖性。

正因为如此，协同过滤在商业应用上也取得了不错的成绩。Amazon，CDNow，MovieFinder，都采用了协同过滤的技术来提高服务质量。缺点是:

1. 用户对商品的评价非常稀疏，这样基于用户的评价所得到的用户间的相似性可能不准确（即稀疏性问题）;

2. 随着用户和商品的增多，系统的性能会越来越低;

3. 如果从来没有用户对某一商品加以评价，则这个商品就不可能被推荐（即最初评价问题）。

因此，现在的电子商务推荐系统都采用了多种技术相结合的推荐技术。[2]

现在很多推荐系统的建立都是通过**矩阵分解**为基础进行分析。假设现在我们有 m 个用户和 n 件产品，已知数据用一个矩阵 $R \in \mathbb{R}^{m \times n}$ 来表示所有用户对各个产品的评分，比如 $R_{i,j}$ 表示了用户 i 对产品 j 的评分。由于在现实应用中用户量和产品数量都会非常的庞大，因此我们的想法是建立 K（$K << min(m,n)$）个产品的**隐含特征**，根据这些特征将矩阵 R 进行分解。所谓隐含特征，就是指商品本身所具有的，但我们并不知道具体是什么的一些特征，有可能是用途，颜色，也有可能是一些语言难以描述的特征。

我们用矩阵 $X \in \mathbb{R}^{m \times K}$ 表示用户对这 K 个隐含特征偏好矩阵，而矩阵 $Y \in \mathbb{R}^{n \times K}$ 表示成品所包含的隐含特征的矩阵。那么我们对整个推荐系统应该有矩阵分解 $R = X.Y^T$。[12]

当然如果我们的矩阵 R 信息足够完整，当我们将 K 取一个很大的数值时，我们可以得到 $\forall 1 \leq i \leq m, 1 \leq j \leq n, r_{i,j} = X_i^T.Y_j$，其中 X_i 表示矩阵 X 的第 i 行，Y_j 表示矩阵 Y 的第 i 行。但是我们的初衷在于找一个较小的 K 值简化计算，因此肯定在预测上会有一定的误差，在这里我们引入代价函数 J 并加上 L_2 正则项：

$$J(X,Y) = \sum_{i,j}[(R_{i,j} - X_i Y_j^T)^2 + \lambda(\|X_i\|_2^2 + \|Y_j\|_2^2)] \tag{4.0.1}$$

所以我们需要寻找两个矩阵 X,Y 使得代价函数 J 最小化。

但是由于矩阵 R 中可能有很多的缺失信息，因此传统的*奇异值分解法（Singular Value Decomposition，SVD)*在这里并不适用。所以需要使用其他的算法来实现这一分解，比如 Stochastic gradient descent 和 Alternating Least Squares 算法。

4.1 Alternating Least Squares(ALS) - 交替最小二乘法 [15]

由于矩阵 R 中有较多元素缺失，为了修正这些缺失数据对我们的我们需要对公式 4.0.1 定义的代价函数中的正则项做一定修改，我们需要使用一种叫做 Tikhonov 正则法 [8]。定义集合 U_i 表示用户 i 评价的所有产品列表，$u_i = card(U_i)$ 表示这些产品的数量，即矩阵 R 第 i 行的非缺失信息个数；定义集合 P_j 表示所有评价过产品 j 所有用户列表，$p_j = card(P_j)$ 表示这些用户的个数，即矩阵 R 第 j 列的非缺失信息个数。

$$J(X,Y) = \sum_{i,j}(R_{i,j} - X_i Y_j^T)^2 + \lambda(\sum_i u_i \|X_i\|_2^2 + \sum_j p_j \|Y_j\|_2^2) \tag{4.1.1}$$

由于在这个函数中有两个变量矩阵存在，当我们处理求这个最小值的优化问题中，我们采用*交替最小二乘法（ALS）*方法。所谓的的 ALS 算法，就是我们交替变化 X,Y 的值：先固定 Y，求 X 的最优值；然后固定 X 的值，求 Y 的最优值。循环进行这两步，直到误差较小。

1. 固定 Y，求 X 的最优值，使得函数 J 对于 X 中所有元素的偏导都为 0：$\forall 1 \leq i \leq m, 1 \leq k \leq K$ $\frac{\partial J}{\partial X_{m,k}} = 0$ 求得

$$X_i^T = (Y_{U_i}^T . Y_{U_i} + \lambda u_i I_k)^{-1} . Y_{U_i}^T . R(i, U_i)^T \tag{4.1.2}$$

其中 I_k 表示 k 阶单位矩阵，即对角线上都为 1，其余值为 0 的 k 阶矩阵。

2. 固定 X，求 Y 的最优值，使得函数 J 对于 Y 中所有元素的偏导都为 0：同理求得

$$Y_j^T = (X_{P_j}^T . X_{P_j} + \lambda p_j I_k)^{-1} . X_{P_j}^T . R(P_j, j) \tag{4.1.3}$$

具体算法如下:

Algorithm 3 ALS 算法

输入: R，K，终止条件（最大循环次数，$\|XY^T\|$ 变化的最小量）;

输出: X,Y 矩阵;

1: 随机生成一个 $n \times K$ 阶矩阵 Y;
2: **repeat**
3: 固定 Y，利用公式 4.1.2 升级计算 X;
4: 固定 X，利用公式 4.1.3 升级计算 Y;
5: **until** 终止条件成立。

MLlib 函数 ALS 中的迭代算法并不能实现平行化，但是我们可以将矩阵 R 分为几个子矩阵分别计算实现协同过滤的平行化计算。MLlib 中的 recommendation/ALS 类包含了实现了推荐系统的一系列函数：

```
from pyspark.mllib.recommendation import ALS, MatrixFactorizationModel, Rating

# Load and parse the data
data = sc.textFile("data/mllib/als/test.data")
ratings = data.map(lambda l: l.split(','))\
.map(lambda l: Rating(int(l[0]), int(l[1]), float(l[2])))

# Build the recommendation model using Alternating Least Squares
rank = 10
numIterations = 10
model = ALS.train(ratings, rank, numIterations)
```

代码 21: 协同过滤系统使用实例

我们来看一下函数 train 中的各个参数定义：

```
class ALS(object):
    def train(cls, ratings, rank, iterations=5, lambda_=0.01, blocks=-1, nonnegative=False,
    seed=None):
```

1. ratings：RDD 形式的评分矩阵，每行包含三个数字 (用户 ID, 产品 ID, 评分)；
2. rank: 产品的隐形特征数量，即 K 值；
3. iterations：ALS 算法最大迭代次数，默认为 5；
4. lambda_: 正则系数，默认为 0.01；
5. blocks: 将矩阵 R 分成几块实现平行运算，默认 =-1 表示自动选择。
6. 逻辑型参数，默认值为 False；若值为 True 则矩阵分解时需要保证 X，Y 中的值非负；
7. seed：生成初始矩阵的一个随机种子；默认为 None，将根据系统时间随机生成。

误差用平均方差 MSE 来表示：$MSE(X,Y) = \frac{1}{mn}\sum_{i,j}(R_{i,j} - X_i Y_j^T)^2$

```
Mean Squared Error = 7.80050150723e-06
```

5 Clustering - 聚类算法

聚类算法（*clustering*）是一种对没有标签的数据，利用数据集表现出来的多种特征，基于它们之间的相似程度进行自动分类的一种算法，又叫做**非监督学习（unsupervised learning）**。

所以我们的数据集一般写成 $(x^{(i)} \in \mathbb{R}^m)_{1\leq i\leq n}$，表示我们的数据集有 n 个数据样本，每个数据包含 m 个特征。

聚类问题在实际应用中存在较多的难题，在技术理论上并不如分类算法发展的那么完善。其一，对于簇类个数 K 的选择就没有特别好的方法；其二，是否所有的特征都值得作为划分依据？是不是存在全局特征和局部特征的区别，由于非监督学习中没有验证集和测试集的存在，很难判断一个模型的好坏；最后，聚类算法的结果是不定向的，也就是说，我们没办法确定分出来的每一类代表了什么结果，只能通过进一步的分析，确定每个类别中几个比较明显的特征。

5.1 k-Means - k 均值算法

k 均值算法（*k-Means*）是一种很典型的基于距离的聚类算法，采用距离作为相似性的评价指标，即认为两个对象的距离越近，其相似度就越大。该算法认为簇是由距离靠近的对象组成的，因此把得到紧凑且独立的簇作为最终目标。从空间上看，就是将靠的比较近的数据点分为一个簇类。

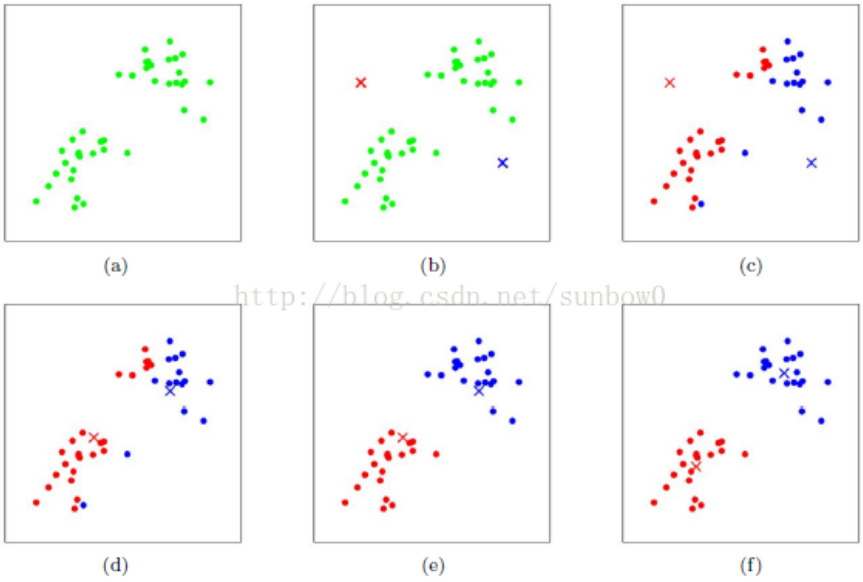


图 11: K-means 聚类算法

K-means 算法以*欧式距离*（*euclidean distance*）作为相似度测度，是求对应某一初始聚类中心向量 V_k 最优分类，使得代价函数 J 最小。算法采用**误差平方和（Mean Squared Error）**准则函数作为聚类准则函数。

Algorithm 4 K-Means 算法

输入： 数据 X，K，终止条件；

输出： K 个簇类的质心；

1: 随机生成 K 个簇类质心 $v_1, .., v_K$;

2: **repeat**

3: 对于每一个数据点 $x^{(i)}$, 寻找离他最近的簇类质心重新归类，$f(x^{(i)}) = argmin_k \ \|x^{(i)} - v_k\|_2^2$;

4: 更新每个质心的坐标：$\forall 1 \leq k \leq K,\ v_k = \mathbb{E}[\{x^{(i)} | f(x^{(i)}) = k\}]$

5: **until** 终止条件成立。

一般是当 K 个质心不再变动（或者变动很小）时终止算法，当然我们也可以限制最大迭代次数。最后得到的 K 个质心决定了数据集中每个点以及新的数据点的分类（寻找最近质心）：$f(x^{(i)}) = argmin_k \ \|x^{(i)} - v_k\|_2^2$。

需要注意的是，在 k-means 算法中对初始质心的选择十分敏感，容易趋向于局部最小值；有些初始点并不能保证我们得到一个很好的分类效果：因此在 k-Means 中我

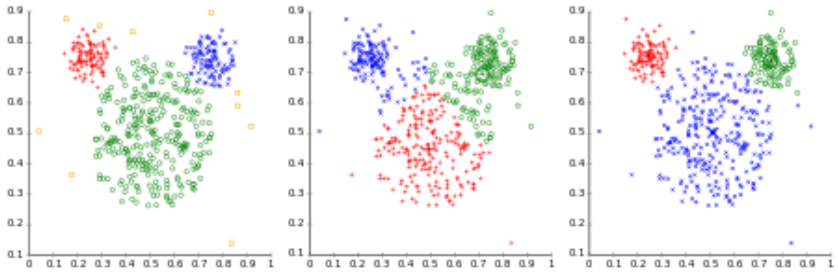


图 12: 不同初始值产生的结果

们偏向于运行多次该算法，最后选择一个使得代价函数最小的模型。

$$J(f) = \sum_{i=1}^{n} \|x^{(i)} - v_{f(x^{(i)})}\|_2^2$$

同时我们有一种改进的选择初始点的方式，叫做 k-Means++ 法，其主要思想是初始的聚类中心之间的相互距离要尽可能的远 [5]：

Algorithm 5 k-Means++ 算法求初始质心

1: 从输入的数据点集合中随机选择一个点作为第一个聚类中心;
2: **repeat**
3: **for** 对于数据集中的每一个点 x **do**
4: 计算它与最近聚类中心 (指已选择的聚类中心) 的距离 $D(x)$;
5: 选择一个新的数据点作为新的聚类中心，选择的原则是：被选取作为聚类中心的概率与 $D(x)^2$ 线性相关;
6: **end for**
7: **until** 选出 k 个聚类中心;
8: 利用这 K 个初始的聚类中心来运行标准的 k-means 算法。

MLlib 函数 MLlib 中的 clustering/KMeans 类可以帮助我们实现处理数据的聚类算法，由于我们需要多次运行该算法以得到最优解，因此该过程可以高度平行。

```
1 from numpy import array
2 from math import sqrt
3 from pyspark.mllib.clustering import KMeans, KMeansModel
4
5 # Load and parse the data
6 data = sc.textFile("data/mllib/kmeans_data.txt")
7 parsedData = data.map(lambda line: array([float(x) for x in line.split(' ')]))
8
```

```
# Build the model (cluster the data)
clusters = KMeans.train(parsedData, 2, maxIterations=10,
    runs=10, initializationMode="random")
```

代码 22: k-Means 算法使用实例

```
class KMeans(object):
    def train(cls, rdd, k, maxIterations=100, runs=1, initializationMode="k-means||",
        seed=None, initializationSteps=5, epsilon=1e-4, initialModel=None):
```

1. rdd：数据集，类型为 vector 的 RDD；
2. k：簇类数目，即 K 值；
3. maxIterations：最大迭代次数，默认 =100；
4. runs[1]：重复运行 k-Means 算法的次数，默认 =1；
5. initializationMode：设置初始值方法，两个值可选
   - "k-means||"(默认)：使用 k-Means++ 方法确定（略有修改）初始质心；
   - "random"：随机生成初始质心。
6. seed：生成初始值的一个随机种子；默认为 None，将根据系统时间随机生成。
7. initializationStep: 使用 k-means++ 方法时的最大步数，默认 =5，即 5 次迭代之内确认最终初始值，一般不需要改变该参数。
8. epsilon: 判断质心移动距离的阀值，默认 =0.0001；
9. initialModel：可以提供自己希望的初始质心值，默认为"None"。

k-Means 模型的优劣一般通过计算它在训练集上的代价函数值来衡量：

$$J(f) = \sum_{i=1}^{n} \|x^{(i)} - v_{f(x^{(i)})}\|_2^2$$

```
# Evaluate clustering by computing Within Set Sum of Squared Errors
def error(point):
    center = clusters.centers[clusters.predict(point)]
    return sqrt(sum([x**2 for x in (point - center)]))

WSSSE = parsedData.map(lambda point: error(point)).reduce(lambda x, y: x + y)
print("Within Set Sum of Squared Error = " + str(WSSSE))
```

代码 23: 计算 MSE 测试 k-Means 模型

[1]从 Spark2.0.0 起该参数不再有效

5.2 Gaussian Mixture - 混合高斯模型

类似于 k-Means 的基础想法，混合高斯模型（*Gaussian Mixture Model*）建立了 K 个不同形状的**高斯分布**（正态分布）函数来拟合数据的分布。我们有 n 个 m 维的数据等待分组，寻找 K 个多维高斯模型 $\mathcal{N}(\mu_1, \Sigma_1), ..., \mathcal{N}(\mu_K, \Sigma_K)$ 以及每个模型的权重 $p_1, ...p_K$。其中 Σ 表示高斯模型的**协方差矩阵**，$\Sigma = \mathbb{E}[(X-\mu)(X-\mu)^T]$，每个元素定义为 $\Sigma_{ij} = Cov(X_i, X_j)$。所以 Σ 应该是一个 m 阶的相似半正定矩阵[2]。

那么对于整个模型的概率密度函数我们可以写成：

$$f_X(x) = \sum_{1 \le k \le K} p_k f_k(x) \tag{5.2.1}$$

其中 $f_k(x)$ 表示第 k 个高斯分布的密度函数，它的具体形式为：

$$f_k(x) = \frac{1}{(2\pi)^{m/2}|\Sigma_k|^{1/2}} exp[-\frac{1}{2}(x-\mu_k)^T\Sigma_k^{-1}(x-\mu_k)]$$

$|\Sigma_k|$ 表示矩阵 Σ_k 的行列式[3]。

如果我们成功找到了这 k 个分布来拟合我们的数据集，那么对于一个数据点 x，它属于簇类 k, C_k 的概率应为:

$$\mathbb{P}[x \in C_k | x] = \frac{p_k f_k(x)}{\sum_{j=1}^{K} p_j f_j(x)}$$

这样我们就可以从 K 个簇类中找到一个最大概率的簇类，将 x 划分进去。

那么接下来我们就需要对模型中的所有参数做一个估计，包括 $\forall 1 \le k \le K, p_k, \mu_k, \Sigma_k$。我们常用最大期望算法（*Expectation Maximization Algorithm*，又译期望最大化算法）来寻找使得 **log-似然函数（log-likelihood function）**最大化的参数。

定义 9 混合高斯模型的 ***log-likelihood*** **函数**定义为：

$$l(\theta, x) = \log(\sum_k p_k f_k(x)) \tag{5.2.2}$$

（θ 代表了所有的参数）。那么对于整个数据集，我们的 *log-likelihood* 函数应该等于

$$l(\theta, X) = \sum_{1 \le i \le n} [log(\sum_k p_k f_k(x^{(i)}))]$$

这个算法的**终止条件**一般指当 log-likelihood 的值变化很小时停止迭代，当然我们也可以同时限制它的最大迭代次数。

[2]半正定矩阵（positive-semidefinite matrix）：对于所有非零 m 阶向量 z，$z^T\Sigma z \ge 0$。

[3]行列式（determinant）：若一个矩阵 A 可逆，则它的行列式非零，记为 $det(A)$ 或 $|A|$。

Algorithm 6 利用 EM 算法估算混合高斯模型

输入： K，数据集 X（n 个 m 维数据），终止条件；

输出： K 个高斯分布模型；

1: 随机选择 K 个数据作为初始 μ_k，初始 p_k 设为 1/K；

2: **repeat**

3: E(xpectation)-step：计算每个数据属于各侧类的概率：

$$w_{i,k} = \mathbb{P}[x^{(i)} \in C_k | x^{(i)}] = \frac{p_k f_k(x^{(i)})}{\sum_{j=1}^{K} p_j f_j(x^{(i)})}$$

记 $N_k = \sum_i w_{i,k}$，而 $\sum_k w_{i,k} = 1$

4: M(aximization)-step: 更新各个参数的值，使得目标函数最大化：

$$\begin{aligned} p_k^{new} &= N_k/n \\ \mu_k^{new} &= \frac{1}{N_k} \sum_{1 \le i \le n} w_{i,k} x^{(i)} \\ \Sigma_k^{new} &= \frac{1}{N_k} \sum_{1 \le i \le n} w_{i,k} (x^{(i)} - \mu_k^{new})(x^{(i)} - \mu_k^{new})^T \end{aligned}$$

5: **until** 终止条件成立

MLlib 函数 在 MLlib 中的 clustering/GaussianMixture 类可以利用 EM 算法来计算各个高斯模型的参数值。

```
from numpy import array
from pyspark.mllib.clustering import GaussianMixture, GaussianMixtureModel

# Load and parse the data
data = sc.textFile("D:/spark/data/mllib/gmm_data.txt")
parsedData = data.map(lambda line: array([float(x) for x in line.strip().split(' ')]))

# Build the model (cluster the data)
gmm = GaussianMixture.train(parsedData, k=3)

for i in range(2):
print("weight = ", gmm.weights[i], "mu = ", gmm.gaussians[i].mu,
    "sigma = ", gmm.gaussians[i].sigma.toArray())
```

```
class GaussianMixture(object):
    def train(cls, rdd, k, convergenceTol=1e-3, maxIterations=100, seed=None, initialModel=None):
```

k-Means 函数中的各个参数的含义：

1. rdd: 数据集，类型为 vector 的 RDD；

2. k：簇类数目，即 K 值；

3. convergenceTol: 收敛判断，当 log-likelihood 值变化小于该值时终止算法，默认 =0.001；

4. maxIterations: 最大迭代次数，默认 =100；

5. seed: 生成初始值的一个随机种子；默认为 None，将根据系统时间随机生成。

6. initialModel: 可以提供初始值，默认为"None"。

生成的模型中包含了所有参数的值：权重，期望和方差矩阵。

```
('weight = ', 0.50939120671271387, 'mu = ', DenseVector([-0.1279, 0.0551]), 'sigma = ', array([
    [ 4.95833501, -2.03232806],
    [-2.03232806, 1.01693751]]))
('weight = ', 0.47073457524492263, 'mu = ', DenseVector([0.0244, 0.0051]), 'sigma = ', array([
    [ 4.78135611, 1.8864649 ],
    [ 1.8864649 , 0.91985744]]))
('weight = ', 0.019874218042363435, 'mu = ', DenseVector([1.7171, -0.0095]), 'sigma = ', array([
    [ 0.52186603, 0.43030463],
    [ 0.43030463, 0.67602191]]))
```

5.3 Power Iteration Clustering (PIC)

PIC(Power Iteration Clustering)算法是一种针对图模型的谱聚类（*spectral clustering*）算法。它的思想是建立对于所有数据之间相似度的关联矩阵（*affinity matrix*），利用这个矩阵的**特征向量**来实现对数据点的分类。由于谱聚类中会用到大量的**线性代数（linear algebra）**知识，理解起来可能需要一定该方面的基础。

对于我们的数据集 $(x^{(i)})_{1\leq i\leq n}$ 首先定义一种**相似度（similarity）**。相似度函数表示了两个数据点各特征之间的相似程度，$s(x^{(i)}, x^{(j)}) \geq 0$，值越大，说明这两个数据越相似。当然我们有时候不能直观地判断两个向量之间的相似程度，于是又引进了另外一个概念 -**距离（distance）**，顾名思义，当两个向量之间的距离越长，两者的相似程度越低。

定义 10 $d : I \times I \leftarrow \mathbb{R}_+$，当 d 满足以下条件时，我们称 d 是一个在空间 I 上的**距离（*distance/metric*）**：

1. $\forall x \in I,\ d(x, x) = 0$

2. $\forall x, y \in I,\ d(x, y) \geq 0$

3. $\forall x, y \in I,\ d(x,y) = d(y,x)$

4. $\forall x, y, z \in I,\ d(x,z) \leq d(x,y) + d(y,z)$

最常见的的距离就是我们的常用的 $\mathbb{R}^n$ 空间上的欧氏距离：$d(x,y) = \|x-y\|_2^2$。在处理数据是，根据不同的数据类型我们可以定义不同的距离函数。我们可以通过计算距离的倒数来实现距离和相似度之间的转换。

我们用一个**关联矩阵（affinity matrix）**A 来记录所有数据两两之间的相似度：$A_{ij} = s(x^{(i)}, x^{(j)}) \in \mathbb{R}^{n\times n}$。我们同时建立一个对角矩阵，叫做 Degree Matrix D，$D_{ii} = \sum_j A_{ij}$ 来记录矩阵 A 的每一行的和。利用矩阵 D 我们可以将关联矩阵**归一化（normalization）**，记标准化后的关联矩阵为 $W = D^{-1}A$，它的最大特征值（eigenvalue）为 1。

定义 11 *对于矩阵 A，它的***特征向量（*eigenvector*）***v *表示那些通过矩阵 A 的变换后方向不变的向量，即*

$$\exists \lambda \in \mathbb{R}, Av = \lambda v$$

*其中 λ 叫做 v 对应的***特征值（*eigenvalue*）***。对于一个 m 维的可逆矩阵，它应该有 m 个不同的特征向量。*

谱聚类（Spectral Clustering） *谱聚类 (Spectral Clustering)*是一种针对**图模型**的聚类方法——将带权无向图划分为两个或两个以上的最优子图，使子图内部尽量相似，而子图间距离尽量距离较远，以达到常见的聚类的目的。其基本思想是利用样本数据的相似矩阵 (拉普拉斯矩阵，Laplacian Matrix) 进行特征分解后得到的特征向量进行聚类。

拉普拉斯矩阵 L 定义为 $L = D - A$，谱聚类的一般步骤：

1. 数据准备，生成数据的**关联矩阵** A；
2. 归一化拉普拉斯矩阵 $L = I_n - D^{-1/2}.A.D^{1/2}$；
3. 生成 L 中最小的 k 个特征值和对应的**特征向量**；
4. 将特征向量利用 **kmeans 聚类**；

而我们的 PI（Power Iteration）方法就是一种估算最大特征向量（即对应特征值最大）的算法，在 PIC 算法中我们一般只计算一个特征向量，然后进行 k-Means 聚类分析得到结果。

M. Meila 和 J. Shi 在 [13] 中定义了一种新型的拉普拉斯矩阵，**Normalized Random Walk Laplacian Matrix**，$L' = I_n - D^{-1}.A = I_n - W$。其中 L' 的特征向

量和 W 的一致，而且 W 的第二大特征向量 (最大的特征值为 1）等于 L' 的第二小特征向量。

利用对矩阵 W 使用 PI 算法计算它的特征向量得到我们的 PIC 算法 [15]：

Algorithm 7 PIC 算法

输入： 矩阵 W，终止条件；

输出： 聚类模型

1: 随机选择 $v^{(0)} \in \mathbb{R}^n$;

2: **repeat**

3: 计算 $v^{(t+1)} = \frac{W.v^{(t)}}{\|W.v^{(t)}\|_1}$，$\delta^{t+1} = |v^{(t+1)} - v^{(t)}|$，增加 t 值；

4: **until** 终止条件成立。

5: 对向量 $v^{(t)}$ 使用 k-Means 聚类。

对于这初始点的生成，我们可以对随机生成的方式加以改进 $v_i^{(0)} = \frac{\sum_j A_{i,j}}{\sum_{i,j} A_{i,j}}$。终止条件一般是对 δ^t 值变化的判断，当 $\delta^t - \delta^{t-1} \approx 0$ 时，我们终止算法，同时也应该加入最大迭代次数。

MLlib 函数 在 MLlib 提供了 PIC 算法实现对图模型的聚类分析。但是需要先对数据做一个转化，根据两两之间的相似度（提前定义）建立一个图模型数据。

```
from pyspark.mllib.clustering import PowerIterationClustering, PowerIterationClusteringModel

# Load and parse the data
data = sc.textFile("D:/spark/data/mllib/pic_data.txt")
similarities = data.map(lambda line: tuple([float(x) for x in line.split(' ')]))

# Cluster the data into two classes using PowerIterationClustering
model = PowerIterationClustering.train(rdd=similarities, k=2, maxIterations=10)

def printResult(x):
    print(str(x.id) + " -> " + str(x.cluster))

model.assignments().foreach(printResult)
```

代码 24: PIC 聚类算法使用实例

```
class PowerIterationClustering(object):

    def train(cls, rdd, k, maxIterations=100, initMode="random"):
```

PIC 算法中，我们所需要用到的参数较少：

1. rdd：图模型的 RDD，每行三个元素 [点 1 编号，点 2 编号，相似度]。可以不包括相同点的信息，相似度为 0；
2. k: 簇类个数，即 K 值；
3. maxIterations: 最大迭代次数，默认 =100；
4. initMode: 初始向量的生成方式，两种模式可选
   - "Random"(默认): 随机生成；
   - "degree": 改进后的生成方法。

在 MLlib 中使用 PIC 算法时，需要自己定义一个相似度（距离）函数，然后利用这个函数将我们的数据集转化为一个**图模型**。PIC 算法可以对图中的已有数据进行聚类，但对新数据的预测判断比较难以实现，需要重新计算新的数据与原数据之间的相似度在加以判断，存储的模型就会比较大。

```
15 -> 1
11 -> 1
1 -> 0
3 -> 0
7 -> 1
...
```

5.4 Latent Dirichlet allocation (LDA) - 隐含狄利克雷分配

隐含狄利克雷分配（*Latent Dirichlet Allocation*）是一种**主题模型** (Topic Model，即从所收集的文档中推测主题)。甚至可以说 LDA 模型现在已经成为了主题建模中的一个标准，是实践中最成功的主题模型之一。

那么何谓“主题”呢?，就是诸如一篇文章、一段话、一个句子所表达的中心思想。不过从统计模型的角度来说，我们是用一个特定的词频分布来刻画主题的，并认为一篇文章、一段话、一个句子是从一个概率模型中生成的。也就是说在主题模型中，主题表现为一系列相关的单词，是这些单词的条件概率。形象来说，主题就是一个桶，里面装了出现概率较高的单词，这些单词与这个主题有很强的相关性。

LDA 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；比如我们可以将

2-3 个单词的构成词组作为文档的特征来提高模型的准确度。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 可以被认为是如下的一个聚类过程：

1. 各个**主题（Topics）**对应于各类的“**质心**”，每一篇文档被视为数据集中的一个样本；
2. 主题和文档都被认为存在一个向量空间中，这个向量空间中的每个特征向量都是词袋模型；
3. 与采用传统聚类方法中采用距离公式来衡量不同的是，LDA 使用一个基于统计模型的方程，而这个统计模型揭示出这些文档都是怎么产生的。

从数学的角度来看，我们将建立一个关于**（单词，主题，文档）的三层贝叶斯模型**。记 $D = (d_i)_{1\leq i\leq M}$ 表示我们的文档集合（Documents）；其中每篇文档由一串单词构成，比如对于文档 d_i 由单词集 $(d_{i1}, ..., d_{im_i})$ 构成，而这些单词应该属于一个字典（Vocabulary），V，里面包含了 N 个不同的单词，$V = \{w_1, ..., w_N\}$。在这些基础上，我们通过训练会给出 K 个主题（topic），记为 $z_1, ..., z_K$，根据这些主题将文档分为 K 类。在这里，我们需要**假设所有单词之间都是独立的**。

根据概率论的**联合分布（joint distribution）**公式，我们得到一个文档分类中基础的理论算式：

$$\mathbb{P}[w|d] = \sum_{k=1}^{K} \mathbb{P}[w|z_k]\mathbb{P}[z_k|d] \tag{5.4.1}$$

其中每一项条件概率所表达的含义：

- $\mathbb{P}[w|d]$ 表示了单词 w 出现在文档 d 中的概率，可以用 w 的词频来表示。（已知）
- $\mathbb{P}[w|z_k]$ 主题 k 中出现单词 w 的概率，我们用概率函数 ϕ_k 来表示这一项，即 $\phi_{kw} = \mathbb{P}[w|z_k]$，这一项对于之后的新的文档的分类（即模型保存）非常重要；
- $\mathbb{P}[z_k|d]$ 文档 d 属于主题 k 的概率，用概率函数 θ_i 表示文档 d_i 属于各个主题的一个概率。这一概率是我们最后需要估计聚类标准。

我们可以将（5.4.1）写成一个矩阵的形式，利用类似于协同过滤的矩阵分解方法求得我们需要得到的概率。但是 LDA 提供了一种更加精确但更加复杂的概率模型来估算这些概率。下面介绍一下该算法的一些思路结构，详细的计算推导过程可以参见 [10]，里面牵扯到很多关于概率论的知识，计算量也比较庞大，在这里不多加叙述。

建立两个新的向量参数：$\alpha \in \mathbb{R}^K$ 表示每个主题的权重；$\beta \in \mathbb{R}^N$ 表示每个单词的权重。在 LDA 模型中我们利用狄利克雷分布（*Dirichlet distribution*）来拟合概率产

"Arts"	"Budgets"	"Children"	"Education"
NEW	MILLION	CHILDREN	SCHOOL
FILM	TAX	WOMEN	STUDENTS
SHOW	PROGRAM	PEOPLE	SCHOOLS
MUSIC	BUDGET	CHILD	EDUCATION
MOVIE	BILLION	YEARS	TEACHERS
PLAY	FEDERAL	FAMILIES	HIGH
MUSICAL	YEAR	WORK	PUBLIC
BEST	SPENDING	PARENTS	TEACHER
ACTOR	NEW	SAYS	BENNETT
FIRST	STATE	FAMILY	MANIGAT
YORK	PLAN	WELFARE	NAMPHY
OPERA	MONEY	MEN	STATE
THEATER	PROGRAMS	PERCENT	PRESIDENT
ACTRESS	GOVERNMENT	CARE	ELEMENTARY
LOVE	CONGRESS	LIFE	HAITI

生，多项分布（*Multinomial distribution*）来拟合各主题在文档上的分布，这也是 LDA 名字的来源。

定义 12 关于参数 $\alpha \in \mathbb{R}^K$ 的 ***K*** **阶狄利克雷分布（*Dirichlet distribution*）**的概率密度函数为：

$$f(x_1, ..., x_K; \alpha) = \frac{1}{B(\alpha)} \prod_{k=1}^{K} x_k^{\alpha_i - 1} \tag{5.4.2}$$

其中向量 x 满足 $x_k > 0$ 且 $\sum_k x_k = 1$；归一化变量 $B(\alpha) = \frac{\prod_{k=1}^{K} \Gamma(\alpha_k)}{\Gamma(\sum_{k=1}^{K} \alpha_k)}$。[4]

狄利克雷分布经常用于模拟一个概率的分布，所以他表示了某个“概率发生的概率”。因此在 LDA 模型中我们假设概率 $\theta_i \sim Dir(\alpha)$，$\phi_k \sim Dir(\beta)$。

定义 13 某随机事件有 K 个可能结局 $A_1, A_2, ..., A_K$，分别将它们的出现次数记为随机变量 $X_1, X_2, ..., X_K$；每个结果发生概率分别是 $p_1\ p_2\ ...\ p_k$。如果改事件满足**多项分布（*Multinomial distribution*）**，那么在 N 次采样的总结果中，A_k 出现 n_k 次这种情况的出现概率 P 为：

$$\mathbb{P}[X_1 = n_1, ..., X_K = n_K] = \begin{cases} \frac{\prod_k (n_k!)}{N!} \prod_k p_k^{n_k} & if \sum_k n_k = N \\ 0 & otherwise \end{cases} \tag{5.4.3}$$

那么对于文档 d_i 我们可以利用一系列的概率分布来预测它里面的各个单词分布：

1. 从 $Dir(\alpha)$ 中取样生成 d_i 的主题分布 $\theta_i \sim Dir(\alpha)$;
2. 利用主题的多项分布 $Multinomial(\theta_i)$ 中取样生成文档 i 第 j 个词 d_{ij} 的主题 $z_{ij} \sim Multinomial(\theta_i)$;

[4]伽马函数 $\Gamma(x) = \int_0^{+\infty} t^{x-1} e^{-t} dt$

3. $Dir(\beta)$ 分布中取样生成主题 z_{ij} 的词语分布 $\phi_{z_{ij}} \sim Dir(\beta)$;

4. 从词语的 $Multinomial(\phi_{z_{ij}})$ 中采样最终生成词语 $w_{ij} \sim Multinomial(\phi_{z_{ij}})$。

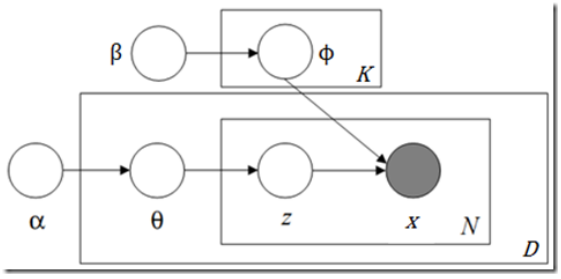


图 13: LDA 概率模型预测

我们的主要目标在于预测每篇文档属于主题 k 的概率：

$$\mathbb{P}[\theta, z|d, \alpha, \beta] = \frac{\mathbb{P}[\theta, z, w|\alpha, \beta]}{\mathbb{P}[d|\alpha, \beta]}$$

但是这个值很难直接算得，所以为了估测参数 α, β 的值，根据之前的假设，我们可以算得：

$$\mathbb{P}[d|\alpha, \beta] = \frac{\Gamma(\sum_k \alpha_k)}{\prod_k \Gamma(\alpha_k)} \int (\prod_{k=1}^{K} \theta_k^{\alpha_k - 1})(\prod_{i=1}^{m} \sum_{k=1}^{K} \prod_{j=1}^{N} (\theta_k \beta_{kj})^{w_i^j}) d\theta \tag{5.4.4}$$

其中 $w_i^j = 1$ 表示文档 d 的第 i 个单词为 w_j，否则为 0. 我们的目的在于找到合适的参数 α, β 使得似然函数 $l(\alpha, \beta) = \sum_{i=1}^{M} \log(\mathbb{P}[d_i|\alpha, \beta])$ 值最大。在这里我们可以利用 EM 算法来估计参数。(假设 α, β 值固定不动，取了两个新的变量 γ 和 ϕ)：

Algorithm 8 EM 算法处理 LDA 中的 variational inference 问题

1: 定义 $\forall k, n\ \phi_{ni}^0 = 1/K$;
2: 定义 $\forall k\ \gamma_k = \alpha_k + N/k$
3: **repeat**
4: **for** n =1:N **do**
5: **for** k=1:K **do**
6: 计算 $\phi_{nk}^{t+1} = \beta_{kw_n} exp(\Psi(\gamma_k^t))$;
7: **end for**
8: 归一化 ϕ_{nk}^{t+1} 使它对 k 求和 =1;
9: **end for**
10: $\gamma^{t+1} = \alpha + \sum_{n=1}^{N} \phi_n^{t+1}$
11: **until** log-likelihood 函数收敛。

其中 Ψ 表示 Digamma 函数[5]。

[5]digamma 函数 $\Psi(x) = \frac{d}{dx} \ln(\Gamma(x)) = \frac{\Gamma'(x)}{\Gamma(x)}$

Spark 中也提供了另一种一种通过在小批量数据上迭代采样实现 online 变分推断，比较节省内存。在线变分预测是一种训练 LDA 模型的技术，它以小批次增量式地处理数据。由于每次处理一小批数据，我们可以轻易地将其扩展应用到大数据集上。MLlib 按照 Hoffman 论文 [11] 里最初提出的算法实现了一种在线变分学习算法。

Algorithm 9 Online 算法处理 LDA

随机定义初始值 λ
while 终止条件不成立 **do**
  E-step:
  for i = 1 to M **do**
    初始化 $\gamma_{ik} = 1$;
    repeat
      $\phi_{iwk} \propto exp(\mathbb{E}_q[log\ \theta_{ik}] + \mathbb{E}_q[log\ \beta_{kw}])$
      $\gamma_{dk} = \alpha + \sum_w \varphi_{iwk} m_{iw}$
    until γ_{ik} 收敛
  end for
  M-step:
  $\lambda = \eta + \sum_i n_{iw} \varphi_{iwk}$
end while

MLlib 函数 LDA 算法主要用于对文档和单词的主题归类，提供了两种参数估计的算法（EM 和 Online 模式）

```
from pyspark.mllib.clustering import LDA, LDAModel
from pyspark.mllib.linalg import Vectors

#Load and parse the data
data = sc.textFile("D:/spark/data/mllib/sample_lda_data.txt")
parsedData = data.map(lambda line: Vectors.dense([float(x) for x in line.strip().split(' ')]))

# Index documents with unique IDs
corpus = parsedData.zipWithIndex().map(lambda x: [x[1], x[0]]).cache()

# Cluster the documents into three topics using LDA
ldaModel = LDA.train(corpus, k=3)
```

```
class LDA(object):
    def train(cls, rdd, k=10, maxIterations=20, docConcentration=-1.0,
        topicConcentration=-1.0, seed=None, checkpointInterval=10, optimizer="em"):
```

1. rdd: 文档中每个单词的出现次数的 RDD，前几列是词典中个单词出现次数，最后一列为文档编号；
2. k: 主题数量，默认 =10；
3. maxIterations：EM 算法的最大迭代次数，默认 =20；
4. docConcentration: Dirichlet 分布的参数 α, 默认为 -1，自动设置;
5. topicConcentration: Dirichlet 分布的参数 β：主题在单词上的先验分布参数。默认为 -1，自动设置；
6. seed: 设置参数的随机种子，默认 =None，表示根据系统时间设定；
7. checkpointInterval：检查点间隔。maxIterations 很大的时候，检查点可以帮助减少 shuffle 文件大小并且可以帮助故障恢复;
8. optimizer: 优化算法的选择：
   - "em"（默认）：传统 EM 算法（Algorithm 8）；
   - "online"：Hoffman 的算法（Algorithm 9）；

我们可以根据得到的模型观察每个主题中各单词的权重（ϕ），归一化后可得到相应的概率:

```
Topic 0:
5.46268609343
20.7387472545
3.52706975512
1.18432470909
7.56863582621
9.62999109599
18.9539387314
2.63380340024
4.5520986752
11.755358845
2.71989198152
```

5.5 Bisecting k-Means - 二分 k 均值算法

二分 *k* 均值算法（*bisecting k-means*）是一种层次聚类（*hierarchical clustering*）算法，它比传统 k-Means 跟高效，而且往往得到不同的分类结果。

层次聚类　层次聚类（hierarchical clustering）是一种比较常用的聚类方式，它通过生成一系列嵌套的聚类树来完成聚类。它可以再一定程度上回答 k 值的选择问题。常见的有两种生成层次的方式：

1. 合并（自下而上）聚类 (**Agglomerative**): 每个数据样本作为一个簇类，将相似的两个类合为一类，逐渐减少簇类的数量；
2. 分裂（自上而下）聚类 (**Divisive**): 将所有数据样本划为一个簇类，选择最优方案将某个簇类一分为二，逐渐增加簇类数量。

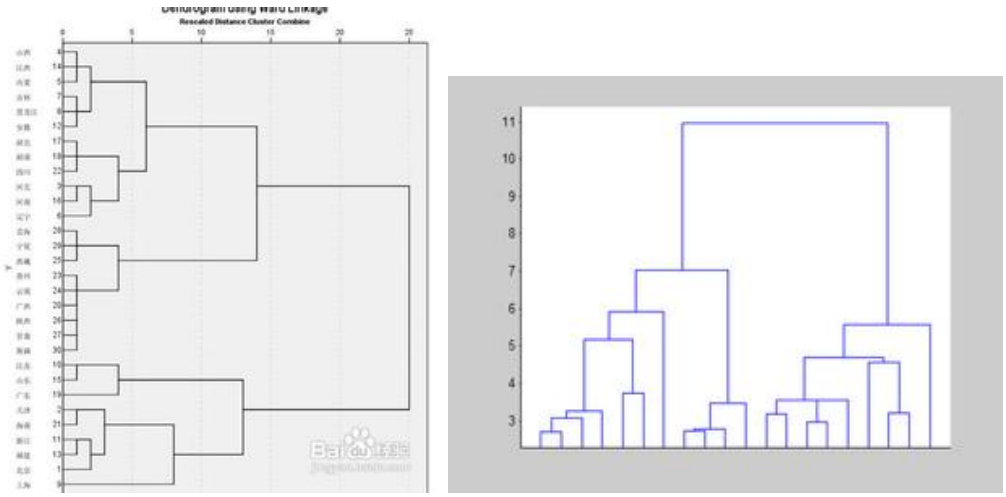


图 14: 层次聚类的聚类树

根据这些建立的聚类树，我们可以选择在某一层“砍一刀”来得到最后的聚类结果以及较合适的 k 值。

Algorithm 10 bisecting k-Means 算法

输入： 数据集 X，K；

输出： K 个质心；

将所有数据归到同一簇类：$f_0(x^{(i)}) = 1$
while 簇类个数小于 K **do**
　for 每个簇类 k **do**
　　执行 2-均值分类，拆成两个簇类；
　　计算代价函数 J 的减小量 ΔJ_k；
　end for
　选择 ΔJ_k 最大的拆分，应用在原数据上。
end while

介绍完层次聚类，我们再回到二分 k 均值算法。它继承了 k-Means 最小化代价函数 $J(f) = \sum_{i=1}^{n} \|x^{(i)} - v_{f(x^{(i)})}\|_2^2$ 的思想，采取了分裂的层次聚类思想，寻找最优的分

裂点使得 J 减小。

与传统的 k-Means 相比，二分 k-Means 加速 K-means 算法的执行速度更快，因为它减少了计算 J 的次数；且它不受初始化问题的影响，因为采用了一种贪心策略，保证了每一步的最优选择。但是作为一种**贪心算法**，我们依然无法保证二分 k-Means 可以取到代价函数 J 的全局最小值。

MLlib 函数 Mllib 中提供的二分 k 均值算法函数的应用例子：

```
from numpy import array
from pyspark.mllib.clustering import BisectingKMeans, BisectingKMeansModel

# Load and parse the data
data = sc.textFile("data/mllib/kmeans_data.txt")
parsedData = data.map(lambda line: array([float(x) for x in line.split(' ')]))

# Build the model (cluster the data)
model = BisectingKMeans.train(rdd=parsedData, k=2, maxIterations=5)
```

代码 25: 二分 k 均值聚类分析数据

```
class BisectingKMeans(object):
    def train(self, rdd, k=4, maxIterations=20, minDivisibleClusterSize=1.0, seed=-1888008604):
```

1. rdd: 训练数据集；
2. k: 簇类总数，默认 =4，最后结果可能比给定值小（如果无法继续划分）；
3. maxIterations: 最大迭代次数
4. minDivisibleClusterSize: 每次划分时，子簇类中的最少数据量 (默认 1.0)：
   - 当值 ≥ 1.0 时，表示每个簇类至少有的数据量；
   - 当值 <1.0 时，每次划分簇类时，子类中数据量的最小百分比。
5. seed: 划分簇类时的一个随机种子，默认 =-1888008604（该类的哈希值）。

我们可以利用得到的模型比较快的计算出代价函数的值：

```
# Evaluate clustering
cost = model.computeCost(parsedData)
print("Bisecting K-means Cost = " + str(cost))
```

代码 26: 测试二分 k 均值模型

```
Bisecting K-means Cost = 0.12
```

5.6 streaming k-means - 流式 k 均值

当我们又实时的数据更新时，流式 k 均值（streaming k-means）可以帮助我们不断地更新所有簇类的质心坐标，完善模型的准确率。假设在 t 时刻，我们的模型定义为 $(v_k^t)_{1\leq k\leq K}$，K 个质心；每个簇类中已有 n_k^t 个数据样本。在 t+1 时刻又输入了一批数据集，利用已有模型对其进行分类，并更新模型

$$v_k^{t+1} = \frac{v_k^t n_k^t \alpha + x_k^t m_k^t}{n_k^t \alpha + m_k^t} \tag{5.6.1}$$

$$n_k^{t+1} = n_k^t + m_k^t$$

其中 m_k 表示新的数据集中被分到 k 簇类的点的个数，x_k 表示这些点的质心。α 被称为 decay factor，可以忽视一部分历史数据（如果历史数据量过大，会影响新数据的一些变化）。下面是 MLLib 中的代码实现：

```
from pyspark import SparkContext
from pyspark.streaming import StreamingContext
from pyspark.mllib.linalg import Vectors
from pyspark.mllib.regression import LabeledPoint
from pyspark.mllib.clustering import StreamingKMeans

# initialization spark
sc = SparkContext(appName="Stream_kMeans")
# Create a local StreamingContext with two working thread and batch interval of 1 second
ssc = StreamingContext(sc, 10)

# we make an input stream of vectors for training,
# as well as a stream of vectors for testing
def parse(lp):
    label = float(lp[lp.find('(') + 1: lp.find(')')])
    vec = Vectors.dense(lp[lp.find('[') + 1: lp.find(']')].split(','))
    return LabeledPoint(label, vec)


trainingData = sc.textFile("D:/spark/data/mllib/kmeans_data.txt") \
    .map(lambda line: Vectors.dense([float(x) for x in line.strip().split(' ')]))

testingData = sc.textFile("D:/spark/data/mllib/streaming_kmeans_data_test.txt").map(parse)

```

```
25 trainingQueue = [trainingData]
26 testingQueue = [testingData]
27
28 trainingStream = ssc.queueStream(trainingQueue)
29 testingStream = ssc.queueStream(testingQueue)
30
31 # We create a model with random clusters and specify the number of clusters to find
32 model = StreamingKMeans(k=2, decayFactor=1.0).setRandomCenters(3, 1.0, 0)
33
34 # Now register the streams for training and testing and start the job,
35 # printing the predicted cluster assignments on new data points as they arrive.
36 model.trainOn(trainingStream)
37
38 result = model.predictOnValues(testingStream.map(lambda lp: (lp.label, lp.features)))
39 result.pprint()
40
41 ssc.start()
42 ssc.stop(stopSparkContext=True, stopGraceFully=True)
```

第 10 行的 ssc 初始了 Spark 的流式环境，第二个参数 10 表示 10 秒接受一次新数据。我们以 trainingData 作为原数据，新接受 testingData 做预测并更新模型。第 32 行中，我们随机建立了一个初始化的模型，设置各种参数。在第 36 行，我们对已有的 Training 数据做 k-Means 聚类，在 38 行接受新数据并预测。41 行表示命令 Spark 开启流式环境，开始执行。

参考文献

[1] 百度百科 - 假设检验. http://baike.baidu.com/link?url=31QQV13j75HNfr_Hqt_OddrGlLSxxZcjwDQjvnL7TZM6jOxs4JCs7i5e9-OXlQEF5LVSscNnz3TjJ51m7WrS0gdVq68AaUUFFQvYNFaQP. Accessed 2016/12/22.

[2] 百度百科 - 协同过滤. http://baike.baidu.com/link?url=6GFmEWz0Hg7JfLXQEcBw69SIJfWDUMCOJBHk2FHef1acHz19nMiZLYtTdrbXFC3C4rnc5CKZQaJplEBFeEqHtLeGuhH6Gj8acl-Ii. Accessed 2016/12/22.

[3] Mllib: Rdd-based api. http://spark.apache.org/docs/latest/mllib-guide.html. Accessed 2016/12/22.

[4] Wikipedia[en] - correlation and dependence. https://en.wikipedia.org/wiki/Correlation_and_dependence. Accessed 2016/12/22.

[5] Wikipedia[en] - k-means++. https://en.wikipedia.org/wiki/K-means%2B%2B. Accessed 2016/12/22.

[6] Wikipedia[en] - loss functions for classification. https://en.wikipedia.org/wiki/Loss_functions_for_classification. Accessed 2016/12/22.

[7] Wikipedia[en] - spearman's rank correlation coefficient. https://en.wikipedia.org/wiki/Spearman%27s_rank_correlation_coefficient. Accessed 2016/12/22.

[8] Wikipedia[en] - tikhonov regularization. https://en.wikipedia.org/wiki/Tikhonov_regularization. Accessed 2016/12/22.

[9] Wikipedia[fr] - loi du χ^2. https://fr.wikipedia.org/wiki/Test_du_%CF%87%C2%B2. Accessed 2016/12/22.

[10] David M. Blei, Andrew Y. Ng, and Michael I. Jordan. Latent dirichlet allocation. *Journal of Machine Learning Research*, 3:993–1022, 2003.

[11] Matthew D. Hoffman, David M. Blei, and Francis R. Bach. Online learning for latent dirichlet allocation. *Advances in Neural Information Processing Systems*, 23:856–864, 2010.

[12] Yehuda Koren, Robert Bell, and Chris Volinsky. Matrix factorization techniques for recommender systems. *Computer*, 42(8):30–37, 2009.

[13] M. Meila and J. Shi. A random walks view of spectral segmentation. *Ai & Statistics*, 2001.

[14] SANDY RYZA. *Spark* 高级数据分析. 人民邮电出版社, 2015.

[15] Yunhong Zhou, Dennis Wilkinson, Robert Schreiber, and Rong Pan. Large-scale parallel collaborative filtering for the netflix prize. In *Algorithmic Aspects in Information and Management, International Conference, Aaim 2008, Shanghai, China, June 23-25, 2008. Proceedings*, pages 337–348, 2008.

最优化算法介绍

什么是最优化（optimization）？

数学模型
给定一个函数$f: \mathbf{R}^n \to \mathbf{R}^m$，在给定集合$A \subset \mathbf{R}^n$中寻找一个元素 x 取得$f(x)$的最大值或者最小值。在一般情况下，我们考虑该函数f的最小值。寻找$x^* = argmin_{x \in A} f(x) = \{x \in A | \forall x' \in A, f(x) \leq f(x')\}$ 。在机器学习中，我们需要广泛应用到这一技术，例如我们在选取目标函数的参数时，需要取代价函数（*cost function*）的最小值。在这种情况下，我们就需要利用最优化算法找到对应的参数值。一个好的优化算法不仅可以快速的提高计算速度，还可以大大减小计算时所需要的占用的内存/硬盘空间。

例如在线性规划（*linear regression*）中我们需要预测自变量 x 和因变量 y 之间线性关系 $y = f_{\theta,b}(x) = \theta x + b$，就必须要预测θ和 b 两个参数。这里我们定义代价函数为 $J(\theta, b) = \sum_i \left(y_i - f_{\theta,b}(x_i)\right)^2$。为了使代价函数最小化，也就是让我们的预测值$f_{\theta,b}(x_i)$更接近准确值$y_i$，我们就可以利用最优化算法寻找合适的$\theta$和 b 的值。

如果你对机器学习算法有一定了解，一定听说过 svm（support vector machine,支持向量机）算法，其实早在 1963 年，Vladimir N. Vapnik 和 Alexey Ya. Chervonenkis 就提出了线性的 svm 算法并在 1992 年提出了带核函数的改进版本。但是直到 1998 年，由 John Platt 提出 SMO（Sequential Minimal Optimization）的优化算法之后才使得用计算机实现 svm 算法成为可能，该算法大大加快了 svm 的计算效率（当然，也离不开计算机硬件的飞速发展）。由此可见优化算法在数学和计算机领域的重要性。

利用目标函数的导数通过多次迭代来求解无约束的最优化问题是求解最优化问题的最常见方法。它实现简单，coding 方便，是训练模型的必备利器之一。今天我们就介绍比较基础的利用迭代思想求解最优解的四种算法。

一些数学概念

◆ 梯度（gradient）
对于一个标量函数f：$\mathbf{R}^n \to \mathbf{R}$，它的梯度定义为对 x 每一维度的一阶偏导，即

$$\nabla_x(f) = \begin{pmatrix} \frac{\partial f}{\partial x_1} \\ \cdots \\ \cdots \\ \frac{\partial f}{\partial x_n} \end{pmatrix} \in \mathbf{R}^n$$

◆ Hesse 矩阵
梯度表示的是函数的一阶偏导，而 Hesse 矩阵则记录了函数的二阶偏导。其第 i 行第 j 列元

素表示为

$$H(f)_{i,j} = \frac{\partial^2 f}{\partial x_i \partial x_j}$$

例如在二次函数$f(x) = \frac{1}{2}x^T Ax + bx + c$（A 是 n 阶对称矩阵，b 是 n 维向量，c 为常数）中，

根据定义我们有$\nabla f(x) = Ax + b$以及$H(f) = A$

◆ Jacobi 矩阵

对于函数$f: \mathbf{R}^n \to \mathbf{R}^m$，可以写为$f(x) = (f_1(x), \dots, f_m(x))$，它的 jacobi 矩阵定义为

$$J(f)_{i,j} = \frac{\partial f_i}{\partial x_j}$$

所以矩阵$J(f)$是一个$m \times n$阶矩阵。

特例，当$m = 1$时，Jocabi 矩阵就是函数的梯度，而且$H(f) = J(\nabla f)$

优化方法

几乎所有的优化算法都会用到迭代的思想，并且在每次迭代更新中，会定义一个向量的**移动方向**p_k和**移动距离**α_k，直到寻找到合适的x^*近似值。

$$x^{k+1} = x^k + \alpha_k p_k$$

◆ **梯度下降法（Gradient Descent）**

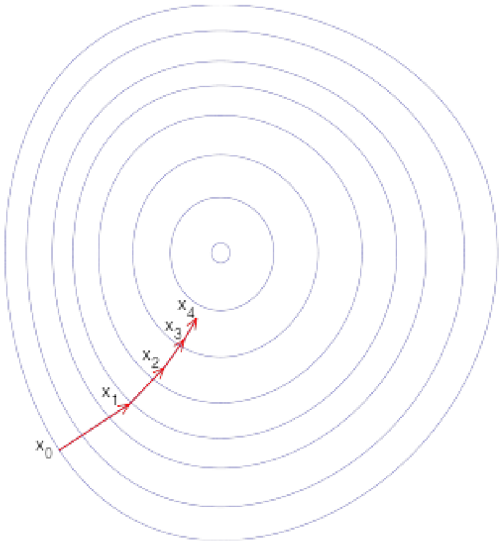
又叫 steepest descent，举一个盲人下山的例子。当盲人下山时，眼睛看不见山谷的方位，如何能沿最短路线迅速下降呢？一般盲人只好靠手前后探索试探着前进的方向，哪儿最陡？一定下降的最快，这种寻求最速下降的方向作为搜索的方向，一步步逼近最低点就是最速下降的基本思想。

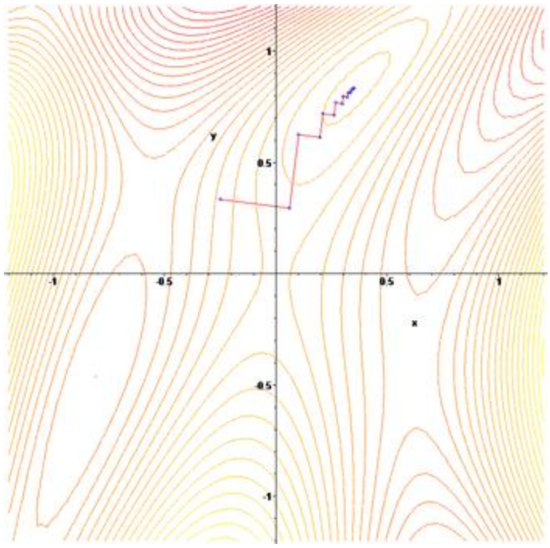
每次运算，定义移动方向（梯度负方向）和移动距离（α_k）。所以每次迭代中

$$x^{k+1} = x^k - \alpha_k \nabla f(x^k)$$

其中α_k又叫学习率（Learning Rate），可以根据需求你可以在每次迭代中重新计算一个全新的最优的α_k（这样可以减少迭代次数，但是会大大增加每次迭代中的计算量）。

当然我们也可以在梯度下降算法中选取一个常数作为学习率，记为α。需要注意我们要取合适的α，使得数列(x_k)收敛：若α过大，则无法收敛；相反若α太小，则收敛缓慢，计算成本高。





注意：该算法求得的是局部最小值，并非全局最小值。根据初始值x_0和α不同，会落到不同的极小值点。如果 f 是凸函数（convex），则取到全局最优解。

粗略来讲，在二次函数中，椭球面的形状受 hesse 矩阵的条件数影响，长轴与短轴对应矩阵的最小特征值和最大特征值的方向，其大小与特征值的平方根成反比，最大特征值与最小特征值相差越大，椭球面越扁，那么优化路径需要走很大的弯路，计算效率很低。

衍生
random gradient descent 随机梯度下降法
batch gradient descent 批量梯度下降法

◆ 牛顿法（Newton's Method/Newton-Raphson）

利用 Taylor's formula 泰勒展开将目标函数二次展开

$$f(x) = f(x^k) + \nabla f(x^k)(x - x^k) + \frac{1}{2}(x - x^k)^T H(f)(x^k)(x - x^k) + o\left(\left|\left|x - x^k\right|\right|^2\right)$$

寻找 f 的最小值，我们知道函数在极值点的导数为 0
利用二阶的泰勒展开式，对 x 求导

$$x^{k+1} = x^k - \alpha_k H(f)(x^k)^{-1} \nabla f(x^k)$$

相比与梯度下降法，牛顿法利用了二阶偏导，对目标函数形状有了更好的预测，所以可以求出近似函数的全局最小值，而且他的收敛速度也比较快。
要注意的是，在这个算法中，我们首先要保证 Hesse 矩阵的可逆性，其次考虑到计算逆矩阵的大量计算成本（$O(n^3)$），当 x 维度过高时，并不适合用这一方法。
如果初始点离极值点较远，那么此时泰勒展开中误差较大，导致方向选择比较糟糕。

◆ Levenburg-Marquardt Algorithm（LMA）

LM 算法的关键是用模型函数 f 对待估参数向量 p 在其领域内做线性近似，忽略掉二阶以上的导数项，从而转化为线性最小二乘问题，它结合了以上两种算法的优点并改进了他们的不

足。LM 算法属于一种“信赖域法”（trust region algorithm），所谓的信赖域法，即是：在最优化算法中，都是要求一个函数的极小值，每一步迭代中，都要求目标函数值是下降的，而信赖域法，顾名思义，就是从初始点开始，先假设一个可以信赖的最大位移 s，然后在以当前点为中心，以 s 为半径的区域内，通过寻找目标函数的一个近似函数（二次的）的最优点，来求解得到真正的位移。在得到了位移之后，再计算目标函数值，如果其使目标函数值的下降满足了一定条件，那么就说明这个位移是可靠的，则继续按此规则迭代计算下去；如果其不能使目标函数值的下降满足一定的条件，则应减小信赖域的范围，再重新求解。

该算法最早运用在最小二乘问题中，对于$S(\theta)=\sum_i\left(y_i-f(x_i,\theta)\right)^2$

$$f(x_i,\theta+\delta_\theta)\approx f(x_i,\theta)+J_\theta(f)(x_i,\theta).\delta_\theta$$

其中$J_\theta(f)$是f函数关于参数θ的 jacobi 矩阵。对于代价函数，我们可以近似认为

$$S(\theta+\delta_\theta)\approx \Sigma_i[y_i-f(x_i,\theta)-J_\theta(f)(x_i,\theta).\delta_\theta]^2$$

当$\frac{\partial S}{\partial\delta_\theta}(\theta+\delta_\theta)=0\Leftrightarrow\left(J_\theta^T.J\right)\delta_B=J_\theta^T(Y-f(X,\theta))$

我们在这一项中加上阻尼系数

$$\left(J_\theta^T.J_\theta+\lambda\mathrm{diag}(J_\theta^T.J_\theta)\right)\delta_B=J_\theta^T(Y-f(X,\theta))$$

λ的大小直接关系了信赖域的大小。当λ较大时，信赖域变小，LM 算法接近梯度下降法；当λ较小时，信赖域大，LM 算法接近牛顿法。

在每次迭代中，我们都需要判断 S 函数减少量，并依此来调整λ的大小。

1. 给定初始量θ_0
2. 判断$J_\theta^T\left(Y-f(X,\theta)\right)>\eta$ 并且没有达到最大迭代次数，执行
3. 符合条件 2 后，重复执行
   - 计算移动向量δ_θ
   - 计算更新值$\theta^{new}=\theta+\delta_\theta$
   - 计算 S 函数的真实减小量和预测值的比例ρ

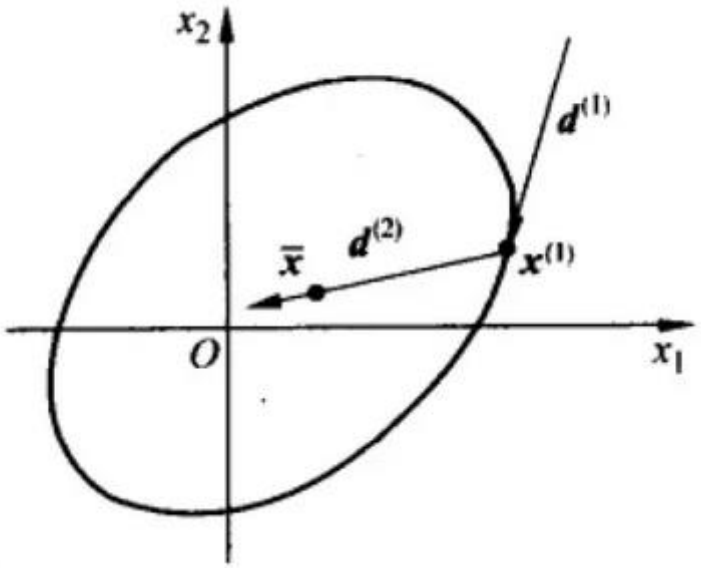
     i. $0<\rho<0.25$ 减小量良好，接受θ^{new}
     ii. $\rho\geq0.25$ 减小值大，接受θ^{new}，并减小λ
     iii. $\rho\leq0$ S 函数变大，拒绝接受θ^{new}，增加λ
4. 不符合条件 2 时停止

注意：LM 算法中用到的 jacobi 矩阵并不是针对目标函数 S 的，而是对于函数 f 的。

◆ **共轭梯度法（conjugate gradients）**

共轭梯度法是解优化问题的有效方法之一，特别是用于二次泛函指标系统。共轭梯度法程序清单容易实现，具有梯度法优点，而在收敛速度方面比梯度法快。

该算法的主要优点在于向量移动方向p_k的选择上。我们选择p_k作为x_k关于 Hesse 矩阵的共轭，i.e.$<x_k,p_k>_H=<x_k,Hp_k>=x_k^T.H.p_k=0$



确定了移动方向之后，可以进一步确定移动距离的值。