机器学习中的数学(1)-回归(regression)、梯度下降(gradient descent)

前言：

上次写过一篇关于贝叶斯概率论的数学，最近时间比较紧，**coding** 的任务比较重，不过还是抽空看了一些机器学习的书和视频，其中很推荐两个：一个是 **stanford** 的 **machine learning** 公开课，在 **verycd** 可下载，可惜没有翻译。不过还是可以看。另外一个是 **prml-pattern recognition and machine learning, Bishop** 的一部反响不错的书，而且是 **2008** 年的，算是比较新的一本书了。

前几天还准备写一个分布式计算的系列，只写了个开头，又换到写这个系列了。以后看哪边的心得更多，就写哪一个系列吧。最近干的事情比较杂，有跟机器学习相关的，有跟数学相关的，也有跟分布式相关的。

这个系列主要想能够用数学去描述机器学习，想要学好机器学习，首先得去理解其中的数学意义，不一定要到能够轻松自如的推导中间的公式，不过至少得认识这些式子吧，不然看一些相关的论文可就看不懂了，这个系列主要将会着重于去机器学习的数学描述这个部分，将会覆盖但不一定局限于回归、聚类、分类等算法。

回归与梯度下降：

回归在数学上来说是给定一个点集，能够用一条曲线去拟合之，如果这个曲线是一条直线，那就被称为线性回归，如果曲线是一条二次曲线，就被称为二次回归，回归还有很多的变种，如 **locally weighted** 回归，**logistic** 回归，等等，这个将在后面去讲。

用一个很简单的例子来说明回归，这个例子来自很多的地方，也在很多的 **open source** 的软件中看到，比如说 **weka**。大概就是，做一个房屋价值的评估系统，一个房屋的价值来自很多地方，比如说面积、房间的数量（几室几厅）、地段、朝向等等，这些影响房屋价值的变量被称为特征**(feature)**，**feature** 在机器学习中是一个很重要的概念，有很多的论文专门探讨这个东西。在此处，为了简单，假设我们的房屋就是一个变量影响的，就是房屋的面积。

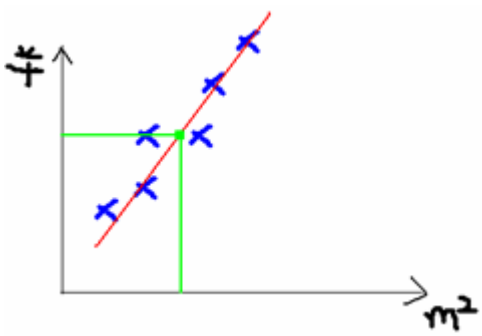
假设有一个房屋销售的数据如下：

面积**(m^2)**	销售价钱（万元）
123	**250**
150	**320**
87	**160**
102	**220**
…	**…**

这个表类似于帝都 **5** 环左右的房屋价钱，我们可以做出一个图，**x** 轴是房屋的面积。**y** 轴是房屋的售价，如下：

如果来了一个新的面积，假设在销售价钱的记录中没有的，我们怎么办呢？

我们可以用一条曲线去尽量准的拟合这些数据，然后如果有新的输入过来，我们可以在将曲线上这个点对应的值返回。如果用一条直线去拟合，可能是下面的样子：



绿色的点就是我们想要预测的点。

首先给出一些概念和常用的符号，在不同的机器学习书籍中可能有一定的差别。

房屋销售记录表 **-** 训练集**(training set)**或者训练数据**(training data)**，是我们流程中的输入数据，一般称为 **x**

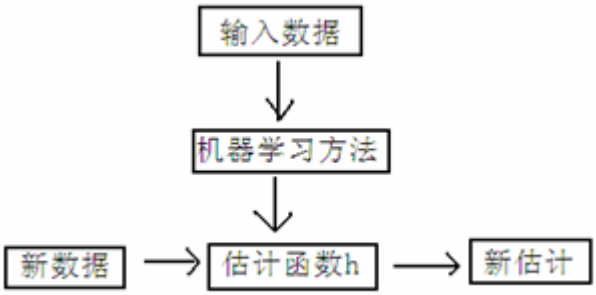
房屋销售价钱 **-** 输出数据，一般称为 **y**

拟合的函数（或者称为假设或者模型），一般写做 **y = h(x)**

训练数据的条目数**(#training set)**，一条训练数据是由一对输入数据和输出数据组成的

输入数据的维度**(**特征的个数，**#features)**，**n**

下面是一个典型的机器学习的过程，首先给出一个输入数据，我们的算法会通过一系列的过程得到一个估计的函数，这个函数有能力对没有见过的新数据给出一个新的估计，也被称为构建一个模型。就如同上面的线性回归函数。



我们用 **X1，X2..Xn** 去描述 **feature** 里面的分量，比如 **x1=**房间的面积，**x2=**房间的朝向，等等，我们可以做出一个估计函数：

$$h(x) = h_\theta(x) = \theta_0 + \theta_1 x_1 + \theta_2 x_2$$

θ 在这儿称为参数，在这儿的意思是调整 **feature** 中每个分量的影响力，就是到底是房屋的面积更重要还是房屋的地段更重要。为了如果我们令 **X0 = 1**，就可以用向量的方式来表示了：

$$h_\theta(x) = \theta^T X$$

我们程序也需要一个机制去评估我们 **θ** 是否比较好，所以说需要对我们做出的 **h** 函数进行评估，一般这个函数称为损失函数（**loss function**）或者错误函数**(error function)**，描述 **h** 函数不好的程度，在下面，我们称这个函数为 **J** 函数

在这儿我们可以做出下面的一个错误函数：

$$J(\theta) = \frac{1}{2}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})^2$$

$$\min_\theta J_\theta$$

这个错误估计函数是去对 **x(i)**的估计值与真实值 **y(i)**差的平方和作为错误估计函数，前面乘上的 **1/2** 是为了在求导的时候，这个系数就不见了。

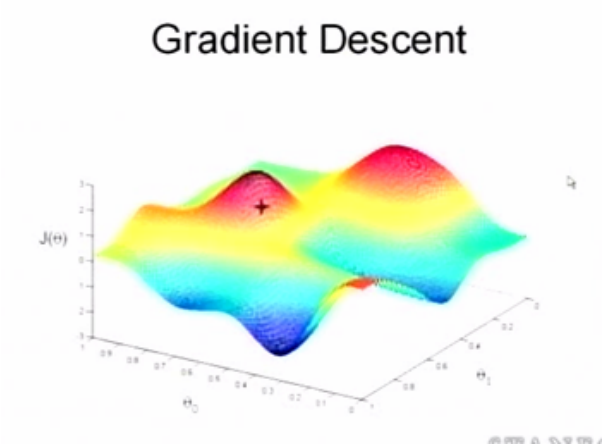
如何调整 **θ** 以使得 **J(θ)**取得最小值有很多方法，其中有最小二乘法**(min square)**，是一种完全是数学描述的方法，在 **stanford** 机器学习开放课最后的部分会推导最小二乘法的公式的来源，这个来很多的机器学习和数学书上都可以找到，这里就不提最小二乘法，而谈谈梯度下降法。

梯度下降法是按下面的流程进行的：

1）首先对 **θ** 赋值，这个值可以是随机的，也可以让 **θ** 是一个全零的向量。

2）改变 **θ** 的值，使得 **J(θ)**按梯度下降的方向进行减少。

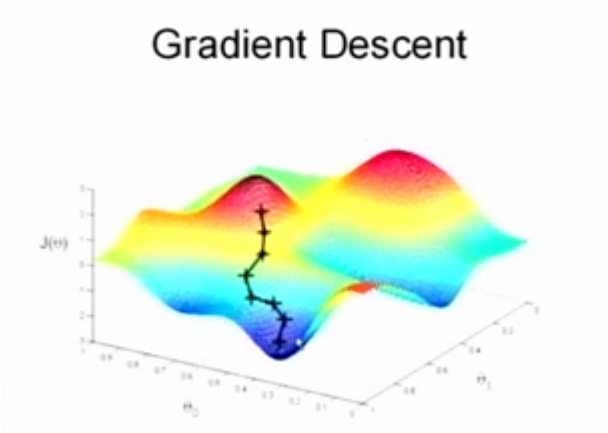
为了更清楚，给出下面的图：



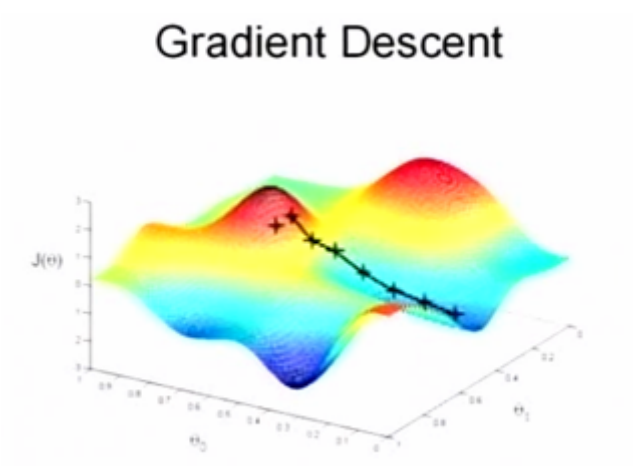
这是一个表示参数 **θ** 与误差函数 **J(θ)**的关系图，红色的部分是表示 **J(θ)**有着比较高的取值，我们需要的是，能够让 **J(θ)**的值尽量的低。也就是深蓝色的部分。**θ0**，**θ1** 表示 **θ** 向量的两个维度。

在上面提到梯度下降法的第一步是给 **θ** 给一个初值，假设随机给的初值是在图上的十字点。

然后我们将 **θ** 按照梯度下降的方向进行调整，就会使得 **J(θ)**往更低的方向进行变化，如图所示，算法的结束将是在 **θ** 下降到无法继续下降为止。



当然，可能梯度下降的最终点并非是全局最小点，可能是一个局部最小点，可能是下面的情况：



上面这张图就是描述的一个局部最小点，这是我们重新选择了一个初始点得到的，看来我们这个算法将会在很大的程度上被初始点的选择影响而陷入局部最小点

下面我将用一个例子描述一下梯度减少的过程，对于我们的函数 **J(θ)**求偏导 **J**：（求导的过程如果不明白，可以温习一下微积分）

$$\frac{\partial}{\partial\theta}J(\theta)=\frac{\partial}{\partial\theta}\frac{1}{2}\sum_{i=1}^{m}(h_\theta(x)-y)^2=(h_\theta(x)-y)x^{(i)}$$

下面是更新的过程，也就是 **θi** 会向着梯度最小的方向进行减少。**θi** 表示更新之前的值，**-**后面的部分表示按梯度方向减少的量，**α** 表示步长，也就是每次按照梯度减少的方向变化多少。

$$\theta_i=\theta_i-\alpha\frac{\partial}{\partial\theta}J(\theta)=\theta_i-\alpha(h_\theta(x)-y)x^{(i)}$$

一个很重要的地方值得注意的是，梯度是有方向的，对于一个向量 **θ**，每一维分量 **θi** 都可以求出一个梯度的方向，我们就可以找到一个整体的方向，在变化的时候，我们就朝着下降最多的方向进行变化就可以达到一个最小点，不管它是局部的还是全局的。

用更简单的数学语言进行描述步骤 **2**）是这样的：

$$\nabla_\theta J=\begin{bmatrix}\frac{\partial}{\partial\theta_0}J\\ \vdots\\ \frac{\partial}{\partial\theta_n}J\end{bmatrix}$$

$$\theta=\theta-\alpha\nabla_\theta J$$

倒三角形表示梯度，按这种方式来表示，**θi** 就不见了，看看用好向量和矩阵，真的会大大的简化数学的描述啊。

总结与预告：

本文中的内容主要取自 **stanford** 的课程第二集，希望我把意思表达清楚了：）本系列的下一篇文章也将会取自 **stanford** 课程的第三集，下一次将会深入的讲讲回归、**logistic** 回归、和 **Newton** 法，不过本系列并不希望做成 **stanford** 课程的笔记版，再往后面就不一定完全与 **stanford** 课程保持一致了。

机器学习中的数学(2)-线性回归，偏差、方差权衡

前言：

距离上次发文章，也快有半个月的时间了，这半个月的时间里又在学习机器学习的道路上摸索着前进，积累了一点心得，以后会慢慢的写写这些心得。写文章是促进自己对知识认识的一个好方法，看书的时候往往不是非常细，所以有些公式、知识点什么的就一带而过，里面的一些具体意义就不容易理解了。而写文章，特别是写科普性的文章，需要对里面的具体意义弄明白，甚至还要能举出更生动的例子，这是一个挑战。为了写文章，往往需要把之前自己认为看明白的内容重新理解一下。

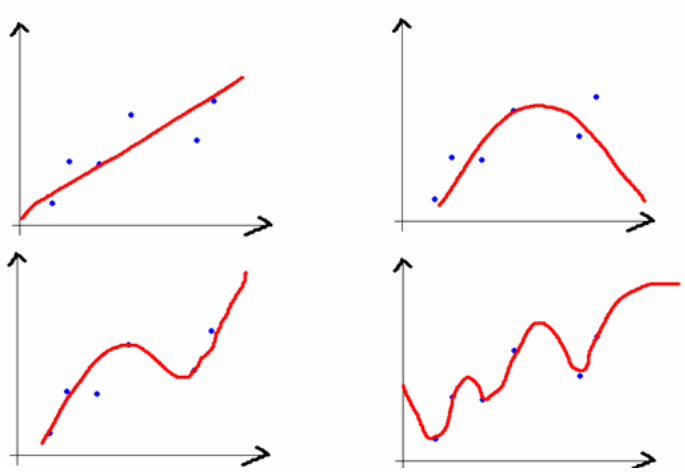
机器学习可不是一个完全的技术性的东西，之前和部门老大在 **outing** 的时候一直在聊这个问题，机器学习绝对不是一个一个孤立的算法堆砌起来的，想要像看《算法导论》这样看机器学习是个不可取的方法，机器学习里面有几个东西一直贯穿全书，比如说数据的分布、最大似然（以及求极值的几个方法，不过这个比较数学了），偏差、方差的权衡，还有特征选择，模型选择，混合模型等等知识，这些知识像砖头、水泥一样构成了机器学习里面的一个个的算法。想要真正学好这些算法，一定要静下心来将这些基础知识弄清楚，才能够真正理解、实现好各种机器学习算法。

今天的主题是线性回归，也会提一下偏差、方差的均衡这个主题。

线性回归定义：

在上一个主题中，也是一个与回归相关的，不过上一节更侧重于梯度这个概念，这一节更侧重于回归本身与偏差和方差的概念。

回归最简单的定义是，给出一个点集 **D**，用一个函数去拟合这个点集，并且使得点集与拟合函数间的误差最小。



上图所示，给出一个点集**(x,y)**，需要用一个函数去拟合这个点集，蓝色的点是点集中的点，而红色的

曲线是函数的曲线，第一张图是一个最简单的模型，对应的函数为 **y = f(x) = ax + b**，这个就是一个线性函数，

第二张图是二次曲线，对应的函数是 **y = f(x) = ax^2 + b**。

第三张图我也不知道是什么函数，瞎画的。

第四张图可以认为是一个 **N** 次曲线，**N = M - 1**，**M** 是点集中点的个数，有一个定理是，对于给定的 **M** 个点，我们可以用一个 **M - 1** 次的函数去完美的经过这个点集。

真正的线性回归，不仅会考虑使得曲线与给定点集的拟合程度最好，还会考虑模型最简单，这个话题我们将在本章后面的偏差、方差的权衡中深入的说，另外这个话题还可以参考我之前的一篇文章：贝叶斯、概率分布与机器学习，里面对模型复杂度的问题也进行了一些讨论。

线性回归**(linear regression)**，并非是指的线性函数，也就是

$$f(x) = a_0x_0 + a_1x_1 + a_2x_2 + \cdots = \vec{a}^T\vec{x}$$

（为了方便起见，以后向量我就不在上面加箭头了）

x0,x1...表示一个点不同的维度，比如说上一节中提到的，房子的价钱是由包括面积、房间的个数、房屋的朝向等等因素去决定的。而是用广义的线性函数：

$$y(x,w) = w^Tx = w_0 + \sum_{j=1}^{M-1} w_j\phi_j(x)$$

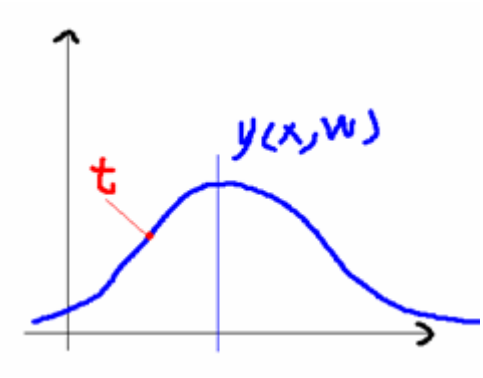
wj 是系数，**w** 就是这个系数组成的向量，它影响着不同维度的 **Φj(x)**在回归函数中的影响度，比如说对于房屋的售价来说，房间朝向的 **w** 一定比房间面积的 **w** 更小。**Φ(x)**是可以换成不同的函数，不一定要求 **Φ(x)=x**，这样的模型我们认为是广义线性模型。

最小二乘法与最大似然：

这个话题在此处有一个很详细的讨论，我这里主要谈谈这个问题的理解。最小二乘法是线性回归中一个最简单的方法，它的推导有一个假设，就是回归函数的估计值与真实值间的误差假设是一个高斯分布。这个用公式来表示是下面的样子：$t = y(x,w) + \varepsilon$ ，**y(x,w)**就是给定了 **w** 系数向量下的回归函数的估计值，而 **t** 就是真实值了，**ε** 表示误差。我们可以接下来推出下面的式子：

$$p(t \mid x, w, \beta) = \mathrm{N}(t \mid y(w,x), \beta^{-1})$$

这是一个简单的条件概率表达式，表示在给定了 **x**，**w**，**β** 的情况下，得到真实值 **t** 的概率，由于 **ε** 服从高斯分布，则从估计值到真实值间的概率也是高斯分布的，看起来像下面的样子：



贝叶斯、概率分布与机器学习这篇文章中对分布影响结果这个话题讨论比较多，可以回过头去看看，由于最小二乘法有这样一个假设，则会导致，如果我们给出的估计函数 **y(x,w)**与真实值 **t** 不是高斯分布的，甚至是一个差距很大的分布，那么算出来的模型一定是不正确的，当给定一个新的点 **x'**想要求出一个估计值 **y'**，与真实值 **t'**可能就非常的远了。

概率分布是一个可爱又可恨的东西，当我们能够准确的预知某些数据的分布时，那我们可以做出一个非常精确的模型去预测它，但是在大多数真实的应用场景中，数据的分布是不可知的，我们也很难去用一个分布、甚至多个分布的混合去表示数据的真实分布，比如说给定了 **1** 亿篇网页，希望用一个现有的分布（比如说混合高斯分布）去匹配里面词频的分布，是不可能的。在这种情况下，我们只能得到词的出现概率，比如 **p(**的**)**的概率是 **0.5**，也就是一个网页有 **1/2** 的概率出现"的"。如果一个算法，是对里面的分布进行了某些假设，那么可能这个算法在真实的应用中就会表现欠佳。最小二乘法对于类似的一个复杂问题，就很无力了

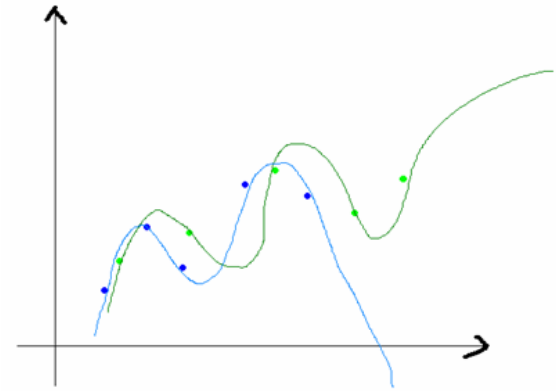
偏差、方差的权衡**(trade-off)**：

偏差**(bias)**和方差**(variance)**是统计学的概念，刚进公司的时候，看到每个人的嘴里随时蹦出这两个词，觉得很可怕。首先得明确的，方差是多个模型间的比较，而非对一个模型而言的，对于单独的一个模型，比如说：

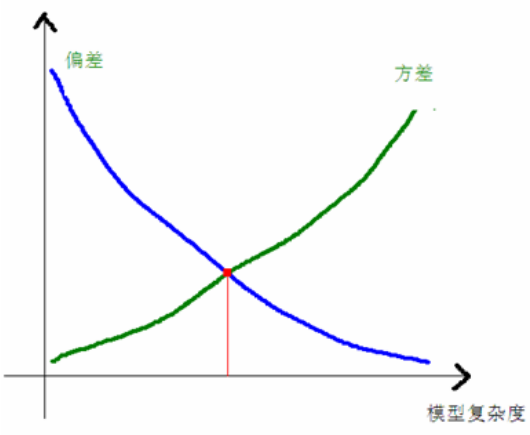
$$f(x) = a_0x_0 + a_1x_1 + a_2x_2 + \cdots = \vec{a}^T\vec{x}$$

这样的一个给定了具体系数的估计函数，是不能说 **f(x)**的方差是多少。而偏差可以是单个数据集中的，也可以是多个数据集中的，这个得看具体的定义。

方差和偏差一般来说，是从同一个数据集中，用科学的采样方法得到几个不同的子数据集，用这些子数据集得到的模型，就可以谈他们的方差和偏差的情况了。方差和偏差的变化一般是和模型的复杂程度成正比的，就像本文一开始那四张小图片一样，当我们一味的追求模型精确匹配，则可能会导致同一组数据训练出不同的模型，它们之间的差异非常大。这就叫做方差，不过他们的偏差就很小了，如下图所示：



上图的蓝色和绿色的点是表示一个数据集中采样得到的不同的子数据集，我们有两个 **N** 次的曲线去拟合这些点集，则可以得到两条曲线（蓝色和深绿色），它们的差异就很大，但是他们本是由同一个数据集生成的，这个就是模型复杂造成的方差大。模型越复杂，偏差就越小，而模型越简单，偏差就越大，方差和偏差是按下面的方式进行变化的**:**



当方差和偏差加起来最优的点，就是我们最佳的模型复杂度。

用一个很通俗的例子来说，现在咱们国家一味的追求 **GDP**，**GDP** 就像是模型的偏差，国家希望现有的 **GDP** 和目标的 **GDP** 差异尽量的小，但是其中使用了很多复杂的手段，比如说倒卖土地、强拆等等，这个增加了模型的复杂度，也会使得偏差（居民的收入分配）变大，穷的人越穷**(**被赶出城市的人与进入城市买不起房的人），富的人越富（倒卖土地的人与卖房子的人）。其实本来模型不需要这么复杂，能够让居民的收入分配与国家的发展取得一个平衡的模型是最好的模型。

最后还是用数学的语言来描述一下偏差和方差：

$$E(L) = \int \{y - h(x)\}^2 p(x)dx + \int \{h(x) - t\}^2 p(x,t)dxdt$$

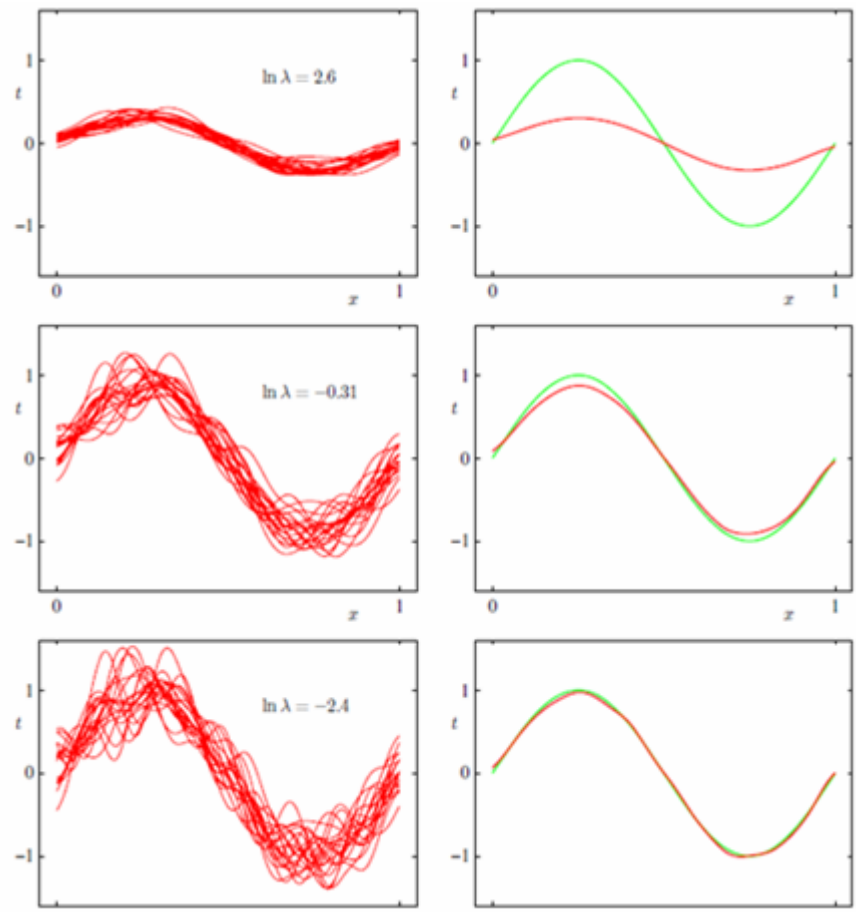
E(L)是损失函数，**h(x)**表示真实值的平均，第一部分是与 **y**（模型的估计函数）有关的，这个部分是由于我们选择不同的估计函数（模型）带来的差异，而第二部分是与 **y** 无关的，这个部分可以认为是模型的固有噪声。

对于上面公式的第一部分，我们可以化成下面的形式：

$$E_D[\{y(x;D)-h(x)\}]=$$
$$\{E_D[y(x;D)]-h(x)\}^2+E_D[\{y(x;D)-E_D[\{y(x;D)]\}^2]$$

这个部分在 **PRML** 的 **1.5.5** 推导，前一半是表示偏差，而后一半表示方差，我们可以得出：损失函数=偏差^2+方差+固有噪音。

下图也来自 **PRML**：



这是一个曲线拟合的问题，对同分布的不同的数据集进行了多次的曲线拟合，左边表示方差，右边表示偏差，绿色是真实值函数。**ln lambda** 表示模型的复杂程度，这个值越小，表示模型的复杂程度越高，在第一行，大家的复杂度都很低（每个人都很穷）的时候，方差是很小的，但是偏差同样很小（国家也很穷），但是到了最后一幅图，我们可以得到，每个人的复杂程度都很高的情况下，不同的函数就有着天壤之别了（贫富差异大），但是偏差就很小了（国家很富有）。

机器学习中的数学(3)-模型组合(Model Combining)之 Boosting 与 Gradient Boosting

前言：

本来上一章的结尾提到，准备写写线性分类的问题，文章都已经写得差不多了，但是突然听说最近 **Team** 准备做一套分布式的分类器，可能会使用 **Random Forest** 来做，下了几篇论文看了看，简单的 **random forest** 还比较容易弄懂，复杂一点的还会与 **boosting** 等算法结合（参见 **iccv09**），对于 **boosting** 也不甚了解，所以临时抱佛脚的看了看。说起 **boosting**，强哥之前实现过一套 **Gradient Boosting Decision Tree**（**GBDT)**算法，正好参考一下。

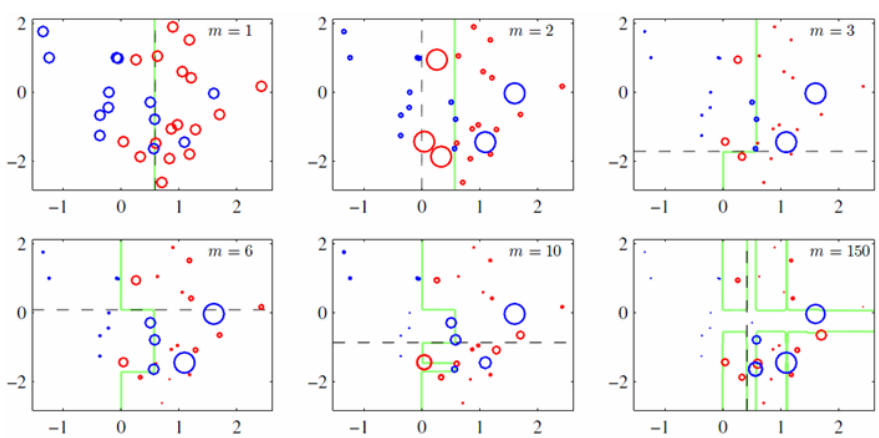
最近看的一些论文中发现了模型组合的好处，比如 **GBDT** 或者 **rf**，都是将简单的模型组合起来，效果比单个更复杂的模型好。组合的方式很多，随机化（比如 **random forest**），**Boosting**（比如 **GBDT**）都是其中典型的方法，今天主要谈谈 **Gradient Boosting** 方法（这个与传统的 **Boosting** 还有一些不同）的一些数学基础，有了这个数学基础，上面的应用可以看 **Freidman** 的 **Gradient Boosting Machine**。

本文要求读者学过基本的大学数学，另外对分类、回归等基本的机器学习概念了解。

本文主要参考资料是 **prml** 与 **Gradient Boosting Machine**。

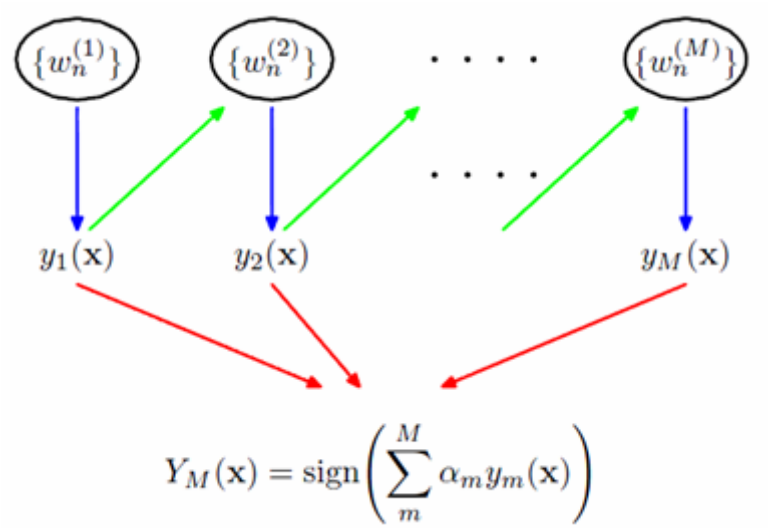
Boosting 方法：

Boosting 这其实思想相当的简单，大概是，对一份数据，建立 **M** 个模型（比如分类），一般这种模型比较简单，称为弱分类器**(weak learner)**每次分类都将上一次分错的数据权重提高一点再进行分类，这样最终得到的分类器在测试数据与训练数据上都可以得到比较好的成绩。



上图（图片来自 **prml p660**）就是一个 **Boosting** 的过程，绿色的线表示目前取得的模型（模型是由前 **m** 次得到的模型合并得到的），虚线表示当前这次模型。每次分类的时候，会更关注分错的数据，上图中，红色和蓝色的点就是数据，点越大表示权重越高，看看右下角的图片，当 **m=150** 的时候，获取的模型已经几乎能够将红色和蓝色的点区分开了。

Boosting 可以用下面的公式来表示：



训练集中一共有 **n** 个点，我们可以为里面的每一个点赋上一个权重 **Wi(0 <= i < n)**，表示这个点的重要程度，通过依次训练模型的过程，我们对点的权重进行修正，如果分类正确了，权重降低，如果分类错了，则权重提高，初始的时候，权重都是一样的。上图中绿色的线就是表示依次训练模型，可以想象得到，程序越往后执行，训练出的模型就越会在意那些容易分错（权重高）的点。当全部的程序执行完后，会得到 **M** 个模型，分别对应上图的 **y1(x)...yM(x)**，通过加权的方式组合成一个最终的模型 **YM(x)**。

我觉得 **Boosting** 更像是一个人学习的过程，开始学一样东西的时候，会去做一些习题，但是常常连一些简单的题目都会弄错，但是越到后面，简单的题目已经难不倒他了，就会去做更复杂的题目，等到他做了很多的题目后，不管是难题还是简单的题都可以解决掉了。

Gradient Boosting 方法：

其实 **Boosting** 更像是一种思想，**Gradient Boosting** 是一种 **Boosting** 的方法，它主要的思想是，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。这句话有一点拗口，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。

下面的内容就是用数学的方式来描述 **Gradient Boosting**，数学上不算太复杂，只要潜下心来看就能看懂：）

可加的参数的梯度表示：

假设我们的模型能够用下面的函数来表示，**P** 表示参数，可能有多个参数组成，**P = {p0,p1,p2....}**，**F(x;P)**表示以 **P** 为参数的 **x** 的函数，也就是我们的预测函数。我们的模型是由多个模型加起来的，**β** 表示每个模型的权重，**α** 表示模型里面的参数。为了优化 **F**，我们就可以优化**{β,α}**也就是 **P**。

$$F(x;P)=F(x;\{\beta_m,\alpha_m\}_1^M)=\sum_{m=1}^{M}\beta_m h(x;\alpha_m)$$

我们还是用 **P** 来表示模型的参数，可以得到，**Φ(P)**表示 **P** 的 **likelihood** 函数，也就是模型 **F(x;P)**的 **loss** 函数，**Φ(P)=...**后面的一块看起来很复杂，只要理解成是一个损失函数就行了，不要被吓跑了。

$$P^*=\arg\min(\Phi(P))$$
$$\Phi(P)=E_{y,x}L(y,F(x;P))$$

既然模型**(F(x;P))**是可加的，对于参数 **P**，

$$P^*=\sum_{m=0}^{M}p_m$$

我们也可以得到下面的式子：这样优化 **P** 的过程，就可以是一个梯度下降的过程了，假设当前已经得到了 **m-1** 个模型，想要得到第 **m** 个模型的时候，我们首先对前 **m-1** 个模型求梯度。得到最快下降的方向，**gm** 就是最快下降的方向。

$$g_m=\{g_{jm}\}=\{[\frac{\partial\Phi(P)}{\partial p_j}]_{p=p_{m-1}}\}$$

这里有一个很重要的假设，对于求出的前 **m-1** 个模型，我们认为是已知的了，不要去改变它，而我们的目标是放在之后的模型建立上。就像做事情的时候，之前做错的事就没有后悔药吃了，只有努力在之后的事情上别犯错：

$$P_{m-1}=\sum_{i=0}^{m-1}p_i$$

我们得到的新的模型就是，它就在 **P** 似然函数的梯度方向。**ρ** 是在梯度方向上下降的距离。

$$p_m=-\rho_m g_m$$

我们最终可以通过优化下面的式子来得到最优的 **ρ**：

$$\rho_m=\arg\min\Phi(P_{m-1}-\rho_m g_m)$$

可加的函数的梯度表示：

上面通过参数 **P** 的可加性，得到了参数 **P** 的似然函数的梯度下降的方法。我们可以将参数 **P** 的可加性推广到函数空间，我们可以得到下面的函数，此处的 **fi(x)**类似于上面的 **h(x;α)**，因为作者的文献中这样使用，我这里就用作者的表达方法：

$$F_{m-1}(x)=\sum_{i=0}^{m-1} f_i(x) \quad \text{类似于} \quad P_{m-1}=\sum_{i=0}^{m-1} p_i$$

同样，我们可以得到函数 **F(x)**的梯度下降方向 **g(x)**

$$g_m(x)=E_y\left[\frac{\partial L(y,F(x))}{F(x)}\,|\,x\right]_{F(x)=F_{m-1}(x)} \quad \text{类似于} \quad g_m=\{g_{jm}\}=\{[\frac{\partial \Phi(P)}{\partial p_j}]_{p=p_{m-1}}\}$$

最终可以得到第 **m** 个模型 **fm(x)**的表达式**:**

$$f_m(x)=-\rho_m g_m(x)$$

通用的 Gradient Descent Boosting 的框架：

下面我将推导一下 **Gradient Descent** 方法的通用形式，之前讨论过的：

$$F(x;P)=\sum_{m=1}^{M}\beta_m h(x;\alpha_m)$$

对于模型的参数**{β,α}**，我们可以用下面的式子来进行表示，这个式子的意思是，对于 **N** 个样本点**(xi,yi)**计算其在模型 **F(x;α,β)**下的损失函数，最优的**{α,β}**就是能够使得这个损失函数最小的**{α,β}**。 $\{\beta_m,\alpha_m\}_1^M$ 表示两个 **m** 维的参数：

$$\{\beta_m,\alpha_m\}_1^M=\arg\min\sum_{i=1}^{N}L(y_i,\sum_{m=1}^{M}\beta_m h(x_i;\alpha_m))$$

写成梯度下降的方式就是下面的形式，也就是我们将要得到的模型 **fm(x)**的参数**{αm,βm}**能够使得 **fm** 的方向是之前得到的模型 **Fm-1(x)**的损失函数下降最快的方向：

$$\beta_m,\alpha_m=\arg\min\sum_{i=1}^{N}L(y_i,F_{m-1}(x_i)+\beta h(x_i;\alpha))$$

对于每一个数据点 **xi** 都可以得到一个 **gm(xi)**，最终我们可以得到一个完整梯度下降方向

$$\overrightarrow{g_m}=\{-g_m(x_i)\}_1^N$$

$$-g_m(x_i)=-\left[\frac{\partial L(y_i,F(x_i))}{\partial F(x_i)}\right]_{F(x)=F_{m-1}(x)}$$

为了使得 **fm(x)**能够在 **gm(x)**的方向上，我们可以优化下面的式子得到，可以使用最小二乘法：

$$\alpha_m = \arg\min \sum_{i=1}^{N} (-g_m(x_i) - \beta h(x;\alpha))^2$$

得到了 **α** 的基础上，然后可以得到 **βm**。

$$\beta_m = \arg\min \sum_{i=1}^{N} L(y_i, F_{m-1}(x_i) + \beta h(x_i;\alpha_m))$$

最终合并到模型中：

$$F_m(x) = F_{m-1}(x) + \rho_m h(x;\alpha_m)$$

算法的流程图如下

Algorithm 1: Gradient_Boost

1. $F_0(\mathbf{x}) = \arg\min_\rho \sum_{i=1}^{N} L(y_i, \rho)$
2. For $m = 1$ to M do:
3. $\tilde{y}_i = -\left[\frac{\partial L(y_i, F(\mathbf{x}_i))}{\partial F(\mathbf{x}_i)}\right]_{F(\mathbf{x})=F_{m-1}(\mathbf{x})}, \; i = 1, N$
4. $\mathbf{a}_m = \arg\min_{\mathbf{a},\beta} \sum_{i=1}^{N} [\tilde{y}_i - \beta h(\mathbf{x}_i;\mathbf{a})]^2$
5. $\rho_m = \arg\min_\rho \sum_{i=1}^{N} L(y_i, F_{m-1}(\mathbf{x}_i) + \rho h(\mathbf{x}_i;\mathbf{a}_m))$
6. $F_m(\mathbf{x}) = F_{m-1}(\mathbf{x}) + \rho_m h(\mathbf{x};\mathbf{a}_m)$
7. endFor

end Algorithm

之后，作者还说了这个算法在其他的地方的推广，其中，**Multi-class logistic regression and classification** 就是 **GBDT** 的一种实现，可以看看，流程图跟上面的算法类似的。这里不打算继续写下去，再写下去就成论文翻译了，请参考文章：**Greedy function Approximation – A Gradient Boosting Machine**，作者 **Freidman**。

总结：

本文主要谈了谈 **Boosting** 与 **Gradient Boosting** 的方法，**Boosting** 主要是一种思想，表示"知错就改"。而 **Gradient Boosting** 是在这个思想下的一种函数（也可以说是模型）的优化的方法，首先将函数分解为可加的形式（其实所有的函数都是可加的，只是是否好放在这个框架中，以及最终的效果如何）。然后进行 **m** 次迭代，通过使得损失函数在梯度方向上减少，最终得到一个优秀的模型。值得一提的是，每次模型在梯度方向上的减少的部分，可以认为是一个"小"的或者"弱"的模型，最终我们会通过加权(也就是每次在梯度方向上下降的距离）的方式将这些"弱"的模型合并起来，形成一个更好的模型。

有了这个 **Gradient Descent** 这个基础，还可以做很多的事情。也在机器学习的道路上更进一步了：）

机器学习中的数学(4)-线性判别分析（LDA），主成分分析(PCA)

前言：

第二篇的文章中谈到，和部门老大一宁出去 **outing** 的时候，他给了我相当多的机器学习的建议，里面涉及到很多的算法的意义、学习方法等等。一宁上次给我提到，如果学习分类算法，最好从线性的入手，线性分类器最简单的就是 **LDA**，它可以看做是简化版的 **SVM**，如果想理解 **SVM** 这种分类器，那理解 **LDA** 就是很有必要的了。

谈到 **LDA**，就不得不谈谈 **PCA**，**PCA** 是一个和 **LDA** 非常相关的算法，从推导、求解、到算法最终的结果，都有着相当的相似。

本次的内容主要是以推导数学公式为主，都是从算法的物理意义出发，然后一步一步最终推导到最终的式子，**LDA** 和 **PCA** 最终的表现都是解一个矩阵特征值的问题，但是理解了如何推导，才能更深刻的理解其中的含义。本次内容要求读者有一些基本的线性代数基础，比如说特征值、特征向量的概念，空间投影，点乘等的一些基本知识等。除此之外的其他公式、我都尽量讲得更简单清楚。

LDA：

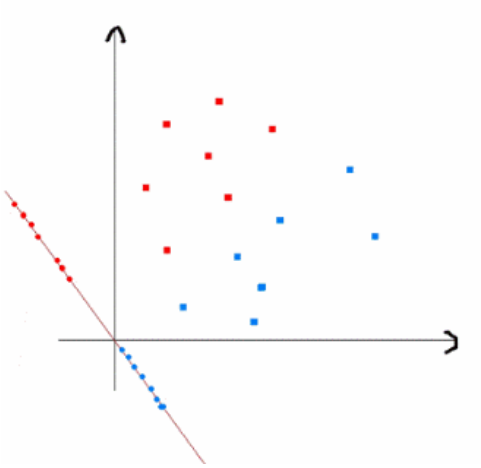
LDA 的全称是 **Linear Discriminant Analysis**（线性判别分析），是一种 **supervised learning**。有些资料上也称为是 **Fisher's Linear Discriminant**，因为它被 **Ronald Fisher** 发明自 **1936** 年，**Discriminant** 这次词我个人的理解是，一个模型，不需要去通过概率的方法来训练、预测数据，比如说各种贝叶斯方法，就需要获取数据的先验、后验概率等等。**LDA** 是在目前机器学习、数据挖掘领域经典且热门的一个算法，据我所知，百度的商务搜索部里面就用了不少这方面的算法。

LDA 的原理是，将带上标签的数据（点），通过投影的方法，投影到维度更低的空间中，使得投影后的点，会形成按类别区分，一簇一簇的情况，相同类别的点，将会在投影后的空间中更接近。要说明白 **LDA**，首先得弄明白线性分类器(**Linear Classifier**)：因为 **LDA** 是一种线性分类器。对于 **K-**分类的一个分类问题，会有 **K** 个线性函数：

$$y_k(x) = w_k^T x + w_{k0}$$

当满足条件：对于所有的 **j**，都有 **Yk > Yj,**的时候，我们就说 **x** 属于类别 **k**。对于每一个分类，都有一个公式去算一个分值，在所有的公式得到的分值中，找一个最大的，就是所属的分类了。

上式实际上就是一种投影，是将一个高维的点投影到一条高维的直线上，**LDA** 最求的目标是，给出一个标注了类别的数据集，投影到了一条直线之后，能够使得点尽量的按类别区分开，当 **k=2** 即二分类问题的时候，如下图所示：



红色的方形的点为 **0** 类的原始点、蓝色的方形点为 **1** 类的原始点，经过原点的那条线就是投影的直线，从图上可以清楚的看到，红色的点和蓝色的点被原点明显的分开了，这个数据只是随便画的，如果在高维的情况下，看起来会更好一点。下面我来推导一下二分类 **LDA** 问题的公式：

假设用来区分二分类的直线（投影函数)为：

$$y = w^T x$$

LDA 分类的一个目标是使得不同类别之间的距离越远越好，同一类别之中的距离越近越好，所以我们需要定义几个关键的值。

类别 **i** 的原始中心点为：（**Di** 表示属于类别 **i** 的点)

$$m_i = \frac{1}{n_i} \sum_{x \in D_i} x$$

类别 **i** 投影后的中心点为：

$$\tilde{m}_i = w^T m_i$$

衡量类别 **i** 投影后，类别点之间的分散程度（方差）为：

$$\tilde{s}_i = \sum_{y \in Y_i} (y - \tilde{m}_i)^2$$

最终我们可以得到一个下面的公式，表示 **LDA** 投影到 **w** 后的损失函数：

$$J(w) = \frac{|\tilde{m}_1 - \tilde{m}_2|^2}{\tilde{s}_1^2 + \tilde{s}_2^2}$$

我们分类的目标是，使得类别内的点距离越近越好（集中），类别间的点越远越好。分母表示每一个类别内的方差之和，方差越大表示一个类别内的点越分散，分子为两个类别各自的中心点的距离的平方，我们最大化 **J(w)**就可以求出最优的 **w** 了。想要求出最优的 **w**，可以使用拉格朗日乘子法，但是现在我们得到的 **J(w)**里面，**w** 是不能被单独提出来的，我们就得想办法将 **w** 单独提出来。

我们定义一个投影前的各类别分散程度的矩阵，这个矩阵看起来有一点麻烦，其实意思是，如果某一个分类的输入点集 **Di** 里面的点距离这个分类的中心店 **mi** 越近，则 **Si** 里面元素的值就越小，如果分类的点都紧紧地围绕着 **mi**，则 **Si** 里面的元素值越更接近 **0.**

$$S_i = \sum_{x \in D_i} (x - m_i)(x - m_i)^T$$

带入 **Si**，将 **J(w)**分母化为：

$$\tilde{s}_i = \sum_{x \in D_i} (w^T x - w^T m_i)^2 = \sum_{x \in D_i} w^T (x - m_i)(x - m_i)^T w = w^T S_i w$$

$$\tilde{s}_1^2 + \tilde{s}_2^2 = w^T (S_1 + S_2) w = w^T S_w w$$

同样的将 **J(w)**分子化为：

$$|\tilde{m}_1 - \tilde{m}_2|^2 = w^T (m_1 - m_2)(m_1 - m_2)^T w = w^T S_B w$$

这样损失函数可以化成下面的形式：

$$J(w) = \frac{w^T S_B w}{w^T S_w w}$$

这样就可以用最喜欢的拉格朗日乘子法了，但是还有一个问题，如果分子、分母是都可以取任意值的，那就会使得有无穷解，我们将分母限制为长度为 **1**（这是用拉格朗日乘子法一个很重要的技巧，在下面将说的 **PCA** 里面也会用到，如果忘记了，请复习一下高数），并作为拉格朗日乘子法的限制条件，带入得到：

$$c(w) = w^T S_B w - \lambda (w^T S_w w - 1)$$

$$\Rightarrow \frac{dc}{dw} = 2 S_B w - 2 \lambda S_w w = 0$$

$$\Rightarrow S_B w = \lambda S_w w$$

这样的式子就是一个求特征值的问题了。

对于 **N(N>2)**分类的问题，我就直接写出下面的结论了：

$$S_W = \sum_{i=1}^{c} S_i$$

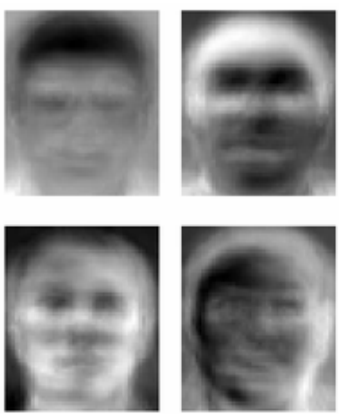
$$S_B = \sum_{i=1}^{c} n_i (m_i - m)(m_i - m)^T$$

$$S_B w_i = \lambda S_w w_i$$

这同样是一个求特征值的问题，我们求出的第 **i** 大的特征向量，就是对应的 **Wi** 了。

这里想多谈谈特征值，特征值在纯数学、量子力学、固体力学、计算机等等领域都有广泛的应用，特征值表示的是矩阵的性质，当我们取到矩阵的前 **N** 个最大的特征值的时候，我们可以说提取到的矩阵主要的成分（这个和之后的 **PCA** 相关，但是不是完全一样的概念）。在机器学习领域，不少的地方都要用到特征值的计算，比如说图像识别、**pagerank**、**LDA**、还有之后将会提到的 **PCA** 等等。

下图是图像识别中广泛用到的特征脸（**eigen face**），提取出特征脸有两个目的，首先是为了压缩数据，对于一张图片，只需要保存其最重要的部分就是了，然后是为了使得程序更容易处理，在提取主要特征的时候，很多的噪声都被过滤掉了。跟下面将谈到的 **PCA** 的作用非常相关。



特征值的求法有很多，求一个 **D * D** 的矩阵的时间复杂度是 **O(D^3),** 也有一些求 **Top M** 的方法，比如说 **power method**，它的时间复杂度是 **O(D^2 * M),** 总体来说，求特征值是一个很费时间的操作，如果是单机环境下，是很局限的。

PCA：

主成分分析（**PCA**）与 **LDA** 有着非常近似的意思，**LDA** 的输入数据是带标签的，而 **PCA** 的输入数据是不带标签的，所以 **PCA** 是一种 **unsupervised learning**。**LDA** 通常来说是作为一个独立的算法存在，给定了训练数据后，将会得到一系列的判别函数（**discriminate function**），之后对于新的输入，就可以进行预测了。而 **PCA** 更像是一个预处理的方法，它可以将原本的数据降低维度，而使得降低了维度的数据之间的方差最大（也可以说投影误差最小，具体在之后的推导里面会谈到）。

方差这个东西是个很有趣的，有些时候我们会考虑减少方差（比如说训练模型的时候，我们会考虑到方差-偏差的均衡），有的时候我们会尽量的增大方差。方差就像是一种信仰（强哥的话），不一定会有很严密的证明，从实践来说，通过尽量增大投影方差的 **PCA** 算法，确实可以提高我们的算法质量。

说了这么多，推推公式可以帮助我们理解。我下面将用两种思路来推导出一个同样的表达式。首先是最大化投影后的方差，其次是最小化投影后的损失（投影产生的损失最小）。

最大化方差法：

假设我们还是将一个空间中的点投影到一个向量中去。首先，给出原空间的中心点：

$$\bar{x} = \frac{1}{N}\sum_{n=1}^{N} x_n$$

假设 **u1** 为投影向量，投影之后的方差为：

$$\frac{1}{N}\sum_{n=1}^{N}\{u_1^T x_n - u_1^T \bar{x}\}^2 = u_1^T S u_1$$

上面这个式子如果看懂了之前推导 **LDA** 的过程，应该比较容易理解，如果线性代数里面的内容忘记了，可以再温习一下，优化上式等号右边的内容，还是用拉格朗日乘子法：

$$u_1^T S u_1 + \lambda_1(1 - u_1^T u_1)$$

将上式求导，使之为 **0**，得到：

$$S u_1 = \lambda_1 u_1$$

这是一个标准的特征值表达式了，**λ** 对应的特征值，**u** 对应的特征向量。上式的左边取得最大值的条件就是 **λ1** 最大，也就是取得最大的特征值的时候。假设我们是要将一个 **D** 维的数据空间投影到 **M** 维的数据空间中（**M < D)**， 那我们取前 **M** 个特征向量构成的投影矩阵就是能够使得方差最大的矩阵了。

最小化损失法：

假设输入数据 **x** 是在 **D** 维空间中的点，那么，我们可以用 **D** 个正交的 **D** 维向量去完全的表示这个空间（这个空间中所有的向量都可以用这 **D** 个向量的线性组合得到）。在 **D** 维空间中，有无穷多种可能找这 **D** 个正交的 **D** 维向量，哪个组合是最合适的呢？

假设我们已经找到了这 **D** 个向量，可以得到：

$$x_n = \sum_{i=1}^{D} \alpha_{ni} u_i$$

我们可以用近似法来表示投影后的点：

$$\tilde{x}_n = \sum_{i=1}^{M} z_{ni} u_i + \sum_{i=M+1}^{D} b_i u_i$$

上式表示，得到的新的 **x** 是由前 **M** 个基的线性组合加上后 **D - M** 个基的线性组合，注意这里的 **z** 是对于每个 **x** 都不同的，而 **b** 对于每个 **x** 是相同的，这样我们就可以用 **M** 个数来表示空间中的一个点，也就是使得数据降维了。但是这样降维后的数据，必然会产生一些扭曲，我们用 **J** 描述这种扭曲，我们的目标是，使得 **J** 最小：

$$J = \frac{1}{N}\sum_{n=1}^{N} \| x_n - \tilde{x}_n \|^2$$

上式的意思很直观，就是对于每一个点，将降维后的点与原始的点之间的距离的平方和加起来，求平均值，我们就要使得这个平均值最小。我们令：

$$\frac{\partial J}{\partial z_{nj}} = 0 \Rightarrow z_{nj} = x_n^T u_j, \quad \frac{\partial J}{\partial b_j} = 0 \Rightarrow b_j = \bar{x}^T u_j$$

将上面得到的 **z** 与 **b** 带入降维的表达式：

$$x_n - \tilde{x}_n = \sum_{i=M+1}^{D} \{(x_n - \bar{x})u_i\}u_i$$

将上式带入 **J** 的表达式得到：

$$J = \frac{1}{N}\sum_{n=1}^{N}\sum_{i=M+1}^{D}(x_n^T u_i - \bar{x}^T u_i)^2 = \sum_{i=M+1}^{D} u_i^T S u_i$$

再用上拉普拉斯乘子法（此处略），可以得到，取得我们想要的投影基的表达式为：

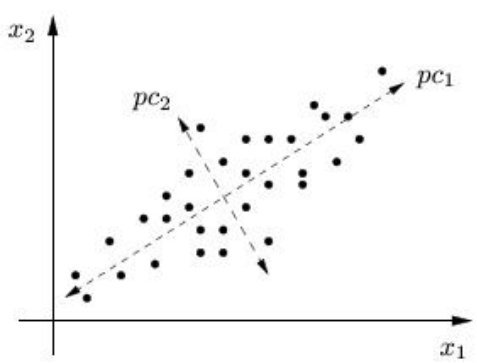
$$Su_i = \lambda_i u_i$$

这里又是一个特征值的表达式，我们想要的前 **M** 个向量其实就是这里最大的 **M** 个特征值所对应的特征向量。证明这个还可以看看，我们 **J** 可以化为：

$$J = \sum_{i=M+1}^{D} \lambda_i$$

也就是当误差 **J** 是由最小的 **D – M** 个特征值组成的时候，**J** 取得最小值。跟上面的意思相同。

下图是 **PCA** 的投影的一个表示，黑色的点是原始的点，带箭头的虚线是投影的向量，**Pc1** 表示特征值最大的特征向量，**pc2** 表示特征值次大的特征向量，两者是彼此正交的，因为这原本是一个 **2** 维的空间，所以最多有两个投影的向量，如果空间维度更高，则投影的向量会更多。



总结：

本次主要讲了两种方法，**PCA** 与 **LDA**，两者的思想和计算方法非常类似，但是一个是作为独立的算法存在，另一个更多的用于数据的预处理的工作。另外对于 **PCA** 和 **LDA** 还有核方法，本次的篇幅比较大了，先不说了，以后有时间再谈：

机器学习中的数学(5)-强大的矩阵奇异值分解(SVD)及其应用

前言：

上一次写了关于 **PCA 与 LDA** 的文章，**PCA** 的实现一般有两种，一种是用特征值分解去实现的，一种是用奇异值分解去实现的。在上篇文章中便是基于特征值分解的一种解释。特征值和奇异值在大部分人的印象中，往往是停留在纯粹的数学计算中。而且线性代数或者矩阵论里面，也很少讲任何跟特征值与奇异值有关的应用背景。奇异值分解是一个有着很明显的物理意义的一种方法，它可以将一个比较复杂的矩阵用更小更简单的几个子矩阵的相乘来表示，这些小矩阵描述的是矩阵的重要的特性。就像是描述一个人一样，给别人描述说这个人长得浓眉大眼，方脸，络腮胡，而且带个黑框的眼镜，这样寥寥的几个特征，就让别人脑海里面就有一个较为清楚的认识，实际上，人脸上的特征是有着无数种的，之所以能这么描述，是因为人天生就有着非常好的抽取重要特征的能力，让机器学会抽取重要的特征，**SVD** 是一个重要的方法。

在机器学习领域，有相当多的应用与奇异值都可以扯上关系，比如做 **feature reduction** 的 **PCA**，做数据压缩（以图像压缩为代表）的算法，还有做搜索引擎语义层次检索的 **LSI**（**Latent Semantic Indexing**）

另外在这里抱怨一下，之前在百度里面搜索过 **SVD**，出来的结果都是俄罗斯的一种狙击枪（**AK47** 同时代的），是因为穿越火线这个游戏里面有一把狙击枪叫做 **SVD**，而在 **Google** 上面搜索的时候，出来的都是奇异值分解（英文资料为主）。想玩玩战争游戏，玩玩 **COD** 不是非常好吗，玩山寨的 **CS** 有神马意思啊。国内的网页中的话语权也被这些没有太多营养的帖子所占据。真心希望国内的气氛能够更浓一点，搞游戏的人真正是喜欢制作游戏，搞 **Data Mining** 的人是真正喜欢挖数据的，都不是仅仅为了混口饭吃，这样谈超越别人才有意义，中文文章中，能踏踏实实谈谈技术的太少了，改变这个状况，从我自己做起吧。

前面说了这么多，本文主要关注奇异值的一些特性，另外还会稍稍提及奇异值的计算，不过本文不准备在如何计算奇异值上展开太多。另外，本文里面有部分不算太深的线性代数的知识，如果完全忘记了线性代数，看本文可能会有些困难。

一、奇异值与特征值基础知识：

特征值分解和奇异值分解在机器学习领域都是属于满地可见的方法。两者有着很紧密的关系，我在接下来会谈到，特征值分解和奇异值分解的目的都是一样，就是提取出一个矩阵最重要的特征。先谈谈特征值分解吧：

1）特征值：

如果说一个向量 **v** 是方阵 **A** 的特征向量，将一定可以表示成下面的形式：

$$Av = \lambda v$$

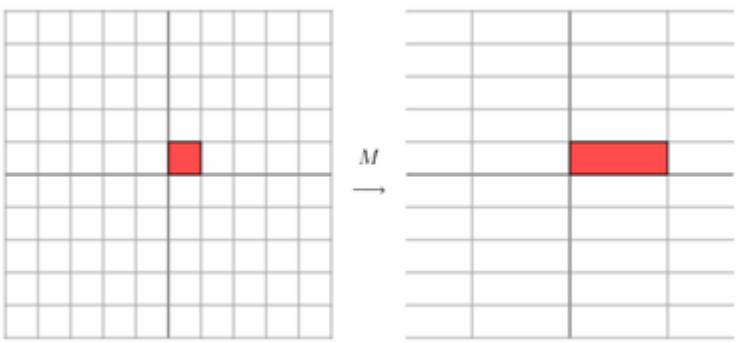
这时候 **λ** 就被称为特征向量 **v** 对应的特征值，一个矩阵的一组特征向量是一组正交向量。特征值分解是将一个矩阵分解成下面的形式：

$$A = Q\Sigma Q^{-1}$$

其中 **Q** 是这个矩阵 **A** 的特征向量组成的矩阵，**Σ** 是一个对角阵，每一个对角线上的元素就是一个特征值。我这里引用了一些参考文献中的内容来说明一下。首先，要明确的是，一个矩阵其实就是一个线性变换，因为一个矩阵乘以一个向量后得到的向量，其实就相当于将这个向量进行了线性变换。比如说下面的一个矩阵：

$$M = \begin{bmatrix} 3 & 0 \\ 0 & 1 \end{bmatrix}$$

它其实对应的线性变换是下面的形式：



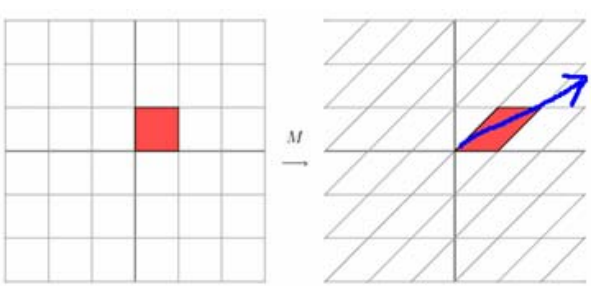
因为这个矩阵 **M** 乘以一个向量**(x,y)**的结果是：

$$\begin{bmatrix} 3 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} 3x \\ y \end{bmatrix}$$

上面的矩阵是对称的，所以这个变换是一个对 **x**，**y** 轴的方向一个拉伸变换（每一个对角线上的元素将会对一个维度进行拉伸变换，当值**>1** 时，是拉长，当值**<1** 时时缩短），当矩阵不是对称的时候，假如说矩阵是下面的样子：

$$M = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix}$$

它所描述的变换是下面的样子：



这其实是在平面上对一个轴进行的拉伸变换（如蓝色的箭头所示），在图中，蓝色的箭头是一个最主要的变化方向（变化方向可能有不止一个），如果我们想要描述好一个变换，那我们就描述好这个变换主要的变化方向就好了。反过头来看看之前特征值分解的式子，分解得到的 **Σ** 矩阵是一个对角阵，里面的特征值是由大到小排列的，这些特征值所对应的特征向量就是描述这个矩阵变化方向（从主要的变化到次要的变化排列）

当矩阵是高维的情况下，那么这个矩阵就是高维空间下的一个线性变换，这个线性变化可能没法通过图片来表示，但是可以想象，这个变换也同样有很多的变换方向，我们通过特征值分解得到的前 **N** 个特征向量，那么就对应了这个矩阵最主要的 **N** 个变化方向。我们利用这前 **N** 个变化方向，就可以近似这个矩阵（变换）。也就是之前说的：提取这个矩阵最重要的特征。总结一下，特征值分解可以得到特征值与特征向量，特征值表示的是这个特征到底有多重要，而特征向量表示这个特征是什么，可以将每一个特征向量理解为一个线性的子空间，我们可以利用这些线性的子空间干很多的事情。不过，特征值分解也有很多的局限，比如说变换的矩阵必须是方阵。

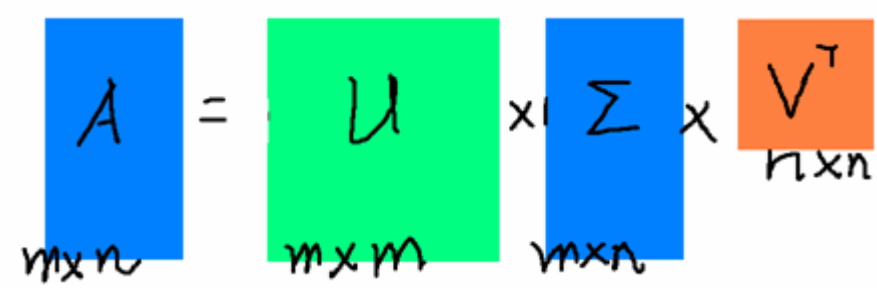
（说了这么多特征值变换，不知道有没有说清楚，请各位多提提意见。）

2）奇异值：

下面谈谈奇异值分解。特征值分解是一个提取矩阵特征很不错的方法，但是它只是对方阵而言的，在现实的世界中，我们看到的大部分矩阵都不是方阵，比如说有 **N** 个学生，每个学生有 **M** 科成绩，这样形成的一个 **N * M** 的矩阵就不可能是方阵，我们怎样才能描述这样普通的矩阵呢的重要特征呢？奇异值分解可以用来干这个事情，奇异值分解是一个能适用于任意的矩阵的一种分解的方法：

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

假设 **A** 是一个 **N * M** 的矩阵，那么得到的 **U** 是一个 **N * N** 的方阵（里面的向量是正交的，**U** 里面的向量称为左奇异向量），**Σ** 是一个 **N * M** 的矩阵（除了对角线的元素都是 **0**，对角线上的元素称为奇异值），**V'(V** 的转置**)**是一个 **N * N** 的矩阵，里面的向量也是正交的，**V** 里面的向量称为右奇异向量），从图片来反映几个相乘的矩阵的大小可得下面的图片



那么奇异值和特征值是怎么对应起来的呢？首先，我们将一个矩阵 **A** 的转置 * **A**，将会得到一个方阵，我们用这个方阵求特征值可以得到：

$$(A^T A)v_i = \lambda_i v_i$$

这里得到的 **v**，就是我们上面的右奇异向量。此外我们还可以得到：

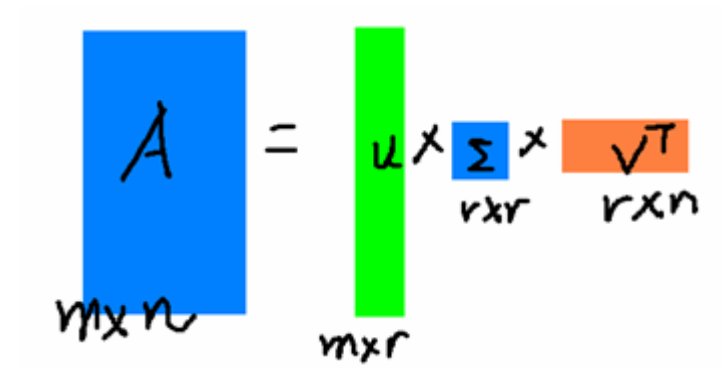
$$\sigma_i = \sqrt{\lambda_i}$$

$$u_i = \frac{1}{\sigma_i} A v_i$$

这里的 **σ** 就是上面说的奇异值，**u** 就是上面说的左奇异向量。奇异值 **σ** 跟特征值类似，在矩阵 **Σ** 中也是从大到小排列，而且 **σ** 的减少特别的快，在很多情况下，前 **10%**甚至 **1%**的奇异值的和就占了全部的奇异值之和的 **99%**以上了。也就是说，我们也可以用前 **r** 大的奇异值来近似描述矩阵，这里定义一下部分奇异值分解：

$$A_{m\times n} \approx U_{m\times r}\Sigma_{r\times r}V^T_{r\times n}$$

r 是一个远小于 **m**、**n** 的数，这样矩阵的乘法看起来像是下面的样子：



右边的三个矩阵相乘的结果将会是一个接近于 **A** 的矩阵，在这儿，**r** 越接近于 **n**，则相乘的结果越接近于 **A**。而这三个矩阵的面积之和（在存储观点来说，矩阵面积越小，存储量就越小）要远远小于原始的矩阵 **A**，我们如果想要压缩空间来表示原矩阵 **A**，我们存下这里的三个矩阵：**U**、**Σ**、**V** 就好了。

二、奇异值的计算：

奇异值的计算是一个难题，是一个 **O(N^3)**的算法。在单机的情况下当然是没问题的，**matlab** 在一秒钟内就可以算出 **1000 * 1000** 的矩阵的所有奇异值，但是当矩阵的规模增长的时候，计算的复杂度呈 **3** 次方增长，就需要并行计算参与了。**Google** 的吴军老师在数学之美系列谈到 **SVD** 的时候，说起 **Google** 实现了 **SVD** 的并行化算法，说这是对人类的一个贡献，但是也没有给出具体的计算规模，也没有给出太多有价值的信息。

其实 **SVD** 还是可以用并行的方式去实现的，在解大规模的矩阵的时候，一般使用迭代的方法，当矩阵的规模很大（比如说上亿）的时候，迭代的次数也

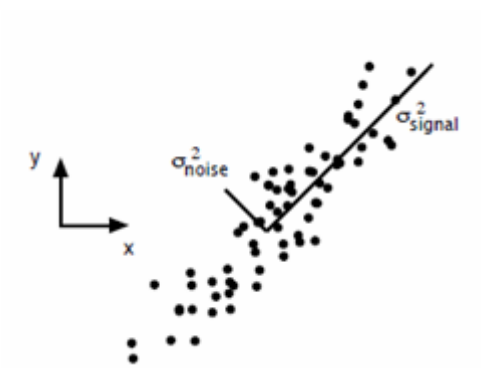
可能会上亿次，如果使用 **Map-Reduce** 框架去解，则每次 **Map-Reduce** 完成的时候，都会涉及到写文件、读文件的操作。个人猜测 **Google** 云计算体系中除了 **Map-Reduce** 以外应该还有类似于 **MPI** 的计算模型，也就是节点之间是保持通信，数据是常驻在内存中的，这种计算模型比 **Map-Reduce** 在解决迭代次数非常多的时候，要快了很多倍。

Lanczos 迭代就是一种解对称方阵部分特征值的方法（之前谈到了，解 **A'* A** 得到的对称方阵的特征值就是解 **A** 的右奇异向量），是将一个对称的方程化为一个三对角矩阵再进行求解。按网上的一些文献来看，**Google** 应该是用这种方法去做的奇异值分解的。请见 **Wikipedia** 上面的一些引用的论文，如果理解了那些论文，也"几乎"可以做出一个 **SVD** 了。

由于奇异值的计算是一个很枯燥，纯数学的过程，而且前人的研究成果（论文中）几乎已经把整个程序的流程图给出来了。更多的关于奇异值计算的部分，将在后面的参考文献中给出，这里不再深入，我还是 **focus** 在奇异值的应用中去。

三、奇异值与主成分分析（PCA）：

主成分分析在上一节里面也讲了一些，这里主要谈谈如何用 **SVD** 去解 **PCA** 的问题。**PCA** 的问题其实是一个基的变换，使得变换后的数据有着最大的方差。方差的大小描述的是一个变量的信息量，我们在讲一个东西的稳定性的时候，往往说要减小方差，如果一个模型的方差很大，那就说明模型不稳定了。但是对于我们用于机器学习的数据（主要是训练数据），方差大才有意义，不然输入的数据都是同一个点，那方差就为 **0** 了，这样输入的多个数据就等同于一个数据了。以下面这张图为例子：



这个假设是一个摄像机采集一个物体运动得到的图片，上面的点表示物体运动的位置，假如我们想要用一条直线去拟合这些点，那我们会选择什么方向的线呢？当然是图上标有 **signal** 的那条线。如果我们把这些点单纯的投影到 **x** 轴或者 **y** 轴上，最后在 **x** 轴与 **y** 轴上得到的方差是相似的（因为这些点的趋势是在 **45** 度左右的方向，所以投影到 **x** 轴或者 **y** 轴上都是类似的），如果我们使用原来的 **xy** 坐标系去看这些点，容易看不出来这些点真正的方向是什么。但是如果我们进行坐标系的变化，横轴变成了 **signal** 的方向，纵轴变成了 **noise** 的方向，则就很容易发现什么方向的方差大，什么方向的方差小了。

一般来说，方差大的方向是信号的方向，方差小的方向是噪声的方向，我们在数据挖掘中或者数字信号处理中，往往要提高信号与噪声的比例，也就是信

噪比。对上图来说，如果我们只保留 **signal** 方向的数据，也可以对原数据进行不错的近似了。

PCA 的全部工作简单点说，就是对原始的空间中顺序地找一组相互正交的坐标轴，第一个轴是使得方差最大的，第二个轴是在与第一个轴正交的平面中使得方差最大的，第三个轴是在与第 **1**、**2** 个轴正交的平面中方差最大的，这样假设在 **N** 维空间中，我们可以找到 **N** 个这样的坐标轴，我们取前 **r** 个去近似这个空间，这样就从一个 **N** 维的空间压缩到 **r** 维的空间了，但是我们选择的 **r** 个坐标轴能够使得空间的压缩使得数据的损失最小。

还是假设我们矩阵每一行表示一个样本，每一列表示一个 **feature**，用矩阵的语言来表示，将一个 **m * n** 的矩阵 **A** 的进行坐标轴的变化，**P** 就是一个变换的矩阵从一个 **N** 维的空间变换到另一个 **N** 维的空间，在空间中就会进行一些类似于旋转、拉伸的变化。

$$A_{m\times n}P_{n\times n}=\widetilde{A}_{m\times n}$$

而将一个 **m * n** 的矩阵 **A** 变换成一个 **m * r** 的矩阵，这样就会使得本来有 **n** 个 **feature** 的，变成了有 **r** 个 **feature** 了（**r < n**），这 **r** 个其实就是对 **n** 个 **feature** 的一种提炼，我们就把这个称为 **feature** 的压缩。用数学语言表示就是：

$$A_{m\times n}P_{n\times r}=\widetilde{A}_{m\times r}$$

但是这个怎么和 **SVD** 扯上关系呢？之前谈到，**SVD** 得出的奇异向量也是从奇异值由大到小排列的，按 **PCA** 的观点来看，就是方差最大的坐标轴就是第一个奇异向量，方差次大的坐标轴就是第二个奇异向量...我们回忆一下之前得到的 **SVD** 式子：

$$A_{m\times n}\approx U_{m\times r}\Sigma_{r\times r}V^{T}{}_{r\times n}$$

在矩阵的两边同时乘上一个矩阵 **V**，由于 **V** 是一个正交的矩阵，所以 **V** 转置乘以 **V** 得到单位阵 **I**，所以可以化成后面的式子

$$A_{m\times n}V_{r\times n}\approx U_{m\times r}\Sigma_{r\times r}V^{T}{}_{r\times n}V_{r\times n}$$

$$A_{m\times n}V_{r\times n}\approx U_{m\times r}\Sigma_{r\times r}$$

将后面的式子与 **A * P** 那个 **m * n** 的矩阵变换为 **m * r** 的矩阵的式子对照看看，在这里，其实 **V** 就是 **P**，也就是一个变化的向量。这里是将一个 **m * n** 的矩阵压缩到一个 **m * r** 的矩阵，也就是对列进行压缩，如果我们想对行进行压缩（在 **PCA** 的观点下，对行进行压缩可以理解为，将一些相似的 **sample** 合并在一起，或者将一些没有太大价值的 **sample** 去掉）怎么办呢？同样我们写出一个通用的行压缩例子：

$$P_{r\times m}A_{m\times n}=\widetilde{A}_{r\times n}$$ 这样就从一个 **m** 行的矩阵压缩到一个 **r** 行的矩阵了，对 **SVD** 来说也是一样的，我们对 **SVD** 分解的式子两边乘以 **U** 的转置 **U'**

$$U_{r\times m}{}^{T}A_{m\times n}\approx\Sigma_{r\times r}V^{T}{}_{r\times n}$$ 这样我们就得到了对行进行压缩的式子。可以看出，其实 **PCA** 几乎可以说是对 **SVD** 的一个包装，如果我们实现了 **SVD**，那也就实现了 **PCA** 了，而且更好的地方是，有了 **SVD**，我们就可以得到两个方向的 **PCA**，如果我们对 **A'A** 进行特征值的分解，只能得到一个方向的 **PCA**。

四、奇异值与潜在语义索引 LSI：

潜在语义索引（**Latent Semantic Indexing**）与 **PCA** 不太一样，至少不是实现了 **SVD** 就可以直接用的，不过 **LSI** 也是一个严重依赖于 **SVD** 的算法，之前吴军老师在矩阵计算与文本处理中的分类问题中谈到：

*“三个矩阵有非常清楚的物理含义。第一个矩阵 **X** 中的每一行表示意思相关的一类词，其中的每个非零元素表示这类词中每个词的重要性（或者说相关性），数值越大越相关。最后一个矩阵 **Y** 中的每一列表示同一主题一类文章，其中每个元素表示这类文章中每篇文章的相关性。中间的矩阵则表示类词和文章雷之间的相关性。因此，我们只要对关联矩阵 **A** 进行一次奇异值分解，**w** 我们就可以同时完成了近义词分类和文章的分类。（同时得到每类文章和每类词的相关性）。”*

上面这段话可能不太容易理解，不过这就是 **LSI** 的精髓内容，我下面举一个例子来说明一下，下面的例子来自 **LSA tutorial**，具体的网址我将在最后的引用中给出：

Index Words	Titles								
	T1	T2	T3	T4	T5	T6	T7	T8	T9
book			1	1					
dads						1			1
dummies		1						1	
estate							1		1
guide	1					1			
investing	1	1	1	1	1	1	1	1	1
market	1		1						
real							1		1
rich						2			1
stock	1		1					1	
value				1	1				

这就是一个矩阵，不过不太一样的是，这里的一行表示一个词在哪些 **title** 中出现了（一行就是之前说的一维 **feature**），一列表示一个 **title** 中有哪些词，（这个矩阵其实是我们之前说的那种一行是一个 **sample** 的形式的一种转置，这个会使得我们的左右奇异向量的意义产生变化，但是不会影响我们计算的过程）。比如说 **T1** 这个 **title** 中就

有 **guide**、**investing**、**market**、**stock** 四个词，各出现了一次，我们将这个矩阵进行 **SVD**，得到下面的矩阵：

book	0.15	-0.27	0.04
dads	0.24	0.38	-0.09
dummies	0.13	-0.17	0.07
estate	0.18	0.19	0.45
guide	0.22	0.09	-0.46
investing	0.74	-0.21	0.21
market	0.18	-0.30	-0.28
real	0.18	0.19	0.45
rich	0.36	0.59	-0.34
stock	0.25	-0.42	-0.28
value	0.12	-0.14	0.23

*

3.91	0	0
0	2.61	0
0	0	2.00

*

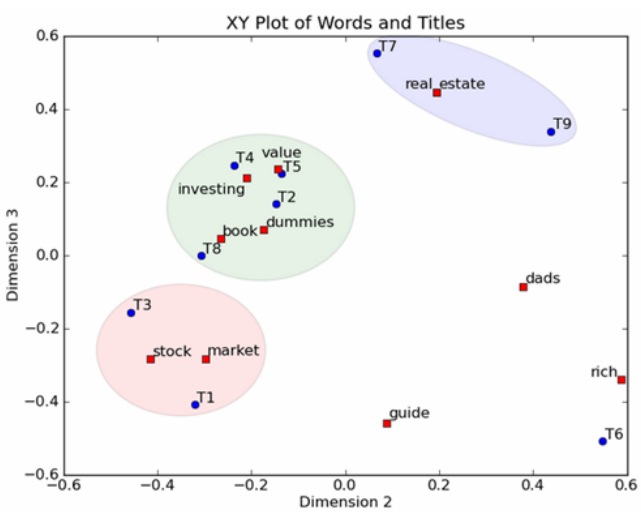
T1	T2	T3	T4	T5	T6	T7	T8	T9
0.35	0.22	0.34	0.26	0.22	0.49	0.28	0.29	0.44
-0.32	-0.15	-0.46	-0.24	-0.14	0.55	0.07	-0.31	0.44
-0.41	0.14	-0.16	0.25	0.22	-0.51	0.55	0.00	0.34

左奇异向量表示词的一些特性，右奇异向量表示文档的一些特性，中间的奇异值矩阵表示左奇异向量的一行与右奇异向量的一列的重要程序，数字越大越重要。

继续看这个矩阵还可以发现一些有意思的东西，首先，左奇异向量的第一列表示每一个词的出现频繁程度，虽然不是线性的，但是可以认为是一个大概的描述，比如 **book** 是 **0.15** 对应文档中出现的 **2** 次，**investing** 是 **0.74** 对应了文档中出现了 **9** 次，**rich** 是 **0.36** 对应文档中出现了 **3** 次；

其次，右奇异向量中一的第一行表示每一篇文档中的出现词的个数的近似，比如说，**T6** 是 **0.49**，出现了 **5** 个词，**T2** 是 **0.22**，出现了 **2** 个词。

然后我们反过头来看，我们可以将左奇异向量和右奇异向量都取后 **2** 维（之前是 **3** 维的矩阵），投影到一个平面上，可以得到：



在图上，每一个红色的点，都表示一个词，每一个蓝色的点，都表示一篇文档，这样我们可以对这些词和文档进行聚类，比如说 **stock** 和 **market** 可以放在一类，因为他们老是出现在一起，**real** 和 **estate** 可以放在一类，**dads**，**guide** 这种词就看起来有点孤立了，我们就不对他们进行合并了。按这样聚类出现的效果，可以提取文档集合中的近义词，这样当用户检索文档的时候，是用语义级别（近义词集合）去检索了，而不是之前的词的级别。这样一减少我们的检索、存储量，因为这样压缩的文档集合和 **PCA** 是异曲同工的，二可以提高我们的用户体验，用户输

入一个词，我们可以在这个词的近义词的集合中去找，这是传统的索引无法做到的。

不知道按这样描述，再看看吴军老师的文章，是不是对 **SVD** 更清楚了？**:-D**

前言：

决策树这种算法有着很多良好的特性，比如说训练时间复杂度较低，预测的过程比较快速，模型容易展示（容易将得到的决策树做成图片展示出来）等。但是同时，单决策树又有一些不好的地方，比如说容易 **over-fitting**，虽然有一些方法，如剪枝可以减少这种情况，但是还是不够的。

模型组合（比如说有 **Boosting**，**Bagging** 等）与决策树相关的算法比较多，这些算法最终的结果是生成 **N(**可能会有几百棵以上）棵树，这样可以大大的减少单决策树带来的毛病，有点类似于三个臭皮匠等于一个诸葛亮的做法，虽然这几百棵决策树中的每一棵都很简单（相对于 **C4.5** 这种单决策树来说），但是他们组合起来确是很强大。

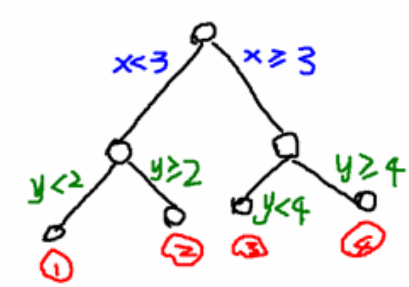
在最近几年的 **paper** 上，如 **iccv** 这种重量级的会议，**iccv 09** 年的里面有不少的文章都是与 **Boosting** 与随机森林相关的。模型组合+决策树相关的算法有两种比较基本的形式 **-** 随机森林与 **GBDT((Gradient Boost Decision Tree)**，其他的比较新的模型组合+决策树的算法都是来自这两种算法的延伸。本文主要侧重于 **GBDT**，对于随机森林只是大概提提，因为它相对比较简单。

在看本文之前，建议先看看机器学习与数学**(3)**与其中引用的论文，本文中的 **GBDT** 主要基于此，而随机森林相对比较独立。

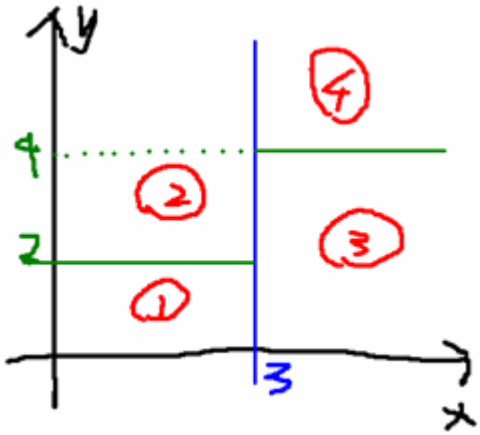
基础内容：

这里只是准备简单谈谈基础的内容，主要参考一下别人的文章，对于随机森林与 **GBDT**，有两个地方比较重要，首先是 **information gain**，其次是决策树。这里特别推荐 **Andrew Moore** 大牛的 **Decision Trees Tutorial**，与 **Information Gain Tutorial**。**Moore** 的 **Data Mining Tutorial** 系列非常赞，看懂了上面说的两个内容之后的文章才能继续读下去。

决策树实际上是将空间用超平面进行划分的一种方法，每次分割的时候，都将当前的空间一分为二，比如说下面的决策树：



就是将空间划分成下面的样子：



这样使得每一个叶子节点都是在空间中的一个不相交的区域，在进行决策的时候，会根据输入样本每一维 **feature** 的值，一步一步往下，最后使得样本落入 **N** 个区域中的一个（假设有 **N** 个叶子节点）

随机森林**(Random Forest):**

随机森林是一个最近比较火的算法，它有很多的优点：

- 在数据集上表现良好
- 在当前的很多数据集上，相对其他算法有着很大的优势
- 它能够处理很高维度（**feature** 很多）的数据，并且不用做特征选择
- 在训练完后，它能够给出哪些 **feature** 比较重要
- 在创建随机森林的时候，对 **generlization error** 使用的是无偏估计
- 训练速度快
- 在训练过程中，能够检测到 **feature** 间的互相影响
- 容易做成并行化方法
- 实现比较简单

随机森林顾名思义，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，就预测这个样本为那一类。

在建立每一棵决策树的过程中，有两点需要注意 **-** 采样与完全分裂。首先是两个随机采样的过程，**random forest** 对输入的数据要进行行、列的采样。对于行采样，采用有放回的方式，也就是在采样得到的样本集合中，可能有重复的样本。假设输入样本为 **N** 个，那么采样的样本也为 **N** 个。这样使得在训练的时候，每一棵树的输入样本都不是全部的样本，使得相对不容易出现 **over-fitting**。然后进行列采样，从 **M** 个 **feature** 中，选择 **m** 个**(m << M)**。之后就是对采样之后的数据使用完全分裂的方式建立出决策树，这样决策树的某一个叶子节点要么是无法继续分裂的，要么里面的所有样本的都是指向的同一个分类。一般很多的决策树算法都一个重要的步骤 **-** 剪枝，但是这里不这样

干，由于之前的两个随机采样的过程保证了随机性，所以就算不剪枝，也不会出现 **over-fitting**。

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。

随机森林的过程请参考 **Mahout 的 random forest** 。这个页面上写的比较清楚了，其中可能不明白的就是 **Information Gain**，可以看看之前推荐过的 **Moore** 的页面。

Gradient Boost Decision Tree:

GBDT 是一个应用很广泛的算法，可以用来做分类、回归。在很多的数据上都有不错的效果。**GBDT** 这个算法还有一些其他的名字，比如说 **MART(Multiple Additive Regression Tree)**，**GBRT(Gradient Boost Regression Tree)**，**Tree Net** 等，其实它们都是一个东西（参考自 **wikipedia – Gradient Boosting**），发明者是 **Friedman**

Gradient Boost 其实是一个框架，里面可以套入很多不同的算法，可以参考一下机器学习与数学**(3)**中的讲解。**Boost** 是"提升"的意思，一般 **Boosting** 算法都是一个迭代的过程，每一次新的训练都是为了改进上一次的结果。

原始的 **Boost** 算法是在算法开始的时候，为每一个样本赋上一个权重值，初始的时候，大家都是一样重要的。在每一步训练中得到的模型，会使得数据点的估计有对有错，我们就在每一步结束后，增加分错的点的权重，减少分对的点的权重，这样使得某些点如果老是被分错，那么就会被“严重关注”，也就被赋上一个很高的权重。然后等进行了 **N** 次迭代（由用户指定），将会得到 **N** 个简单的分类器（**basic learner**），然后我们将它们组合起来（比如说可以对它们进行加权、或者让它们进行投票等），得到一个最终的模型。

而 **Gradient Boost** 与传统的 **Boost** 的区别是，每一次的计算是为了减少上一次的残差**(residual)**，而为了消除残差，我们可以在残差减少的梯度**(Gradient)**方向上建立一个新的模型。所以说，在 **Gradient Boost** 中，每个新的模型的简历是为了使得之前模型的残差往梯度方向减少，与传统 **Boost** 对正确、错误的样本进行加权有着很大的区别。

在分类问题中，有一个很重要的内容叫做 **Multi-Class Logistic**，也就是多分类的 **Logistic** 问题，它适用于那些类别数**>2** 的问题，并且在分类结果中，样本 **x** 不是一定只属于某一个类可以得到样本 **x** 分别属于多个类的概率（也可以说样本 **x** 的估计 **y** 符合某一个几何分布），这实际上是属于 **Generalized Linear Model** 中讨论的内容，这里就先不谈了，以后有机会再用一个专门的章节去做吧。这里就用一个结论：如果一个分类问题符合几何分布，那么就可以用 **Logistic** 变换来进行之后的运算。

假设对于一个样本 **x**，它可能属于 **K** 个分类，其估计值分别为 **F1(x)...FK(x)**，**Logistic** 变换如下，**logistic** 变换是一个平滑且将数据规范化（使得向量的长度为 **1**）的过程，结果为属于类别 **k** 的概率 **pk(x)**，

$$p_k(\mathbf{x}) = \exp(F_k(\mathbf{x})) \Big/ \sum_{l=1}^{K} \exp(F_l(\mathbf{x}))$$

对于 **Logistic** 变换后的结果，损失函数为：

$$L\left(\{y_k, F_k(\mathbf{x})\}_1^K\right) = -\sum_{k=1}^{K} y_k \log p_k(\mathbf{x})$$

其中，**yk** 为输入的样本数据的估计值，当一个样本 **x** 属于类别 **k** 时，**yk = 1**，否则 **yk = 0**。

将 **Logistic** 变换的式子带入损失函数，并且对其求导，可以得到损失函数的梯度：

$$\tilde{y}_{ik} = -\left[\frac{\partial L(\{y_{il}, F_l(\mathbf{x}_i)\}_{l=1}^K)}{\partial F_k(\mathbf{x}_i)}\right]_{\{F_l(\mathbf{x})=F_{l,m-1}(\mathbf{x})\}_1^K} = y_{ik} - p_{k,m-1}(\mathbf{x}_i)$$

上面说的比较抽象，下面举个例子：

假设输入数据 **x** 可能属于 **5** 个分类（分别为 **1,2,3,4,5**），训练数据中，**x** 属于类别 **3**，则 **y = (0, 0, 1, 0, 0)**，假设模型估计得到的 **F(x) = (0, 0.3, 0.6, 0, 0)**，则经过 **Logistic** 变换后的数据 **p(x) = (0.16,0.21,0.29,0.16,0.16)**，**y - p** 得到梯度 **g**：**(-0.16, -0.21, 0.71, -0.16, -0.16)**。观察这里可以得到一个比较有意思的结论：

假设 **gk** 为样本当某一维（某一个分类）上的梯度**:**

gk>0 时，越大表示其在这一维上的概率 **p(x)**越应该提高，比如说上面的第三维的概率为 **0.29**，就应该提高，属于应该往"正确的方向"前进

越小表示这个估计越"准确"

gk<0 时，越小，负得越多表示在这一维上的概率应该降低，比如说第二维 **0.21** 就应该得到降低。属于应该朝着"错误的反方向"前进

越大，负得越少表示这个估计越"不错误 "

总的来说，对于一个样本，最理想的梯度是越接近 **0** 的梯度。所以，我们要能够让函数的估计值能够使得梯度往反方向移动（**>0** 的维度上，往负方向移动，**<0** 的维度上，往正方向移动）最终使得梯度尽量**=0**），并且该算法在会严重关注那些梯度比较大的样本，跟 **Boost** 的意思类似。

得到梯度之后，就是如何让梯度减少了。这里是用的一个迭代+决策树的方法，当初始化的时候，随便给出一个估计函数 **F(x)**（可以让 **F(x)**是一个随机的值，也可以让 **F(x)=0**），然后之后每迭代一步就根据当前每一个样本的梯度的情况，建立一棵决策树。就让函数往梯度的反方向前进，最终使得迭代 **N** 步后，梯度越小。

这里建立的决策树和普通的决策树不太一样，首先，这个决策树是一个叶子节点数 **J** 固定的，当生成了 **J** 个节点后，就不再生成新的节点了。

算法的流程如下**:**（参考自 **treeBoost** 论文）

Algorithm 6: L_K_TreeBoost

$F_{k0}(\mathbf{x}) = 0, \quad k = 1, K$

For $m = 1$ to M do:

$p_k(\mathbf{x}) = \exp(F_k(\mathbf{x})) / \sum_{l=1}^{K} \exp(F_l(\mathbf{x})), \quad k = 1, K$

For $k = 1$ to K do:

$\tilde{y}_{ik} = y_{ik} - p_k(\mathbf{x}_i), \quad i = 1, N$

$\{R_{jkm}\}_{j=1}^{J} = J$-terminal node $tree(\{\tilde{y}_{ik}, \mathbf{x}_i\}_1^N)$

$\gamma_{jkm} = \frac{K-1}{K} \frac{\sum_{\mathbf{x}_i \in R_{jkm}} \tilde{y}_{ik}}{\sum_{\mathbf{x}_i \in R_{jkm}} |\tilde{y}_{ik}| (1 - |\tilde{y}_{ik}|)}, \quad j = 1, J$

$F_{km}(\mathbf{x}) = F_{k,m-1}(\mathbf{x}) + \sum_{j=1}^{J} \gamma_{jkm} 1(\mathbf{x} \in R_{jkm})$

endFor

endFor

end Algorithm

0 1 2 3 4 5 6 7

0. 表示给定一个初始值

1. 表示建立 **M** 棵决策树（迭代 **M** 次）

2. 表示对函数估计值 **F(x)**进行 **Logistic** 变换

3. 表示对于 **K** 个分类进行下面的操作（其实这个 **for** 循环也可以理解为向量的操作，每一个样本点 **xi** 都对应了 **K** 种可能的分类 **yi**，所以 **yi, F(xi), p(xi)** 都是一个 **K** 维的向量，这样或许容易理解一点）

4. 表示求得残差减少的梯度方向

5. 表示根据每一个样本点 **x**，与其残差减少的梯度方向，得到一棵由 **J** 个叶子节点组成的决策树

6. 为当决策树建立完成后，通过这个公式，可以得到每一个叶子节点的增益（这个增益在预测的时候用的）

每个增益的组成其实也是一个 **K** 维的向量，表示如果在决策树预测的过程中，如果某一个样本点掉入了这个叶子节点，则其对应的 **K** 个分类的值是多少。比如说，**GBDT** 得到了三棵决策树，一个样本点在预测的时候，也会掉入 **3** 个叶子节点上，其增益分别为（假设为 **3** 分类的问题）：

(0.5, 0.8, 0.1)，(0.2, 0.6, 0.3)，(0.4, 0.3, 0.3)，那么这样最终得到的分类为第二个，因为选择分类 **2** 的决策树是最多的。

7. 的意思为，将当前得到的决策树与之前的那些决策树合并起来，作为新的一个模型**(**跟 **6** 中所举的例子差不多**)**

GBDT 的算法大概就讲到这里了，希望能够弥补一下上一篇文章中没有说清楚的部分：）