文本上的算法

v3.0

路彦雄(@yanxionglu)

yanxionglu@gmail.com

目录

版本变化：
v1.0——完成理论篇
v2.0——完成应用篇
v2.1——增加 GBDT 模型
v2.2——修正第六章部分内容，且增加推荐系统
v3.0——增加程序优化章节；增加 word multi-embedding 部分；丰富 deep learning、learning to rank 和相似度计算部分；修改其他部分笔误

序：《文本上的算法》主要分两大部分：第一部分是理论篇，主要介绍机器学习的基础和一些具体算法；第二部分应用篇，主要是一些 NLP 的应用，比如：搜索引擎原理是什么？它为什么要建索引？有什么理论基础吗？之所以抽时间写这个文档，是由于以下方面的考虑：1、这些都是我个人的一些总结和思考，零零散散的，所以想整理成一个稍微正式一点的文档，方便查阅；2、这些知识在平常的工作中都会经常用到，整理成这个较正式文档也可以当作读者的一种参考。3、除了必不可少的公式外，尽量以更口语化的方式表达出来，抛弃掉繁琐的证明，提取出算法的核心，尽可能达到深入浅出。由于本人水平有限，难免会有一些错误，希望大家不吝指出。

理论篇

第一章、你必须知道的一些基础知识

要想明白机器学习，一些概率论和信息论的基本知识一定要知道，本章就简单的回顾下这些知识（本章可跳过阅读）。本文如不特殊声明，无下标的变量（例如 x）均为向量，有下标的（例如x_i）均为标量。

1.1 概率论

概率就是描述一个事件发生的可能性。我们生活中绝大多数事件都是不确定的，每一件事情的发生都有一定的概率（确定的事件就是

100%的概率），天气预报说明天有雨，那么它也只是说明天下雨的概率很大。再比如：掷骰子，我把一个骰子掷出去，问某一个面朝上的概率是多少？在骰子没有做任何手脚的情况下，直觉告诉你任何一个面朝上的概率都是 1/6，如果你只掷几次很难得出这个结论，但是如果你掷上 1 万次或更多，那么必然可以得出任何一个面朝上的概率都是 1/6 的结论，这就是大数定理：当试验次数（样本）足够多的时候，事件出现的频率无限接近于该事件真实发生的概率。

假如我们用概率函数$p(x)$来表示随机变量$x \in X$的概率，那么就要满足如下两个特性：

$$0 \leq p(x) \leq 1$$

$$\sum_{x \in X} p(x) = 1$$

联合概率$p(x, y)$表示两个事件共同发生的概率。假如这两个事件相互独立，那么就有联合概率$p(x, y) = p(x)p(y)$。

条件概率$p(y|x)$是指在已知事件x发生的情况下，事件y发生的概率，且有：$p(y|x) = p(x, y)/p(x)$。如果这两个事件相互独立，那么$p(y|x)$与$p(y)$相等。

联合概率和条件概率分别对应两个模型：生成模型和判别模型，这两个模型将在下一章中解释。

概率分布的均值称为**期望**，定义如下：

$$E[X] = \sum_{x \in X} x\, p(x)$$

期望就是对每个可能的取值x，与其对应的概率值p(x)，进行相乘求和。假如一个随机变量的概率分布是均匀分布，那么它的期望就等于均值，因为它的概率分布$p(x) = 1/N$。

概率分布的**方差**定义如下：

$$\mathrm{Var}[X] = \sum_{x \in X} (x - E[x])^2 \, p(x) = E[(X - E[X])^2]$$

可以看出，方差是表示随机变量偏离期望的大小，所以它是衡量数据的波动性，方差越小表示数据越稳定，反之方差越大表示数据的波动性越大。

另外，你还需要知道的几个常用的概率分布：均匀分布、正态分布、二项分布、泊松分布、指数分布等等，你还可以了解下矩阵的东西，因为所有公式都可以表示成矩阵形式。

1.2 信息论

假如一个朋友告诉你外面下雨了，你也许觉得不怎么新奇，因为下雨是很平常的一件事情，但是如果他告诉你他见到外星人了，那么你就会觉得很好奇：真的吗？外星人长什么样？同样两条信息，一条信息量很少，一条信息量很大，很有价值，那么这个价值怎么量化呢？就需要信息熵，一个随机变量X的**信息熵**定义如下：

$$H(X) = -\sum_{x \in X} p(x) \log p(x)$$

信息越少，事件（变量）的不确定性越大，它的信息熵也就越大，需要搞明白该事件所需要的额外信息就越大，也就是说搞清楚小概率事

件所需要额外的信息就越大，比如说，为什么大多数人愿意相信专家的话，因为专家在他专注的领域了解的知识（信息量）多，所以他对某事件的看法越透彻，不确定性就越小，那么他所传达出来的信息量就越大，听众搞明白该事件所需要的额外信息就越少。不过，在这个利益至上的社会，有时对专家说的话也只能呵呵了。总之，记住一句话：**信息熵表示的是不确定性的度量。**信息熵越大，不确定性越大。

联合熵的定义为：

$$H(X,Y) = -\sum_{x\in X, y\in Y} p(x,y)\,\log p(x,y)$$

联合熵描述的是一对随机变量X和Y的不确定性。

条件熵的定义为：

$$H(Y|X) = -\sum_{x\in X, y\in Y} p(x,y)\,\log p(y|x)$$

条件熵衡量的是在一个随机变量X已知的情况下，另一个随机变量Y的不确定性。

两个随机变量X和Y，它们的**互信息**定义为：

$$I(X;Y) = \sum_{x\in X, y\in Y} p(x,y)\,\log \frac{p(x,y)}{p(x)p(y)}$$

互信息是衡量两个随机变量（事件）的相关程度，当X和Y完全相关时，它们的互信息就是 1；反之，当X和Y完全无关时，它们的互信息就是 0。

互信息和熵有如下关系：

$$I(X;Y) = H(X) - H(X|Y) = H(Y) - H(Y|X)$$

相对熵（又叫交叉熵或者 KL 距离）的定义如下：

$$D(P||Q) = \sum_{x \in X} p(x) \log \frac{p(x)}{q(x)}$$

相对熵是衡量相同事件空间里两个概率分布（函数）的差异程度（不同于前面的熵和互信息，它们衡量的是随机变量的关系），当两个概率分布完全相同时，他们的相对熵就为 0，当他们的差异增加时，相对熵就会增加。相对熵又叫 KL 距离，但是它不满足距离定义的三个条件中的两个：1、非负性（满足）；2、对称性（不满足）；3、三角不等式 （不满足）。

好了，介绍了这么多概念公式，那么我们来个实际的例子，在文本处理中，有个很重要的数据就是词的互信息，上面说了，互信息是衡量两个随机变量（事件）的相关程度，那么词的互信息，就是衡量两个词的相关程度，比如，“计算机”和“硬件”的互信息就比“计算机”和“杯子”的互信息要大，因为它们更相关。那么如何在大量的语料下统计出词与词的互信息呢？公式中可以看到需要计算三个值：$p(x)$、$p(y)$和$p(x,y)$，它们分别表示x独立出现的概率，y独立出现的概率，x 和 y同时出现的概率。前两个很容易计算，直接统计下词频然后除以总词数就知道了，最后一个也很容易，统计一下x 和 y同时出现（通常会限定一个窗口）的频率除以所有无序对的个数就可以了。这样，词的互信息就计算出来了，这种统计最适合使用 Map-Reduce 来计算。

1.3 贝叶斯法则

贝叶斯法则是概率论的一部分，之所以单独拿出来写，是因为它真的很重要。它是托马斯•贝叶斯生前在《机遇理论中一个问题的解》中提出的一个当时叫“逆概率”问题，贝叶斯逝世后，由他的一个朋友替他发表了该论文，后来在这一理论基础上，逐渐形成了贝叶斯学派。

贝叶斯法则的定义如下：

$$p(x|y) = \frac{p(y|x)p(x)}{p(y)}$$

$p(x|y)$称为后验概率，$p(y|x)$称为似然概率，$p(x)$称为先验概率，$p(y)$一般称为标准化常量。也就是说，后验概率可以用似然概率和先验概率来表示。就这个公式，非常非常有用，很多模型的基础就是它，比如：贝叶斯模型估计、机器翻译、Query 纠错、搜索引擎等等，在以后的章节中，大家经常会看到这个公式。

好了，这个公式看着这么简单，到底能有多大作用呢？那我们先拿中文分词来说说这个公式如何应用的。

中文分词在中文自然语言处理中可以算是最低层，最基本的一个技术了，因为几乎所有的文本处理任务都要首先经过分词这步操作，那么到底要怎么对一句话分词呢？最简单的方法就是查字典，如果这个词在词典中出现了，那么就是一个词，当然，查字典要有一些策略，最常用的就是最大匹配法，最大匹配法是怎么回事呢？举个例子来说，比如要对“中国地图”来分词，先拿“中”去查字典，发现“中”在

字典里（单个词肯定在字典里），这时肯定不能返回，要接着查，“中国”也在字典里，然后再查“中国地”，发现没在字典里，那么“中国”就是一个词了；然后同样的处理剩下的句子。所以，最大匹配法就是匹配最长在字典中出现的词。查字典法有两种：正向最大匹配法和反向最大匹配法，一个是从前向后匹配，一个是从后向前匹配。但是查字典法会遇到一个自然语言处理中很棘手的问题：歧义问题，如何解决歧义问题呢？

我们就以“学历史知识”为例来说明，使用正向最大匹配法，我们把“学历史知识”从头到尾扫描匹配一遍，就被分成了“学历\史\知识”，很显然，这种分词不是我们想要的；但是如果我们使用反向最大匹配法从尾到头扫描匹配一遍，那就会分成“学/历史/知识”，这才是我们想要的分词结果。可以看出查字典法可以用来分词，就会存在二义性，一种解决办法就是分别从头到尾和从尾到头匹配，在这个例子中，我们分别从头到尾和从尾到头匹配后将得到“学历\史\知识”和“学/历史/知识”，很显然，这两个分词都有“知识”，那么说明“知识”是正确的分词，然后就看“学历\史”和“学/历史”哪个是正确的，从我们人的角度看，很自然想到“学/历史”是正确的，为什么呢？因为1、在“学历\史”中“史”这个词单独出现的几率很小，在现实中我们几乎不会单独使用这个词；2、“学历”和“史”同时出现的概率也要小于“学”和“历史”同时出现的概率，所以“学/历史”这种分词将会胜出。这只是我们大脑的猜测，有什么数学方法证明呢？有，那就是基于统计概率模型。

我们的数学模型表示如下：假设用户输入的句子用S表示，把S分词后可能的结果表示为A：A_1，A_2, …, A_k（A_i表示词），那么我们就是求条件概率p(A|S)达到最大值的那个分词结果，这个概率不好求出，这时贝叶斯法则就用上派场了，根据贝叶斯公式改写为:

$$p(A|S) = \frac{p(S|A)p(A)}{p(S)}$$

显然， p(S)是一个常数，那么公式相当于改写成：

$$p(A|S) \propto p(S|A)p(A)$$

其中，p(S|A)表示(A)这种分词生成句子S的可能性；p(A)表示(A)这种分词本身的可能性。

下面的事情就很简单了，对于每种分词计算一下p(S|A)p(A)这个值，然后取最大的，得到的就是最靠谱的分词结果。比如“学历史知识”（用S表示）可以分为如下两种（当然，实际情况就不止两种情况了）：“学历\史\知识”（用A表示，A_1=学历，A_2=史，A_3=知识）和“学/历史/知识”（用B表示，B_1=学，B_2=历史，B_3=知识），那么我们分别计算一下p(S|A)p(A)和p(S|B)p(B)，哪个大，说明哪个就是好的分词结果。

但是$p(S|A_1, A_2, \dots A_k)$p$(A_1, A_2, \dots A_k)$这个公式并不是很好计算，$p(S|A_1, A_2, \dots A_k)$可以认为就是 1，因为由$A_1, A_2, \dots A_k$必然能生成S, 那么剩下就是如何计算$p(A_1, A_2, \dots A_k)$？

在数学中，要想简化数学模型，那就是假设。我们假设句子中一个词的出现概率只依赖于它前面的那个词(当然可以依赖于它前面的m个词），这样，根据全概率公式：

$$p(A_1, A_2, \dots A_k) =$$

$$P(A_1)P(A_2|A_1)P(A_3|A_2,A_1)\dots P(A_k|A_1,A_1,\dots,A_{k-1})$$

就可以改写为：

$$p(A_1,A_2,\dots A_k)=P(A_1)P(A_2|A_1)P(A_3|A_2)\dots P(A_k|A_{k-1})$$

接下来的问题就是如何估计$P(A_i|A_{i-1})$。然而，$P(A_i|A_{i-1})=P(A_{i-1},A_i)/P(A_{i-1})$，这个问题变得很简单，只要数一数这对词（$A_{i-1},A_i$）在统计的文本中前后相邻出现了多少次，以及$A_{i-1}$本身在同样的文本中出现了多少次，然后用两个数一除就可以了。

上面计算$p(A_1,A_2,\dots A_k)$的过程其实就是**统计语言模型**，然而真正在计算语言模型的时候要对公式进行平滑操作。Zipf 定律指出：一个单词出现的频率与它在频率表里的排名（按频率从大到小）成反比。这说明对于语言中的大多数词，它们在语料中的出现是稀疏的，数据稀疏会导致所估计的分布不可靠，更严重的是会出现零概率问题，因为$P(A_{i-1},A_i)$的值有可能为 0，这样整个公式的得分就为 0，而这种情况是很不公平的，所以平滑就是解决这种零概率问题。具体的平滑算法读者可以参考下论文：《An empirical study of smoothing techniques for language modeling》。

然而在实际系统中，由于性能等因素，很少使用语言模型来分词消歧，而是使用序列标注、共现和一些规则等方法来消歧。

第二章、我们生活在一个寻求最优的世界里

金庸小说里一般一个人的内功越高他的武功就越高，练了易筋经之后，随便练点什么功夫都能成为高手。同样的，最优化模型就是机器学习的内功，几乎每个机器学习模型背后都是一个最优化模型。这章讲解最优化模型。

2.1 最优化问题

通常人们会认为商人最精明，因为他们总是希望付出最小的成本来获得最大的收益。其实，我们每个人都生活在一个寻求最优的世界里，因为人心是贪婪的，人们永远想得到他们认为最好的东西。我们买东西的时候，是不希望花尽可能少的钱买到质量更好的物品？我们从一个地方去到另一个地方，是不希望尽可能走最捷径的路线？

科学抽象于生活，科学服务于生活。所以，每个机器学习背后都是一个最优化问题。一般的最优化形式表示如下：

$$\min \quad f(x)$$

$$\text{s.t.} \quad h(x) = 0$$

$$g(x) \leq 0$$

f(x)是目标函数，h(x)和g(x)分别是约束条件，有的问题可以没有约束条件（只有f(x)，称为无约束优化；只有f(x)和h(x)称为有等式约束优化；f(x)和h(x)、g(x)都有称为有不等式约束优化）。

那么目标函数到底是什么呢？

来了一个机器学习问题以后，肯定会有一个真实模型可以解决它，但是我们并不知道这个真实模型是什么（如果能知道的话，那就不用学习了，直接用就可以了），那么就要设计一个模型来代替真实模型（假设$h = f(x)$为你设计的模型，$y = R(x)$为真实模型，$x = \{x_1, .., x_N\}$为整个模型的输入），那么怎么才能说你设计的这个模型很好呢？很简单，只要你设计的模型和真实模型的误差越小，那就说明你的模型越好，误差通常使用损失函数来表示，常用的有以下几种：

平方损失：$L\big(y, f(x)\big) = (f(x) - y)^2$

绝对损失：$L\big(y, f(x)\big) = |f(x) - y|$

合页损失：$L\big(y, f(x)\big) = \max(0, 1 - y \cdot f(x))$

似然损失：$L\big(y, f(x)\big) = -\log P(y|x)$

似然损失的最小化，就是求$\log P(y|x)$的最大化，这就是后面专门要说的最大似然估计。

而损失函数（误差）的期望，称为期望风险，学习的目标就是使期望风险最小，即（M 为样本数）：

$$\min \frac{1}{M} \sum_{i=1}^{M} L\left(y^i, f\left(x^i, \theta\right)\right)$$

上面不是说真实模型是不知道的么？那么它们的期望风险自然也没法计算了，怎么最小化这个期望风险啊？期望风险是指你设计的模型和真实模型的期望误差，不知道真实模型这个自然求不出来了；虽然我们不知道真实模型$y = R(x)$是什么，但是如果我们知道所有的输入x和它对应的输出y的话，那么我们也没必要知道这个模型R是什么了，因为你给任何一个输入x我都可以给你计算一个最优的y（显然也不可

能得到，能得到的话，也没必要求模型了）。那好，那我们找一些输入x'（它肯定是x的子集），然后用人工的笨办法把x'的所有最优y'算出来（$D = \{x', y'\}$称为样本对），这样，我们在计算期望风险的时候，就可以用计算好的y'直接替代真实模型$y = R(x)$就可以了，用这种方法计算出来的风险就是经验风险，根据大数定理，当样本对趋于无穷大时，经验风险也就越接近期望风险。所以，我们就可以用**经验风险最小化**来估计期望风险。

但是，我们的样本对有限，就导致经验风险估计期望风险并不理想，会产生过拟合现象。过拟合现象就是你把样本数据拟合的太完美，也可以说是模型复杂度很高，然而到未知数据中却拟合的很差（这种对未知数据的预测能力叫做泛化能力），相反，欠拟合现象就是在样本数据上拟合的不好，在未知数据上也不好。所以，为了尽可能避免过拟合现象的出现，就要对模型的复杂度进行惩罚，这就是正则化，一般正则化，就是对模型的参数进行惩罚。这样，就相当于目标函数变成了：

$$\min \frac{1}{M}\sum_{i=1}^{M} L\left(y^i, f(x^i, \theta)\right) + \gamma J(f)$$

这也叫**结构风险最小化**。正则化公式可以有很多种，比如，L_0范数、L_1范数、L_2范数等，例如下面的正则化公式：

$$J(f) = \frac{1}{2}||\theta||^2 = \frac{1}{2}\sum_{i=1}^{N} {\theta_i}^2$$

现在还有个问题，对于一个优化问题，我的模型可以有好多种选择，最简单的比如$f(x,\theta)$中θ选的不同，那么最终结果就不同，如何确定一个好的模型呢？那就需要交叉验证。

交叉验证就是随机的把样本数据分成：训练集、验证集。首先在训练集中训练出各种模型$f_1(x,\theta), f_2(x,\theta), \ldots$，然后在验证集上评价各个模型的误差，选出一个误差最小的模型就是好的模型。在这儿，就要解释两个概念：偏差和方差。偏差是衡量单个模型的误差，比如：$f_1(x,\theta)$这个模型的偏差就可以用$L_1 = (f_1(x,\theta) - y)^2$来表示，$f_2(x,\theta)$这个模型的偏差可以用$L_2 = (f_2(x,\theta) - y)^2$来表示，所以偏差是衡量单个模型自身的好坏，它并不管别的模型怎么样；而方差是用来多个模型间比较，它并不管自己这个模型和真实模型的误差多大，而是从别的模型来衡量自己的好坏，也就是它认为所有模型的平均值，就可以代表真实模型（这也有个潜在假设：大多数情况是正常无噪声的，否则平均值也代表不了真实模型），那么它和这个平均值比较就可以了，比如：$f_1(x,\theta)$这个模型的方差就可以用$(f_1(x,\theta) - (L_1 + L_2)/2)^2$来表示。从这儿，就可以得出一些结论，一个模型越复杂，偏差就越小，方差就越大；相反，一个模型越简单，偏差就越大，方差就越小，这两个概念就是一个博弈的过程，最好的模型就是偏差和方差之和最优的模型。

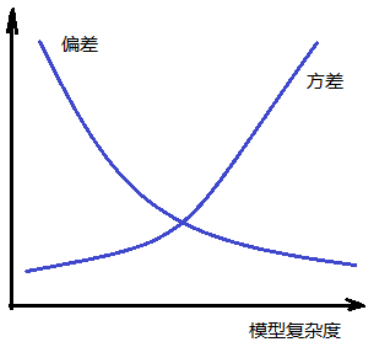


图 2.1

这就是最优化模型，你需要根据实际问题设计一个模型，设计出它的目标函数，然后可以根据交叉验证选个最好的模型（如果你的数据较好，这步有时可以省略）。

2.2 最大似然估计/最大后验估计

上面的讲解中可以看出，对一个最优化问题，我们首先要选定模型$h = f(x, \theta)$，但是这个模型h会有无穷多个选择，到底要选哪个呢？如果你现在要急用钱，那你首先想到的是找家人或者朋友去借钱，总不能找奥巴马去借吧；同样，选模型我们也是优先选择我们熟悉的模型，比如：线性模型，高斯模型等，因为这些模型我们研究的已经很透彻了，只需根据样本数据代入公式就可以求解出它们的参数θ，这就是参数估计；如果对一无所知的模型估计参数，那就是非参数估计。

参数估计也有很多种方法，最常用的就是最小二乘法、最大似然估计和贝叶斯估计等。

最大似然估计和贝叶斯估计分别代表了频率派和贝叶斯派的观点。频率派认为，参数是客观存在的，只是未知而已，因此，频率派最关心最大似然函数，只要参数求出来了，给定自变量 x，y 也就知道了。假设我们观察的变量是 x，观察的变量取值（样本）为$x = \{x_1, .., x_N\}$，要估计的模型参数是θ，x 的分布函数是$p(x|\theta)$。那么最大似然函数就是θ的一个估计值，它使得事件发生的可能性最大，即：

$$\theta_{MLE} = \text{argmax}_{\theta} p(x|\theta)$$

通常，我们认为 x 是独立同分布的，即有：

$$p(x|\theta) = \prod_{i=1}^{N} p(x_i|\theta)$$

由于连乘会可能造成浮点下溢，所以通常就最大化对数形式，也就是：

$$\theta_{MLE} = \text{argmax}_\theta\{\sum_{i=1}^{N} \log p(x_i|\theta)\}$$

所以最大似然估计的一般求解流程就是：

（1）写出似然函数：$\mathcal{L}(\theta) = p(x|\theta) = \prod_{i=1}^{N} p(x_i|\theta)$。

（2）对似然函数取 log：$L(\theta) = \log\mathcal{L}(\theta) = \sum_{i=1}^{N} \log p(x_i|\theta)$。

（3）求$\text{argmax}_\theta L(\theta)$（或者$\text{argmin}_\theta(-L(\theta))$）：求解方法见下一节。

最大似然估计中θ是固定的一个值，只要这个θ能很好的拟合样本x就是好的，前面说了，它拟合样本数据很好，不一定拟合未知数据就很好（过拟合现象）。所以用频率派的理论可以得出很多扭曲事实的结论：只要我没看到过飞机相撞，那么飞机永远就不可能相撞。这时，贝叶斯学派就开始说了，参数θ也应该是随机变量（$p(\theta)$），和一般随机变量没有本质区别，它也有概率（θ取不同值的概率），也就是尽管我没看到飞机相撞，但是飞机还是有一定概率可能相撞，正是因为参数不能固定，当给定一个输入 x 后，我们不能用一个确定的 y 表示输出结果，必须用一个概率的方式表达出来。所以，我们希望知道所有θ在获得观察数据x后的分布情况，也就是后验概率$p(\theta|x)$，根据贝叶斯公式我们有：

$$p(\theta|x) = \frac{p(x|\theta)p(\theta)}{p(x)} = \frac{p(x|\theta)p(\theta)}{\int p(x|\theta)p(\theta)d\theta}$$

可惜的是，上面的后验概率通常是很难计算的，因为要对所有的参数进行积分，而且，这个积分其实就是所有θ的后验概率的汇总，其实它是与最优θ是无关的，而我们只关心最优θ。在这种情况下，我们采用了一种近似的方法求后验概率，这就是最大后验估计：

$$\theta_{MAP} = \text{argmax}_{\theta} p(\theta|x) = \text{argmax}_{\theta} p(x|\theta)p(\theta)$$

最大后验估计相比最大似然估计，只是多了一项先验概率，它正好体现了贝叶斯认为参数也是随机变量的观点，在实际运算中通常通过超参数给出先验分布。**最大似然估计其实是经验风险最小化的一个例子，而最大后验估计是结构风险最小化的一个例子。**如果样本数据足够大，最大后验概率和最大似然估计趋向于一致，如果样本数据为 0，最大后验就仅由先验概率决定，就像推荐系统中对于一个毫无历史数据的用户，只能给他推荐热门（先验概率高）的内容了。尽管最大后验估计看着要比最大似然估计完善，但是由于最大似然估计简单，很多方法还是使用最大似然估计，也就是频率派和贝叶斯派谁也没把谁取代掉。

最大似然估计和最大后验估计已经懂了，那么顺带说下最大二乘估计。对于最小二乘法估计，当从模型总体随机抽取 M 组样本观测值后，最合理的参数估计值应该是使得模型能最好地拟合样本数据，也就是估计值和观测值之差的平方和最小。而对于最大似然估计，当从模型总体随机抽取 M 组样本观测值后，最合理的参数估计值应该是使得从模型中抽取该 M 组样本观测值的概率最大。显然，这是从不同原理出发的两种参数估计方法。而且，最小二乘估计有个假设：模型服从高斯分布。

现在我们已经知道如何构造最优化问题的目标函数了，以及进行参数估计的一些方法，剩下的问题就是如何具体的求解参数了？

2.3 梯度下降法

大多数最优化问题（凸规划）是有全局最优解的（下面左图），而有的最优化问题只有局部最优解，当然局部最优解有可能就是全局最优解，但是不容易使得得到的局部最优解就正好是全局最优解（下面右图）。求解最优化问题，本质上就是怎么向最优解移动，达到某个条件（收敛）就停止。

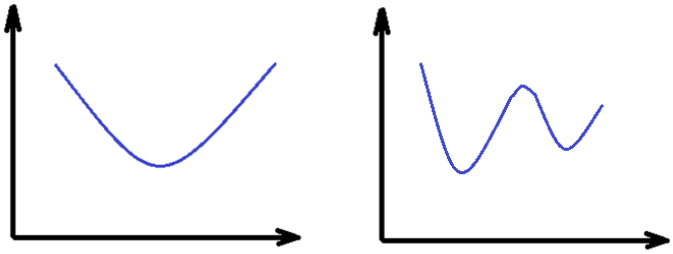


图 2.2

有的最优化问题有解析解，可以直接求解，那么怎么求解呢？直接对其求导数，然后令导数为零，就可以解出候选最优解了。

有的最优化问题没有解析解，只能通过一些启发式算法（遗传算法等）或者数值计算的方法来求解。对于无约束优化问题常用的算法大概有：梯度下降法，牛顿法/拟牛顿法，共轭梯度法，坐标下降法等（这些方法使用导数，还有些算法是不使用导数的）。而对于有约束优化问题大多是通过拉格朗日乘子法转换成无约束问题来求解。

本节就以线性回归模型来看下梯度下降法到底是怎么回事？假设有如下表示的线性函数：

$$f(x,\theta) = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \cdots = \sum_{i=0}^{N} \theta_i x_i = \theta^T x$$

如果它的损失函数使用平方损失函数，则有（没有正则化）：

$$L(\theta) = \frac{1}{2}\sum_{j=1}^{M}\left(f\left(x^j,\theta\right) - y^j\right)^2$$ （M 为样本数）

梯度下降法指出，**函数f在某点x沿着梯度相反的方向下降最快，**也就是说我从某点x出发，沿着梯度相反的方向移动，就可以走到最优解(使得损失函数最小)了。梯度是什么呢？在这儿就是导数。所以它的每一步的迭代公式就是：

$$\theta_i = \theta_i + \alpha\left(-\frac{\partial}{\partial\theta_i}L(\theta)\right)$$

$$= \theta_i - \alpha\frac{\partial}{\partial\theta_i}L(\theta)$$

其中α是个定值，就是学习速度，控制每步移动的幅度。

相反，如果求损失函数最大化的话，就是梯度上升法：

$$\theta_i = \theta_i + \alpha\frac{\partial}{\partial\theta_i}L(\theta)$$

又有：

$$-\frac{\partial}{\partial\theta_i}L(\theta) = \sum_{j=1}^{M}\left(y^j - f\left(x^j,\theta\right)\right){x_i}^j$$

所以最终的迭代公式就是：

$$\theta_i = \theta_i + \alpha\sum_{j=1}^{M}\left(y^j - f\left(x^j,\theta\right)\right){x_i}^j$$

梯度下降法的流程就是：

while (直到收敛)：

$$\theta_i = \theta_i + \alpha\sum_{j=1}^{M}\left(y^j - f\left(x^j,\theta\right)\right){x_i}^j \ (i = 0..N)$$

这就是梯度下降法的迭代流程，首先任意选定一个θ，然后使用公式迭代，一直到收敛（θ的变化小于一个阈值）停止，就解出了参数θ。

可以看出，公式中有个求和，也就是每次迭代都需要计算全部样本，所以当样本M很大时，计算代价很大，那么就需要考虑个好的办法减少计算，这就是随机梯度下降法。

随机梯度下降法并不计算梯度的精确值，而是计算一个估计值，也就是每次迭代都是基于一个样本，迭代流程就成为：

for j=1 to M：

$$\theta_i = \theta_i + \alpha\left(y^j - f\left(x^j, \theta\right)\right) x_i^j \ (i = 0..N)$$

这个公式有什么好处呢？它可以并行计算，因为样本间是无关的。所以当样本特别大时，可以使用随机梯度下降法，它的缺点是通常找到的最小值是最优解的近似值。当样本数较小时，一般也不用梯度下降法，因为它收敛太慢了（尤其接近最优解的时候），而是使用 LBFGS 或者 CG 等来训练。

在这儿要记住一个观点：1、要想得到最优解就要付出更大的代价。就像生活中，吊儿郎当的人活得会比较轻松，因为他做事不需要最优；谨慎细致的人就活得比较累，每件事情都要做到最好，不同性格的人有不同的生活态度，没有对错。2、好多问题没有最优解或者最优解无法求出，那么就要用近似的方法来求解出近似最优解。生活中好多事情压根没办法做到十全十美，所以我们只能尽自己最大努力达到尽可能好。

梯度下降法是最简单且很好理解的一个算法，理解了这个算法，其他求解算法（牛顿法/拟牛顿法，共轭梯度法，坐标下降法等）就较容易了。

对于无约束优化问题我们已经知道一些求解算法了，那么还有两类问题：等式约束优化问题和不等式优化问题。

对于等式约束优化问题：

$$\min \quad f(x)$$

$$s.t. \quad h(x) = 0$$

写出它的拉格朗日乘子法：

$$L(x, \alpha) = f(x) + \alpha h(x)$$

之后的求解方法就和求解析解一样了，用$L(x, \alpha)$分别对x, α求导数，然后令它们的导数为零，就可以分别解出x, α的候选最优解了。

对于不等式约束优化问题：

$$\min \quad f(x)$$

$$s.t. \quad h(x) = 0$$

$$g(x) \leq 0$$

写出它的拉格朗日乘子法：

$$L(x, \alpha, \beta) = f(x) + \alpha h(x) + \beta g(x) \quad (\beta \geq 0)$$

但是上式要想有和原不等式约束优化问题一样的最优解，必须满足 KKT 条件：（1）$L(x, \alpha, \beta)$分别对x求导为零；（2）$\beta g(x) = 0$；（3）$g(x) \leq 0$；（4）$\beta \geq 0$，$\alpha \neq 0$；（5）$h(x) = 0$。

当原问题不太好解决的时候可以利用拉格朗日乘数法得到其对偶问题，满足强对偶性条件时它们的解是一致的。那么什么是对偶问题呢？还是以上述不等式约束问题为例说明，写出它的拉格朗日函数：

$$L(x,\alpha,\beta) = f(x) + \alpha h(x) + \beta g(x) \quad (\beta \geq 0)$$

然后定义一个函数：$\theta_p(x) = \max_{\alpha,\beta,\beta\geq 0} L(x,\alpha,\beta)$

这个函数是α,β的函数，如果x违反原始问题的约束条件，即$h(x) \neq 0$或者$g(x) > 0$，那么我们总是可以调整α和$\beta \geq 0$来使得$\theta_p(x)$有最大值为正无穷，而只有$h(x)$和$g(x)$都满足约束时，$\theta_p(x)$为$f(x)$，也就是说$\theta_p(x)$的取值是：$f(x)$或者∞。所以$\min f(x)$就转为求$\min_x \theta_p(x)$了，即：

$$\min_x \theta_p(x) = \min_x \max_{\alpha,\beta,\beta\geq 0} L(x,\alpha,\beta)$$（它的最优解记为p^*）

再定义一个函数：$\theta_d(\alpha,\beta) = \min_x L(x,\alpha,\beta)$，则有：

$$\max_{\alpha,\beta,\beta\geq 0} \theta_d(\alpha,\beta) = \max_{\alpha,\beta,\beta\geq 0} \min_x L(x,\alpha,\beta)$$（它的最优解记为d^*）

$\theta_p(x)$和$\theta_d(x)$互为对偶问题。可以证明：$d^* \leq p^*$，如果满足强对偶性：目标函数和所有不等式约束函数是凸函数，等式约束函数是仿射函数（形如$y = w^t x + b$），且所有不等式约束都是严格的约束，那么$d^* = p^*$，就是说原问题和对偶问题的解一致。

对偶原理告诉我们，如果$\min f(x) = \min_x \max_{\alpha,\beta,\beta\geq 0} L(x,\alpha,\beta)$不好求解，那么就求解$\max_{\alpha,\beta,\beta\geq 0} \min_x L(x,\alpha,\beta)$，后面我们可以看到它的应用。

总结一下：最优化问题其实很简单，首先需要一个模型：目标函数和约束函数（函数中的变量通常就是特征，下章解释），不同的问题会对应不同的模型，需要自己设计；然后对该模型的参数进行求解。

第三章、让机器可以像人一样学习

再讲机器学习(Machine Learning)之前，先要灌输三个名词：训练数据、特征、模型。它们就是机器学习的核心，模型就是上一章讲解过的最优化问题。

3.1 何为机器学习(Machine Learning)

我们首先看看人是如何学习的。

小明（万能主角出场）在校园里看到两排东西（一排图 3.1，一排图 3.2），但他不认识它们是什么，于是他就去问同学，同学瞥了一眼就说：左边的是汽车；右边的是摩托车。

图 3.1

图 3.2

小明仔细观察了下，左边这个叫汽车的比较大，而且有四个轮子；右边叫摩托车的比较小，而且只有两个轮子。于是小明的大脑中就形成了这么一个概念（模型一）：

$$\text{if}(\text{大，而且有四个轮子})\ \text{then}\ \text{它是汽车}$$

$$\text{if}(\text{小，而且有两个轮子})\ \text{then}\ \text{它是摩托车}$$

太好了，小明学会了如何区分汽车和摩托车，然后高高兴兴的回家了，但是在路上，他见到了图 3.3 这么个东西，根据他之前学到的知识，它应该就是摩托车，于是他就说到：这是个摩托车。路人甲听到了，说：小朋友，这不是摩托车，这是自行车。

图 3.3

小明很沮丧，但是他没有放弃，他回家仔细想了想：如果我有更多可参考的汽车和摩托车的样例，然后提取出更多他们的特点，那么我一定能区分的很好。没错！于是小明总结了更多的不同点，最后他大脑中就形成了这么另一个概念（模型二）：

$$\text{if}\left(\begin{array}{c}\text{大，}\\ \text{有四个轮子，}\\ \text{轮子大小一样，}\\ \text{四个门，}\\ \text{轮胎较粗，}\\ \text{座位大，}\\ \text{零部件个数} > N\text{，}\\ \text{等等}\end{array}\right)\ \text{then 它是汽车}$$

$$\text{if}\left(\begin{array}{c}\text{小，}\\ \text{有两个轮子，}\\ \text{轮子大小一样，}\\ \text{没有门，}\\ \text{轮胎较细，}\\ \text{座位小，}\\ \text{零部件个数} < M\text{，}\\ \text{等等}\end{array}\right)\ \text{then 它是摩托车}$$

小明想通了以后，觉得这下他会区分汽车和摩托车了吧，结果有一天他见到了图 3.4，这货前后轮胎大小不一样？前面还有个豹子头？不满足摩托车条件啊？小明迷茫了！

图 3.4

于是小明跑去问他爸爸，他爸爸说，你上面那些条件限制的太死板了，有些条件是可有可无的（说白了这些条件需要惩罚），我告诉你一个较好的区分方法吧（模型三）：

$$\text{if}\begin{pmatrix}\text{大，}\\ \text{有至少四个轮子，}\\ \text{使用汽油，}\\ \text{其他不重要}\end{pmatrix}\ \text{then 它是汽车}$$

$$\text{if}\begin{pmatrix}\text{小，}\\ \text{有两个轮子，}\\ \text{使用汽油，}\\ \text{其他不重要}\end{pmatrix}\ \text{then 它是摩托车}$$

小明想了想，这个比他前两个都好（当然不是最好的），以后就这样区分汽车和摩托车了。小明学会了如何区分汽车和摩托车。

前面说过一句话：“科学抽象于生活，科学服务于生活。”。所以，机器学习的过程其实就是模拟人学习的过程（当然，人比机器更智能，机器智能还远达不到人的程度）。从这个例子中可以看出，“小”，“有两个轮子”，“使用汽油”等就叫**特征**，那个 if，then 就叫**模型**，小明的模型很简单，各个特征都满足才可以。图 3.1 和图 3.2 就是**训练**

数据（图 3.2 下面的自行车可以理解为是个噪声数据，一般来说，噪声数据难免会有），小明的模型一很简单，即没有把训练数据分的很好，也没有把新的测试数据分的很好，这就是欠拟合现象；模型二很复杂，虽然把训练数据区分的很好，但是对新的测试数据（图 3.4）不能很好的区分，这就是过拟合现象，模型三是较理想的一个模型。

这就是机器学习，它的核心就是特征，模型，训练数据（标注数据或未标注数据），首先建立一个模型，然后抽取一些特征，最后在训练数据中把模型参数学习出来就可以了，这个是监督学习，它需要标注训练数据（还有一大类叫非监督学习，不需要标注训练数据，比如：聚类问题），前面说了，训练数据趋于无穷多时，模型训练的越好，但是现实中拿到更多的训练数据代价太大，再加上特征表示和模型本身都不会是最优的，所以机器学习一般得到的都是近似解，就像小明利用模型三也会有区分不出来的，也就是说机器学习只能解决大部分情况，而总会有些个例会解决不了。

可以看出，不同的机器学习任务就需要不同的特征和模型，有的问题模型是可以通用的（比如分类问题），但是特征却不能通用，需要根据不同的问题来选取，如果特征也是由机器学习出来的那该有多好，所以深度学习的一个目标就是自动学习特征，后面会讲到深度学习。

当训练出模型后，对一个输入，提取同样的特征，然后使用模型来进行预测，而这个模型$y = f(x)$一般就是条件概率分布：$p(y|x)$。

监督学习通常分为两个模型：生成模型和判别模型。

判别模型（它的概率图是无向图）是求解条件概率的$p(y|x)$，然后直接进行预测，比如：逻辑回归，SVM，神经网络，CRF 都是判别模型，所以判别模型求解的是条件概率。

生成模型（它的概率图是有向图）是首先求解两个概率$p(x|y)$和$p(y)$，然后根据贝叶斯法则求出后验概率$p(y|x)$，再进行预测，比如：朴素贝叶斯，HMM，贝叶斯网络，LDA 都是生成模型，又因为$p(x,y) = p(x|y) * p(y)$，所以生成模型求解的是联合概率。

举例来说，比如观察到一只狮子，要判断是美洲狮还是非洲狮？按照判别模型的思路，我们首先需要有一定的资料，机器学习上称为训练集，比如过去观察到的狮子的特征可以得到一个预测函数，之后把我们当前观察的狮子的一些特征提取出来，输入到预测函数中，得到一个值，就知道它是什么狮子了。而对于生成模型，我们先从所有美洲狮的特征中学习得到美洲狮的模型，同样得到非洲狮的模型，然后提取当前观察的狮子的特征，放到两个模型中，看哪个概率更大，就是什么狮子，这就是生成模型。可以看出，判别模型只需要关注类的边界就可以了，并不需要知道每一类到底是什么分布，这样它只需要有限样本就可以搞定；而生成模型要得到每类的具体分布，然后根据每个分布去判断类别，它的训练集自然需要无限样本，学习复杂度也高。造成两个模型样本空间不同的原因在于条件概率我们“已经知道”了一部分信息，这部分已经知道的信息缩小了可能取值的范围，即缩小了它的样本空间。

由生成模型可以得到判别模型，但由判别模型得不到生成模型。

接下来就看些具体的机器学习算法。在这儿要提醒一下，几乎每个机器学习模型都有假设，所以具体的应用场景或者数据应该尽可能接近所使用机器学习模型的假设。

3.2 逻辑回归(Logistic Regression)

现在我有个分类任务要做：判断一封邮件是否为垃圾邮件。什么是垃圾邮件呢？总得有个定义吧，那我们姑且先把这类邮件归为垃圾邮件：里面有广告推广、有诈骗、有非法活动等的邮件。

我们先选用 2.3 节中的线性回归模型来完成这个任务，即$y = f(x) = \theta^T x$，x为特征向量（比如：x_1表示出现广告词的次数，x_2表示是否含有电话号码，x_3表示是否含有网址链接，等等）；y为分类结果，y=1 表示为垃圾邮件，y=0 表示为非垃圾邮件。那么对于这个二分类问题就可以设置个阈值来判断：

$$y = \begin{cases} 1, & \text{if } \theta^T x \geq 0.5 \\ 0, & \text{if } \theta^T x < 0.5 \end{cases}$$

但是线性回归的输出y的取值范围随着特征向量x的变化可以是任何数值，如果要使我们上面设定阈值的分类方法有效，必须对线性回归的输出值映射到一个固定的范围，这就需要请出逻辑回归（Logistic Regression）。

逻辑回归在线性回归的输出y上引入了一个函数g(z)：

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

该函数（称为 sigmoid 函数）的作用就是可以把某个值映射到 0,1 区间，它的曲线图大致如下所示：

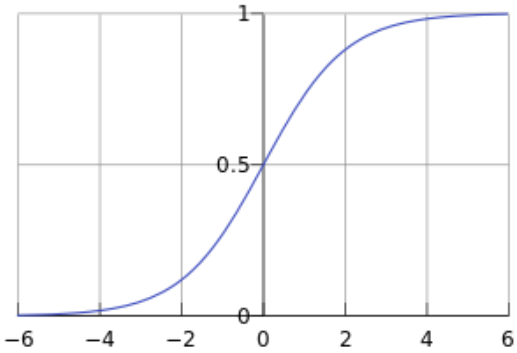


图 3.5

这样，整个逻辑回归公式就为：

$$h_\theta(x) = g(\theta^T x) = \frac{1}{1 + e^{-\theta^T x}}$$

好了，现在**特征向量**x有了，**模型**$h_\theta(x)$也有了，**数据**也很容易得到，请人标注一批数据即可，那剩下的问题就是如何求解模型的参数θ？

如果我们选用和线性回归模型一样的平方损失作为目标函数（没有正则化），即：

$$L(\theta) = \frac{1}{2}\sum_{j=1}^{M}\left(h_\theta(x^j) - y^j\right)^2$$ （M 为样本数）

那么就会有个问题，由于$h_\theta(x)$是 sigmoid 函数，导致目标函数不是凸函数（如下图所示），那么就没有最优解，所以我们就需要做些工作使得目标函数是凸函数。

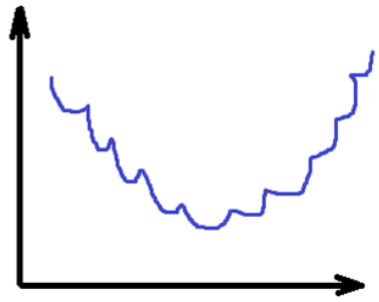


图 3.6

对于二分类问题，假设：

$$p(y=1|x,\theta)=h_\theta(x)$$

那么：

$$p(y=0|x,\theta)=1-h_\theta(x)$$

根据这两个概率，可以写出概率分布（二项分布）为：

$$p(y|x,\theta)=(h_\theta(x))^y(1-h_\theta(x))^{1-y}$$

这时就可以写出似然函数了：

$$\begin{aligned}\mathcal{L}(\theta) &= p(Y|X,\theta)\\ &=\prod_{j=1}^{M}p\left(y^j|x^j,\theta\right)\\ &=\prod_{j=1}^{M}(h_\theta(x^j))^{y^j}(1-h_\theta(x^j))^{1-y^j}\end{aligned}$$

然后就可以使用最大似然估计来求解了（也可以取负数求最小）。

（1）似然函数取 log：

$$L(\theta)=\log\mathcal{L}(\theta)=\sum_{j=1}^{M}y^j\log\left(h_\theta(x^j)\right)+(1-y^j)\log(1-h_\theta(x^j))$$

（2）对$L(\theta)$求导：

$$\frac{\partial}{\partial\theta_i}L(\theta)=\sum_{j=1}^{M}(y^j\frac{1}{g(\theta^Tx^j)}-(1-y^j)\frac{1}{1-g(\theta^Tx^j)})\frac{\partial}{\partial\theta_i}g(\theta^Tx^j)=$$

$$\sum_{j=1}^{M}\left(y^j\frac{1}{g(\theta^Tx^j)}-(1-y^j)\frac{1}{1-g(\theta^Tx^j)}\right)g(\theta^Tx^j)(1-g(\theta^Tx^j))\frac{\partial}{\partial\theta_i}\theta^Tx^j$$

$$=\sum_{j=1}^{M}(y^j(1-g(\theta^Tx^j))-(1-y^j)g(\theta^Tx^j))x_i{}^j$$

$$= \sum_{j=1}^{M} (y^j - h_\theta(x^j)) x_i{}^j$$

这里用了一个推导公式：$g'(z) = g(z)(1 - g(z))$。

（3）最后的迭代公式是（最大化，梯度上升法）：

$$\theta_i = \theta_i + \alpha \sum_{j=1}^{M} (y^j - h_\theta(x^j)) x_i{}^j$$

看到了吗？它的迭代公式形式是不和前面线性回归模型的梯度下降法的迭代公式一样呢（除了$h_\theta(x)$不同）？所以逻辑回归的求解速度也是很快的。这样就完成了这个垃圾邮件判断的任务了。

上面的求解假设是二类问题，那么如果是多类问题怎么办呢？假设有 k 个类别，即$y^j \epsilon \{1,2,..,k\}$，也就是我们希望输出每个类别下的概率，一般多分类使用的函数是 softmax 函数，即

$$p(y = c|x, \theta_c) = h_{\theta_c}(x) = s({\theta_c}^T x) = \frac{e^{-{\theta_c}^T \mathbf{x}}}{\sum_{l=1..k} e^{-{\theta_l}^T \mathbf{x}}}$$

写出似然函数：

$$\mathcal{L}(\theta) = p(Y|X, \theta)$$

$$= \prod_{j=1}^{M} \prod_{l=1}^{k} p(y^j = l | x^j, \theta_l)^{[y^j = l]}$$

对该似然函数取 log：

$$L(\theta) = \log \mathcal{L}(\theta) = \sum_{j=1}^{M} \sum_{l=1}^{k} [y^j = l] \, log p(y^j = l | x^j, \theta_l)$$

其中$[*]$表示*满足则为 1，否则为 0。

然后对$L(\theta)$求导：

$$\frac{\partial}{\partial \theta_{l,\mathrm{i}}} \mathrm{L}(\theta) = \frac{\partial \mathrm{L}(\theta)}{\partial \mathrm{p}} \frac{\partial \mathrm{p}}{\partial {\theta_{l,\mathrm{i}}}^{T} \mathbf{x}} \frac{\partial {\theta_{l,\mathrm{i}}}^{T} \mathbf{x}}{\partial \theta_{l,\mathrm{i}}} \quad \text{（推导省略）}$$

$$= \sum_{j=1}^{M} \{([\mathrm{y}^{\mathrm{j}} = l] - \mathrm{p}(\mathrm{y}^{\mathrm{j}} = l | \mathrm{x}^{\mathrm{j}}, \theta_l)) \times {\mathrm{x}_{\mathrm{i}}}^{\mathrm{j}}\}$$

$$= \sum_{j=1}^{M} \{([\mathrm{y}^{\mathrm{j}} = l] - \mathrm{h}_{\theta_l}(\mathrm{x}^{\mathrm{j}})) \times {\mathrm{x}_{\mathrm{i}}}^{\mathrm{j}}\}$$

这样就可以使用梯度上升法迭代训练了，其实逻辑回归是 Softmax 回归的特例，即 k=2 的情况。

那么逻辑回归还有什么用呢？目前谷歌、百度等各大公司都是使用逻辑回归来对广告点击率（Click-through Rate）进行预估。

搜索广告相比传统广告效果更好，就是因为它利用了用户主动的搜索意图。任何一种广告的目的一是为了赚钱，二是为了尽可能多的赚钱，简单来说，就是最大化：流量×每千次展示收益（CPM），而每千次展示收益 = 展示之后被点击的概率(CTR) × 一次点击的收入，那么提高 CTR 就自然会提高收入，也就是把最可能被用户点击的广告展示出来。而这个 CTR 就是使用逻辑回归计算出来的，一般每天产生的训练数据（线上用户真实点击与否的数据）将有上亿个，而且特征通常也会有近亿个，通常有三大类特征：流量特征（query 分词、主题、地域、覆盖率、页面特征等等）、广告特征（切词、专名、url、飘红、主题等等）和用户特征（用户兴趣、cookie、主题等等）。所以 CTR 的逻辑回归模型有亿级别特征和百亿级的训练数据，必须依靠分布式并行来搞定。

3.3 最大熵模型(Maximum Entropy Model)/条件随机场(CRF)

最大熵模型是我个人比较喜欢的模型之一，它背后的原理其实非常简单：当我们对一个随机变量的分布预测时，对已知条件一定要满足外，对未知数据一无所知时，不要做任何主观假设，要同等对待。这时，它们的概率分布最均匀，风险就越小，概率分布最均匀就意味着信息熵最大，所以就叫最大熵模型。

如果不好理解的话，我们就举个例子来说。现在有一场很激烈的篮球比赛，比分是 81:82，而且比赛时间只剩最后 3 秒，球权在落后的一方，庆幸的是，你就是领先这一队的教练，现在你的任务就是暂停之后防守住对方的最后一次进攻，不让他投进，否则就输球了，而且已知对方球队的 5 个球员（A,B,C,D,E）中，A 是超级球星，所以他进行最后一投的概率非常大；E 是个防守悍将，进攻端很差，所以他最后一投的概率非常小，他的主要目的应该是掩护使得 A 能顺利投球；其他三个都是能力相当的普通球员。那么你就有很多战术来完成这次防守，其中就包括下面两种方案：（1）让你们队中防守最好的球员去防守 A，然后让本来防守 E 的球员主要去协防 A（这样 E 几乎没人防守了），其他三个人一对一防守对方；（2）同样，让防守最好的球员去防守 A，防守 E 的球员去协防 A，剩下三个人（B,C,D）的防守同上面方案不同，你让一个球员防守 B，然后让防守 C 的球员花很大精力也去协防 B，D 是一对一防守。那么你觉得上面两个方案哪个好呢？很显然是方案（1）啊，因为方案（2）中为什么要对球员 C 减轻防守，他和球员 B、D 的能力相当啊，那么如果不是 A 最后一投的话，C 在轻防守下命中率就会

增加，那么你输球的概率就自然增加了。所以对完全未知的情况不要做任何主观假设，平等对待，风险才能最小。

好，那我们开始看看最大熵模型是怎么回事？事先我们需要介绍几个概念，假设样本集为$D = \{X, Y\}$，X 为输入，Y 为输出，比如对于文本分类问题，X 就为输入文本，Y 就是类别号；对于词性标注问题，X 就为词，Y 就是词性，等等。可以看出，在不同的问题中，输入 X 和输出 Y 比较多样化，为了模型表示方便，我们需要将输入 X 和输出 Y 表示为一些特征。对于某个(x_0, y_0)，定义特征函数：

$$f(x,y) = \begin{cases} 1: y = y_0 \text{and } x = x_0 \\ 0: \text{other} \end{cases}$$

那么特征函数的**样本期望**就可以表示为：

$$\bar{p}(f) = \sum_{x,y} \bar{p}(x,y) f(x,y)$$

而特征函数的**模型期望**表示为：

$$p(f) = \sum_{x,y} p(x,y) f(x,y) = \sum_{x,y} \bar{p}(x) p(y|x) f(x,y)$$

样本期望和模型期望到底是什么东西呢？还记得之前提到的经验风险和期望风险的区别吗？它们的区别也一样，样本期望是从样本数据中计算的，模型期望是从我们希望要求解的最优模型中计算的，而且，模型期望最终简化为条件概率$p(y|x)$，它就是我们要求解的模型，因为$\bar{p}(x)$是从样本中统计来的，根据最大似然法，数一数 x 出现的次数除以总次数就得到了，同样样本期望中的$\bar{p}(x,y)$也是数一数(x,y)同时出现的次数除以总次数就可以得到了。

机器学习就是从样本中学习到真实模型，也就是说模型期望就应该尽可能的等于从数据中观察到的样本期望，这样就出现了一个约束条件：$\bar{p}(f) = p(f)$。那么目标函数是什么呢？开头说了：要使熵（条件熵）最大。

那么目标函数就是：$\text{argmax}_p H(p) = -\sum_{x,y} \bar{p}(x)p(y|x)\log p(y|x)$

所以最大熵模型就是（$p(y|x)$是变量）：

$$\text{argmax}_p H(p) = -\sum_{x,y} \bar{p}(x)p(y|x)\log p(y|x)$$

$$\text{s.t.} \begin{cases} \forall f_i (i = 1..k) \sum_{x,y} \bar{p}(x,y) f_i(x,y) = \sum_{x,y} \bar{p}(x)p(y|x)f_i(x,y) \\ \forall x \sum_{y} p(y|x) = 1 \end{cases}$$

可以把最大化转换为最小化问题，即：

$$\text{argmin}_p -H(p) = \sum_{x,y} \bar{p}(x)p(y|x)\log p(y|x)$$

$$\text{s.t.} \begin{cases} \forall f_i (i = 1..k) \sum_{x,y} \bar{p}(x,y) f_i(x,y) = \sum_{x,y} \bar{p}(x)p(y|x)f_i(x,y) \\ \forall x \sum_{y} p(y|x) = 1 \end{cases}$$

OK，模型构建好了，那剩下的就是如何求解模型了，这是个等式约束优化，就可以用拉格朗日乘子法来求解，写出拉格朗日函数：

$$L(p,\alpha) = -H(p) + \sum_{i=1..k} \alpha_i \left(\sum_{x,y} \bar{p}(x,y) f_i(x,y) - \sum_{x,y} \bar{p}(x)p(y|x)f_i(x,y) \right) + \alpha_0 \left(\sum_{y} p(y|x) - 1 \right)$$

不幸的是，直接对参数求导使它们等于零，没法求解出各个参数，因为参数互相耦合到一起了，所以就要尝试其他方法求解了。

还记得前面讲的对偶原理吗？原问题是：

$$\min_p \max_\alpha L(p,\alpha)$$

对偶问题是：

$$\max_\alpha \min_p L(p,\alpha)$$

由于$L(p,\alpha)$是凸函数，所以原始问题和对偶问题是等价的。这样我们就求解对偶问题，首先求解$\min_p L(p,\alpha)$。

对 p 求导，求解$\frac{\partial}{\partial p}L(p,\alpha)=0$，即：

$$\frac{\partial}{\partial p}L(p,\alpha)=\sum_{x,y}\bar{p}(x)(\log p(y|x)+1)-\sum_{x,y}\sum_{i=1..k}\alpha_i\bar{p}(x)f_i(x,y)+\sum_y\alpha_0$$

$$=\sum_{x,y}\bar{p}(x)[\log p(y|x)+1-\sum_{i=1..k}\alpha_i f_i(x,y)+\alpha_0]$$

解得：

$$p^*(y|x)=e^{\sum_i\alpha_i f_i(x,y)-\alpha_0-1}=Z(x)e^{\sum_i\alpha_i f_i(x,y)}$$

又：$\forall x\ \sum_y p(y|x)=1$，那么

$$\sum_y p^*(y|x)=\sum_y Z(x)e^{\sum_i\alpha_i f_i(x,y)}=1$$

得到：$Z(x)=\frac{1}{\sum_y e^{\sum_i\alpha_i f_i(x,y)}}$

总结一下，最大熵模型的条件概率分布为：

$$p(y|x)=Z(x)e^{\sum_i\alpha_i f_i(x,y)}$$

$$Z(x)=\frac{1}{\sum_y e^{\sum_i\alpha_i f_i(x,y)}}$$

这样$\min_p L(p,\alpha)$的最优解$p^*(y|x)$就求解出来了，剩下就是求：

$$\max_\alpha \min_p L(p,\alpha)=\max_\alpha p^*(y|x)$$

它是α的函数，只要α求解出来了，最大熵模型$p(y|x)$也就求解出来了，但是很显然，α的解析解无法直接求出（有指数函数），那么就需要数值方法来求解了，比如：GIS，IIS，LBFGS 等方法。

在这简单介绍下 IIS 求解的流程（里面有很多证明，请参考论文《The Improved Iterative Scaling Algorithm: A Gentle Introduction》），我们现在已经知道了条件概率$p(y|x)$的表达式了，那么就可以用最大似然估计（最大熵和最大似然估计训练结果一致，只是考虑的方式不同：最大熵是以样本数据的熵值最大化为目标；最大似然估计是以样本数据的概率值最大化作为目标，既然训练结果一致，那就找个简单的求解），写出它的 log 似然函数：

$$L'(\alpha) = \sum_{x,y} \bar{p}(x,y) \log p(y|x)$$

$p(y|x)$我们已经求解出来了，是α的函数，那么$L'(\alpha)$自然也是α的函数了，但是如果用$L'(\alpha)$对 α求导的话，也是无法解出α的，那也就需要迭代法：$\alpha = \alpha + \delta$，满足$L'(\alpha + \delta) \geq L'(\alpha)$，也就是每步迭代都要向最优解移动，要想移动的快，也就是要使$L'(\alpha + \delta) - L'(\alpha)$尽可能大，也就是最大化$L'(\alpha + \delta) - L'(\alpha)$即可（它是$\delta_i$的函数），如果$L'(\alpha + \delta) - L'(\alpha)$无法表示出来，那就找个下限$L'(\alpha + \delta) - L'(\alpha) \geq \varphi(\delta)$，然后就是每次迭代求解$\varphi(\delta)$的最大值了。

IIS 的算法流程如下：

（1）设$\alpha_i = 0$，$i = 1..k$。

（2）for i=1 to k:

2.1 $\frac{\partial}{\partial \delta} \varphi(\delta_i) = \sum_{x,y} \bar{p}(x,y) f_i(x,y) - \sum_x \bar{p}(x) \sum_y p(y|x) f_i(x,y) e^{\delta_i f^{\#}(x,y)}$

2.2 令$\frac{\partial}{\partial \delta} \varphi(\delta_i) = 0$，解出$\delta_i$。（其中$f^{\#}(x,y) = \sum_{i=1..k} f_i(x,y)$）

2.3 $\alpha_i = \alpha_i + \delta_i$

至此，最大熵模型介绍完了，然后简单看下条件随机场（CRF）是什么东西。大家已经知道最大熵模型的条件概率分布为：

$$p(y|x) = Z(x)\exp(\sum_i \alpha_i f_i(x,y))$$

$$Z(x) = \frac{1}{\sum_y \exp(\sum_i \alpha_i f_i(x,y))}$$

而且，它的概率图，如下：

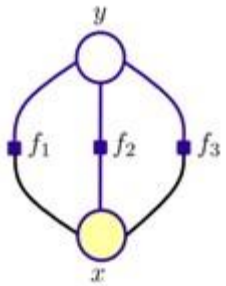


图 3.7

而 CRF 的条件概率分布是：

$$p(y|x) = Z(x)\exp(\sum_i \alpha_i \sum_{t=1..N} f_i(x_t, y_t, y_{t-1}))$$

$$Z(x) = \frac{1}{\sum_y \exp(\sum_i \alpha_i \sum_{t=1..N} f_i(x_t, y_t, y_{t-1}))}$$

而且，它的概率图如下：

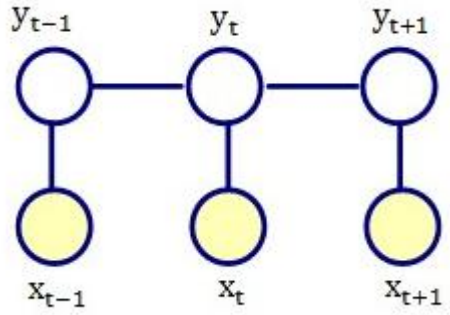


图 3.8

从概率图大致可以看出区别了，CRF 利用了上下文信息，而且最后的条件概率也是全局最优（无标记偏置问题），所以它对标注问题（分词、词性标注、实体识别等）效果更好一点，至于 CRF 的细节大家可以参考一下论文《Conditional Random Fields: An Introduction》，它的参数求解过程和最大熵很类似：最大似然估计，然后使用 IIS，LBFGS 等算法求解参数，《Discriminative training methods for hidden Markov models: Theory and experiments with perceptron algorithms》这篇文章提出了一种更简单的训练方法。

3.4 主题模型(Topic Model)

主题模型（Topic Model）可是自然语言处理（NLP）中非常有影响力的模型之一，因为它的初衷就是解决 NLP 中的语义问题，尽管离真正意义上的语义有一定距离。那么什么是“主题”呢？一段话的主题就是小学时学到的中心思想，但是这个中心思想太泛了，计算机要怎么表示呢？那就是用一些最能反映中心思想的词来表示（这就产生了一个假设：bag of words，这个假设是指一个文档被表示为一堆单词的无序组合，不考虑语法、词序等，现在 bag of words 假设其实也是 NLP 任务的一个基本假设），比如下面一段新闻内容：“*站在互联网整个行业的角度，我认为微信是一个极具生命力和想象空间的移动产品，它的布局和设计都有可能颠覆移动互联网的明天。*”我们可以看出，它的主题应该就是“微信”、“移动互联网”等。所以，对于一

篇文档，我们希望能得到它的主题（一些词），以及这些词属于哪些主题的概率，这样我们就可以进一步分析文档了。

主题模型有不少算法，最经典的两个是：PLSA（Probabilistic Latent Semantic Analysis）和 LDA（Latent Dirichlet Allocation）。

首先来看下 PLSA 模型。$d \in D$表示文档，$w \in V$表示词语，z 表示隐含的主题。$P(d_i)$表示单词在文档d_i中出现的概率，$P(z_k|d_i)$表示主题z_k在给定文档d_i下出现的概率，$P(w_j|z_k)$表示单词w_j在给定主题z_k下出现的概率。PLSA 是个典型的生成模型，根据下方的图模型可以写出文档中每个词的生成概率：

$$P(d_i, w_j) = P(d_i)P(w_j|d_i) = P(d_i)\sum_{k=1..K} P(w_j|z_k)p(z_k|d_i)$$

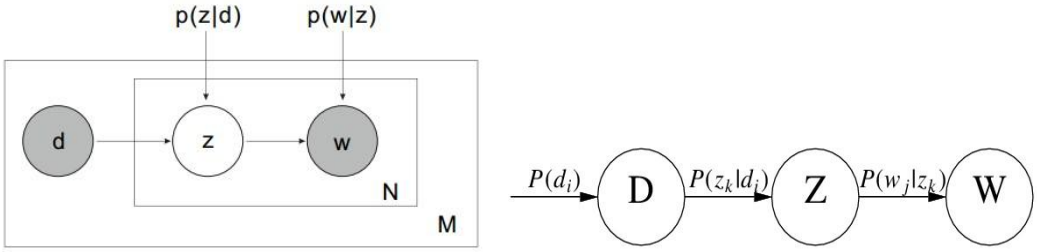


图 3.9（两种不同的表示方法）

由于词和词之间是互相独立的，文档和文档间也是相互独立的，那么整个样本集的分布为：

$$P(D, V) = \prod_{i=1..N}\prod_{j=1..M} P(d_i, w_j)^{n(w_j, d_i)}$$

首先，我们看看，PLSA 模型待估计的参数是什么？就是$\theta = \{P(w_j|z_k), P(z_k|d_i)\}$，那么样本集的 log 似然函数就为：

$$L(\theta) = \log P(D, V|\theta) = \sum_{i=1..M}\sum_{j=1..N} n(w_j, d_i)\log P(d_i, w_j)$$

其中，$n(w,d)$表示单词 w 在文档 d 中出现的次数，$n(d)$表示文档 d 中词的个数。

好，现在有了 log 似然函数，那么就可以对其求导解出参数了，我们知道我们的参数θ共有$(N \times K + M \times K)$个，而且自变量是包含在对数和中，这就意味着这个方程组的求解很困难，那就要考虑其他方法求解了，而且我们这个问题还包含隐藏变量，就要使用 EM 算法。

EM 算法就是根据已经观察到的变量对隐藏变量进行学习的方法。既然没办法最大化$L(\theta)$，那么$L(\theta)$总该有个下限吧，我们就优化这个下限，不断迭代提高这个下限，那么就可以得到近似最大解了，这个下限其实就是似然函数的期望。通常，EM 算法得到的是局部近似解。

首先对参数$P(w_j|z_k)$和 $P(z_k|d_i)$赋值随机值。

EM 算法第一步 E-step：求隐藏变量的后验概率：

$$P(z_k|d_i,w_j) = \frac{P(w_j|z_k)P(z_k|d_i)}{\sum_{k=1..K} P(w_j|z_k)P(z_k|d_i)}$$

EM 算法第二步 M-step：最大化似然函数的下限，解出新的参数。

那么，现在的问题就是找到$L(\theta)$的下限，可以推导出：

$$L(\theta) = \sum_{i=1..M}\sum_{j=1..N} n(w_j,d_i)\log \sum_{k=1..K} P(z_k|d_i)\,P(w_j|z_k)$$

$$\geq \sum_{i=1..M}\sum_{j=1..N} n(w_j,d_i) \sum_{k=1..K} P(z_k|d_i,w_j)\log(P(z_k|d_i)P(w_j|z_k)) = Q(\theta)$$

$Q(\theta)$就是下限，那么最大化它就可以了，但是还有约束条件：

$$\sum_{j=1..N} P(w_j|z_k) = 1$$

$$\sum_{k=1..K} P(z_k|d_i) = 1$$

那么这时写出拉格朗日函数：

$$H = \sum_{i=1..M} \sum_{j=1..N} n(w_j, d_i) \sum_{k=1..K} P(z_k|d_i, w_j) \log\left(P(z_k|d_i)P(w_j|z_k)\right) + \sum_{k=1..K} \tau_k\left(\sum_{j=1..N} P(w_j|z_k) - 1\right) + \sum_{i=1..M} \rho_i\left(\sum_{k=1..K} P(z_k|d_i) - 1\right)$$

然后分别对$P(w_j|z_k)$和$P(z_k|d_i)$求导（注意：$P(z_k|d_i, w_j)$是已知的，在 E-step 已经计算好了），然后联合约束条件，解得：

$$P(w_j|z_k) = \frac{\sum_{i=1..M} n(w_j, d_i)P(z_k|d_i, w_j)}{\sum_{j=1..N}\sum_{i=1..M} n(w_j, d_i)P(z_k|d_i, w_j)}$$

$$P(z_k|d_i) = \frac{\sum_{j=1..N} n(w_j, d_i)P(z_k|d_i, w_j)}{n(d_i)}$$

然后把$P(w_j|z_k)$和$P(z_k|d_i)$代入到 E-step 接着迭代。

在这儿，也总结下这个经典的 EM 算法，假设 log 似然函数为$L(\theta) = \log\sum_Z P(X, Z|\theta)$，那么 EM 算法步骤为：

（1）随机初始化θ_i。

（2）EM 步骤：

while $(|\theta_{i+1} - \theta_i| < \delta)$:

E-step：计算后验概率$P(Z|X, \theta_i)$。

M-step：$\theta_{i+1} = \text{argmax}_\theta \sum_Z P(Z|X, \theta_i)\log P(X, Z|\theta)$

至此，PLSA 算法就求解完了，它和其他算法的求解过程其实没多大区别，唯一的是多了隐藏变量，然后使用 EM 算法求解。

PLSA 求出了所有$P(z_k|d_i)$和$P(w_j|z_k)$，也就是文档d_i属于主题z_k的概率和主题z_k下各个单词w_j的概率，但是 PLSA 并没有考虑参数的先验知识，这时候出现了另一个改进的算法：LDA，它对参数增加了先验分布（所以理论上 LDA 比 PLSA 不容易过拟合），也就是说参数也是一个分布，这个分布的参数（参数的参数）叫做超参数。

LDA 是个挺复杂的模型，我们简单看下它的大致思路，下图就是从论文中截取的 LDA 的图模型。

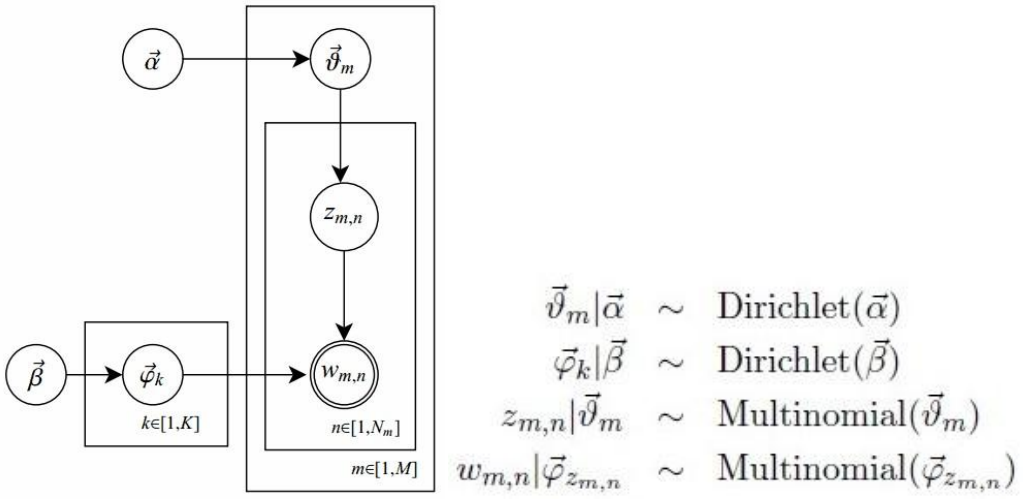


图 3.10

上图中，$w_{m,n}$是观察到的变量（文档 m 中第 n 个词），其他都是参数或者隐藏变量，$\vec{\alpha}$，$\vec{\beta}$就是超参数。$\phi = \{\overrightarrow{\varphi_k}\}$表示主题和词之间的分布，是一个$M \times K$的矩阵，$\Theta = \{\overrightarrow{\vartheta_m}\}$表示文档和主题之间的分布，是一个$K \times V$的矩阵。M 文档数，K 主题数，V 词数，$N_m$文档 m 的长度。

LDA 有个假设：$\vec{\alpha}$->$\overrightarrow{\vartheta_m}$服从 Dirichlet 分布，$\overrightarrow{\vartheta_m}$->$z_{m,n}$服从 Multinomial 分布，$\vec{\beta}$->$\overrightarrow{\varphi_k}$服从 Dirichlet 分布，$\overrightarrow{\varphi_k}$->$w_{m,n}$服从 Multinomial 分布，大家知道 Dirichlet 分布和 Multinomial 分布是一对共轭分布，那么为什么 LDA 要假设共轭分布呢？当某个似然概率异常复杂时，后验概率计算会叫人非常头大，使用共轭先验可以简化问题！而且 Dirichlet 分布和

Multinomial 分布都研究透了，它们的概率分布啊，期望啊等都是可以直接求解出来的，在推导最终参数时就可以直接使用了。

假设某个分布（观察变量为X），要估计其中的参数θ。参数θ有个先验分布$p(\theta)$，贝叶斯法则告诉我们$p(\theta|X) \propto p(X|\theta)p(\theta)$, 可以知道，若$p(\theta)$与$p(X|\theta)$有相同的函数形式，那么后验概率$p(\theta|X)$和它们($p(\theta)$与$p(X|\theta)$)也有相同的函数形式，这样使得θ的后验概率与先验概率具有相同的表达式，只是参数不同而已！所以选择与似然函数共轭的先验，得到的后验函数只是参数调整后的先验函数。

同样，可以写出 LDA 的似然函数，然后使用最大似然估计求解参数，但是似然函数参数耦合太大，无法求解出来，而且包括隐藏变量，所以就像 PLSA 一样，想到了 EM 算法，但是 LDA 相比 PLSA 还有一个困难：后验概率无法求解出来（EM 算法中 E-step 要求解后验概率），那么这时就又要近似计算了：变分-EM 算法。

变分推理是一种近似计算后验概率的方法，首先寻找一个和原来不能直接求解的后验概率等价或者近似的函数 Q，然后就通过求解最优近似函数 Q 的参数来近似得到原后验概率的参数。

这样变分-EM 算法迭代的流程如下：

（1）设定参数的初始值；

（2）E-step：计算近似函数 Q；

（3）M-step：最大化近似函数 Q，解出参数；

求解 LDA 的参数还有一种方法：Gibbs Sampling。

Gibbs Sampling 算法是 MCMC 的一个特例，如果某个概率 P(X)不易求得，那么可以交替的固定某一维度x_i，然后通过其他维度$x_{\neg i}$（去除x_i的其他所有值）的值来抽样近似求解，也就是说，Gibbs 采样就是用条件分布的采样来替代全概率分布的采样。

回到 LDA，使用 Gibbs Sampling，就是要迭代求解：

$$P(z_{m,n} = k|\vec{w}, z_{\neg(m,n)}, \vec{\alpha}, \vec{\beta})$$

这个公式要利用联合概率$P(\vec{w}, \vec{z}|\vec{\alpha}, \vec{\beta})$，而联合概率的计算其实就是利用 Dirichlet 分布和 Multinomial 分布推导出来的，这两个分布是已知的，所以$P(z_{m,n} = k|\vec{w}, z_{\neg(m,n)}, \vec{\alpha}, \vec{\beta})$自然可以求解出来了。

当 Gibbs Sampling 收敛后，根据图模型就可以求解后验分布了：$P(\overrightarrow{\vartheta_m}|\overrightarrow{z_m}, \vec{\alpha})$和$P(\overrightarrow{\varphi_k}|\vec{z}, \vec{\alpha}, \vec{\beta})$（共轭分布的性质：它们就和先验分布一样，只是参数不同，所以可以求解出来），然后使用它们的期望就可以解出所有$\overrightarrow{\varphi_k}$和$\overrightarrow{\vartheta_m}$了。

所以整个基于 Gibbs Sampling 的 LDA 算法流程为：

（1）初始化参数。

（2）对所有文档 m=1..M

对文档 m 中所有的单词 n=1..N_m

-采样每个单词$w_{m,n}$对应的主题$z_{m,n}$=k；

-对单词$w_{m,n}$的主题 k 增加计数（“文档-主题”计数，“文档-主题”总数，“主题-单词”计数，“主题-单词”总数）；

（3）迭代步骤：

对所有文档 m=1..M

对文档 m 中所有的单词 n=1..N_m

-对单词$w_{m,n}$的主题 k 减少计数；

-根据$P(z_{m,n} = \tilde{k}|\vec{w}, z_{\neg(m,n)}, \vec{\alpha}, \vec{\beta})$采样新主题；

-对单词$w_{m,n}$的新主题$\tilde{k}$增加计数；

收敛后，利用公式计算所有$\overrightarrow{\varphi_k}$和$\overrightarrow{\vartheta_m}$；

至此 LDA 就求解出来了，如果想深入了解 LDA 的每个细节：下面的文章不可不读（当然还有其他文章，不一一列举了）：

《Variational Message Passing and its Applications》

《Latent Dirichlet Allocation》

《Parameter estimation for text analysis》

《The Expectation Maximization Algorithm A short tutorial》

《Gibbs Sampling in the Generative Model of Latent Dirichlet Allocation》

乍一看 LDA 很难理解，但是从最终推导的公式来看，代码还是很清晰的，下面就是 GibbsSampling 的核心代码，其实还是很容易理解的：

```
// --- model parameters and variables ---
int M; // dataset size (i.e., number of docs)
int V; // vocabulary size
int K; // number of topics
double alpha, beta; // LDA hyperparameters
int niters; // number of Gibbs sampling iterations
double * p; // temp variable for sampling
int ** z; // topic assignments for words, size M x doc.size()
int ** nw; // cwt[i][j]: number of instances of word/term i assigned to topic j, size V x K
int ** nd; // na[i][j]: number of words in document i assigned to topic j, size M x K
int * nwsum; // nwsum[j]: total number of words assigned to topic j, size K
int * ndsum; // nasum[i]: total number of words in document i, size M
double ** theta; // theta: document-topic distributions, size M x K
double ** phi; // phi: topic-word distributions, size K x V
int model::init_est()
{
//...
	srandom(time(0)); // initialize for random number generation
	z = newint*[M];
	for (m = 0; m < ptrndata->M; m++)
	{
		int N = ptrndata->docs[m]->length;
		z[m] = newint[N];
```

```
            // initialize for z
            for (n = 0; n < N; n++)
            {
                int topic = (int)(((double)random() / RAND_MAX) * K);
                z[m][n] = topic;

                // number of instances of word i assigned to topic j
                nw[ptrndata->docs[m]->words[n]][topic] += 1;
                // number of words in document i assigned to topic j
                nd[m][topic] += 1;
                // total number of words assigned to topic j
                nwsum[topic] += 1;
            }
            // total number of words in document i
            ndsum[m] = N;
        }
//...
        return 0;
}
void model::estimate()
{
        printf( "Sampling %d iterations!\n" , niters);

        int last_iter = liter;
        for (liter = last_iter + 1; liter <= niters + last_iter; liter++)
        {
            printf( "Iteration %d ...\n", liter);

            // for all z_i
            for (int m = 0; m < M; m++)
            {
                for (int n = 0; n < ptrndata->docs[m]->length; n++)
                {
                    // (z_i = z[m][n])
                    // sample from p(z_i|z_-i, w)
                    int topic = sampling(m, n);
                    z[m][n] = topic;
                }
            }

            if (savestep > 0)
            {
                if (liter % savestep == 0)
                {
                    // saving the model
                    printf( "Saving the model at iteration %d ...\n" , liter);
                    compute_theta();
                    compute_phi();
                    save_model(utils::generate_model_name(liter));
                }
            }
        }//end for

        printf( "Gibbs sampling completed!\n" );
        printf( "Saving the final model!\n" );
        compute_theta();
        compute_phi();
        liter--;
        save_model(utils::generate_model_name(-1));
}
int model::sampling(int m, int n)
{
        // remove z_i from the count variables
        int topic = z[m][n];
        int w = ptrndata->docs[m]->words[n];
        nw[w][topic] -= 1;
        nd[m][topic] -= 1;
```

```
        nwsum[topic] -= 1;
        ndsum[m] -= 1;

        double Vbeta = V * beta;
        double Kalpha = K * alpha;

        // do multinomial sampling via cumulative method
        for (int k = 0; k < K; k++)
        {
                p[k] = (nw[w][k] + beta) / (nwsum[k] + Vbeta) *
                        (nd[m][k] + alpha) / (ndsum[m] + Kalpha);
        }

        // cumulate multinomial parameters
        for (int k = 1; k < K; k++)
        {
                p[k] += p[k - 1];
        }

        // scaled sample because of unnormalized p[]
        double u = ((double )random() / RAND_MAX) * p[K - 1];

        for (topic = 0; topic < K; topic++)
        {
                if (p[topic] > u)
                        break;
        }

        // add newly estimated z_i to count variables
        nw[w][topic] += 1;
        nd[m][topic] += 1;
        nwsum[topic] += 1;
        ndsum[m] += 1;

        return topic;
}
void model::compute_theta()
{
        for (int m = 0; m < M; m++) {
                for (int k = 0; k < K; k++) {
                        theta[m][k] = (nd[m][k] + alpha) / (ndsum[m] + K * alpha);
                }
        }
}
void model::compute_phi()
{
        for (int k = 0; k < K; k++) {
                for (int w = 0; w < V; w++) {
                        phi[k][w] = (nw[w][k] + beta) / (nwsum[k] + V * beta);
                }
        }
}
// ------    end -------
```

3.5 深度学习(Deep Learning)

深度学习（Deep Leaning）是 Hinton 于 2006 年提出来的，然而在 2012 年开始，它变的异常火爆，几乎到处都能听见 deep learning，但

是它本质上还是一个机器学习模型，只是解决问题的思路有所不同。之前我们讲过，传统的机器学习需要提取特征，然后建立模型学习，但是特征是人工提取，如果不需要人工参与，那该多好啊，深度学习就是解决这个问题，所以它也叫无监督特征学习。

前面说过，机器学习的过程其实就是模拟人学习的过程，那么人最聪明的地方是大脑，如果机器能模拟大脑，那势必会更聪明些。20 世纪 80 年代末的神经网络就是模拟人脑的一个学习模型，最经典的就是 BP 神经网络，它有三层网络模型（输入层，隐藏层，输出层），如图 3.11 所示，它的学习算法非常重要，所以我们有必要介绍一下这个网络，它的公式如下，这也就是正向传播：

$$z^2 = W^1 x + b^1$$

$$a^2 = f(z^2)$$

$$z^{l+1} = W^l a^l + b^l$$

$$a^{l+1} = f(z^{l+1})$$

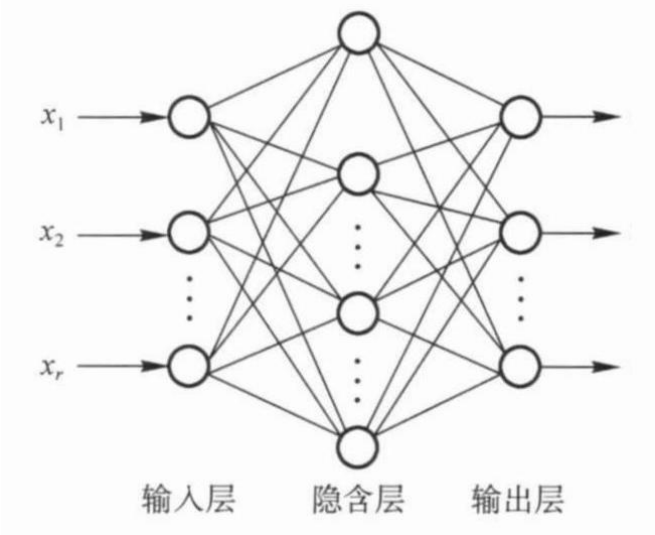


图 3.11

其中，f 是激励函数，常用的激励函数有 sigmoid 函数，tanh 函数等，有的输出层使用 softmax 函数（见逻辑回归章节）。那么怎么求解其中的参数（每一层的 W 和 b）呢？使用梯度下降法来迭代，即：

$$W^l = W^l - \alpha \frac{\delta L(W,b)}{\delta W^l}$$

$$\mathrm{b}^l = \mathrm{b}^l - \alpha \frac{\delta L(W,b)}{\delta \mathrm{b}^l}$$

其中$L(W,b)$就是损失函数，前面已经讲过，有平方损失，似然损失等。

$$\frac{\delta L(W,b)}{\delta W^l} = \frac{\delta L(W,b)}{\delta z^{l+1}} \frac{\delta z^{l+1}}{\delta W^l} = \frac{\delta L(W,b)}{\delta z^{l+1}} a^l$$

$$\frac{\delta L(W,b)}{\delta \mathrm{b}^l} = \frac{\delta L(W,b)}{\delta z^{l+1}} \frac{\delta z^{l+1}}{\delta \mathrm{b}^l} = \frac{\delta L(W,b)}{\delta z^{l+1}}$$

剩下的问题就是求解$\delta L(W,b)/\delta z^{l+1}$了，令$\sigma^l = \delta L(W,b)/\delta z^l$

对于输出层：

$$\sigma^n = \frac{\delta L(W,b)}{\delta z^n} = \frac{\delta L(W,b)}{\delta f} \mathrm{f}'(z^n)$$

对于每个隐藏层：

$$\sigma^l = \frac{\delta L(W,b)}{\delta z^l} = \frac{\delta L(W,b)}{\delta z^{l+1}} \frac{\delta z^{l+1}}{\delta a^l} \frac{\delta a^l}{\delta z^l} = (W^l)^T \sigma^{l+1} \cdot \mathrm{f}'(z^l)$$

这样就可以求解出所有参数了，上面我的推导是使用的矩阵形式，下面我们从代码里来具体看这个算法的细节：

```
//bp.h
#ifndef _BP_H_
#define _BP_H_
#include <time.h>
#include <stdlib.h>
#include <stdio.h>
#include <math.h>

class CBackProp
{
private:
```

```
    double **m_delta;        //每个单元的误差
    double ***m_weight;    //权重W和b
    int m_layer_number;     //网络的层数
    int *m_layer_size;       //每层网络的节点数
    double m_beta;            //学习速率
    double m_alpha;           //冲量率
    double **m_out;           //每个单元的输出
    double ***m_prevDWt; //前一次迭代的权重更值

public:
    ~CBackProp();
    CBackProp(int layer_number, int *layer_size, double beta, double alpha);

    //sigmoid函数
    double Sigmoid(double in) const { return (double)(1/(1+exp(-in))); }
    //sigmoid函数导数
    double SigmoidDerivative(double gx) const { return gx *(1 - gx); }

    //损失函数
    double MSE(double *target) const;
    //损失函数导数
    double MSEDerivative(double *gx, int i) const;

    //BP训练
    void BackProp(double *in, double *target);
    //前向传播
    void FeedForward(double *in);

    //获得第i个输出
    double GetOut(int i) const{ return m_out[m_layer_number-1][i]; }
};
#endif

//bp.cpp
#include "bp.h"
CBackProp::CBackProp(int layer_number, int *layer_size, double beta, double alpha) : m_beta(beta),m_alpha(alpha)
{
    m_layer_number = layer_number;
    m_layer_size = new int[m_layer_number];
    for(int i = 0; i < m_layer_number; i++){
        m_layer_size[i] = layer_size[i];
    }

    m_out = new double*[m_layer_number];
    for(int i = 0; i < m_layer_number; i++){
        m_out[i] = new double[m_layer_size[i]];
    }

    m_delta = new double*[m_layer_number];
    for(int i = 1; i < m_layer_number; i++){
        m_delta[i] = new double[m_layer_size[i]];
    }

    m_weight = new double**[m_layer_number];
    for(int i = 1;i < m_layer_number; i++){
        m_weight[i] = new double*[m_layer_size[i]];
    }
    for(int i = 1;i < m_layer_number; i++){
        for(int j = 0;j < m_layer_size[i]; j++){
            m_weight[i][j] = new double[m_layer_size[i-1]+1];
        }
    }

    m_prevDWt = new double**[m_layer_number];
    for(int i = 1;i < m_layer_number; i++){
        m_prevDWt[i] = new double*[m_layer_size[i]];
    }
```

```
    for(int i = 1; i < m_layer_number; i++){
        for(int j = 0;j < m_layer_size[i]; j++){
            m_prevDWt[i][j] = new double[m_layer_size[i-1]+1];
        }
    }

    for(int i = 1;i < m_layer_number; i++)
        for(int j = 0;j < m_layer_size[i]; j++)
            for(int k = 0; k < m_layer_size[i-1]+1; k++)
                m_prevDWt[i][j][k] = (double)0.0;

    //初始化随机数
    srand((unsigned)(time(NULL)));
    for(int i = 1;i < m_layer_number; i++)
        for(int j = 0;j < m_layer_size[i]; j++)
            for(int k = 0; k < m_layer_size[i-1]+1; k++)
                m_weight[i][j][k] = (double)(rand())/(RAND_MAX/2) - 1;
}

CBackProp::~CBackProp()
{
    for(int i = 0;i < m_layer_number; i++)
        delete[] m_out[i];
    delete[] m_out;

    for(int i = 1;i < m_layer_number; i++)
        delete[] m_delta[i];
    delete[] m_delta;

    for(int i = 1; i < m_layer_number; i++)
        for(int j = 0; j < m_layer_size[i]; j++)
            delete[] m_weight[i][j];
    for(int i = 1; i < m_layer_number; i++)
        delete[] m_weight[i];
    delete[] m_weight;

    for(int i = 1;i < m_layer_number; i++)
        for(int j = 0;j < m_layer_size[i]; j++)
            delete[] m_prevDWt[i][j];
    for(int i = 1;i < m_layer_number; i++)
        delete[] m_prevDWt[i];
    delete[] m_prevDWt;

    delete[] m_layer_size;
}

double CBackProp::MSE(double *target) const
{
    double mse = 0.0;
    for(int i = 0; i < m_layer_size[m_layer_number-1]; i++)
        mse += (target[i]-m_out[m_layer_number-1][i]) * (target[i]-m_out[m_layer_number-1][i]);
    return mse / 2.0;
}
double CBackProp::MSEDerivative(double *gx, int i) const
{
    return gx[i] - m_out[m_layer_number-1][i];
}

void CBackProp::FeedForward(double *in)
{
    double sum = 0.0;

    //m_out[0]存输入
    for(int i = 0; i < m_layer_size[0]; i++)
        m_out[0][i] = in[i];

    for(int i = 1; i < m_layer_number; i++){
```

```
            for(int j = 0; j < m_layer_size[i]; j++){
                sum = 0.0;
                for(int k = 0; k < m_layer_size[i-1]; k++){
                    sum += m_out[i-1][k]*m_weight[i][j][k];
                }
                sum += m_weight[i][j][m_layer_size[i-1]]; // + b
                m_out[i][j] = Sigmoid(sum);
            }
        }
}

void CBackProp::BackProp(double *in, double *target)
{
    double sum = 0.0;

    //更新out
    FeedForward(in);

    //计算输出层的delta
    for(int i = 0;i < m_layer_size[m_layer_number-1]; i++){
        m_delta[m_layer_number-1][i] = SigmoidDerivative(m_out[m_layer_number-1][i]) *
MSEDerivative(target, i);
    }

    //计算隐藏层的delta
    for(int i = m_layer_number-2; i > 0; i--){
        for(int j = 0; j < m_layer_size[i]; j++){
            sum = 0.0;
            for(int k = 0; k < m_layer_size[i+1]; k++){
                sum += m_delta[i+1][k] * m_weight[i+1][k][j];
            }
            m_delta[i][j] = SigmoidDerivative(m_out[i][j]) * sum;
        }
    }

    //冲量
    for(int i = 1; i < m_layer_number; i++){
        for(int j = 0; j < m_layer_size[i]; j++){
            for(int k = 0; k < m_layer_size[i-1]; k++){
                m_weight[i][j][k] += m_alpha * m_prevDWt[i][j][k];
            }
            m_weight[i][j][m_layer_size[i-1]] += m_alpha * m_prevDWt[i][j][m_layer_size[i-1]];
        }
    }

    //梯度下降调整权重
    for(int i = 1; i < m_layer_number; i++){
        for(int j = 0; j < m_layer_size[i]; j++){
            for(int k = 0; k < m_layer_size[i-1]; k++){
                m_prevDWt[i][j][k] = m_beta * m_delta[i][j] * m_out[i-1][k];
                m_weight[i][j][k] += m_prevDWt[i][j][k];     //更新W
            }
            m_prevDWt[i][j][m_layer_size[i-1]] = m_beta * m_delta[i][j];
            m_weight[i][j][m_layer_size[i-1]] += m_prevDWt[i][j][m_layer_size[i-1]]; //更新b
        }
    }
}
```

这种浅层网络它有很多缺点：初始值的选取是随机的，很容易收敛到局部最优解，导致过拟合；如果增加隐藏层的话，就会使传递到

前面的梯度越来越稀疏，收敛速度很慢；它没有利用海量的未标注数据。

所以为了克服浅层神经网络的训练缺陷，深度学习是在海量数据中采用贪婪式的逐层学习方法：首先是无监督训练，单独训练一层，然后把该层的输出作为下一层的输入，使用相同的方法一直向上训练；然后到最上层是个有监督的从上到下的微调。所以一般来说使用深度学习有两个框架：

（1）无监督学习+有监督学习：首先从大量未标注数据中无监督逐层学习特征，然后把学习到的特征放到传统的监督学习方法来学习模型。

（2）有监督学习：首先逐层学习，到网络最上层是一个分类器（例如：softmax 分类器），整个是一套深层网络模型。

常用的深度学习模型有：

Auto-Encoders（自动编码器）

Sparse Coding（稀疏编码）

RBMs（限制玻尔兹曼机）

DBNs（深信念网络）

CNN（卷积深度网络）

RNN（递归神经网络）

深度学习的优势在于海量训练数据和好的训练模型（个人认为海量数据起的作用更大），既然有海量训练数据，那么对计算性能就会有很高的要求，一般都是使用 GPU 来训练深度学习模型。

我们看看深度学习在文本上是否有新颖的应用。我们前面提到过，在自然语言处理中有个假设：bag of words，对于一篇文档，它由很多词组成，如果把字典中所有词按照字母序排成一个很长的向量，那么每篇文档就可以使用这个长向量来表示了，某个词出现了，就标记为 1，没有出现的词标记为 0，可以想象，每个文档将会是一个稀疏向量。把文档表示成向量就会有很多缺点，比如：“电脑”和“计算机”有很大的语义相似性，但是表示成向量它们就没有语义关系了，因为它们在词典中的不同位置，不管用什么方法（cos）都没法计算出很好的相似性，也就是说把词表示成一个槽位是不合理的，那么我们可不可以把词表示出更丰富的信息呢？这就是词表示（Word Embedding）。

词表示是一个很好的模型，而且个人认为它也是目前 NLP 方面取得最大的进展（其他的尝试底层的输入几乎都是词向量），它相当于把词“表示”了更丰富的信息，这样就有可能看到 bag of word 看不到的信息，举个例子，假如让你计算一下 375+24，我想你一定很快就口算出来了，但是，如果让你计算 375*24，我想你就不能很快的口算出来了，我如果把这两个数“表示”成另一种形式：375=3*5*5*5；24=3*2*2*2，那么 375*24 = (3*5*5*5) *(3*2*2*2)，这样是不就可以很快口算出来了。这就是“表示”的好处。

对于词的表示就是想办法把词表示成了一个向量，既然表示成向量了，那么两个词之间就可以计算相似性了（cos）。词表示都是通过神经语言模型来训练得到的，目前大致有两种框架：NNLM 和 RNNLM。

NNLM（Neural Net Language Model）最经典的当属 Bengio 于 2001 年发表的通过语言模型得到词表示的方法了，如图 3.12 所示。

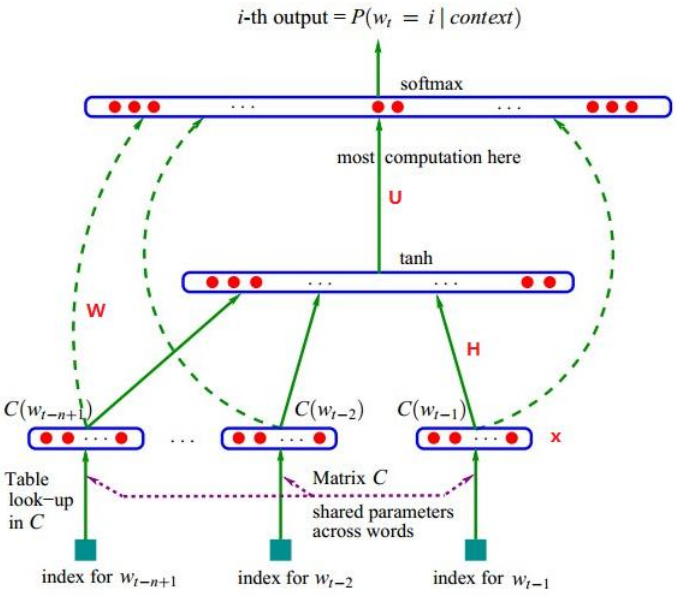


图 3.12

这是一个三层语言模型（有的理解为四层，即把输入层分成了输入层和投影层），它根据已知的前 n-1 个词（$w_{t-n+1},..,w_{t-1}$）来预测下一个词w_t，C(w)表示词 w 对应的词向量（C 就是所有词的词向量，是|V|×m 的矩阵），|V|表示词的总个数，m 表示词向量的维数，h 表示隐藏层个数，U（|V|×h 的矩阵）是隐藏层到输出层的参数，W（|V|×(n-1)m 的矩阵）是输入层到输出层的参数。

整个网络的输出计算公式为：

$$y = b + Wx + U\tanh(d + Hx)$$

解释下，网络的输入层就是根据前 n-1 词（$w_{t-n+1},..,w_{t-1}$），找到它们在 C 中对应的向量（$C(w_{t-n+1}),..,C(w_{t-1})$），然后连成一个(n-1)m 维的一维向量 x，即 $x=(C(w_{t-n+1}),..\ C(,w_{t-1}))$；网络的隐藏层和普通神经网络一样做一个线性变换，然后取 tanh；网络的输出层总共有|V|个节点，每个节点表示的就是根据已知的前 n-1 个词，下一个词是该词的概率，这个概率最后使用 softmax 进行归一化，即有（b + Wx表示从输入层到输出层直接有个线性变换）：

$$P(w_t|w_{t-n+1},..,w_{t-1}) = \frac{e^{y_{w_t}}}{\sum_i e^{y_i}}$$

那么，这个模型的参数为$\theta = (b, d, W, U, H, C)$，求解模型就可以使用梯度下降法求解了，即：

$$\theta = \theta + \varepsilon \frac{\partial \log P(w_t|w_{t-n+1},..,w_{t-1})}{\partial \theta}$$

可以看出，这个模型的计算复杂度很高，尤其是隐藏层到输出层的那个大矩阵相乘，所以，为了降低计算复杂度，提出了一些改进的模型，比如下面的 LBL（log-bilinear language model），框架图如图 3.13 所示。

它的网络的计算公式为：

$$h_{w_n} = \sum_{i=1..n-1} H_i \times C(w_i)$$

$$y = C(w_i)^T h_{w_n}$$

$$P(w_n|w_{n-1},..,w_1) = \frac{e^{y_{w_n}}}{\sum_i e^{y_i}}$$

其中$C(w)$就是词 w 对应的词向量。

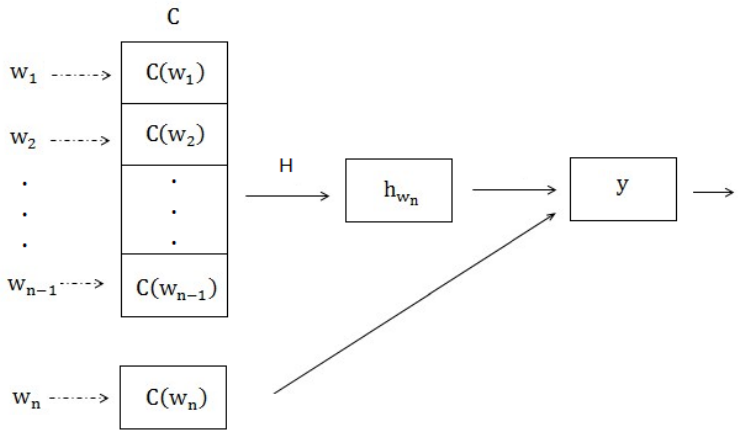


图 3.13

这个模型其实最好理解，h 隐藏层就是前 n-1 个词经过 H 变换之后得来的，也就是前 n-1 个词经过 H 这个变换之后就可以用来预测第 n

个词，然后输出层是什么呢？就是隐藏层和第 n 个词本身做内积，内积是可以表示相似度的，也就是说输出层是隐藏层所预测的第 n 个词和真实的第 n 个词的相似度，最后用 softmax 把概率归一一下。

RNNLM（Recurrent Neural Net Language Model）和上面的方法原理一样，但是思路有些许不同，框架图如图 3.14 所示。

它的网络的计算公式为：

$$s(t) = \text{sigmoid}\big(Uw(t) + Ws(t-1)\big)$$

$$P(w_{t+1}|w_t, s(t-1)) = y(t) = \text{softmax}(Vs(t))$$

其中，w(t)表示当前词w_t的向量，它这个向量的大小就是所有词汇的个数，所以 w(t)是个很长的且只有一个元素为 1 的向量，$s(t-1)$是上次的隐藏层，y(t)是输出层，它和 w(t)具有相同的维数，代表根据当前词w_t和隐藏层对词w_{t+1}的预测概率，由于 w(t)只有一个元素为 1，所以$Uw(t)$就是该词对应的词向量。整个过程就是来一个词，就和上一个隐藏层联合计算下一个隐藏层，然后反复进行这个操作，所以它对上下文信息利用的非常好。

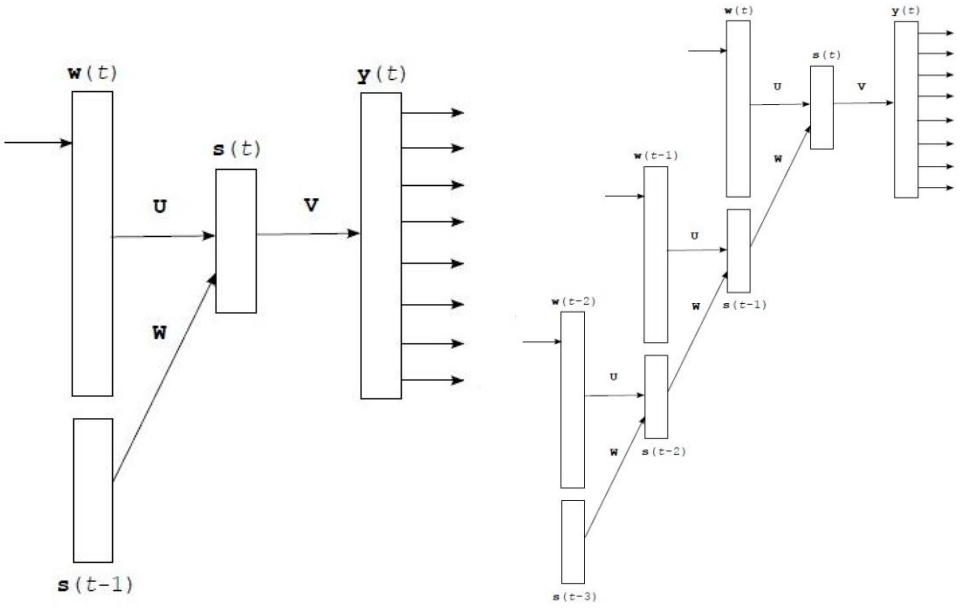


图 3.14

那么怎么评价词向量到底靠不靠谱呢？主要还是看它应用到具体任务的指标。既然将词表示成了向量，那么向量之间就可以计算内积、加减等运算，如果两个词有语义关系，那么它们就可以体现在向量上表示出来，就是计算内积（cos），我们试下找一些词，然后找出和它 cos 值最相似的 top 词看看结果，如下图 3.15 所示（在 60G 文本语料中训练得到），可以看出效果还可以，既然单个词可以计算相似度，那么很自然的想法就是对向量加减运算后也是向量，也可以计算相似性：C(中国)-C(美国)+C(华盛顿)的词向量应该和 C(北京)最相似。但是只能说热门词效果会比较好，对于不太用的词效果也并不是很理想，这个主要和语料有很大关系，语料越大，那么词向量就越好。很显然，

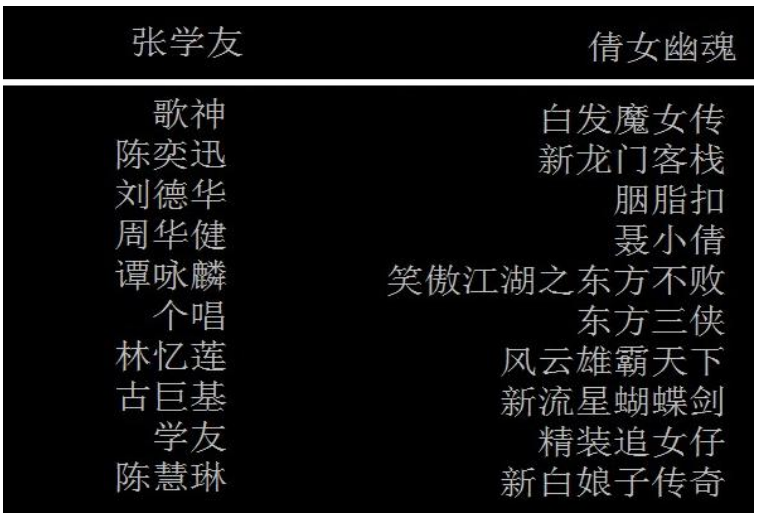


图 3.15

数据越多（大数据），对统计方法来说越有利，而且如果在同一领域（topic）内的话，数据就相对不会太稀疏，效果又会更好，所以一般我们处理任务的方法就是，先在通用的领域上完成某个任务，要想达到更好的效果，就要细分领域来处理（所以分类问题是任何领域都会涉及到的问题）。

词表示（Word Embedding）只可以给一个词训练出一个向量，然而在实际应用中，一些词会有多个意思（比如：苹果，小米等），那么一个向量就表示不了多个语义，所以我们设计了一个很简单的模型，我们称之为 Word Multi-Embedding，可以训练出一个词的多个语义向量。那么怎么将词的丰富含义融合到词表示模型中呢？有这么几种方法：

（1）使用 Topic model 可以得到每个词在每个类别的概率，然后可以融合到词表示模型中，但是该方法对长尾词效果并不好。

（2）使用词聚类可以将词类别聚出来，而且可以得到聚类中心点，然后对每一个词都可以根据上下文归入某一类，然后融合到词表示模型中，该方法在训练非常慢，在我们的实验中，500M 语料都得单机多线程训练 3 天，而且该方法同（1）都有一个缺点就是长尾类别没法聚出来。

（3）如果能事先挖掘出各个词属于某些类别的先验概率，那么也可以融合到词表示中，我们使用这种方法。

我们的模型流程如下：

Word Multi-Embedding 算法训练流程：

Input: 数据集(D 个句子)，句子窗口 n，上下文窗口 d；多义项词典
Output：词向量 $w_i, i = 1..N$

for t = 1..D do
K = word number of t's sentence
w *= lookup the K words*
for i = 1..K do
context(w_i*) =* $C^{w_i} = \{w_{i-d}, .. w_{i-1}, w_{i+1}, .. w_{i+d} | w_i\}$
class(w_i, z*) =* $Z^{w_i,z} = \{w_j, j = 1..m\}$
$z = argmax_z \sum_{a=1..2d} \sum_{b=1..m} S(C_a^{w_i}, Z_b^{w_i,z})$ *//use cosine*
$\theta(b, d, W^{i,z}, U, H, C) = \theta(b, d, W^{i,z}, U, H, C) + \varepsilon \frac{\partial log P(w_{i,z} | w_{i-n+1}, .., w_{i-1})}{\partial \theta}$

以下是一些根据向量计算相似度的实验效果：

```
Enter word or sentence (EXIT to break): 自重
```

自重_构件		自重_言行	
起重量	0.886117	自警	0.832679
载重量_物理	0.873391	光明磊落	0.828974
承重量	0.871370	自省	0.825810
轴重	0.863795	自爱	0.823834
荷载	0.839435	胸怀坦荡	0.816198
初速	0.823503	洁身自好	0.815774
重量	0.820806	严以律己	0.807913
载重	0.820695	自励	0.804034
吨位	0.815461	宽以待人	0.802081
220mm	0.814216	一身正气	0.793094
700mm	0.814177	不徇私情	0.785230
桨叶	0.809769	明哲保身	0.785225
功率	0.808533	出以公心	0.784128
容积	0.808082	择善固执	0.783217
耗油率	0.807036	正派	0.783193

```
Enter word or sentence (EXIT to break): 制服
```

制服_式样		制服_制伏	
便装	0.930641	制伏	0.938556
警服	0.914627	劫匪	0.910118
便服	0.912806	歹徒	0.909464
身穿	0.901339	破门而入	0.893938
警装	0.893544	逃走	0.892542
工作服	0.883687	逃跑	0.886329
迷彩服	0.878245	束手就擒	0.885968
身着	0.873902	行凶	0.883689
厨师服	0.872912	抢匪	0.881547
穿着	0.871052	匪徒	0.877356
着装	0.862232	擒获	0.876738
皮夹克	0.850473	持刀者	0.871989
鸭舌帽	0.844225	凶器	0.867511
夹克	0.840502	乘其不备	0.866660
军绿色	0.840028	拒捕	0.865053

```
Enter word or sentence (EXIT to break): 苹果
```

苹果_水果		苹果_公司	
土豆	0.896936	谷歌	0.897802
香蕉	0.893269	戴尔	0.881227
花生	0.891218	微软	0.878704
圣女果	0.883681	惠普	0.869296
西瓜	0.882138	rim	0.864034
青枣	0.882046	zune	0.862256
芒果	0.881503	realnetworks	0.858963
香瓜	0.877793	pc	0.854004
草莓	0.876297	alienware	0.853922
桔子	0.873596	索尼	0.850880
橘子	0.871722	appstore	0.849236
哈密瓜	0.871591	palm	0.847549
西红柿	0.869526	google	0.843413
菠萝	0.867869	摩托罗拉	0.839573
橙子	0.863555	dell	0.839144

图 3.16

目前词向量一般也有两种用法：（1）把词向量当做一个特征，加入到现有的 NLP 任务中。我们曾经尝试将词向量作为命名实体识别的一个额外特征，效果虽然有提升，但是并不像在图像和语音应用中那么明显；（2）直接把词向量作为神经网络的输入完成一些 NLP 任务，例如，《Natural Language Processing (Almost) from Scratch》这篇文章的工作，我以 POS 任务简单介绍下这种方法的思路，如下图 3.17 所示（比作者论文中的那个图要好理解），首先来了一句话之后要进行 lookup 操作，即寻找词的特征，词向量特征的大小是 50 维，紧接着有一个 5 维 caps 特征（是否大写，是否数字等），然后是一个 5 维后缀特征（最多 455 个），这就组成了每个词的输入特征（共 60 维）；然后经过 linear 操作，映射到隐藏层，它有 300 维大小；做一个 Handtanh 操作之后，还是使用 linear 操作映射到输出层，输出层中每个词的维数是 45（pos 有 45 个标记），最后经过 Viterbi 算法找到每个词的最佳

POS_TAG，这就是作者论文使用深度学习来处理 NLP 任务的大致思路，其他任务也类似，大家可以详读一下论文。

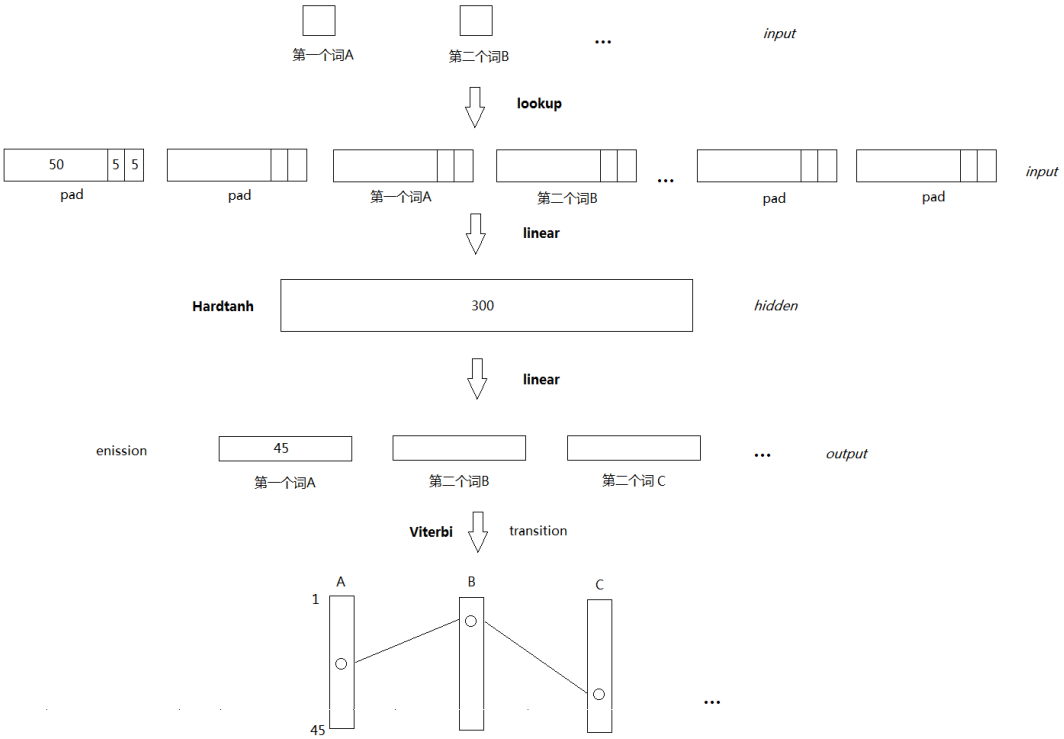


图 3.17

目前来说，深度学习在 NLP 的应用还不能用惊艳来形容，还有待大家共同努力尝试。

3.6 其他：kNN，k-means，DT/Boosting/GBDT，SVM

3.6.1 kNN

kNN（k 最近邻）算法的思想比较简单：如果一个样本在特征空间中的 k 个最相似（即特征空间中“距离”最近）的样本中的大多数属于某一个类别，则该样本也属于这个类别。它是一种监督学习的分类算法。

3.6.2 k-means

k-means（k 均值）聚类算法是假定在欧式空间下，并且 k 是事先确定的。首先随机选取 k 个质心（一般会选择尽可能相互远的点），然后将各个数据项分配给“距离”最近的质心点，分配后，该类下的质心就会要更新（比如：变成该类下所有节点的平均值），该分配过程一直下去，直到聚类结果不再变化。

3.6.3 DT/Boosting/GBDT

DT(Decision Tree，决策树)

决策树是一个树结构。其每个非叶子节点表示一个特征属性上的判断，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，判断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。其实就是从跟节点开始，进行 if-else 判断一直到叶子节点，就得到了分类结果。

决策树的构造，也就是如何根据特征属性确定每个分支，指导思想就是这个分支一定要能最大化的区分不同类。为了确定哪个属性最适合用来拆分，算法会计算相应的信息增益，所谓某个特征的信息增益，就是指该特征对数据样本的分类的不确定性的减少的程度，也就是说信息增益越大的特征具有更强的区分能力，是指当前熵与两个新群组经加权平均后的熵之间的差值。算法会针对每个特征属性计算相应的信息增益，然后从中选出信息增益最大的特征属性。具体算法有 ID3 算法和 C4.5 算法。

Boosting

Boosting 算法的思想非常简单，它就是将若干个弱分类器（或者叫弱模型）组合起来，形成一个强大的强分类器（或者叫强模型），其中最流行的一种就是 AdaBoost 算法，在此算法中，每个样本都被赋予一个权重（初始时权重都相同），代表该样本被某个弱分类器选入训练集的概率，如果某个样本点已经被准确地分类，那么在构造下一个弱分类器时，它被选中的概率就被降低；相反，如果某个样本没有被准确地分类，那么它的权重就会提高，意味着该样本将更大机会的进入下一个弱分类器的训练集，就这样经过 T 次循环，就得到了 T 个弱分类器，把这 T 个弱分类器按一定的权重叠起来就得到了最终的强分类器模型。

GBDT(Gradient Boosting Decision Tree)

GBDT 是 Friedman 于 1999 年在《Greedy Function Approximation: A Gradient Boosting Machine》中提出来的。要想理解 GBDT 算法，就要知道两个概念 Gradient Boosting 和 Boosting Decision Tree，Gradient Boosting 与传统 Boosting 的区别是，每一次的计算是为了减少上一次弱模型的残差，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弱模型。也就是说，Gradient Boosting 中，每个新的弱模型的建立是为了使得之前弱模型的残差往梯度方向减少，与传统 Boosting 对样本直接进行加权有着很大的区别；Boosting Decision Tree 和传统 Boosting 的最大区别是其中的弱分类器（或者叫弱模型）是一个决策树，在 GBDT 中这个决策树是回归树，而不是分类树，下面来大致解释下这个模型：

1、初始化：$\mathrm{f}_0(\mathrm{x}) = \mathrm{argmin}_\gamma \sum_{i=1}^{N} L(y_i, \gamma)$

2、for m = 1 to M：

(a) for i = 1,2,…,N, 计算（负梯度）：

$$\mathrm{r}_{im} = -[\frac{\partial L(y_i, f(x_i))}{\partial f(x_i)}]_{f=f_{m-1}}$$

(b)用$\{(x_i, \mathrm{r}_{im})\}_{i=1}^{N}$拟合一个回归树，得到第 m 棵树的叶子节点区域$\mathrm{R}_{jm}, j = 1,2,\dots,J_m$

(c) for $j = 1,2,\dots,J_m$，计算（最优下降步长）：

$$\gamma_{jm} = argmin_\gamma \sum_{x_i \in R_{jm}} L(y_i, f_{m-1}(x_i) + \gamma)$$

(d) 更新$\mathrm{f}_m(\mathrm{x}) = \mathrm{f}_{m-1}(\mathrm{x}) + \sum_{j=1}^{J_m} \gamma_{jm} I(x \in R_{jm})$

3、输出模型$\hat{\mathrm{f}}(\mathrm{x}) = \mathrm{f}_M(\mathrm{x}) = \sum_{i=0}^{M} f_i(x)$

其中$\mathrm{h}_m(\mathrm{x}) = \sum_{j=1}^{J_m} \gamma_{jm} I(x \in R_{jm})$就是一个弱模型（回归树），$L$是损失函数（还记得前面讲的损失函数吗），$\mathrm{r}_{im}$就是梯度，看到了吧，中间就是用梯度（严格说是负梯度）来建立回归树，也就是说 GBDT 其实就是在前向分布算法（Forward stagewise additive modeling）上用了负梯度来建立回归树。

3.6.4 SVM

SVM（Support Vector Machine，支持向量机）自从 Cortes 和 Vapnik 于 1995 年提出来以后，由于它完备的理论背景，可以从线性可分扩展到线性不可分的情况，效果也很不错，逐渐风光起来了，它是一个很经典的有监督分类模型。

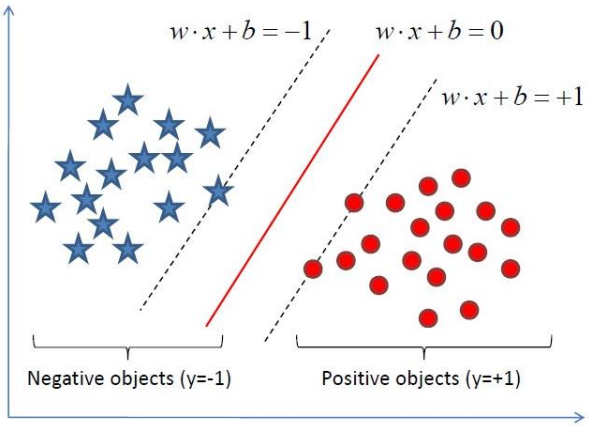


图 3.18

SVM 既然是一个机器学习模型，那么它就会有目标函数和约束条件，如图 3.18 所示，蓝色的是负样本点，都满足$w \cdot x + b \leq -1$；红色的是正样本点，都满足$w \cdot x + b \geq 1$，中间的红线是分类面$w \cdot x + b = 0$，那么要想使得红色和蓝色分的越好，就要使正负样本**线**上的点（正负样本线上的点都分开了，那么它们两侧的点自然也分开了）离分类面越远越好，即（点到面的距离公式）：$2 \times |w \cdot x + b|/||w||$，又因为$|w \cdot x + b| = 1$（因为是在正负样本线$w \cdot x + b = \pm 1$上），所以它的优化目标函数就是最大化几何间隔$2/||w||$（相当于最小化$||w||^2/2$），它的约束条件就是样本点必须在负样本线（$w \cdot x + b = -1$）和正样本线$w \cdot x + b = +1$的两侧，不能落入两者中间，即：

$$\min_{w,b} ||w||^2/2$$

$$s.t.\ \ y_i(w \cdot x_i + b) \geq 1 \quad (i = 1..N)$$

但是样本集免不了会有噪声，也就是会有样本不满足约束条件，落入了正负样本线之间，因此就要引入一个松弛变量，允许一些样本不满足约束条件，但是要惩罚，这样整个最优化函数就又变了，即：

$$\min_{w,b,\varepsilon} ||w||^2/2 + C \sum_{i=1..N} \varepsilon_i$$

$$s.t.\ \ y_i(w \cdot x_i + b) \geq 1 - \varepsilon_i \quad (i = 1..N)$$

$$\varepsilon_i \geq 0 \ \ (\mathrm{i} = 1..\mathrm{N})$$

现在 SVM 求解线性分类没什么问题了，那么如果线性方法无法区分呢？如果将线性无法区分的问题映射到高维空间（x->ϕ(x)），然后在高维空间可以线性区分的话，那就可以同样使用之前的线性方法了（如图 3.19），即：

$$\min_{\mathrm{w,b}} ||\mathrm{w}||^2/2$$

$$\mathrm{s.t.} \ \ \mathrm{y_i}(\mathrm{w} \cdot \phi(\mathrm{x_i}) + \mathrm{b}) \geq 1 \ \ \ (\mathrm{i} = 1..\mathrm{N})$$

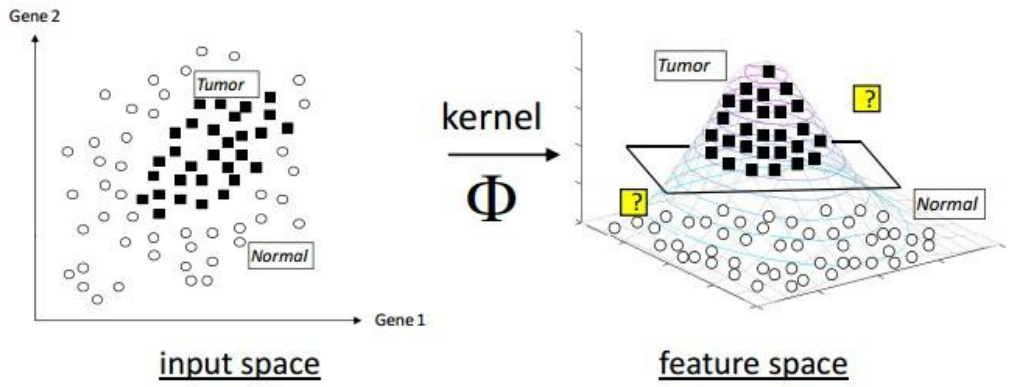


图 3.19

求解上面那个问题，可以使用对偶问题（还记得最大熵时讲的对偶问题吗？）来求解，最后就转化为下面的最优化问题了：

$$\min_{\alpha} \frac{1}{2} \sum_{\mathrm{i=1..N}} \sum_{\mathrm{i=1..N}} \alpha_\mathrm{i}\alpha_\mathrm{j}\mathrm{y_i}\mathrm{y_j}\left(\phi(\mathrm{x_i}) \cdot \phi(\mathrm{x_j})\right) - \sum_{\mathrm{i=1..N}} \alpha_\mathrm{i}$$

$$\mathrm{s.t.} \ \ \sum_{\mathrm{i=1..N}} \alpha_\mathrm{i}\mathrm{y_i} = 0$$

$$C \geq \alpha_\mathrm{i} \geq 0 \ \ \ (\mathrm{i} = 1..\mathrm{N})$$

问题来了，如何从低维空间映射到高维空间呢？也就是ϕ(x)如何选取？而且还要在高维空间计算$\phi(\mathrm{x_i}) \cdot \phi(\mathrm{x_j})$。这时就要引入核函数，它的作用就是把两个低维空间的向量映射到高维空间，而且它同时把它们在高维空间里的向量内积值都算好了，也就是核函数$K\left(\mathrm{x_i}, \mathrm{x_j}\right) =$

$\phi(x_i) \cdot \phi(x_j)$，一个函数干了两件事，先映射到高维空间然后计算好了内积，使得我们不用关心高维空间到底是什么了，因为我们直接通过核函数拿到了结果。这样，在核函数$K(x_i, x_j)$给定的条件下，可以利用求解线性分类问题的方法求解非线性分类问题。

那么有了最优化模型，而且是个凸二次规划问题，就可以使用 SMO 算法求解了，SMO 算法其实就是分而治之的思想。SVM 具体细节可以参考相关论文（很多），下面一篇是 SVM 开创原论文，一篇是讲的很好的 ppt：

《Support-Vector Networks》

《A Gentle Introduction to Support Vector Machines in Biomedicine》

应用篇

第四章、如何计算的更快

随着数据量的爆炸式增长，如何存储和计算海量数据就成了一个问题，所以解决这个问题的分布式系统逐渐成为目前必不可少的技术之一。而对于线上业务怎么更快更好的完成用户请求也是很重要的。

4.1 程序优化

在讲程序优化之前，必须先得知道程序是怎么运行的。比如，我们刚编写了一个 printf(“hello world\n”)程序，要执行它，从键盘敲入“./hello”的时候，shell 会逐一读取字符到寄存器，然后把它们放到存储

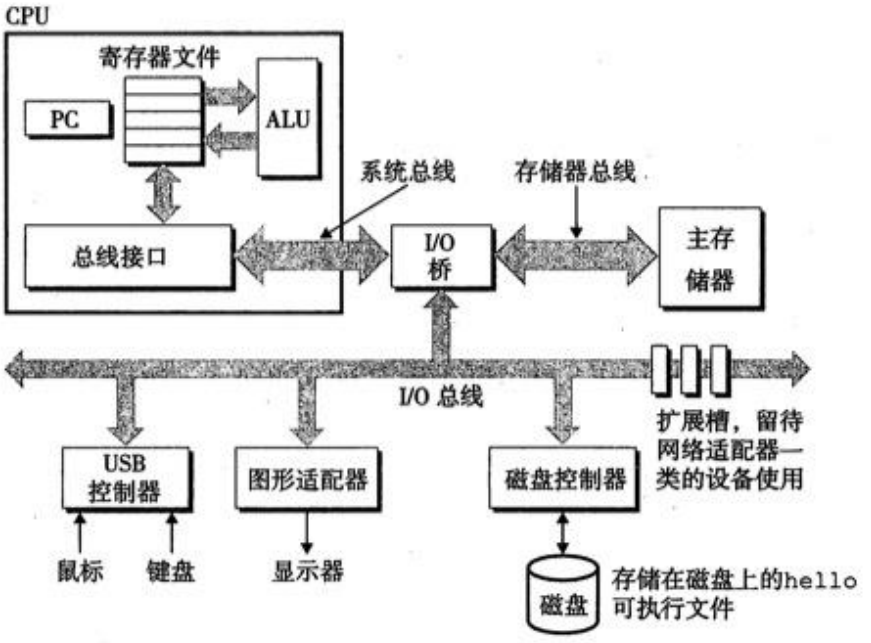


图 1-4 一个典型系统的硬件组成

CPU：中央处理单元；ALU：算术/逻辑单元；PC：程序计数器；USB：通用串行总线

图 4.1

器中，当按下回车的时候，shell 就会执行一系列命令，这些命令会将

hello 目标文件中的代码和数据从磁盘拷贝到主存（利用 DMA 可以不通过处理器而直接从磁盘到主存），一旦 hello 目标文件中的代码和数据被加载到了存储器中，处理器就开始执行 hello 程序，它将“hello world\n”串从存储器拷贝到寄存器堆，再从寄存器中拷贝到显示器上。这样整个过程就结束了。我们可以看到，整个过程系统花费了大量的时间把信息从一个地方拷贝到另一个地方，hello 程序的机器指令开始时是在磁盘上，程序加载时，它们被拷贝到主存，当处理器运行程序时，指令又从主存拷贝到处理器。这样就会有个问题需要系统设计者考虑，就是怎样设计存储器使这些拷贝操作尽可能的快。

存储器的设计如图 4.2 所示，可以将上一层次的存储器看作是下一层次存储器的高速缓存，越上层的速度越快但存储量越小，相反，越下层的速度越慢但存储量越大。这些内容来自《深入理解计算机系统》，这是一本至少要读两遍的书。

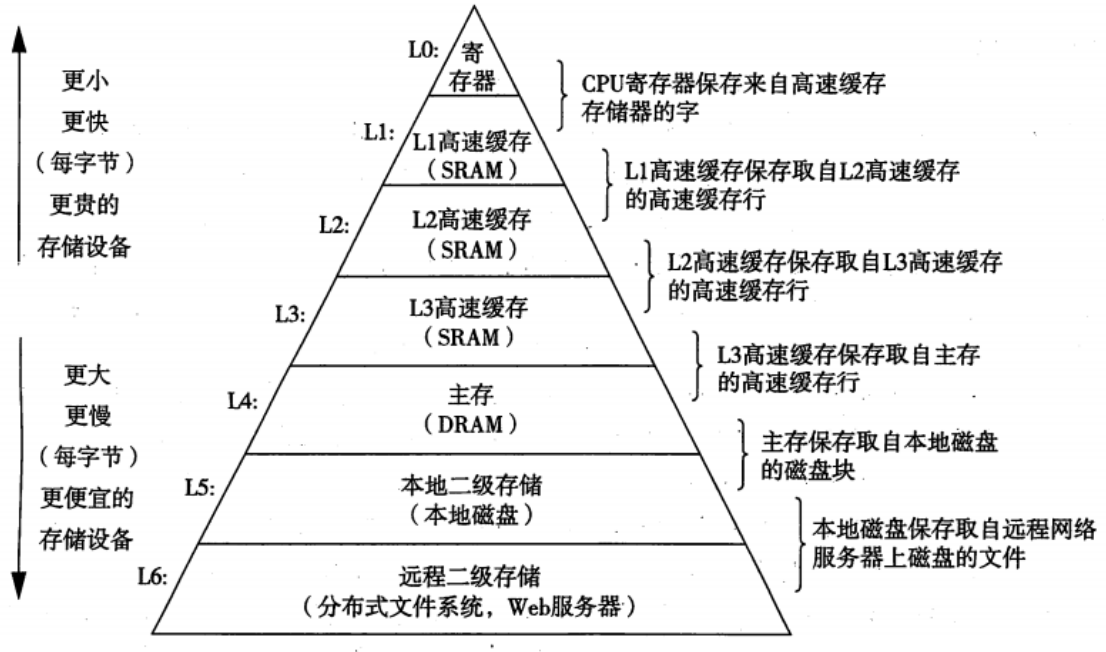


图 4.2

现在我们来说程序优化，程序优化从大的方面来讲可以分为三个

级别：1）系统级；2）算法/数据结构级；3）代码级。系统级对性能的影响最大，其次是算法/数据结构级，再次是代码级。系统级注重系统的整个流程，负载均衡等等，后面要专门讲的分布式也算是系统级的优化，下面只讲下算法/数据结构级和代码级。

毋庸置疑，一个好的算法或数据结构对系统乃至整个产业都会有很大影响，快速傅里叶变换之所以在推动数字信号处理的快速发展起了很大作用，就是它将傅里叶变换从 O(N^2)变成了 O(NlogN)，可以算是一个质的改进。所以说，我们在写程序的时候选用什么样的算法，什么样的数据结构都很重要，前提是你要知道目前有哪些算法或者数据结构可以解决你的问题，比如：优先队列，Trie 树，跳表，Bloom Filter 等这些大学没学到的算法你知道它们有什么作用吗？比如，在我们的一个模板系统中，我们将正则匹配改成了 Trie 树+优先队列+各种剪枝，性能就有质的提升。然而有些实际问题并不能使用现有的算法搞定，必须根据自己的业务需求，自己设计算法和数据结构，这完全就看个人能力了。除了算法本身的特点外，还有一个很重要的思想就是：空间换时间。比如：在我们的一个排序中要将一个 0-1 亿的数当做一个特征使用起来，我们设计了一个双 sigmoid 函数将这个值映射到了 0-1 的浮点数，用到了指数函数，为了能加速，我们事先将 0-1 亿个值的双 sigmoid 值计算好存储起来，当使用的时候就直接读取就行了，对系统性能也是有提升的；再比如，假如下面这段程序会被很多次的调用，那么可以事先将 a 数组按照 TYPE 分到各个 a_TYPE 数组中，再做相应的处理，这样做的好处是：1）降低了循环次数；2）消除了 if 判断。相信我，这些小的细节优化一定会有提升的，只要你愿意去发现去优

化。

```
for (int i = 0; i < A_NUM; ++i)
{
    if (TYPE == a[i])
        ; //do something
}
```

代码级优化也很重要（好多人其实不愿意做这个事），从上面图 4.2 可以看出，越上层存储器越快，上层可以看成是下层的缓存，其实计算机的确会缓存，比如读磁盘的时候并不是一个字节一个字节读，而是一次读一个块到内存（尽管用户程序就是读一个字节），这样读下一个字节的时候就不用再和磁盘交互了，寄存器缓存也是会做一样的事情。这就衍生出两个很重要的指导思想：1）尽可能使用上层存储器，能使用寄存器的时候就尽量不要使用内存，能使用内存的时候就尽量不要使用磁盘。2）计算机具有局部性。例如：尽可能多用局部变量，因为局部变量大多会缓存在寄存器中，访问速度快；尽可能少调用函数，函数参数尽可能少，参数尽可能是指针或者引用，因为可以减少拷贝；处理的数据尽可能紧凑且少，因为可以大概率的缓存到上层存储器中，这也是数据压缩的目的之一；尽可能顺序读写而不要随机读写，且尽量多使用刚读取的数据，因为程序局部性原理，最近使用的数据附近的数据会缓存起来；尽可能使用内存，而不是磁盘，在这儿简单说下磁盘读取的原理，如图 4.3，假设要读取③位置的数据，磁头首先需要从它现在的位置④处移动到③的位置（这个叫寻道时间），读取数据时，像⑤那样盘片转动之后，盘片③上的数据就超过了磁头的位置而无法读取到，盘片就必须再转一圈（这个叫做旋转时间），读取到数据后，就需要通过总线⑨传输到 CPU 中（这个叫传送时间）。当然

操作系统为了提高性能，减少磁盘旋转，每次会读一个块（4Kb 左右）的数据，而不是一个字节。看到了为什么内存比磁盘读取速度快的原因了吧，磁盘读取数据耗时是寻道时间+旋转时间+传送时间，而内存几乎就是传送时间，而且它两的传送时间也不在一个量级上。SSD（固态硬盘）虽然不需要物理移动即可高效搜索到数据，但是由于总线速度的瓶颈以及其他结构的影响速度也是无法和内存相比的。

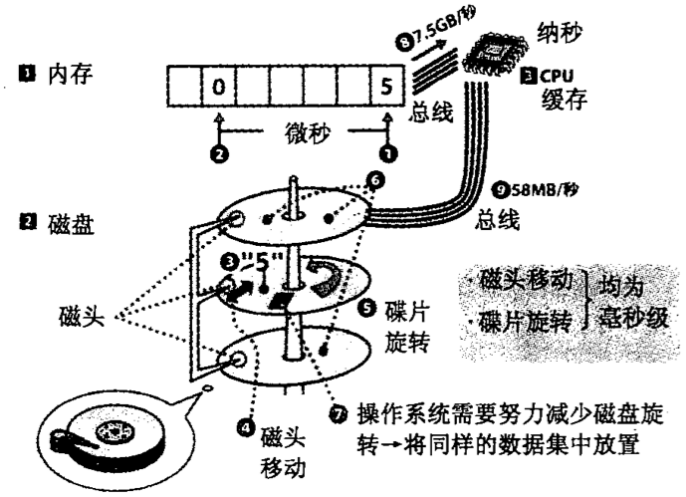


图 4.3

总之，要想写出高效的程序，除了学习算法和数据结构外，计算机结构必须要了如指掌，这是根本，所谓万变不离其宗，然后就是多写程序，多看书，多思考，最重要的一点就是态度，要想优化一定会有很多点可以优化，就看你愿不愿意做了。

4.2 为什么要分布式系统

一天，小明的老板给了他一个任务：让他去计算一对浮点数相乘。对小明这种码农来说自然容易得很，一行代码就搞定了；过了一会儿，小明老板又来找他，说他手上有 1 万对浮点数需要相乘，对小明来说

也很容易，循环一下就搞定了；过了两天，老板又找小明来了，说他手上现在有 1 百亿对浮点数，让小明尽快去把相乘结果给他。小明想，容易啊，把前面那个循环程序拿来一跑就行了啊，于是，小明开始在它的 PC 上跑程序了，但是令小明郁闷的是，时间一分一秒的过去，就是跑不出结果啊（读者不妨试试，1 百亿个浮点数相乘，在你的 PC 上要跑多久）…

随着互联网的发展，数据是爆炸式增长的，那么如何存储和计算大规模的数据就是一个非常棘手的问题了。小明连 1 百亿个浮点数都计算不出来，然而浮点相乘算是最简单的算法了，稍微复杂的算法，那更跑不出来了。

既然遇到了问题，那就应该想办法解决。怎么解决呢？一种方法：小明想，如果老板能给他配一台牛逼点的机器，比如中科院的超级计算机，那不是很快就计算出来了吗？但是这可能吗？为了计算 1 百亿个浮点数相乘，买一台超级计算机，这显然成本太高了啊，这个方法显然行不通。那就需要第二种方法：用术语说叫分治思想，用俗语说就是人海战术。既然 1 百亿个数没法计算，那么就多找几台机器，机器差点都无所谓，每台机器计算一部分，然后最后汇总起来，不就快了。没错，这其实就是分布式计算的思想。

分布式系统主要包括两部分：分布式存储和分布式计算。要了解这些东西，必不可少要看的两篇 google 的论文：《The Google File System》和《MapReduce: Simplified Data Processing on Large Clusters》。目前，分布式存储根据不同的业务产生了不同的数据库，尤其随着数据量的增大，就诞生了另一类数据库：NoSQL 数据库，它主要包括：（1）key-value

数据库，代表有：Redis，LevelDB，Dynamo 等；（2）列式数据库，代表有：BigTable，Hypertable，Cassandra 等；（3）文档数据库，代表有：CouchDB，MongoDB 等；（4）图形数据库，代表有：OrientDB，GraphDB 等。分布式计算模型大概有这几种：（1）多线程，最基本的方法；（2）Graphics Processing Units，利用图形处理器的高度并行结构来提高速度；（3）Message Passing Interface，一种消息传递编程模型；（4）MapReduce。

要想设计好分布式系统，其实不是一件简单的事情，需要考虑很多事情：集群负载均衡，数据的正确性和完整性，服务器的错误处理等等。本人不是专业搞分布式的，也只是看了些论文和书籍，也就不详细说了，读者感兴趣可以参考相应的文献，下面只是从应用的角度来看看分布式系统的用处。

4.3 Hadoop

Hadoop 是一个软件平台，是 Apache 开源组织的一个分布式计算开源框架，可以让你很容易地开发和运行处理海量数据的应用。Hadoop 框架中最核心的设计就是：MapReduce 和 HDFS，也可以说，Hadoop 是基于分布式文件系统（HDFS）的 MapReduce 的实现，也可以说，它是 google 的那两篇经典文章的开源实现。现在，几乎每家大公司都会部署 hadoop 集群来进行大数据运算，开源真是好东西啊！

分布式文件系统（HDFS）

HDFS 采用 master/slave 架构。一个 HDFS 集群是由一个 Namenode 和一定数目的 Datanodes 组成。Namenode 是一个中心服务器，负责管

理文件系统的名字空间(namespace)以及客户端对文件的访问。集群中的 Datanode 一般是一个节点一个，负责管理它所在节点上的存储。HDFS 暴露了文件系统的名字空间，用户能够以文件的形式在上面存储数据。从内部看，一个文件其实被分成一个或多个数据块，这些块存储在一组 Datanode 上。Namenode 执行文件系统的名字空间操作，比如打开、关闭、重命名文件或目录。它也负责确定数据块到具体 Datanode 节点的映射。Datanode 负责处理文件系统客户端的读写请求。在 Namenode 的统一调度下进行数据块的创建、删除和复制。

MapReduce

MapReduce 任务是用来处理键/值对的。该框架将转换每个输入的记录成一个键/值对，每对数据会被输入给 Map 作业。Map 任务的输出是一套键/值对，原则上，输入是一个键/值对，但是，输出可以是多个键/值对。之后，它对 Map 输出键/值对分组和排序。然后，对排序的每个键值对调用一次 Reduce 方法，它的输出是一个键值和一套关联的数据值。Reduce 方法可以输出任意数量的键/值对，这将被写入工作输出目录下的输出文件。这个过程如下图 4.4 所示。

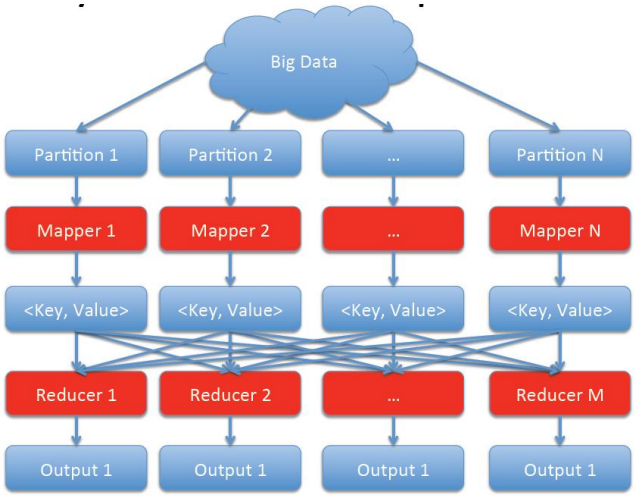


图 4.4

如果不好理解，那就以那个最经典的计算词频来说，假设有一堆分好词的文本，现在的任务是要统计每个词的频率（出现的次数），如图 4.5 所示。

首先把文本分成若干份；然后每一份数据，分配给一个 Mapper，这个 Mapper 的职责就是将每个词赋值为 1（即 key 为该词，value 为 1）；之后每个 Mapper 会把自己的数据按照某种规律（比如按字母序，这样同一个词才能分配到一个 Reducer）发给相应的 Reducer，Reducer 的职责就是把相同 key 的 value 累加起来；这样，就统计完词频了。

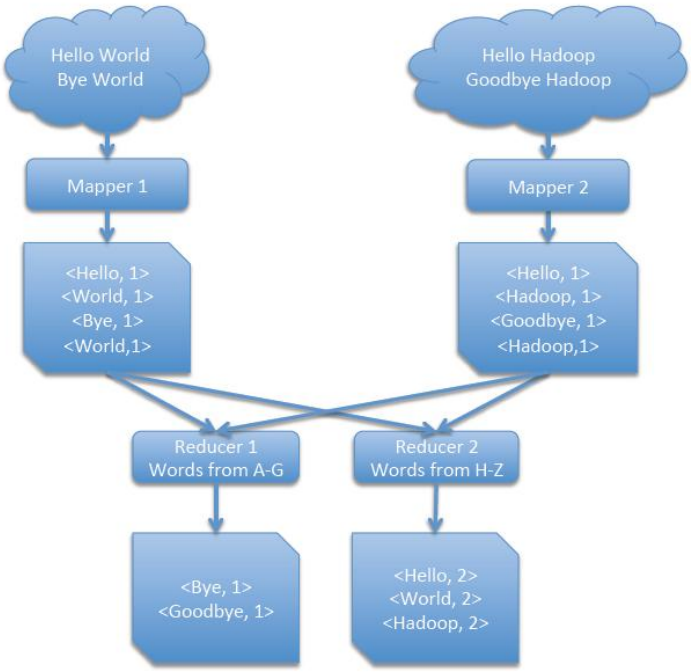


图 4.5

Hadoop 的编程方式有两种：Pipes 和 Streaming。直接上代码吧，下面的例子是笔者在 2011 年写的，现在版本的 hadoop 又增加了不少新特性了。

Pipes 方式：

首先，建立相应的目录：

```
> hadoop fs –mkdir name
> hadoop fs –mkdir name/input
>hadoop fs –put file1.txt file2.txt name/input
```

1、编写程序（wordcount.cpp）

```
#include<algorithm>
#include<limits>
#include<string>
#include"stdint.h"
#include"hadoop/Pipes.hh"
#include"hadoop/TemplateFactory.hh"
#include"hadoop/StringUtils.hh"
usingnamespace std;
class WordCountMapper:publicHadoopPipes::Mapper
{
public:
	WordCountMapper(HadoopPipes::TaskContext&context){}
	void map(HadoopPipes::MapContext& context)
	{
		string line =context.getInputValue();
		vector<string>word = HadoopUtils::splitString(line, " ");
		for (unsignedint i=0; i<word.size(); i++)
		{
			context.emit(word[i],HadoopUtils::toString(1));
		}
	}
};
class WordCountReducer:publicHadoopPipes::Reducer
{
public:
	WordCountReducer(HadoopPipes::TaskContext&context){}
	void reduce(HadoopPipes::ReduceContext& context)
	{
		int count = 0;
		while (context.nextValue())
		{
			count +=HadoopUtils::toInt(context.getInputValue());
		}
		context.emit(context.getInputKey(),HadoopUtils::toString(count));
	}
};
int main(int argc, char **argv)
{
	returnHadoopPipes::runTask(HadoopPipes::TemplateFactory<WordCountMapper,WordCountReducer>());
}
```

2、编写 makefile

```
CC = g++
HADOOP_INSTALL =../../data/users/hadoop/hadoop/
PLATFORM = Linux-amd64-64
CPPFLAGS = -m64-I$(HADOOP_INSTALL)/c++/$(PLATFORM)/include

wordcount:wordcount.cpp
	$(CC) $(CPPFLAGS) $< -Wall -L$(HADOOP_INSTALL)/c++/$(PLATFORM)/lib-lhadooppipes -lhadooputils
-lpthread -g -O2 -o $@
```

3、编译程序并且放入 hadoop 系统

```
> make wordcount
> hadoop fs –put wordcount name/worcount
```

4、编写配置文件（job_config.xml）

```
<?xml version="1.0"?>
<configuration>
<property>
	<name>mapred.job.name</name>
	<value>WordCount</value>
</property>

<property>
	<name>mapred.reduce.tasks</name>
<value>10</value>
</property>

<property>
<name>mapred.task.timeout</name>
```

```
<value>180000</value>
</property>

<property>
<name>hadoop.pipes.executable</name>
<value>/user/hadoop/name/wordcount</value>
<description> Executable path is given as"path#executable-name"
                sothat the executable will havea symlink in working directory.
                This can be used for gdbdebugging etc.
</description>
</property>

<property>
<name>mapred.create.symlink</name>
<value>yes</value>
</property>

<property>
<name>hadoop.pipes.java.recordreader</name>
<value>true</value>
</property>

<property>
<name>hadoop.pipes.java.recordwriter</name>
<value>true</value>
</property>
</configuration>

<property>
<name>mapred.child.env</name>
<value>LD_LIBRARY_PATH=/data/lib</value><!--如果用到动态库：lib 库的路径，要保证每台机器上都有，现在的版本直接用命令打个包就行了-->
<description>User added environment variables for the task tracker child
        processes. Example :
        1) A=foo   This will set the env variable A to foo
        2) B=$B:c This is inherit tasktracker's B env variable.
</description>
</property>
<property>
<name>mapred.cache.files</name>
<value>/user/hadoop/name/data#data</value><!--如果用到外部文件：hadoop 上的 data 路径，程序中fopen("data/file.txt", "r")，现在的版本直接用命令打个包就行了-->
</property>
```

5、运行程序

> hadoop pipes -conf ./job_config.xml -input/user/hadoop/name/input/* -output /user/hadoop/name/output -program/user/hadoop/name/wordcount

(注：output 文件夹在运行前不能建立，系统会自己建立)

这个例子很简单，只是统计词频，但是，实际的数据挖掘比较复杂，尤其涉及到中文，很多情况下要进行分词，那就要初始化一些分词句柄及空间，然后分词处理，其实可以将 MapReduce 程序看成普通的 C++程序，要初始化东西，放到构造函数，具体处理放到 Map 和 Reduce 里。

Streaming 方式：

1、编写 map 程序（map.cpp）

```
#include<string>
#include<iostream>
usingnamespace std;

int main()
{
    string line;
    while(cin>>line)//如果是中文的话，用fgets(char*, int n, stdin)读进来，再分词处理
    {
        cout<<line<<"\t"<<1<<endl;
    }
    return 0;
}
```

>>g++ -o map map.cpp

2、编写 reduce 程序（reduce.cpp）

```
#include<map>
#include<string>
#include<iostream>
usingnamespace std;

int main()
{
	string key;
	string value;
	map<string,int> word_count;
	map<string,int> :: iterator it;
	while(cin>>key)
	{
		cin>>value;
		it= word_count.find(key);
		if(it!= word_count.end())
		{
			++(it->second);
		}
		else
		{
			word_count.insert(make_pair(key,1));
		}
	}

	for(it= word_count.begin(); it != word_count.end(); ++it)
		cout<<it->first<<"\t"<<it->second<<endl;

return 0;
}
```

>>g++ -o reduce reduce.cpp

3、需要统计的文件，并提交至 hadoop 中

File1.txt：hello hadoop helloworld

File2.txt：this is a firsthadoop

>>hadoop fs –put File1.txt File2.txt ans

4、运行程序

>>hadoop jar /data/users/hadoop/hadoop/contrib/streaming/hadoop-streaming-0.20.9.jar -file map -file reduce -input ans/* -output output1 -mapper /data/name/hadoop_streaming/map -reducer /data/name/hadoop_streaming/reduce

第五章、你要知道的一些术语

要想了解以后的章节，尤其是搜索引擎，那么一些基本的术语就必须要知道。

5.1 tf/df/idf

首先需要声明的是我们一般说 tf，df，idf 都是指某个词的 tf，df，idf，也可以说这三个术语是词的属性。

tf 就是词频，它的全称是 term frequency。这个概念最容易理解，就是某个词出现的次数，出现几次，该词的 tf 就是几。它一般表示的是一个词的局部信息。

df 就是文档频率，它的全称就是 document frequency。它就是指某个词的文档频率，这个词在多少个文档中出现，那么，该词的 df 就是几。

idf 是逆文档频率，它的全称就是 inverse document frequency。它是词重要性的一个很好地衡量。试想，如果某个词在非常多的文档中都出现过，比如：“的”、“了”这些词，那么它是不就不太重要；相反，如果某个词在很少的文档中出现过，那么它是不就相对来说比较重要。简单来说就是物以稀为贵。那么，idf 怎么计算呢？公式很简单（有些会稍有不同）：

$$\mathrm{idf_i} = \log \frac{|\mathrm{D}|}{\mathrm{df_i}}$$

也就是用|D|（语料库中的总文档数）除以该词的 df，然后取 log 就可

以了。idf 因为是大量语料库中统计的，所以它一般表示一个词的全局信息。

5.2 PageRank

作为一个搞文本的人士，尤其是与搜索引擎相关的，如果连 PageRank 都没听说过，那真是太土鳖了，就算是不知道它的具体原理，至少听说过这个名字吧，这可是 Google 的最经典的算法之一。没关系，看完本节，你也就知道 PageRank 到底是什么玩意了。

PageRank 是用来衡量网页重要性的一个指标。计算网页重要性有很多方法，比如通过计算网页本身：内容好不好？网页规不规范？排版好不好？等等因素也可以计算出来一个得分，这种方法只是利用了网页本身，而 PageRank 则不同，它利用了整个互联网络。

PageRank 的核心思想就是投票原则。互联网中的网页之间是有链接关系的，也就是说任何网页有可能和其他网页有链接关系（要么 A 指向 B，要么 A 被 C 指向），这就构成了一个非常大的矩阵，如果指向某个网页的链接非常多且质量越高，那么该网页质量也就很高，A 网页链接到了 B 网页，说明 A 网页给 B 网页投了一票。这个思想很容易理解，就像投票一样，获得票数越多的人，也就是说明他越重要。那么 PageRank 怎么计算呢？它的公式如下：

$$\mathrm{PR(A)} = (1-\mathrm{d}) + \mathrm{d}\sum_{\mathrm{i=1..n}} \frac{\mathrm{PR(T_i)}}{\mathrm{C(T_i)}}$$

其中，网页$\mathrm{T_1}...\mathrm{T_n}$链向网页 A，$\mathrm{C(A)}$表示网页 A 的外链数量，d

为阻尼系数（0<d<1）, PR(A)就表示网页 A 的 PageRank 值。

从公式中就可以看出 PageRank 的思想，但是现在又有个问题出来了，怎么计算？看公式很容易想到的是迭代，给每个网页一个初始的 PageRank 值，反复迭代直到收敛就得到所有网页最终的 Pagerank 值了，但是问题是：这个迭代问题是否真的收敛？庆幸的是，的确是收敛的，大家有兴趣的可以看下论文中的证明《Deeper Inside PageRank》。

至此你已经知道了什么是 Pagerank 了，PageRank 在实际计算的时候其实用的是矩阵形式，那问题又来了，假设有十亿个网页，那么这个矩阵就会有一百亿亿个元素（每个网页自然可能链接到其他任何网页，一个网页肯定是和少量网页有链接关系，和绝大多数都没有链接关系，所以这个矩阵是个稀疏矩阵），如此大的矩阵相乘运算，计算量是非常大的，那么怎么解决呢？就是使用前面讲的分布式计算框架 MapReduce 来计算了。

5.3 相似度计算

相似度计算是一个很重要的东西，大家可以想一想，有多少地方可以使用到：搜索引擎中计算 query 和文档的相关度？问答系统中计算问题和答案的相似度？广告系统中计算 query 和广告词的匹配程度？推荐系统中要给某个用户推荐某件物品，是否要计算这件物品和这个用户兴趣的相似度呢？太多太多了，接下来就介绍些相似度计算的方法。

相似度一定是指两个东西（姑且分别用 Q 和 D 表示）的相似度，

而这两个东西可以是任何形式的，比如，文本，图片，声音等等，最终要计算相似度，那必须把这些东西抽象成数学形式，说白了，就是怎么用数字把这些东西表示出来，一般会表示成向量或者矩阵。那如果表示成了向量，计算相似度就可以使用大家在数学课上学的知识了。目前来说，大致有这么两类方法：

●距离方法

常用的距离方法有：余弦距离，欧氏距离，汉明距离，明氏距离，曼哈顿距离，Jaccard 距离和 Jaccard 相似系数，皮尔森相关系数，编辑距离等等。这些所谓的距离其实都是一些固定的公式而已，没什么难的。编辑距离其实和上面的距离有些不同，它是指两个串，由一个变成另一个所需的最少的编辑次数，这个编辑就包括：替换，插入，删除操作。余弦距离是用的比较多的，它的公式如下：

$$\mathrm{sim(Q,D)} = \cos(Q,D) = \frac{\sum_{i=1..n} Q_i \times D_i}{\sqrt{\sum_{i=1..n} {Q_i}^2}\sqrt{\sum_{i=1..n} {D_i}^2}}$$

其中，Q_i和D_i分别为向量所在位的值。$\mathrm{sim(Q,D)}$越接近 1 越相似，越接近 0 越不相似。

还有皮尔森相关系数在推荐系统用的也较多，它的公式如下：

$$\mathrm{sim(Q,D)} = \mathrm{r}(Q,D) = \frac{\sum_{i=1..n}(Q_i - \bar{Q}) \times (D_i - \bar{D})}{\sqrt{\sum_{i=1..n}(Q_i - \bar{Q})^2}\sqrt{\sum_{i=1..n}(D_i - \bar{D})^2}}$$

其中，，Q_i和D_i分别为向量所在位的值，$\bar{Q}$和$\bar{D}$分别是 Q 和 D 的平均值。$\mathrm{r}(Q,D)$的范围是-1（弱相关）到 1（强相关）。

●Hash 方法

Hash 方法主要有： minhash 和 simhash。minhash 的主要目的是降维，它的主要原理是基于这个结论：两个集合经随机转换后得到的两个最小 hash 值相等的概率等于两集合的 Jaccard 相似度。而 simhash 是通过设计一个 hash 方法，使要内容相近的事物生的 hash 签名也相近，hash 签名的相近程度，就反映出了事物间的相似程度。很少使用这两个算法来计算文本相似度，一般都是用来网页去重，去重也算是相似度的一种应用，越相似的就越是重复的。

好了，既然学会了这么多计算相似度的数学方法，那么应用到文本上又需要怎么计算呢？也是有一些模型的，我从特征的角度把它分为如下四种：

Bool 模型

最早的计算相似度的模型就是 Bool 模型了，假设我们要计算两个句子（Q 和 D）的相似度，它们自然都是用一些词组成的，怎么量化使得可计算呢？那就用向量来表示，比如表示成（0，…，1，…，0，1，…，0，1，0…）。为什么向量里面那么多…呢，那是因为我们必须把向量都要统一起来，意思就是说，哪个词在向量的哪个位置上，都必须是确定的。要想确定，那就必须把所有的汉字都按某个序（字典序）排出来，然后各个词对号入座，这样整个向量的长度就是词典中词的个数。Q 和 D 中的每个词分别出现在什么位置那么就给相应的位置标记上 1，其他地方都为 0，这样表示成向量之后就可以计算 Q 和 D 的余弦相似度了，真正计算的时候会有些技巧，没必要都写成这么长的向量。

tf*idf 模型（增加词权重特征）

上面的模型已经可以用来计算相似度了，但是这时候，聪明的读者就发现问题了，上面模型中向量只用了词是否出现，这样不对啊，有些词重要，有些词不重要，那么不同的词对计算相似度应该要有不同的贡献，显然 Bool 模型没法表示啊！没错！提出了问题，那么就应该改进，这时候就有了 tf*idf 模型。Bool 模型是只要这个词出现，那么它的位置上就是 1，否则就是 0，而 tf*idf 模型却有不同，只要这个词出现，那么它的位置就是该词的 tf*idf，否则就是 0，看到了吧，利用了词的局部信息 tf 和全局信息 idf，计算方法也是余弦相似度，即Q_i和D_i分别为向量所在位的 tf*idf（没有出现的词的 tf 自然为 0，那么 tf*idf 也就是 0 了）。

BM25 模型（增加了长度特征）

非常好，tf*idf 模型相比之前的 bool 模型有了很大的优化，但是这时候又有人提出问题了，既然模型中用到了 tf，那么句子越长，潜在的 tf 就可能越大，也就是说，长的句子会比短的句子沾光。有了问题，接着优化，那么就出现了更好的模型 BM25 模型，BM25 模型相比 tf*idf 模型，多利用了一个长度特征，大家看它的公式就会发现，在分母有个与长度相关的式子（有些文献公式稍有不同）：

$$\mathrm{sim}_{\mathrm{BM25}}(\mathrm{Q},\mathrm{D}) = \sum_{\mathrm{i}} \frac{(\mathrm{k}_1+1)\mathrm{tf}_{\mathrm{q_i}}}{\mathrm{tf}_{\mathrm{q_i}} + \mathrm{k}_1[(1-\mathrm{b}) + \mathrm{b}\frac{|\mathrm{D}|}{\mathrm{avgdl}}]} \times \mathrm{IDF}(\mathrm{q_i})$$

k_1,b 为参数，|D|为文档长度，avgdl 为文档平均长度，具体公式的讲解可以参考一下 wiki 或论文，比如：《The Probabilistic Relevance

Framework:BM25 and Beyond》

Proximity 模型（增加了位置特征）

BM25 又优化了不少啊，已经算很不错了，但是又有人说了，你这个模型不好啊，比如我有个句子 Q（老大/的/幸福），分别去和两个句子 D1（老大/的/幸福）和 D2（幸福/的/老大）计算相似度，发现两个的分数一样啊，但是很显然 Q 和 D1 的相似度要比 Q 和 D2 的相似度要高啊！没错，那就接着优化，这时候就出现一些优化算法了，这些优化算法又多利用了一个位置特征，比如这篇论文《Term Proximity Scoring for Keyword-Based Retrieval Systems》，大家可以看到，其实它是在 BM25 模型后面又加了一个与位置相关的公式，这样 Q 和 D1 的相似度就会比 Q 和 D2 的相似度要高。实际使用的时候也不一定就严格按照论文的公式来，自己可以设计不同的位置公式来计算。

语义特征模型（增加了 Topic 特征）

前面的那些模型已经很不错了，能解决大多数问题了，但是又有人提出了问题：你上面的模型都是**关键词模型**，也就是说模型中的词必须严格一样，比如，要想和句子（计算机/好用）匹配，其他句子必须出现“计算机”或者“好用”，否则得分都是 0，这显然也不合理啊，比如我另外有个句子（电脑/耐用），它和句子（计算机/好用）肯定不应该是 0 啊，或多或少都有一定的相关度啊！没错，这时候就又有人提出改进的算法：语义特征模型，也就是在计算相似度的时候加入了语义特征，最简单的就是同义词（其实任何两个词之间都有相似度分

数，同义词只是个特例，它们的相似度高而已）。那么怎么得到语义特征呢？topic model（回顾下前面讲的主题模型，它计算出了所有的$P(z_k|d_i)$和$P(w_j|d_i)$，z_k就是主题），一般很少直接使用语义模型，而是和关键词模型加权：

$$sim = \lambda sim_{term}(Q,D) + (1-\lambda) sim_{TM}(Q,D)$$

其中$sim_{term}(Q,D)$就是前面讲的关键词模型，而$sim_{TM}(Q,D)$就是将要讲的语义相似度模型。计算这个$sim_{TM}(Q,D)$大致也有这么几种：一种是语言模型：

$$sim_{TM}(Q,D) = P(Q|D) = \prod_{w \in Q} P(w|D)$$

$$= \prod_{w \in Q} (\lambda P_{LM}(w|D) + (1-\lambda)(\sum_{k=1..K} P(w|z_k)p(z_k|D)))$$

另一种是 cos，不同在于计算权重不是使用 tf*idf，而是使用 topic model 的结果，即权重就是$P(w_j|d_i) = \sum_{k=1..K} P(w_j|z_k)p(z_k|d_i)$；还有方法是将 Q 和 D 的相似度看成是它们主题分布的相似度，也就是直接计算$P(Z|Q)$和$P(Z|D)$的相似度；另外有些模型使用了其他的方法（例如 KL 距离）来衡量（为什么两个 KL，因为 KL 不满足对称性，只有这样，才可以使得$sim_{TM}(Q,D)$和$sim_{TM}(D,Q)$得分相同）：

$$sim_{TM}(Q,D) = 1 - \frac{1}{2}[KL(P_{(Z|Q)}||P_{(Z|D)}) + KL(P_{(Z|D)}||P_{(Z|Q)}]$$

细节大家可以参考一些论文，比如这几篇《Learning the Similarity of Documents:An Information-Geometric Approach to Document Retrieval and Categorization》、《LDA Based Similarity Modeling for Question Answering》和《Regularizing Ad Hoc Retrieval Scores》。

句法特征模型（增加了句法特征）

有了语义模型，算是改进不少了，但是还有问题，比如两个句子 A（我的显示器的颜色）和 B（我的显示器变颜色），用以上的模型算的分数都不低，但是这两句话意思完全不一样，分数不应该高啊，是的。这时候就有人加入句法特征来优化了，也就是把两个句子的句法树的匹配程度考虑了进去。

尽管有了这么多模型，从搜索引擎关键词匹配的程度来说已经可用性非常高了，但是要想计算好真正意义上的语义相似度其实还是挺困难的。

深度表示模型（增加语义特征）

在深度学习的时候讲过，可以把一个词表示成向量，那么如果句子也能表示成语义向量，计算相似度就非常容易了，随着深度学习的发展，这种思路成了可能，在微软的论文《Learning Deep Structured Semantic Models for Web Search using Clickthrough Data》中提出了一种

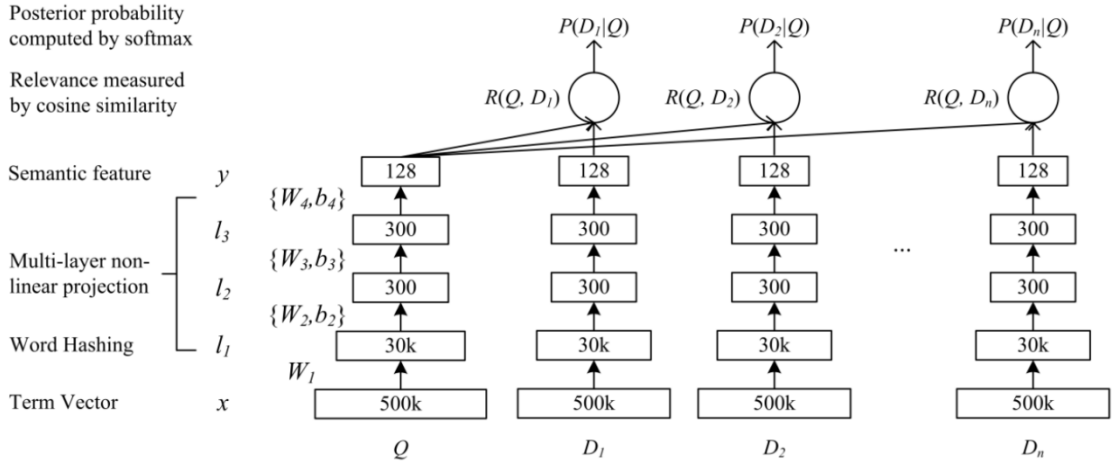
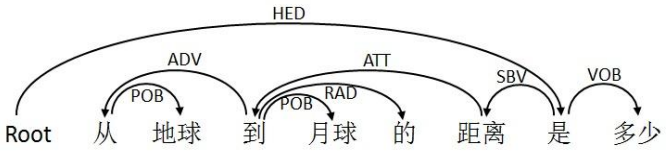


图 5.1

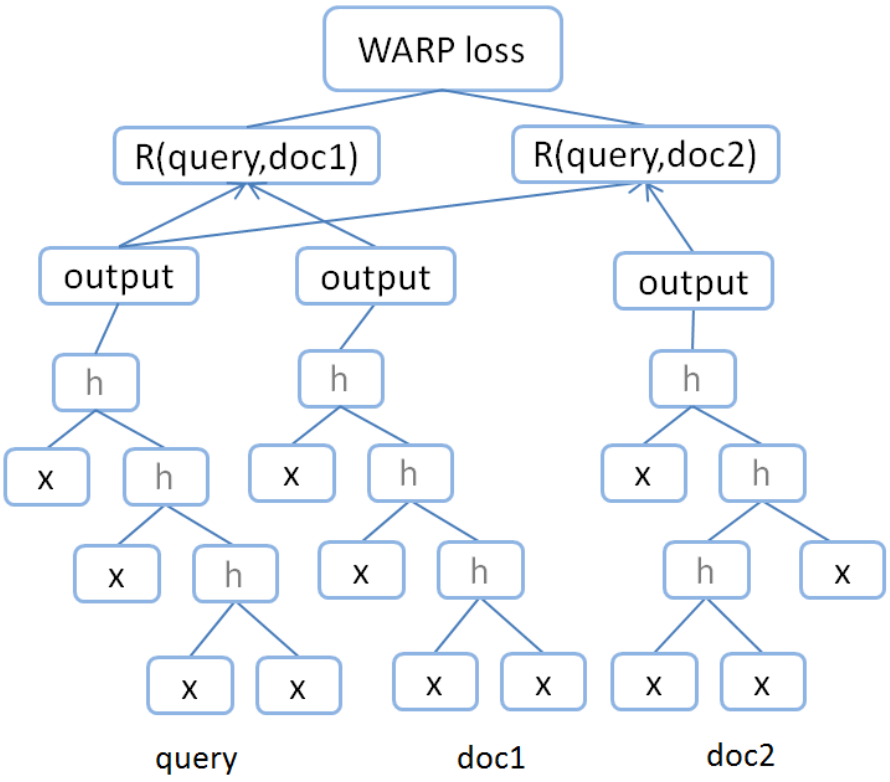
方法（如图 5.1）将句子逐层表示成语义空间的特征向量，然后就可以

计算句子相似度。这类方法是有监督的学习方法，需要线下标注一批相似问题对（随机取其他问题对作为反例）。

在我们的句子相似度算法中是由多个模型共同组成的，其中一个深度学习模型就是在句法分析树上使用 RNN 模型训练得到的，效果还不错（这种方法个人认为语料更重要些），简单说下这个模型，大家都知道句法分析树的结构如下图 5.2(a)所示，我们要在这种句法关系上做 RNN，如图 5.2(b)所示。



(a)



(b)

图 5.2

其中，x 就是每个词的向量，W 是每种关系及每个词对应的参数，

h 就是隐藏层，它的计算公式如下：

$$\mathrm{h_n} = f(\mathrm{W}_v \cdot x_w + \mathrm{b} + \sum_{k \in K(n)} \mathrm{W_{R(n,k)}} \cdot \mathrm{h_k})$$

这个模型不光可以用来计算句子相似度，还可以用来计算问题和答案的匹配得分进行排序。

对于计算句子相似度，单纯使用这个模型，还是有很多解决不了的问题，所以我们还结合了其他模型来计算句子相似度。

第六章、搜索引擎是什么玩意儿

正是 Google 这家搜索引擎公司的巨大成功，才把文本处理相关的技术推向了一个新的高度，其实经过这么多年的发展，搜索引擎技术其实已经很成熟了，如果要搞一个网页搜索出来其实难度降低了不少，但是要把效果搞好，还是有门槛的，假如搞一个垂直搜索引擎其实已经没什么门槛了，那么搜索引擎到底是什么东西呢？这章我们就揭开它的面纱，我更多的是以自己的理解和经验从一个实用的角度去介绍。

6.1 搜索引擎原理

搜索引擎的工作原理是怎样的？很多人都会告诉你一个所谓的大致流程：建索引，然后读索引，归并，计算相关性返回给用户。但是他们并没有告诉你为什么是这样的一个流程，也就是大多数人其实不知道为什么, 那么我们就来分析下这个为什么？

假设 Q 为用户要查询的关键词（Query）；D_i为所有网页集合中第 i 个网页；$\mathrm{P}(D_i|\mathrm{Q})$就表示给定一个 Q，第 i 个网页满足了用户需求的概率，那么搜索引擎干的事就是根据用户的输入 Query（当然它还包括一些隐性的信息，比如地域等），在所有的网页集合中计算$\mathrm{P}(D_i|\mathrm{Q})$，并排序返回给用户。如果网页集合少的话，比如几千个，那么我们完全可以按照之前的相关性方法把 Query 和每一个网页的相关性计算出来，然后排个序。但是现在问题来了，现在互联网上的网页数量多的惊人（google 号称索引了 1 万亿个网页），试想，这么多网页，刚才那种方法计算量太大必然行不通，因为绝大多数网页是和用户查询不相关的，

不需要去计算，那么就要想办法解决这个问题（过滤掉不相关的）？既然没办法直接计算$\mathrm{P}(D_i|\mathrm{Q})$，那么就计算它的等价形式或者近似形式（这是搞研究很重要的一个方法，想想看前面哪些机器学习模型也用到了这个方法），使用贝叶斯公式：

$$\mathrm{P}(D_i|\mathrm{Q}) = \frac{\mathrm{P}(\mathrm{Q}|D_i)\mathrm{P}(D_i)}{\mathrm{P}(\mathrm{Q})}$$

我们分析下右边式子的各个因子：

$\mathrm{P}(\mathrm{Q})$：它只与 query 自身有关，对于同一个 query 来说，它都是一样的，所以我们就可以不计算这个值了。但是这个因子对分析整个 query 来说还是有用的，对于热门 query，$\mathrm{P}(\mathrm{Q})$较大，必须使右边式子的分子更大才能满足用户需求，也就是说对排序要求越高；相反，对于冷门 query，$\mathrm{P}(\mathrm{Q})$较小，右边式子的分子不用太大都能满足用户需求，也就是说对排序要求较低。其实很容易理解，比如用户搜“腾讯网”（热门 query），如果哪个搜索引擎第一条不是腾讯网的首页链接，那么这个搜索引擎的效果就太差了，没法容忍；比如用户从某处复制了一串很长的文字（冷门 query），放到了搜索引擎中去搜，那么只要搜索引擎能返回包含该串的结果就行了，至于排到第一位还是第二位或是其他位置，只要不是太靠后，其实都是可以接受的。

$\mathrm{P}(D_i)$：第 i 个网页的重要程度，这个因子就非常好，因为它与用户查询无关，完全可以线下计算好，还记得前面讲的 PageRank 吗？这个值你就可以简单的认为就是网页的 PageRank 值（现在的搜索引擎其实还是融入了很多其他计算因素，可以认为 PageRank 现在只是其中的一个特征）。

$P(Q|D_i)$：这个因子可以理解为，已知一个给定的网页满足了用户的需求，那么用户的查询是 q 的概率，这个值反映的是一个网页满足不同需求之间的比较。但是这个值也很难直接计算，因为没办法穷举出所有的 Query，怎么办？我们假设 Query 是由若干个词组成（$Q = t_1, \dots, t_n$），那么：

$$P(Q|D_i) = P(t_1, \dots, t_n|D_i)$$

一种策略，如果我们认为$t_1, \dots, t_n$之间相互独立，那么就有：

$$P(Q|D_i) = P(t_1, \dots, t_n|D_i) = P(t_1|D_i)P(t_2|D_i) \dots P(t_n|D_i)$$

看到了吗？后边式子只与单个的词有关，从 Query 级别转换成词级别就意味着可以线下计算了（从工程的角度，把线上的计算移到线下计算也是优化的方法之一），因为每个网页也都是词组成的，$P(t_1|D_i)$，$P(t_2|D_i)$，…，$P(t_n|D_i)$都可以线下计算好存起来，需要的时候取出来就可以了，同一个词t_i会在若干个文档中出现，那么只要按某一种方式存储起来，就可以一次全部取出来t_i所在的所有文档，这种存储方式就叫**索引**。这下明白为什么要建索引了吧。一个 Query 由若干个词组成，上面的假设$t_1, \dots, t_n$之间相互独立只是其中一种分解策略，根据具体的情况还会有不同的分解策略，比如某些词必须合并到一起，某些词可以替换成另外的词等等，这些都是 Query 分析及页面分析要完成的事情，无论运用哪种或者多种分解策略，都可以分别将它们查到的结果（每一种策略的结果叫一个队列），按它们满足各自查询的概率最终归并起来。

以上这种解释书籍上把它叫做概率语言模型，它也是我比较喜欢的一个模型，因为它回答了很多为什么。大家知道，计算语言模型需

要平滑（如果某个$P(t_i|D)=0$，则整个得分就为 0），但是很多商用引擎很少使用平滑技术，而是用 query 分析来处理。

从上面的式子读者就会说搜索引擎相关性最重要的就是计算$P(Q|D_i)$和$P(D_i)$，不准确！还有一个比较重要的，就是左边的那个式子$P(D_i|Q)$，也许，你就会问：左边那个不是不好计算吗？能计算的话，还用右边的式子干什么呢？解释一下，大家看到由于右边式子中的$P(Q|D_i)$不好计算，所以对它进行了变形，例如：

$$P(Q|D_i)=P(t_1,\dots,t_n|D_i)=P(t_1|D_i)P(t_2|D_i)\dots P(t_n|D_i)$$

这个式子是有个前提的，就是认为$t_1,\dots,t_n$之间相互独立，这是个假设，换句话说，右边式子要想能计算必须在某个假设（上面假设只是其中一种）下，假设意味着和实际是有区别的，所以得有个方法来修正不在假设范围内的情况。因此如果直接计算$P(D_i|Q)$，哪怕方法很简单都会有很大的帮助，如果不知道所有的 Query，解决一部分 Query 也可以，那用什么方法呢？**点击**！现在搜索引擎都会记录用户在哪个 Query 下点击了哪些网页，以及顺序，时间等。最简单的情况，给定一个 Query，$P(D_i|Q)$是不可以用所有用户在该 Query 下点击第 i 个文档的次数来表示，点的越多说明该文档越能满足用户需求，实际使用中是按照点击在已有相关性上调权，而不是直接用来排序。

6.2 搜索引擎架构

知道了搜索引擎原理，那么整个搜索引擎的工作流程是怎么样的呢？如图 6.1 所示，分别是线下计算和线上计算两大模块。

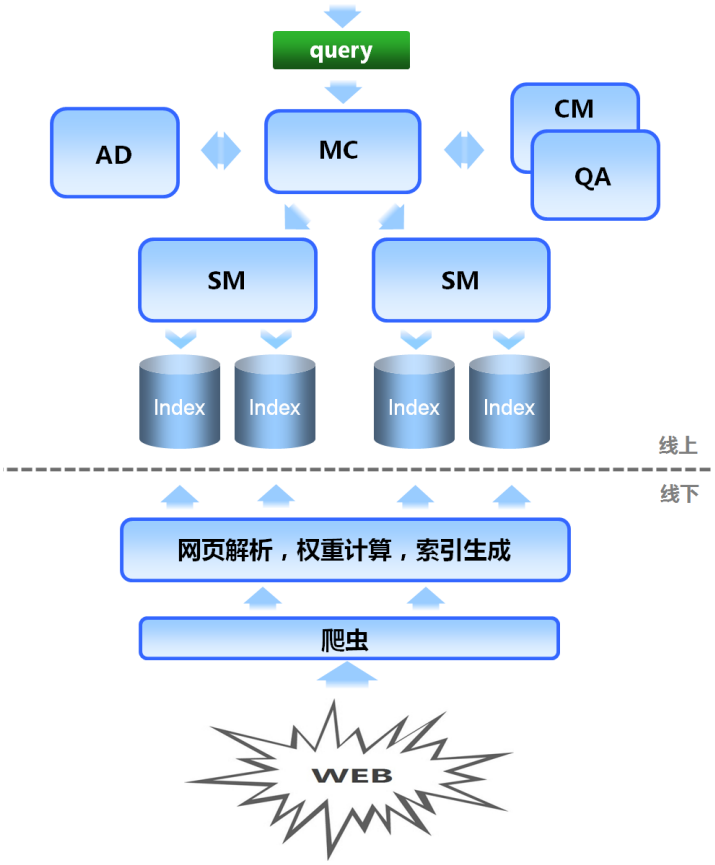


图 6.1

线下模块：

爬虫会从互联网上抓取所有网页，抓取下来之后就要做很多事情：网页解析，权重计算（包括计算 PageRank 值），然后要对网页建立索引，一般网页是要分库分机来建索引的。

线上模块：

当用户输入一个 query 之后，它首先会发到主控模块（MC，main control），该模块接着会把 query 输入给 query 分析模块（QA，query analysis），然后获得 query 分析结果，之后把所有这些信息发给每一个搜索模块（SM，search model），搜索模块就会从每个索引库（index）中读相关的索引并且筛选排序，将结果返回给主控模块，主控模块这时候还要经过一系列调权（其中包括点击调权，所以也要调用点击模型（CM，click model））进行 rerank，最后从摘要库（AD，abstract data）里把相应的

摘要获取到一并将结果返回给用户。这就是整个搜索的流程。

6.3 搜索引擎核心模块

搜索引擎涉及到的内容很多，在这节我们只看下几个核心模块都干些什么事情。

爬虫

爬虫的主要职责就是从互联网上抓取网页，它要求抓的尽可能全，尽可能快。还记得前面说过整个网络因为链接构成了一个图，既然是一个图，遍历图的方法就有两种：深度遍历和广度遍历，所以爬虫一般的工作方式就是，首先给定一些初始链接种子，然后对这些链接进行深度或者广度遍历获得更多的链接，这里有个问题需要注意，就是不要反复再去抓取已经抓取过的链接。

网页解析计算

这个过程包括：网页解析，权重计算，索引生成。

网页解析。网页抓取下来之后就要对它解析，格式转换（网络上会有不同格式例如 html、doc、pdf 等的文件）、去掉 html 等标签，解析出网页标题，正文，去掉网页上的广告等等。

权重计算。权重计算主要包括这么几个步骤：特征提取，权重计算。特征提取就是从网页中提取出相应的特征信息，比如：title，meta 等，还会根据网页的基础特征计算一些高级特征出来，比如：主题匹配度等。权重计算包括 term 级别的计算和网页级别的计算。term 级别计算相当于计算 term 和该网页之间的相关度（就是前面说的$\mathrm{P}(\mathrm{t_n}|D_i)$），

不同的词出现在不同的位置，它的重要程度就不一样，比如在 title 中就比在正文中出现要重要；就算都出现在正文中，由于它在不同位置或者有不同的标签也会对它的权重计算有影响。网页级别的计算就是计算该网页的一些特征，比如：时间因子，页面是否丰富，网页主题，超链权重（PageRank）等，这些信息会用于调权。

索引生成

索引生成。在索引生成之前，还要对进行**索引扩充**，对 term 的同义词（不同的搜索引擎有不同的扩充策略）等也要进行索引，最终生成索引库。因为数据量太大，索引库一般是要分库的，分库分为两个维度：横向和纵向。不同的搜索引擎会有不同等级的分库法，但是总的原则就是分为高质量库和低质量库，其实像个金字塔，塔上面是高质量库且网页数量少，塔下面是低质量库且网页数量大，纵向就是指先在高质量库中检索，如果满足需求就返回，否则再去低质量库中检索；不管是高质量库还是低质量库都有很大的数据量，不可能一台机器搞定，所以要把它们分成小库分别部署到不同的机器上，所以横向就是去检索高质量库或者低质量库的时候，会去同时检索每台机器的小库，然后汇总归并。索引结构是什么样子的，在讲检索的时候再说。

以上讲的都是线下需要完成的一些工作，下面我们再看看线上需要做哪些工作。

query 分析模块（QA）

query 分析在搜索引擎中是很重要的一个模块。它的主要职责就是对用户输入的 query 进行各种分析计算供下游检索使用。它主要包括这么几个模块：

1、term 级别分析

term 级别主要包括这么几个任务：分词/专名识别，term 重要性，term 紧密度。

分词/专名识别。对中文的处理，分词肯定是少不了的。专名识别方法包括词表识别（影视名、音乐名等）和机器学习识别（人名、地名、机构名等），还记得前面讲的 CRF 模型吗？专名识别的主要用处其实还是后面说的 term 紧密度上。

term 重要性计算。一个 query 会包含多个 term，那么就要给每个 term 计算一个权重，权重越高越重要。一般来说计算一个 term 的重要性使用的方法都是 tf*idf，tf 是局部信息，idf 是全局信息，但是在 query 中一般来说 tf 都是 1，如果使用 tf*idf 的话，相当于只是使用了 idf，效果肯定不好，所以优化的思路都是根据 query 的特点对“tf”进行优化，也就是用某一种方法计算局部信息，然后和 idf 相乘。当计算出来 term 重要性之后，它有什么作用呢？一是用来计算相关性，二是对不重要的词可以省略，扩大召回。

term 紧密度。term 紧密度是用来计算 term 之间的位置信息的。举例来说，有个 query 是“老大的幸福在线观看”（分词结果为：老大/的/幸福/在线/观看），在这个 query 中，“老大的幸福”就是最紧密的，因为它是个**专有名词**（电视剧名），意味着召回的结果中，它们必须合并到一起，不能分开；“在线/观看”是其次紧密的，召回的结果中，它们尽量合并到一起，分开也可以接受；“老大的幸福”和“在线观看”是不紧密的，召回的结果中，这两个短语间可以出现其他词语。看到了吧，term 紧密度也是用来相关性排序的，和 query 位置信息越相同

的自然排的越靠前。

2、query 级别分析

query 级别分析最主要包括这么几个任务：query 需求识别，query 时效性判断，当然还有地域信息，属性归一等。

query 需求识别。query 需求识别说白了就是看该 query 是什么类别的，比如：是视频需求，是百科需求，是小说需求等等，因为不同的类别对相应的网页有不同的提权。一般 query 需求识别的方法有这么几种：（1）模板匹配；（2）基于分类思想的识别方法；（3）根据点击反馈来判断意图；

query 时效性判断。时效性一般分为三种：泛时效性，周期时效性，突发时效性。泛时效性是指有些 query 永远具有时效性特性，比如：减肥，永远是热门话题；周期时效性是指具有周期性的事件，比如：高考，世界杯等；突发时效性是指突然发生的事件，比如：哪里又发生军事冲突了，哪个高官又被小三反腐了等等。前两个时效性根据历史信息是可以积累的；突发时效性可以根据 query log 的分布变化检测出来。

3、query 变换

query 变换其实也是属于 query 级别的分析，但是它比较重要，单独拿出来讲，query 变换说简单点就是一个 query 可以替换成另一个 query 而不改变原来的意思，主要包括：同义改写，纠错改写，省略变换等，还有些关联扩展等。

同义改写。同义改写其实是指 query 中某些 term 可以替换成其他同义的 term 而不影响这个句子的意思，例如：“招商银行官网”-“招

行官网”。

纠错改写。纠错改写是指 query 中有些是用户输错的，需要根据用户的意图把它改写正确，例如：“天龙八步在线观看”-“天龙八部在线观看”。

省略变换。省略变换是指 query 中有些词省略之后不影响整个意思，例如：“招行客服电话”-“招行电话”。

其实 query 变换不管哪种变换，技术都是一样的，目的都是 term A 替换成 term B 后整个句子的意思没有变化，省略变换其实就是 term A 替换成空，是一个特例而已。在 query 变换中一定要强调的是，query 变换一定是句子级别的替换，否则会出现严重的 badcase，例如：“南航网上值机”如果替换成“南京航空航天大学网上值机”就是错误的，意思偏离了（南航的同义词是南京航空航天大学和中国南方航空）。大多数搜索引擎的替换其实都不是句子级别的替换，有的直接替换成同义词然后去读索引，然后对同义结果降权，这其实是不好的处理方法。既然 query 变换是句子级别的，那就相当于是一个新的 query，既然是新的 query，那就像对待原 query 一样去检索，不同的 query 召回的结果放到不同的队列中，最后根据权重归并起来。

Query 变换技术其实和机器翻译有点类似，给某个 query S，找到最佳的替换 query T 的模型就是：

$$T^* = \text{argmax}_T P(S|T)P(T)$$

其中 P(T)就是语言模型，P(S|T)就是 query T 替换成 query S 的转换概率。这样就遇到了和机器翻译差不多的一些问题：替换对的挖掘及概率计算，语言模型的参数估计，最优解的搜索算法等等。

主控模块（MC）

主控模块也是很重要的一个模块。它主要有两个职责：调度和Rerank。

调度。调度其实比较好理解，它把接收到的用户 query，发给 QA，获得 query 分析结果，然后把结果发给 SM 模块，获得结果，然后进行 Rerank 操作，然后从 AD 模块中获得摘要信息，返回给用户。

Rerank。在 MC 模块已经拿到了各个 SM 模块返回的结果。这时就可以对这些结果进行调权，大概有这么多的因素：页面质量调权，Query 需求调权，时效性调权，多样性调权，点击调权等等。**点击模型**算是很重要的一个模块了，它需要线下对 query 的点击行为做很多计算，它有好几种模型，读者可以参考下这篇文章《A Dynamic Bayesian Network Click Model for Web Search Ranking》，对 DBN 模型讲的非常清楚。

还记得前面说的每个 query 变换结果返回一个队列，那么返回的结果就有多个队列，这就包括单队列内调权和多队列调权。当然，也可以从 AD 模块获得网页的正排信息，进行更精细的排序。

搜索模块（SM）

搜索模块也是非常重要的模块，它负责把结果从索引库中召回并按照相关性返回给 MC 模块。但是大型搜索引擎并不是像 6.1 原理中讲的那样，计算相关性返回给用户，因为计算量太大，而是逐步筛选再返回给用户。首先说下整个流程，再说部分细节。

SM 模块首先从索引库中获得最基本的结果，这时候它会进行**一次过滤**，一次过滤的方法一般就是先索引归并，然后计算相关性：term 权重*位置信息，term 权重其实就是 query 分析时计算的权重和线下计

算 term 权重的平均；看到了吧，一般搜索引擎的相关性公式只到了 proximity 层面（回顾前面相似度计算），当然也可以加入语义信息。当一次筛选之后，还会进行**二次过滤**，二次过滤会根据页面属性进行筛选（比如页面质量，死链等）。前两次筛选其实是在单库（记得前面说过 index 是分库的吗），最终把多库结果汇总之后也可能还要进行**过滤**。这样 SM 模块就得到了最终的 url 列表。

这里可以看到，排序的方法都是人工设定公式，自然会有不少参数，而调参又是很头疼的一件事情，于是，人们就很自然想到用机器学习来解决这个问题。还记得前面讲的最优化问题和机器学习吗？于是乎就产生了一种方法：Leaning to Rank，用机器学习的方法来排序，如图 6.2 所示。

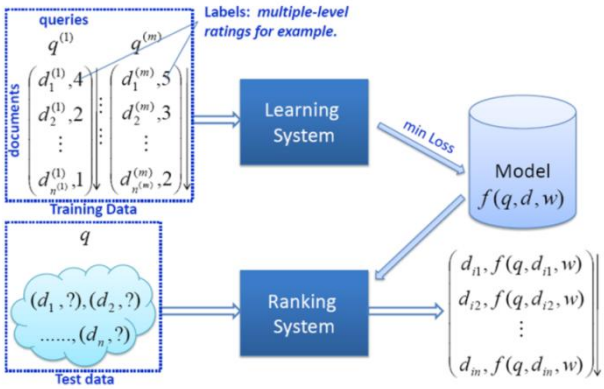


图 6.2

它的模型主要有三种：Pointwise，Pairwise 和 Listwise，说白了就是把排序模型转换成分类或者回归机器学习最容易搞定的问题，特征就包括 query 和 doc 的特征以及相关的一系列特征，Leaning to Rank 的论文已经很多了，稍微有点机器学习基础看懂它的原理一点儿都不难。在这儿简单说下这三个模型，1）Pointwise 方法的主要思想是将排序问

题转化为多类分类问题或者回归问题，也就是将（query，di）的标注结果（一般五个类别，即 Perfect=5, Excellent=4, Good=3, Fair=2, Bad=1）作为一个类别。这就有个假设，就是 query 独立的，即只要（query, document）的相关度相同，比如都为“perfect”，那么它们就被放在同一个类别里，即属于同一类的实例，而无论 query 是什么。这种方法比较简单，效果也不是很好。实现 Pointwise 方法有 McRank。2）Pairwise 方法的主要思想是将排序问题转化为二元分类问题。对于同一个 query，在它的所有相关文档集里，对任两个不同 label 的文档，都可以得到一个训练实例对，比如说是（d1，d2）分别对应 label 为 5 和 3，那么对于这个训练实例对，给它赋予类别+1（5>3），反之则赋予类别-1。于是，按照这种方式，我们就得到了二元分类器训练所需的样本了。该方法不再是 query 独立的，因为它只对同一个 query 里的文档集生成训练样本的。但是它也有缺点，比如说 query1 对应的相关文档集大小为 6，query2 的相关文档集大小为 1000，那么从后者构造的训练样本数远远大于前者，从而使得分类器对相关文档集小的 query 所产生的训练实例区分不好。实现 Pairwise 方法有 RankSVM,还有 RankNet，FRank 和 RankBoost 等。3）Listwise 方法的主要思想是直接对排序结果进行优化。对于一个给定的 query，以及其对应的 document list，现在已经得到了一个标注好的分数列表 z（例如，每个文档的点击率等）。然后采用某种排序函数 f 给每个 document 打分，得到一个预测排序得分列表 y，然后再采用某种损失函数计算 loss（ListNet 采用交叉熵，即将 z 和 y 带入交叉熵公式中作为损失函数来用来学习参数）。实现 Listwise 方法有 ListNet，RankCosine，SVM-MAP 等。现在不少搜索引擎都已经开始

加入了 Leaning to Rank 方法，但并不是直接用 Learning to Rank 排序，而是用它来计算一些高级特征供上层排序使用。

前面我们一直在提索引，以及索引归并，那么索引到底是什么样子呢？索引结构简单说如图 6.3 所示（term->postings list）。建索引的过程就是把文档的正排信息（一个文档有若干个 term 组成，每个 term 有位置，权重等属性）建立索引（倒排信息，从 term 找到所对应的的文档）。

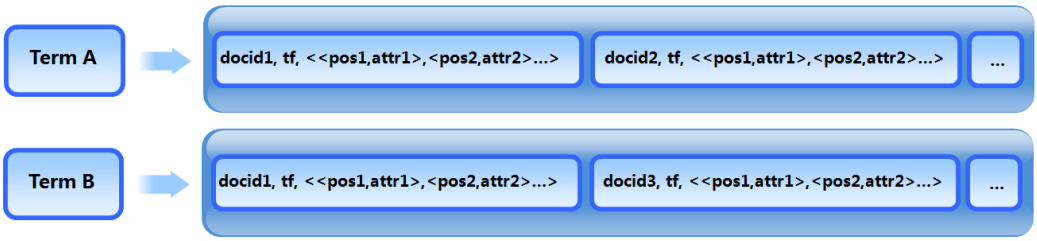


图 6.3

还记得前面在搜索引擎原理中讲的索引就是根据 term 很快找到包含它的文档，所以一个索引结构其实可以简单看作就是 key-list 结构。例如图 6.3 中，term A 在 docid1 和 docid2 中出现，且它在 docid1 中分别出现在 pos1 和 pos2 的位置上，且属性（例如权重）分别是 attr1 和 attr2，是不是很好理解，通常一个 term 后面的 list 中文档是要按某种方式排序的，一般是按照文档号递增来存储的，怎么排序取决于用什么算法来归并。好，现在设计好了索引结构，对所有文档建立了这样一个倒排索引库，那么剩下的问题就是，来一个 query，它分词之后有可能有多个 term，那么怎么找到都包含这些 term 的文档呢？也就是要怎么归并这些 key-list。索引归并主要有两种方法：TAAT 和 DAAT。

Term-At-A-Time (TAAT)：在 TAAT 的查询处理过程中，它每次只打开

一个 term 对应的倒排链（list），然后对其进行完整的遍历。所以它每处理一个文档只能得到这个词对这篇文档的贡献，只有处理完所有 term 的倒排链后，才能获到文档的完整得分。因此，TAAT 查询处理方法通常需要一个数组来保存文档的临时分数，这个数组的大小通常与文档集规模相当，所以当文档集规模很大时，这个额外数组存储开销会变的非常大。

Document-At-A-Time (DAAT)：在 DAAT 的查询处理过程中，它首先会打开各个 term 对应的倒排链；然后同时对这些倒排链进行遍历。每次对当前文档号最小的文档计算相关性得分，在处理下一篇文档之前，它会完整的计算出当前处理文档的最终相关性得分。因此，DAAT 的查询处理过程中，它只需使用较少存储空间来保存当前得分最高的文档及其得分，通常这些数据使用优先队列来存储。

当数据量较大的时候通常采用 DAAT 的方式进行查询归并，用的更多的是基于文档号递增排序的倒排索引结构。假设使用 DAAT 方式来归并的话，还是有问题，一般数据量大的话，每个 term 对应的倒排链（list）会很长，如果 list 中每个文档都遍历的话，性能还是不高，那么有没有办法跳过不可能出现在结果中的文档呢？有，跳表就是其中一种方法。

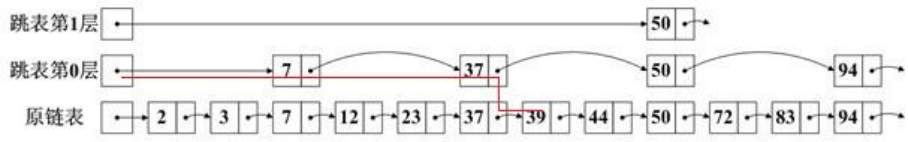


图 6.4

跳表可是一种简单且高效的数据结构，著名的开源 kv 系统 redis 和 leveldb 都使用了跳表作为它们的核心数据结构。跳表是一个有序链表，它在原链表上设置了若干层的有序链表，每一层都比上一层的节

点要少。比如要查找图 6.4 中 39 这个元素，就按照图中红色路线就可以查找到，比在原链表上查找路径要短（跳过了很多元素），这也就是它的优势。所以使用跳表可以加快链表归并。具体跳表的细节网上很多了，或者大家可以参考原论文：《Skip Lists: a Probabilistic Alternative to Balanced Trees》。

对于搜索一般有三个常用的查询表达式：AND、OR 和 NOT，这三种查询表达式也比较好理解，举个例子就知道了：A AND B 表示 A 和 B 必须同时出现；A OR B 表示 A 和 B 必须出现其中之一；A NOT B 表示出现 A 但不能出现 B。下面我们看看还有什么更好的算法能同时解决 AND 和 OR 查询，又能提高性能。

从前面使用跳表可以看出，跳表的优化只对 AND 查询有用，对 OR 查询不起作用（读者可以思考下为什么），所以要同时满足 OR 查询那就要另想方法来优化了，优化的思路就是剪枝限界，很常用的方法。在 DAAT 查询处理方式中，有两个经典的动态索引剪枝算法：MaxScore 和 Weak-AND（WAND，是 AND 查询和 OR 查询的扩展）。它们都是通过计算每个 term 的贡献上限来估计文档的相关性上限，从而建立一个阈值对倒排中的结果进行减枝，从而达到提速的效果（也就是有些文档不用再计算相关性了，直接跳过）。WAND 算法首先要估计每个 term 对相关性贡献的上限，最简单的相关性就是 TF*IDF，IDF 是固定的，所以就是估计一个 term 在文档中的词频 TF 上限，一般 TF 需要归一化，即除以文档所有词的个数，因此，就是要估算一个 term 在文档中所能占到的最大比例，这个线下计算即可。知道了一个 term 的相关性上界值，就可以知道一个 query 和一个文档的相关性上限值，就是他们共同

term 的相关性上限值的和，然后和一个阈值（一般是 Top-N 的最小值）比较，小于就跳过。MaxScore 和 WAND 算法都是精确的动态索引剪枝算法，其精确性主要通过以下三方面进行保证：(1)对候选文档估算的得分都是高估的，即估算得到的分数大于或等于文档的真实得分。(2)跳过的文档的估算分数都小于当前 Top-N 的阈值，即不存文档真实分数大于 Top-N 的阈值而未被加入到 Top-N 结果中的情况。(3)对于 Top-N 结果列表中的文档，都进行了完整的打分。即 Top-N 结果中的文档得到的文档与完全遍历查询处理对这些文档处理得到的分数是一样的。所以这两个剪枝算法得到的 Top-N 结果与完全遍历查询处理得到的 Top-N 结果是完全一致的。因此，它们是精确的动态索引剪枝方法。思路很好理解，就是把不可能得高分的文档减掉，不再进入后面的计算，具体细节大家可以参考下相关论文，例如《Efficient Query Evaluation using a Two-Level Retrieval Process》。还有一种做法是从词的角度进行限定，还记得前面讲的在 query 分析的时候计算 term 重要性，term 重要性的作用不光是计算相关性，还有一个作用就是，对于重要的词必须出现，对于较不重要的词出现更好但并不是必须，对于一点儿都不重要的词压根就可以不去读它的索引链，这就衍生出来三种词的查询逻辑：and，or，not（为了区分前面的查询表达式，用小写）。and 操作表示该词必须出现，or 操作表示出现更好，但是不是必须，not 操作表示该词不能出现，用的较少，主要还是 and 和 or 操作用的较多，这种说法是针对词来说的，也就是 and、or 和 not 操作都是单个词的属性（不要和前面的 AND 查询和 OR 查询混了，AND 查询和 OR 查询是描述词之间的关系），这样，读索引归并的时候就必须保证 and 操作的词必须出

现，否则直接跳过计算。这种做法和前面讲的动态索引剪枝算法最大的区别是它对 term 重要性分析的要求提高了，举个例子来说，一个 query 由三个 term：A、B 和 C 组成，计算出它们的权重分别是 0.9,0.89 和 0.88（权重区分度不大，意味着不一定 term A 就是最重要的，term C 就是最不重要的，除非 term 重要性分析的非常准确，但是 100%的准确率几乎不可能做到），那么如果在不存在同时包含这三个词的文档的前提下，为了增加召回（对于小数据量的搜索系统召回问题就更加凸显了），就有可能丢掉一个 term 来扩大召回，后者的做法那肯定是丢掉 term C（因为它的权重最低），然后召回同时包含 term A 和 term B 的文档；而对于动态索引剪枝算法来说，它有可能召回同时包含 term B 和 term C 的文档（自然会召回同时包含 term A 和 term B 的文档），这种情况下，动态索引剪枝算法就显得更好了。所以一个好的架构一定要注意的其中一点就是要尽可能降低短板效应对整个系统的影响。

上面讲的索引结构其实是较简单的一种形式，真正设计的时候就会很复杂了，会把不同的信息分到不同的结构中，较简单的比如会构建四个不同的数据结构，分别是词典库（存放 term 及其在索引库的位置偏移，一般常驻内存），位置库（存放 term 在文档中的位置信息，根据数据规模决定存放磁盘还是内存），属性库（存放 term 的其他属性，词频，权重等等，也是根据数据规模决定存放磁盘还是内存），跳表库（存放跳表指针，一般常驻内存）；查找的时候首先查词典库，然后根据位置偏移查找相关信息，对于磁盘索引的话，也可能会用到其他的一些数据结构（例如有序数组、B+树或者 LSM 树等，这些数据结构的目的就是提高读写性能，因为磁盘 I/O 操作性能不高，要尽可能的

减少磁盘 I/O 操作），有些索引还需要压缩存储等等，对一般的网页还有可能要分域（比如：标题域，正文域等），所以设计一个高效的索引系统也是需要很多的技术。

讲完了搜索引擎的技术，那么搜索引擎怎么评价呢？每次更新完算法都要知道效果是变好了还是变坏了吧，常用的搜索引擎评价指标有：NDCG，MAP，MRR，AB-testing 等等，我们只看下用的比较多的 NDCG 是怎么回事。

DCG（discounted cumulative gain）这种方法基于两点假设：(1)高相关性的文档比边缘相关的文档要有用得多；(2)一个相关文档的排序位置越靠后，对于用户的价值就越低。这种方法为相关性设定了等级，作为衡量一篇文档的有用性或者增益，DCG 方法的定义（有的文献公式会有不同）为：

$$\mathrm{DCG_p} = \mathrm{rel_1} + \sum_{\mathrm{i=2}}^{\mathrm{p}} \frac{\mathrm{rel_i}}{\log_2 \mathrm{i}}$$

$\mathrm{rel_i}$为在排序位置为 i 的文档的相关性等级（一般相关性等级分为 5 等：非常差（$\mathrm{rel_i} = 1$），差（$\mathrm{rel_i} = 2$），一般（$\mathrm{rel_i} = 3$），好（$\mathrm{rel_i} = 4$），非常好（$\mathrm{rel_i} = 5$））。为了便于平均不同查询的评价值，可以通过将每个排序位置上的 DCG 值与该查询的最优排序的 DCG 值（人工标注的）进行比较，得出一个归一化的值，即：

$$\mathrm{NDCG_p} = \frac{\mathrm{DCG_p}}{\mathrm{IDCG_p}}$$

$\mathrm{IDCG_p}$即为某一查询的理想的 DCG 值。

举例来说，一组 6 个文档的排序的相关性等级和理想等级分别为：

3,2,3,0,0,1

3,3,3,2,2,2

可以计算出它们对应的 DCG 值分别为：

3,5,6.89,6.89,6.78,7.28

3,6,7.89,8.89,9.75,10.52

那么$\mathrm{NDCG}_6 = \frac{7.28}{10.52} = 0.69$。

至此这个搜索引擎设计到的核心模块就已经讲完了，想必大家对搜索引擎已经不再陌生了，当然还有很多模块了，比如：输入提示、相关搜索、反作弊，网页去重等等工作也都是必不可少的。在工程上，还有一个很重要的功能就是缓存（Cache），cache 就是把经常用到的数据缓存起来，如果要再使用的话，不用发送到下游直接读取就可以了，所以 cache 的最大作用就是提高性能，在好多模块都可以使用 cache 来提高性能（如果 cache 命中率不高的话，花在 cache 数据替换的时间反而很高，也就没必要使用了），而且 cache 还有个时间周期的问题，就是说 cache 是缓存一定时间段内的数据，过了这个时间段就要清除，重新生成 cache，这样能保证新数据进来。总体来说，网页搜索引擎还是有门槛的，数据量大，计算要求也高，但是对于垂直搜索引擎就简单不少了，一般垂直搜索引擎都是自己的数据，很多模块都不需要了（例如：PageRank 计算，爬虫，解析等），而且数据量小，计算要求也低。搜索引擎还有很重要的一个工作，就是根据 query log 和点击数据进行各种挖掘工作，比如，新词，扩展等等，可以说 query log 和点击数据算是搜索引擎特有的优质资源。

6.4 搜索广告

搜索引擎靠什么盈利呢？广告。要想了解广告，几个常用术语必须要知道：

CPC(Cost Per Click)：每次点击的费用，按照广告被点击的次数收费。

CPM(Cost Per Mille/Cost Per Thousand Click-Through)：每千次印象（展示）费用，广告每展示 1000 次的费用。

CPA(Cost Per Action)：每次行动的费用，按照用户对广告所采取的行动（完成一次交易、产生一个注册用户等）收费。

CPS(Cost Per Sales)：以每次实际销售产品数量来收费。

CPL(Cost Per Leads)：根据每次广告产生的引导（通过特定链接，注册成功后）付费。

CPD(Cost Per Day)：按天计费。

CTR(Click Through Rate)：广告的点击率。

CVR(Click Value Rate)：广告的转化率。

RPM(Revenue Per Mille)：广告每千次展示的收入。

广告大致分为：品牌广告和效果广告。品牌广告一般是用来树立品牌形象，目的在于提升长期的离线转化率，它不要求你当时就产生购买动作，但是希望当你需要该产品的时候能想到这个品牌。效果广告又大致分为：展示广告和搜索广告。展示广告是指广告系统找到该网民与 Context 上下文（网页等）满足相应投放设置的广告；搜索广告是指广告系统找到与用户输入的 Query（及网民）相关且满足投放设置的广告（一条广告包含拍卖词 Bidword，出价，创意等信息，如图 6.5

所示）。

图 6.5

广告本质上就是达到广告主（买广告的企业/人），用户和广告公司的共赢，而广告公司要做的事就是给广告主找到最合适（后面解释什么是最合适）的用户，也可以说是想办法让广告（广告的一系列特征）与用户（用户特征）和展示广告的网页（页面特征）最符合，这样，展示广告和搜索广告就有区别了。展示广告就是展示与页面和用户最符合的广告，比如 google adsense，百度网盟等；而搜索广告不同，它是展示与用户输入的 query（也会包括用户特征）最符合的广告，比如 google adwords，百度凤巢等。

首先简单的说下展示广告，目前展示广告效果最好的就是 RTB(Real-Time Bidding，实时竞价)广告，它相比传统的广告最大的优点是卖“人”而不再是卖广告位，也就是说同一个广告位，不同人（兴趣不同）会看到不同的广告，所以更加精准。RTB 广告的实现需要不同的参与方合作才能完成：1）Ad Exchange(广告交易平台)，是整个服务的核心，它相当于交易所的功能；2）DSP(Demand Side Platform，需求方平台)，是供广告主使用的平台，广告主可以在这个平台设置受众目标，

投放区域和标价等；3）SSP(Sell Side Platform，供应方平台)，是供供应方（站长等）使用的平台，供应方可以在 SSP 上提交他们的广告位；4）DMP(Date Management Platform，数据管理平台)，是面向各个平台的，主要是用于数据管理，数据分析等。

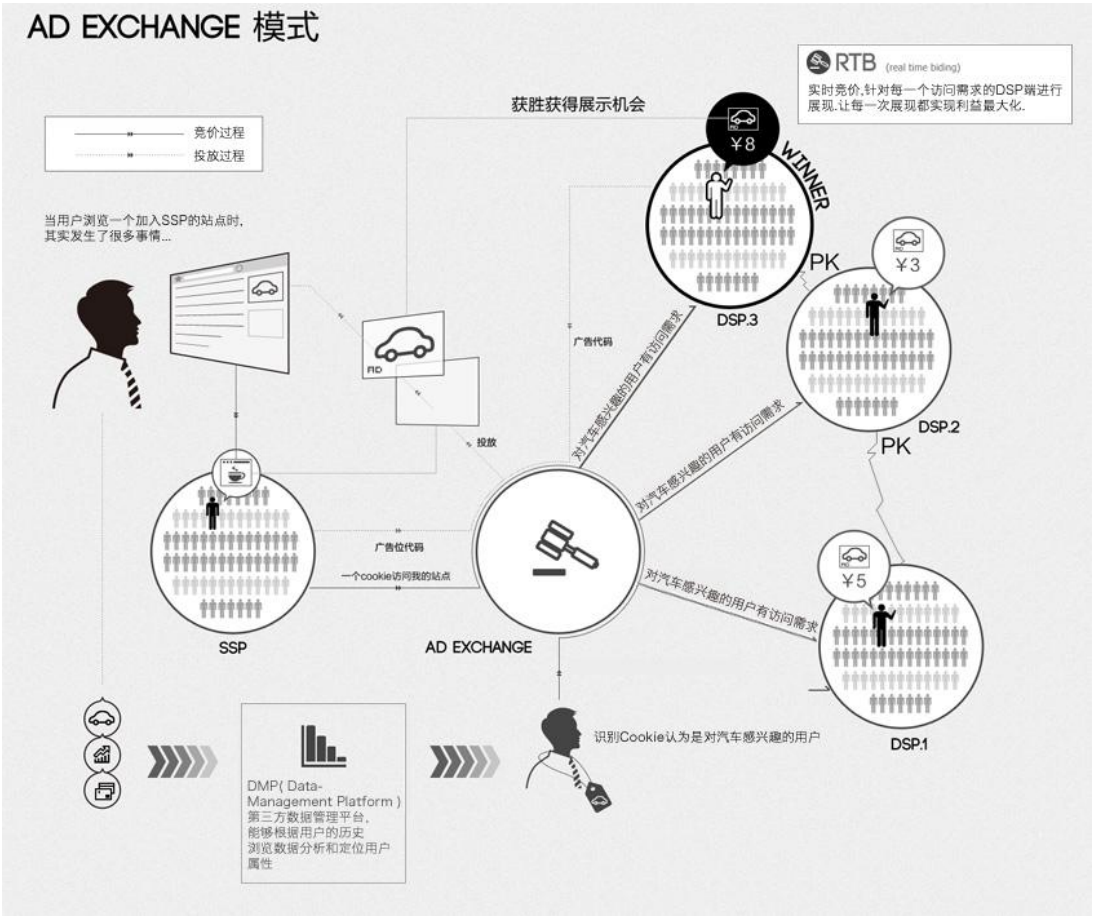


图 6.6

举个例子来说 RTB 广告的运作流程（大致同图 6.6），腾讯网有个广告位进入到了某个 SSP 平台，而这个 SSP 平台把这个广告位的每次展示都放到某个 AD Exchange 的交易平台中。现在有两个广告主，一个是卖体育用品的耐克公司，另一个是卖汽车的宝马公司。耐克选择使用了 DSP1 平台，设定的规则是：如果某用户是体育爱好者，那么帮我出价 1 块钱去竞拍这次的广告展示;宝马公司选择了 DSP2 平台，设定的规则是：如果某用户是汽车爱好者，那么帮我出 2 块钱去竞拍

这次的广告展示。这时，有一个用户刚要浏览腾讯网，于是 AD Exchange 告诉 DSP1 和 DSP2，并且把 Ad Exchange 记录的用户唯一标识 cookie 传给 DSP1 和 DSP2，DSP1 和 DSP2 根据这个 cookie，去 DMP 里找这个 cookie 的数据（找到就会有更多这个 cookie 的信息，找不到就使用自己存储的该 cookie 的信息），比如，这个时候 DSP1 通过 cookie 发现这个用户昨天搜索过“篮球”的关键词，DSP1 根据这个行为，把这个用户归为体育爱好者，于是按照广告主耐克公司的要求，DSP1 告诉 AD Exchange 平台，我这边有个耐克公司的客户，愿意为这次的广告展示出价 1 块钱；DSP2 通过 cookie 发现这个用户昨天还去浏览过某个汽车网，DSP2 根据这个行为把这个用户归为汽车爱好者，于是按照广告主宝马公司的要求，DSP2 告诉 AD Exchange 平台，我这边有个宝马公司的客户，愿意为这次的广告展示出价 2 块钱。在 AD Exchange 拿到 DSP1 和 DSP2 这两家的出价数据之后，比较发现 DSP2 出价最高，于是 AD Exchange 告诉 DSP2 说你竞拍成功，同时告诉 DSP1 说你的价格比较低，竞拍失败。在收到 AD Exchange 返回的数据之后，DSP2 就会把广告主宝马公司的广告创意给到 AD Exchange，AD Exchange 就会把宝马公司的广告在腾讯网上的这个广告位上展示了。

广告交易平台(Ad Exchange)模式，它相比传统的广告网络(Ad Network)模式最大优点个人认为其中之一就是 Ad Exchange 更偏重于对用户的精准投放，而 Ad Network 更偏重于对展示广告的页面和广告的匹配。

下面再以搜索广告来说下整个框架。

一个搜索广告系统还是很复杂的，我把它主要分为四大部分：业

务系统，检索系统，计费统计系统，反作弊系统。这几个模并不是单独存在的，都是要相互通信的。

业务系统主要是用来给广告主管理个人信息，管理广告，购买广告等操作，还要对用户的消费产生报表等；计费统计系统是用来对用户的点击进行计费，并修改广告主的余额等，还要记录一些其他点击信息等；反作弊系统就是要打击各种作弊行为，保护广告主的利益；检索系统就是要完成广告的整个检索过程，检索系统的整个架构如图 6.7 所示。

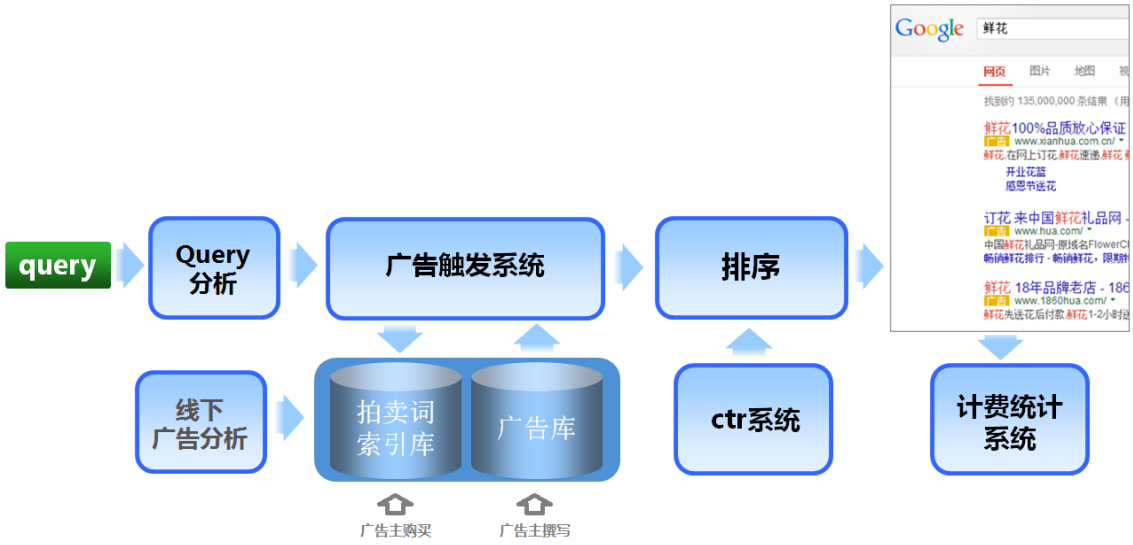


图 6.7

其实，搜索广告的检索系统和搜索引擎的系统很相似。

首先，用户输入 query 之后，先是进行 query 分析，和前面搜索引擎的 query 分析差不多，做些分词、query 变换、意图分析等等；

然后，把 query 分析结果发送给触发系统，触发系统算是比较核心的一个模块了，它的作用就是根据 query 找出最相关的一些广告，首先它去拍卖词（bidword）索引库中匹配出相关的拍卖词，匹配模式有三种形式：精确匹配，短语匹配，广泛匹配。精确匹配，就是当广告商

购买的关键词与网民 query 一模一样时，就可以（注意是可以而非必须）展现该广告；短语匹配的一个原则：保证 query 的语义是关键词语义的一个子集。广泛匹配，在短语匹配的基础上进一步放松限制，在分词、倒序、同义替换、上下位替换的基础上增加了近义替换、相关词等操作，这三种模式都是广告主买拍卖词时可以选择的。当匹配出拍卖词之后，还要计算 query-bidword 的相关性来做初步筛选，之后再根据拍卖词去广告库中找到相应的广告，而且还要进行一些过滤操作，比如广告主选择自己的广告必须出现在北京，那么上海的用户搜索就应该过滤掉这个广告。

排序模块就是对召回的所有广告排序，那么广告的排序规则是什么呢？就是 Q*Bid，Bid 就是拍卖词的出价，Q 是广告质量，通常都是使用 pctr 来表示（一般会有个参数来调整 Q 和 Bid 的权重），pctr 就是预估的点击率，也就是用户越可能点击且出价高的越要排在前面，这也就是前面说的最合适的广告。还记得前面讲的逻辑回归吗？ctr 预估就是使用逻辑回归来计算的，一般会用到三个维度的特征：query 特征、用户特征（cookie，session 等）和广告特征（广告的 title，创意，landing page 等），每一个维度的特征都是很大的，所以一般来说，ctr 预估的特征都是百亿甚至千亿级别的。

最后，就是对排好序的 top-N 广告展示（质量差，不相关的广告就不会展示）了，当用户点击之后，就会触发计费统计系统，也就是说如果是正常的点击（反作弊系统），那么就要扣掉广告主的钱，扣钱准则就是使用的广义二阶价格（GSP），即：price(i)=bid(i+1)*Q(i+1)/Q(i) + 0.01（i+1 表示 i 的下一位广告），有的会考虑用户体验进去。至于为什

么要这么计算，就涉及到一些博弈学的东西了。

这样，整个广告的检索系统就介绍完了，它和搜索引擎很相似，但也有很多不同，比如排序算法，再比如检索，网页量大，而广告的拍卖词库量少，所以广告的检索对 query 分析的要求就要更高（也需要线下对广告词扩展，比如利用点击过该广告的历史 query 来提高召回）。

6.5 推荐系统

推荐系统本可以单独成一章的，但是它和搜索系统或多或少有些关联，姑且先就放在这一节来讲吧。

什么是推荐系统，推荐系统一定是给某个“用户”推荐了某些“物品”（在社交网络中“物品”也可以是“人”）。所以“用户”和“物品”就是推荐系统中的两个关键，根据对这两个点的不同定位就会产生不同的算法：所有“用户”相同对待来处理（这样就等于不考虑用户的因素）？还是按不同属性大致划分之后来处理（将用户分类/聚类来处理）？或是每个“用户”都不同对待来处理（完全的个性化）？对于“物品”也是有类似这样的区分：不考虑物品的具体内容；对物品分类/聚类；完全区分物品。

推荐系统并没有搜索系统那么一个清晰的架构流程，大致来说，推荐系统使用最多的有如下两类方法：

基于协同过滤的推荐算法

基于内容的推荐算法

还有些基于知识的推荐，就是根据用户的一些约束条件等来匹配

出待推荐的物品。我们重点看下上述两类方法。

基于协同过滤的推荐算法（Collaborative Filtering，CF）

协同过滤是个典型的利用集体智慧的方法，它的思想非常容易理解：如果某个用户和你的兴趣相似，那么他喜欢的物品很可能就是你喜欢的。也可以说协同过滤第一步需要收集用户兴趣，然后根据用户兴趣计算相似用户或者物品，之后就可以进行推荐了。所以就产生了两类协同过滤方法：基于用户的协同过滤（User-based CF）和基于物品的协同过滤（Item-based CF）。

用户和物品的关系可以表示成一个矩阵（表），如下表所示。每一行代表一个用户对相应的物品的打分（1-5 分，分越高越好），现在我们要做的就是根据协同过滤计算出小明对 Item5 的打分，如果打分高，那么就可以把 Item5 推荐给小明了。

	Item1	Item2	Item3	Item4	Item5
小明	5	3	4	4	?
User1	3	1	2	3	3
User2	4	3	4	3	5
User3	3	3	1	5	4
User4	1	5	5	2	1

基于用户的协同过滤就是首先根据用户记录（上表），找到和目标用户相似的 N 个用户，然后根据这 N 个用户对目标物品的打分计算出目标用户对目标物品的预测打分，然后决定要不要推荐。首先就是要计算用户之间的相似度，用户间的相似度使用皮尔森相关系数效果好一点（回顾前面的相似度计算），这样，就可以计算出小明和其他四个用户的相似度了（读者可以带入公式计算一下），发现，小明与 User1 和 User2 最相似。接下来就要根据 User1 和 User2 对 Item5 的打分来预

测小明对 Item5 的打分，使用如下公式：

$$\mathrm{p(u,i)} = \overline{r_u} + \frac{\sum_{v \in N} sim(u,v) \times (r_{v,i} - \overline{r_v})}{\sum_{v \in N} sim(u,v)}$$

其中，$\mathrm{p(u,i)}$表示用户 u 对物品 i 的打分的预测值，$sim(u,v)$表示用户 u 和用户 v 的皮尔森相关系数，$r_{v,i}$表示用户 v 对物品 i 的打分，$\overline{r_u}$表示用户 u 的平均打分。

这样我们就可以计算出小明对所有其他物品的打分预测值，然后把高的 top-n 推荐给小明就可以了。这就是基于用户的协同过滤算法。

基于物品的协同过滤的思想是利用其他用户来计算物品相似度进而计算预测值。还是以上面的小明为例，我们首先计算出和 Item5 最相似的物品，物品间的相似度使用余弦相似度（更多的是改进的余弦相似度，它考虑了平均值）效果好一点，确定了物品相似度之后，就可以通过计算小明对所有 Item5 相似物品的加权评分综合来预测小明对 Item5 的预测值了，例如，如下公式（符号意思同之前一样）：

$$\mathrm{p(u,i)} = \frac{\sum_{j \in rel(u)} sim(j,i) \times r_{u,j}}{\sum_{j \in rel(u)} sim(j,i)}$$

这两个方法各有什么优缺点吗？User-based CF 需要实时计算用户间的相似度，所以计算量大且频繁，扩展性也不强；而 Item-based CF 是计算物品的相似度可以在线下计算，减少了线上的计算量，所以大型电子商务（例如，amazon）一般使用这种方法。当网站用户量和物品数量很多时，计算量还是很大的，有人为了降低计算量就提出了更简单的 Slope One 预测器《Slope One Predictors for Online Rating-Based Collaborative Filtering》。

协同过滤方法还有两个比较大的问题：1）数据稀疏问题。用户一

般只会评价少部分物品，所以矩阵就会非常稀疏，稀疏矩阵对最终的精度有很大影响；2）冷启动问题。对于还未做过任何评分的用户或者从未被评分过的物品就没办法直接使用协同过滤方法了。

协同过滤算法还有一种分法叫基于模型的协同过滤算法（有的把Item-based CF 归入这一类），例如：使用奇异值分解（SVD）对矩阵处理，这种方法同时也起到了降维的作用，大家可以参考这篇文章《Application of Dimensionality Reduction in Recommender System -- A Case Study》；还有就是把预测某个用户对某个物品的评分看成分类问题（比如，前面的例子就是分成 1-5 等），这样就可以使用贝叶斯等分类器来计算出该用户对某个物品的打分是属于哪个类别。另外的一种方法就是关联规则，通过用户记录学习出一组关联规则集合，然后就可以通过这些规则预测用户对物品的打分了。

协同过滤方法都是在矩阵上根据其他用户对某个用户的操作，而且它并不考虑物品的具体内容是什么，而物品的内容也是很有信息量的，就是下面要说的基于内容的推荐算法。

基于内容的推荐算法

该算法相当于淡化了“用户”，更多的从内容（物品特征属性的描述）的相似度来决定是否推荐，自然需要求内容的相似度了（呵呵，看到相似度有多重要了吧），从文本的角度来说，大多数都可以转化为前面讲的那些相似度方法。这个就不说了，参考前面的相似度计算章节。

从大的方面来说，基于内容的推荐算法大致有这么几种方法：

1、分类/聚类方法

把推荐问题可以看成是一个分类问题，根据用户对已有内容的打分训练一个分类模型（贝叶斯，SVM，决策树等等）；而聚类问题，就是看和某物品聚到一起的其他问题的打分情况来决定对这个物品要不要推荐。

2、搜索方法

搜索方法个人觉得这才是最能体现**内容**的一类方法，它就是通过搜索要推荐的物品找出和它最相似的物品，然后推荐给用户。它和搜索引擎有什么区别，最大的区别就是搜索引擎提交的是 query，而在这儿提交的是物品，根据不同的场景（视频推荐，新闻推荐，APP 推荐等）会有不同的搜索条件和排序策略。这时，我们可以想一想搜索引擎是不也可以看作一个推荐系统，它是根据 query 来推荐网页（所以很多技术从宏观的角度看都是相通的）？一个区别是搜索引擎的结果一定要精确（因为用户明确要的就是与 query 最相关的内容），而推荐系统结果却比较发散一点，只要不是太差都可以接受（这也决定了搜索引擎比推荐系统要难做的原因之一，搜索引擎就那么几家，推荐系统却满天飞，越精确的要求就越高）。大家有兴趣可以参考一下 google 最新的一篇文章《Up Next: Retrieval Methods for Large Scale Related Video Suggestion》就是使用类似的方法。

以上就是推荐系统的大致的算法，但是真正实用的时候就要针对不用的应用场合（视频推荐，新闻推荐，APP 推荐等）设计不同的算法和策略（有的是多种算法的结合），《App Recommendation: A Contest between Satisfaction and Temptation》这篇文章作者就提出了一种 APP 推荐模型。比如我手机上已经装了一款天气预报的 APP 那么再给我推

荐另一款天气预报的 APP，我下载的愿望就非常低（想到国内最大的某电子商务网站，经常就是你买什么推荐什么，咦…），但是对于电影这种问题就小一点，我正在看《谍影重重》，你给我推荐《谍影重重 2》和《谍影重重 3》，我觉得还行。总之，推荐系统更多的是和场合相关，很难把一个场合的推荐算法完全照搬到另一个场合上使用。推荐系统的目的就是能产生某种行为（点击、购买等），所以推荐系统的本质其实就是预测能力，因此，一个好的推荐系统就是能最大化达到目的的预测器。

结尾：多学习，多思考！《文本上的算法》目前先写这么多了，参考了很多很多论文和书籍，就不一一罗列了，以后有什么心得的话，再补充上去或者把某些章节进行深入探讨，希望这些东西对大家的工作或者学习有所帮助和启发。