# 第六章 海量数据处理

# 6.0本章导读

所谓海量数据处理，是指基于海量数据的存储、处理、和操作。正因为数据量太大，所以导致要么无法在较短时间内迅速解决，要么无法一次性装入内存。

事实上，针对时间问题，可以采用巧妙的算法搭配合适的数据结构（如布隆过滤器、哈希、位图、堆、数据库、倒排索引、Trie树）来解决；而对于空间问题，可以采取分而治之（哈希映射）的方法，也就是说，把规模大的数据转化为规模小的，从而各个击破。

此外，针对常说的单机及集群问题，通俗来讲，单机就是指处理装载数据的机器有限（只要考虑CPU、内存、和硬盘之间的数据交互），而集群的意思是指机器有多台，适合分布式处理或并行计算，更多考虑节点与节点之间的数据交互。

一般说来，处理海量数据问题，有以下十种典型方法：

* 1.哈希分治；
* 2.simhash算法；
* 3.外排序；
* 4.MapReduce；
* 5.多层划分；
* 6.位图；
* 7.布隆过滤器；
* 8.Trie树；
* 9.数据库；
* 10.倒排索引。

受理论之限，本章将摒弃绝大部分的细节，只谈方法和模式论，注重用最通俗、最直白的语言阐述相关问题。最后，有一点必须强调的是，全章行文是基于面试题的分析基础之上的，具体实践过程中，还得视具体情况具体分析，且各个场景下需要考虑的细节也远比本章所描述的任何一种解决方案复杂得多。

# 6.1关联式容器

一般来说，STL容器分为：

* 序列式容器(vector/list/deque/stack/queue/heap)，和关联式容器。
  + 其中，关联式容器又分为set(集合)和map(映射表)两大类，以及这两大类的衍生体multiset(多键集合)和multimap(多键映射表)，这些容器均以RB-tree（red-black tree, 红黑树）完成。
  + 此外，还有第3类关联式容器，如hashtable(散列表)，以及以hashtable为底层机制完成的hash\_set(散列集合)/hash\_map(散列映射表)/hash\_multiset(散列多键集合)/hash\_multimap(散列多键映射表)。也就是说，set/map/multiset/multimap都内含一个RB-tree，而hash\_set/hash\_map/hash\_multiset/hash\_multimap都内含一个hashtable。

所谓关联式容器，类似关联式数据库，每笔数据或每个元素都有一个键值(key)和一个实值(value)，即所谓的Key-Value(键-值对)。当元素被插入到关联式容器中时，容器内部结构(RB-tree/hashtable)便依照其键值大小，以某种特定规则将这个元素放置于适当位置。

包括在非关联式数据库中，比如，在MongoDB内，文档(document)是最基本的数据组织形式，每个文档也是以Key-Value（键-值对）的方式组织起来。一个文档可以有多个Key-Value组合，每个Value可以是不同的类型，比如String、Integer、List等等。

{ "name" : "July",

"sex" : "male",

"age" : 23 }

**set/map/multiset/multimap**

set，同map一样，所有元素都会根据元素的键值自动被排序，因为set/map两者的所有各种操作，都只是转而调用RB-tree的操作行为，不过，值得注意的是，两者都不允许两个元素有相同的键值。

不同的是：set的元素不像map那样可以同时拥有实值(value)和键值(key)，set元素的键值就是实值，实值就是键值，而map的所有元素都是pair，同时拥有实值(value)和键值(key)，pair的第一个元素被视为键值，第二个元素被视为实值。

至于multiset/multimap，他们的特性及用法和set/map完全相同，唯一的差别就在于它们允许键值重复，即所有的插入操作基于RB-tree的insert\_equal()而非insert\_unique()。

**hash\_set/hash\_map/hash\_multiset/hash\_multimap**

hash\_set/hash\_map，两者的一切操作都是基于hashtable之上。不同的是，hash\_set同set一样，同时拥有实值和键值，且实质就是键值，键值就是实值，而hash\_map同map一样，每一个元素同时拥有一个实值(value)和一个键值(key)，所以其使用方式，和上面的map基本相同。但由于hash\_set/hash\_map都是基于hashtable之上，所以不具备自动排序功能。为什么？因为hashtable没有自动排序功能。

至于hash\_multiset/hash\_multimap的特性与上面的multiset/multimap完全相同，唯一的差别就是它们hash\_multiset/hash\_multimap的底层实现机制是hashtable（而multiset/multimap，上面说了，底层实现机制是RB-tree），所以它们的元素都不会被自动排序，不过也都允许键值重复。

所以，综上，说白了，什么样的结构决定其什么样的性质，因为set/map/multiset/multimap都是基于RB-tree之上，所以有自动排序功能，而hash\_set/hash\_map/hash\_multiset/hash\_multimap都是基于hashtable之上，所以不含有自动排序功能，至于加个前缀multi\_无非就是允许键值重复而已。

# 6.2分而治之

### 方法介绍

对于海量数据而言，由于无法一次性装进内存处理，导致我们不得不把海量的数据通过hash映射分割成相应的小块数据，然后再针对各个小块数据通过hash\_map进行统计或其它操作。

那什么是hash映射呢？简单来说，就是为了便于计算机在有限的内存中处理big数据，我们通过一种映射散列的方式让数据均匀分布在对应的内存位置(如大数据通过取余的方式映射成小数存放在内存中，或大文件映射成多个小文件)，而这个映射散列方式便是我们通常所说的hash函数，设计的好的hash函数能让数据均匀分布而减少冲突。

### 问题实例

**1、海量日志数据，提取出某日访问百度次数最多的那个IP**

**分析**：百度作为国内第一大搜索引擎，每天访问它的IP数量巨大，如果想一次性把所有IP数据装进内存处理，则内存容量明显不够，故针对数据太大，内存受限的情况，可以把大文件转化成（取模映射）小文件，从而大而化小，逐个处理。

换言之，先映射，而后统计，最后排序。

**解法**：具体分为以下3个步骤

* 1.分而治之/hash映射
  + 首先把这一天访问百度日志的所有IP提取出来，然后逐个写入到一个大文件中，接着采用映射的方法，比如%1000，把整个大文件映射为1000个小文件。
* 2.hash\_map统计
  + 当大文件转化成了小文件，那么我们便可以采用hash\_map(ip, value)来分别对1000个小文件中的IP进行频率统计，再找出每个小文件中出现频率最大的IP。
* 3.堆/快速排序
  + 统计出1000个频率最大的IP后，依据各自频率的大小进行排序(可采取堆排序)，找出那个频率最大的IP，即为所求。

**注**：Hash取模是一种等价映射，不会存在同一个元素分散到不同小文件中去的情况，即这里采用的是%1000算法，那么同一个IP在hash后，只可能落在同一个文件中，不可能被分散的。

**2、寻找热门查询，300万个查询字符串中统计最热门的10个查询**

**原题**：搜索引擎会通过日志文件把用户每次检索使用的所有检索串都记录下来，每个查询串的长度为1-255字节。假设目前有一千万个记录，请你统计最热门的10个查询串，要求使用的内存不能超过1G。

**分析**：这些查询串的重复度比较高，虽然总数是1千万，但如果除去重复后，不超过3百万个。一个查询串的重复度越高，说明查询它的用户越多，也就是越热门。

由上面第1题，我们知道，数据大则划为小的，例如一亿个ip求Top 10，可先%1000将ip分到1000个小文件中去，并保证一种ip只出现在一个文件中，再对每个小文件中的ip进行hash\_map统计并按数量排序，最后归并或者最小堆依次处理每个小文件的top10以得到最后的结果。

但对于本题，数据规模比较小，能一次性装入内存。因为根据题目描述，虽然有一千万个Query，但是由于重复度比较高，故去除重复后，事实上只有300万的Query，每个Query255Byte，因此我们可以考虑把他们都放进内存中去（300万个字符串假设没有重复，都是最大长度，那么最多占用内存3M\*1K/4=0.75G。所以可以将所有字符串都存放在内存中进行处理）。

所以我们放弃分而治之/hash映射的步骤，直接上hash\_map统计，然后排序。So，针对此类典型的TOP K问题，采取的对策往往是：hash\_map + 堆。

**解法**：

* 1.hash\_map统计
  + 先对这批海量数据预处理。具体方法是：维护一个Key为Query字串，Value为该Query出现次数的hash\_map，即hash\_map(Query, Value)，每次读取一个Query，如果该字串不在Table中，那么加入该字串，并将Value值设为1；如果该字串在Table中，那么将该字串的计数加1 即可。最终我们在O(N)的时间复杂度内用hash\_map完成了统计；
* 2.堆排序
  + 借助堆这个数据结构，找出Top K，时间复杂度为N‘logK。即借助堆结构，我们可以在log量级的时间内查找和调整/移动。因此，维护一个K(该题目中是10)大小的小根堆，然后遍历300万的Query，分别和根元素进行对比。所以，我们最终的时间复杂度是：O(n) + N' \* O(logk），其中，N为1000万，N’为300万。

关于第2步堆排序，可以维护k个元素的最小堆，即用容量为k的最小堆存储最先遍历到的k个数，并假设它们即是最大的k个数，建堆费时O（k），并调整堆(费时O(logk))后，有k1>k2>...kmin（kmin设为小顶堆中最小元素）。继续遍历数列，每次遍历一个元素x，与堆顶元素比较，若x>kmin，则更新堆（x入堆，用时logk），否则不更新堆。这样下来，总费时O（klogk+（n-k）logk）=O（n\*logk）。此方法得益于在堆中，查找等各项操作时间复杂度均为logk。

当然，你也可以采用trie树，关键字域存该查询串出现的次数，没有出现为0。最后用10个元素的最小推来对出现频率进行排序。

**3、有一个1G大小的一个文件，里面每一行是一个词，词的大小不超过16字节，内存限制大小是1M。返回频数最高的100个词**

**解法**：

* 1.分而治之/hash映射
  + 顺序读取文件，对于每个词x，取hash(x)%5000，然后把该值存到5000个小文件（记为x0,x1,...x4999）中。这样每个文件大概是200k左右。当然，如果其中有的小文件超过了1M大小，还可以按照类似的方法继续往下分，直到分解得到的小文件的大小都不超过1M。
* 2.hash\_map统计
  + 对每个小文件，采用trie树/hash\_map等统计每个文件中出现的词以及相应的频率。
* 3.堆/归并排序
  + 取出出现频率最大的100个词（可以用含100个结点的最小堆）后，再把100个词及相应的频率存入文件，这样又得到了5000个文件。最后就是把这5000个文件进行归并（类似于归并排序）的过程了。

**4、海量数据分布在100台电脑中，想个办法高效统计出这批数据的TOP10**

**解法一**：

如果同一个数据元素只出现在某一台机器中，那么可以采取以下步骤统计出现次数TOP10的数据元素：

* 1.堆排序
  + 在每台电脑上求出TOP 10，可以采用包含10个元素的堆完成（TOP 10小，用最大堆，TOP 10大，用最小堆，比如求TOP10大，我们首先取前10个元素调整成最小堆，如果发现，然后扫描后面的数据，并与堆顶元素比较，如果比堆顶元素大，那么用该元素替换堆顶，然后再调整为最小堆。最后堆中的元素就是TOP 10大）。
* 2.组合归并
  + 求出每台电脑上的TOP 10后，然后把这100台电脑上的TOP 10组合起来，共1000个数据，再利用上面类似的方法求出TOP 10就可以了。

**解法二**：

但如果同一个元素重复出现在不同的电脑中呢，比如拿两台机器求top 2的情况来说：

* 第一台的数据分布及各自出现频率为：a(50)，b(50)，c(49)，d(49) ，e(0)，f(0)
  + 其中，括号里的数字代表某个数据出现的频率，如a(50)表示a出现了50次。
* 第二台的数据分布及各自出现频率为：a(0)，b(0)，c(49)，d(49)，e(50)，f(50)

这个时候，你可以有两种方法：

* 遍历一遍所有数据，重新hash取摸，如此使得同一个元素只出现在单独的一台电脑中，然后采用上面所说的方法，统计每台电脑中各个元素的出现次数找出TOP 10，继而组合100台电脑上的TOP 10，找出最终的TOP 10。
* 或者，暴力求解：直接统计统计每台电脑中各个元素的出现次数，然后把同一个元素在不同机器中的出现次数相加，最终从所有数据中找出TOP 10。

**5、有10个文件，每个文件1G，每个文件的每一行存放的都是用户的query，每个文件的query都可能重复。要求你按照query的频度排序**

**解法一**：

* 1.hash映射
  + 顺序读取10个文件，按照hash(query)%10的结果将query写入到另外10个文件（记为a0,a1,..a9）中。这样新生成的文件每个的大小大约也1G（假设hash函数是随机的）。
* 2.hash\_map统计
  + 找一台内存在2G左右的机器，依次对用hash\_map(query, query\_count)来统计每个query出现的次数。注：hash\_map(query, query\_count)是用来统计每个query的出现次数，不是存储他们的值，出现一次，则count+1。
* 3.堆/快速/归并排序
  + 利用快速/堆/归并排序按照出现次数进行排序，将排序好的query和对应的query\_cout输出到文件中，这样得到了10个排好序的文件（记为[https://github.com/julycoding/The-Art-Of-Programming-By-July/raw/master/ebook/images/8/8.1/8.1.2.gif](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/8/8.1/8.1.2.gif)）。最后，对这10个文件进行归并排序（内排序与外排序相结合）。

**解法二**：

一般query的总量是有限的，只是重复的次数比较多而已，可能对于所有的query，一次性就可以加入到内存了。这样，我们就可以采用trie树/hash\_map等直接来统计每个query出现的次数，然后按出现次数做快速/堆/归并排序就可以了。

**解法三**：

与解法1类似，但在做完hash，分成多个文件后，可以交给多个文件来处理，采用分布式的架构来处理（比如MapReduce），最后再进行合并。

**6、给定a、b两个文件，各存放50亿个url，每个url各占64字节，内存限制是4G，让你找出a、b文件共同的url？**

**解法**：

可以估计每个文件安的大小为5G×64=320G，远远大于内存限制的4G。所以不可能将其完全加载到内存中处理。考虑采取分而治之的方法。

* 1.分而治之/hash映射
  + 遍历文件a，对每个url求取[https://github.com/julycoding/The-Art-Of-Programming-By-July/raw/master/ebook/images/8/8.1/8.1.3.gif](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/8/8.1/8.1.3.gif)，然后根据所取得的值将url分别存储到1000个小文件（记为[https://github.com/julycoding/The-Art-Of-Programming-By-July/raw/master/ebook/images/8/8.1/8.1.4.gif](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/8/8.1/8.1.4.gif)，这里漏写个了a1）中。这样每个小文件的大约为300M。遍历文件b，采取和a相同的方式将url分别存储到1000小文件中（记为[https://github.com/julycoding/The-Art-Of-Programming-By-July/raw/master/ebook/images/8/8.1/8.1.5.gif](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/8/8.1/8.1.5.gif)）。这样处理后，所有可能相同的url都在对应的小文件（[https://github.com/julycoding/The-Art-Of-Programming-By-July/raw/master/ebook/images/8/8.1/8.1.6.gif](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/8/8.1/8.1.6.gif)）中，不对应的小文件不可能有相同的url。然后我们只要求出1000对小文件中相同的url即可。
* 2.hash\_set统计
  + 求每对小文件中相同的url时，可以把其中一个小文件的url存储到hash\_set中。然后遍历另一个小文件的每个url，看其是否在刚才构建的hash\_set中，如果是，那么就是共同的url，存到文件里面就可以了。

**7、100万个数中找出最大的100个数**

**解法一**：采用局部淘汰法。选取前100个元素，并排序，记为序列L。然后一次扫描剩余的元素x，与排好序的100个元素中最小的元素比，如果比这个最小的要大，那么把这个最小的元素删除，并把x利用插入排序的思想，插入到序列L中。依次循环，知道扫描了所有的元素。复杂度为O(100万\*100)。

**解法二**：采用快速排序的思想，每次分割之后只考虑比轴大的一部分，知道比轴大的一部分在比100多的时候，采用传统排序算法排序，取前100个。复杂度为O(100万\*100)。

**解法三**：在前面的题中，我们已经提到了，用一个含100个元素的最小堆完成。复杂度为O(100万\*lg100)。

### 举一反三

**1**、怎么在海量数据中找出重复次数最多的一个？

提示：先做hash，然后求模映射为小文件，求出每个小文件中重复次数最多的一个，并记录重复次数。然后找出上一步求出的数据中重复次数最多的一个就是所求（具体参考前面的题）。

**2**、上千万或上亿数据（有重复），统计其中出现次数最多的前N个数据。

提示：上千万或上亿的数据，现在的机器的内存应该能存下。所以考虑采用hash\_map/搜索二叉树/红黑树等来进行统计次数。然后就是取出前N个出现次数最多的数据了，可以用第2题提到的堆机制完成。

**3**、一个文本文件，大约有一万行，每行一个词，要求统计出其中最频繁出现的前10个词，请给出思想，给出时间复杂度分析。

提示：这题是考虑时间效率。用trie树统计每个词出现的次数，时间复杂度是O(n\*le)（le表示单词的平准长度）。然后是找出出现最频繁的前10个词，可以用堆来实现，前面的题中已经讲到了，时间复杂度是O(n\*lg10)。所以总的时间复杂度，是O(n\*le)与O(n\*lg10)中较大的哪一个。

**4**、1000万字符串，其中有些是重复的，需要把重复的全部去掉，保留没有重复的字符串。请怎么设计和实现？

提示：这题用trie树比较合适，hash\_map也行。当然，也可以先hash成小文件分开处理再综合。

**5**、一个文本文件，找出前10个经常出现的词，但这次文件比较长，说是上亿行或十亿行，总之无法一次读入内存，问最优解。

提示：首先根据用hash并求模，将文件分解为多个小文件，对于单个文件利用上题的方法求出每个文件件中10个最常出现的词。然后再进行归并处理，找出最终的10个最常出现的词。

# 6.3 simhash算法

### 方法介绍

### 背景

如果某一天，面试官问你如何设计一个比较两篇文章相似度的算法？可能你会回答几个比较传统点的思路：

* 一种方案是先将两篇文章分别进行分词，得到一系列特征向量，然后计算特征向量之间的距离（可以计算它们之间的欧氏距离、海明距离或者夹角余弦等等），从而通过距离的大小来判断两篇文章的相似度。
* 另外一种方案是传统hash，我们考虑为每一个web文档通过hash的方式生成一个指纹（finger print）。

下面，我们来分析下这两种方法。

* 采取第一种方法，若是只比较两篇文章的相似性还好，但如果是海量数据呢，有着数以百万甚至亿万的网页，要求你计算这些网页的相似度。你还会去计算任意两个网页之间的距离或夹角余弦么？想必你不会了。
* 而第二种方案中所说的传统加密方式md5，其设计的目的是为了让整个分布尽可能地均匀，但如果输入内容一旦出现哪怕轻微的变化，hash值就会发生很大的变化。

举个例子，我们假设有以下三段文本：

* the cat sat on the mat
* the cat sat on a mat
* we all scream for ice cream

使用传统hash可能会得到如下的结果：

* irb(main):006:0> p1 = 'the cat sat on the mat'
  + irb(main):007:0> p1.hash => 415542861
* irb(main):005:0> p2 = 'the cat sat on a mat'
  + irb(main):007:0> p2.hash => 668720516
* irb(main):007:0> p3 = 'we all scream for ice cream'
  + irb(main):007:0> p3.hash => 767429688 "

可理想当中的hash函数，需要对几乎相同的输入内容，产生相同或者相近的hash值，换言之，hash值的相似程度要能直接反映输入内容的相似程度，故md5等传统hash方法也无法满足我们的需求。

### 出世

车到山前必有路，来自于GoogleMoses Charikar发表的一篇论文“detecting near-duplicates for web crawling”中提出了simhash算法，专门用来解决亿万级别的网页的去重任务。

simhash作为locality sensitive hash（局部敏感哈希）的一种：

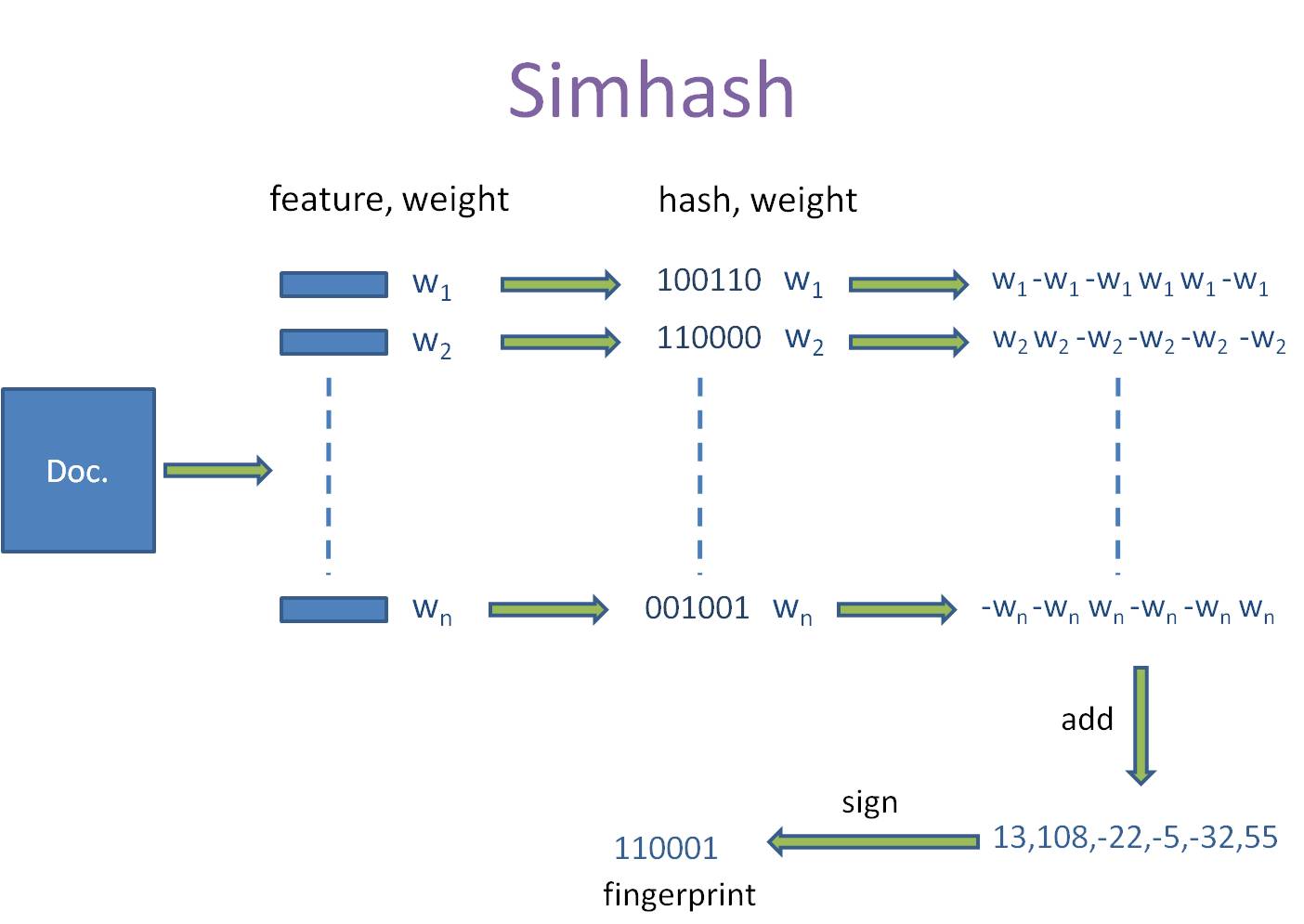
* 其主要思想是降维，将高维的特征向量映射成低维的特征向量，通过两个向量的Hamming Distance来确定文章是否重复或者高度近似。
  + 其中，Hamming Distance，又称汉明距离，在信息论中，两个等长字符串之间的汉明距离是两个字符串对应位置的不同字符的个数。也就是说，它就是将一个字符串变换成另外一个字符串所需要替换的字符个数。例如：1011101 与 1001001 之间的汉明距离是 2。至于我们常说的字符串编辑距离则是一般形式的汉明距离。

如此，通过比较多个文档的simHash值的海明距离，可以获取它们的相似度。

### 流程

simhash算法分为5个步骤：分词、hash、加权、合并、降维，具体过程如下所述：

* 分词
  + 给定一段语句，进行分词，得到有效的特征向量，然后为每一个特征向量设置1-5等5个级别的权重（如果是给定一个文本，那么特征向量可以是文本中的词，其权重可以是这个词出现的次数）。例如给定一段语句：“CSDN博客结构之法算法之道的作者July”，分词后为：“CSDN 博客 结构 之 法 算法 之 道 的 作者 July”，然后为每个特征向量赋予权值：CSDN(4) 博客(5) 结构(3) 之(1) 法(2) 算法(3) 之(1) 道(2) 的(1) 作者(5) July(5)，其中括号里的数字代表这个单词在整条语句中的重要程度，数字越大代表越重要。
* hash
  + 通过hash函数计算各个特征向量的hash值，hash值为二进制数01组成的n-bit签名。比如“CSDN”的hash值Hash(CSDN)为100101，“博客”的hash值Hash(博客)为“101011”。就这样，字符串就变成了一系列数字。
* 加权
  + 在hash值的基础上，给所有特征向量进行加权，即W = Hash \* weight，且遇到1则hash值和权值正相乘，遇到0则hash值和权值负相乘。例如给“CSDN”的hash值“100101”加权得到：W(CSDN) = 1001014 = 4 -4 -4 4 -4 4，给“博客”的hash值“101011”加权得到：W(博客)=1010115 = 5 -5 5 -5 5 5，其余特征向量类似此般操作。
* 合并
  + 将上述各个特征向量的加权结果累加，变成只有一个序列串。拿前两个特征向量举例，例如“CSDN”的“4 -4 -4 4 -4 4”和“博客”的“5 -5 5 -5 5 5”进行累加，得到“4+5 -4+-5 -4+5 4+-5 -4+5 4+5”，得到“9 -9 1 -1 1”。
* 降维
  + 对于n-bit签名的累加结果，如果大于0则置1，否则置0，从而得到该语句的simhash值，最后我们便可以根据不同语句simhash的海明距离来判断它们的相似度。例如把上面计算出来的“9 -9 1 -1 1 9”降维（某位大于0记为1，小于0记为0），得到的01串为：“1 0 1 0 1 1”，从而形成它们的simhash签名。

其流程如下图所示：[](https://camo.githubusercontent.com/efd5caf946abb03b9fe83ccb1ab85208d32d1c60/687474703a2f2f646c2e69746579652e636f6d2f75706c6f61642f6174746163686d656e742f3433373432362f62616634323337382d653632352d333564322d396138392d3437313532346133353564382e6a7067)

### 应用

* 每篇文档得到SimHash签名值后，接着计算两个签名的海明距离即可。根据经验值，对64位的 SimHash值，海明距离在3以内的可认为相似度比较高。
  + 海明距离的求法：异或时，只有在两个比较的位不同时其结果是1 ，否则结果为0，两个二进制“异或”后得到1的个数即为海明距离的大小。

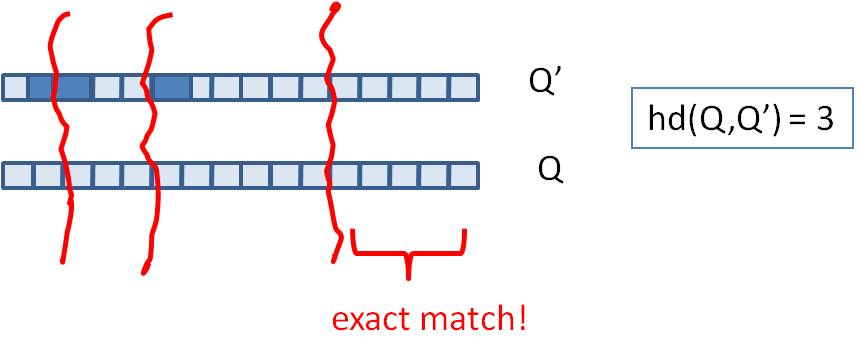
举个例子，上面我们计算到的“CSDN博客”的simhash签名值为“1 0 1 0 1 1”，假定我们计算出另外一个短语的签名值为“1 0 1 0 0 0”，那么根据异或规则，我们可以计算出这两个签名的海明距离为2，从而判定这两个短语的相似度是比较高的。

换言之，现在问题转换为：对于64位的SimHash值，我们只要找到海明距离在3以内的所有签名，即可找出所有相似的短语。

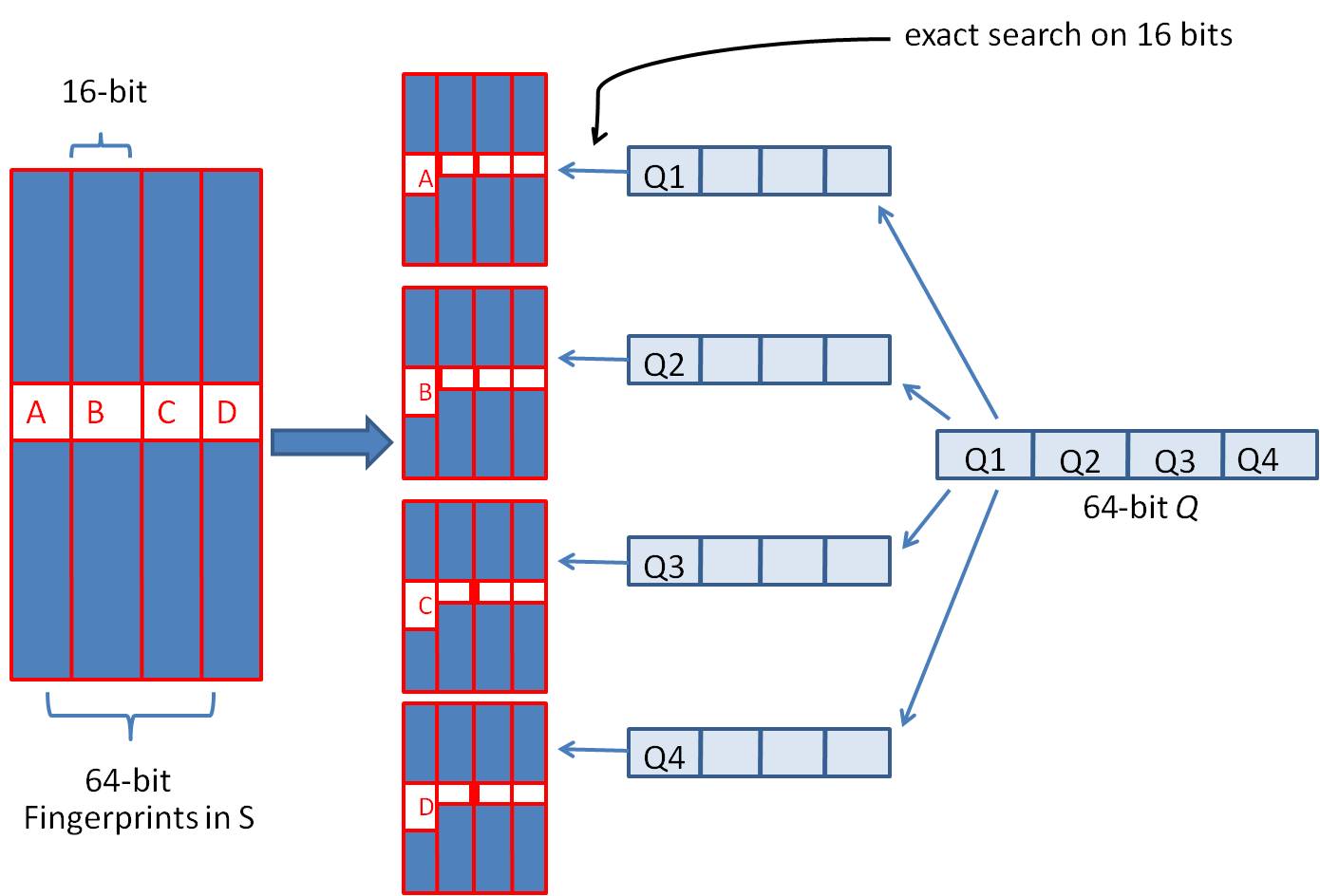
但关键是，如何将其扩展到海量数据呢？譬如如何在海量的样本库中查询与其海明距离在3以内的记录呢？

* 一种方案是查找待查询文本的64位simhash code的所有3位以内变化的组合
  + 大约需要四万多次的查询。
* 另一种方案是预生成库中所有样本simhash code的3位变化以内的组合
  + 大约需要占据4万多倍的原始空间。

这两种方案，要么时间复杂度高，要么空间复杂度复杂，能否有一种方案可以达到时空复杂度的绝佳平衡呢？答案是肯定的：

* 我们可以把 64 位的二进制simhash签名均分成4块，每块16位。根据鸽巢原理（也称抽屉原理），如果两个签名的海明距离在 3 以内，它们必有一块完全相同。如下图所示：[](https://camo.githubusercontent.com/d6cc444c5a3db896a0b3aae6333cb6dcc42598cc/687474703a2f2f646c2e69746579652e636f6d2f75706c6f61642f6174746163686d656e742f3433373535392f36383937313964662d353462372d333138632d626339302d6532383966383433343462392e6a7067)
* 然后把分成的4 块中的每一个块分别作为前16位来进行查找，建倒排索引。

具体如下图所示：

[](https://camo.githubusercontent.com/52160c9c5b3160034d76d1ccd0e9e7f514b46c80/687474703a2f2f646c2e69746579652e636f6d2f75706c6f61642f6174746163686d656e742f3433373538362f62373262386463322d393133392d333037382d616432342d6236383966363466643731612e6a7067)

如此，如果样本库中存有2^34（差不多10亿）的simhash签名，则每个table返回2^(34-16)=262144个候选结果，大大减少了海明距离的计算成本。

* 假设数据是均匀分布，16位的数据，产生的像限为2^16个，则平均每个像限分布的文档数则为2^34/2^16 = 2^(34-16)) ，四个块返回的总结果数为 4\* 262144 （大概 100 万）。
  + 这样，原本需要比较10亿次，经过索引后，大概只需要处理100万次。

（部分内容及图片参考自：<http://grunt1223.iteye.com/blog/964564> ，后续图片会全部重画）

### 问题实例

待续。

@复旦李斌：simhash不是google发明的，是princeton的人早在stoc02上发表的。google在www07上的那篇论文只是在网页查重上应用了下。事实上www07中的算法是stoc02中随机超平面的一个极其巧妙的实现，bit差异的期望正好等于原姶向量的余弦。

# 6.4外排序

## 方法介绍

所谓外排序，顾名思义，即是在内存外面的排序，因为当要处理的数据量很大，而不能一次装入内存时，此时只能放在读写较慢的外存储器（通常是硬盘）上。

外排序通常采用的是一种“排序-归并”的策略。

* 在排序阶段，先读入能放在内存中的数据量，将其排序输出到一个临时文件，依此进行，将待排序数据组织为多个有序的临时文件；
* 尔后在归并阶段将这些临时文件组合为一个大的有序文件，也即排序结果。

假定现在有20个数据的文件A：{5 11 0 18 4 14 9 7 6 8 12 17 16 13 19 10 2 1 3 15}，但一次只能使用仅装4个数据的内容，所以，我们可以每趟对4个数据进行排序，即5路归并，具体方法如下述步骤：

* 我们先把“大”文件A，分割为a1，a2，a3，a4，a5等5个小文件，每个小文件4个数据
  + a1文件为：5 11 0 18
  + a2文件为：4 14 9 7
  + a3文件为：6 8 12 17
  + a4文件为：16 13 19 10
  + a5文件为：2 1 3 15
* 然后依次对5个小文件分别进行排序
  + a1文件完成排序后：0 5 11 18
  + a2文件完成排序后：4 7 9 14
  + a3文件完成排序后：6 8 12 17
  + a4文件完成排序后：10 13 16 19
  + a5文件完成排序后：1 2 3 15
* 最终多路归并，完成整个排序
  + 整个大文件A文件完成排序后：0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19

## 问题实例

**1、给10^7个数据量的磁盘文件排序**

输入：给定一个文件，里面最多含有n个不重复的正整数（也就是说可能含有少于n个不重复正整数），且其中每个数都小于等于n，n=10^7。 输出：得到按从小到大升序排列的包含所有输入的整数的列表。 条件：最多有大约1MB的内存空间可用，但磁盘空间足够。且要求运行时间在5分钟以下，10秒为最佳结果。

**解法一**：位图方案

你可能会想到把磁盘文件进行归并排序，但题目要求你只有1MB的内存空间可用，所以，归并排序这个方法不行。

熟悉位图的朋友可能会想到用位图来表示这个文件集合。例如正如编程珠玑一书上所述，用一个20位长的字符串来表示一个所有元素都小于20的简单的非负整数集合，边框用如下字符串来表示集合{1,2,3,5,8,13}：

0 1 1 1 0 1 0 0 1 0 0 0 0 1 0 0 0 0 0 0

上述集合中各数对应的位置则置1，没有对应的数的位置则置0。

参考编程珠玑一书上的位图方案，针对我们的10^7个数据量的磁盘文件排序问题，我们可以这么考虑，由于每个7位十进制整数表示一个小于1000万的整数。我们可以使用一个具有1000万个位的字符串来表示这个文件，其中，当且仅当整数i在文件中存在时，第i位为1。采取这个位图的方案是因为我们面对的这个问题的特殊性：

1. 输入数据限制在相对较小的范围内，
2. 数据没有重复，
3. 其中的每条记录都是单一的整数，没有任何其它与之关联的数据。

所以，此问题用位图的方案分为以下三步进行解决：

* 第一步，将所有的位都置为0，从而将集合初始化为空。
* 第二步，通过读入文件中的每个整数来建立集合，将每个对应的位都置为1。
* 第三步，检验每一位，如果该位为1，就输出对应的整数。

经过以上三步后，产生有序的输出文件。令n为位图向量中的位数（本例中为1000 0000），程序可以用伪代码表示如下：

//磁盘文件排序位图方案的伪代码

//copyright@ Jon Bentley

//July、updated，2011.05.29。

//第一步，将所有的位都初始化为0

for i ={0,....n}

bit[i]=0;

//第二步，通过读入文件中的每个整数来建立集合，将每个对应的位都置为1。

for each i in the input file

bit[i]=1;

//第三步，检验每一位，如果该位为1，就输出对应的整数。

for i={0...n}

if bit[i]==1

write i on the output file

上述的位图方案，共需要扫描输入数据两次，具体执行步骤如下：

第一次，只处理1—4999999之间的数据，这些数都是小于5000000的，对这些数进行位图排序，只需要约5000000/8=625000Byte，也就是0.625M，排序后输出。 第二次，扫描输入文件时，只处理4999999-10000000的数据项，也只需要0.625M（可以使用第一次处理申请的内存）。 因此，总共也只需要0.625M位图的的方法有必要强调一下，就是位图的适用范围为针对不重复的数据进行排序，若数据有重复，位图方案就不适用了。

不过很快，我们就将意识到，用此位图方法，严格说来还是不太行，空间消耗10^7/8还是大于1M（1M=1024\*1024空间，小于10^7/8）。

既然如果用位图方案的话，我们需要约1.25MB（若每条记录是8位的正整数的话，则10000000/(102410248) ~= 1.2M）的空间，而现在只有1MB的可用存储空间，那么究竟该作何处理呢？

**解法二**：多路归并

诚然，在面对本题时，还可以通过计算分析出可以用如2的位图法解决，但实际上，很多的时候，我们都面临着这样一个问题，文件太大，无法一次性放入内存中计算处理，那这个时候咋办呢？分而治之，大而化小，也就是把整个大文件分为若干大小的几块，然后分别对每一块进行排序，最后完成整个过程的排序。k趟算法可以在kn的时间开销内和n/k的空间开销内完成对最多n个小于n的无重复正整数的排序。

比如可分为2块（k=2，1趟反正占用的内存只有1.25/2M），1~4999999，和5000000~9999999。先遍历一趟，首先排序处理1~4999999之间的整数（用5000000/8=625000个字的存储空间来排序0~4999999之间的整数），然后再第二趟，对5000001~1000000之间的整数进行排序处理。

**解法总结**

1、关于本章中位图和多路归并两种方案的时间复杂度及空间复杂度的比较，如下：

时间复杂度 空间复杂度

位图 O(N) 0.625M

多位归并 O(Nlogn) 1M

（多路归并，时间复杂度为O（kn/klogn/k ），严格来说，还要加上读写磁盘的时间，而此算法绝大部分时间也是浪费在这上面）

2、bit-map

适用范围：可进行数据的快速查找，判重，删除，一般来说数据范围是int的10倍以下

基本原理及要点：使用bit数组来表示某些元素是否存在，比如8位电话号码

扩展：bloom filter可以看做是对bit-map的扩展

## 举一反三

* 1. 已知某个文件内包含一些电话号码，每个号码为8位数字，统计不同号码的个数。8位最多99 999 999，大概需要99m个bit，大概10几m字节的内存即可。

# 6.5分布式处理之MapReduce

## 方法介绍

MapReduce是一种计算模型，简单的说就是将大批量的工作（数据）分解（MAP）执行，然后再将结果合并成最终结果（REDUCE）。这样做的好处是可以在任务被分解后，可以通过大量机器进行并行计算，减少整个操作的时间。但如果你要我再通俗点介绍，那么，说白了，Mapreduce的原理就是一个归并排序。

适用范围：数据量大，但是数据种类小可以放入内存

基本原理及要点：将数据交给不同的机器去处理，数据划分，结果归约。

### 基础架构

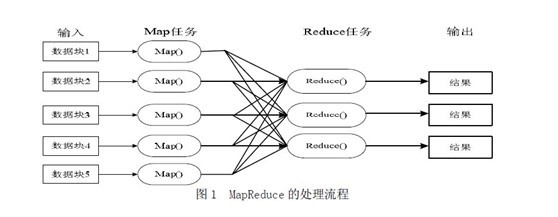
想读懂此文，读者必须先要明确以下几点，以作为阅读后续内容的基础知识储备：

1. MapReduce是一种模式。
2. Hadoop是一种框架。
3. Hadoop是一个实现了MapReduce模式的开源的分布式并行编程框架。

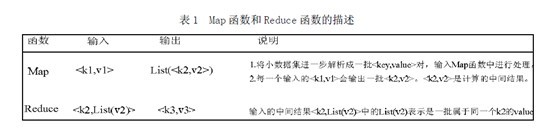
所以，你现在，知道了什么是MapReduce，什么是hadoop，以及这两者之间最简单的联系，而本文的主旨即是，一句话概括：**在hadoop的框架上采取MapReduce的模式处理海量数据**。下面，咱们可以依次深入学习和了解MapReduce和hadoop这两个东西了。

### MapReduce模式

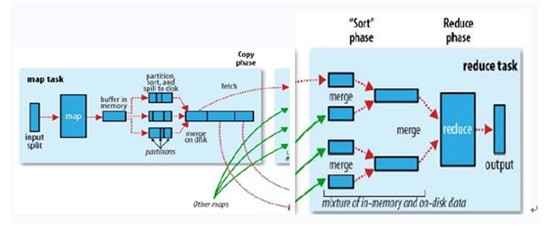
前面说了，MapReduce是一种模式，一种什么模式呢?一种云计算的核心计算模式，一种分布式运算技术，也是简化的分布式编程模式，它主要用于解决问题的程序开发模型，也是开发人员拆解问题的方法。

Ok，光说不上图，没用。如下图所示，MapReduce模式的主要思想是将自动分割要执行的问题（例如程序）拆解成Map（映射）和Reduce（化简）的方式，流程图如下图1所示：  
[](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/8/8.2/8.2.1.gif)

在数据被分割后通过Map函数的程序将数据映射成不同的区块，分配给计算机机群处理达到分布式运算的效果，在通过Reduce 函数的程序将结果汇整，从而输出开发者需要的结果。

MapReduce借鉴了函数式程序设计语言的设计思想，其软件实现是指定一个Map函数，把键值对(key/value)映射成新的键值对(key/value)，形成一系列中间结果形式的key/value 对，然后把它们传给Reduce(规约)函数，把具有相同中间形式key的value合并在一起。Map和Reduce函数具有一定的关联性。函数描述如表1 所示：  
[](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/8/8.2/8.2.2.gif)

MapReduce致力于解决大规模数据处理的问题，因此在设计之初就考虑了数据的局部性原理，利用局部性原理将整个问题分而治之。MapReduce集群由普通PC机构成，为无共享式架构。在处理之前，将数据集分布至各个节点。处理时，每个节点就近读取本地存储的数据处理（map），将处理后的数据进行合并（combine）、排序（shuffle and sort）后再分发（至reduce节点），避免了大量数据的传输，提高了处理效率。无共享式架构的另一个好处是配合复制（replication）策略，集群可以具有良好的容错性，一部分节点的down机对集群的正常工作不会造成影响。

ok，你可以再简单看看下副图，整幅图是有关hadoop的作业调优参数及原理，图的左边是MapTask运行示意图，右边是ReduceTask运行示意图：  
[](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/8/8.2/8.2.3.gif)

如上图所示，其中map阶段，当map task开始运算，并产生中间数据后并非直接而简单的写入磁盘，它首先利用内存buffer来对已经产生的buffer进行缓存，并在内存buffer中进行一些预排序来优化整个map的性能。而上图右边的reduce阶段则经历了三个阶段，分别Copy->Sort->reduce。我们能明显的看出，其中的Sort是采用的归并排序，即merge sort。

## 问题实例

1. The canonical example application of MapReduce is a process to count the appearances of each different word in a set of documents:
2. 海量数据分布在100台电脑中，想个办法高效统计出这批数据的TOP10。
3. 一共有N个机器，每个机器上有N个数。每个机器最多存O(N)个数并对它们操作。如何找到N^2个数的中数(median)？

# 6.6 多层划分

## 方法介绍

多层划分法，本质上还是分而治之的思想，因为元素范围很大，不能利用直接寻址表，所以通过多次划分，逐步确定范围，然后最后在一个可以接受的范围内进行。

## 问题实例

**1、2.5亿个整数中找出不重复的整数的个数，内存空间不足以容纳这2.5亿个整数**

分析：有点像鸽巢原理，整数个数为2^32,也就是，我们可以将这2^32个数，划分为2^8个区域(比如用单个文件代表一个区域)，然后将数据分离到不同的区域，然后不同的区域在利用bitmap就可以直接解决了。也就是说只要有足够的磁盘空间，就可以很方便的解决。

**2、5亿个int找它们的中位数**

分析：首先我们将int划分为2^16个区域，然后读取数据统计落到各个区域里的数的个数，之后我们根据统计结果就可以判断中位数落到那个区域，同时知道这个区域中的第几大数刚好是中位数。然后第二次扫描我们只统计落在这个区域中的那些数就可以了。

实际上，如果不是int是int64，我们可以经过3次这样的划分即可降低到可以接受的程度。即可以先将int64分成2^24个区域，然后确定区域的第几大数，在将该区域分成2^20个子区域，然后确定是子区域的第几大数，然后子区域里的数的个数只有2^20，就可以直接利用direct addr table进行统计了。

# 6.7 Bitmap

## 方法介绍

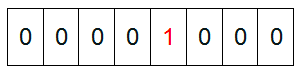
### 什么是Bit-map

所谓的Bit-map就是用一个bit位来标记某个元素对应的Value， 而Key即是该元素。由于采用了Bit为单位来存储数据，因此在存储空间方面，可以大大节省。

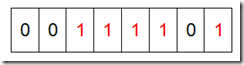
来看一个具体的例子，假设我们要对0-7内的5个元素(4,7,2,5,3)排序（这里假设这些元素没有重复）。那么我们就可以采用Bit-map的方法来达到排序的目的。要表示8个数，我们就只需要8个Bit（1Bytes），首先我们开辟1Byte的空间，将这些空间的所有Bit位都置为0(如下图：)

[https://github.com/julycoding/The-Art-Of-Programming-By-July/raw/master/ebook/images/9/9.2/9.2.1.gif](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/9/9.2/9.2.1.gif)

然后遍历这5个元素，首先第一个元素是4，那么就把4对应的位置为1（可以这样操作 p+(i/8)|(0×01<<(i%8)) 当然了这里的操作涉及到Big-ending和Little-ending的情况，这里默认为Big-ending）,因为是从零开始的，所以要把第五位置为一（如下图）：

[](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/9/9.2/9.2.2.gif)

然后再处理第二个元素7，将第八位置为1,，接着再处理第三个元素，一直到最后处理完所有的元素，将相应的位置为1，这时候的内存的Bit位的状态如下：

[](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/9/9.2/9.2.3.gif)

然后我们现在遍历一遍Bit区域，将该位是一的位的编号输出（2，3，4，5，7），这样就达到了排序的目的。下面的代码给出了一个BitMap的用法：排序。

//定义每个Byte中有8个Bit位

#include ＜memory.h＞

#define BYTESIZE 8

void SetBit(char \*p, int posi)

{

for(int i=0; i < (posi/BYTESIZE); i++)

{

p++;

}

\*p = \*p|(0x01<<(posi%BYTESIZE));//将该Bit位赋值1

return;

}

void BitMapSortDemo()

{

//为了简单起见，我们不考虑负数

int num[] = {3,5,2,10,6,12,8,14,9};

//BufferLen这个值是根据待排序的数据中最大值确定的

//待排序中的最大值是14，因此只需要2个Bytes(16个Bit)

//就可以了。

const int BufferLen = 2;

char \*pBuffer = new char[BufferLen];

//要将所有的Bit位置为0，否则结果不可预知。

memset(pBuffer,0,BufferLen);

for(int i=0;i<9;i++)

{

//首先将相应Bit位上置为1

SetBit(pBuffer,num[i]);

}

//输出排序结果

for(int i=0;i<BufferLen;i++)//每次处理一个字节(Byte)

{

for(int j=0;j<BYTESIZE;j++)//处理该字节中的每个Bit位

{

//判断该位上是否是1，进行输出，这里的判断比较笨。

//首先得到该第j位的掩码（0x01＜＜j），将内存区中的

//位和此掩码作与操作。最后判断掩码是否和处理后的

//结果相同

if((\*pBuffer&(0x01<<j)) == (0x01<<j))

{

printf("%d ",i\*BYTESIZE + j);

}

}

pBuffer++;

}

}

int \_tmain(int argc, \_TCHAR\* argv[])

{

BitMapSortDemo();

return 0;

}

可进行数据的快速查找，判重，删除，一般来说数据范围是int的10倍以下

### 基本原理及要点

使用bit数组来表示某些元素是否存在，比如8位电话号码.

## 问题实例

**1、在2.5亿个整数中找出不重复的整数，注，内存不足以容纳这2.5亿个整数**

**解法一**：采用2-Bitmap（每个数分配2bit，00表示不存在，01表示出现一次，10表示多次，11无意义）进行，共需内存2^32 \* 2 bit=1 GB内存，还可以接受。然后扫描这2.5亿个整数，查看Bitmap中相对应位，如果是00变01，01变10，10保持不变。所描完事后，查看bitmap，把对应位是01的整数输出即可。

**解法二**：也可采用与第1题类似的方法，进行划分小文件的方法。然后在小文件中找出不重复的整数，并排序。然后再进行归并，注意去除重复的元素。”

**2、给40亿个不重复的unsigned int的整数，没排过序的，然后再给一个数，如何快速判断这个数是否在那40亿个数当中？**

**解法一**：可以用位图/Bitmap的方法，申请512M的内存，一个bit位代表一个unsigned int值。读入40亿个数，设置相应的bit位，读入要查询的数，查看相应bit位是否为1，为1表示存在，为0表示不存在。

# 6.8 Bloom Filter

## 方法介绍

### 一、什么是Bloom Filter

Bloom Filter，被译作称布隆过滤器，是一种空间效率很高的随机数据结构，Bloom filter可以看做是对bit-map的扩展,它的原理是：

* 当一个元素被加入集合时，通过K个Hash函数将这个元素映射成一个位阵列（Bit array）中的K个点，把它们置为1\*\*。检索时，我们只要看看这些点是不是都是1就（大约）知道集合中有没有它了：
  + 如果这些点有任何一个0，则被检索元素一定不在；
  + 如果都是1，则被检索元素很可能在。

其可以用来实现数据字典，进行数据的判重，或者集合求交集。

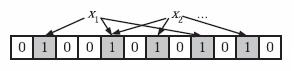
但Bloom Filter的这种高效是有一定代价的：在判断一个元素是否属于某个集合时，有可能会把不属于这个集合的元素误认为属于这个集合（false positive）。因此，Bloom Filter不适合那些“零错误”的应用场合。而在能容忍低错误率的应用场合下，Bloom Filter通过极少的错误换取了存储空间的极大节省。

#### 1.1、集合表示和元素查询

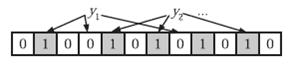
下面我们具体来看Bloom Filter是如何用位数组表示集合的。初始状态时，Bloom Filter是一个包含m位的位数组，每一位都置为0。

[https://github.com/julycoding/The-Art-Of-Programming-By-July/raw/master/ebook/images/9/9.3/9.3.1.jpg](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/9/9.3/9.3.1.jpg)

为了表达S={x1, x2,…,xn}这样一个n个元素的集合，Bloom Filter使用k个相互独立的哈希函数（Hash Function），它们分别将集合中的每个元素映射到{1,…,m}的范围中。对任意一个元素x，第i个哈希函数映射的位置hi(x)就会被置为1（1≤i≤k）。注意，如果一个位置多次被置为1，那么只有第一次会起作用，后面几次将没有任何效果。在下图中，k=3，且有两个哈希函数选中同一个位置（从左边数第五位，即第二个“1“处）。

[](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/9/9.3/9.3.2.jpg)

在判断y是否属于这个集合时，我们对y应用k次哈希函数，如果所有hi(y)的位置都是1（1≤i≤k），那么我们就认为y是集合中的元素，否则就认为y不是集合中的元素。下图中y1就不是集合中的元素（因为y1有一处指向了“0”位）。y2或者属于这个集合，或者刚好是一个false positive。

[](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/9/9.3/9.3.3.jpg)

#### 1.2、错误率估计

前面我们已经提到了，Bloom Filter在判断一个元素是否属于它表示的集合时会有一定的错误率（false positive rate），下面我们就来估计错误率的大小。在估计之前为了简化模型，我们假设kn1, x2,…,xn}的所有元素都被k个哈希函数映射到m位的位数组中时，这个位数组中某一位还是0的概率是：

其中1/m表示任意一个哈希函数选中这一位的概率（前提是哈希函数是完全随机的），(1-1/m)表示哈希一次没有选中这一位的概率。要把S完全映射到位数组中，需要做kn次哈希。某一位还是0意味着kn次哈希都没有选中它，因此这个概率就是（1-1/m）的kn次方。令p = e-kn/m是为了简化运算，这里用到了计算e时常用的近似：

令ρ为位数组中0的比例，则ρ的数学期望E(ρ)= p’。在ρ已知的情况下，要求的错误率（false positive rate）为：

(1-ρ)为位数组中1的比例，(1-ρ)k就表示k次哈希都刚好选中1的区域，即false positive rate。上式中第二步近似在前面已经提到了，现在来看第一步近似。p’只是ρ的数学期望，在实际中ρ的值有可能偏离它的数学期望值。M. Mitzenmacher已经证明[2] ，位数组中0的比例非常集中地分布在它的数学期望值的附近。因此，第一步的近似得以成立。分别将p和p’代入上式中，得：

相比p’和f’，使用p和f通常在分析中更为方便。

#### 1.3、最优的哈希函数个数

既然Bloom Filter要靠多个哈希函数将集合映射到位数组中，那么应该选择几个哈希函数才能使元素查询时的错误率降到最低呢？这里有两个互斥的理由：如果哈希函数的个数多，那么在对一个不属于集合的元素进行查询时得到0的概率就大；但另一方面，如果哈希函数的个数少，那么位数组中的0就多。为了得到最优的哈希函数个数，我们需要根据上一小节中的错误率公式进行计算。

先用p和f进行计算。注意到f = exp(k ln(1 − e−kn/m))，我们令g = k ln(1 − e−kn/m)，只要让g取到最小，f自然也取到最小。由于p = e-kn/m，我们可以将g写成

[img](https://camo.githubusercontent.com/18745855685838b90b4765ca2f3d76ee0cec6f05/687474703a2f2f63686172742e617069732e676f6f676c652e636f6d2f63686172743f6368743d74782663686c3d673d2d253543667261632537426d2537442537426e2537442535436c6e2870292535436c6e28312d7029)

根据对称性法则可以很容易看出当p = 1/2，也就是k = ln2· (m/n)时，g取得最小值。在这种情况下，最小错误率f等于(1/2)k≈ (0.6185)m/n。另外，注意到p是位数组中某一位仍是0的概率，所以p = 1/2对应着位数组中0和1各一半。换句话说，要想保持错误率低，最好让位数组有一半还空着。

需要强调的一点是，p = 1/2时错误率最小这个结果并不依赖于近似值p和f。同样对于f’ = exp(k ln(1 − (1 − 1/m)kn))，g’ = k ln(1 − (1 − 1/m)kn)，p’ = (1 − 1/m)kn，我们可以将g’写成

同样根据对称性法则可以得到当p’ = 1/2时，g’取得最小值。

#### 1.4、位数组的大小

下面我们来看看，在不超过一定错误率的情况下，Bloom Filter至少需要多少位才能表示全集中任意n个元素的集合。假设全集中共有u个元素，允许的最大错误率为є，下面我们来求位数组的位数m。

假设X为全集中任取n个元素的集合，F(X)是表示X的位数组。那么对于集合X中任意一个元素x，在s = F(X)中查询x都能得到肯定的结果，即s能够接受x。显然，由于Bloom Filter引入了错误，s能够接受的不仅仅是X中的元素，它还能够є (u - n)个false positive。因此，对于一个确定的位数组来说，它能够接受总共n + є (u - n)个元素。在n + є (u - n)个元素中，s真正表示的只有其中n个，所以一个确定的位数组可以表示

[img](https://camo.githubusercontent.com/ec8494457b4173b98b9db1d22a6bb7219a5caf87/687474703a2f2f63686172742e617069732e676f6f676c652e636f6d2f63686172743f6368743d74782663686c3d435f2537426e253242253543657073696c6f6e28752d6e292537442535456e)

个集合。m位的位数组共有2m个不同的组合，进而可以推出，m位的位数组可以表示

[img](https://camo.githubusercontent.com/b07e4a1b453ba68feb7be42bce7c7f3692791682/687474703a2f2f63686172742e617069732e676f6f676c652e636f6d2f63686172743f6368743d74782663686c3d322535456d435f2537426e253242253543657073696c6f6e28752d6e292537442535456e)

个集合。全集中n个元素的集合总共有

[img](https://camo.githubusercontent.com/6ae48ae48f73fdfc1dce6c5b333473fcc184264e/687474703a2f2f63686172742e617069732e676f6f676c652e636f6d2f63686172743f6368743d74782663686c3d435f253742752537442535456e)

个，因此要让m位的位数组能够表示所有n个元素的集合，必须有

即：

上式中的近似前提是n和єu相比很小，这也是实际情况中常常发生的。根据上式，我们得出结论：在错误率不大于є的情况下，m至少要等于n log2(1/є)才能表示任意n个元素的集合。

上一小节中我们曾算出当k = ln2· (m/n)时错误率f最小，这时f = (1/2)k= (1/2)mln2 / n。现在令f≤є，可以推出

这个结果比前面我们算得的下界n log2(1/є)大了log2e≈ 1.44倍。这说明在哈希函数的个数取到最优时，要让错误率不超过є，m至少需要取到最小值的1.44倍。

## 问题实例

**1、给你A,B两个文件，各存放50亿条URL，每条URL占用64字节，内存限制是4G，让你找出A,B文件共同的URL。如果是三个乃至n个文件呢？**

**分析**：如果允许有一定的错误率，可以使用Bloom filter，4G内存大概可以表示340亿bit。将其中一个文件中的url使用Bloom filter映射为这340亿bit，然后挨个读取另外一个文件的url，检查是否与Bloom filter，如果是，那么该url应该是共同的url（注意会有一定的错误率）。”

# 6.9 Trie树（字典树）

## 方法介绍

### 1.1、什么是Trie树

Trie树，即字典树，又称单词查找树或键树，是一种树形结构。典型应用是用于统计和排序大量的字符串（但不仅限于字符串），所以经常被搜索引擎系统用于文本词频统计。它的优点是最大限度地减少无谓的字符串比较，查询效率比较高。

Trie的核心思想是空间换时间，利用字符串的公共前缀来降低查询时间的开销以达到提高效率的目的。

它有3个基本性质：

1. 根节点不包含字符，除根节点外每一个节点都只包含一个字符。
2. 从根节点到某一节点，路径上经过的字符连接起来，为该节点对应的字符串。
3. 每个节点的所有子节点包含的字符都不相同。

### 1.2、树的构建

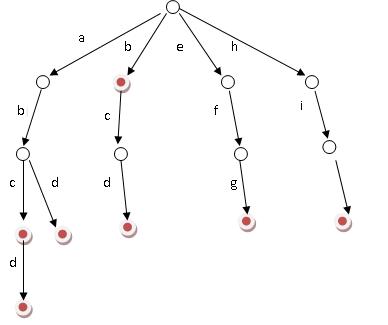
咱们先来看一个问题：假如现在给你10万个长度不超过10的单词，对于每一个单词，我们要判断它出没出现过，如果出现了，求第一次出现在第几个位置。对于这个问题，我们该怎么解决呢？

如果我们用最傻的方法，对于每一个单词，我们都要去查找它前面的单词中是否有它。那么这个算法的复杂度就是O(n^2)。显然对于10万的范围难以接受。

换个思路想：

* 假设我要查询的单词是abcd，那么在它前面的单词中，以b，c，d，f之类开头的显然不必考虑，而只要找以a开头的中是否存在abcd就可以了。
  + 同样的，在以a开头中的单词中，我们只要考虑以b作为第二个字母的，一次次缩小范围和提高针对性，这样一个树的模型就渐渐清晰了。

即如果现在有b，abc，abd，bcd，abcd，efg，hii 这6个单词，我们可以构建一棵如下图所示的树：

[](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/8/8.4/1.jpg)

如上图所示，对于每一个节点，从根遍历到他的过程就是一个单词，如果这个节点被标记为红色，就表示这个单词存在，否则不存在。

那么，对于一个单词，只要顺着他从根走到对应的节点，再看这个节点是否被标记为红色就可以知道它是否出现过了。把这个节点标记为红色，就相当于插入了这个单词。

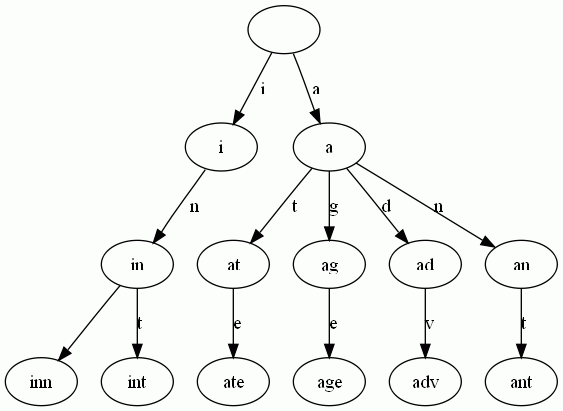
这样一来我们查询和插入可以一起完成，所用时间仅仅为单词长度（在这个例子中，便是10）。这就是一棵trie树。

我们可以看到，trie树每一层的节点数是26^i级别的。所以为了节省空间，我们还可以用动态链表，或者用数组来模拟动态。而空间的花费，不会超过单词数×单词长度。

### 1.3、查询

Trie树是简单但实用的数据结构，通常用于实现字典查询。我们做即时响应用户输入的AJAX搜索框时，就是Trie开始。本质上，Trie是一颗存储多个字符串的树。相邻节点间的边代表一个字符，这样树的每条分支代表一则子串，而树的叶节点则代表完整的字符串。和普通树不同的地方是，相同的字符串前缀共享同一条分支。

下面，再举一个例子。给出一组单词，inn, int, at, age, adv, ant, 我们可以得到下面的Trie：

[](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/8/8.4/2.gif)

可以看出：

* 每条边对应一个字母。
* 每个节点对应一项前缀。叶节点对应最长前缀，即单词本身。
* 单词inn与单词int有共同的前缀“in”, 因此他们共享左边的一条分支，root->i->in。同理，ate, age, adv, 和ant共享前缀"a"，所以他们共享从根节点到节点"a"的边。

查询操纵非常简单。比如要查找int，顺着路径i -> in -> int就找到了。

搭建Trie的基本算法也很简单，无非是逐一把每则单词的每个字母插入Trie。插入前先看前缀是否存在。如果存在，就共享，否则创建对应的节点和边。比如要插入单词add，就有下面几步：

1. 考察前缀"a"，发现边a已经存在。于是顺着边a走到节点a。
2. 考察剩下的字符串"dd"的前缀"d"，发现从节点a出发，已经有边d存在。于是顺着边d走到节点ad
3. 考察最后一个字符"d"，这下从节点ad出发没有边d了，于是创建节点ad的子节点add，并把边ad->add标记为d。

## 问题实例

**1、一个文本文件，大约有一万行，每行一个词，要求统计出其中最频繁出现的前10个词，请给出思想，给出时间复杂度分析**

**提示**：用trie树统计每个词出现的次数，时间复杂度是O(n\*le)（le表示单词的平均长度），然后是找出出现最频繁的前10个词。当然，也可以用堆来实现，时间复杂度是O(n\*lg10)。所以总的时间复杂度，是O(n\*le)与O(n\*lg10)中较大的哪一个。

**2、寻找热门查询**

**原题**：搜索引擎会通过日志文件把用户每次检索使用的所有检索串都记录下来，每个查询串的长度为1-255字节。假设目前有一千万个记录，这些查询串的重复读比较高，虽然总数是1千万，但是如果去除重复和，不超过3百万个。一个查询串的重复度越高，说明查询它的用户越多，也就越热门。请你统计最热门的10个查询串，要求使用的内存不能超过1G。

**提示**：利用trie树，关键字域存该查询串出现的次数，没有出现为0。最后用10个元素的最小推来对出现频率进行排序。

# 6.10数据库

## 方法介绍

当遇到大数据量的增删改查时，一般把数据装进数据库中，从而利用数据的设计实现方法，对海量数据的增删改查进行处理。

# 6.11倒排索引(Inverted index)

## 方法介绍

倒排索引是一种索引方法，被用来存储在全文搜索下某个单词在一个文档或者一组文档中的存储位置的映射，常被应用于搜索引擎和关键字查询的问题中。

以英文为例，下面是要被索引的文本：

T0 = "it is what it is"

T1 = "what is it"

T2 = "it is a banana"

我们就能得到下面的反向文件索引：

"a": {2}

"banana": {2}

"is": {0, 1, 2}

"it": {0, 1, 2}

"what": {0, 1}

检索的条件"what","is"和"it"将对应集合的交集。

正向索引开发出来用来存储每个文档的单词的列表。正向索引的查询往往满足每个文档有序频繁的全文查询和每个单词在校验文档中的验证这样的查询。在正向索引中，文档占据了中心的位置，每个文档指向了一个它所包含的索引项的序列。也就是说文档指向了它包含的那些单词，而反向索引则是单词指向了包含它的文档，很容易看到这个反向的关系。

## 问题实例

**1、文档检索系统，查询那些文件包含了某单词，比如常见的学术论文的关键字搜索**

**提示**：建倒排索引。

# 6.15本章海量数据的习题

**1**有100W个关键字，长度小于等于50字节。用高效的算法找出top10的热词，并对内存的占用不超过1MB。

提示：老题，与caopengcs讨论后，得出具体思路为：

* 先把100W个关键字hash映射到小文件，根据题意，100W*50B = 50*10^6B = 50M，而内存只有1M，故干脆搞一个hash函数 % 50，分解成50个小文件；
* 针对对每个小文件依次运用hashmap(key，value)完成每个key的value次数统计，后用堆找出每个小文件中value次数最大的top 10；-最后依次对每两小文件的top 10归并，得到最终的top 10。

此外，很多细节需要注意下，举个例子，如若hash映射后导致分布不均的话，有的小文件可能会超过1M，故为保险起见，你可能会说根据数据范围分解成50~500或更多的小文件，但到底是多少呢？我觉得这不重要，勿纠结答案，虽准备在平时，但关键还是看临场发挥，保持思路清晰关注细节即可。

**2**

单机5G内存，磁盘200T的数据，分别为字符串，然后给定一个字符串，判断这200T数据里面有没有这个字符串，怎么做？ 如果查询次数会非常的多, 怎么预处理？

提示：如果数据是200g且允许少许误差的话，可以考虑用布隆过滤器Bloom Filter。但本题是200T，得另寻良策，具体解法请读者继续思考。

**3**

现在有一个大文件，文件里面的每一行都有一个group标识（group很多，但是每个group的数据量很小），现在要求把这个大文件分成十个小文件，要求：

* 1、同一个group的必须在一个文件里面；
* 2、切分之后，要求十个小文件的数据量尽可能均衡。

**7**

服务器内存1G，有一个2G的文件，里面每行存着一个QQ号（5-10位数），怎么最快找出出现过最多次的QQ号。

**8**

尽量高效的统计一片英文文章（总单词数目）里出现的所有英文单词，按照在文章中首次出现的顺序打印输出该单词和它的出现次数。

**9**

在人人好友里，A和B是好友，B和C是好友，如果A　和C不是好友，那么C是A的二度好友，在一个有10万人的数据库里，如何在时间０（ｎ）里，找到某个人的十度好友。

**12**

海量记录，记录形式如下： TERMID URLNOCOUNT urlno1 urlno2 ..., urlnon，请问怎么考虑资源和时间这两个因素，实现快速查询任意两个记录的交集，并集等，设计相关的数据结构和算法。

**14**

有一亿个整数，请找出最大的1000个，要求时间越短越好，空间占用越少越好。

**18**

10亿个int型整数，如何找出重复出现的数字。

**19**

有2G的一个文本文档，文件每行存储的是一个句子，每个单词是用空格隔开的。问：输入一个句子，如何找到和它最相似的前10个句子。

提示：可用倒排文档。

**20**

某家视频网站，每天有上亿的视频被观看，现在公司要请研发人员找出最热门的视频。 该问题的输入可以简化为一个字符串文件，每一行都表示一个视频id，然后要找出出现次数最多的前100个视频id，将其输出，同时输出该视频的出现次数。

* 1.假设每天的视频播放次数为3亿次，被观看的视频数量为一百万个，每个视频ID的长度为20字节，限定使用的内存为1G。请简述做法，再写代码。
* 2.假设每个月的视频播放次数为100亿次，被观看的视频数量为1亿，每个视频ID的长度为20字节，一台机器被限定使用的内存为1G。

提示：万变不离其宗，分而治之/Hash映射 + Hash统计 + 堆/快速/归并排序。

**21**

有一个log文件，里面记录的格式为：

QQ号 时间 flag

123456 14：00：00 0

123457 14：00：01 1

其中flag=0表示登录 flag=1表示退出

问：统计一天平均在线的QQ数。

**22**

一个文本，一万行，每行一个词，统计出现频率最高的前10个词（词的平均长度为Len）。并分析时间复杂度。

**23**

在一个文件中有 10G 个整数，乱序排列，要求找出中位数。内存限制为 2G。只写出思路即可。

**24**

一个url指向的页面里面有另一个url,最终有一个url指向之前出现过的url或空，这两种情形都定义为null。这样构成一个单链表。给两条这样单链表，判断里面是否存在同样的url。url以亿级计，资源不足以hash。

**25**

一个1G大小的一个文件，里面每一行是一个词，词的大小不超过16字节，内存限制大小是1M。返回频数最高的100个词。

**26**

1000万字符串，其中有些是重复的，需要把重复的全部去掉，保留没有重复的字符串。请怎么设计和实现？

**27**有10个文件，每个文件1G，每个文件的每一行都存放的是用户的query，每个文件的query都可能重复。要你按照query的频度排序。

**28**

现有一200M的文本文件，里面记录着IP地址和对应地域信息，如

202.100.83.56 北京 北京大学

202.100.83.120 北京 人民大学

202.100.83.134 北京 中国青年政治学院

211.93.120.45 长春市 长春大学

211.93.120.129 吉林市 吉林大学

211.93.120.200 长春 长春KTV

现有6亿个IP地址，请编写程序，读取IP地址便转换成IP地址相对应的城市，要求有较好的时间复杂度和空间复杂度。

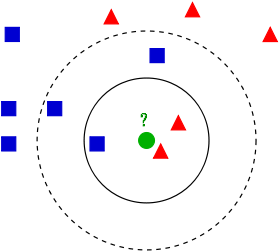
# 第七章 机器学习

# 7.1 K近邻算法

### 1.1、什么是K近邻算法

何谓K近邻算法，即K-Nearest Neighbor algorithm，简称KNN算法，单从名字来猜想，可以简单粗暴的认为是：K个最近的邻居，当K=1时，算法便成了最近邻算法，即寻找最近的那个邻居。为何要找邻居？打个比方来说，假设你来到一个陌生的村庄，现在你要找到与你有着相似特征的人群融入他们，所谓入伙。

用官方的话来说，所谓K近邻算法，即是给定一个训练数据集，对新的输入实例，在训练数据集中找到与该实例最邻近的K个实例（也就是上面所说的K个邻居），这K个实例的多数属于某个类，就把该输入实例分类到这个类中。根据这个说法，咱们来看下引自维基百科上的一幅图：

[](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/10/10.2/10.2.2.png)

如上图所示，有两类不同的样本数据，分别用蓝色的小正方形和红色的小三角形表示，而图正中间的那个绿色的圆所标示的数据则是待分类的数据。也就是说，现在，我们不知道中间那个绿色的数据是从属于哪一类（蓝色小正方形or红色小三角形），下面，我们就要解决这个问题：给这个绿色的圆分类。

* 我们常说，物以类聚，人以群分，判别一个人是一个什么样品质特征的人，常常可以从他/她身边的朋友入手，所谓观其友，而识其人。我们不是要判别上图中那个绿色的圆是属于哪一类数据么，好说，从它的邻居下手。但一次性看多少个邻居呢？从上图中，你还能看到：
* 如果K=3，绿色圆点的最近的3个邻居是2个红色小三角形和1个蓝色小正方形，少数从属于多数，基于统计的方法，判定绿色的这个待分类点属于红色的三角形一类。 如果K=5，绿色圆点的最近的5个邻居是2个红色三角形和3个蓝色的正方形，还是少数从属于多数，基于统计的方法，判定绿色的这个待分类点属于蓝色的正方形一类。

于此我们看到，当无法判定当前待分类点是从属于已知分类中的哪一类时，我们可以依据统计学的理论看它所处的位置特征，衡量它周围邻居的权重，而把它归为(或分配)到权重更大的那一类。这就是K近邻算法的核心思想。

### 1.2、近邻的距离度量表示法

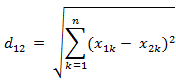
上文第一节，我们看到，K近邻算法的核心在于找到实例点的邻居，这个时候，问题就接踵而至了，如何找到邻居，邻居的判定标准是什么，用什么来度量。这一系列问题便是下面要讲的距离度量表示法。但有的读者可能就有疑问了，我是要找邻居，找相似性，怎么又跟距离扯上关系了？

这是因为特征空间中两个实例点的距离和反应出两个实例点之间的相似性程度。K近邻模型的特征空间一般是n维实数向量空间，使用的距离可以使欧式距离，也是可以是其它距离，既然扯到了距离，下面就来具体阐述下都有哪些距离度量的表示法，权当扩展。

1.**欧氏距离**，最常见的两点之间或多点之间的距离表示法，又称之为欧几里得度量，它定义于欧几里得空间中，如点 x = (x1,...,xn) 和 y = (y1,...,yn) 之间的距离为：  
[https://github.com/julycoding/The-Art-Of-Programming-By-July/raw/master/ebook/images/10/10.2/10.2.3.png](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/10/10.2/10.2.3.png)

(1)二维平面上两点a(x1,y1)与b(x2,y2)间的欧氏距离：  
[https://github.com/julycoding/The-Art-Of-Programming-By-July/raw/master/ebook/images/10/10.2/10.2.4.png](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/10/10.2/10.2.4.png)

(2)三维空间两点a(x1,y1,z1)与b(x2,y2,z2)间的欧氏距离：  
[https://github.com/julycoding/The-Art-Of-Programming-By-July/raw/master/ebook/images/10/10.2/10.2.5.png](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/10/10.2/10.2.5.png)

(3)两个n维向量a(x11,x12,…,x1n)与 b(x21,x22,…,x2n)间的欧氏距离：  
[](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/10/10.2/10.2.6.png)

也可以用表示成向量运算的形式：  
[https://github.com/julycoding/The-Art-Of-Programming-By-July/raw/master/ebook/images/10/10.2/10.2.7.png](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/10/10.2/10.2.7.png)

其上，二维平面上两点欧式距离，代码可以如下编写：

//unixfy：计算欧氏距离

double euclideanDistance(const vector<double>& v1, const vector<double>& v2)

{

assert(v1.size() == v2.size());

double ret = 0.0;

for (vector<double>::size\_type i = 0; i != v1.size(); ++i)

{

ret += (v1[i] - v2[i]) \* (v1[i] - v2[i]);

}

return sqrt(ret);

}

2.**曼哈顿距离**，我们可以定义曼哈顿距离的正式意义为L1-距离或城市区块距离，也就是在欧几里得空间的固定直角坐标系上两点所形成的线段对轴产生的投影的距离总和。例如在平面上，坐标（x1, y1）的点P1与坐标（x2, y2）的点P2的曼哈顿距离为：[https://github.com/julycoding/The-Art-Of-Programming-By-July/raw/master/ebook/images/10/10.2/10.2.8.png](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/10/10.2/10.2.8.png)，要注意的是，曼哈顿距离依赖座标系统的转度，而非系统在座标轴上的平移或映射。

通俗来讲，想象你在曼哈顿要从一个十字路口开车到另外一个十字路口，驾驶距离是两点间的直线距离吗？显然不是，除非你能穿越大楼。而实际驾驶距离就是这个“曼哈顿距离”，此即曼哈顿距离名称的来源， 同时，曼哈顿距离也称为城市街区距离(City Block distance)。

(1)二维平面两点a(x1,y1)与b(x2,y2)间的曼哈顿距离  
[https://github.com/julycoding/The-Art-Of-Programming-By-July/raw/master/ebook/images/10/10.2/10.2.9.png](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/10/10.2/10.2.9.png)

(2)两个n维向量a(x11,x12,…,x1n)与 b(x21,x22,…,x2n)间的曼哈顿距离  
[https://github.com/julycoding/The-Art-Of-Programming-By-July/raw/master/ebook/images/10/10.2/10.2.10.png](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/10/10.2/10.2.10.png)

3.**切比雪夫距离**，若二个向量或二个点p 、and q，其座标分别为及，则两者之间的切比雪夫距离定义如下：[https://github.com/julycoding/The-Art-Of-Programming-By-July/raw/master/ebook/images/10/10.2/10.2.11.png](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/10/10.2/10.2.11.png)，

这也等于以下Lp度量的极值：[https://github.com/julycoding/The-Art-Of-Programming-By-July/raw/master/ebook/images/10/10.2/10.2.12.png](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/10/10.2/10.2.12.png)，因此切比雪夫距离也称为L∞度量。

以数学的观点来看，切比雪夫距离是由一致范数（uniform norm）（或称为上确界范数）所衍生的度量，也是超凸度量（injective metric space）的一种。

在平面几何中，若二点p及q的直角坐标系坐标为[https://github.com/julycoding/The-Art-Of-Programming-By-July/raw/master/ebook/images/10/10.2/10.2.13.png](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/10/10.2/10.2.13.png)及[https://github.com/julycoding/The-Art-Of-Programming-By-July/raw/master/ebook/images/10/10.2/10.2.14.png](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/10/10.2/10.2.14.png)，则切比雪夫距离为：[https://github.com/julycoding/The-Art-Of-Programming-By-July/raw/master/ebook/images/10/10.2/10.2.15.png](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/10/10.2/10.2.15.png)。

玩过国际象棋的朋友或许知道，国王走一步能够移动到相邻的8个方格中的任意一个。那么国王从格子(x1,y1)走到格子(x2,y2)最少需要多少步？。你会发现最少步数总是max( | x2-x1 | , | y2-y1 | ) 步 。有一种类似的一种距离度量方法叫切比雪夫距离。

(1)二维平面两点a(x1,y1)与b(x2,y2)间的切比雪夫距离  
[https://github.com/julycoding/The-Art-Of-Programming-By-July/raw/master/ebook/images/10/10.2/10.2.16.png](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/10/10.2/10.2.16.png)

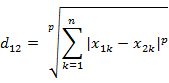
(2)两个n维向量a(x11,x12,…,x1n)与 b(x21,x22,…,x2n)间的切比雪夫距离 　　[https://github.com/julycoding/The-Art-Of-Programming-By-July/raw/master/ebook/images/10/10.2/10.2.17.png](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/10/10.2/10.2.17.png)

这个公式的另一种等价形式是[https://github.com/julycoding/The-Art-Of-Programming-By-July/raw/master/ebook/images/10/10.2/10.2.18.png](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/10/10.2/10.2.18.png)

4.**闵可夫斯基距离(Minkowski Distance)**，闵氏距离不是一种距离，而是一组距离的定义。

(1) 闵氏距离的定义

两个n维变量a(x11,x12,…,x1n)与 b(x21,x22,…,x2n)间的闵可夫斯基距离定义为：

[](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/10/10.2/10.2.19.png)

其中p是一个变参数。

当p=1时，就是曼哈顿距离

当p=2时，就是欧氏距离

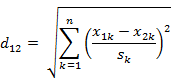
当p→∞时，就是切比雪夫距离

根据变参数的不同，闵氏距离可以表示一类的距离。

5.**标准化欧氏距离 (Standardized Euclidean distance)**，标准化欧氏距离是针对简单欧氏距离的缺点而作的一种改进方案。标准欧氏距离的思路：既然数据各维分量的分布不一样，那先将各个分量都“标准化”到均值、方差相等。至于均值和方差标准化到多少，先复习点统计学知识。

假设样本集X的数学期望或均值(mean)为m，标准差(standard deviation，方差开根)为s，那么X的“标准化变量”X\*表示为：(X-m）/s，而且标准化变量的数学期望为0，方差为1。 即，样本集的标准化过程(standardization)用公式描述就是：  
[https://github.com/julycoding/The-Art-Of-Programming-By-July/raw/master/ebook/images/10/10.2/10.2.20.png](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/10/10.2/10.2.20.png)

标准化后的值 = ( 标准化前的值 － 分量的均值 ) /分量的标准差

经过简单的推导就可以得到两个n维向量a(x11,x12,…,x1n)与 b(x21,x22,…,x2n)间的标准化欧氏距离的公式：  
[](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/10/10.2/10.2.21.png)

如果将方差的倒数看成是一个权重，这个公式可以看成是一种加权欧氏距离(Weighted Euclidean distance)。

6.**马氏距离(Mahalanobis Distance)**

（1）马氏距离定义

有M个样本向量X1~Xm，[协方差矩阵](http://zh.wikipedia.org/wiki/%E5%8D%8F%E6%96%B9%E5%B7%AE%E7%9F%A9%E9%98%B5)记为S，均值记为向量μ，则其中样本向量X到u的马氏距离表示为：  
[https://github.com/julycoding/The-Art-Of-Programming-By-July/raw/master/ebook/images/10/10.2/10.2.22.png](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/10/10.2/10.2.22.png)

（协方差矩阵中每个元素是各个矢量元素之间的协方差Cov(X,Y)，Cov(X,Y) = E{ [X-E(X)] [Y-E(Y)]}，其中E为数学期望）

而其中向量Xi与Xj之间的马氏距离定义为：  
[https://github.com/julycoding/The-Art-Of-Programming-By-July/raw/master/ebook/images/10/10.2/10.2.23.png](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/10/10.2/10.2.23.png)

若协方差矩阵是单位矩阵（各个样本向量之间独立同分布）,则公式就成了：  
[https://github.com/julycoding/The-Art-Of-Programming-By-July/raw/master/ebook/images/10/10.2/10.2.24.png](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/10/10.2/10.2.24.png)

也就是欧氏距离了。

若协方差矩阵是对角矩阵，公式变成了标准化欧氏距离。

(2)马氏距离的优缺点：量纲无关，排除变量之间的相关性的干扰。

「微博上的seafood高清版点评道：原来马氏距离是根据协方差矩阵演变，一直被老师误导了，怪不得看Killian在05年NIPS发表的LMNN论文时候老是看到协方差矩阵和半正定，原来是这回事」

7.**巴氏距离（Bhattacharyya Distance）**，在统计中，Bhattacharyya距离测量两个离散或连续概率分布的相似性。它与衡量两个统计样品或种群之间的重叠量的Bhattacharyya系数密切相关。Bhattacharyya距离和Bhattacharyya系数以20世纪30年代曾在印度统计研究所工作的一个统计学家A. Bhattacharya命名。同时，Bhattacharyya系数可以被用来确定两个样本被认为相对接近的，它是用来测量中的类分类的可分离性。

（1）巴氏距离的定义

对于离散概率分布 p和q在同一域 X，它被定义为：  
[https://github.com/julycoding/The-Art-Of-Programming-By-July/raw/master/ebook/images/10/10.2/10.2.25.png](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/10/10.2/10.2.25.png)

其中：  
[https://github.com/julycoding/The-Art-Of-Programming-By-July/raw/master/ebook/images/10/10.2/10.2.26.png](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/10/10.2/10.2.26.png)

是Bhattacharyya系数。

对于连续概率分布，Bhattacharyya系数被定义为：  
[https://github.com/julycoding/The-Art-Of-Programming-By-July/raw/master/ebook/images/10/10.2/10.2.27.png](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/10/10.2/10.2.27.png)

在[https://github.com/julycoding/The-Art-Of-Programming-By-July/raw/master/ebook/images/10/10.2/10.2.28.jpg](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/10/10.2/10.2.28.jpg)这两种情况下，巴氏距离[https://github.com/julycoding/The-Art-Of-Programming-By-July/raw/master/ebook/images/10/10.2/10.2.29.png](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/10/10.2/10.2.29.png)并没有服从三角不等式.（值得一提的是，Hellinger距离不服从三角不等式[https://github.com/julycoding/The-Art-Of-Programming-By-July/raw/master/ebook/images/10/10.2/10.2.30.jpg](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/10/10.2/10.2.30.jpg)）。

对于多变量的高斯分布[https://github.com/julycoding/The-Art-Of-Programming-By-July/raw/master/ebook/images/10/10.2/10.2.31.png](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/10/10.2/10.2.31.png)，[https://github.com/julycoding/The-Art-Of-Programming-By-July/raw/master/ebook/images/10/10.2/10.2.32.png](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/10/10.2/10.2.32.png)， 和是手段和协方差的分布[https://github.com/julycoding/The-Art-Of-Programming-By-July/raw/master/ebook/images/10/10.2/10.2.33.png](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/10/10.2/10.2.33.png)。

需要注意的是，在这种情况下，第一项中的Bhattacharyya距离与马氏距离有关联。

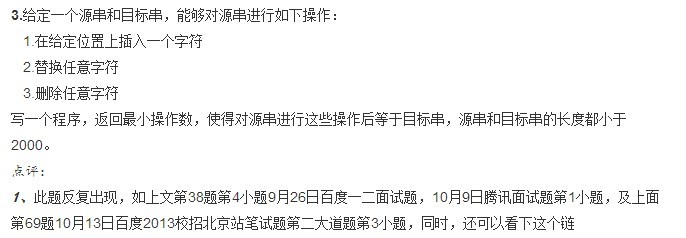
（2）Bhattacharyya系数

Bhattacharyya系数是两个统计样本之间的重叠量的近似测量，可以被用于确定被考虑的两个样本的相对接近。

计算Bhattacharyya系数涉及集成的基本形式的两个样本的重叠的时间间隔的值的两个样本被分裂成一个选定的分区数，并且在每个分区中的每个样品的成员的数量，在下面的公式中使用  
[https://github.com/julycoding/The-Art-Of-Programming-By-July/raw/master/ebook/images/10/10.2/10.2.34.png](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/10/10.2/10.2.34.png)

考虑样品a 和 b ，n是的分区数，并且[https://github.com/julycoding/The-Art-Of-Programming-By-July/raw/master/ebook/images/10/10.2/10.2.35.png](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/10/10.2/10.2.35.png)，[https://github.com/julycoding/The-Art-Of-Programming-By-July/raw/master/ebook/images/10/10.2/10.2.36.png](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/10/10.2/10.2.36.png)被一个 和 b i的日分区中的样本数量的成员。更多介绍请参看：<http://en.wikipedia.org/wiki/Bhattacharyya_coefficient>。

8.**汉明距离(Hamming distance)**， 两个等长字符串s1与s2之间的汉明距离定义为将其中一个变为另外一个所需要作的最小替换次数。例如字符串“1111”与“1001”之间的汉明距离为2。应用：信息编码（为了增强容错性，应使得编码间的最小汉明距离尽可能大）。 或许，你还没明白我再说什么，不急，看下上篇blog中第78题的第3小题整理的一道面试题目，便一目了然了。如下图所示：

[](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/10/10.2/10.2.37.jpg)

//动态规划：

//f[i,j]表示s[0...i]与t[0...j]的最小编辑距离。

f[i,j] = min { f[i-1,j]+1, f[i,j-1]+1, f[i-1,j-1]+(s[i]==t[j]?0:1) }

//分别表示：添加1个，删除1个，替换1个（相同就不用替换）。

与此同时，面试官还可以继续问下去：那么，请问，如何设计一个比较两篇文章相似性的算法？（这个问题的讨论可以看看这里：<http://t.cn/zl82CAH>，及这里关于simhash算法的介绍：<http://www.cnblogs.com/linecong/archive/2010/08/28/simhash.html>），接下来，便引出了下文关于夹角余弦的讨论。（[*上篇blog*](http://blog.csdn.net/v_july_v/article/details/7974418)中第78题的第3小题给出了多种方法，读者可以参看之。同时，程序员编程艺术系列第二十八章将详细阐述这个问题）

9.**夹角余弦(Cosine)** ，几何中夹角余弦可用来衡量两个向量方向的差异，机器学习中借用这一概念来衡量样本向量之间的差异。

(1)在二维空间中向量A(x1,y1)与向量B(x2,y2)的夹角余弦公式：  
[https://github.com/julycoding/The-Art-Of-Programming-By-July/raw/master/ebook/images/10/10.2/10.2.38.png](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/10/10.2/10.2.38.png)

(2) 两个n维样本点a(x11,x12,…,x1n)和b(x21,x22,…,x2n)的夹角余弦  
[https://github.com/julycoding/The-Art-Of-Programming-By-July/raw/master/ebook/images/10/10.2/10.2.39.png](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/10/10.2/10.2.39.png)

类似的，对于两个n维样本点a(x11,x12,…,x1n)和b(x21,x22,…,x2n)，可以使用类似于夹角余弦的概念来衡量它们间的相似程度，即：  
[https://github.com/julycoding/The-Art-Of-Programming-By-July/raw/master/ebook/images/10/10.2/10.2.40.png](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/10/10.2/10.2.40.png)

夹角余弦取值范围为[-1,1]。夹角余弦越大表示两个向量的夹角越小，夹角余弦越小表示两向量的夹角越大。当两个向量的方向重合时夹角余弦取最大值1，当两个向量的方向完全相反夹角余弦取最小值-1。

10.**杰卡德相似系数(Jaccard similarity coefficient)**

(1) 杰卡德相似系数

两个集合A和B的交集元素在A，B的并集中所占的比例，称为两个集合的杰卡德相似系数，用符号J(A,B)表示。  
[https://github.com/julycoding/The-Art-Of-Programming-By-July/raw/master/ebook/images/10/10.2/10.2.41.png](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/10/10.2/10.2.41.png)

杰卡德相似系数是衡量两个集合的相似度一种指标。

(2) 杰卡德距离

与杰卡德相似系数相反的概念是杰卡德距离(Jaccard distance)。

杰卡德距离可用如下公式表示：  
[https://github.com/julycoding/The-Art-Of-Programming-By-July/raw/master/ebook/images/10/10.2/10.2.42.png](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/10/10.2/10.2.42.png)

杰卡德距离用两个集合中不同元素占所有元素的比例来衡量两个集合的区分度。

(3) 杰卡德相似系数与杰卡德距离的应用

可将杰卡德相似系数用在衡量样本的相似度上。

举例：样本A与样本B是两个n维向量，而且所有维度的取值都是0或1，例如：A(0111)和B(1011)。我们将样本看成是一个集合，1表示集合包含该元素，0表示集合不包含该元素。

M11 ：样本A与B都是1的维度的个数

M01：样本A是0，样本B是1的维度的个数

M10：样本A是1，样本B是0 的维度的个数

M00：样本A与B都是0的维度的个数

依据上文给的杰卡德相似系数及杰卡德距离的相关定义，样本A与B的杰卡德相似系数J可以表示为：  
[https://github.com/julycoding/The-Art-Of-Programming-By-July/raw/master/ebook/images/10/10.2/10.2.43.png](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/10/10.2/10.2.43.png)

这里M11+M01+M10可理解为A与B的并集的元素个数，而M11是A与B的交集的元素个数。而样本A与B的杰卡德距离表示为J'：  
[https://github.com/julycoding/The-Art-Of-Programming-By-July/raw/master/ebook/images/10/10.2/10.2.44.png](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/10/10.2/10.2.44.png)

11.**皮尔逊系数(Pearson Correlation Coefficient)**

在具体阐述皮尔逊相关系数之前，有必要解释下什么是相关系数 ( Correlation coefficient )与相关距离(Correlation distance)。

相关系数 ( Correlation coefficient )的定义是：  
[https://github.com/julycoding/The-Art-Of-Programming-By-July/raw/master/ebook/images/10/10.2/10.2.45.png](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/10/10.2/10.2.45.png)

(其中，E为数学期望或均值，D为方差，D开根号为标准差，E{ [X-E(X)] [Y-E(Y)]}称为随机变量X与Y的协方差，记为Cov(X,Y)，即Cov(X,Y) = E{ [X-E(X)] [Y-E(Y)]}，而两个变量之间的协方差和标准差的商则称为随机变量X与Y的相关系数，记为[https://github.com/julycoding/The-Art-Of-Programming-By-July/raw/master/ebook/images/10/10.2/10.2.46.jpg](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/10/10.2/10.2.46.jpg))

相关系数衡量随机变量X与Y相关程度的一种方法，相关系数的取值范围是[-1,1]。相关系数的绝对值越大，则表明X与Y相关度越高。当X与Y线性相关时，相关系数取值为1（正线性相关）或-1（负线性相关）。

具体的，如果有两个变量：X、Y，最终计算出的相关系数的含义可以有如下理解：

当相关系数为0时，X和Y两变量无关系。

当X的值增大（减小），Y值增大（减小），两个变量为正相关，相关系数在0.00与1.00之间。

当X的值增大（减小），Y值减小（增大），两个变量为负相关，相关系数在-1.00与0.00之间。

相关距离的定义是：

[https://github.com/julycoding/The-Art-Of-Programming-By-July/raw/master/ebook/images/10/10.2/10.2.47.png](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/10/10.2/10.2.47.png)

OK，接下来，咱们来重点了解下皮尔逊相关系数。

在统计学中，皮尔逊积矩相关系数（英语：Pearson product-moment correlation coefficient，又称作 PPMCC或PCCs, 用r表示）用于度量两个变量X和Y之间的相关（线性相关），其值介于-1与1之间。

通常情况下通过以下取值范围判断变量的相关强度：

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

在自然科学领域中，该系数广泛用于度量两个变量之间的相关程度。它是由卡尔·皮尔逊从弗朗西斯·高尔顿在19世纪80年代提出的一个相似却又稍有不同的想法演变而来的。这个相关系数也称作“皮尔森相关系数r”。

**(1)皮尔逊系数的定义**：

两个变量之间的皮尔逊相关系数定义为两个变量之间的协方差和标准差的商：  
[https://github.com/julycoding/The-Art-Of-Programming-By-July/raw/master/ebook/images/10/10.2/10.2.48.png](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/10/10.2/10.2.48.png)

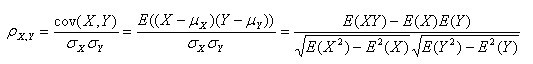
以上方程定义了总体相关系数, 一般表示成希腊字母ρ(rho)。基于样本对协方差和方差进行估计，可以得到样本标准差, 一般表示成r：  
[https://github.com/julycoding/The-Art-Of-Programming-By-July/raw/master/ebook/images/10/10.2/10.2.49.png](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/10/10.2/10.2.49.png)

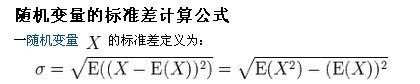
一种等价表达式的是表示成标准分的均值。基于(Xi, Yi)的样本点，样本皮尔逊系数是  
[https://github.com/julycoding/The-Art-Of-Programming-By-July/raw/master/ebook/images/10/10.2/10.2.50.png](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/10/10.2/10.2.50.png)

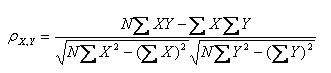
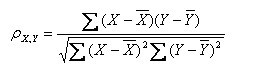
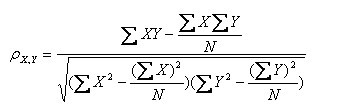
其中[https://github.com/julycoding/The-Art-Of-Programming-By-July/raw/master/ebook/images/10/10.2/10.2.51.png](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/10/10.2/10.2.51.png)、[https://github.com/julycoding/The-Art-Of-Programming-By-July/raw/master/ebook/images/10/10.2/10.2.52.png](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/10/10.2/10.2.52.png)及[https://github.com/julycoding/The-Art-Of-Programming-By-July/raw/master/ebook/images/10/10.2/10.2.53.png](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/10/10.2/10.2.53.png)，分别是标准分、样本平均值和样本标准差。

或许上面的讲解令你头脑混乱不堪，没关系，我换一种方式讲解，如下：

假设有两个变量X、Y，那么两变量间的皮尔逊相关系数可通过以下公式计算：

* 公式一：[](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/10/10.2/10.2.54.jpg)

注：勿忘了上面说过，“皮尔逊相关系数定义为两个变量之间的协方差和标准差的商”，其中标准差的计算公式为：[](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/10/10.2/10.2.55.jpg)

* 公式二：[](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/10/10.2/10.2.56.jpg)
* 公式三：[](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/10/10.2/10.2.57.jpg)
* 公式四：[](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/10/10.2/10.2.58.jpg)

以上列出的四个公式等价，其中E是[数学期望](http://zh.wikipedia.org/wiki/%E6%95%B0%E5%AD%A6%E6%9C%9F%E6%9C%9B)，cov表示[协方差](http://zh.wikipedia.org/wiki/%E5%8D%8F%E6%96%B9%E5%B7%AE)，N表示变量取值的个数。

**(2)皮尔逊相关系数的适用范围**

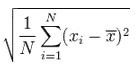
当两个变量的标准差都不为零时，相关系数才有定义，皮尔逊相关系数适用于：

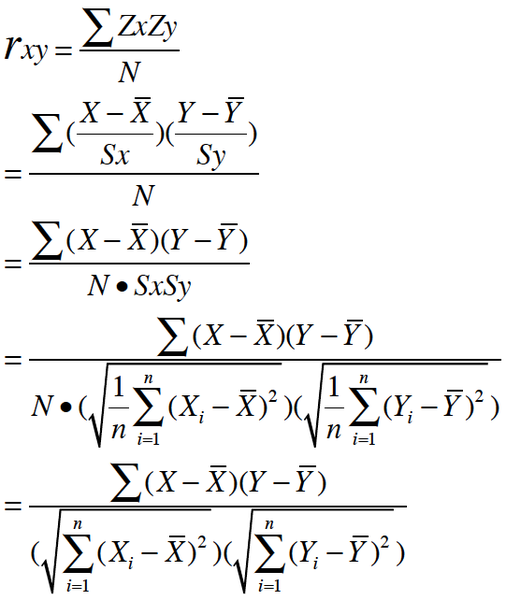
1. 两个变量之间是线性关系，都是连续数据。
2. 两个变量的总体是正态分布，或接近正态的单峰分布。
3. 两个变量的观测值是成对的，每对观测值之间相互独立。

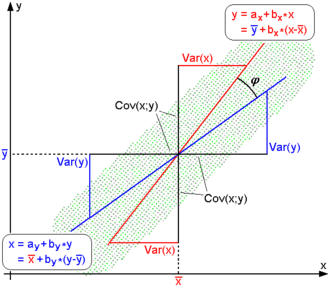
**(3)如何理解皮尔逊相关系数**

rubyist：皮尔逊相关系数理解有两个角度

其一, 按照高中数学水平来理解, 它很简单, 可以看做将两组数据首先做Z分数处理之后, 然后两组数据的乘积和除以样本数，Z分数一般代表正态分布中, 数据偏离中心点的距离.等于变量减掉平均数再除以标准差.(就是高考的标准分类似的处理)

样本标准差则等于变量减掉平均数的平方和，再除以样本数，最后再开方，也就是说，方差开方即为标准差，样本标准差计算公式为：  
[](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/10/10.2/10.2.59.jpg)

所以, 根据这个最朴素的理解,我们可以将公式依次精简为:  
[](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/10/10.2/10.2.60.png)

其二, 按照大学的线性数学水平来理解, 它比较复杂一点,可以看做是两组数据的向量夹角的余弦。下面是关于此皮尔逊系数的几何学的解释，先来看一幅图，如下所示：  
[](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/10/10.2/10.2.61.png)

回归直线： y=gx(x) [红色] 和 x=gy(y) [蓝色]

如上图，对于没有中心化的数据, 相关系数与两条可能的回归线y=gx(x) 和 x=gy(y) 夹角的余弦值一致。 对于没有中心化的数据 (也就是说, 数据移动一个样本平均值以使其均值为0), 相关系数也可以被视作由两个随机变量 向量 夹角 的 余弦值（见下方）。

举个例子，例如，有5个国家的国民生产总值分别为 10, 20, 30, 50 和 80 亿美元。 假设这5个国家 (顺序相同) 的贫困百分比分别为 11%, 12%, 13%, 15%, and 18% 。 令 x 和 y 分别为包含上述5个数据的向量: x = (1, 2, 3, 5, 8) 和 y = (0.11, 0.12, 0.13, 0.15, 0.18)。

利用通常的方法计算两个向量之间的夹角 (参见 数量积), 未中心化 的相关系数是:  
[https://github.com/julycoding/The-Art-Of-Programming-By-July/raw/master/ebook/images/10/10.2/10.2.62.png](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/10/10.2/10.2.62.png)

我们发现以上的数据特意选定为完全相关: y = 0.10 + 0.01 x。 于是，皮尔逊相关系数应该等于1。将数据中心化 (通过E(x) = 3.8移动 x 和通过 E(y) = 0.138 移动 y ) 得到 x = (−2.8, −1.8, −0.8, 1.2, 4.2) 和 y = (−0.028, −0.018, −0.008, 0.012, 0.042), 从中  
[https://github.com/julycoding/The-Art-Of-Programming-By-July/raw/master/ebook/images/10/10.2/10.2.63.png](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/10/10.2/10.2.63.png)

**(4)皮尔逊相关的约束条件**

从以上解释, 也可以理解皮尔逊相关的约束条件:

1. 两个变量间有线性关系
2. 变量是连续变量
3. 变量均符合正态分布,且二元分布也符合正态分布
4. 两变量独立

在实践统计中,一般只输出两个系数,一个是相关系数,也就是计算出来的相关系数大小,在-1到1之间;另一个是独立样本检验系数,用来检验样本一致性。

简单说来，各种“距离”的应用场景简单概括为，空间：欧氏距离，路径：曼哈顿距离，国际象棋国王：切比雪夫距离，以上三种的统一形式:闵可夫斯基距离，加权：标准化欧氏距离，排除量纲和依存：马氏距离，向量差距：夹角余弦，编码差别：汉明距离，集合近似度：杰卡德类似系数与距离，相关：相关系数与相关距离。

### 1.3、K值的选择

除了上述1.2节如何定义邻居的问题之外，还有一个选择多少个邻居，即K值定义为多大的问题。不要小看了这个K值选择问题，因为它对K近邻算法的结果会产生重大影响。如李航博士的一书「统计学习方法」上所说：

1. 如果选择较小的K值，就相当于用较小的领域中的训练实例进行预测，“学习”近似误差会减小，只有与输入实例较近或相似的训练实例才会对预测结果起作用，与此同时带来的问题是“学习”的估计误差会增大，换句话说，K值的减小就意味着整体模型变得复杂，容易发生过拟合；
2. 如果选择较大的K值，就相当于用较大领域中的训练实例进行预测，其优点是可以减少学习的估计误差，但缺点是学习的近似误差会增大。这时候，与输入实例较远（不相似的）训练实例也会对预测器作用，使预测发生错误，且K值的增大就意味着整体的模型变得简单。
3. K=N，则完全不足取，因为此时无论输入实例是什么，都只是简单的预测它属于在训练实例中最多的累，模型过于简单，忽略了训练实例中大量有用信息。

在实际应用中，K值一般取一个比较小的数值，例如采用[交叉验证](http://zh.wikipedia.org/zh/%E4%BA%A4%E5%8F%89%E9%A9%97%E8%AD%89)法（简单来说，就是一部分样本做训练集，一部分做测试集）来选择最优的K值。

# 7.2支持向量机

第一层、了解SVM

支持向量机，因其英文名为support vector machine，故一般简称SVM，通俗来讲，它是一种二类分类模型，其基本模型定义为特征空间上的间隔最大的线性分类器，其学习策略便是间隔最大化，最终可转化为一个凸二次规划问题的求解。

### 1.1、线性分类

理解SVM，咱们必须先弄清楚一个概念：线性分类器。

1.1.1、分类标准

考虑一个二类的分类问题，数据点用x来表示，类别用y来表示，可以取1或者-1，分别代表两个不同的类，且是一个n 维向量，w^T中的T代表转置。一个线性分类器的学习目标就是要在n维的数据空间中找到一个分类[超平面](http://zh.wikipedia.org/wiki/%E8%B6%85%E5%B9%B3%E9%9D%A2)，其方程可以表示为：

[https://github.com/julycoding/The-Art-Of-Programming-By-July/raw/master/ebook/images/svm/1.1.1.jpg](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/svm/1.1.1.jpg)

可能有读者对类别取1或-1有疑问，事实上，这个1或-1的分类标准起源于logistic回归，为了过渡的自然性，咱们就再来看看这个logistic回归。

1.1.2、1或-1分类标准的起源：logistic回归

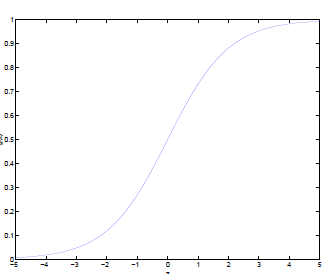
Logistic回归目的是从特征学习出一个0/1分类模型，而这个模型是将特性的线性组合作为自变量，由于自变量的取值范围是负无穷到正无穷。因此，使用logistic函数（或称作sigmoid函数）将自变量映射到(0,1)上，映射后的值被认为是属于y=1的概率。

假设函数

[https://github.com/julycoding/The-Art-Of-Programming-By-July/raw/master/ebook/images/svm/1.1.2-1.png](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/svm/1.1.2-1.png)

其中x是n维特征向量，函数g就是logistic函数。

而[https://github.com/julycoding/The-Art-Of-Programming-By-July/raw/master/ebook/images/svm/1.1.2-2.png](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/svm/1.1.2-2.png)的图像是

[](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/svm/1.1.2-3.png)

可以看到，通过logistic函数将自变量从无穷映射到了(0,1)，而假设函数就是特征属于y=1的概率。

[https://github.com/julycoding/The-Art-Of-Programming-By-July/raw/master/ebook/images/svm/1.1.2-4.png](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/svm/1.1.2-4.png)

从而，当我们要判别一个新来的特征属于哪个类时，只需求[https://github.com/julycoding/The-Art-Of-Programming-By-July/raw/master/ebook/images/svm/1.1.2-5.png](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/svm/1.1.2-5.png)，若大于0.5就是y=1的类，反之属于y=0类。

再审视一下[https://github.com/julycoding/The-Art-Of-Programming-By-July/raw/master/ebook/images/svm/1.1.2-5.png](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/svm/1.1.2-5.png)，发现[https://github.com/julycoding/The-Art-Of-Programming-By-July/raw/master/ebook/images/svm/1.1.2-5.png](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/svm/1.1.2-5.png)只和[https://github.com/julycoding/The-Art-Of-Programming-By-July/raw/master/ebook/images/svm/1.1.2-6.png](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/svm/1.1.2-6.png)有关，[https://github.com/julycoding/The-Art-Of-Programming-By-July/raw/master/ebook/images/svm/1.1.2-6.png](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/svm/1.1.2-6.png)>0，那么[https://github.com/julycoding/The-Art-Of-Programming-By-July/raw/master/ebook/images/svm/1.1.2-5.png](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/svm/1.1.2-5.png)>0.5，g(z)只不过是用来映射，真实的类别决定权还在[https://github.com/julycoding/The-Art-Of-Programming-By-July/raw/master/ebook/images/svm/1.1.2-6.png](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/svm/1.1.2-6.png)。还有当[https://github.com/julycoding/The-Art-Of-Programming-By-July/raw/master/ebook/images/svm/1.1.2-6.png](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/svm/1.1.2-6.png)>>0时，[https://github.com/julycoding/The-Art-Of-Programming-By-July/raw/master/ebook/images/svm/1.1.2-5.png](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/svm/1.1.2-5.png)=1，反之[https://github.com/julycoding/The-Art-Of-Programming-By-July/raw/master/ebook/images/svm/1.1.2-5.png](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/svm/1.1.2-5.png)=0。如果我们只从[https://github.com/julycoding/The-Art-Of-Programming-By-July/raw/master/ebook/images/svm/1.1.2-6.png](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/svm/1.1.2-6.png)出发，希望模型达到的目标无非就是让训练数据中y=1的特征[https://github.com/julycoding/The-Art-Of-Programming-By-July/raw/master/ebook/images/svm/1.1.2-6.png](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/svm/1.1.2-6.png)>>0，而是y=0的特征[https://github.com/julycoding/The-Art-Of-Programming-By-July/raw/master/ebook/images/svm/1.1.2-6.png](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/svm/1.1.2-6.png)<<0。Logistic回归就是要学习得到[https://github.com/julycoding/The-Art-Of-Programming-By-July/raw/master/ebook/images/svm/1.1.2-7.png](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/svm/1.1.2-7.png)，使得正例的特征远大于0，负例的特征远小于0，强调在全部训练实例上达到这个目标。

1.1.3、形式化标示

我们这次使用的结果标签是y=-1,y=1，替换在logistic回归中使用的y=0和y=1。

同时将[https://github.com/julycoding/The-Art-Of-Programming-By-July/raw/master/ebook/images/svm/1.1.2-7.png](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/svm/1.1.2-7.png)替换成w和b。

以前的[https://github.com/julycoding/The-Art-Of-Programming-By-July/raw/master/ebook/images/svm/1.1.3-1.png](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/svm/1.1.3-1.png)，其中认为[https://github.com/julycoding/The-Art-Of-Programming-By-July/raw/master/ebook/images/svm/1.1.3-9.png](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/svm/1.1.3-9.png)，现在我们替换为b；

后面的[https://github.com/julycoding/The-Art-Of-Programming-By-July/raw/master/ebook/images/svm/1.1.3-2.png](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/svm/1.1.3-2.png) 替换为[https://github.com/julycoding/The-Art-Of-Programming-By-July/raw/master/ebook/images/svm/1.1.3-3.png](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/svm/1.1.3-3.png)（即[https://github.com/julycoding/The-Art-Of-Programming-By-July/raw/master/ebook/images/svm/1.1.3-4.png](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/svm/1.1.3-4.png)）。

这样，我们让[https://github.com/julycoding/The-Art-Of-Programming-By-July/raw/master/ebook/images/svm/1.1.3-5.png](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/svm/1.1.3-5.png)，进一步[https://github.com/julycoding/The-Art-Of-Programming-By-July/raw/master/ebook/images/svm/1.1.3-6.png](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/svm/1.1.3-6.png)。也就是说除了y由y=0变为y=-1，只是标记不同外，与logistic回归的形式化表示没区别。

再明确下假设函数

[https://github.com/julycoding/The-Art-Of-Programming-By-July/raw/master/ebook/images/svm/1.1.3-7.png](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/svm/1.1.3-7.png)

上面提到过我们只需考虑的[https://github.com/julycoding/The-Art-Of-Programming-By-July/raw/master/ebook/images/svm/1.1.2-6.png](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/svm/1.1.2-6.png)正负问题，而不用关心g(z)，因此我们这里将g(z)做一个简化，将其简单映射到y=-1和y=1上。映射关系如下：

[https://github.com/julycoding/The-Art-Of-Programming-By-July/raw/master/ebook/images/svm/1.1.3-8.png](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/svm/1.1.3-8.png)

于此，想必已经解释明白了为何线性分类的标准一般用1 或者-1 来标示。

### 1.2、线性分类的一个例子

假定现在有一个一个二维平面，如下图所示，平面上有两种不同的点，分别用两种不同的颜色表示，一种为红颜色的点，另一种为蓝颜色的点，如果我们要在这个二维平面上找到一个可行的超平面的话，那么这个超平面可以是下图中那根红颜色的线(在二维空间中，超平面就是一条直线)。

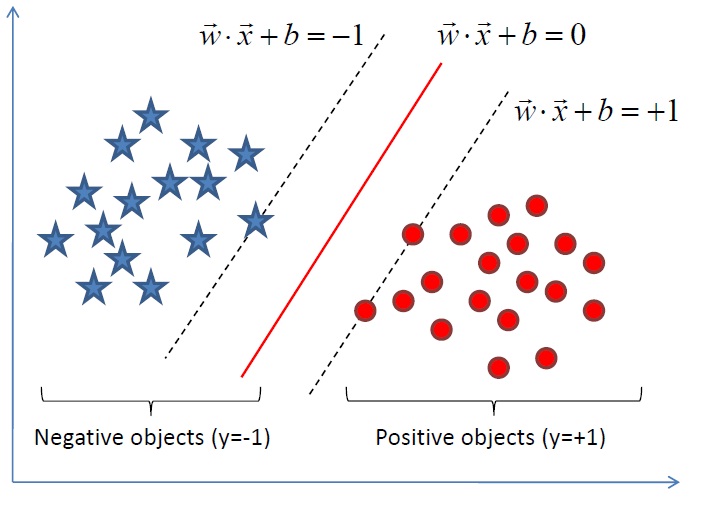
[](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/svm/1.2-1.png)

从上图中我们可以看出，这条红颜色的线作为一个超平面，把红颜色的点和蓝颜色的点分开来了，在超平面一边的数据点所对应的y全是 -1 ，而在另一边全是1。

接着，我们可以令分类函数：

[https://github.com/julycoding/The-Art-Of-Programming-By-July/raw/master/ebook/images/svm/1.2-2.jpeg](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/svm/1.2-2.jpeg)

显然，如果 f(x)=0 ，那么x是位于超平面上的点。我们不妨要求对于所有满足 f(x)0 则对应 y=1 的数据点。

[](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/svm/1.2-3.jpeg)

注：上图中，定义特征到结果的输出函数[https://github.com/julycoding/The-Art-Of-Programming-By-July/raw/master/ebook/images/svm/1.2-4.jpeg](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/svm/1.2-4.jpeg)，与我们之前定义的[https://github.com/julycoding/The-Art-Of-Programming-By-July/raw/master/ebook/images/svm/1.2-2.jpeg](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/svm/1.2-2.jpeg)实质是一样的。为什么？因为无论是，还是，不影响最终优化结果。下文你将看到，当我们转化到优化[https://github.com/julycoding/The-Art-Of-Programming-By-July/raw/master/ebook/images/svm/1.2-5.jpg](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/svm/1.2-5.jpg)的时候，为了求解方便，会把yf(x)令为1，即yf(x)是y(w^x + b)，还是y(w^x - b)，对我们要优化的式子max1/||w||已无影响。

从而在我们进行分类的时候，将数据点 x代入 f(x) 中，如果得到的结果小于 0 ，则赋予其类别 -1 ，如果大于 0 则赋予类别 1 。如果 f(x)=0，则很难办了，分到哪一类都不是。

此外，有些时候，或者说大部分时候数据并不是线性可分的，这时满足这样条件的超平面可能就根本不存在，这里咱们先从最简单的情形开始推导，就假设数据都是线性可分的，亦即这样的超平面是存在的。

### 1.3、函数间隔Functional margin与几何间隔Geometrical margin

一般而言，一个点距离超平面的远近可以表示为分类预测的确信或准确程度。

在超平面w\*x+b=0确定的情况下，|w\*x+b|能够相对的表示点x到距离超平面的远近，而w\*x+b的符号与类标记y的符号是否一致表示分类是否正确，所以，可以用量y\*(w\*x+b)的正负性来判定或表示分类的正确性和确信度。

于此，我们便引出了定义样本到分类间隔距离的函数间隔functional margin的概念。

1.3.1、函数间隔Functional margin

我们定义函数间隔functional margin 为：

[https://github.com/julycoding/The-Art-Of-Programming-By-July/raw/master/ebook/images/svm/1.3.1-1.jpeg](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/svm/1.3.1-1.jpeg)

接着，我们定义超平面(w，b)关于训练数据集T的函数间隔为超平面(w，b)关于T中所有样本点(xi，yi)的函数间隔最小值，其中，x是特征，y是结果标签，i表示第i个样本，有：

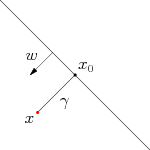
[https://github.com/julycoding/The-Art-Of-Programming-By-July/raw/master/ebook/images/svm/1.2-7.jpeg](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/svm/1.2-7.jpeg)= min[https://github.com/julycoding/The-Art-Of-Programming-By-July/raw/master/ebook/images/svm/1.2-7.jpeg](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/svm/1.2-7.jpeg)i (i=1，...n)

然与此同时，问题就出来了：上述定义的函数间隔虽然可以表示分类预测的正确性和确信度，但在选择分类超平面时，只有函数间隔还远远不够，因为如果成比例的改变w和b，如将他们改变为2w和2b，虽然此时超平面没有改变，但函数间隔的值f(x)却变成了原来的2倍。

事实上，我们可以对法向量w加些约束条件，使其表面上看起来规范化，如此，我们很快又将引出真正定义点到超平面的距离--几何间隔geometrical margin的概念（很快你将看到，几何间隔就是函数间隔除以个||w||，即yf(x) / ||w||）。

1.3.2、点到超平面的距离定义：几何间隔Geometrical margin

对于一个点 x ，令其垂直投影到超平面上的对应的为 x0 ，w 是垂直于超平面的一个向量，[https://github.com/julycoding/The-Art-Of-Programming-By-July/raw/master/ebook/images/svm/1.2-7.jpeg](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/svm/1.2-7.jpeg)为样本x到分类间隔的距离，

[](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/svm/1.3.2-1.png)

我们有

[https://github.com/julycoding/The-Art-Of-Programming-By-July/raw/master/ebook/images/svm/1.3.2-2.jpeg](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/svm/1.3.2-2.jpeg)

其中，||w||表示的是范数。

又由于 x0 是超平面上的点，满足 f(x0)=0 ，代入超平面的方程即可算出：

[https://github.com/julycoding/The-Art-Of-Programming-By-July/raw/master/ebook/images/svm/1.3.2-3.jpeg](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/svm/1.3.2-3.jpeg)

不过这里的[https://github.com/julycoding/The-Art-Of-Programming-By-July/raw/master/ebook/images/svm/1.2-7.jpeg](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/svm/1.2-7.jpeg)是带符号的，我们需要的只是它的绝对值，因此类似地，也乘上对应的类别 y即可，因此实际上我们定义 几何间隔geometrical margin 为(注：别忘了，上面[https://github.com/julycoding/The-Art-Of-Programming-By-July/raw/master/ebook/images/svm/1.2-7.jpeg](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/svm/1.2-7.jpeg)的定义，[https://github.com/julycoding/The-Art-Of-Programming-By-July/raw/master/ebook/images/svm/1.2-7.jpeg](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/svm/1.2-7.jpeg)=y(wTx+b)=yf(x) )：

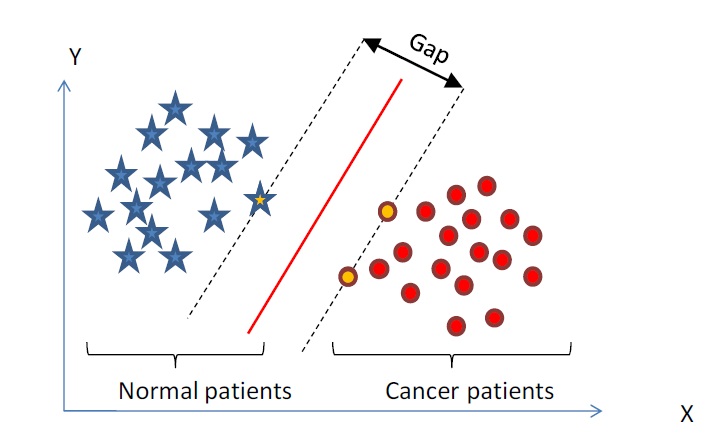
[https://github.com/julycoding/The-Art-Of-Programming-By-July/raw/master/ebook/images/svm/1.3.2-4.jpeg](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/svm/1.3.2-4.jpeg)

代人相关式子可以得出：yi\*(w/||w|| + b/||w||)。

综上，函数间隔y(wx+b)=yf(x)实际上就是|f(x)|，只是人为定义的一个间隔度量；而几何间隔|f(x)|/||w||才是直观上的点到超平面距离。

### 1.4、最大间隔分类器Maximum Margin Classifier的定义

由上，我们已经知道，函数间隔functional margin 和 几何间隔geometrical margin 相差一个的缩放因子。按照我们前面的分析，对一个数据点进行分类，当它的 margin 越大的时候，分类的 confidence 越大。对于一个包含 n 个点的数据集，我们可以很自然地定义它的 margin 为所有这n个点的 margin 值中最小的那个。于是，为了使得分类的 confidence 高，我们希望所选择的超平面hyper plane 能够最大化这个 margin 值。

[](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/svm/1.4-1.jpeg)

且

functional margin 明显是不太适合用来最大化的一个量，因为在 hyper plane 固定以后，我们可以等比例地缩放 w 的长度和 b 的值，这样可以使得[https://github.com/julycoding/The-Art-Of-Programming-By-July/raw/master/ebook/images/svm/1.2-2.jpeg](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/svm/1.2-2.jpeg)的值任意大，亦即 functional margin可以在 hyper plane 保持不变的情况下被取得任意大，

而 geometrical margin 则没有这个问题，因为除上了[https://github.com/julycoding/The-Art-Of-Programming-By-July/raw/master/ebook/images/svm/1.4-4.jpeg](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/svm/1.4-4.jpeg)这个分母，所以缩放 w 和 b 的时候[https://github.com/julycoding/The-Art-Of-Programming-By-July/raw/master/ebook/images/svm/1.2-7.jpeg](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/svm/1.2-7.jpeg)的值是不会改变的，它只随着 hyper plane 的变动而变动，因此，这是更加合适的一个 margin 。

这样一来，我们的 maximum margin classifier 的目标函数可以定义为：

[https://github.com/julycoding/The-Art-Of-Programming-By-July/raw/master/ebook/images/svm/1.4-2.jpeg](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/svm/1.4-2.jpeg)

当然，还需要满足一定的约束条件：

[https://github.com/julycoding/The-Art-Of-Programming-By-July/raw/master/ebook/images/svm/1.4-3.jpg](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/svm/1.4-3.jpg)

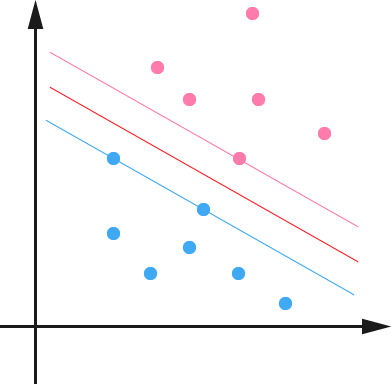
其中[https://github.com/julycoding/The-Art-Of-Programming-By-July/raw/master/ebook/images/svm/1.4-5.jpeg](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/svm/1.4-5.jpeg) (等价于[https://github.com/julycoding/The-Art-Of-Programming-By-July/raw/master/ebook/images/svm/1.2-7.jpeg](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/svm/1.2-7.jpeg)= [https://github.com/julycoding/The-Art-Of-Programming-By-July/raw/master/ebook/images/svm/1.2-7.jpeg](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/svm/1.2-7.jpeg)/||w||，故有稍后的 [https://github.com/julycoding/The-Art-Of-Programming-By-July/raw/master/ebook/images/svm/1.2-7.jpeg](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/svm/1.2-7.jpeg)=1 时， [https://github.com/julycoding/The-Art-Of-Programming-By-July/raw/master/ebook/images/svm/1.2-7.jpeg](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/svm/1.2-7.jpeg)= 1 / ||w||)，处于方便推导和优化的目的，我们可以令[https://github.com/julycoding/The-Art-Of-Programming-By-July/raw/master/ebook/images/svm/1.2-7.jpeg](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/svm/1.2-7.jpeg)=1(对目标函数的优化没有影响) ，此时，上述的目标函数[https://github.com/julycoding/The-Art-Of-Programming-By-July/raw/master/ebook/images/svm/1.2-7.jpeg](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/svm/1.2-7.jpeg)转化为：

[https://github.com/julycoding/The-Art-Of-Programming-By-July/raw/master/ebook/images/svm/1.4-6.jpg](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/svm/1.4-6.jpg)

其中，s.t.，即subject to的意思，它导出的是约束条件。

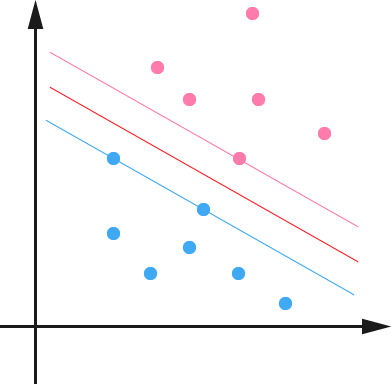
通过求解这个问题，我们就可以找到一个 margin 最大的 classifier ，通过最大化 margin ，我们使得该分类器对数据进行分类时具有了最大的 confidence，从而设计决策最优分类超平面。

如下图所示，中间的红色线条是 Optimal Hyper Plane ，另外两条线到红线的距离都是等于[https://github.com/julycoding/The-Art-Of-Programming-By-July/raw/master/ebook/images/svm/1.2-7.jpeg](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/svm/1.2-7.jpeg)的([https://github.com/julycoding/The-Art-Of-Programming-By-July/raw/master/ebook/images/svm/1.2-7.jpeg](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/svm/1.2-7.jpeg)便是上文所定义的geometrical margin，当令[https://github.com/julycoding/The-Art-Of-Programming-By-July/raw/master/ebook/images/svm/1.2-7.jpeg](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/svm/1.2-7.jpeg)=1时，[https://github.com/julycoding/The-Art-Of-Programming-By-July/raw/master/ebook/images/svm/1.2-7.jpeg](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/svm/1.2-7.jpeg)便为1/||w||，而我们上面得到的目标函数便是在相应的约束条件下，要最大化这个1/||w||值)：

[](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/svm/1.4-7.png)

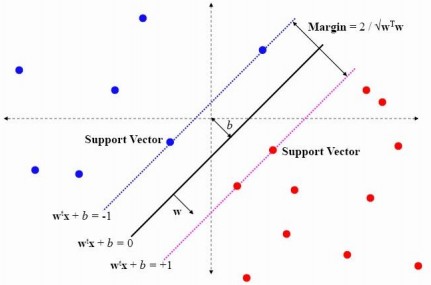
### 1.5、到底什么是Support Vector

通过上节1.4节最后一张图：

[](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/svm/1.4-7.png)

我们可以看到两个支撑着中间的 gap 的超平面，到中间的纯红线separating hyper plane 的距离相等，即我们所能得到的最大的 geometrical margin，而“支撑”这两个超平面的必定会有一些点，而这些“支撑”的点便叫做支持向量Support Vector。

换言之，Support Vector便是下图中那蓝色虚线和粉红色虚线上的点：

[](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/svm/1.5-1.jpeg)

很显然，由于这些 supporting vector 刚好在边界上，所以它们满足[https://github.com/julycoding/The-Art-Of-Programming-By-July/raw/master/ebook/images/svm/1.5-2.jpeg](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/svm/1.5-2.jpeg)，而对于所有不是支持向量的点，也就是在“阵地后方”的点，则显然有[https://github.com/julycoding/The-Art-Of-Programming-By-July/raw/master/ebook/images/svm/1.5-3.jpeg](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/svm/1.5-3.jpeg)。

**第二层、深入SVM**

### 2.1、从线性可分到线性不可分

2.1.1、从原始问题到对偶问题的求解

根据我们之前得到的目标函数（subject to导出的则是约束条件）：

[https://github.com/julycoding/The-Art-Of-Programming-By-July/raw/master/ebook/images/svm/1.4-6.jpg](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/svm/1.4-6.jpg)

由于求[https://github.com/julycoding/The-Art-Of-Programming-By-July/raw/master/ebook/images/svm/2.1.1-1.png](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/svm/2.1.1-1.png)的最大值相当于求[https://github.com/julycoding/The-Art-Of-Programming-By-July/raw/master/ebook/images/svm/2.1.1-2.png](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/svm/2.1.1-2.png)的最小值，所以上述目标函数等价于：

[https://github.com/julycoding/The-Art-Of-Programming-By-July/raw/master/ebook/images/svm/2.1.1-3.jpg](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/svm/2.1.1-3.jpg)

这样，我们的问题成为了一个凸优化问题，因为现在的目标函数是二次的，约束条件是线性的，所以它是一个凸二次规划问题。这个问题可以用任何现成的 [QP (Quadratic Programming)](http://en.wikipedia.org/wiki/Quadratic_programming) 的优化包进行求解，一言以蔽之：在一定的约束条件下，目标最优，损失最小。

进一步，虽然这个问题确实是一个标准的 QP 问题，但由于它的特殊结构，我们可以通过 [Lagrange Duality](http://en.wikipedia.org/wiki/Lagrange_duality#The_strong_Lagrangian_principle:_Lagrange_duality) 变换到对偶变量 (dual variable) 的优化问题，这样便可以找到一种更加有效的方法来进行求解，而且通常情况下这种方法比直接使用通用的 QP 优化包进行优化要高效得多。

换言之，除了用解决QP问题的常规方法之外，还可以通过求解对偶问题得到最优解，这就是线性可分条件下支持向量机的对偶算法，这样做的优点在于：一者对偶问题往往更容易求解；二者可以自然的引入核函数，进而推广到非线性分类问题。

那什么是Lagrange duality？简单地来说，通过给每一个约束条件加上一个 Lagrange multiplier(拉格朗日乘值)，即引入拉格朗日乘子[https://github.com/julycoding/The-Art-Of-Programming-By-July/raw/master/ebook/images/svm/2.1.1-4.jpeg](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/svm/2.1.1-4.jpeg)，如此我们便可以通过拉格朗日函数将约束条件融和到目标函数里去(也就是说把条件融合到一个函数里头，现在只用一个函数表达式便能清楚的表达出我们的问题)：

[](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/svm/2.1.1-5.jpg)

然后我们令

[https://github.com/julycoding/The-Art-Of-Programming-By-July/raw/master/ebook/images/svm/2.1.1-6.jpg](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/svm/2.1.1-6.jpg)

容易验证：

当某个约束条件不满足时，例如[https://github.com/julycoding/The-Art-Of-Programming-By-July/raw/master/ebook/images/svm/2.1.1-7.jpeg](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/svm/2.1.1-7.jpeg)，那么我们显然有[https://github.com/julycoding/The-Art-Of-Programming-By-July/raw/master/ebook/images/svm/2.1.1-8.jpeg](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/svm/2.1.1-8.jpeg)（只要令[https://github.com/julycoding/The-Art-Of-Programming-By-July/raw/master/ebook/images/svm/2.1.1-9.jpeg](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/svm/2.1.1-9.jpeg)即可）。

而当所有约束条件都满足时，则有[https://github.com/julycoding/The-Art-Of-Programming-By-July/raw/master/ebook/images/svm/2.1.1-10.jpeg](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/svm/2.1.1-10.jpeg)，亦即我们最初要最小化的量[https://github.com/julycoding/The-Art-Of-Programming-By-July/raw/master/ebook/images/svm/2.1.1-17.jpg](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/svm/2.1.1-17.jpg)。

因此，在要求约束条件得到满足的情况下最小化[https://github.com/julycoding/The-Art-Of-Programming-By-July/raw/master/ebook/images/svm/2.1.1-11.jpeg](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/svm/2.1.1-11.jpeg)，实际上等价于直接最小化[https://github.com/julycoding/The-Art-Of-Programming-By-July/raw/master/ebook/images/svm/2.1.1-12.jpeg](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/svm/2.1.1-12.jpeg)（当然，这里也有约束条件，就是≥0,i=1,…,n） ，因为如果约束条件没有得到满足，[https://github.com/julycoding/The-Art-Of-Programming-By-July/raw/master/ebook/images/svm/2.1.1-11.jpeg](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/svm/2.1.1-11.jpeg)会等于无穷大，自然不会是我们所要求的最小值。

具体写出来，我们现在的目标函数变成了：

[https://github.com/julycoding/The-Art-Of-Programming-By-July/raw/master/ebook/images/svm/2.1.1-13.jpg](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/svm/2.1.1-13.jpg)

这里用[https://github.com/julycoding/The-Art-Of-Programming-By-July/raw/master/ebook/images/svm/2.1.1-14.jpeg](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/svm/2.1.1-14.jpeg)表示这个问题的最优值，这个问题和我们最初的问题是等价的。不过，现在我们来把最小和最大的位置交换一下：

[https://github.com/julycoding/The-Art-Of-Programming-By-July/raw/master/ebook/images/svm/2.1.1-15.jpg](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/svm/2.1.1-15.jpg)

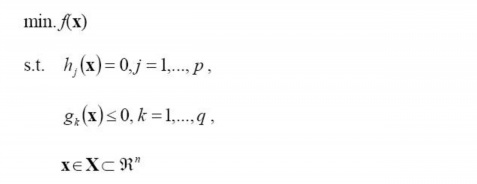
当然，交换以后的问题不再等价于原问题，这个新问题的最优值用[https://github.com/julycoding/The-Art-Of-Programming-By-July/raw/master/ebook/images/svm/2.1.1-16.jpeg](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/svm/2.1.1-16.jpeg)来表示。并且，我们有[https://github.com/julycoding/The-Art-Of-Programming-By-July/raw/master/ebook/images/svm/2.1.1-16.jpeg](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/svm/2.1.1-16.jpeg)≤[https://github.com/julycoding/The-Art-Of-Programming-By-July/raw/master/ebook/images/svm/2.1.1-14.jpeg](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/svm/2.1.1-14.jpeg) ，这在直观上也不难理解，最大值中最小的一个总也比最小值中最大的一个要大。总之，第二个问题的最优值[https://github.com/julycoding/The-Art-Of-Programming-By-July/raw/master/ebook/images/svm/2.1.1-16.jpeg](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/svm/2.1.1-16.jpeg)在这里提供了一个第一个问题的最优值[https://github.com/julycoding/The-Art-Of-Programming-By-July/raw/master/ebook/images/svm/2.1.1-14.jpeg](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/svm/2.1.1-14.jpeg)的一个下界，在满足某些条件的情况下，这两者相等，这个时候我们就可以通过求解第二个问题来间接地求解第一个问题。

也就是说，下面我们可以先求L 对w、b的极小，再求L 对[https://github.com/julycoding/The-Art-Of-Programming-By-July/raw/master/ebook/images/svm/2.1.1-4.jpeg](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/svm/2.1.1-4.jpeg)的极大。而且，之所以从minmax的原始问题[https://github.com/julycoding/The-Art-Of-Programming-By-July/raw/master/ebook/images/svm/2.1.1-14.jpeg](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/svm/2.1.1-14.jpeg)，转化为maxmin的对偶问题[https://github.com/julycoding/The-Art-Of-Programming-By-July/raw/master/ebook/images/svm/2.1.1-16.jpeg](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/svm/2.1.1-16.jpeg)，一者因为[https://github.com/julycoding/The-Art-Of-Programming-By-July/raw/master/ebook/images/svm/2.1.1-16.jpeg](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/svm/2.1.1-16.jpeg)是[https://github.com/julycoding/The-Art-Of-Programming-By-July/raw/master/ebook/images/svm/2.1.1-14.jpeg](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/svm/2.1.1-14.jpeg)的近似解，二者，转化为对偶问题后，更容易求解。

2.1.2、KKT条件

与此同时，上段说“在满足某些条件的情况下”，这所谓的“满足某些条件”就是要满足KKT条件。那KKT条件的表现形式是什么呢？

据维基百科：[KKT 条件](http://en.wikipedia.org/wiki/Karush%E2%80%93Kuhn%E2%80%93Tucker_conditions)的介绍，一般地，一个最优化数学模型能够表示成下列标准形式：

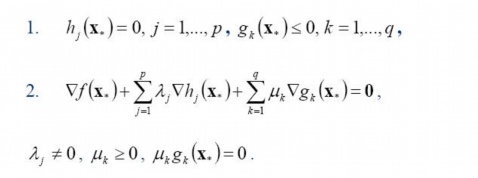
[](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/svm/2.1.2-1.jpg)

其中，f(x)是需要最小化的函数，h(x)是等式约束，g(x)是不等式约束，p和q分别为等式约束和不等式约束的数量。同时，我们得明白以下两个定理：

凸优化的概念：[https://github.com/julycoding/The-Art-Of-Programming-By-July/raw/master/ebook/images/svm/2.1.2-2.png](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/svm/2.1.2-2.png)为一凸集， [https://github.com/julycoding/The-Art-Of-Programming-By-July/raw/master/ebook/images/svm/2.1.2-3.png](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/svm/2.1.2-3.png)为一凸函数。凸优化就是要找出一点 [https://github.com/julycoding/The-Art-Of-Programming-By-July/raw/master/ebook/images/svm/2.1.2-4.png](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/svm/2.1.2-4.png)，使得每一 [https://github.com/julycoding/The-Art-Of-Programming-By-July/raw/master/ebook/images/svm/2.1.2-8.png](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/svm/2.1.2-8.png)满足 [https://github.com/julycoding/The-Art-Of-Programming-By-July/raw/master/ebook/images/svm/2.1.2-5.png](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/svm/2.1.2-5.png)。

KKT条件的意义：它是一个非线性规划（Nonlinear Programming）问题能有最优化解法的必要和充分条件。

而KKT条件就是指上面最优化数学模型的标准形式中的最小点 x\* 必须满足下面的条件：

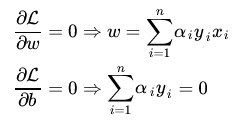
[](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/svm/2.1.2-6.jpg)

经过论证，我们这里的问题是满足 KKT 条件的（首先已经满足Slater condition，再者f和gi也都是可微的，即L对w和b都可导），因此现在我们便转化为求解第二个问题。

也就是说，现在，咱们的原问题通过满足一定的条件，已经转化成了对偶问题。而求解这个对偶学习问题，分为3个步骤，首先要让L(w，b，a) 关于 w 和 b 最小化，然后求对α的极大，最后利用SMO算法求解对偶因子。

2.1.3、对偶问题求解的3个步骤

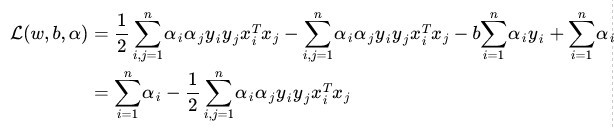
1）、首先固定[https://github.com/julycoding/The-Art-Of-Programming-By-July/raw/master/ebook/images/svm/2.1.1-4.jpeg](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/svm/2.1.1-4.jpeg)，要让 L 关于 w 和 b 最小化，我们分别对w，b求偏导数，即令 ∂L/∂w 和 ∂L/∂b 等于零（对w求导结果的解释请看本文评论下第45楼回复）：

[](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/svm/2.1.2-7.jpeg)

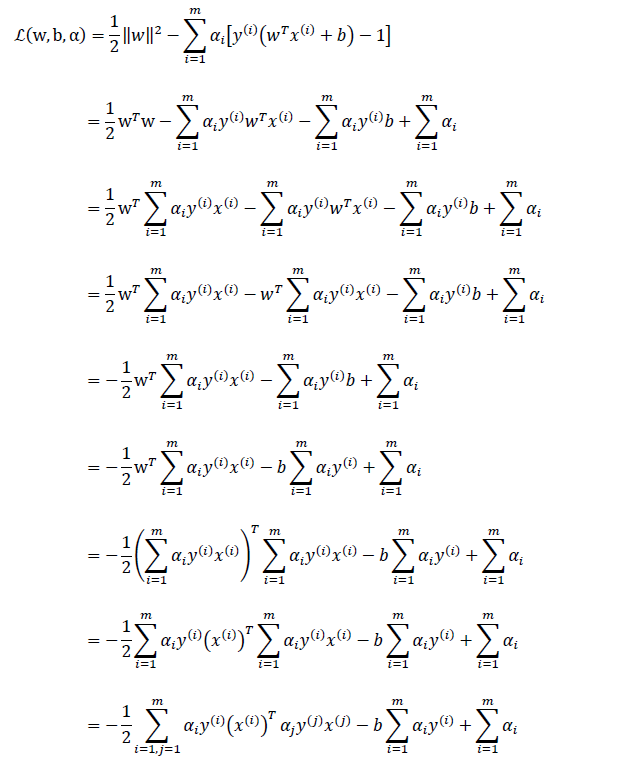
以上结果代回上述的 L：

[](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/svm/2.1.3-1.jpg)

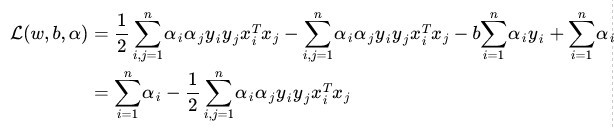
得到：

[](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/svm/2.1.3-2.jpg)

提醒：有读者可能会问上述推导过程如何而来？说实话，其具体推导过程是比较复杂的，如下图所示：

[](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/svm/2.1.3-3.png)

最后，得到：

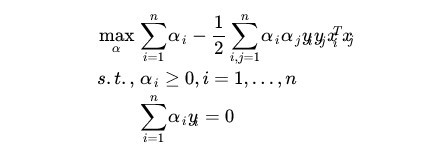
[](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/svm/2.1.3-2.jpg)

如 jerrylead所说：“倒数第4步”推导到“倒数第3步”使用了线性代数的转置运算，由于ai和yi都是实数，因此转置后与自身一样。“倒数第3步”推导到“倒数第2步”使用了(a+b+c+…)(a+b+c+…)=aa+ab+ac+ba+bb+bc+…的乘法运算法则。最后一步是上一步的顺序调整。

L(从上面的最后一个式子，我们可以看出，此时的拉格朗日函数只包含了一个变量，那就是[https://github.com/julycoding/The-Art-Of-Programming-By-July/raw/master/ebook/images/svm/2.1.1-12.jpeg](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/svm/2.1.1-12.jpeg)，然后下文的第2步，求出了[https://github.com/julycoding/The-Art-Of-Programming-By-July/raw/master/ebook/images/svm/2.1.1-12.jpeg](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/svm/2.1.1-12.jpeg)便能求出w，和b，由此可见，上文第1.2节提出来的核心问题：分类函数[https://github.com/julycoding/The-Art-Of-Programming-By-July/raw/master/ebook/images/svm/1.2-2.jpeg](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/svm/1.2-2.jpeg)也就可以轻而易举的求出来了。

2）、求对[https://github.com/julycoding/The-Art-Of-Programming-By-July/raw/master/ebook/images/svm/2.1.1-4.jpeg](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/svm/2.1.1-4.jpeg)的极大，即是关于对偶问题的最优化问题，从上面的式子得到：

(不得不提醒下读者：经过上面第一个步骤的求w和b，得到的拉格朗日函数式子已经没有了变量w，b，只有[https://github.com/julycoding/The-Art-Of-Programming-By-July/raw/master/ebook/images/svm/2.1.1-4.jpeg](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/svm/2.1.1-4.jpeg)，而反过来，求得的将能导出w，b的解，最终得出分离超平面和分类决策函数。为何呢？因为如果求出了[https://github.com/julycoding/The-Art-Of-Programming-By-July/raw/master/ebook/images/svm/2.1.1-12.jpeg](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/svm/2.1.1-12.jpeg)，根据[https://github.com/julycoding/The-Art-Of-Programming-By-July/raw/master/ebook/images/svm/2.1.3-4.jpg](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/svm/2.1.3-4.jpg)，即可求出w。然后通过[https://github.com/julycoding/The-Art-Of-Programming-By-July/raw/master/ebook/images/svm/2.1.3-5.png](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/svm/2.1.3-5.png)，即可求出b )

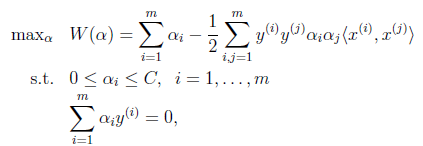
[](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/svm/2.1.3-6.jpg)

3）、如前面所说，这个问题有更加高效的优化算法，即我们常说的SMO算法。

2.1.4、序列最小最优化SMO算法

细心的读者读至上节末尾处，怎么拉格朗日乘子[https://github.com/julycoding/The-Art-Of-Programming-By-July/raw/master/ebook/images/svm/2.1.1-4.jpeg](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/svm/2.1.1-4.jpeg)的值可能依然心存疑惑。实际上，关于[https://github.com/julycoding/The-Art-Of-Programming-By-July/raw/master/ebook/images/svm/2.1.1-4.jpeg](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/svm/2.1.1-4.jpeg)的求解可以用一种快速学习算法即SMO算法，这里先简要介绍下。

OK，当：

[](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/svm/2.1.4-1.png)

要解决的是在参数[https://github.com/julycoding/The-Art-Of-Programming-By-July/raw/master/ebook/images/svm/2.1.4-2.png](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/svm/2.1.4-2.png)上求最大值W的问题，至于[https://github.com/julycoding/The-Art-Of-Programming-By-July/raw/master/ebook/images/svm/2.1.4-3.png](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/svm/2.1.4-3.png)和[https://github.com/julycoding/The-Art-Of-Programming-By-July/raw/master/ebook/images/svm/2.1.4-4.png](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/svm/2.1.4-4.png)都是已知数（其中 是一个参数，用于控制目标函数中两项（“寻找 margin 最大的超平面”和“保证数据点偏差量最小”）之间的权重。和上文最后的式子对比一下，可以看到唯一的区别就是现在 dual variable [https://github.com/julycoding/The-Art-Of-Programming-By-July/raw/master/ebook/images/svm/2.1.1-4.jpeg](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/svm/2.1.1-4.jpeg)多了一个上限 C ，关于C的具体由来请查看下文第2.3节）。

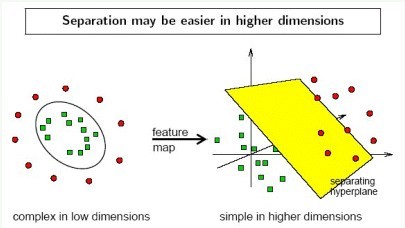
要了解这个SMO算法是如何推导的，请跳到下文第3.5节、SMO算法。

到目前为止，我们的 SVM 还比较弱，只能处理线性的情况，下面我们将引入核函数，进而推广到非线性分类问题。

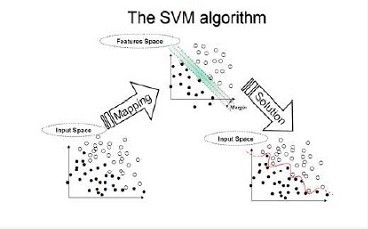
**2.2、核函数Kernel**

2.2.1、特征空间的隐式映射：核函数

在线性不可分的情况下，支持向量机通过某种事先选择的非线性映射(核函数)将输入变量映射到一个高维特征空间，在这个空间中构造最优分类超平面。我们使用SVM进行数据集分类工作的过程首先是同预先选定的一些非线性映射将输入空间映射到高维特征空间(下图很清晰的表达了通过映射到高维特征空间，而把平面上本身不好分的非线性数据分了开来)：

[](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/svm/2.2.1-1.jpg)

使得在高维属性空间中有可能最训练数据实现超平面的分割，避免了在原输入空间中进行非线性曲面分割计算，且在处理高维输入空间的分类时，这种方法尤其有效，其工作原理如下图所示：

[](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/svm/2.2.1-2.jpg)

而在我们遇到核函数之前，如果用原始的方法，那么在用线性学习器学习一个非线性关系，需要选择一个非线性特征集，并且将数据写成新的表达形式，这等价于应用一个固定的非线性映射，将数据映射到特征空间，在特征空间中使用线性学习器，因此，考虑的假设集是这种类型的函数：

[https://github.com/julycoding/The-Art-Of-Programming-By-July/raw/master/ebook/images/svm/2.2.1-3.jpg](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/svm/2.2.1-3.jpg)

这里ϕ：X->F是从输入空间到某个特征空间的映射，这意味着建立非线性学习器分为两步：

首先使用一个非线性映射将数据变换到一个特征空间F，

然后在特征空间使用线性学习器分类。

这意味着假设可以表达为训练点的线性组合，因此决策规则可以用测试点和训练点的内积来表示：

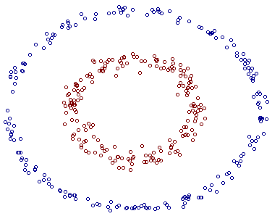
[https://github.com/julycoding/The-Art-Of-Programming-By-July/raw/master/ebook/images/svm/2.2.1-4.jpg](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/svm/2.2.1-4.jpg)

如果有一种方式可以在特征空间中直接计算内积〈φ(xi · φ(x)〉，就像在原始输入点的函数中一样，就有可能将两个步骤融合到一起建立一个非线性的学习器，这样直接计算法的方法称为核函数方法，于是，核函数便横空出世了。

定义：核是一个函数K，对所有x，z(-X，满足[https://github.com/julycoding/The-Art-Of-Programming-By-July/raw/master/ebook/images/svm/2.2.1-5.jpg](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/svm/2.2.1-5.jpg)，这里φ是从X到内积特征空间F的映射。

2.2.2、核函数：如何处理非线性数据

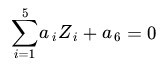
我们已经知道，如果是线性方法，所以对非线性的数据就没有办法处理。举个例子来说，则是如下图所示的两类数据，分别分布为两个圆圈的形状，这样的数据本身就是线性不可分的，此时咱们该如何把这两类数据分开呢？

[](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/svm/2.2.2-1.png)

此时，一个理想的分界应该是一个“圆圈”而不是一条线（超平面）。如果用 X1 和 X2 来表示这个二维平面的两个坐标的话，我们知道一条二次曲线（圆圈是二次曲线的一种特殊情况）的方程可以写作这样的形式：

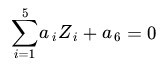
[https://github.com/julycoding/The-Art-Of-Programming-By-July/raw/master/ebook/images/svm/2.2.2-2.jpeg](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/svm/2.2.2-2.jpeg)

如果我们构造另外一个五维的空间，其中五个坐标的值分别为 Z1=X1, Z2=X21, Z3=X2, Z4=X22, Z5=X1X2，那么显然，上面的方程在新的坐标系下可以写作：

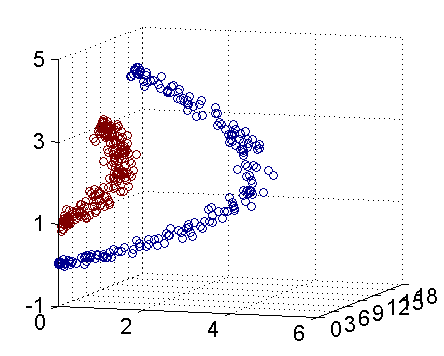
[](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/svm/2.2.2-3.jpeg)

关于新的坐标 Z ，这正是一个 hyper plane 的方程！也就是说，如果我们做一个映射 ϕ:R2→R5 ，将 X 按照上面的规则映射为 Z ，那么在新的空间中原来的数据将变成线性可分的，从而使用之前我们推导的线性分类算法就可以进行处理了。这正是 Kernel 方法处理非线性问题的基本思想。

再进一步描述 Kernel 的细节之前，不妨再来看看这个例子映射过后的直观例子。具体来说，我这里的超平面实际的方程是这个样子（圆心在 X2 轴上的一个正圆）：

[](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/svm/2.2.2-3.jpeg)

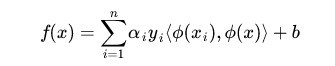
因此我只需要把它映射到 Z1=X21, Z2=X22, Z3=X2 这样一个三维空间中即可，下图即是映射之后的结果，将坐标轴经过适当的旋转，就可以很明显地看出，数据是可以通过一个平面来分开的：

[](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/svm/2.2.2-4.gif)

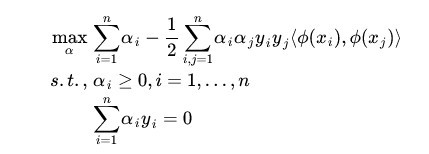
回忆一下，我们上一次2.1节中得到的最终的分类函数是这样的：

[](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/svm/2.2.2-5.jpg)

映射过后的空间是：

[](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/svm/2.2.2-6.jpg)

而其中的 α 也是通过求解如下 dual 问题而得到的：

[](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/svm/2.2.2-7.jpg)

这样一来问题就解决了吗？其实稍想一下就会发现有问题：在最初的例子里，我们对一个二维空间做映射，选择的新空间是原始空间的所有一阶和二阶的组合，得到了五个维度；如果原始空间是三维，那么我们会得到 19 维的新空间，这个数目是呈爆炸性增长的，这给 ϕ(⋅) 的计算带来了非常大的困难，而且如果遇到无穷维的情况，就根本无从计算了。所以就需要 Kernel 出马了。

还是从最开始的简单例子出发，设两个向量[https://github.com/julycoding/The-Art-Of-Programming-By-July/raw/master/ebook/images/svm/2.2.2-8.jpg](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/svm/2.2.2-8.jpg)和[https://github.com/julycoding/The-Art-Of-Programming-By-July/raw/master/ebook/images/svm/2.2.2-9.jpg](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/svm/2.2.2-9.jpg)，而ϕ(⋅)即是到前面2.2.1节说的五维空间的映射，因此映射过后的内积为：

[https://github.com/julycoding/The-Art-Of-Programming-By-July/raw/master/ebook/images/svm/2.2.2-10.jpg](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/svm/2.2.2-10.jpg)

（公式说明：上面的这两个推导过程中，所说的前面的五维空间的映射，这里说的前面便是文中2.2.1节的所述的映射方式，仔细看下2.2.1节的映射规则，再看那第一个推导，其实就是计算x1，x2各自的内积，然后相乘相加即可，第二个推导则是直接平方，去掉括号，也很容易推出来）

另外，我们又注意到：

[https://github.com/julycoding/The-Art-Of-Programming-By-July/raw/master/ebook/images/svm/2.2.2-11.jpg](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/svm/2.2.2-11.jpg)

二者有很多相似的地方，实际上，我们只要把某几个维度线性缩放一下，然后再加上一个常数维度，具体来说，上面这个式子的计算结果实际上和映射

[https://github.com/julycoding/The-Art-Of-Programming-By-July/raw/master/ebook/images/svm/2.2.2-12.jpg](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/svm/2.2.2-12.jpg)

之后的内积[https://github.com/julycoding/The-Art-Of-Programming-By-July/raw/master/ebook/images/svm/2.2.2-13.jpg](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/svm/2.2.2-13.jpg)的结果是相等的，那么区别在于什么地方呢？

一个是映射到高维空间中，然后再根据内积的公式进行计算； 而另一个则直接在原来的低维空间中进行计算，而不需要显式地写出映射后的结果。 （公式说明：上面之中，最后的两个式子，第一个算式，是带内积的完全平方式，可以拆开，然后，通过凑一个得到，第二个算式，也是根据第一个算式凑出来的）

回忆刚才提到的映射的维度爆炸，在前一种方法已经无法计算的情况下，后一种方法却依旧能从容处理，甚至是无穷维度的情况也没有问题。

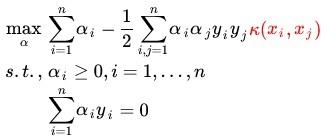
我们把这里的计算两个向量在隐式映射过后的空间中的内积的函数叫做核函数 (Kernel Function) ，例如，在刚才的例子中，我们的核函数为：

[https://github.com/julycoding/The-Art-Of-Programming-By-July/raw/master/ebook/images/svm/2.2.2-14.jpg](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/svm/2.2.2-14.jpg)

核函数能简化映射空间中的内积运算——刚好“碰巧”的是，在我们的 SVM 里需要计算的地方数据向量总是以内积的形式出现的。对比刚才我们上面写出来的式子，现在我们的分类函数为：

[https://github.com/julycoding/The-Art-Of-Programming-By-July/raw/master/ebook/images/svm/2.2.2-15.jpg](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/svm/2.2.2-15.jpg)

其中 由如下 dual 问题计算而得：

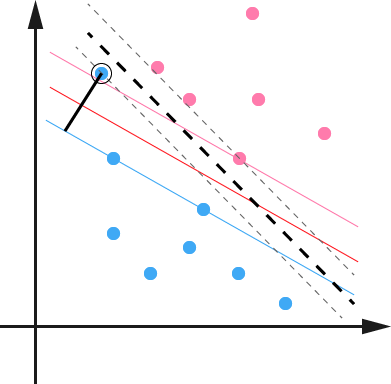
[](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/svm/2.2.2-16.jpg)

这样一来计算的问题就算解决了，避开了直接在高维空间中进行计算，而结果却是等价的。

**2.3、使用松弛变量处理 outliers 方法**

在本文第一节最开始讨论支持向量机的时候，我们就假定，数据是线性可分的，亦即我们可以找到一个可行的超平面将数据完全分开。后来为了处理非线性数据，在上文2.2节使用 Kernel 方法对原来的线性 SVM 进行了推广，使得非线性的的情况也能处理。虽然通过映射 ϕ(⋅) 将原始数据映射到高维空间之后，能够线性分隔的概率大大增加，但是对于某些情况还是很难处理。

例如可能并不是因为数据本身是非线性结构的，而只是因为数据有噪音。对于这种偏离正常位置很远的数据点，我们称之为 outlier ，在我们原来的 SVM 模型里，outlier 的存在有可能造成很大的影响，因为超平面本身就是只有少数几个 support vector 组成的，如果这些 support vector 里又存在 outlier 的话，其影响就很大了。例如下图：

[](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/svm/2.3-1.png)

用黑圈圈起来的那个蓝点是一个 outlier ，它偏离了自己原本所应该在的那个半空间，如果直接忽略掉它的话，原来的分隔超平面还是挺好的，但是由于这个 outlier 的出现，导致分隔超平面不得不被挤歪了，变成途中黑色虚线所示（这只是一个示意图，并没有严格计算精确坐标），同时 margin 也相应变小了。当然，更严重的情况是，如果这个 outlier 再往右上移动一些距离的话，我们将无法构造出能将数据分开的超平面来。

为了处理这种情况，SVM 允许数据点在一定程度上偏离一下超平面。例如上图中，黑色实线所对应的距离，就是该 outlier 偏离的距离，如果把它移动回来，就刚好落在原来的超平面上，而不会使得超平面发生变形了。

我们原来的约束条件为：

[https://github.com/julycoding/The-Art-Of-Programming-By-July/raw/master/ebook/images/svm/2.3-3.jpg](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/svm/2.3-3.jpg)

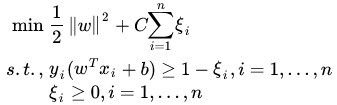
现在考虑到outlier问题，约束条件变成了：

[https://github.com/julycoding/The-Art-Of-Programming-By-July/raw/master/ebook/images/svm/2.3-4.jpg](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/svm/2.3-4.jpg)

其中[https://github.com/julycoding/The-Art-Of-Programming-By-July/raw/master/ebook/images/svm/2.3-5.jpg](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/svm/2.3-5.jpg)称为松弛变量 (slack variable) ，对应数据点[https://github.com/julycoding/The-Art-Of-Programming-By-July/raw/master/ebook/images/svm/2.3-6.jpg](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/svm/2.3-6.jpg)允许偏离的 functional margin 的量。当然，如果我们运行[https://github.com/julycoding/The-Art-Of-Programming-By-July/raw/master/ebook/images/svm/2.3-7.jpg](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/svm/2.3-7.jpg)任意大的话，那任意的超平面都是符合条件的了。所以，我们在原来的目标函数后面加上一项，使得这些[https://github.com/julycoding/The-Art-Of-Programming-By-July/raw/master/ebook/images/svm/2.3-7.jpg](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/svm/2.3-7.jpg)的总和也要最小：

[https://github.com/julycoding/The-Art-Of-Programming-By-July/raw/master/ebook/images/svm/2.3-8.jpg](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/svm/2.3-8.jpg)

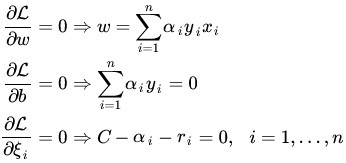
其中 C 是一个参数，用于控制目标函数中两项（“寻找 margin 最大的超平面”和“保证数据点偏差量最小”）之间的权重。注意，其中 [https://github.com/julycoding/The-Art-Of-Programming-By-July/raw/master/ebook/images/svm/2.3-9.png](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/svm/2.3-9.png)是需要优化的变量（之一），而 C 是一个事先确定好的常量。完整地写出来是这个样子：

[](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/svm/2.3-10.jpg)

用之前的方法将限制或约束条件加入到目标函数中，得到新的拉格朗日函数，如下所示：

[https://github.com/julycoding/The-Art-Of-Programming-By-July/raw/master/ebook/images/svm/2.3-11.jpg](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/svm/2.3-11.jpg)

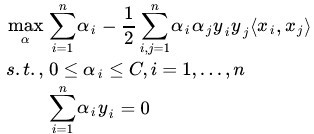
分析方法和前面一样，转换为另一个问题之后，我们先让[https://github.com/julycoding/The-Art-Of-Programming-By-July/raw/master/ebook/images/svm/2.3-20.png](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/svm/2.3-20.png)针对w、b和[https://github.com/julycoding/The-Art-Of-Programming-By-July/raw/master/ebook/images/svm/2.3-7.jpg](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/svm/2.3-7.jpg)最小化：

[](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/svm/2.3-12.jpg)

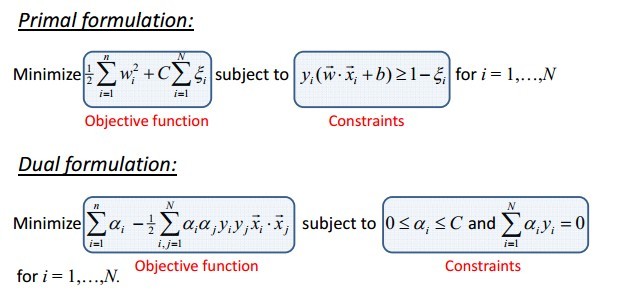
将 w 带回 [https://github.com/julycoding/The-Art-Of-Programming-By-July/raw/master/ebook/images/svm/2.3-20.png](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/svm/2.3-20.png)并化简，得到和原来一样的目标函数：

[https://github.com/julycoding/The-Art-Of-Programming-By-July/raw/master/ebook/images/svm/2.3-13.jpg](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/svm/2.3-13.jpg)

不过，由于我们得到[https://github.com/julycoding/The-Art-Of-Programming-By-July/raw/master/ebook/images/svm/2.3-14.jpg](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/svm/2.3-14.jpg)而又有ri >= 0（作为 Lagrange multiplier 的条件），因此有[https://github.com/julycoding/The-Art-Of-Programming-By-July/raw/master/ebook/images/svm/2.3-15.jpg](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/svm/2.3-15.jpg)，所以整个 dual 问题现在写作：

[](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/svm/2.3-16.jpg)

把前后的结果对比一下（错误修正：图中的Dual formulation中的Minimize应为maxmize）：

[](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/svm/2.3-17.jpeg)

可以看到唯一的区别就是现在 dual variable [https://github.com/julycoding/The-Art-Of-Programming-By-July/raw/master/ebook/images/svm/2.1.1-4.jpeg](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/svm/2.1.1-4.jpeg)多了一个上限 C 。而 Kernel 化的非线性形式也是一样的，只要把[https://github.com/julycoding/The-Art-Of-Programming-By-July/raw/master/ebook/images/svm/2.3-18.jpg](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/svm/2.3-18.jpg)换成[https://github.com/julycoding/The-Art-Of-Programming-By-July/raw/master/ebook/images/svm/2.3-19.jpg](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/svm/2.3-19.jpg)即可。这样一来，一个完整的，可以处理线性和非线性并能容忍噪音和 outliers 的支持向量机才终于介绍完毕了。

行文至此，可以做个小结，不准确的说，SVM它本质上即是一个分类方法，用w^T+b定义分类函数，于是求w、b，为寻最大间隔，引出1/2||w||^2，继而引入拉格朗日因子，化为对拉格朗日乘子a的求解（求解过程中会涉及到一系列最优化或凸二次规划等问题），如此，求w.b与求a等价，而a的求解可以用一种快速学习算法SMO，至于核函数，是为处理非线性情况，若直接映射到高维计算恐维度爆炸，故在低维计算，等效高维表现。

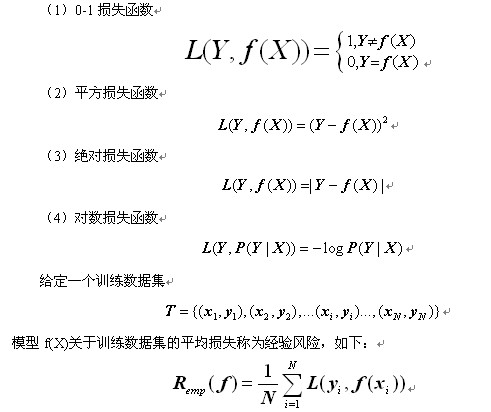
**第三层、扩展SVM**

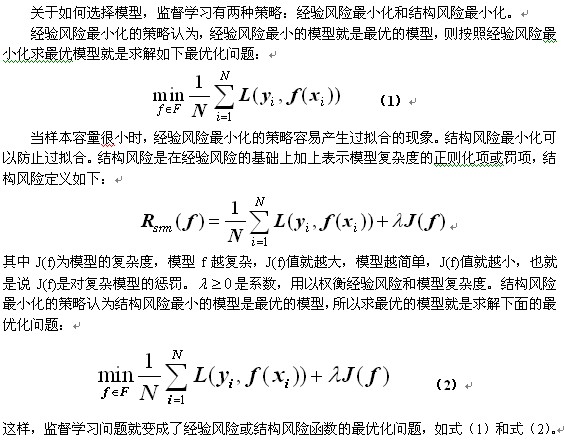
**3.1、损失函数**

在本文1.0节有这么一句话“支持向量机(SVM)是90年代中期发展起来的基于统计学习理论的一种机器学习方法，通过寻求结构化风险最小来提高学习机泛化能力，实现经验风险和置信范围的最小化，从而达到在统计样本量较少的情况下，亦能获得良好统计规律的目的。”但初次看到的读者可能并不了解什么是结构化风险，什么又是经验风险。要了解这两个所谓的“风险”，还得又从监督学习说起。

监督学习实际上就是一个经验风险或者结构风险函数的最优化问题。风险函数度量平均意义下模型预测的好坏，模型每一次预测的好坏用损失函数来度量。它从假设空间F中选择模型f作为决策函数，对于给定的输入X，由f(X)给出相应的输出Y，这个输出的预测值f(X)与真实值Y可能一致也可能不一致，用一个损失函数来度量预测错误的程度。损失函数记为L(Y, f(X))。

常用的损失函数有以下几种（基本引用自《统计学习方法》）：

[](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/svm/3.3-1.jpeg)

[](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/svm/3.3-2.jpeg)

如此，SVM有第二种理解，即最优化+损失最小，或如@夏粉\_百度所说“可从损失函数和优化算法角度看SVM，boosting，LR等算法，可能会有不同收获”。

关于损失函数，还可以看看张潼的这篇《Statistical behavior and consistency of classification methods based on convex risk minimization》。各种算法中常用的损失函数基本都具有fisher一致性，优化这些损失函数得到的分类器可以看作是后验概率的“代理”。

此外，他还有另外一篇论文《Statistical analysis of some multi-category large margin classification methods》，在多分类情况下margin loss的分析，这两篇对Boosting和SVM使用的损失函数分析的很透彻。

**3.2、SMO算法**

在上文2.1.2节中，我们提到了求解对偶问题的序列最小最优化SMO算法，但并未提到其具体解法。

事实上，SMO算法是由Microsoft Research的John C. Platt在1998年发表的一篇[论文](http://research.microsoft.com/en-us/um/people/jplatt/smoTR.pdf)《Sequential Minimal Optimization A Fast Algorithm for Training Support Vector Machines》中提出，它很快成为最快的二次规划优化算法，特别针对线性SVM和数据稀疏时性能更优。

接下来，咱们便参考John C. Platt的[这篇](http://research.microsoft.com/en-us/um/people/jplatt/smoTR.pdf)文章来看看SMO的解法是怎样的。

3.2.1、SMO算法的解法

咱们首先来定义特征到结果的输出函数为

[https://github.com/julycoding/The-Art-Of-Programming-By-July/raw/master/ebook/images/svm/1.2-4.jpeg](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/svm/1.2-4.jpeg)

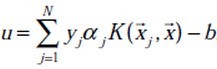
再三强调，这个u与我们之前定义的[https://github.com/julycoding/The-Art-Of-Programming-By-July/raw/master/ebook/images/svm/1.2-2.jpeg](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/svm/1.2-2.jpeg)实质是一样的。

接着，咱们重新定义咱们原始的优化问题，权当重新回顾，如下：

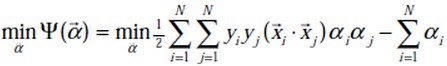
[https://github.com/julycoding/The-Art-Of-Programming-By-July/raw/master/ebook/images/svm/3.5.1-1.jpeg](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/svm/3.5.1-1.jpeg)

求导得到：

[https://github.com/julycoding/The-Art-Of-Programming-By-July/raw/master/ebook/images/svm/3.5.1-2.jpeg](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/svm/3.5.1-2.jpeg)

代入[https://github.com/julycoding/The-Art-Of-Programming-By-July/raw/master/ebook/images/svm/1.2-4.jpeg](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/svm/1.2-4.jpeg)中，可得[](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/svm/3.5.1-3.jpeg)。

引入对偶因子后，得：

[](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/svm/3.5.1-24.jpeg)

s.t：[https://github.com/julycoding/The-Art-Of-Programming-By-July/raw/master/ebook/images/svm/3.5.1-4.png](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/svm/3.5.1-4.png)且[https://github.com/julycoding/The-Art-Of-Programming-By-July/raw/master/ebook/images/svm/3.5.1-5.png](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/svm/3.5.1-5.png)

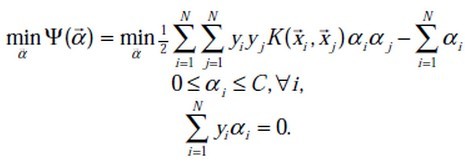
注：这里得到的min函数与我们之前的max函数实质也是一样，因为把符号变下，即有min转化为max的问题，且yi也与之前的[https://github.com/julycoding/The-Art-Of-Programming-By-July/raw/master/ebook/images/svm/3.5.1-6.png](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/svm/3.5.1-6.png)等价，yj亦如此。

经过加入松弛变量后，模型修改为：

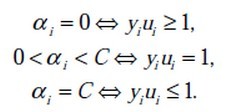
[https://github.com/julycoding/The-Art-Of-Programming-By-July/raw/master/ebook/images/svm/3.5.1-7.jpeg](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/svm/3.5.1-7.jpeg)

[https://github.com/julycoding/The-Art-Of-Programming-By-July/raw/master/ebook/images/svm/3.5.1-8.jpeg](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/svm/3.5.1-8.jpeg)

从而最终我们的问题变为：

[](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/svm/3.5.1-9.jpeg)

继而，根据KKT条件可以得出其中取值的意义为：

[](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/svm/3.5.1-10.jpeg)

这里的还是拉格朗日乘子(问题通过拉格朗日乘法数来求解)

对于第1种情况，表明[https://github.com/julycoding/The-Art-Of-Programming-By-July/raw/master/ebook/images/svm/2.1.1-12.jpeg](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/svm/2.1.1-12.jpeg)是正常分类，在边界内部（我们知道正确分类的点yi\*f(xi)>=0）；

对于第2种情况，表明了[https://github.com/julycoding/The-Art-Of-Programming-By-July/raw/master/ebook/images/svm/2.1.1-12.jpeg](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/svm/2.1.1-12.jpeg)是支持向量，在边界上；

对于第3种情况，表明了[https://github.com/julycoding/The-Art-Of-Programming-By-July/raw/master/ebook/images/svm/2.1.1-12.jpeg](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/svm/2.1.1-12.jpeg)是在两条边界之间；

而最优解需要满足KKT条件，即上述3个条件都得满足，以下几种情况出现将会出现不满足：

[https://github.com/julycoding/The-Art-Of-Programming-By-July/raw/master/ebook/images/svm/3.5.1-11.jpeg](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/svm/3.5.1-11.jpeg)<=1但是[https://github.com/julycoding/The-Art-Of-Programming-By-July/raw/master/ebook/images/svm/2.1.1-12.jpeg](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/svm/2.1.1-12.jpeg)<C则是不满足的,而原本[https://github.com/julycoding/The-Art-Of-Programming-By-July/raw/master/ebook/images/svm/2.1.1-12.jpeg](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/svm/2.1.1-12.jpeg)=C

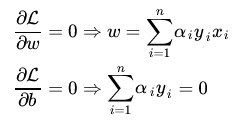
[https://github.com/julycoding/The-Art-Of-Programming-By-July/raw/master/ebook/images/svm/3.5.1-11.jpeg](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/svm/3.5.1-11.jpeg)>=1但是[https://github.com/julycoding/The-Art-Of-Programming-By-July/raw/master/ebook/images/svm/2.1.1-12.jpeg](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/svm/2.1.1-12.jpeg)>0则是不满足的而原本[https://github.com/julycoding/The-Art-Of-Programming-By-July/raw/master/ebook/images/svm/2.1.1-12.jpeg](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/svm/2.1.1-12.jpeg)=0

[https://github.com/julycoding/The-Art-Of-Programming-By-July/raw/master/ebook/images/svm/3.5.1-11.jpeg](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/svm/3.5.1-11.jpeg)=1但是[https://github.com/julycoding/The-Art-Of-Programming-By-July/raw/master/ebook/images/svm/2.1.1-12.jpeg](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/svm/2.1.1-12.jpeg)=0或者[https://github.com/julycoding/The-Art-Of-Programming-By-July/raw/master/ebook/images/svm/2.1.1-12.jpeg](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/svm/2.1.1-12.jpeg)=C则表明不满足的，而原本应该是0<[https://github.com/julycoding/The-Art-Of-Programming-By-July/raw/master/ebook/images/svm/2.1.1-12.jpeg](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/svm/2.1.1-12.jpeg)<C

所以要找出不满足KKT条件的这些ai，并更新这些ai，但这些ai又受到另外一个约束，即

[https://github.com/julycoding/The-Art-Of-Programming-By-July/raw/master/ebook/images/svm/3.5.1-12.png](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/svm/3.5.1-12.png)

注：别忘了2.1.1节中，L对a、b求偏导，得到：

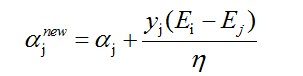
[](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/svm/3.5.1-13.jpeg)

因此，我们通过另一个方法，即同时更新ai和aj，要求满足以下等式：

[https://github.com/julycoding/The-Art-Of-Programming-By-July/raw/master/ebook/images/svm/3.5.1-14.gif](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/svm/3.5.1-14.gif)

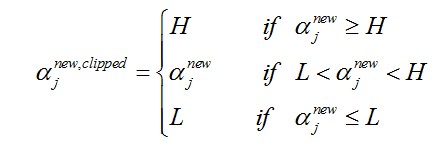
就能保证和为0的约束。

利用yiai+yjaj=常数，消去ai，可得到一个关于单变量aj的一个凸二次规划问题，不考虑其约束0<=aj<=C,可以得其解为：

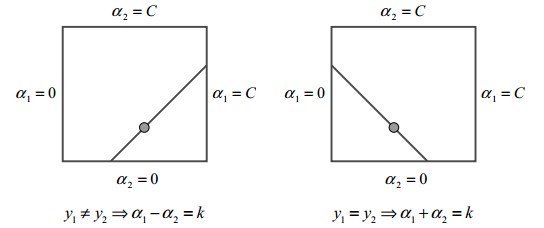
[](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/svm/3.5.1-15.gif)

这里[https://github.com/julycoding/The-Art-Of-Programming-By-July/raw/master/ebook/images/svm/3.5.1-16.jpeg](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/svm/3.5.1-16.jpeg)，，表示旧值。

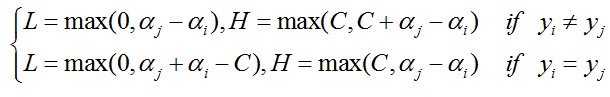
然后考虑约束0<=aj<=C可得到a的解析解为：

[](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/svm/3.5.1-17.gif)

把SMO中对于两个参数求解过程看成线性规划来理解来理解的话，那么下图所表达的便是约束条件：[https://github.com/julycoding/The-Art-Of-Programming-By-July/raw/master/ebook/images/svm/3.5.1-14.gif](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/svm/3.5.1-14.gif)

[](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/svm/3.5.1-18.jpeg)

根据yi和yj同号或异号，可得出两个拉格朗日乘子的上下界分别为：

[](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/svm/3.5.1-19.gif)

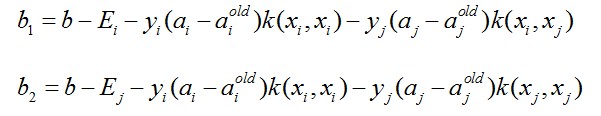
对于[https://github.com/julycoding/The-Art-Of-Programming-By-July/raw/master/ebook/images/svm/3.5.1-20.gif](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/svm/3.5.1-20.gif)。

那么如何求得ai和aj呢？

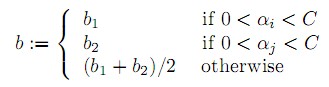
对于ai，即第一个乘子，可以通过刚刚说的那3种不满足KKT的条件来找；

而对于第二个乘子aj可以找满足条件 ：[https://github.com/julycoding/The-Art-Of-Programming-By-July/raw/master/ebook/images/svm/3.5.1-21.gif](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/svm/3.5.1-21.gif)求得。

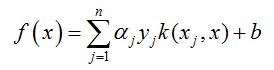
而b的更新则是：

[](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/svm/3.5.1-25.gif)

在满足下述条件：

[](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/svm/3.5.1-22.gif)

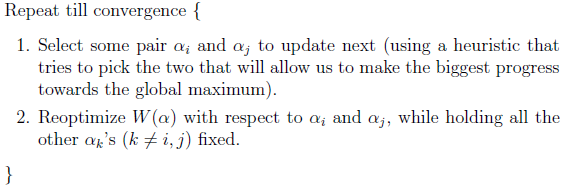
下更新b，且每次更新完两个乘子的优化后，都需要再重新计算b，及对应的Ei值。 最后更新所有ai，y和b，这样模型就出来了，从而即可求出咱们开头提出的分类函数

[](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/svm/3.5.1-23.gif)

此外，这里也有一篇类似的文章，大家可以参考下。

3.2.2、SMO算法的步骤

这样，SMO的主要步骤如下：

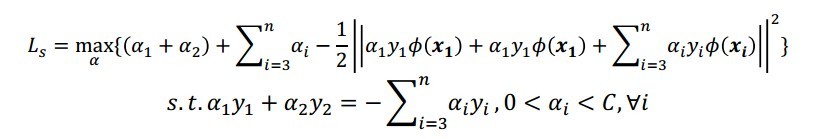
[](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/svm/3.5.2-1.png)

意思是，

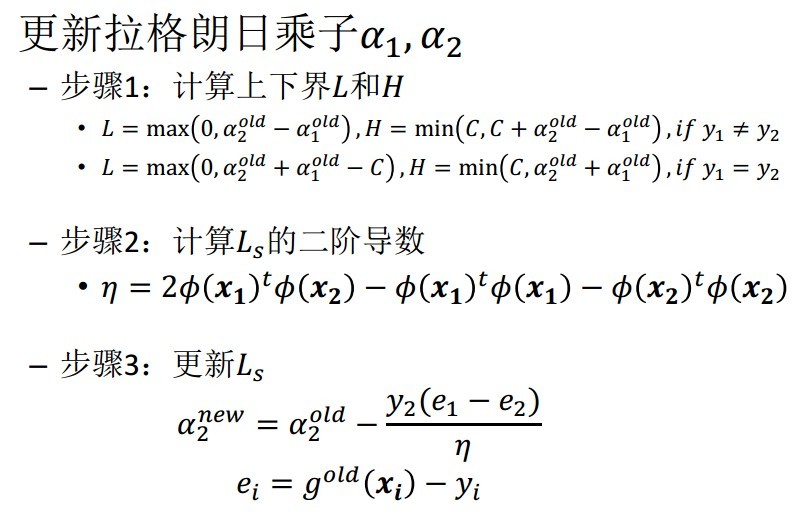
第一步选取一对ai和aj，选取方法使用启发式方法；

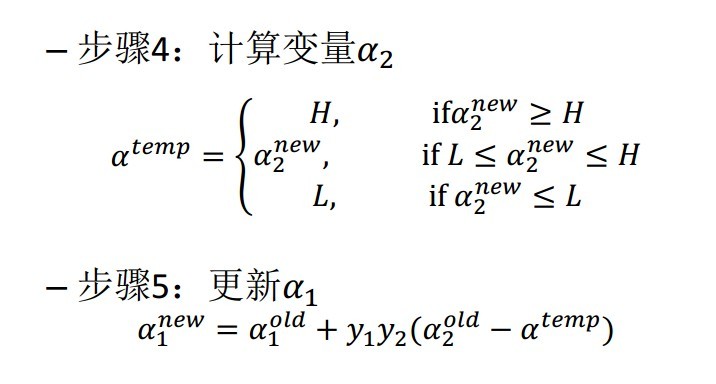
第二步，固定除ai和aj之外的其他参数，确定W极值条件下的ai，由aj表示ai。

假定在某一次迭代中，需要更新，对应的拉格朗日乘子，，那么这个小规模的二次规划问题写为：

[](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/svm/3.5.2-2.jpeg)

那么在每次迭代中，如何更新乘子呢？引用[这里](http://staff.ustc.edu.cn/%7Eketang/PPT/PRLec5.pdf)的两张PPT说明下：

[](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/svm/3.5.2-3.jpeg)

[](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/svm/3.5.2-4.jpeg)

知道了如何更新乘子，那么选取哪些乘子进行更新呢？具体选择方法有以下两个步骤：

步骤1：先“扫描”所有乘子，把第一个违反KKT条件的作为更新对象，令为a2；

步骤2：在所有不违反KKT条件的乘子中，选择使|E1 −E2|最大的a1（注：别忘了，其中[https://github.com/julycoding/The-Art-Of-Programming-By-July/raw/master/ebook/images/svm/3.5.2-5.png](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/svm/3.5.2-5.png)，而[https://github.com/julycoding/The-Art-Of-Programming-By-July/raw/master/ebook/images/svm/1.2-4.jpeg](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/svm/1.2-4.jpeg)，求出来的E代表函数ui对输入xi的预测值与真实输出类标记yi之差）。

值得一提的是，每次更新完两个乘子的优化后，都需要再重新计算b，及对应的Ei值。

与此同时，乘子的选择务必遵循两个原则：

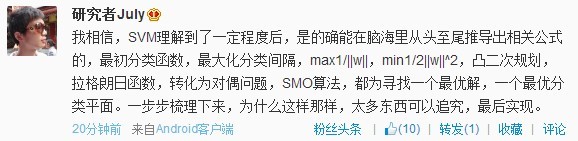
使乘子能满足KKT条件

对一个满足KKT条件的乘子进行更新，应能最大限度增大目标函数的值（类似于[梯度](http://zh.wikipedia.org/zh-cn/%E6%A2%AF%E5%BA%A6)下降）

综上，SMO算法的基本思想是将Vapnik在1982年提出的Chunking方法推到极致，SMO算法每次迭代只选出两个分量ai和aj进行调整，其它分量则保持固定不变，在得到解ai和aj之后，再用ai和aj改进其它分量。与通常的分解算法比较，尽管它可能需要更多的迭代次数，但每次迭代的计算量比较小，所以该算法表现出整理的快速收敛性，且不需要存储核矩阵，也没有矩阵运算。

3.5.3、SMO算法的实现

行文至此，我相信，SVM理解到了一定程度后，是的确能在脑海里从头至尾推导出相关公式的，最初分类函数，最大化分类间隔，max1/||w||，min1/2||w||^2，凸二次规划，拉格朗日函数，转化为对偶问题，SMO算法，都为寻找一个最优解，一个最优分类平面。一步步梳理下来，为什么这样那样，太多东西可以追究，最后实现。如下图所示：

[](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/svm/3.5.3-1.jpeg)

至于下文中将阐述的核函数则为是为了更好的处理非线性可分的情况，而松弛变量则是为了纠正或约束少量“不安分”或脱离集体不好归类的因子。

台湾的林智仁教授写了一个封装SVM算法的[libsvm](http://www.csie.ntu.edu.tw/%7Ecjlin/libsvm/)库，大家可以看看，此外[这里](http://www.pami.sjtu.edu.cn/people/gpliu/document/libsvm_src.pdf)还有一份libsvm的注释文档。

除了在这篇论文《fast training of support vector machines using sequential minimal optimization》中platt给出了SMO算法的逻辑代码之外，[这里](http://blog.csdn.net/techq/article/details/6171688)也有一份SMO的实现代码，大家可以看下。

读者评论

本文发表后，[微博](http://weibo.com/julyweibo)上的很多朋友给了不少意见，以下是节选的一些精彩评论：

“压力”陡增的评论→//@藏了个锋：我是看着July大神的博文长大的啊//@zlkysl：就是看了最后那一篇才决定自己的研究方向为SVM的。-- <http://weibo.com/1580904460/zraWk0u6u?mod=weibotime>。

@张金辉：“SVM的三重境界，不得不转的一篇。其实Coursera的课堂上Andrew Ng讲过支持向量机，但显然他没有把这作为重点，加上Ng讲支持向量机的方法我一时半会难以完全消化，所以听的也是一知半解。真正开始了解支持向量机就是看的这篇“三重境界”，之后才对这个算法有了大概的概念，以至如何去使用，再到其中的原理为何，再到支持向量机的证明等。总之，这篇文章开启了我长达数月的研究支持向量机阶段，直到今日。”-- [http://zhan.renren.com/profile/249335584?from=template#!//tag/三重境界](http://zhan.renren.com/profile/249335584?from=template#!//tag/%E4%B8%89%E9%87%8D%E5%A2%83%E7%95%8C)。

@孤独之守望者："最后，推出svm的cost function 是hinge loss，然后对比其他的方法的cost function，说明其实他们的目标函数很像，那么问题是svm为什么这么popular呢？您可以再加些VC dimension跟一些error bound的数学，点一下，提供一个思路和方向"。-- <http://weibo.com/1580904460/AiohoyDwq?mod=weibotime>。

@夏粉\_百度：“在面试时，考察SVM可考察机器学习各方面能力：目标函数,优化过程,并行方法，算法收敛性,样本复杂度，适用场景,调参经验，不过个人认为考察boosting和LR也还不错啊。此外，随着统计机器学习不断进步，SVM只被当成使用了一个替代01损失hinge研究，更通用的方法被提出，损失函数研究替代损失与贝叶斯损失关系，算法稳定性研究替代损失与推广性能关系,凸优化研究如何求解凸目标函数，SVM,boosting等算法只是这些通用方法的一个具体组建而已。”

@居里猴姐：关于SVM损失函数的问题，可以看看张潼老师的这篇《Statistical behavior and consistency of classification methods based on convex risk minimization》。各种算法中常用的损失函数基本都具有fisher一致性，优化这些损失函数得到的分类器可以看作是后验概率的“代理”。此外，张潼老师还有另外一篇论文《Statistical analysis of some multi-category large margin classification methods》，在多分类情况下margin loss的分析，这两篇对Boosting和SVM使用的损失函数分析的很透彻。

@夏粉\_百度：SVM用了hinge损失，hinge损失不可导，不如其它替代损失方便优化并且转换概率麻烦。核函数也不太用，现在是大数据时代，样本非常大，无法想象一个n^2的核矩阵如何存储和计算。 而且，现在现在非线性一般靠深度学习了。//@Copper\_PKU:请教svm在工业界的应用典型的有哪些？工业界如何选取核函数，经验的方法？svm的训练过程如何优化？

@Copper\_PKU：July的svm tutorial 我个人觉得还可以加入和修改如下部分：(1) 对于支持向量解释，可以结合图和拉格朗日参数来表达，松弛中sv没有写出来. (2) SMO算法部分，加入Joachims论文中提到的算法，以及SMO算法选取workset的方法，包括SMO算法的收敛判断，还有之前共轭梯度求解方法，虽然是较早的算法，但是对于理解SMO算法有很好的效果。模型的优化和求解都是迭代的过程，加入历史算法增强立体感。-- [http://weibo.com/1580904460/Akw6dl3Yk#\_rnd1385474436177。](http://weibo.com/1580904460/Akw6dl3Yk#_rnd1385474436177%E3%80%82)

//@廖临川: 之所以sgd对大训练集的效果更好，1.因为SGD优化每次迭代使用样本子集，比使用训练全集（尤其是百万数量级）要快得多；2.如果目标函数是凸的或者伪凸的，SGD几乎必然可以收敛到全局最优；否则，则收敛到局部最优；3.SGD一般不需要收敛到全局最优，只要得到足够好的解，就可以立即停止。//@Copper\_PKU：sgd的核心思想：是迭代训练，每拿到一个样本就算出基于当前w(t) 的loss function，t代表训练第t次，然后进行下一w（t+1）的更新，w(t+1)=w(t)-(learning rate) \* loss function的梯度，这个类比神经网络中bp中的参数训练方法。 sample by sample就是每次仅处理一个样本 而不是一个batch。

//@Copper\_PKU：从损失函数角度说：primal问题可以理解为正则化项+lossfunction，求解目标是在两个中间取平衡 如果强调loss function最小则会overfitting，所以有C参数。 //@研究者July：SVM还真就是在一定限定条件下，即约束条件下求目标函数的最优值问题，同时，为减少误判率，尽量让损失最小。

...

参考文献及推荐阅读

《支持向量机导论》，[美] Nello Cristianini / John Shawe-Taylor 著；

支持向量机导论一书的支持网站：<http://www.support-vector.net/>；

《数据挖掘导论》，[美] Pang-Ning Tan / Michael Steinbach / Vipin Kumar 著；

《数据挖掘：概念与技术》，(加)Jiawei Han;Micheline Kamber 著；

《数据挖掘中的新方法：支持向量机》，邓乃扬 田英杰 著；

《支持向量机--理论、算法和扩展》，邓乃扬 田英杰 著；

支持向量机系列，pluskid：<http://blog.pluskid.org/?page_id=683>；

<http://www.360doc.com/content/07/0716/23/11966_615252.shtml>；

数据挖掘十大经典算法初探；

《模式识别支持向量机指南》，C.J.C Burges 著；

《统计学习方法》，李航著(第7章有不少内容参考自支持向量机导论一书，不过，可以翻翻看看)；

《统计自然语言处理》，宗成庆编著，第十二章、文本分类；

SVM入门系列，Jasper：<http://www.blogjava.net/zhenandaci/category/31868.html>；

最近邻决策和SVM数字识别的实现和比较，作者不详；

斯坦福大学机器学习课程原始讲义：<http://www.cnblogs.com/jerrylead/archive/2012/05/08/2489725.html>；

斯坦福机器学习课程笔记：<http://www.cnblogs.com/jerrylead/tag/Machine%20Learning/>；

<http://www.cnblogs.com/jerrylead/archive/2011/03/13/1982639.html>；

SMO算法的数学推导：<http://www.cnblogs.com/jerrylead/archive/2011/03/18/1988419.html>；

数据挖掘掘中所需的概率论与数理统计知识、上；

关于机器学习方面的文章，可以读读：<http://www.cnblogs.com/vivounicorn/category/289453.html>；

数学系教材推荐：<http://blog.sina.com.cn/s/blog_5e638d950100dswh.html>；

《神经网络与机器学习(原书第三版)》，[加] Simon Haykin 著；

正态分布的前世今生：<http://t.cn/zlH3Ygc>；

《数理统计学简史》，陈希孺院士著；

《最优化理论与算法(第2版)》，陈宝林编著；

A Gentle Introduction to Support Vector Machines in Biomedicine：<http://www.nyuinformatics.org/downloads/supplements/SVM_Tutorial_2010/Final_WB.pdf>，此PPT很赞，除了对引入拉格朗日对偶变量后的凸二次规划问题的深入度不够之外，其它都挺好，配图很精彩，本文有几张图便引自此PPT中；

来自卡内基梅隆大学carnegie mellon university(CMU)的讲解SVM的PPT：<http://www.autonlab.org/tutorials/svm15.pdf>；

发明libsvm的台湾林智仁教授06年的机器学习讲义SVM：<http://wenku.baidu.com/link?url=PWTGMYNb4HGUrUQUZwTH2B4r8pIMgLMiWIK1ymVORrds_11VOkHwp-JWab7IALDiors64JW_6mD93dtuWHwFWxsAk6p0rzchR8Qh5_4jWHC>；

[http://staff.ustc.edu.cn/~ketang/PPT/PRLec5.pdf](http://staff.ustc.edu.cn/%7Eketang/PPT/PRLec5.pdf)；

Introduction to Support Vector Machines (SVM)，By Debprakash Patnai M.E (SSA)，<https://www.google.com.hk/url?sa=t&rct=j&q=&esrc=s&source=web&cd=1&ved=0CCwQFjAA&url=http%3a%2f%2fwww%2epws%2estu%2eedu%2etw%2fccfang%2findex%2efiles%2fAI%2fAI%26ML-Support%2520Vector%2520Machine-1%2eppt&ei=JRR6UqT5C-iyiQfWyIDgCg&usg=AFQjCNGw1fTbpH4ltQjjmx1d25ZqbCN9nA>；

多人推荐过的libsvm：[http://www.csie.ntu.edu.tw/~cjlin/libsvm/](http://www.csie.ntu.edu.tw/%7Ecjlin/libsvm/)；

《machine learning in action》，中文版为《机器学习实战》；

SMO算法的提出：Sequential Minimal Optimization A Fast Algorithm for Training Support Vector Machines：<http://research.microsoft.com/en-us/um/people/jplatt/smoTR.pdf>；

《统计学习理论的本质》，[美] Vladimir N. Vapnik著，非常晦涩，不做过多推荐；

张兆翔，机器学习第五讲之支持向量机[http://irip.buaa.edu.cn/~zxzhang/courses/MachineLearning/5.pdf](http://irip.buaa.edu.cn/%7Ezxzhang/courses/MachineLearning/5.pdf)；

VC维的理论解释：<http://www.svms.org/vc-dimension/>，中文VC维解释<http://xiaoxia001.iteye.com/blog/1163338>；

来自NEC Labs America的Jason Weston关于SVM的讲义[http://www.cs.columbia.edu/~kathy/cs4701/documents/jason\_svm\_tutorial.pdf](http://www.cs.columbia.edu/%7Ekathy/cs4701/documents/jason_svm_tutorial.pdf)；

来自MIT的SVM讲义：[http://www.mit.edu/~9.520/spring11/slides/class06-svm.pdf](http://www.mit.edu/%7E9.520/spring11/slides/class06-svm.pdf)；

PAC问题：[http://www.cs.huji.ac.il/~shashua/papers/class11-PAC2.pdf](http://www.cs.huji.ac.il/%7Eshashua/papers/class11-PAC2.pdf)；

百度张潼老师的两篇论文：《Statistical behavior and consistency of classification methods based on convex risk minimization》[http://home.olemiss.edu/~xdang/676/Consistency\_of\_Classification\_Convex\_Risk\_Minimization.pdf，《Statistical](http://home.olemiss.edu/%7Exdang/676/Consistency_of_Classification_Convex_Risk_Minimization.pdf%EF%BC%8C%E3%80%8AStatistical) analysis of some multi-category large margin classification methods》；

<http://jacoxu.com/?p=39>；

《矩阵分析与应用》，清华张贤达著；

SMO算法的实现：<http://blog.csdn.net/techq/article/details/6171688>；

常见面试之机器学习算法思想简单梳理：<http://www.cnblogs.com/tornadomeet/p/3395593.html>；

矩阵的wikipedia页面：<http://zh.wikipedia.org/wiki/%E7%9F%A9%E9%98%B5>；

最小二乘法及其实现：<http://blog.csdn.net/qll125596718/article/details/8248249>；

统计学习方法概论：<http://blog.csdn.net/qll125596718/article/details/8351337>；

<http://www.csdn.net/article/2012-12-28/2813275-Support-Vector-Machine>；

A Tutorial on Support Vector Regression：<http://alex.smola.org/papers/2003/SmoSch03b.pdf>；SVR简明版：[http://www.cmlab.csie.ntu.edu.tw/~cyy/learning/tutorials/SVR.pdf](http://www.cmlab.csie.ntu.edu.tw/%7Ecyy/learning/tutorials/SVR.pdf)。

SVM Org：<http://www.support-vector-machines.org/>；

R. Collobert. Large Scale Machine Learning. Université Paris VI phd thesis. 2004：<http://ronan.collobert.com/pub/matos/2004_phdthesis_lip6.pdf>；

Making Large-Scale SVM Learning Practical：<http://www.cs.cornell.edu/people/tj/publications/joachims_99a.pdf>；

文本分类与SVM：<http://blog.csdn.net/zhzhl202/article/details/8197109>；

Working Set Selection Using Second Order Information for Training Support Vector Machines：[http://www.csie.ntu.edu.tw/~cjlin/papers/quadworkset.pdf](http://www.csie.ntu.edu.tw/%7Ecjlin/papers/quadworkset.pdf)；

SVM Optimization: Inverse Dependence on Training Set Size：<http://icml2008.cs.helsinki.fi/papers/266.pdf>；

Large-Scale Support Vector Machines: Algorithms and Theory：[http://cseweb.ucsd.edu/~akmenon/ResearchExam.pdf](http://cseweb.ucsd.edu/%7Eakmenon/ResearchExam.pdf)；

凸优化的概念：<http://cs229.stanford.edu/section/cs229-cvxopt.pdf>；

《凸优化》，作者: Stephen Boyd / Lieven Vandenberghe，原作名: Convex Optimization；

Large-scale Non-linear Classification: Algorithms and Evaluations，Zhuang Wang，讲了很多SVM算法的新进展：<http://ijcai13.org/files/tutorial_slides/te2.pdf>；