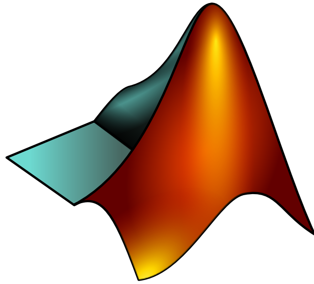
Recipes for Faster Matlab Code

http://freemind.pluskid.org/programming/recipes-for-faster-matlab-code

posted on Free Mind on June 21, 2012
generated with pandoc on December 3, 2015
category: Programming

tags: Tools, Matlab



Matlab 在 Research 中用得非常多，确实也是非常方便实用，只是有一个问题就是写 Matlab 代码的时候经常需要用一些比较~~奇怪~~独特的方式来思考和处理问题，否则写出来的代码虽然同样能工作，但是速度上可能会差上几百几千倍。这里有几个关键词：向量化、缓存、稀疏性等。不过由于 Matlab 在这方面确实 “问题” 比较大，所以关于如何写更高效的 Matlab 代码的文章也已经非常多了。但是刚巧最近 bdahz 小朋友问我一段 Matlab 代码为什么又短又奇怪但是速度又快，所以我觉得正好可以拿这个作为一个例子，罗嗦一下 Matlab 编程时候的一些注意事项，也许会对其他人也有所帮助。

这次我们的例子是 K-medoids 算法，我在以前的 blog 中也介绍过。简单地来说就是 K-means 算法的一个变种，只是在选取中心的时候 K-means 是直接计算所有点的平均值，而 K-medoids 则要求中心点必须是数据点中某一个，所以 K-means 的优化如果是一个数值计算问题的话，K-medoids 应该属于离散优化，通常离散优化需要穷举搜索来求解，所以计算上会更难一些。不过实现得好的话，也是可以比较高效的，比如这个版本的 Matlab K-medoids。

接下来我们就用这个作为例子分析一下写高效的 Matlab 代码需要注意的一些问题，先把代码贴出来吧：

```
function [label, energy, index] = kmedoids(X,k)
% X: d x n data matrix
% k: number of cluster
% Written by Mo Chen (sth4nth@gamil.com)
v = dot(X,X,1);
D = bsxfun(@plus,v,v')-2*(X'*X);
n = size(X,2);
[~, label] = min(D(randsample(n,k),:));
last = 0;
while any(label ~= last)
    [~, index] = min(D*sparse(1:n,label,1,n,k,n));
    last = label;
    [val, label] = min(D(index,:),[],1);
```

```
14 end
15 energy = sum(val);
```

从这段代码里我们可以看到写高效的 Matlab 代码的首要注意事项是：把代码写得晦涩难懂……呃，开各玩笑 `^_^bb`，不过也确实是这样，其实这样的问题在各种语言中都是存在的：教学或者示例用的代码通常和真正实际项目中的代码差别很大，实际中往往掺杂各种错误处理呀边界处理之类的，变得很复杂；不过代码清晰度最大的敌人往往还是优化。为了让代码运行效率更高效所做的各种努力几乎都会很严重或者非常严重地破坏代码的可理解性，使得原本很清晰的算法变得面目全非。

通常人们解决这类问题的办法就是把一些通用的优化机制总结起来，实现到编译器里面去，让编译器来做这些 dirty work。就 C/C++ 来说的话，现在的编译器虽然离完美还差的很远，但是在优化方面也算是已经非常强大了。可惜的是 Matlab 在这方面似乎做得不是很好——虽然 Matlab 严格地来说是没有编译器的。比如说，Matlab 里面用 `for` 循环是非常慢的，导致大家都不太敢用 `for` 循环，于是 Matlab 后来说提供了 `for` 的 JIT 机制，据说加快速度，但是似乎结果仍然是非常慢。所以没办法，幸运的是 Matlab 代码通常都是比较短的。

回到我们的例子上，首先看第 5、6 两行，这两行做的事情实际上就是计算所以数据点之间的 pair-wise distance，放在变量 `D` 里。由于 pair-wise distance 在算法中要被用到很多次，并且是不会变化的，所以一开始把它计算并存储下来后面直接用，这是所有语言里都通用的一个加速方法，或者也可以说成是空间换时间，因为如果数据量比较大 pair-wise distance 矩阵在内存中无法存下来的话，就没有足够的空间来换时间了。

然后我们来看看这个 pair-wise distance 矩阵是怎么计算出来的。首先第 5 行 `dot` 函数参考 Matlab 的帮助文档就知道计算了矩阵 `X` 每一列和自己的点乘。然后第 6 行用了一个奇怪的函数 `bsxfun`。让我们先忽略这个函数，来看一下两个点 x 和 y 之间的距离应该是怎么计算的，定义如下：

$$d(x,y) = \|x-y\| = \sqrt{\sum_{i=1}^{m}(x^i-y^i)^2}$$

但是由于我们这里只需要比较距离之间的相对大小，所以可以省略一个开平方根的计算，使用 “平方距离”：

$$D(x,y) = \|x-y\|^2 = \sum_{i=1}^{m}(x^i-y^i)^2$$

当然，我们说了，在 Matlab 里用 `for` 循环来计算是很慢的，所以我们要用**向量化**的方法来计算，可以这样写 `sum((x-y).^2)`。但是这里的问题是，我们要计算很多点之间的 pair-wise distance，虽然每一对点之间的距离可以这样算的话，要计算所有点之间的距离，好像仍然无法避免

两重 `for` 循环来遍历所有的点。但是那样又会很慢了，所以我们需要更加深层次的向量化，首先展开距离公式

$$D(x,y) = \|x-y\|^2 = \|x\|^2 + \|y\|^2 - 2x^Ty$$

这样把距离的计算分成了三个部分，前面两个部分都是计算向量的 norm （的平方），而第三个部分是计算向量内积。这样的形式的好处是可以方便地对一堆点同时进行计算：例如，对于矩阵 X 的每一列的 norm 平方，就可以用我们刚才提到的 `dot` 函数一次算出来，也是代码中第 5 行干的事情。接下来是内积，这个也简单，通过矩阵乘法的公式就可以知道，如果 $Z = X^TX$ 的话，那么

$$Z_{ij} = x_i^T x_j$$

其中 x_i 是矩阵 X 的第 i 列。所以一次矩阵乘法 `X'*X` 就可以把所有 pair-wise 内积全部算出来，不用任何循环。所以接下来只要把三个部分加起来就可以了，不过这里还有一个问题：虽然 `X'*X` 是得到的一个形状合适的矩阵，但是 `dot(X,X)` 得到的却是一个向量。为了看得更清楚一点，我们分别用 $\bar{X}$、$\hat{X}$ 和 Z 表示 pair-wise distance 计算的三个部分，按理说应该计算得到三个形状相同的矩阵，然后相加起来：

$$D_{ij} = \bar{X}_{ij} + \hat{X}_{ij} - 2Z_{ij}$$

显然 $\bar{X}_{ij} = \|x_i\|^2$，而 $\hat{X}_{ij} = \|x_j\|^2$，所以对于 $\bar{X}$ 来说，列坐标是无关紧要的，如果记之前 `dot` 得到的结果向量为 `v` 的话，$\bar{X}$ 应该是向量 `v` 按列不断重复而得到的矩阵；类似的，$\hat{X}$ 应该是 `v` 转置之后按行重复得到的矩阵。在 Matlab 中经常需要这样的操作，用 `repmat` 即可完成，所以，下面的代码实际上就可以计算 pair-wise distance 矩阵：

```
1 v = dot(X,X);
2 D = repmat(v, length(v),1) + repmat(v', 1, length(v)) - 2*(X'*X);
```

这里又碰到了一个空间换时间的问题：由于我们希望用向量化的方式 “同时” 计算所有点对的距离，所以我们需要把 `v` 扩张成 $\bar{X}$ 和 $\hat{X}$ 这两个矩阵，需要的存储空间从 n 变到了 $2n^2$，并且存储的都是重复的元素，如果用 `for` 循环一个一个地计算的话，这些多余的空间当然是可以避免的，但是 Matlab 的 `for` 又很慢。不过由于这个问题出现得非常多，于是 Matlab 提供了一个解决方案：`bsxfun`。详细的文档可以看 Matlab 的帮助，讲得很清楚，简单地来说，`bsxfun` 就是对矩阵的每个元素做同一个操作，基本等价于于写一些 `for` 来对矩阵元素做计算，不同的是速度快了许多许多倍。另外还有一个特点就是传给 `bsxfun` 的矩阵如果某一个维度上 size 是 1 的话，在那个维度上它会根据传进来的其他矩阵做 “重

复扩展”，所做的事情和我们人肉用 `repmat` 是一样的，只是实现方式并不是这样，它并不会生成临时矩阵，所以在内存方面绝对占有。

原来代码里其实就是用 `bsxfun` 做了我们刚才用 `repmat` 做的事情。下面的代码对比了三种方法：

```
function test_dist(m, n)
    X = rand(m,n);
    D = use_bsxfun(X);
    D = use_repmat(X);
    D = use_for(X);
end

function D=use_bsxfun(X)
    v = dot(X,X);
    D = bsxfun(@plus,v,v')-2*(X'*X);
end

function D=use_repmat(X)
    v = dot(X,X);
    D = repmat(v,length(v),1) + repmat(v',1,length(v)) - 2*(X'*X);
end

function D=use_for(X)
    D = zeros(size(X,2));
    for i=1:size(D,1)
        for j=i+1:size(D,2)
            D(i,j) = sum((X(:,i)-X(:,j)).^2);
        end
    end
    D = max(D,D');
end
```

用 Matlab 的 Profiler 运行一下（顺便说一下，Matlab 的 Profiler 是非常好用的工具，也是提升代码性能的重要工具，用善用），在我这里，`bsxfun`、`repmat` 和用循环的方式的运行时间（`m=1000,n=1000`）分别是 0.26、0.18 和 8.22。循环比 `repmat` 慢了近 50 倍，`bsxfun` 和 `repmat` 速度差不多，但是内存更省一些，一般推荐使用 `bsxfun`。

然后是第 8 行，先是用 `randsample` 随机选出 `k` 个点作为初始 center，然后为每个数据点计算 label：也就是找出它们与 `k` 个 center 距离最近的那个所对应的 index。这也是用向量化的方法一次性计算的，因为 Matlab 的 `min` 函数能够支持向量化操作，事实上 Matlab 的大多数基本函数都支持向量化操作，多看一下文档会有好处。

然后是第 11 行，这一行的目的是根据每一类的数据点重新选点每类的中心点，这一步中就是 K-means 和 K-medoids 不同的地方：K-medoids 由于要求类中心必须是数据点中的某一个，所以这里需要用遍历搜索的方法：遍历该类中的所有数据点，选中最优的中心。这里最优的定义是：该中心到该类的其他点的距离之和最小，这个是和 K-means 的定义一致的。不过从代码里来看，这里显然又用了向量化的方法而不是循环来处理了搜索。

让我们来看一下代码里是怎么做的：代码里的 `sparse` 函数（具体用法请参考 Matlab 帮助）构造了一个 $n \times k$ 的稀疏矩阵，不妨暂时记为 M，如果第 i 个数据点属于第 j 类的话，那么 $M_{ij} = 1$，否则等于 0。然后用 pair-wise distance 矩阵 D 去乘上 M，得到一个 $n \times k$ 的矩阵暂时记为 N。来看一下 N_{ij}，它是 D 的第 i 行和 M 的第 j 列内积的结果。M 的第 j 列标记了所有属于第 j 类的点，其他位置全部是零，因此这样内积的结果就是所有第 j 类中的数据点到数据点 i 的距离之和。因此，对于第 j 类来说，只要求得 N 的第 j 列中数值最小的那个下标对应的数据点，即是最优的中心点，而 `min` 函数是可以对于一个矩阵所有列同时求最小的，也就是代码中该行达到的目的。

这里除了向量化之外还有一个注意事项就是**稀疏矩阵**。稀疏矩阵并不一定是很高效的，比如对里面的元素进行下标随机访问就会很慢，但是有许多其他操作则可以很快（如果用了合适的函数的话），比如矩阵相乘、矩阵遍历（寻找非零元素或者寻找最大、最小值等）、解方程、求特征向量和特征值。比如说求特征向量，如果矩阵是稀疏的，那么可以用 `eigs` 来进行求解，它的一个优点是可以只求想要的几个解，而不像 `eig` 那样必须把所有解全部求出来，并且由于它是用迭代法，其中主要涉及到一些矩阵向量乘积之类的，用稀疏矩阵进行运算也会很快。当然迭代法的缺点就是可能误差比直接求解更大一些，数值稳定性也更差一些。另外就是当数据矩阵本身维度非常大但是又非常稀疏的时候，用稀疏矩阵非常节约内存。下面是一个简单的测试例子（运行时间在注释里）：

```
1 X = sprand(300, 2000, 0.01);
2 S = X'*X;
3
4 [A B] = eigs(S);       % time:  0.16
5 [A B] = eigs(full(S)); % time:  0.91
6 [A B] = eig(full(S));  % time: 13.76
```

运行时间差异还是比较清楚的，也就不用我多解释了。不过有一点需要注意的是，`eigs` 在没有指定个数的情况下默认是只求 6 个特征值和特征向量的，所以和 `eig` 把所有的特征值特征向量全部求出来其实也并不是可以直接比较的。但是许多时候我们需要的都不是所有的特征向量和特征值，如果真的需要全部的话，用 `eigs` 来计算可能并不是一个合适的选择，届时可以自己尝试和比较一下。由于求特征值和特征向量在各种

基于 Graph 的方法（像 Laplacian Eigenmaps、Laplacian Regularized Least Square 等）中用得非常多，所以这些还是很有用的。这里可以顺便简单说一下稀疏矩阵的存储。当然是有各种存储方式的，比较基本的比如按行存储：矩阵是一个链表，把矩阵的每一行链起来，而每一行也是一个链表，把该行的非零元素链起来；类似的有按列存储；此外可能还有完全按元素存储，可以看成一个表格，如果 `(i,j)` 位置有非零值的话，就索引到该值。根据不同的存储方式，计算效率也会不一样，比如一个按行存储的矩阵乘以一个按列存储的矩阵，就可以很快，因为矩阵相乘的计算方式就是左边的行和右边的列做内积；但是如果反过来，一个按列存储的矩阵乘以一个按行存储的矩阵的话，就会比较麻烦了。Matlab 里做得比较好的是把稀疏矩阵搞得很透明，你不用关心它底层到底是怎么存储的，大多数时候就像使用普通矩阵一样用就 OK 了，并且性能也挺不错。

回到我们原来的代码，第 13 行和第 8 行是一样的，只是现在使用计算出来的中心而不是随机选出来的。剩下的就没有什么好解释的了。这里的 stop condition 是中心点不再变动，intuitively 想一想对于 K-medoids 来说这样的 stop condition 似乎在一些比较特殊的情况下可能会出现来回振荡不停下来的结果，不过那个不是我们今天要关注的问题了。

最后总结一下，要写出高效的 Matlab 代码的一些注意事项：

- Profiler: Matlab 的 Profiler 是非常好用的，要善于利用这个工具，同所有其他编程语言一样，找准 bottleneck 是进行优化的最重要的一步，如果只是想当然地去搞的话可能浪费了大把的精力又没有把性能改善多少而且还把代码搞得一团糟。
- Sparsity: 如果问题是有稀疏性质的，那么可以尝试一下用稀疏矩阵和配套的那些操作。
- Vectorization: 向量化可以说是 Matlab 编程的一个特点，就好像函数式编程总是一堆 `map` 呀 `filter` 呀 `reduce` 呀之类的一样。用好向量化是改善 Matlab 性能的关键。要多尝试和练习，逐渐习惯向量化的思维方式。特别是矩阵相乘呀、分块之类的要熟练，例如我们在介绍代码第 11 行的时候构造的那个矩阵 M，通常称作 indicator matrix，元素只有 `0` 和 `1`，一般用于表示哪些元素被选出来了。这个矩阵不论是在计算上还是在公式推导上都经常被用到。