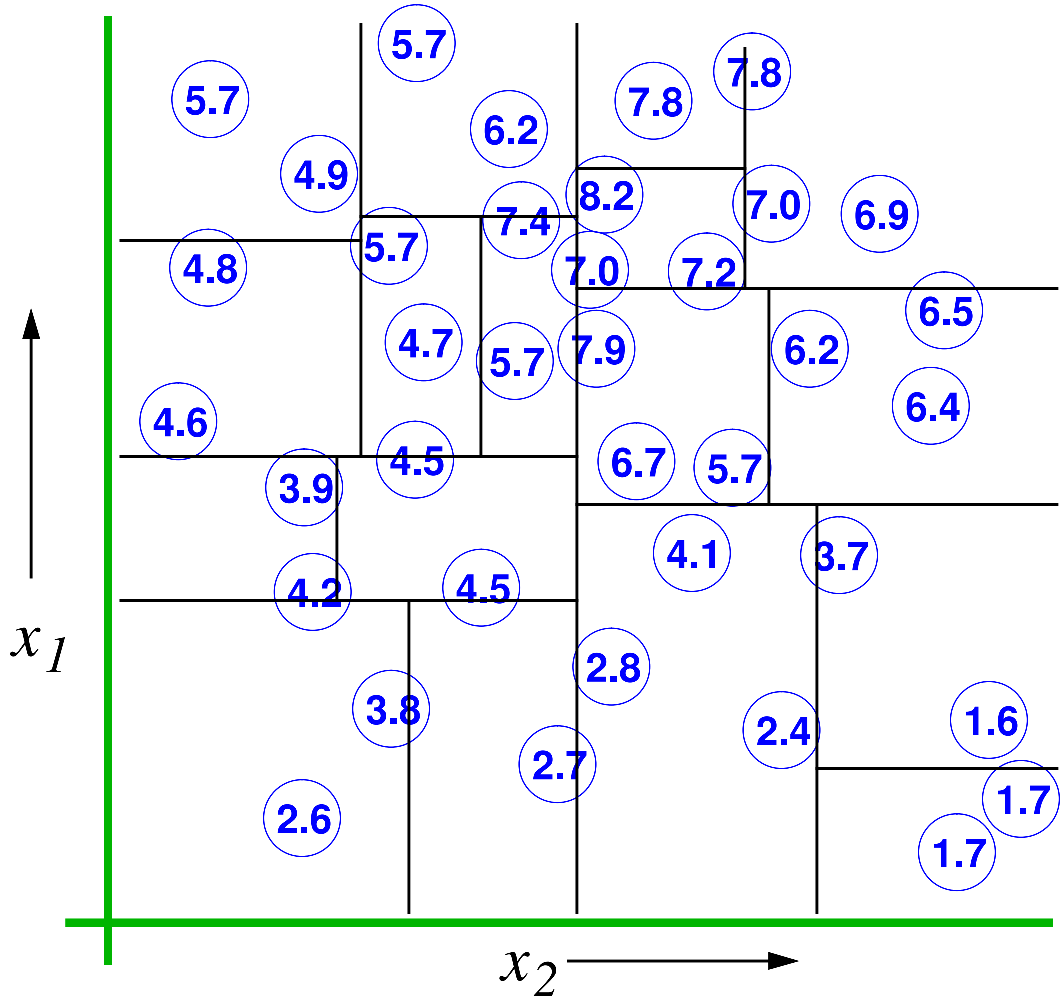
Idea 2: cache output surface

- First, decide which instance-based algorithm you'll use, fix the kernel width etc.
- **Caching step:** perform lots of (slow) predictions but save the results in some interpolative table.
  - Simple (but expensive): cache in a multidimensional array.
  - Fancy (still expensive): cache in a *k*d-tree and predict with multilinear interpolation [Grosse, 1992].
  - Simple, cheaper, but discontinuous: cache linear models inside leaves of a tree. e.g. [Quinlan, 1993].

BUT: Must commit to a kernel width during caching. Lose regression analysis. Caching step can be very very very expensive. Incrementally adding points can be hard.

Idea 3: Range Searching

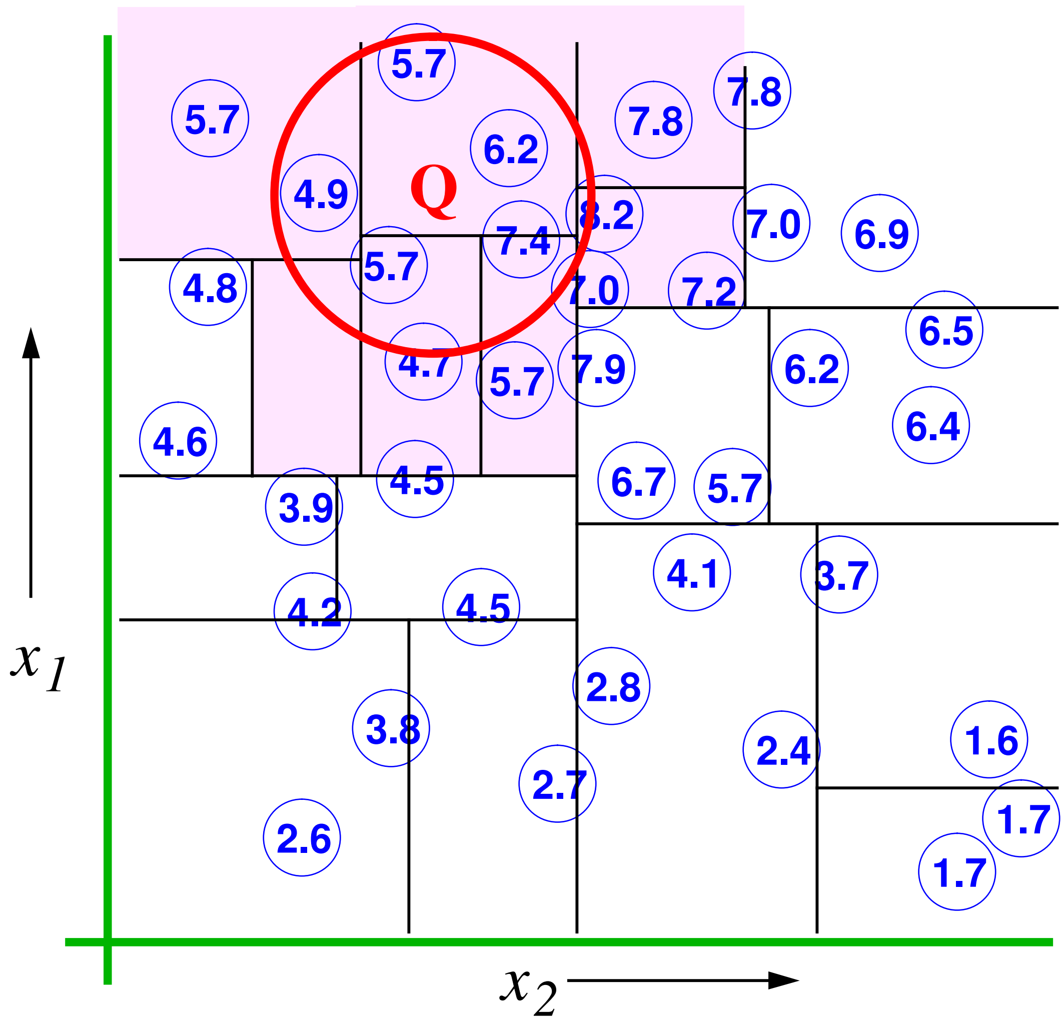
Embed your data in the leaves of a *k*d-tree [Friedman et al, 1977]



If you have a small kernel width, then beyond a certain distance from your query, all weights will be very close to zero.

You can quickly search the tree to get the set of points with non-zero-weight.

Idea 3: Range Searching



PRO: For small kernel widths much cheaper than linear search. Different queries can have different kernels. And you can do full regression analysis.

CON: For large kernel widths or very dense datasets, you're still in trouble.

This talk: Multiresolution Approach

What do we really need in order to do a locally weighted regression query?

$$\beta = \left(\Sigma_k\, w_k^{\,2}\, \mathbf{x}_k^{\mathrm{T}}\, \mathbf{x}_k\right)^{-1} \left(\Sigma_k\, w_k^{\,2}\, \mathbf{x}_k^{\mathrm{T}}\, y_k\right)$$

Let's look at how we can build the

$\left(\Sigma_k\, w_k^{\,2}\, \mathbf{x}_k^{\mathrm{T}}\, \mathbf{x}_k\right)$ matrix...

$$\begin{bmatrix} \sum w_k^2 x_{k1}^2 & \sum w_k^2 x_{k1} x_{k2} & .. & \sum w_k^2 x_{k1} x_{kN} \\ \sum w_k^2 x_{k1} x_{k2} & \sum w_k^2 x_{k2}^2 & .. & \sum w_k^2 x_{k2} x_{kN} \\ : & : & & : \\ \sum w_k^2 x_{k1} x_{kN} & \sum w_k^2 x_{k2} x_{kN} & .. & \sum w_k^2 x_{kN}^2 \end{bmatrix}$$

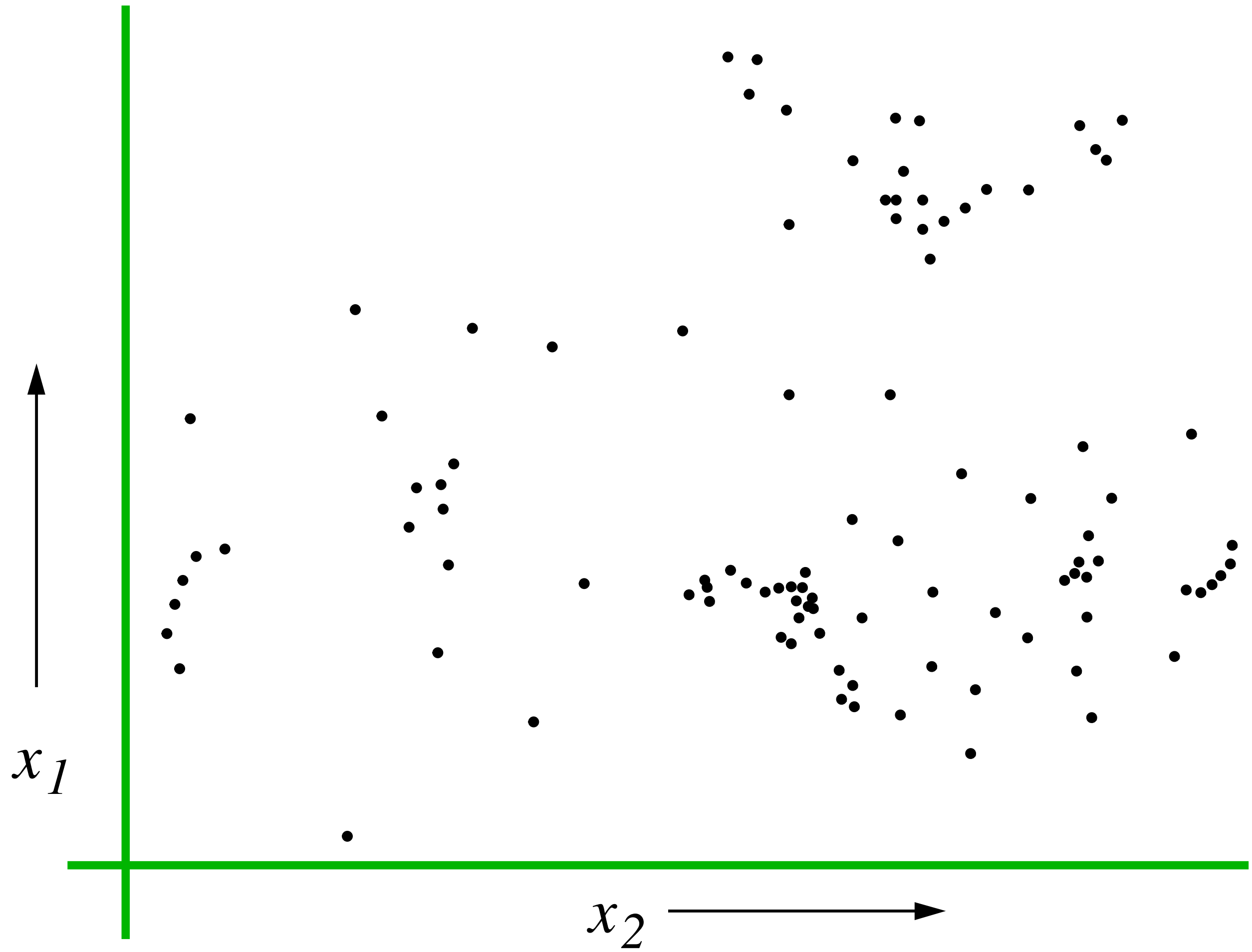
The slow way:

wwxtx = MxM matrix of zeros.

for (k = 1 ; k <= N ; k := k + 1)

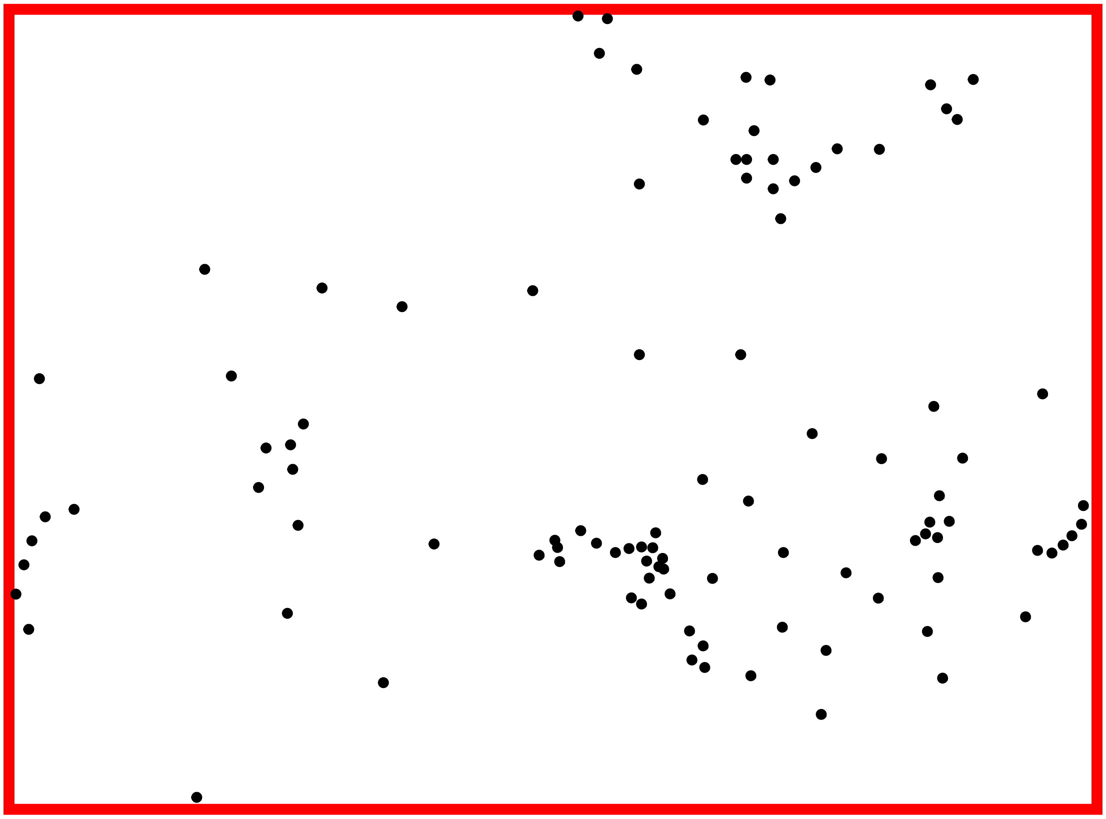
wwxtx := wwxtx + $w_k^{\,2}\, \mathbf{x}_k^{\mathrm{T}}\, \mathbf{x}_k$

A multiresolution structure



Take your data and build a kind of *k*d-tree structure with extra cached information...

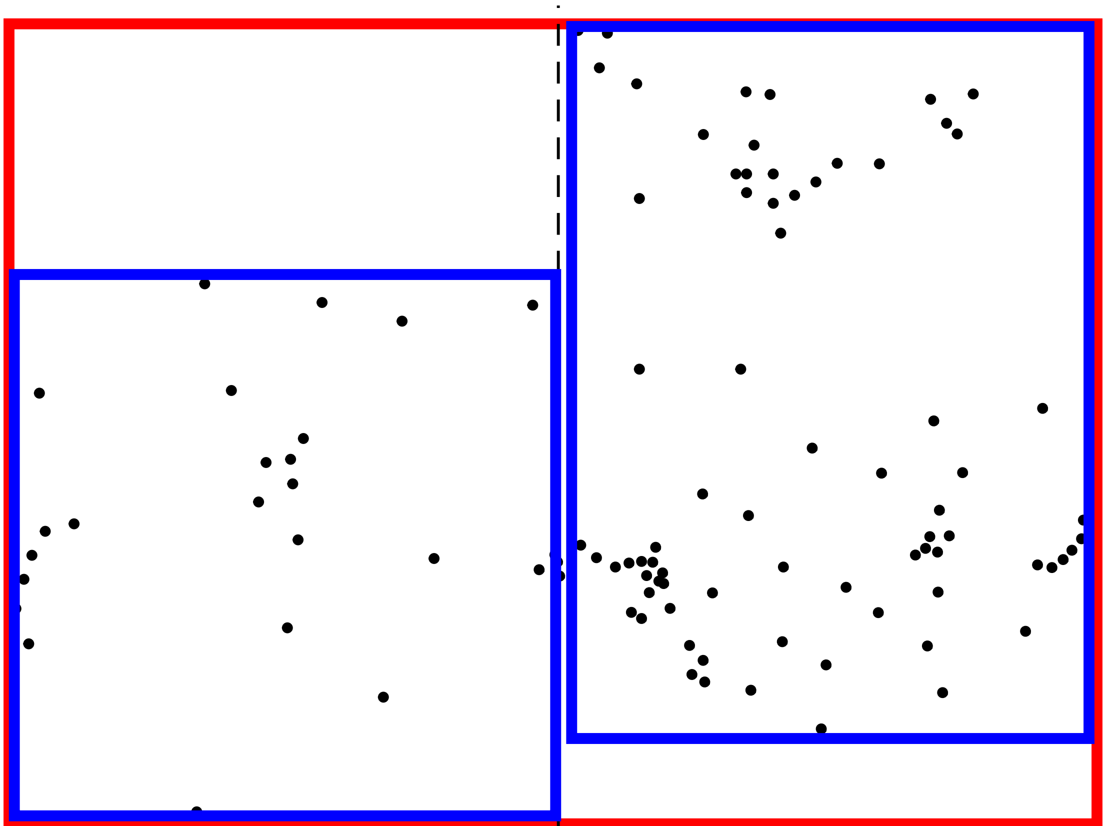
Top level of tree



NumPoints = 88
BoundingBox=(1.2,1.7)(6.9,3.8)

$$\Sigma x_k^{\mathrm{T}} x_k = \begin{pmatrix} 1200 & 830 \\ 830 & 733 \end{pmatrix}$$

Second level of tree



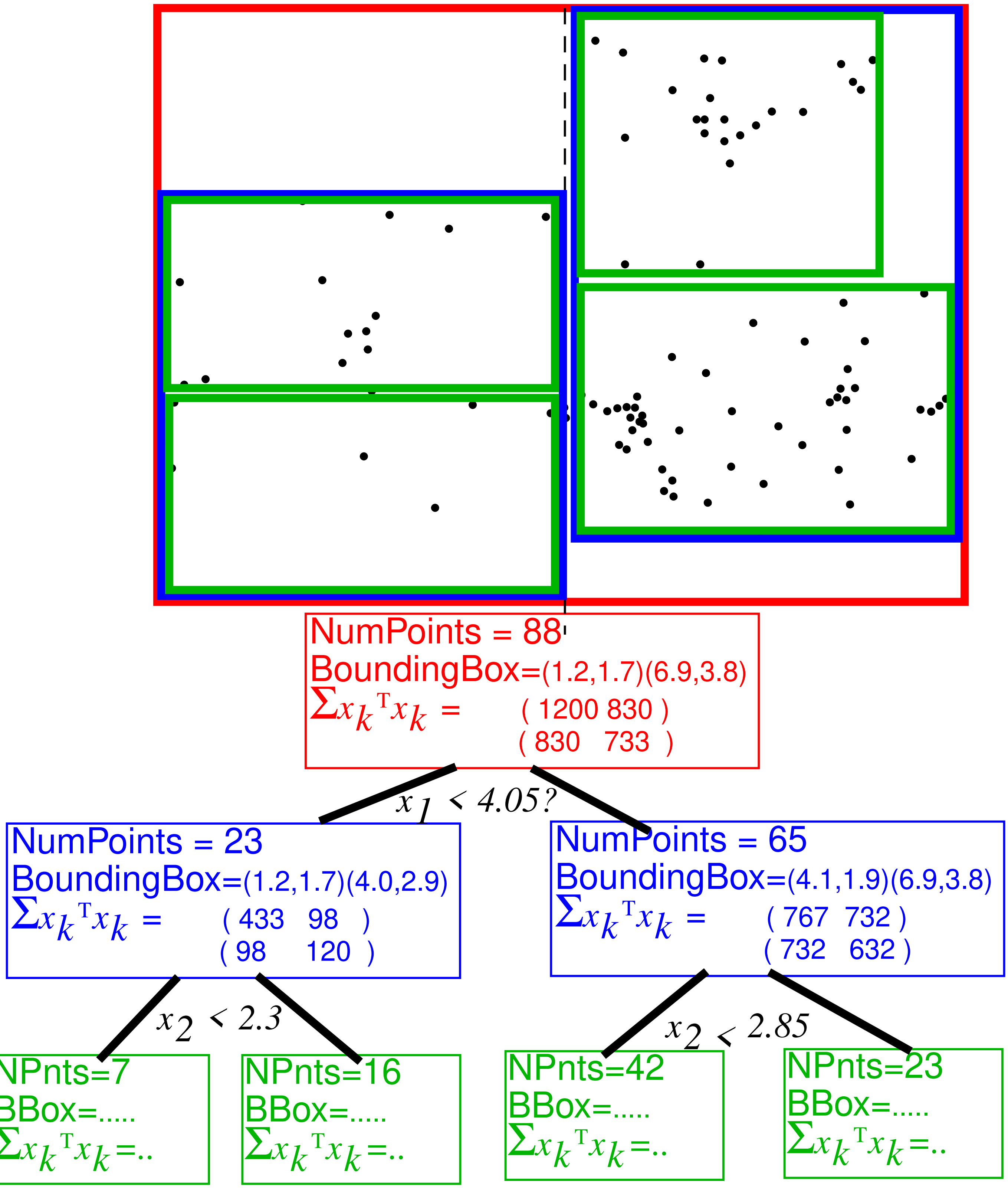
NumPoints = 88
BoundingBox=(1.2,1.7)(6.9,3.8)
$$\Sigma x_k^T x_k = \begin{pmatrix} 1200 & 830 \\ 830 & 733 \end{pmatrix}$$

$x_1 < 4.05?$

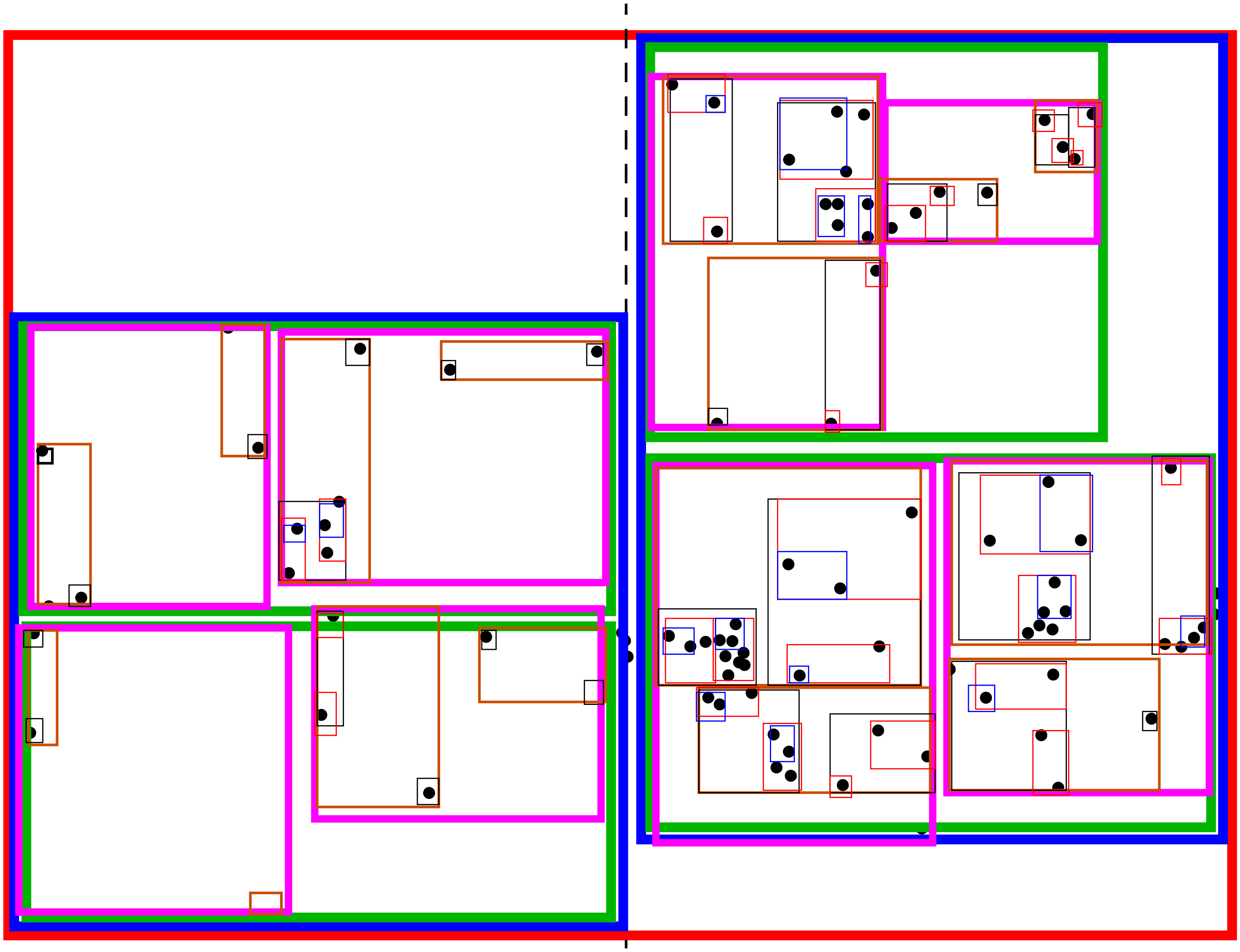
NumPoints = 23
BoundingBox=(1.2,1.7)(4.0,2.9)
$$\Sigma x_k^T x_k = \begin{pmatrix} 433 & 98 \\ 98 & 120 \end{pmatrix}$$

NumPoints = 65
BoundingBox=(4.1,1.9)(6.9,3.8)
$$\Sigma x_k^T x_k = \begin{pmatrix} 767 & 732 \\ 732 & 632 \end{pmatrix}$$

Third level of tree



Further levels of tree



Points for discussion:

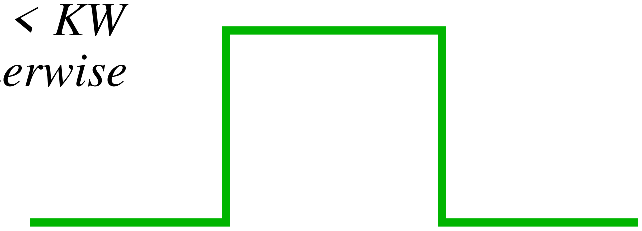
- The tree can be added to incrementally
- Subtract out centroids from $\Sigma x_k{}^T x_k$ matrices for numerical stability
- You must similarly compute $\Sigma x_k{}^T y_k$ and $\Sigma y_k{}^2$

This slide is beautiful but took a long time to create. Appreciate it.

Using the tree.

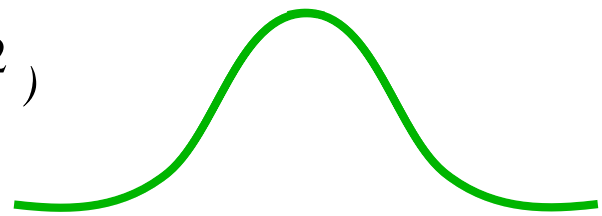
If our weighting function was...

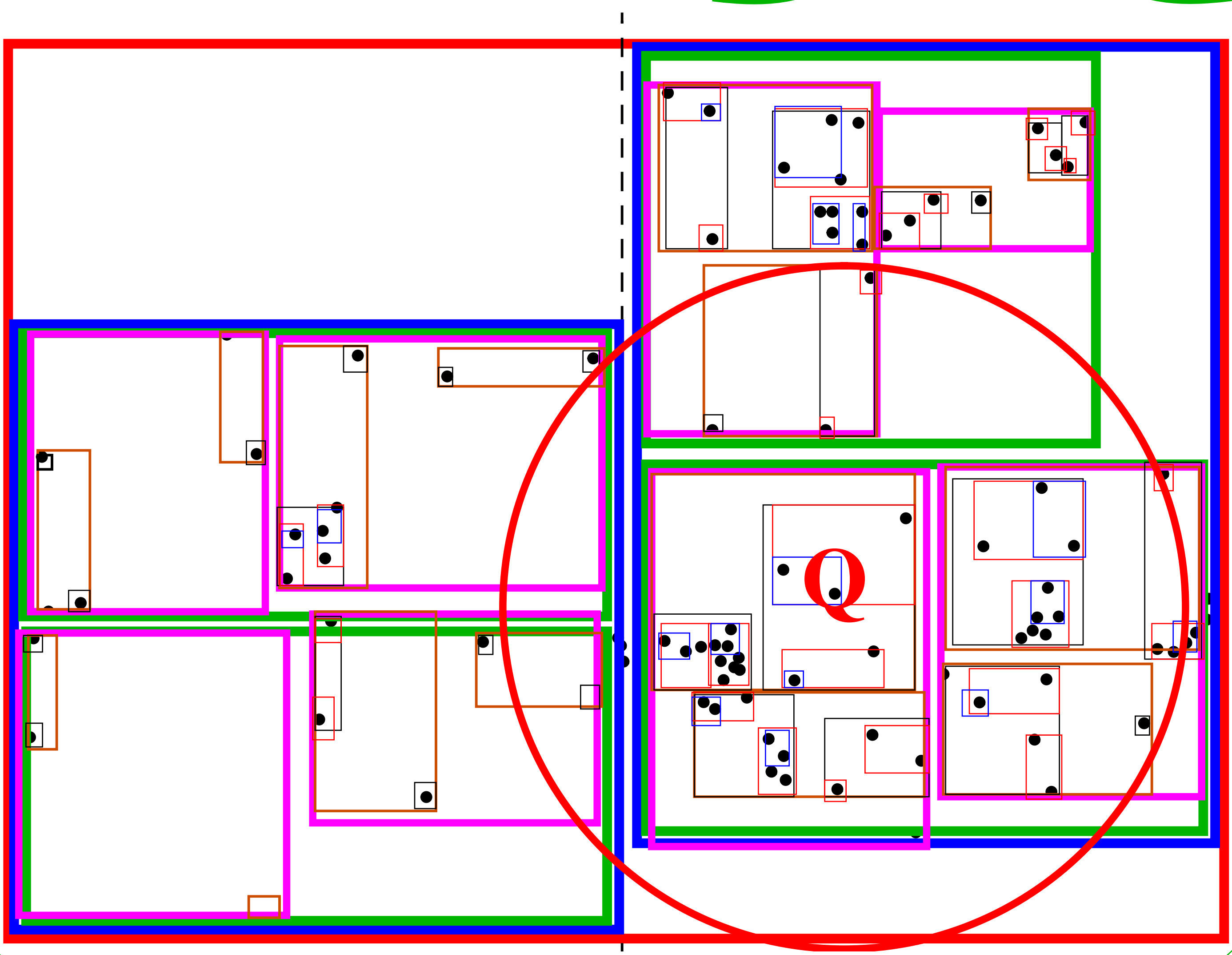
$$W(Dist) = 1 \text{ if } Dist < KW$$
$$= 0 \text{ otherwise}$$



Instead of...

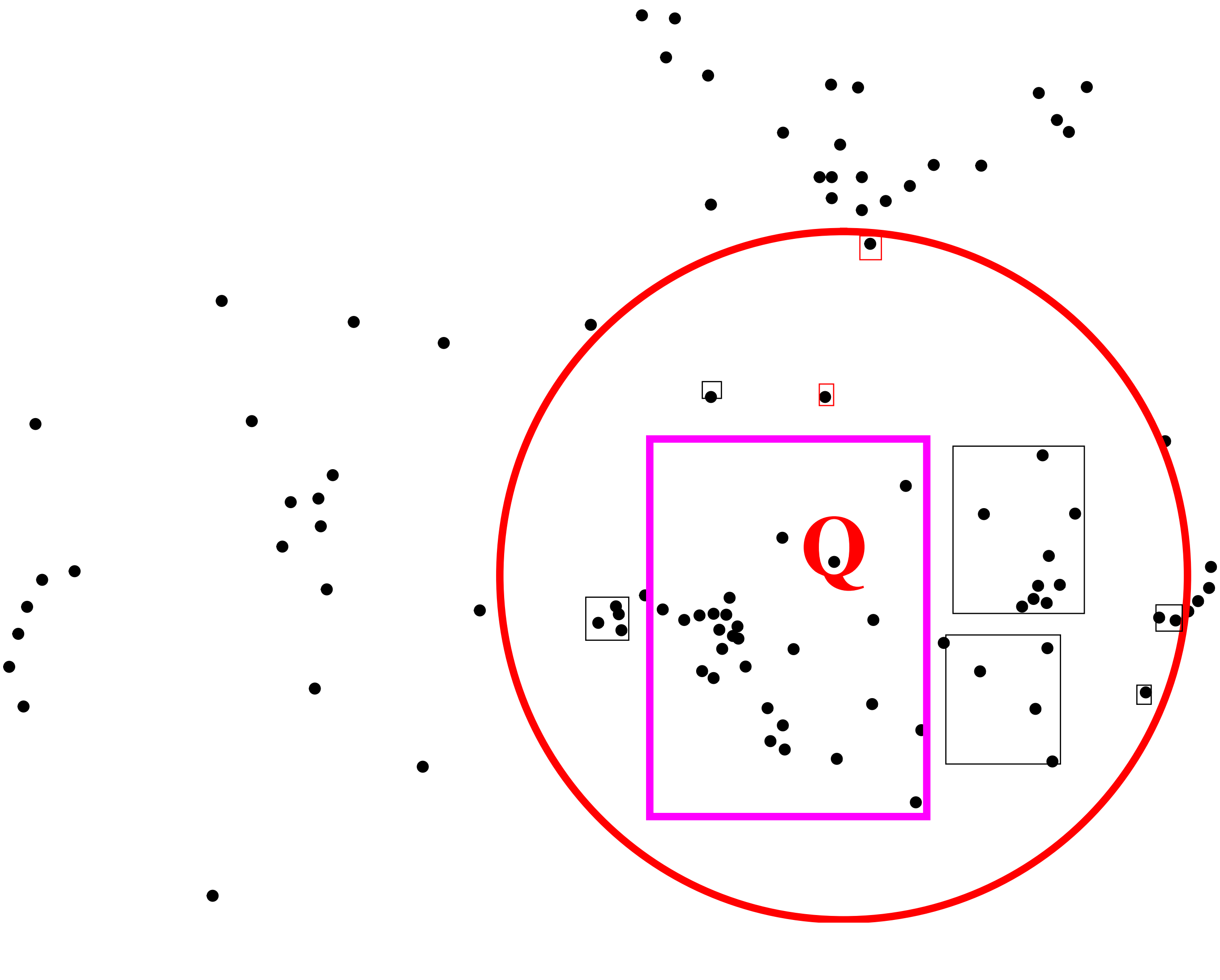
$$W(Dist) = \exp(-dist^2 / Kw^2)$$





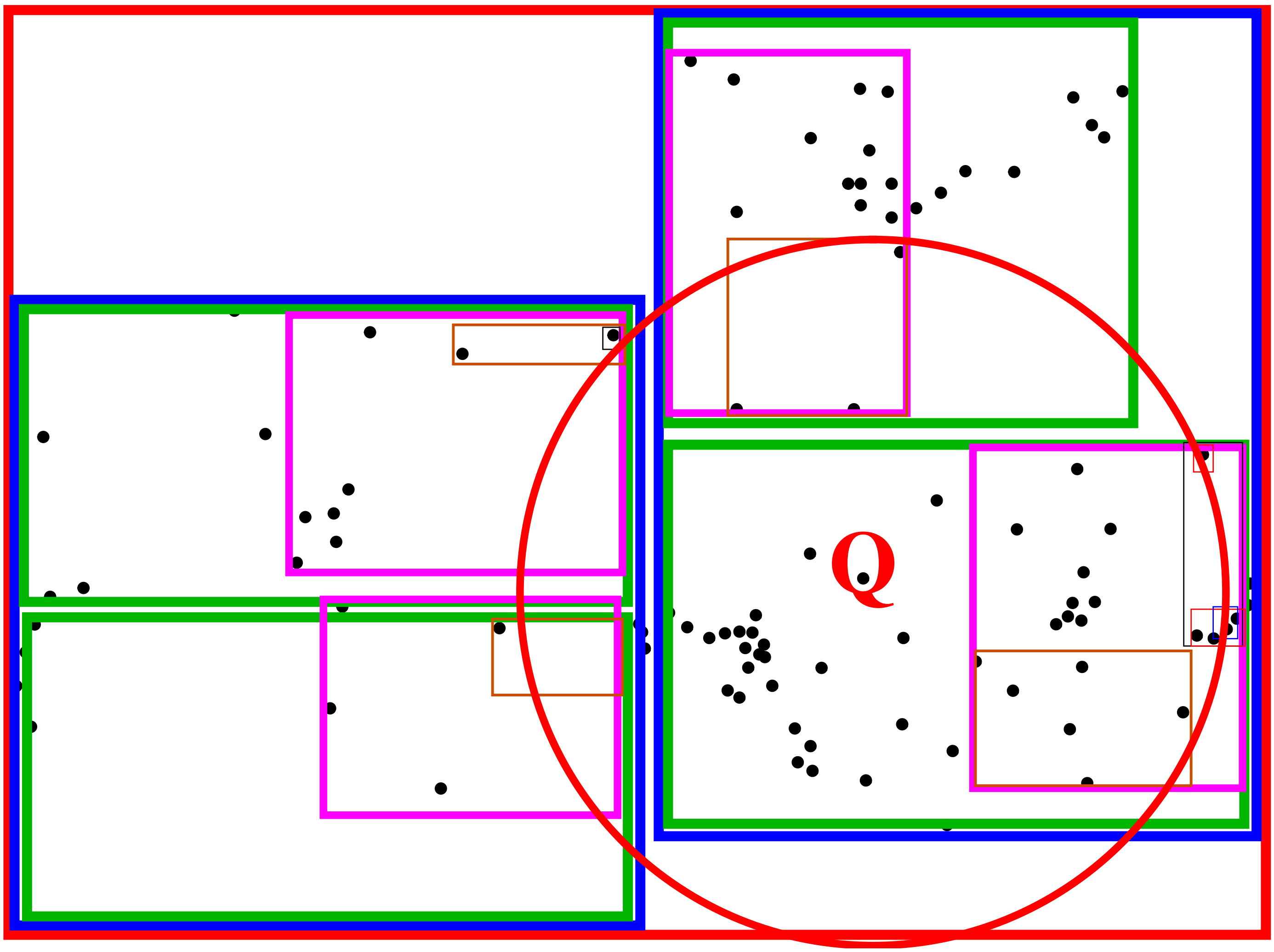
Using the tree

If we had a step weighting function then we would merely add together matrices of all nodes that were entirely within the critical distance .



Using the tree

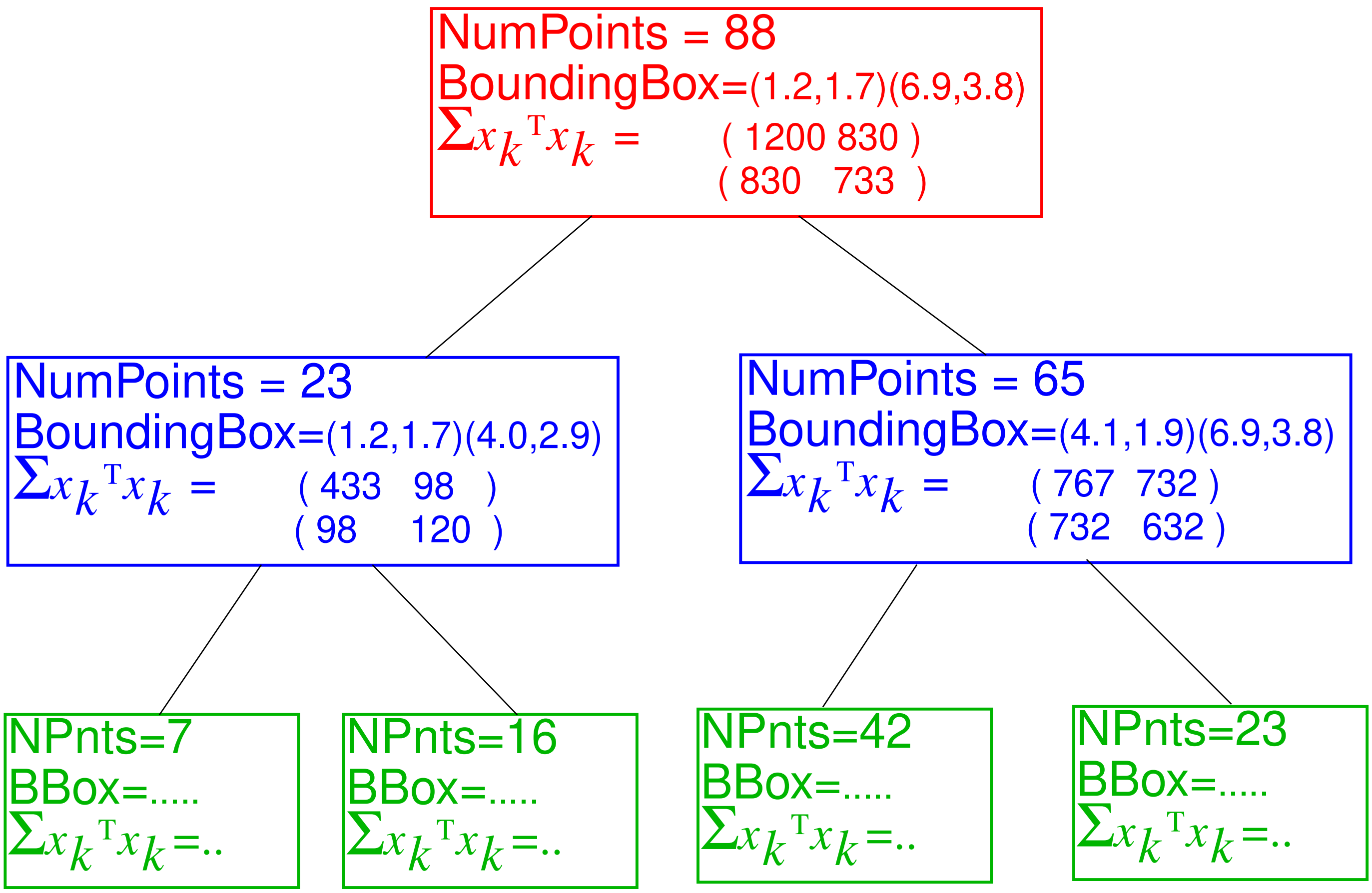
To find this set of nodes , we'd merely need to visit all nodes that intersect the critical distance ..



With a Gaussian weight fn...

Recurse down the tree.

- At each node you must compute the weighted sum $\sum_k w_k^2 \mathbf{x}_k^T \mathbf{x}_k$ of all points below in the tree.
- If node has only one point $\mathbf{x}_j$, return $w_j^2 \mathbf{x}_j^T \mathbf{x}_j$
- Else compute $\sum_k w_k^2 \mathbf{x}_k^T \mathbf{x}_k$ of your left subtree and in your right subtree. Return their sum.



- But ***SOMETIMES*** you may discover that you can avoid searching a node ....