# Instance-based Learning

## (a.k.a. memory-based) (a.k.a. non-parametric regression) (a.k.a. case-based) (a.k.a kernel-based)

## Part II: Regression

Note to other teachers and users of these slides. Andrew would be delighted if you found this source material useful in giving your own lectures. Feel free to use these slides verbatim, or to modify them to fit your own needs. PowerPoint originals are available. If you make use of a significant portion of these slides in your own lecture, please include this message, or the following link to the source repository of Andrew's tutorials: http://www.cs.cmu.edu/~awm/tutorials . Comments and corrections gratefully received.

**Ronald J. Williams**

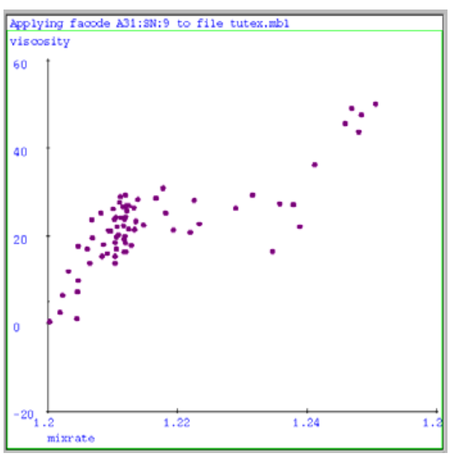
**CSG220**

**Spring 2007**

Adapted from parts of two Andrew Moore tutorials:
*Instance-Based Learning*
and
*Eight More Classic Machine Learning Algorithms*



# Starting Point




Simple, uni-variate case

***We've obtained some numeric data.***

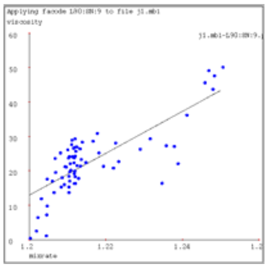
***How do we exploit it?***

General, multi variate case

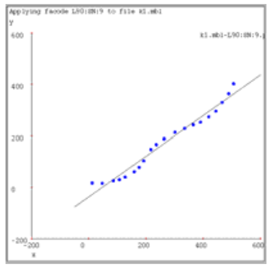
| Steel Temp | Line Speed | Slab Width | Temp Stage2 | Cool Setpt | Cool Gain |
|---|---|---|---|---|---|
| -0.4788 | 0.3849 | 0.0357 | 0.8384 | 0.6620 | -0.9512 |
| 0.3470 | -0.6488 | 0.6629 | -0.5087 | 0.6845 | -0.0647 |
| 0.8622 | -0.5367 | -0.2459 | -0.1438 | -0.7267 | -0.6119 |
| -0.2099 | -0.5975 | 0.0614 | -0.7648 | 0.0222 | -0.7623 |
| -0.4627 | 0.6218 | 0.9254 | 0.6081 | -0.8739 | -0.6439 |
| 0.7341 | 0.0745 | -0.2650 | -0.4510 | -0.4548 | 0.5753 |
| 0.4237 | -0.5143 | -0.0731 | 0.0385 | -0.8068 | 0.3098 |
| -0.1267 | 0.8105 | -0.6619 | -0.0768 | -0.8738 | -0.3367 |
| 0.0332 | 0.7754 | 0.7718 | -0.5440 | 0.6237 | 0.5113 |
| 0.2668 | 0.8777 | -0.5690 | -0.5151 | 0.6493 | 0.7705 |
| -0.5682 | 0.8457 | 0.5669 | -0.7359 | -0.3394 | 0.5880 |
| . | . | . | . | . | . |
| . | . | . | . | . | . |
| . | . | . | . | . | . |

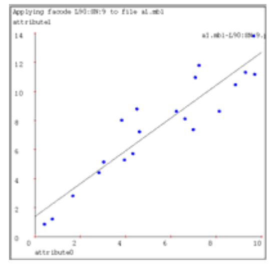
# Why not just use Linear Regression?



Here, linear regression manages to capture a significant trend in the data, but there is visual evidence of bias.



Here, linear regression appears to have a much better fit, but the bias is very clear.

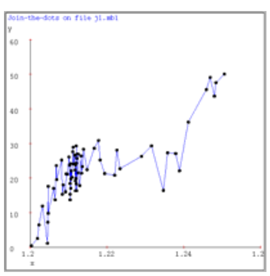


Here, linear regression may indeed be the right thing.

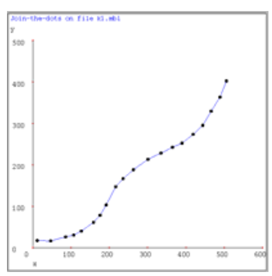
**Bias:** the underlying choice of model *(in this case, a line)* cannot, with any choice of parameters *(constant term and slope)* and with any amount of data *(the dots)* capture the full relationship.



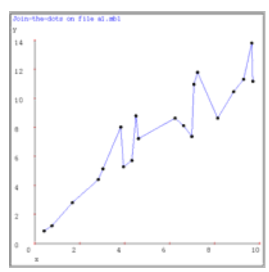
# Why not just Join the Dots?



Here, joining the dots is clearly fitting noise.



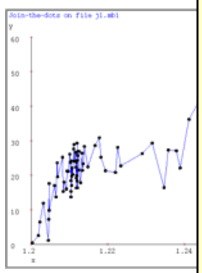
Here, joining the dots looks very sensible.



Again, a clear case of noise fitting.

Why is fitting the noise so bad?

# Why n



- You will tend to make somewhat bigger prediction errors on new data than if you filtered the noise perfectly.
- You don't get good gradient estimates or noise estimates.
- You can't make sensible confidence intervals.
- It's morally wrong.
- **Also:** Join the dots is *much harder* to implement for multivariate inputs.

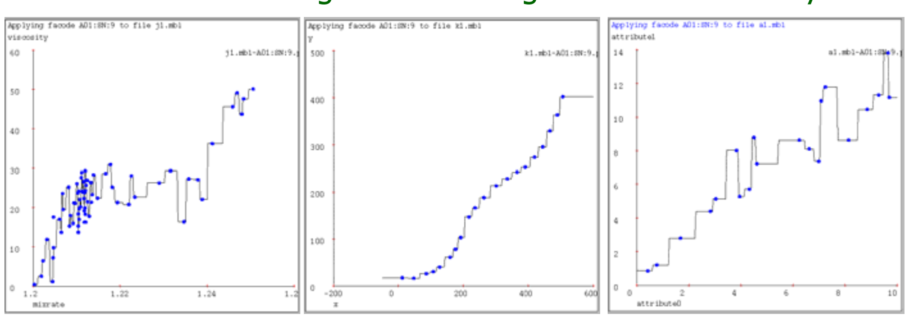
Here, joining the clearly fitting noise. l nsible. noise fitting.

Why is fitting the noise so bad?



# One-Nearest Neighbor

…One nearest neighbor for fitting is described shortly…



**Similar to Join The Dots with two Pros and one Con.**

- PRO: It is easy to implement with multivariate inputs.
- CON: It no longer interpolates locally.
- PRO: An excellent introduction to instance-based learning…

# Univariate 1-Nearest Neighbor

Given datapoints $(x_1,y_1)$ $(x_2,y_2)$..$(x_N,y_N)$,where we assume $y_i=f(s_i)$ for some unknown function $f$.

Given query point $x_q$, your job is to predict $\hat{y}=\hat{f}(x_q)$.

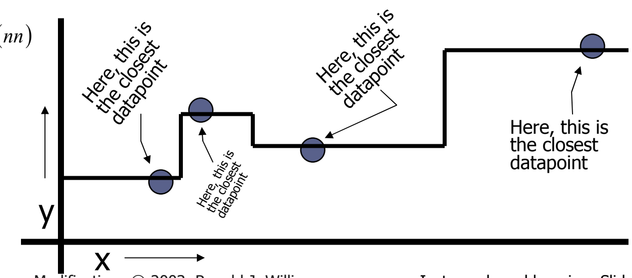
Nearest Neighbor:

1. Find the closest $x_i$ in our set of datapoints

$$i(nn)=\underset{i}{\operatorname{argmin}}\left|x_i - x_q\right|$$

2. Predict $\hat{y}=y_{i(nn)}$

Here's a dataset with one input, one output and four datapoints.



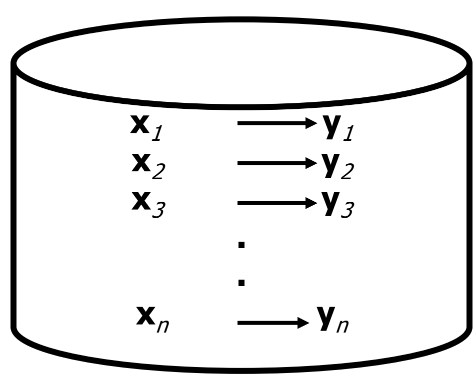




## *1-Nearest Neighbor is an example of….*
## **Instance-based learning**

A function approximator that has been around since about 1910.

To make a prediction, search database for similar datapoints, and fit with the local points.




**Four things make a memory based learner:**

- A distance metric
- How many nearby neighbors to look at?
- A weighting function (optional)
- How to fit with the local points?

# Nearest Neighbor

**Four things make a memory based learner:**

1. *A distance metric*
   **Euclidian**
2. *How many nearby neighbors to look at?*
   **One**
3. *A weighting function (optional)*
   **Unused**
4. *How to fit with the local points?*
   **Just predict the same output as the nearest neighbor.**

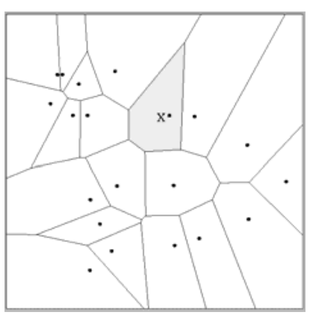
 

# Multivariate Distance Metrics

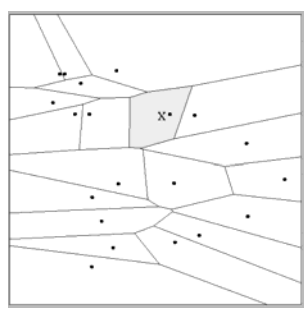
Suppose the input vectors x1, x2, …xn are two dimensional:

$\mathbf{x}_1 = ( x_{11} , x_{12} ) , \mathbf{x}_2 = ( x_{21} , x_{22} ) , \ldots \mathbf{x}_N = ( x_{N1} , x_{N2} )$.

One can draw the nearest-neighbor regions in input space.




$Dist(\mathbf{x}_i,\mathbf{x}_j) = (x_{i1} - x_{j1})^2 + (x_{i2} - x_{j2})^2$




$Dist(\mathbf{x}_i,\mathbf{x}_j) = (x_{i1} - x_{j1})^2 + (3x_{i2} - 3x_{j2})^2$

The relative scalings in the distance metric affect region shapes.

# Euclidean Distance Metric

$$D(\mathrm{x},\mathrm{x}') = \sqrt{\sum_i \sigma_i^{\,2}\left(x_i - x'_i\right)^2}$$

Or equivalently,

$$D(\mathrm{x},\mathrm{x}') = \sqrt{(\mathrm{x}-\mathrm{x}')^T \sum (\mathrm{x}-\mathrm{x}')}$$

where

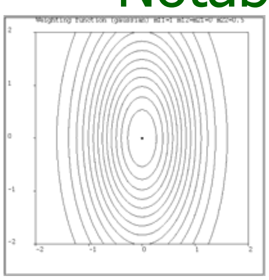
$$\sum = \begin{bmatrix} \sigma_1^2 & 0 & \cdots & 0 \\ 0 & \sigma_2^2 & \cdots & 0 \\ \cdots & \cdots & \cdots & \cdots \\ 0 & 0 & \cdots & \sigma_N^2 \end{bmatrix}$$

Other Metrics…

- Mahalanobis, Rank-based, Correlation-based (Stanfill+Waltz, Maes' Ringo system…)

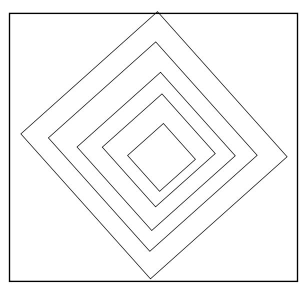


# Notable Distance Metrics

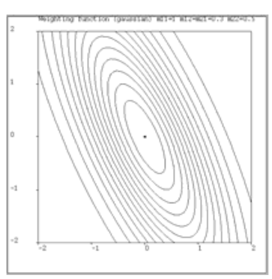


**Scaled Euclidian ($L_2$)**

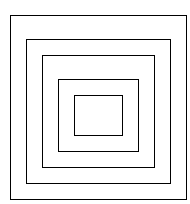
**$L_1$ norm (absolute)**



**Mahalanobis (here, $\Sigma$ on the previous slide is not necessarily diagonal, but is symmetric**

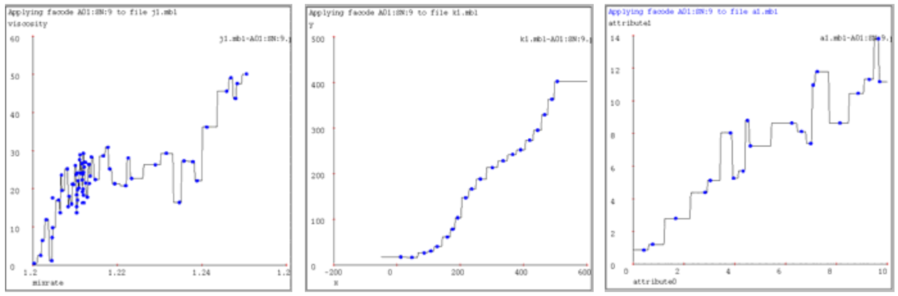


**$L_{infinity}$ *(max)* norm**



..let's leave distance metrics for now, and go back to….

## One-Nearest Neighbor



**Objection:**

That noise-fitting is really objectionable.

What's the most obvious way of dealing with it?

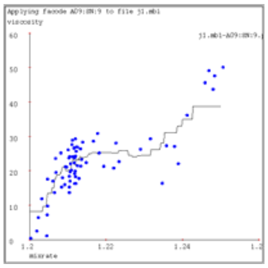



## k-Nearest Neighbor

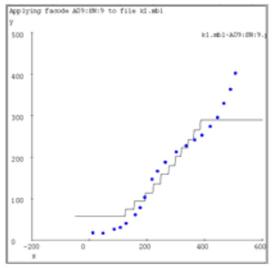
**Four things make a memory based learner:**

1. *A distance metric*
   **Euclidian**
2. *How many nearby neighbors to look at?*
   **k**
3. *A weighting function (optional)*
   **Unused**
4. *How to fit with the local points?*
   **Just predict the average output among the k nearest neighbors.**

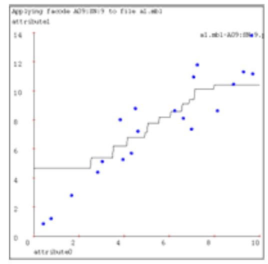
# k-Nearest Neighbor (here k=9)



A magnificent job of noise-smoothing. Three cheers for 9-nearest-neighbor.
But the lack of gradients and the jerkiness isn't good.



Appalling behavior! Loses all the detail that join-the-dots and 1-nearest-neighbor gave us, yet smears the ends.



Fits much less of the noise, captures trends. But still, frankly, pathetic compared with linear regression.

**K-nearest neighbor for function fitting smoothes away noise, but there are clear deficiencies.**

What can we do about all the discontinuities that k-NN gives us?

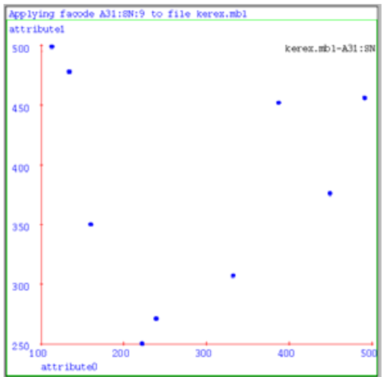



# Kernel Regression

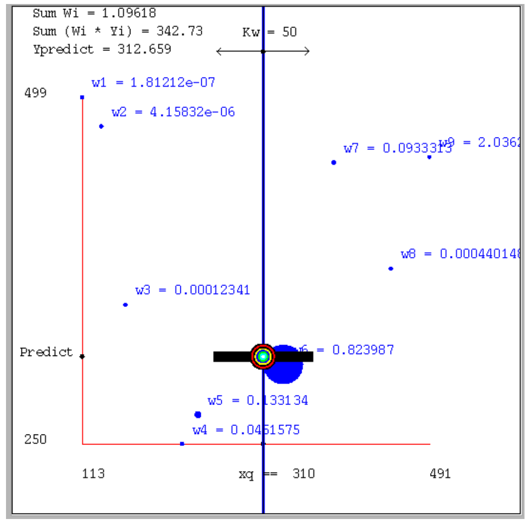
**Four things make a memory based learner:**

1. *A distance metric*
   Scaled Euclidian
2. *How many nearby neighbors to look at?*
   All of them
3. *A weighting function (optional)*
   $w_i = \exp(-D(x_i, query)^2 / K_w^2)$
   Nearby points to the query are weighted strongly, far points weakly. The $K_W$ parameter is the **Kernel Width**. Very important.
4. *How to fit with the local points?*
   Predict the weighted average of the outputs:
   predict = $\Sigma w_i y_i / \Sigma w_i$

# Kernel Regression in Pictures



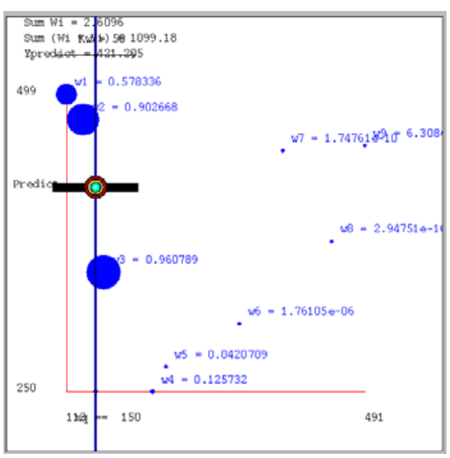
Take this dataset…

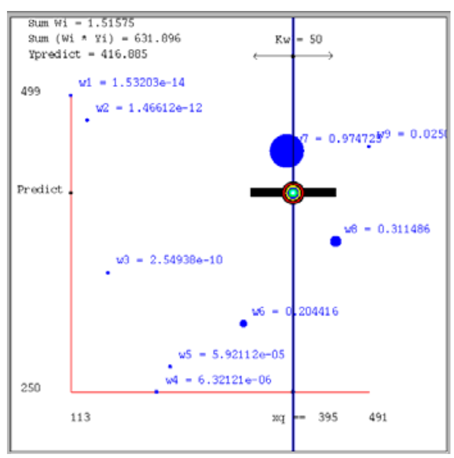



..and do a kernel prediction with $x_q$ (query) = 310, $K_w$ = 50.



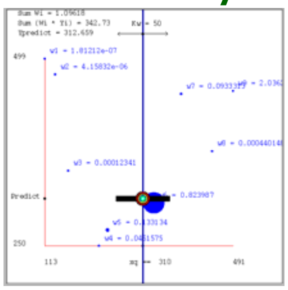
# Varying the Query



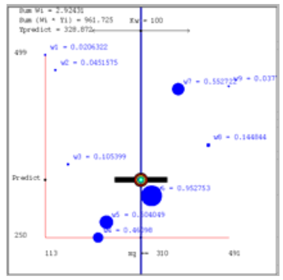

$x_q$ = 150    $x_q$ = 395

# Varying the kernel width



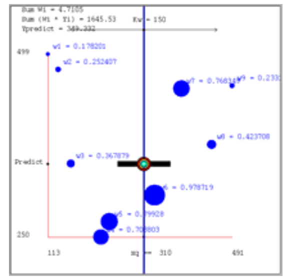
$x_q = 310$

$K_W = 50$ (see the double arrow at top of diagram)



$x_q = 310$ (the same)

$K_W = 100$



$x_q = 310$ (the same)

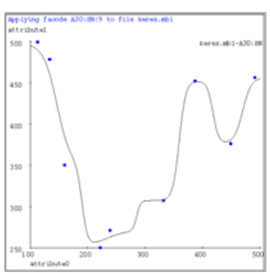
$K_W = 150$

**Increasing the kernel width $K_w$ means further away points get an opportunity to influence you.**

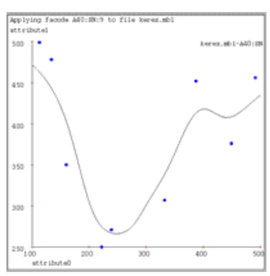
As $K_w \rightarrow$ infinity, the prediction tends to the global average.

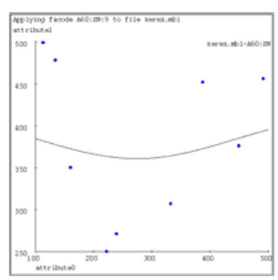
# Kernel Regression Predictions



$K_W$=10



$K_W$=20



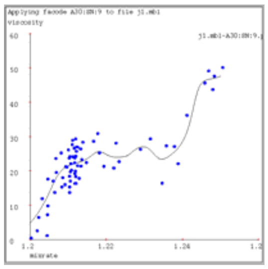
$K_W$=80

**Increasing the kernel width $K_w$ means further away points get an opportunity to influence you.**

As $K_w \rightarrow$ infinity, the prediction tends to the global average.

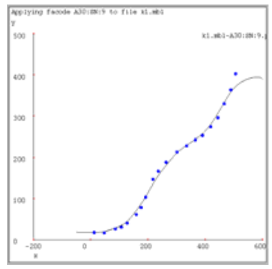
 

# Kernel Regression on our test cases



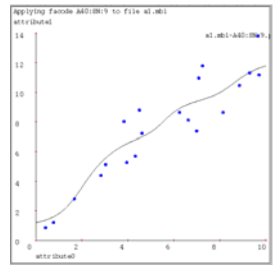
KW=1/32 of x-axis width.

It's nice to see a smooth curve at last. But rather bumpy. If Kw gets any higher, the fit is poor.



KW=1/32 of x-axis width.

Quite splendid. Well done, kernel regression. The author needed to choose the right $K_W$ to achieve this.



KW=1/16 axis width.

Nice and smooth, but are the bumps justified, or is this overfitting?

Choosing a good $K_w$ is important. Not just for Kernel Regression, but for all the locally weighted learners we're about to see.



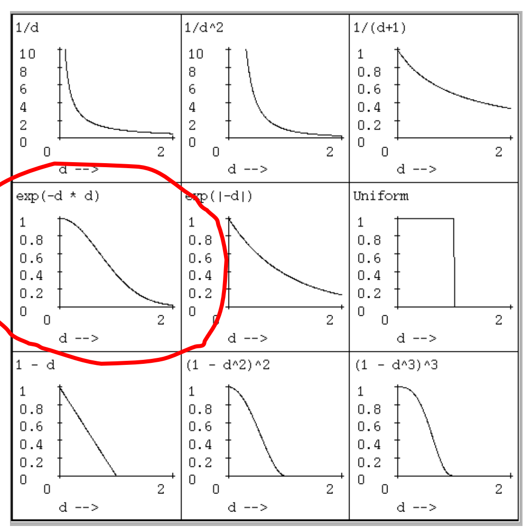
# Weighting functions

Let

$d = D(x_i, x_{query})/K_W$

Then here are some commonly used weighting functions…

(we use a Gaussian)



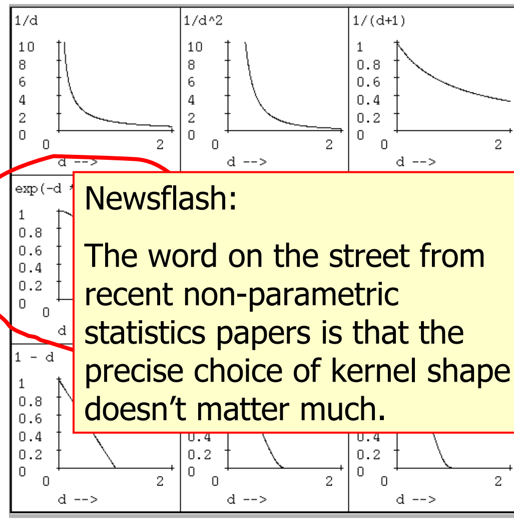
# Weighting functions

Let

$d = D(x_i, x_{query}) / K_W$

Then here are some commonly used weighting functions…

(we use a Gaussian)

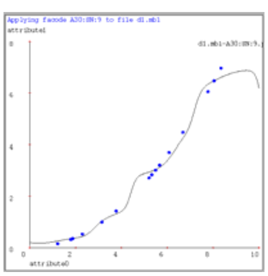



Newsflash:

The word on the street from recent non-parametric statistics papers is that the precise choice of kernel shape doesn't matter much.

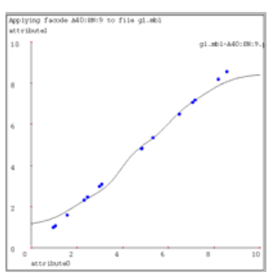



# Kernel Regression can look bad



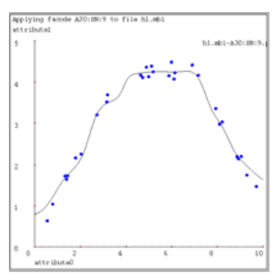
KW = Best.

Clearly not capturing the simple structure of the data.. Note the complete failure to extrapolate at edges.



KW = Best.

Also much too local. Why wouldn't increasing KW help? Because then it would all be "smeared".



KW = Best.

Three noisy linear segments. But best kernel regression gives poor gradients.

**Time to try something more powerful…**

# *Locally Weighted Regression*

**Kernel Regression:**

Take a very very conservative function approximator called AVERAGING. Locally weight it.

**Locally Weighted Regression:**

Take a conservative function approximator called LINEAR REGRESSION. Locally weight it.

*Let's Review Linear Regression….*

## Unweighted Linear Regression

You're lying asleep in bed. Then Nature wakes you.

**YOU**: "Oh. Hello, Nature!"

**NATURE**: "I have a coefficient β in mind. I took a bunch of real numbers called $x_1$, $x_2$ ..$x_N$ thus: $x_1$=3.1,$x_2$=2, …$x_N$=4.5.

For each of them ($k=1,2,..N$), I generated $y_k = \beta x_k + \varepsilon_k$

where $\varepsilon_k$ is a Gaussian (i.e. Normal) random variable with mean 0 and standard deviation σ. The $\varepsilon_k$'s were generated independently of each other.

Here are the resulting $y_i$'s: $y_1$=5.1 , $y_2$=4.2 , …$y_N$=10.2"

**You:** "Uh-huh."

**Nature:** "So what do you reckon β is then, eh?"

**WHAT IS YOUR RESPONSE?**

# Global Linear Regression: $y_k=\beta x_k+\varepsilon_k$

$$\text{prob}(y_k \| x_k, \beta) \sim \text{Gaussian, mean } \beta x_k, \text{ std. dev. } \sigma$$

$$\text{prob}(y_k \| x_k, \beta) = K \exp\left(\frac{-(y_k-\beta x_k)^2}{2\sigma^2}\right)$$

$$\text{prob}(y_1, y_2, \ldots y_N \| x_1, x_2, \ldots, x_N, \beta) = \prod_{k=1}^{N} K \exp\left(\frac{-(y_k-\beta x_k)^2}{2\sigma^2}\right)$$

Which value of β makes the $y_1$, $y_2$..$y_N$ values most likely?

$$\hat{\beta} = \underset{\beta}{\arg\max}\ \text{prob}(y_1, y_2, \ldots y_N \| x_1, x_2, \ldots, x_N, \beta)$$

$$= \underset{\beta}{\arg\max}\ \log \text{prob}(y_1, y_2, \ldots y_N \| x_1, x_2, \ldots, x_N, \beta)$$

$$= \underset{\beta}{\arg\max}\ N \log K - \frac{1}{2\sigma^2} \sum_{k=1}^{N} (y_k - \beta x_k)^2$$

$$= \underset{\beta}{\arg\min} \sum_{k=1}^{N} (y_k - \beta x_k)^2$$



# Least squares unweighted linear regression

Write $E(\beta) = \sum_k (y_k - \beta x_k)^2$, so $\hat{\beta} = \underset{\beta}{\arg\min}\ E(\beta)$

To minimize $E(\beta)$, set

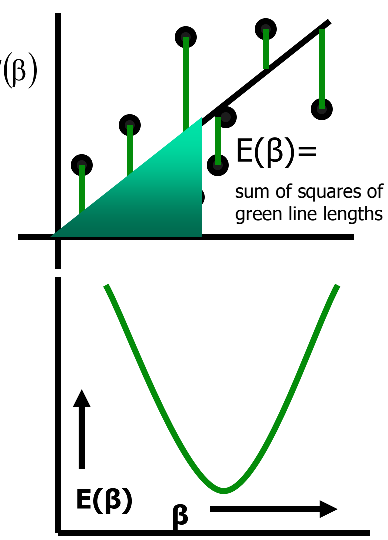
$$\frac{\partial}{\partial \beta} E(\beta) = 0$$

so

$$0 = \frac{\partial}{\partial \beta} E(\beta) = -2\sum_k x_k y_k + 2\beta \sum_k x_k^2$$

giving

$$\hat{\beta} = \left(\sum_k x_k^2\right)^{-1} \sum_k x_k y_k$$



## Multivariate unweighted linear regression

Nature supplies $N$ input vectors. Each input vector $x_k$ is $D$-dimensional: $\mathbf{x}_k = ( x_{k1}, x_{k2} .. x_{kD} )$ . Nature also supplies N corresponding output values $y_1$ .. $y_N$.

$$X = \begin{bmatrix} x_{11} & x_{12} & \cdots & x_{1D} \\ x_{21} & x_{22} & \cdots & x_{2D} \\ \vdots & \vdots & & \vdots \\ x_{N1} & x_{N2} & \cdots & x_{ND} \end{bmatrix} \quad Y = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_N \end{bmatrix} \quad \text{we are told } y_k = \left( \sum_{j=1}^{D} \beta_j x_{kj} \right) + \varepsilon_k$$

We must estimate β = ($\beta_1$, $\beta_2$ … $\beta_D$). It's easily shown using matrices instead of scalars on the previous slide that

$$\hat{\beta} = \left(X^T X\right)^{-1} X^T Y$$

Note that $X^T X$ is a D x D positive definite symmetric matrix, and $X^T Y$ is a D x 1 vector:
$$\left(X^T X\right)_{ij} = \sum_{k=1}^{N} x_{ki} x_{kj} \qquad \left(X^T Y\right)_i = \sum_{k=1}^{N} x_{ki} y_i$$




## The Pesky Constant Term

**Now:** Nature doesn't guarantee that the line/hyperplane passes through the origin.

**In other words:** Nature says

$$y_k = \beta_0 + \left( \sum_{j=1}^{D} \beta_j x_{kj} \right) + \varepsilon_k$$

"No problem," you reply. "Just add one extra input variable, $x_{k0}$, which is always 1"

$$\begin{bmatrix} x_{11} & x_{12} & \cdots & x_{1D} \\ x_{21} & x_{22} & \cdots & x_{2D} \\ \vdots & \vdots & & \vdots \\ x_{N1} & x_{N2} & \cdots & x_{ND} \end{bmatrix} \rightarrow \begin{bmatrix} 1 & x_{11} & x_{12} & \cdots & x_{1D} \\ 1 & x_{21} & x_{22} & \cdots & x_{2D} \\ \vdots & \vdots & \vdots & & \vdots \\ 1 & x_{N1} & x_{N2} & \cdots & x_{ND} \end{bmatrix}$$

# Locally Weighted Regression

Four things make a memory-based learner:

1. *A distance metric*

   Scaled Euclidian

2. *How many nearby neighbors to look at?*

   All of them

3. *A weighting function (optional)*

   $w_k = exp(-D(x_k, x_{query})^2 / K_w^2)$

   Nearby points to the query are weighted strongly, far points weakly. The $K_w$ parameter is the **Kernel Width**.

4. *How to fit with the local points?*

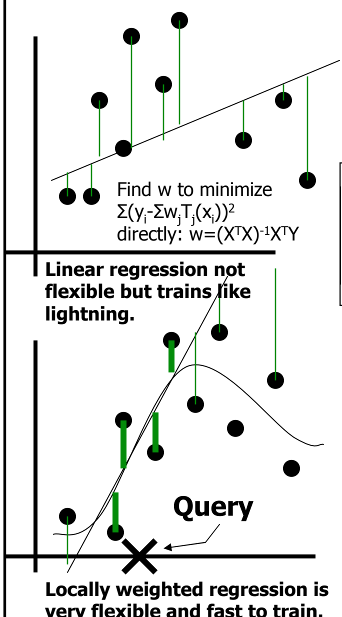
   First form a local linear model. Find the $\underline{\beta}$ that minimizes the locally weighted sum of squared residuals:

$$\hat{\beta} = \underset{\beta}{\operatorname{argmin}} \sum_{k=1}^{N} w_k^2 \left(y_k - \beta^T x_k\right)^2$$

Then predict $y_{predict} = \underline{\beta}^T x_{query}$



## How LWR works




1. For each point ($\mathbf{x}_k, y_k$) compute $w_k$.
2. Let $WX = Diag(w_1,..w_N)X$

$$\begin{bmatrix} x_{11} & x_{12} & \cdots & x_{1D} \\ x_{21} & x_{22} & \cdots & x_{2D} \\ \vdots & \vdots & & \vdots \\ x_{N1} & x_{N2} & \cdots & x_{ND} \end{bmatrix} \rightarrow \begin{bmatrix} w_1 & w_1 x_{11} & w_1 x_{12} & \cdots & w_1 x_{1D} \\ w_2 & w_2 x_{21} & w_2 x_{22} & \cdots & w_2 x_{2D} \\ \vdots & \vdots & \vdots & & \vdots \\ w_N & w_N x_{N1} & w_N x_{N2} & \cdots & w_N x_{ND} \end{bmatrix}$$

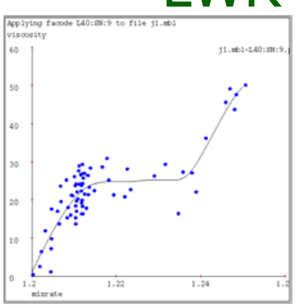
*X* **- - >** *WX*

3. Let $WY = Diag(w_1,..w_N)Y$, so that $y_k \rightarrow w_k y_k$
4. $\beta = (WX^T WX)^{-1}(WX^T WY)$

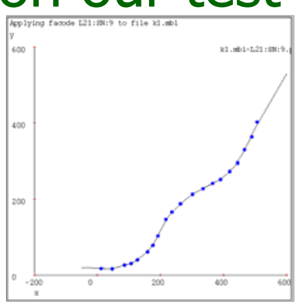
```
Input X matrix of inputs: X[k] [i] = i'th component of k'th input point.
Input Y matrix of outputs: Y[k] = k'th output value.
Input xq = query input.  Input kwidth.

WXTWX = empty (D+1) x (D+1) matrix
WXTWY = empty (D+1) x 1        matrix

for ( k = 1 ; k <= N ; k = k + 1 )
    /* Compute weight of kth point  */
    wk = weight_function( distance( xq , X[k] ) / kwidth )

    /* Add to (WX) ^T (WX) matrix */
    for ( i = 0 ; i <= D ; i = i + 1 )
           for ( j = 0 ; j <= D ; j = j + 1 )
                     if ( i == 0 ) xki = 1 else xki = X[k] [i]
                     if ( j == 0 ) xkj = 1 else xkj = X[k] [j]
                     WXTWX [i] [j] = WXTWX [i] [j] + wk * wk * xki * xkj

    /*  Add to (WX) ^T (WY) vector */
    for ( i = 0 ; i <= D ; i = i + 1 )
            if ( i == 0 ) xki = 1 else xki = X[k] [i]
            WXTWY [i] = WXTWY [i] + wk * wk * xki * Y[k]

/* Compute the local beta.  Call your favorite linear equation solver.  Recommend Cholesky
    Decomposition for speed.  Recommend Singular Val Decomp for Robustness. */
beta = (WXTWX)^-1 (WXTWY)
ypredict = beta[0] + beta[1]*xq[1] + beta[2]*xq[2] + … beta[D]*xq[D]
```

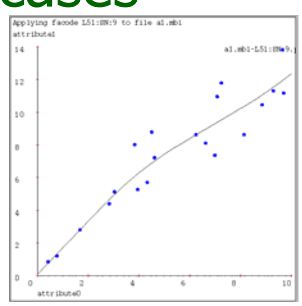
# LWR on our test cases



KW = 1/16 of x-axis width.



KW = 1/32 of x-axis width.

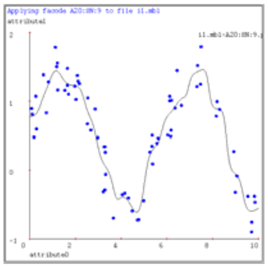


KW = 1/8 of x-axis width.

Nicer and smoother, but even now, are the bumps justified, or is this overfitting?

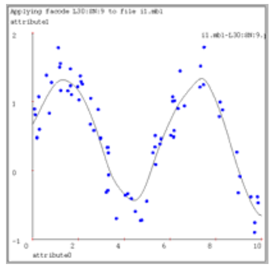
 

# Locally weighted Polynomial regression



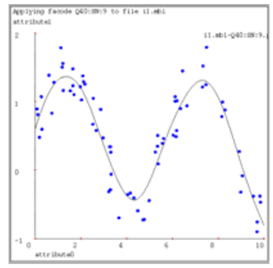
Kernel Regression
Kernel width $K_W$ at optimal level.

KW = 1/100 x-axis



LW Linear Regression
Kernel width $K_W$ at optimal level.

KW = 1/40 x-axis



LW Quadratic Regression
Kernel width $K_W$ at optimal level.

KW = 1/15 x-axis

Local quadratic regression is easy: just add quadratic terms to the WXTWX matrix. As the regression degree increases, the kernel width can increase without introducing bias.

# When's Quadratic better than Linear?

- It can let you use a wider kernel without introducing bias.
- Sometimes you want more than a prediction, you want an estimate of the local Hessian. Then quadratic is your friend!
- But in higher dimensions is appallingly expensive, and needs a lot of data. (Why?)
- Two "Part-way-between-linear-and-quadratic" polynomials:
  - "Ellipses": Add $x_i^2$ terms to the model, but not cross-terms (no $x_i x_j$ where $i=j$)
  - "Circles": Add only one extra term to the model:

$$x_{D+1} = \sum_{j=1}^{D} x_j^2$$

- Incremental insertion of polynomial terms is well established in conventional regression (GMDH,AIM): potentially useful here too

# Locally Weighted Learning: Variants

- Range Searching: Average of all neighbors within a given range
- Range-based linear regression: Linear regression on all points within a given range
- Linear Regression on K-nearest-neighbors
- Weighting functions that decay to zero at the kth nearest neighbor
- Locally weighted Iteratively Reweighted Least Squares
- Locally weighted Logistic Regression
- Locally weighted classifiers
- Multilinear Interpolation
- Kuhn-Triangulation-based Interpolation
- Spline Smoothers

# Local Weighted Learning: Pros & Cons vs Neural Nets

**Local weighted learning has some advantages:**

- Can fit low dimensional, very complex, functions very accurately. Neural nets require considerable tweaking to do this.
- You can get meaningful confidence intervals, local gradients back, not merely a prediction.
- Training, adding new data, is almost free.
- "One-shot" learning---not incremental
- Variable resolution.
- Doesn't forget old training data unless statistics warrant.
- Cross-validation is cheap

**Neural Nets have some advantages:**

- With large datasets, MBL predictions are slow (although kdtree approximations, and newer cache approximations help a lot).
- Neural nets can be trained directly on problems with hundreds or thousands of inputs (e.g. from images). MBL would need someone to define a smaller set of image features instead.
- Nets learn incrementally.

# Radial Basis Functions (RBFs)

| $X_1$ | $X_2$ | $Y$ |
|---|---|---|
| 3 | 2 | 7 |
| 1 | 1 | 3 |
| : | : | : |

**X**=

| | |
|---|---|
| 3 | 2 |
| 1 | 1 |
| : | : |

**y**=

| |
|---|
| 7 |
| 3 |
| : |

Not necessarily instance-based, but can be.

Clearly related in any case.

**Z**=

| | | | | | |
|---|---|---|---|---|---|
| … | … | … | … | … | … |
| … | … | … | … | … | … |
| … | … | … | … | … | … |

**y**=

| |
|---|
| 7 |
| 3 |
| : |

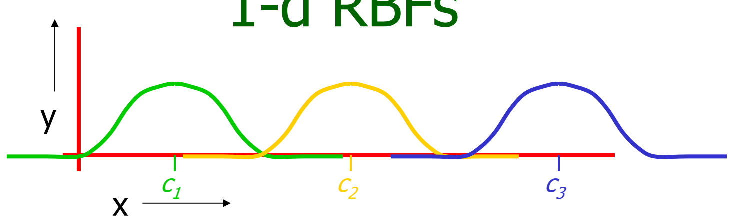
***z***=*(list of radial basis function evaluations)*

$\beta=(\mathbf{Z}^T\mathbf{Z})^{-1}(\mathbf{Z}^T\mathbf{y})$

$y^{est} = \beta_0 + \beta_1 x_1 + \ldots$



# 1-d RBFs

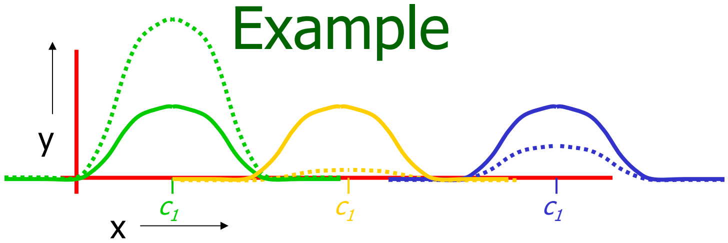



$y^{est} = \beta_1 \phi_1(x) + \beta_2 \phi_2(x) + \beta_3 \phi_3(x)$

where

$\phi_i(x) = KernelFunction( | x - c_i | / KW)$

# Example



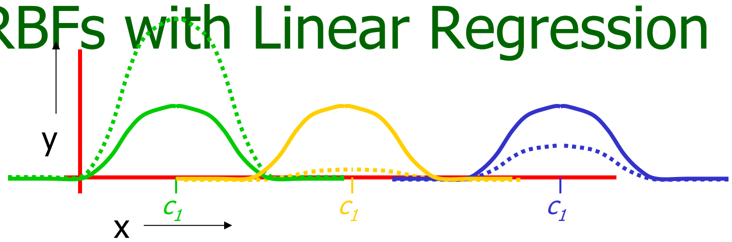

$y^{est} = 2\phi_1(x) + 0.05\phi_2(x) + 0.5\phi_3(x)$

where

$\phi_i(x) = KernelFunction(\,|\,x - c_i\,|\,/\,KW)$




# RBFs with Linear Regression




$y^{est} = 2\phi_1(x) + 0.05\phi_2(x) + 0.5\phi_3(x)$

where

$\phi_i(x) = KernelFunction(\,|\,x - c_i\,|\,/\,KW)$

# RBFs with Linear Regression

All $c_i$'s are held constant (initialized randomly or on a grid in m-dimensional input space)

*KW* also held constant (initialized to be large enough that there's decent overlap between basis functions*

*Usually much better than the crappy overlap on my diagram

$y^{est} = 2\phi_1(x) + 0.05\phi_2(x$ … $5\phi_3(x)$

where

$\phi_i(x) = KernelFunction(\,|\,x - c_i\,|\,/\,KW)$

then given $Q$ basis functions, define the matrix $Z$ such that $Z_{kj} = KernelFunction(\,|\,x_k - c_i\,|\,/\,KW)$ where $x_k$ is the kth vector of inputs

And as before, $\beta = (\mathbf{Z}^T\mathbf{Z})^{-1}(\mathbf{Z}^T\mathbf{y})$



# RBFs with NonLinear Regression

Allow the $c_i$'s to adapt to the data (initialized randomly or on a grid in m-dimensional input space)

*KW* allowed to adapt to the data. (Some folks even let each basis function have its own $KW_j$, permitting fine detail in dense regions of input space)

$y^{est} = 2\phi_1(x) + 0.05\phi_2(x$ … $5\phi_3(x)$

where

$\phi_i(x) = KernelFunction(\,|\,x - c_i\,|\,/\,KW)$

But how do we now find all the $\beta_j$'s, $c_i$'s and *KW*?

## RBFs with NonLinear Regression

Allow the $c_i$'s to adapt to the data (initialized randomly or on a grid in m-dimensional input space)

*KW* allowed to adapt to the data. (Some folks even let each basis function have its own $KW_j$, permitting fine detail in dense regions of input space)

$y^{est} = 2\phi_1(x) + 0.05\phi_2(x) \ldots 5\phi_3(x)$

where

$\phi_i(x) = KernelFunction(\,|\,x - c_i\,|\,/\,KW)$

But how do we now find all the $\beta_j$'s, $c_i$'s and $KW$?

Answer: Gradient Descent




## RBFs with NonLinear Regression

Allow the $c_i$'s to adapt to the data (initialized randomly or on a grid in m-dimensional input space)

*KW* allowed to adapt to the data. (Some folks even let each basis function have its own $KW_j$, permitting fine detail in dense regions of input space)

$y^{est} = 2\phi_1(x) + 0.05\phi_2(x) \ldots 5\phi_3(x)$

where

$\phi_i(x) = KernelFunction(\,|\,x - c_i\,|\,/\,KW)$

But how do we now find all the $\beta_j$'s, $c_i$'s and $KW$?

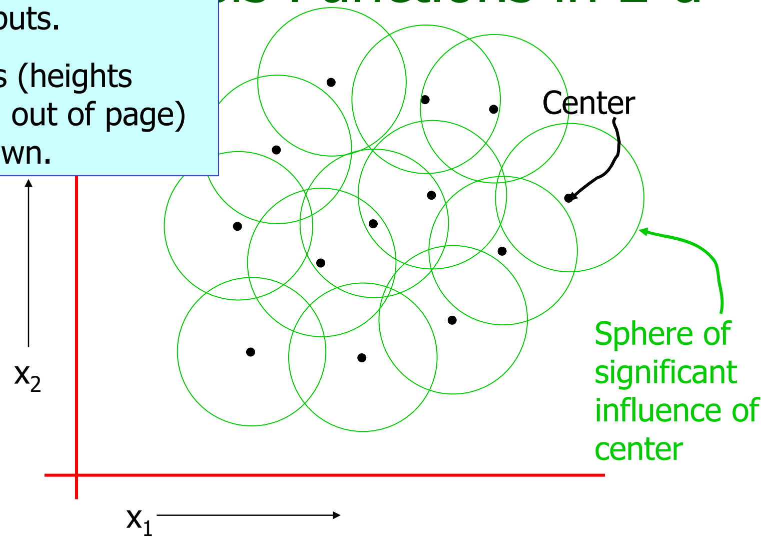
(But I'd like to see, or hope someone's already done, a hybrid, where the $c_i$'s and *KW* are updated with gradient descent while the $\beta_j$'s use matrix inversion)

Answer: Gradient Descent

# Radial Basis Functions in 2-d

Two inputs.

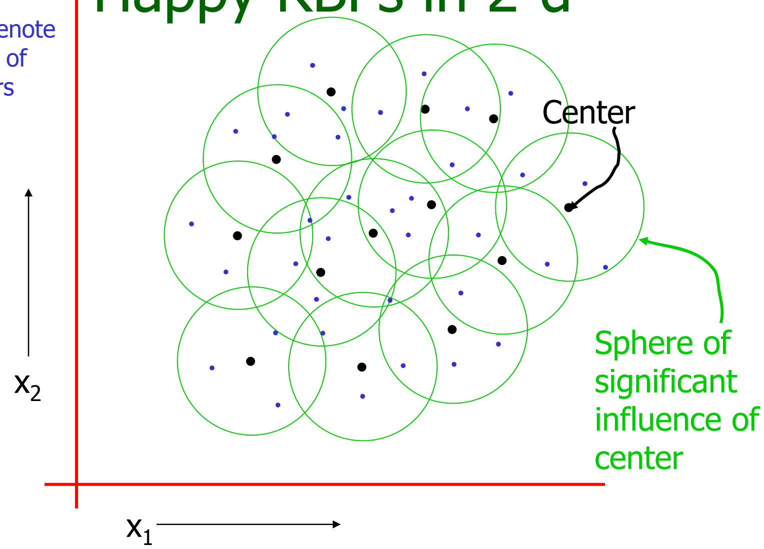
Outputs (heights sticking out of page) not shown.






# Happy RBFs in 2-d

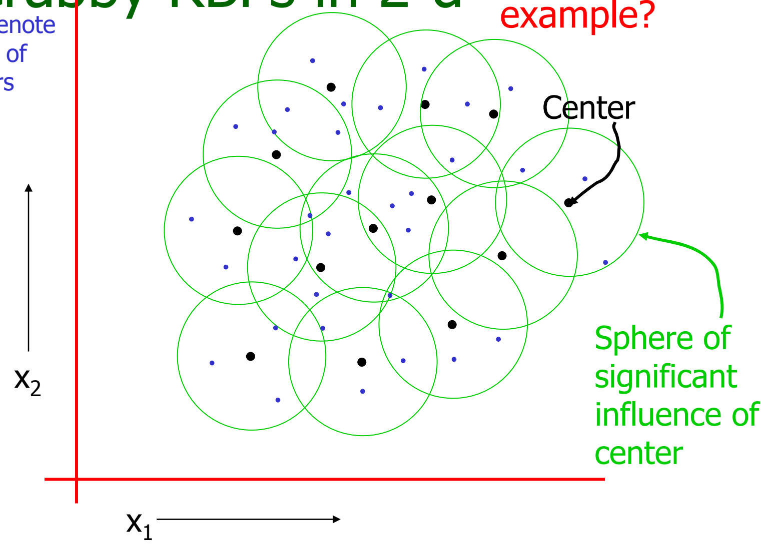
Blue dots denote coordinates of input vectors



What's the problem in this example?

# Crabby RBFs in 2-d

Blue dots denote coordinates of input vectors



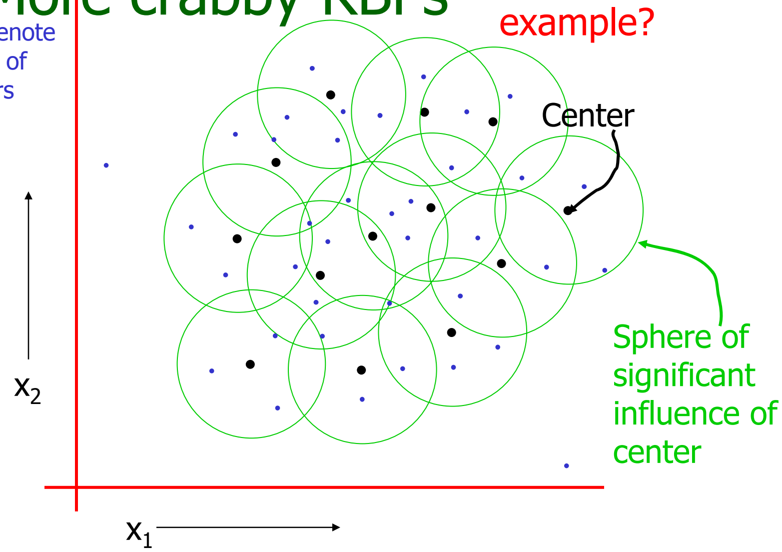


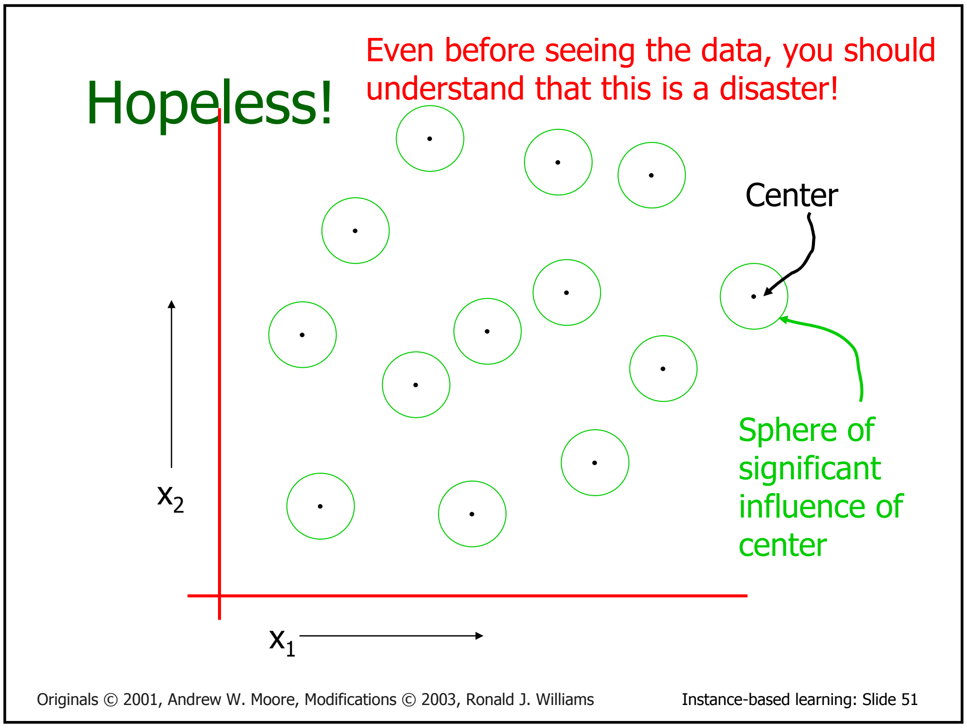


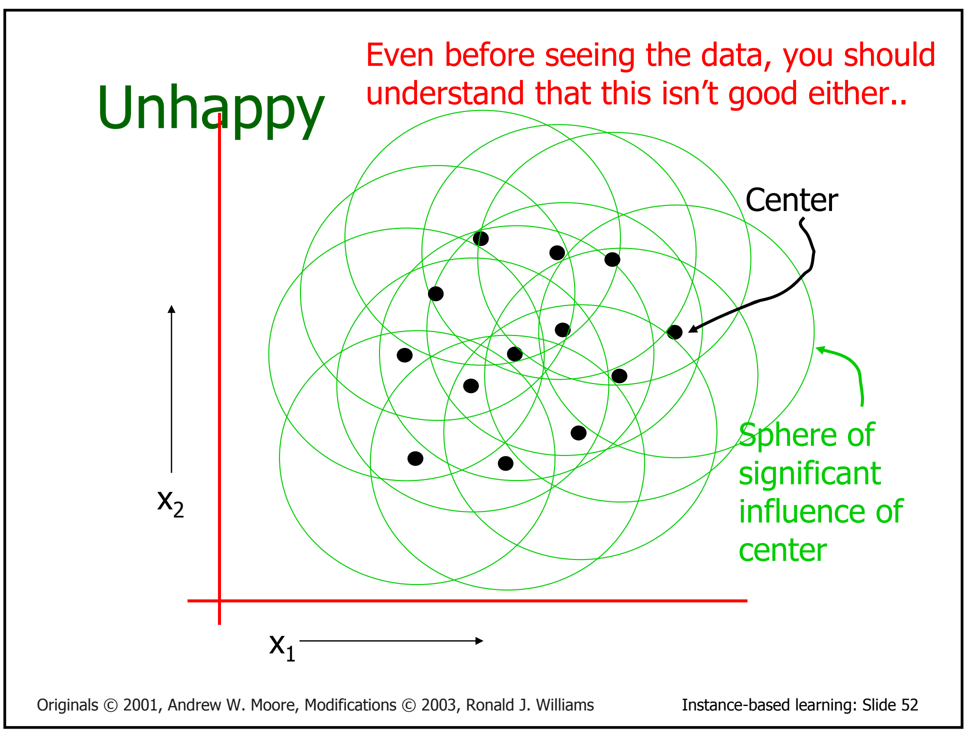

And what's the problem in this example?

# More crabby RBFs

Blue dots denote coordinates of input vectors




Hopeless!
Even before seeing the data, you should understand that this is a disaster!
Center
Sphere of significant influence of center
x2
x1
Originals © 2001, Andrew W. Moore, Modifications © 2003, Ronald J. Williams
Instance-based learning: Slide 51


Unhappy
Even before seeing the data, you should understand that this isn't good either..
Center
Sphere of significant influence of center
x2
x1
Originals © 2001, Andrew W. Moore, Modifications © 2003, Ronald J. Williams
Instance-based learning: Slide 52

## Instance-Based?

- Centers can be chosen to lie at data points
  - In that case almost like Kernel Regression, except no weight normalization
  - Sometimes choose centers to be a subset of the data points
- True instance-based considered *lazy method*
  - RBF is actually an *eager* method like
    - Decision Trees
    - Neural Nets
    - etc.



## What we have covered

- Problems of bias for unweighted regression, and noise-fitting for "join the dots" methods
- Nearest Neighbor and k-nearest neighbor regression
- Distance Metrics
- Kernel Regression
- Weighting functions
- Locally weighted regression: concept and implementation
- Multivariate Issues
- Other Locally Weighted variants
- Where to use locally weighted learning for modeling?
- Locally weighted pros and cons
- Radial Basis Functions