Neural Nets:

Many possible refs

e.g., Mitchell Chapter 4

Neural Networks

Machine Learning – 10701/15781

Carlos Guestrin

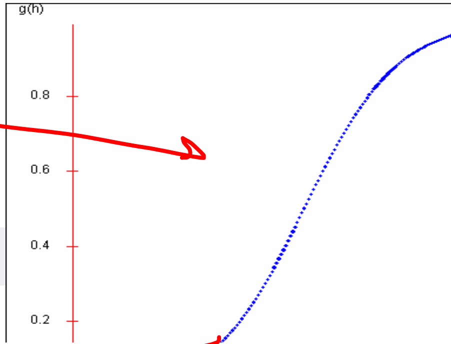
Carnegie Mellon University

February 15th, 2006

Announcements

- Recitations stay on Thursdays
  - 5-6:30pm in Wean 5409
  - This week: Cross Validation and Neural Nets
- **Homework 2**
  - Due next Monday, Feb. 20th
  - Updated version online with more hints
  - Start early

Logistic regression



logistic f. or sigmoid

- P(Y|X) represented by:

$$P(Y=1 \mid x, W) = \frac{1}{1+e^{-(w_0+\sum_i w_i x_i)}}$$

$$= g(w_0 + \sum_i w_i x_i)$$

- Learning rule – MLE:

$$\frac{\partial \ell(W)}{\partial w_i} = \sum_j x_i^j [y^j - P(Y^j = 1 \mid x^j, W)]$$

$$= \sum_j x_i^j [y^j - g(w_0 + \sum_i w_i x_i^j)]$$

optimize cond. Likelihood

$$w_i \leftarrow w_i + \eta \sum_j x_i^j \delta^j$$

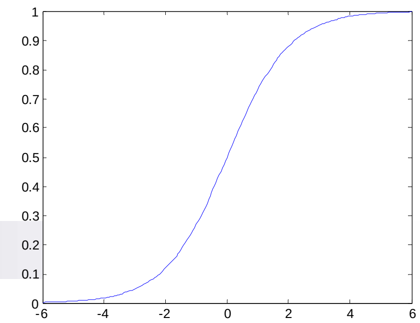
learn rate

feature

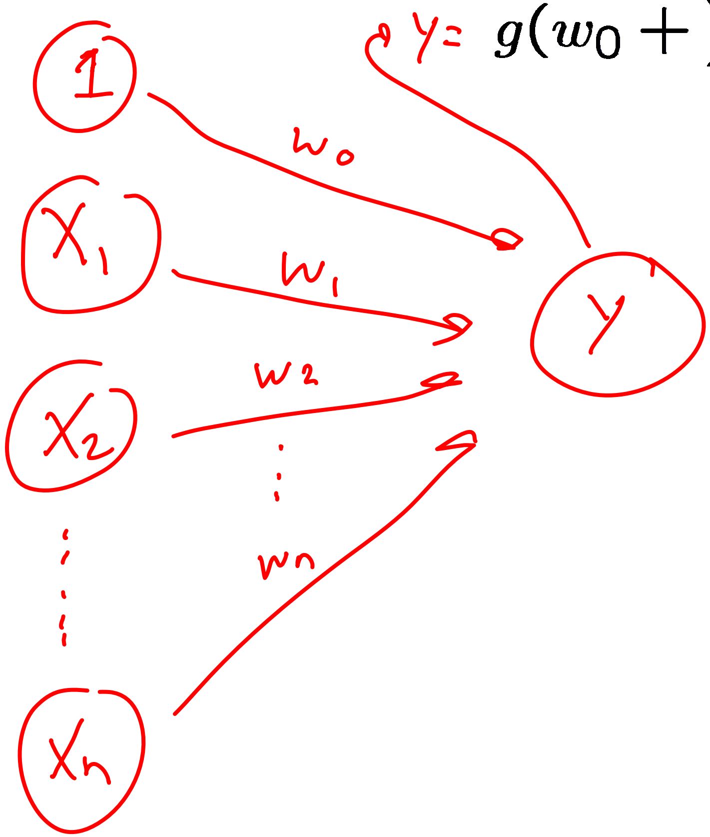
$$\delta^j = y^j - g(w_0 + \sum_i w_i x_i^j)$$

diff. true value classifier value

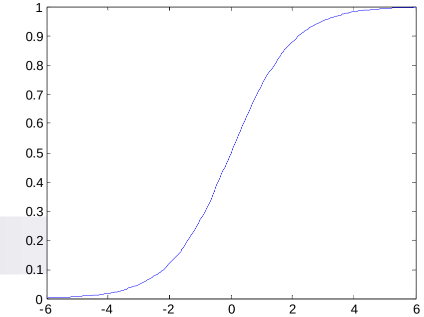
Perceptron as a graph



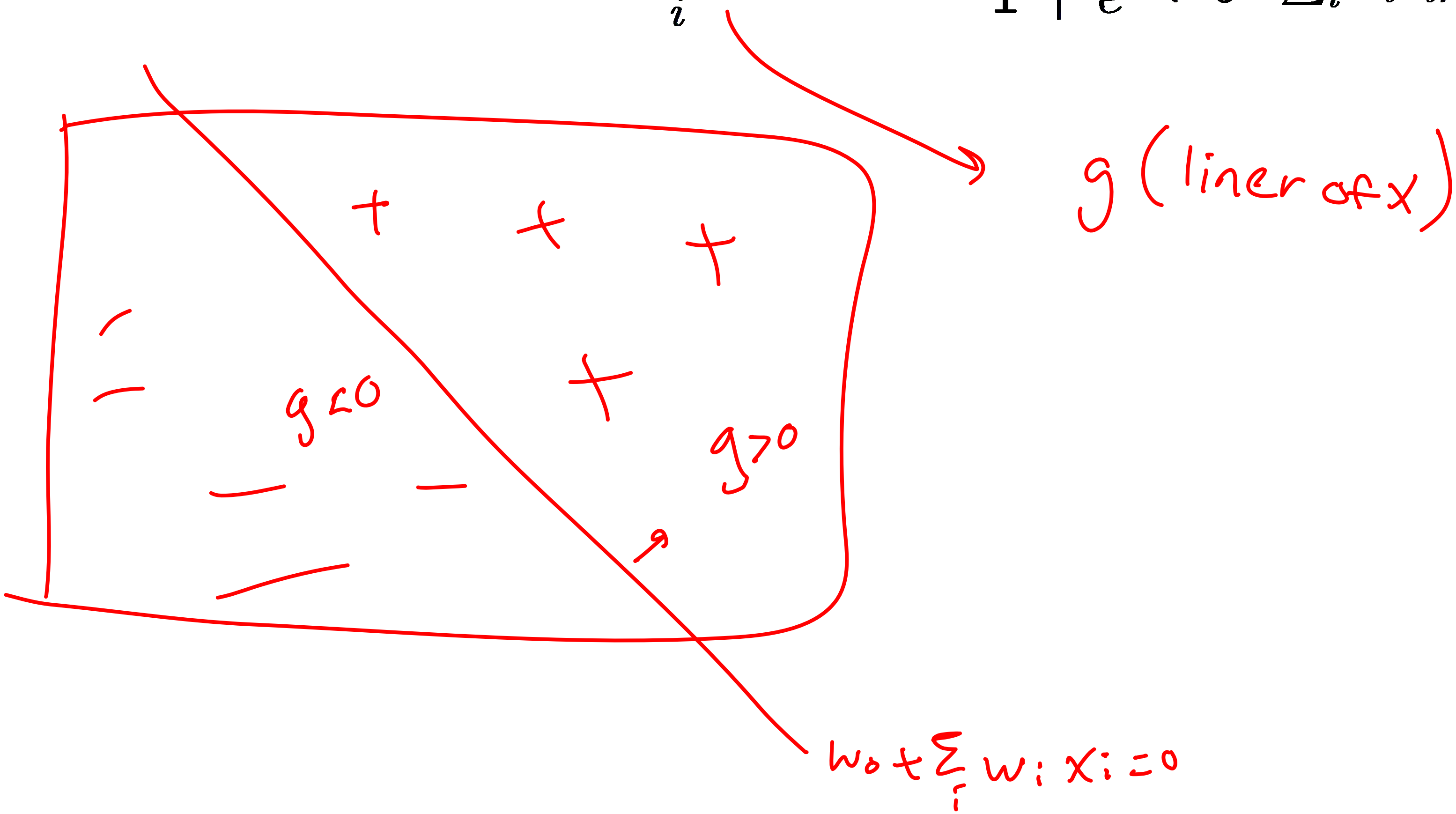
$$y = g(w_0 + \sum_i w_i x_i) = \frac{1}{1 + e^{-(w_0 + \sum_i w_i x_i)}}$$



Linear perceptron classification region



$$g(w_0 + \sum_i w_i x_i) = \frac{1}{1 + e^{-(w_0 + \sum_i w_i x_i)}}$$



The perceptron learning rule

$$w_i \leftarrow w_i + \eta \sum_j x_i^j \delta^j$$

learn rate, example, delta

perceptron

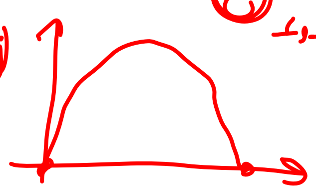
loss function: squared error

$$\delta^j = [y^j - g(w_0 + \sum_i w_i x_i^j)]g^j(1-g^j)$$

how well classify

extra term

$$g^j = g(w_0 + \sum_i w_i x_i^j)$$



more: unhappy with 50/50 classification

loss function: Cond. likelihood

logistic regression

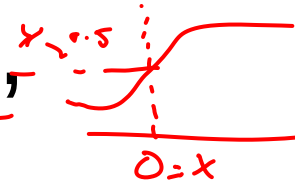
- Compare to MLE:

$$w_i \leftarrow w_i + \eta \sum_j x_i^j \delta^j$$

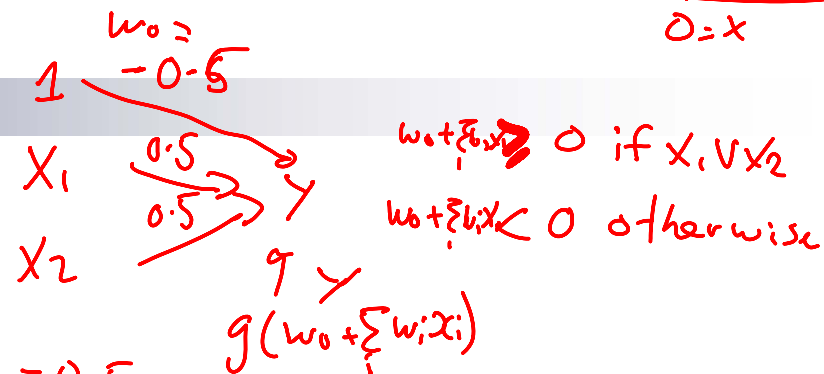
$$\delta^j = [y^j - g(w_0 + \sum_i w_i x_i^j)]$$

also unhappy with 50/50

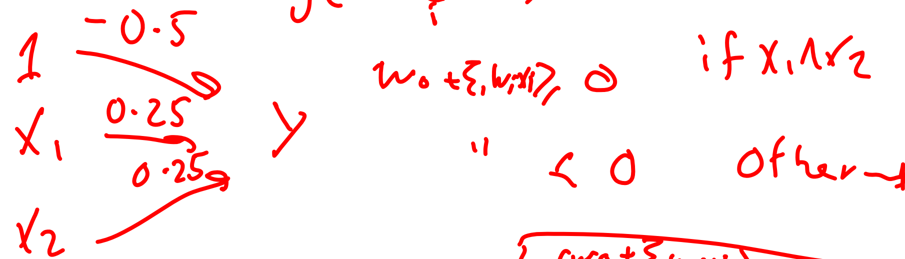
Percepton, linear classification, Boolean functions



- Can learn $x_1 \vee x_2$ (or)



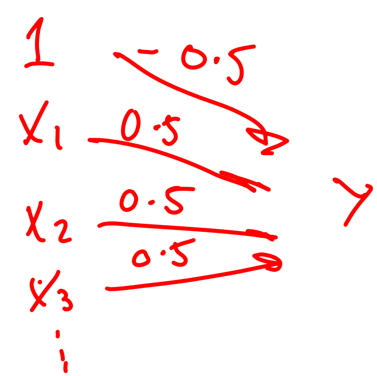
- Can learn $x_1 \wedge x_2$ (and)

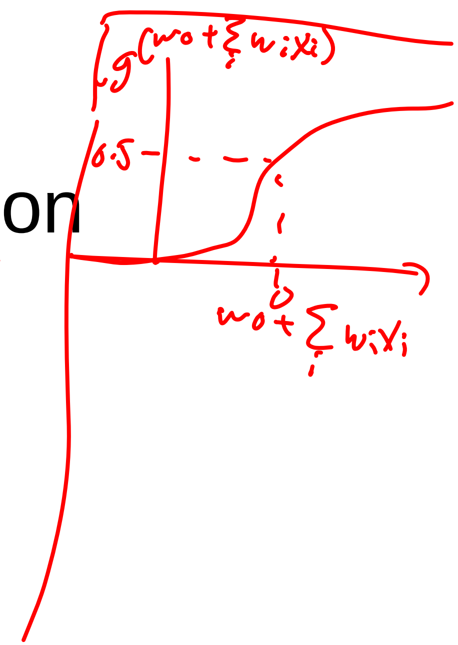


- Can learn any conjunction or disjunction

$x_1 \vee x_2 \vee x_3 \ldots$

disjunction

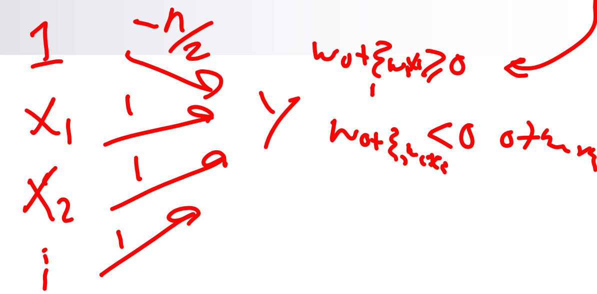




Percepton, linear classification, Boolean functions

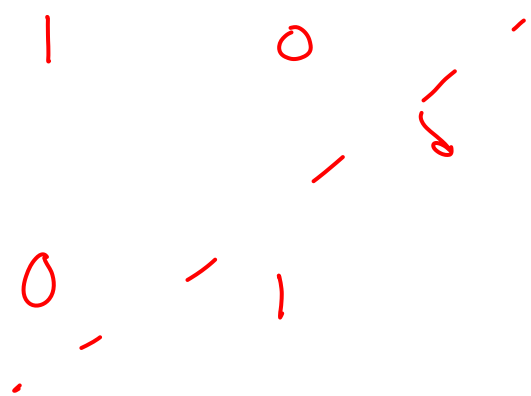
- Can learn majority

more than half xi are true :



- Can perceptrons do everything?

Cannot learn XOR



no st. line separating points

Going beyond linear classification

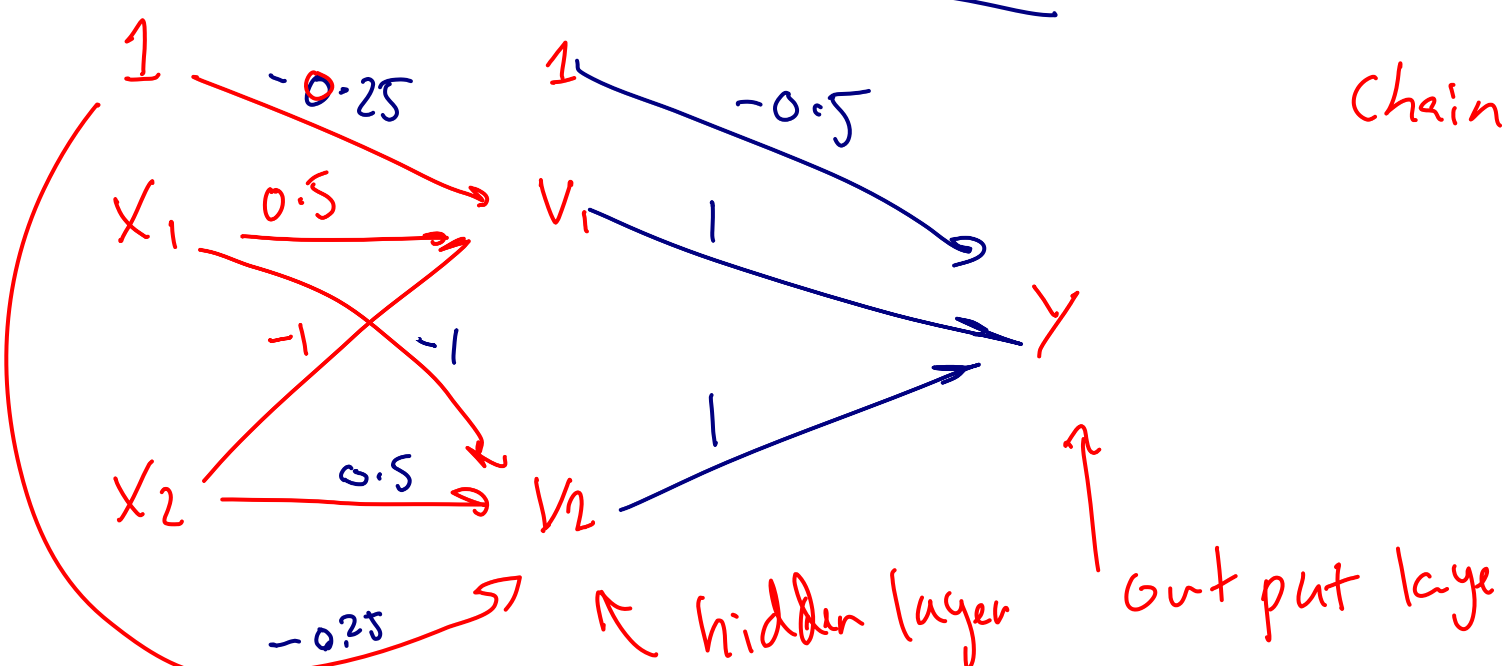
- Solving the XOR problem

input layer

$X_1 \text{ XOR } X_2 = X_1 \wedge \neg X_2 \vee \neg X_1 \wedge X_2$

$V_1 = X_1 \wedge \neg X_2$, $V_2 = \neg X_1 \wedge X_2$

$Y = V_1 \vee V_2$

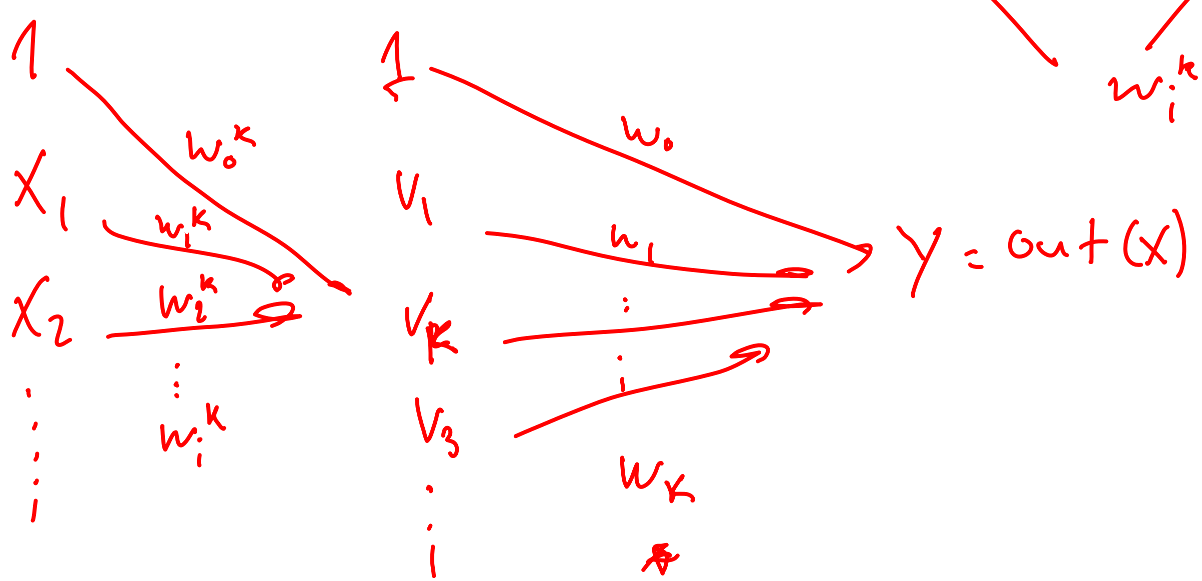


Chain perceptrons

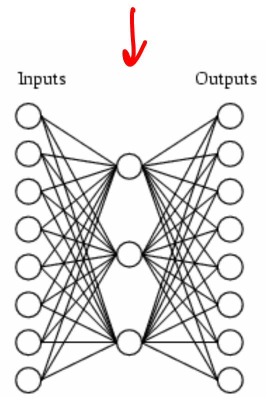
Hidden layer

- Perceptron: $$out(\mathbf{x}) = g(w_0 + \sum_i w_i x_i)$$
- 1-hidden layer:

$$out(\mathbf{x}) = g\left(w_0 + \sum_k w_k g(w_0^k + \sum_i w_i^k x_i)\right)$$



Example data for NN with hidden layer



A target function:

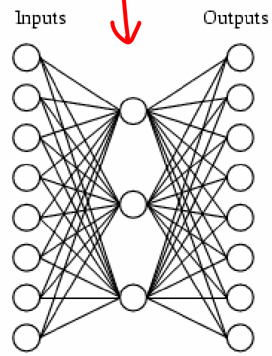
Input		Output
10000000	$\rightarrow$	10000000
01000000	$\rightarrow$	01000000
00100000	$\rightarrow$	00100000
00010000	$\rightarrow$	00010000
00001000	$\rightarrow$	00001000
00000100	$\rightarrow$	00000100
00000010	$\rightarrow$	00000010
00000001	$\rightarrow$	00000001

Can this be learned??

Learned weights for hidden layer

A network:

dimensionality reduction NN.

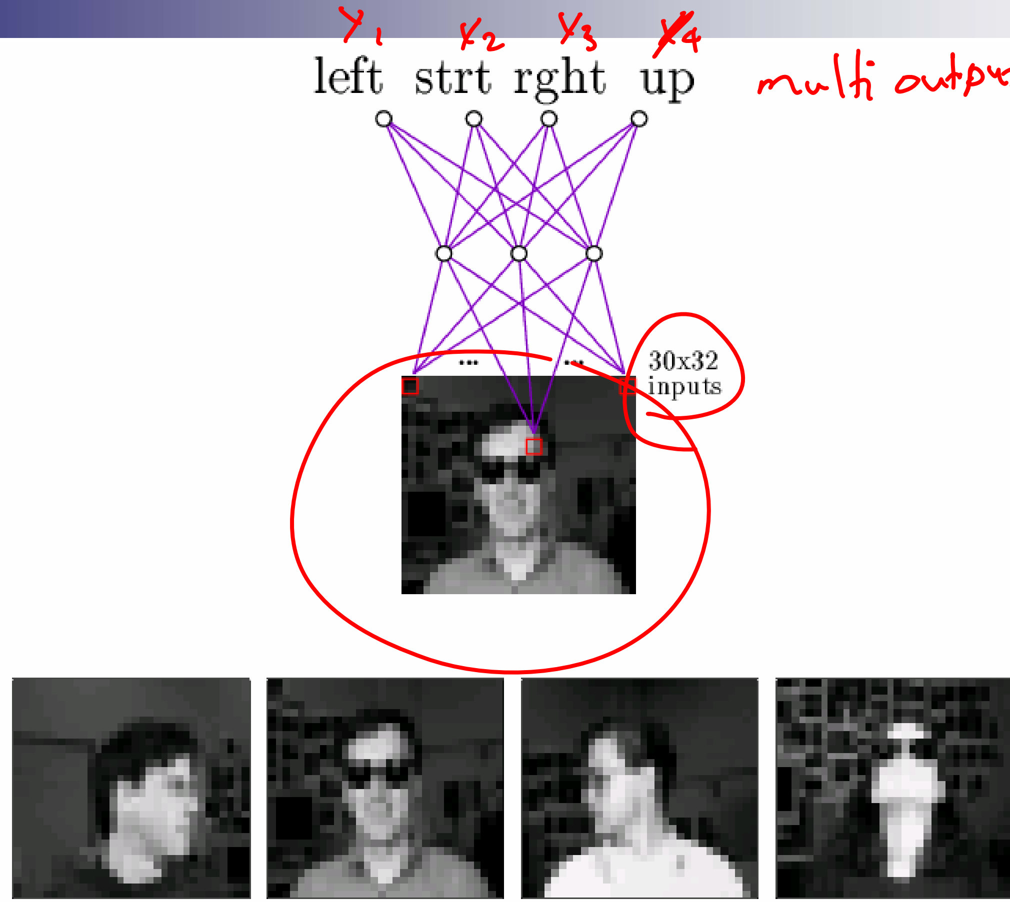


learned values

Learned hidden layer representation:

Input		Hidden Values				Output
10000000	→	.89	.04	.08	→	10000000
01000000	→	.01	.11	.88	→	01000000
00100000	→	.01	.97	.27	→	00100000
00010000	→	.99	.97	.71	→	00010000
00001000	→	.03	.05	.02	→	00001000
00000100	→	.22	.99	.99	→	00000100
00000010	→	.80	.01	.98	→	00000010
00000001	→	.60	.94	.01	→	00000001

NN for images



multi output

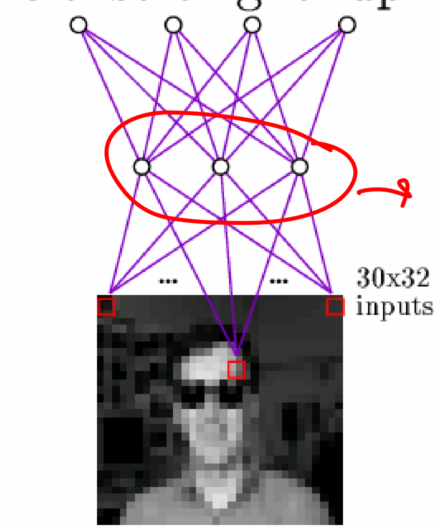
$y = \begin{pmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \end{pmatrix}$

class $y = \arg\max_i y_i$

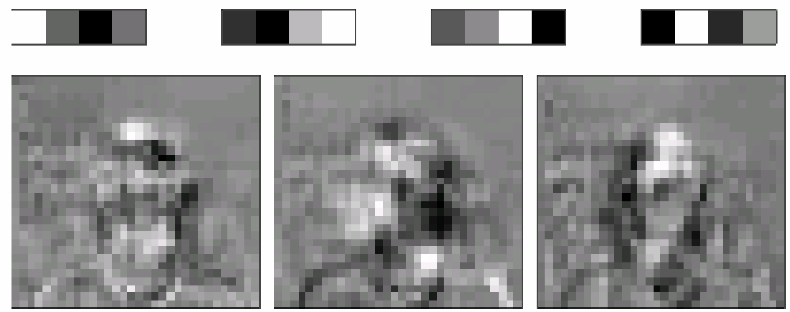
Typical input images

90% accurate learning head pose, and recognizing 1-of-20 faces

Weights in NN for images



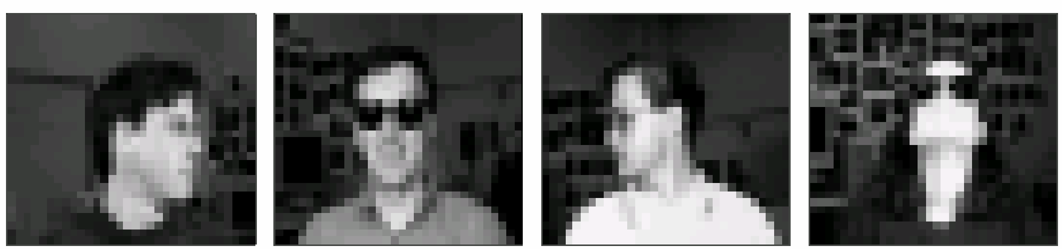
Learned Weights



reacts to up

to left

to straight



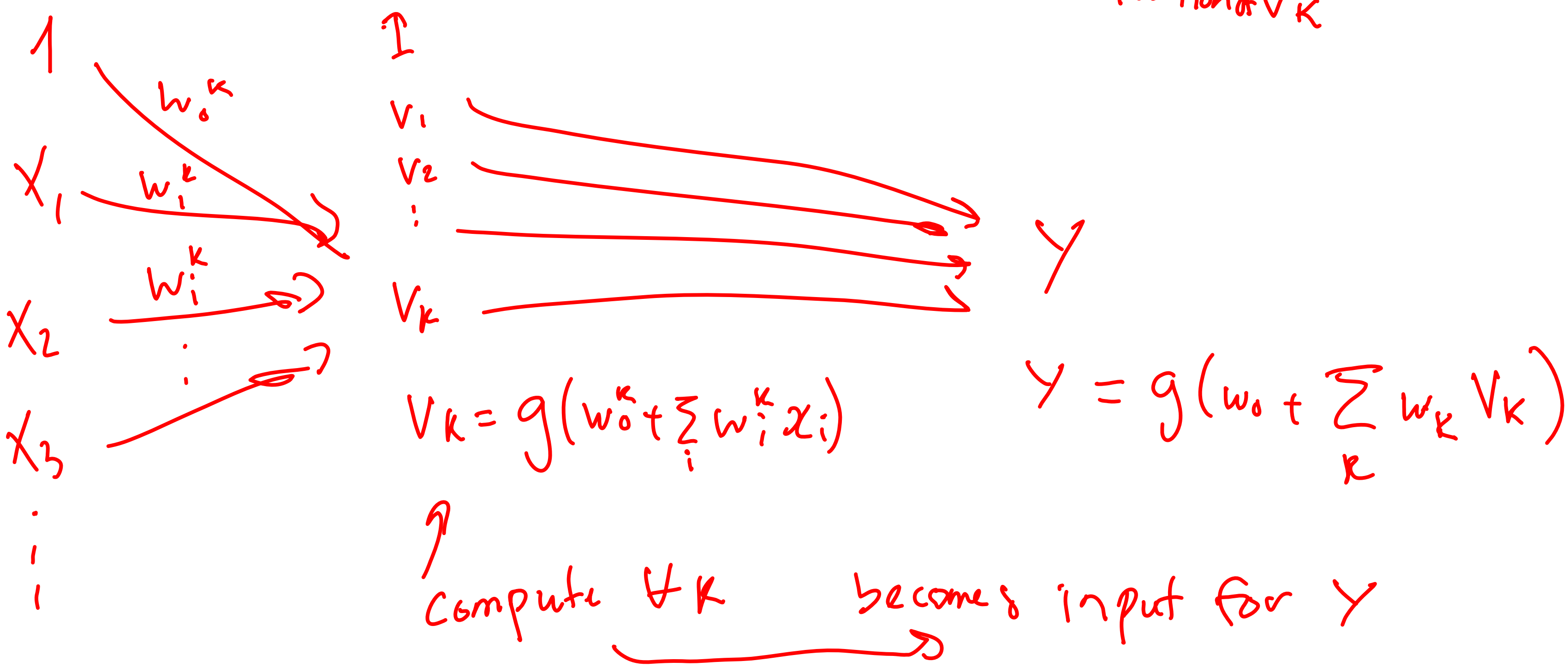
Typical input images

Forward propagation for 1-hidden layer - Prediction

- 1-hidden layer:

$$out(\mathbf{x}) = g\left(w_0 + \sum_k w_k g(w_0^k + \sum_i w_i^k x_i)\right)$$

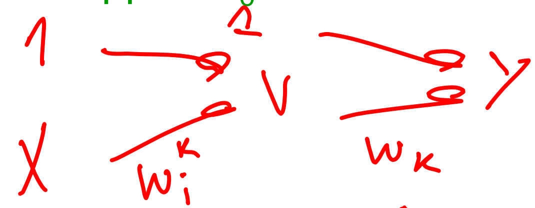
activation of V_k



Gradient descent for 1-hidden layer – Back-propagation: Computing $\frac{\partial \ell(W)}{\partial w_k}$

to simplify dropped w_0

Dropped w_0 to make derivation simpler



$$\ell(W) = \frac{1}{2}\sum_j [y^j - out(\mathbf{x}^j)]^2$$

$$out(\mathbf{x}) = g\left(\sum_{k'} w_{k'} g(\sum_{i'} w_{i'}^{k'} x_{i'})\right)$$

$$\frac{\partial \ell(W)}{\partial w_k} = \sum_j -[y - out(\mathbf{x}^j)]\frac{\partial out(\mathbf{x}^j)}{\partial w_k}$$

$$\frac{\partial out(x)}{\partial w_k} = \frac{d}{w_k} g\Big(\sum_{k'} w_{k'} \underbrace{g(\sum_i w_i^{k'} x_i)}_{v_{k'}}\Big)$$

$$= v_k \cdot g'(y).$$

$$= v_k\, g(y)(1 - g(y))$$

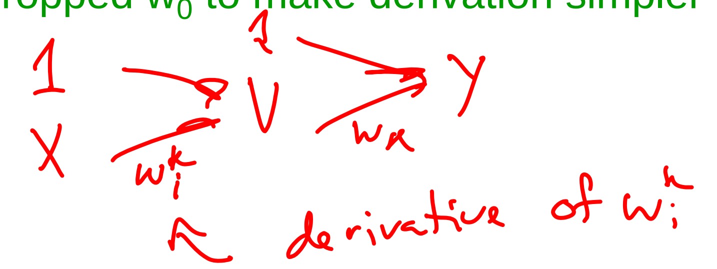
$$g'(x) = g(x)(1-g(x))$$

$$g'(x) = \frac{dg}{dx}$$

↑ derivative

Gradient descent for 1-hidden layer – Back-propagation: Computing $\frac{\partial \ell(W)}{\partial w_i^k}$

Dropped w_0 to make derivation simpler



$$\ell(W) = \frac{1}{2}\sum_j [y^j - out(\mathbf{x}^j)]^2$$

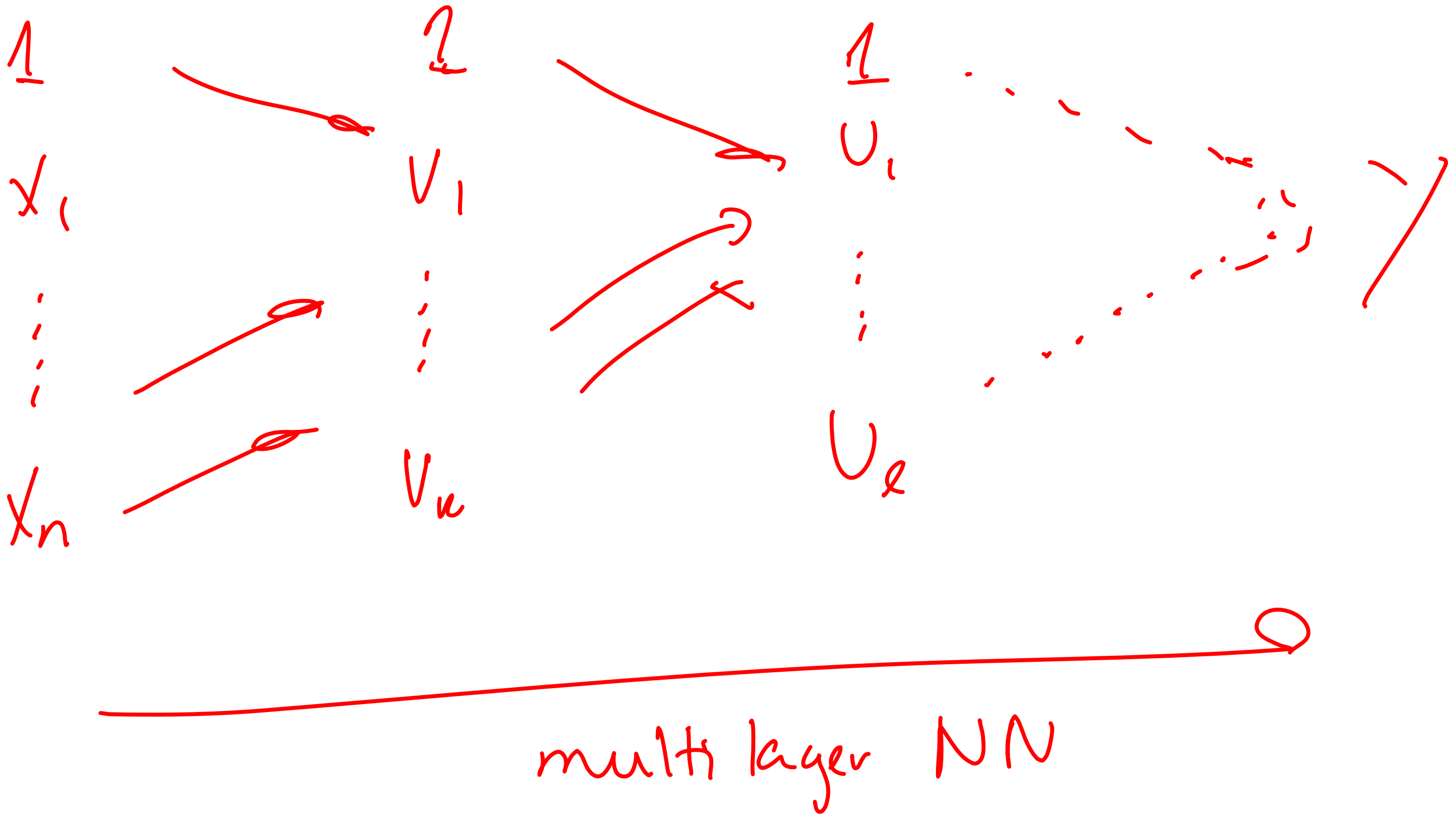
$$out(\mathbf{x}) = g\left(\sum_{k'} w_{k'} g(\sum_{i'} w_{i'}^{k'} x_{i'})\right)$$

$$\frac{\partial \ell(W)}{\partial w_i^k} = \sum_j -[y - out(\mathbf{x}^j)]\frac{\partial out(\mathbf{x}^j)}{\partial w_i^k}$$

$$\frac{\partial out(x)}{\partial w_i^k} = \frac{\partial}{\partial w_i^k} g\Big(\sum_{k'} w_{k'}\, g\big(\underbrace{\textstyle\sum_{i'} w_{i'}^{k'} x_{i'}}_{v_{k'}}\big)\Big)$$

$$= g'\Big(\sum_{k'} w_{k'}\, g(v_{k'})\Big) \cdot \underbrace{\frac{\partial}{\partial w_i^k}\Big[\sum_{k'} w_{k'}\, g\big(\textstyle\sum_{i'} w_{i'}^{k'} x_{i'}\big)\Big]}_{w_k\, g'(\sum_{i'} w_{i'}^{k'} x_{i'}) \cdot x_i}$$

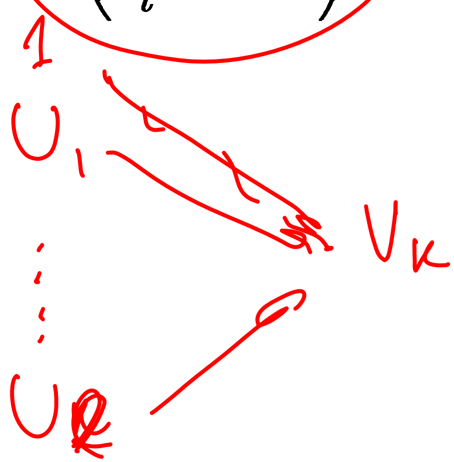
Multilayer neural networks



Forward propagation – prediction

- Recursive algorithm
- Start from input layer
- Output of node V_k with parents $U_1, U_2, \ldots$:

$$V_k = g\left(\sum_i w_i^k U_i\right)$$

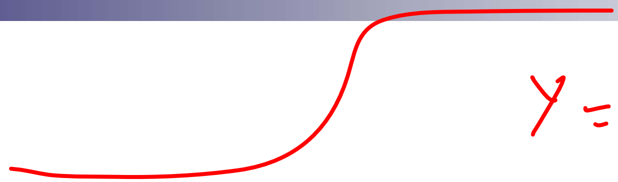


Back-propagation – learning

- Just gradient descent!!!
- Recursive algorithm for computing gradient
- For each example
  - Perform forward propagation
  - Start from output layer
  - Compute gradient of node V_k with parents $U_1, U_2, \ldots$
  - Update weight w_i^k

Many possible response functions

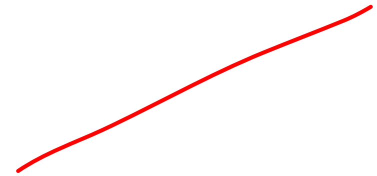
- Sigmoid



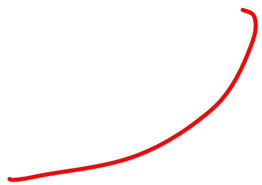
$$y = g\left(w_0 + \sum_i w_i x_i\right)$$

- Linear

$$y = w_0 + \sum_i w_i x_i$$

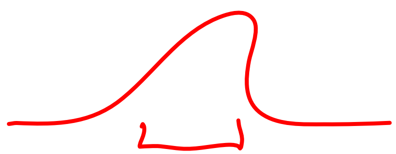


- Exponential



$$y = e^{w_0 + \sum_i w_i x_i}$$

- Gaussian

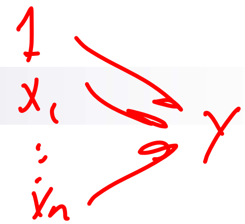


$$y = e^{-\left(w_0 + \sum_i w_i x_i\right)^2}$$

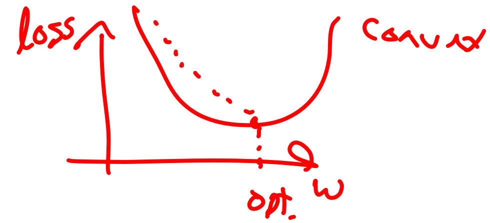
return true if input in this range

- …

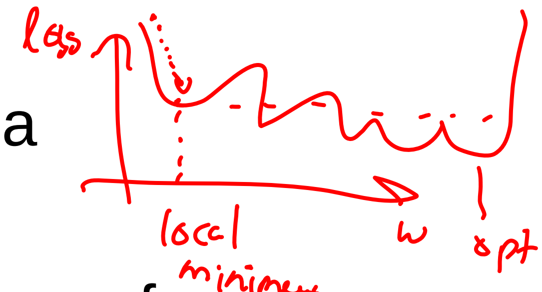
Convergence of backprop



- Perceptron leads to convex optimization
  - Gradient descent reaches **global minima**

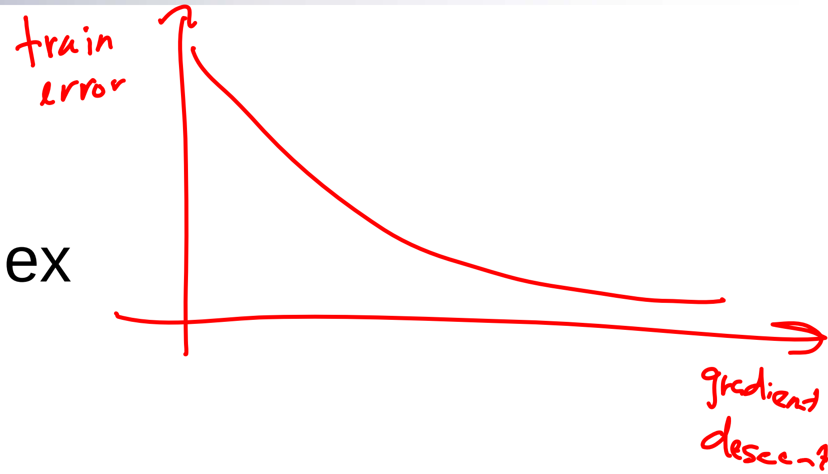


- Multilayer neural nets **not convex**
  - Gradient descent gets stuck in local minima
  - Hard to set learning rate
  - Selecting number of hidden units and layers = fuzzy process
  - NNs falling in disfavor in last few years
  - We'll see later in semester, *kernel trick* is a good alternative
  - Nonetheless, neural nets are one of the most used ML approaches

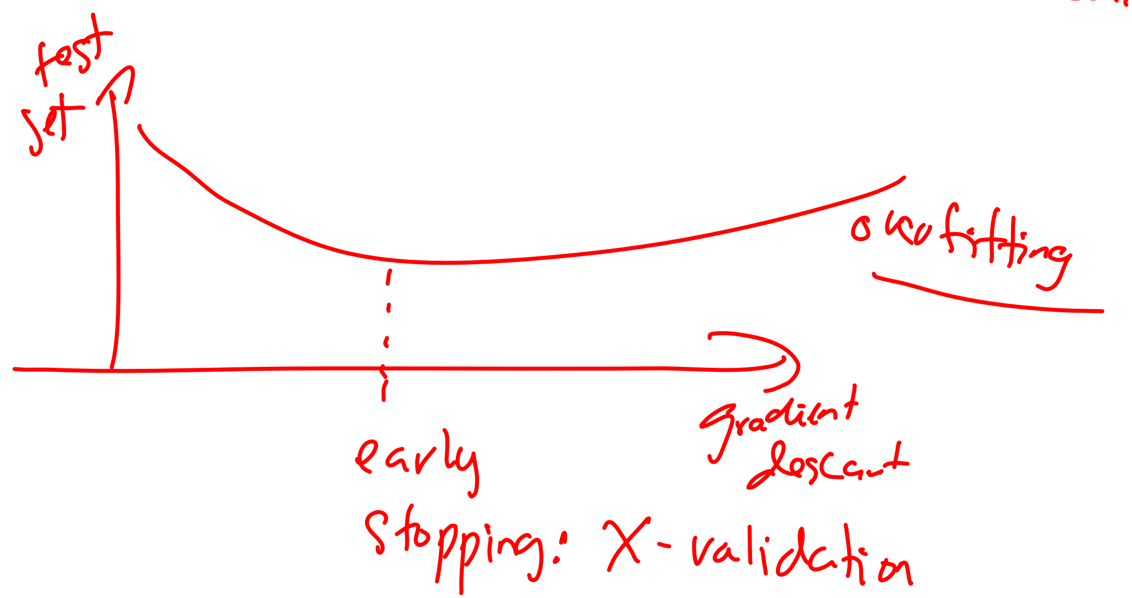


Training set error

- Neural nets represent complex functions
  - Output becomes more complex with gradient steps



- Training set error



What about test set error?

Overfitting

- Output fits training data "too well"
  - Poor test set accuracy
- Overfitting the training data
  - Related to bias-variance tradeoff
  - One of central problems of ML
- Avoiding overfitting?
  - More training data
  - Regularization
  - Early stopping

What you need to know about neural networks

- Perceptron: relate to L.R.
  - Representation
  - Perceptron learning rule
  - Derivation
- Multilayer neural nets
  - Representation
  - Derivation of backprop
  - Learning rule
- Overfitting
  - Definition
  - Training set versus test set
  - Learning curve

Instance-based Learning

Machine Learning – 10701/15781

Carlos Guestrin

Carnegie Mellon University

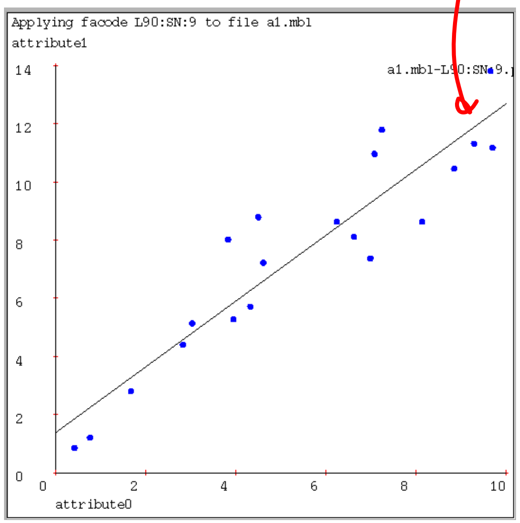
February 15th, 2006

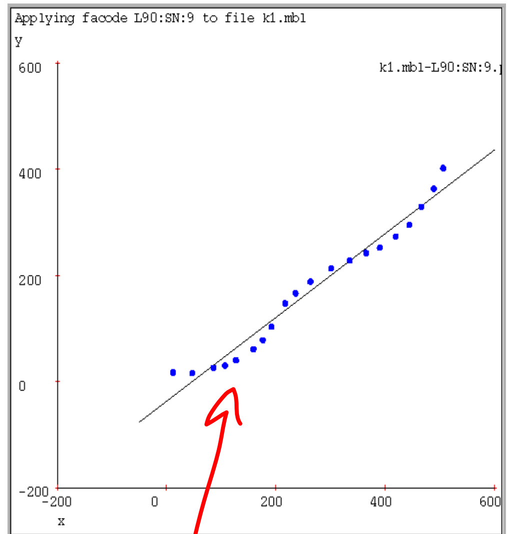
Announcements

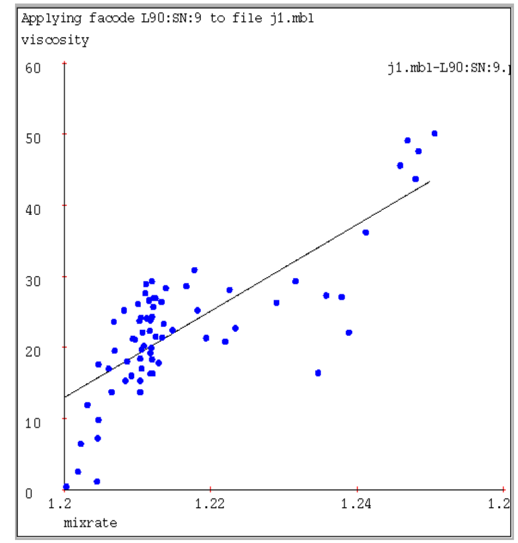
- Reminder: Second homework due Monday 21st

Why not just use Linear Regression?

ax+b







pretty good

missing trend

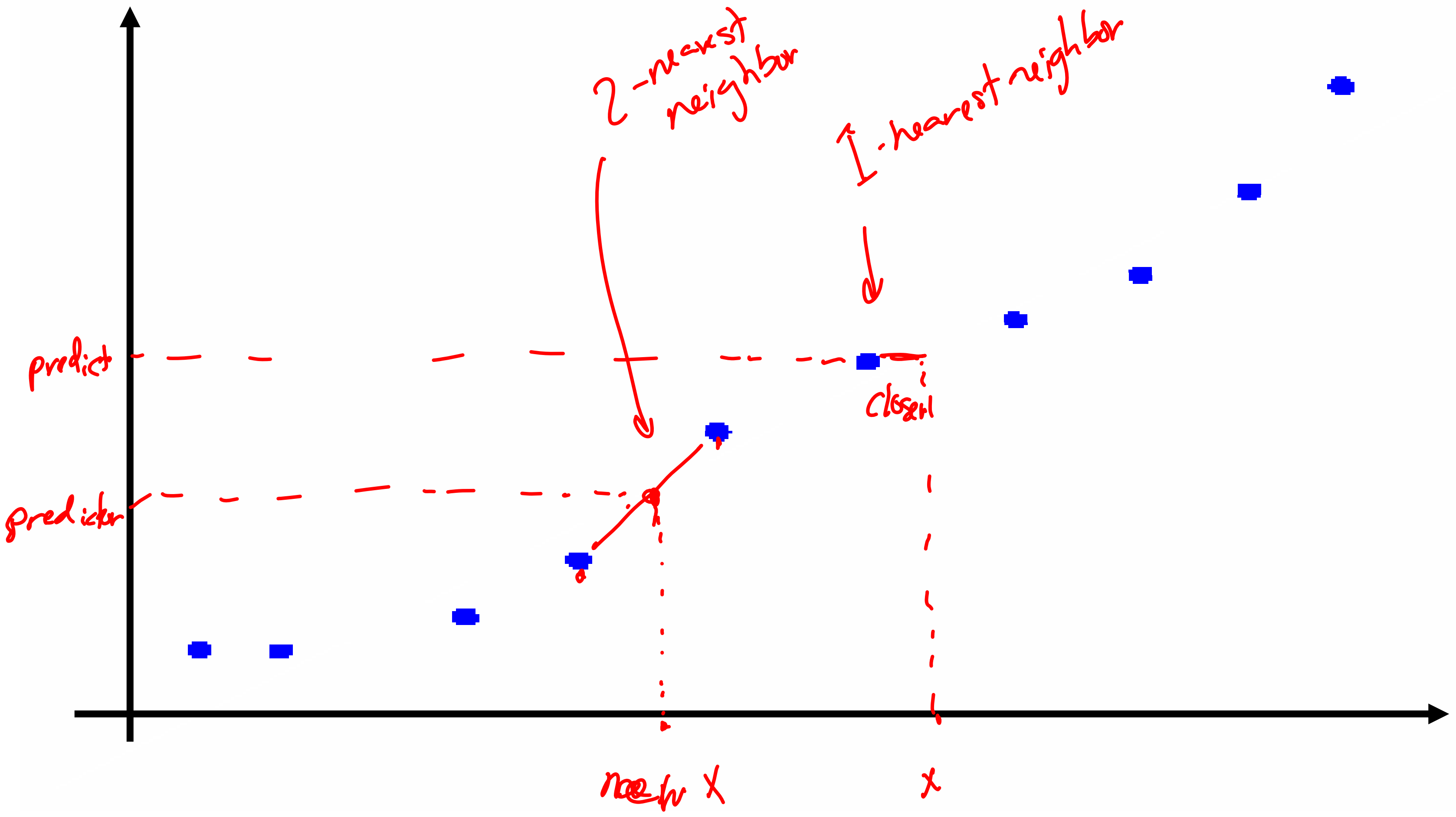
Could more basis functions

bad job!

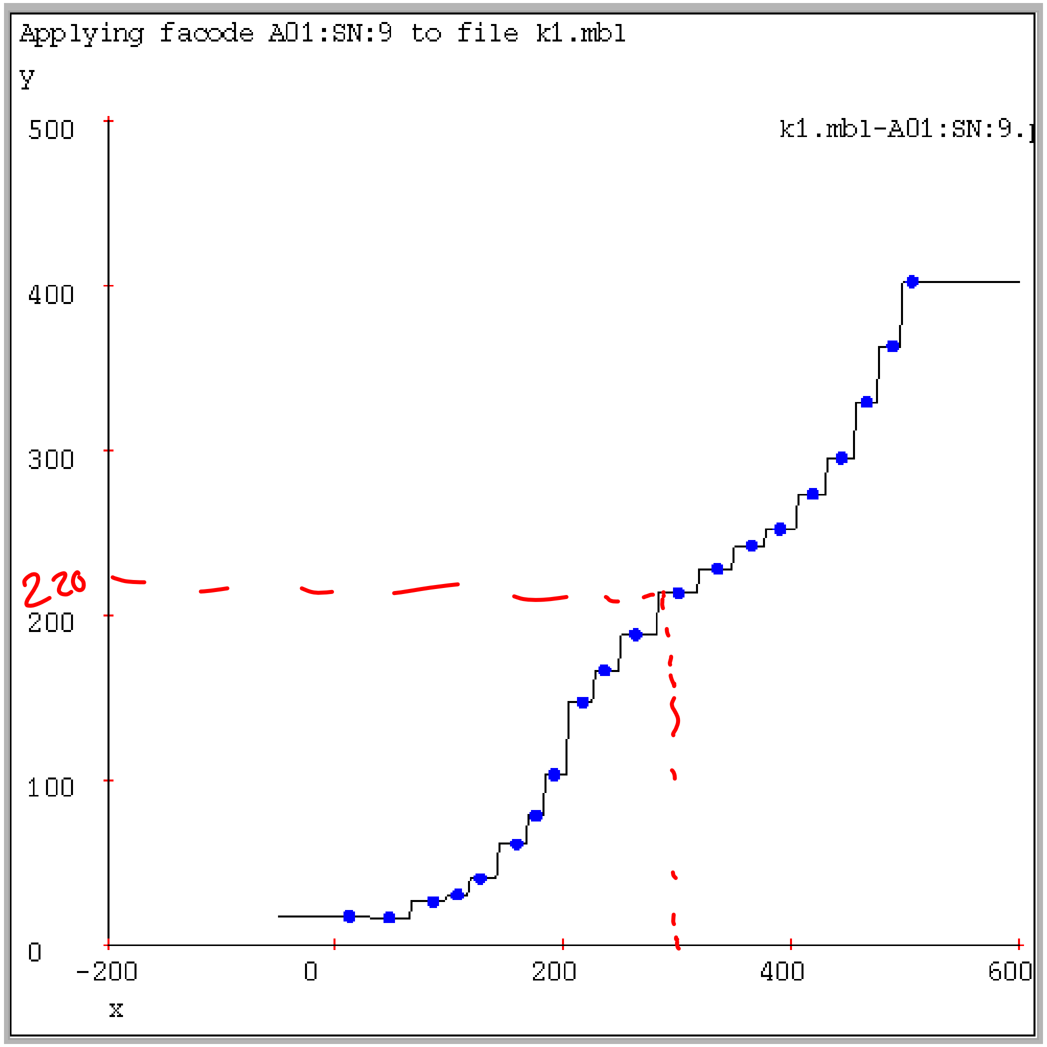
not sure what do...

lots more basis functions

Using data to predict new data



1-Nearest neighbor



Univariate 1-Nearest Neighbor

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

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

Nearest Neighbor:

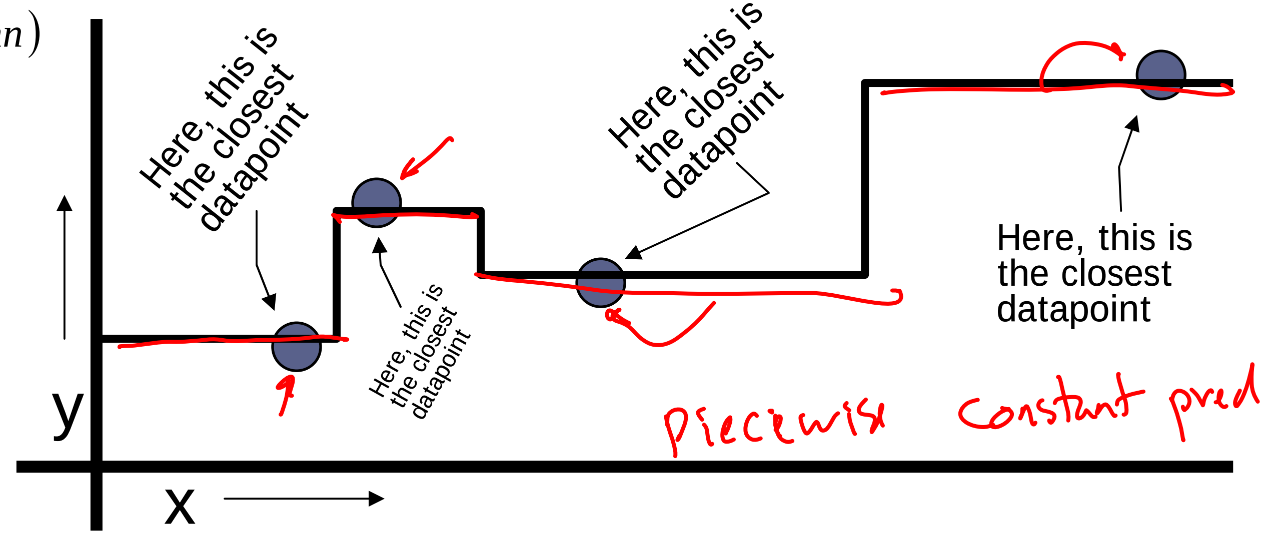
1. Find the closest x_i in our set of datapoints

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

closest in dataset

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

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

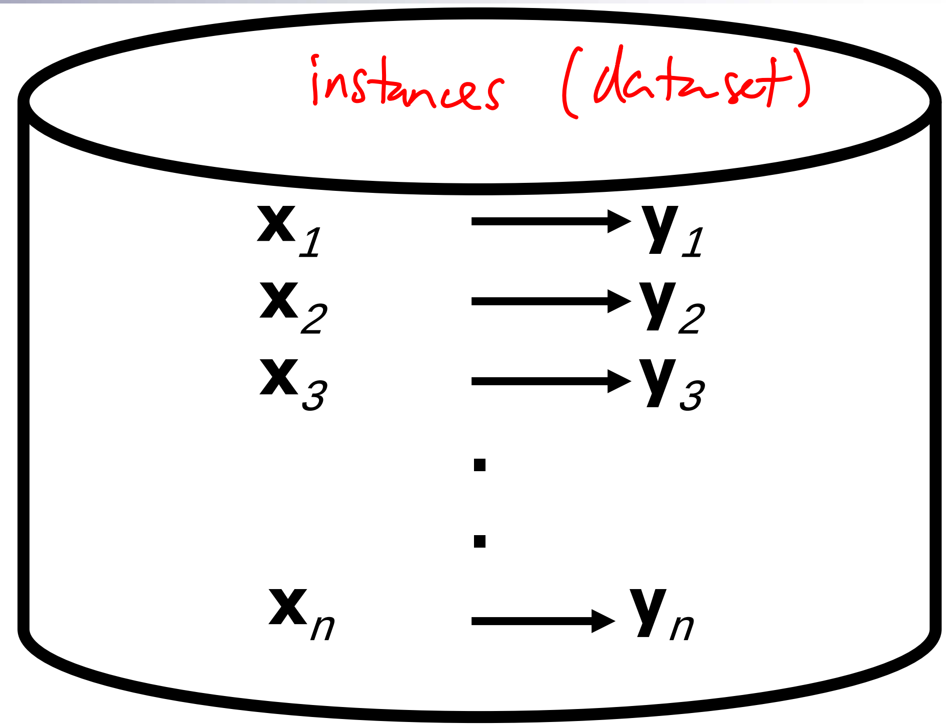


1-Nearest Neighbor is an example of….

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 — define "closest"
- How many nearby neighbors to look at?
- A weighting function (optional)
- How to fit with the local points?

1-Nearest Neighbor

Four things make a memory based learner:

1. *A distance metric* input x: $i^* = \arg\min_i \|x_i - x\|_2$

   Euclidian (and many more)
2. *How many nearby neighbors to look at?*

   One
3. *A weighting function (optional)*

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

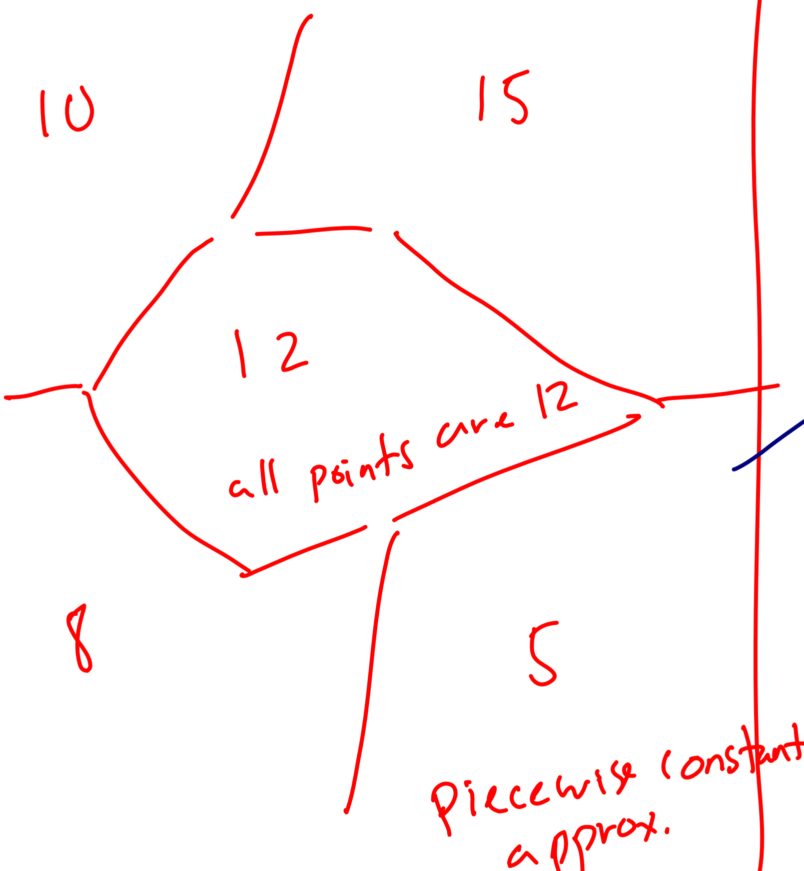
   Just predict the same output as the nearest neighbor.

output y_{i^*}

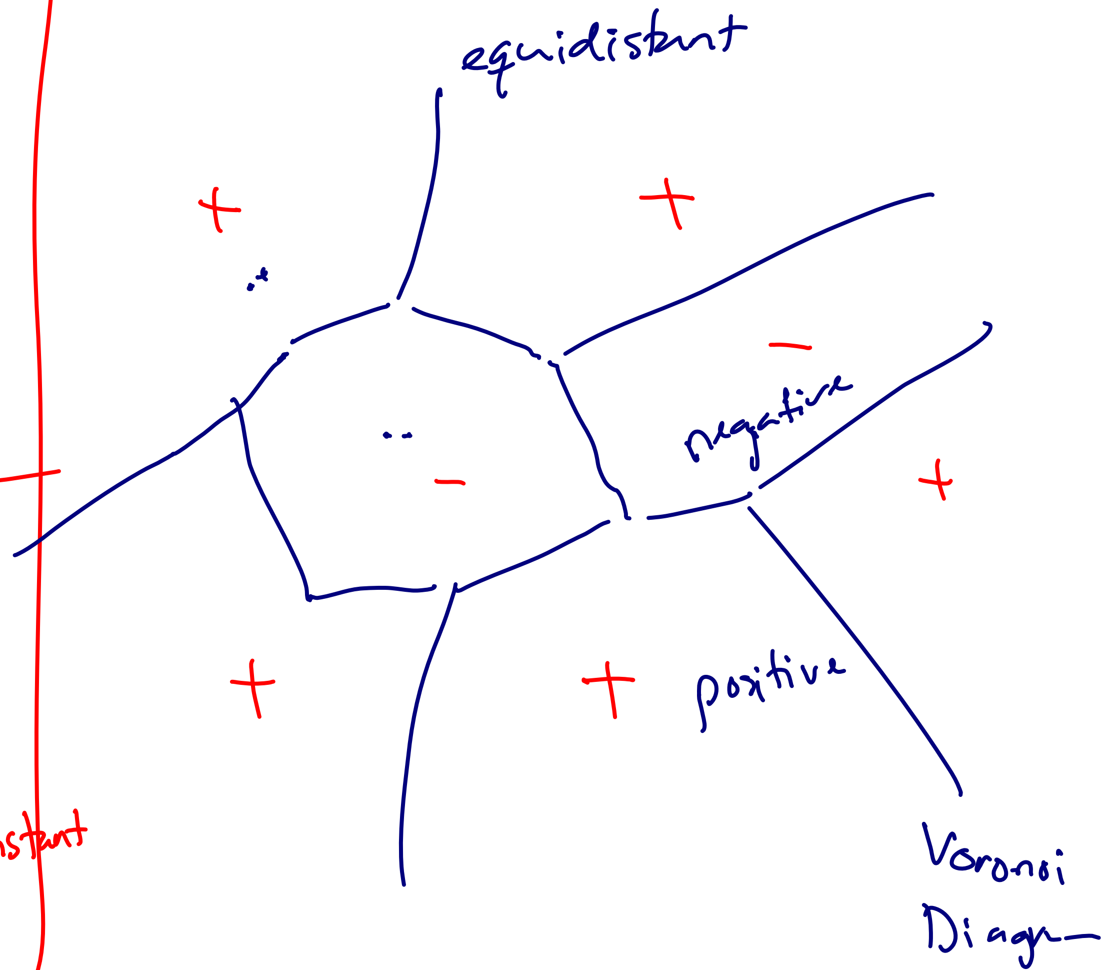
Multivariate 1-NN examples

nearest-neighbor

Regression



Classification

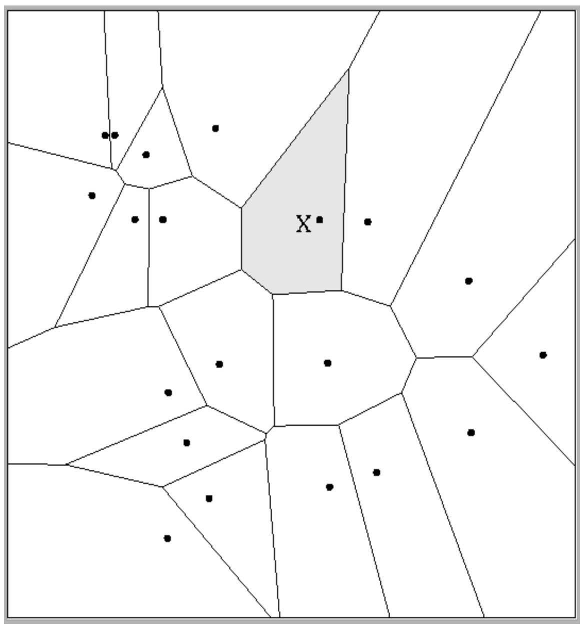


Multivariate distance metrics

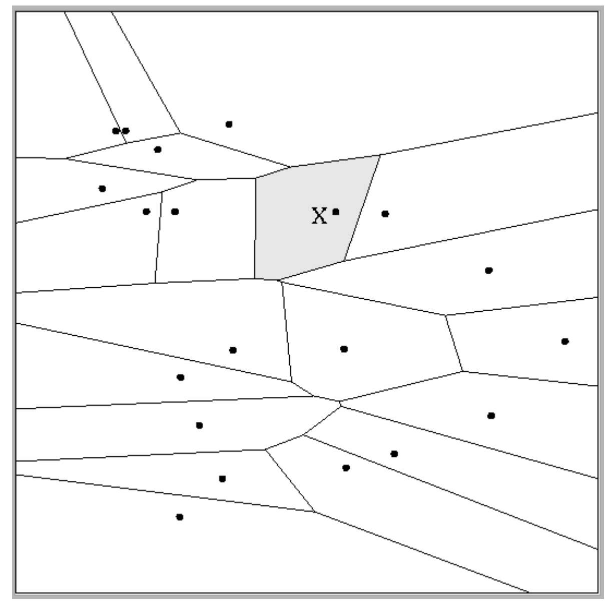
Suppose the input vectors x1, x2, …xn are two dimensional:

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

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,

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

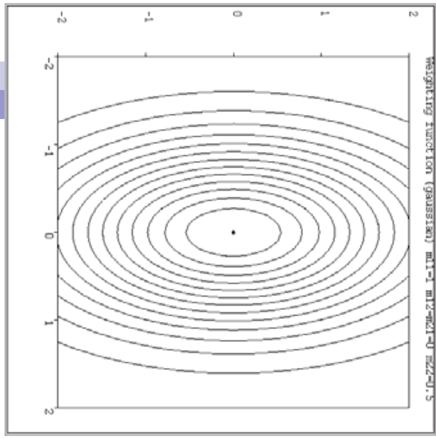
where

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

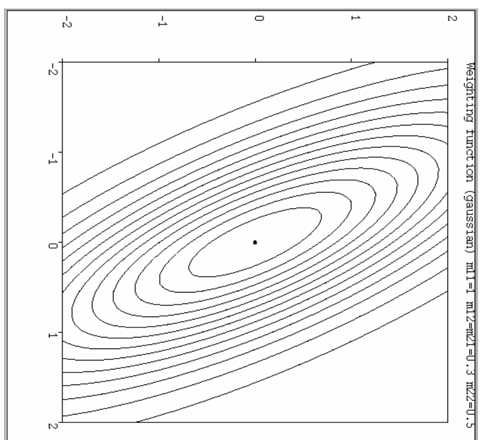
Other Metrics…

- Mahalanobis, Rank-based, Correlation-based,…

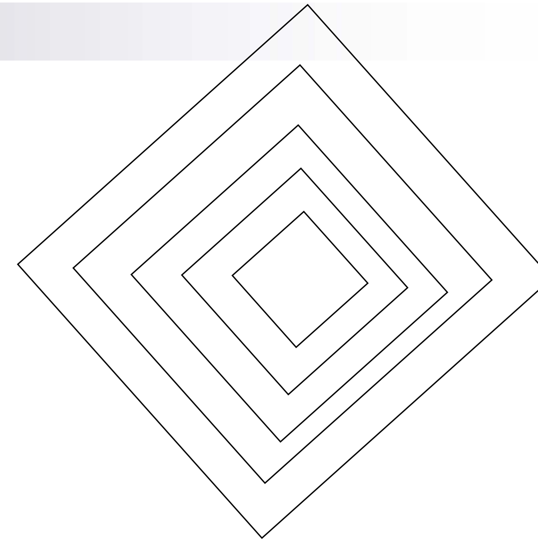
Notable distance metrics (and their level sets)



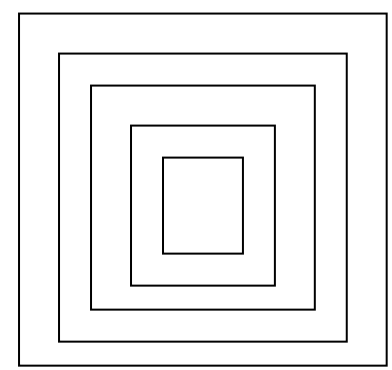
Scaled Euclidian (L_2)



Mahalanobis (here, Σ on the previous slide is not necessarily diagonal, but is symmetric



L_1 norm (absolute)



L∞ *(max) norm*

Consistency of 1-NN

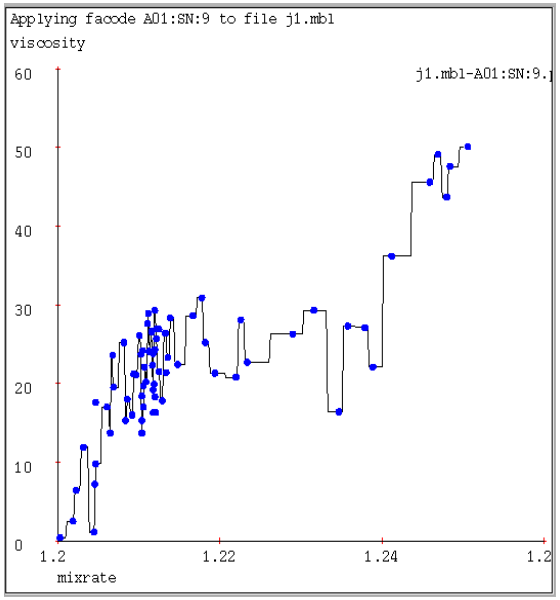
- Consider an estimator f_n trained on n examples
  - e.g., 1-NN, neural nets, regression,...
- Estimator is *consistent* if prediction error goes to zero as amount of data increases
  - e.g., for no noise data, consistent if:

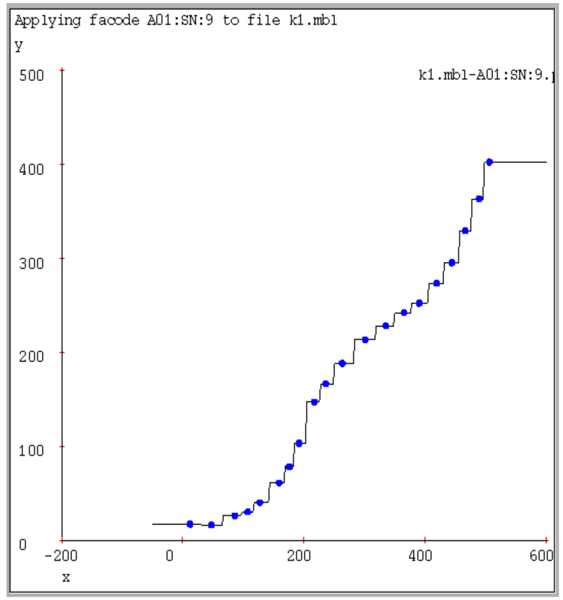
$$\lim_{n \to \infty} MSE(f_n) = 0$$

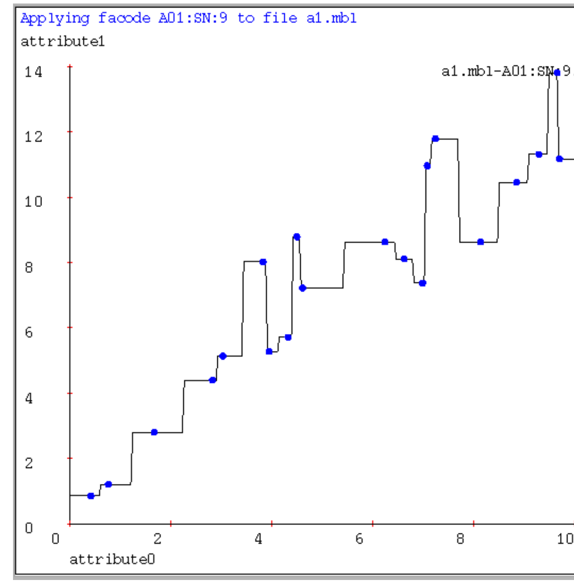
- Regression is not consistent!
  - Representation bias
- **1-NN is consistent** (under some mild fineprint)

What about variance???

1-NN overfits?





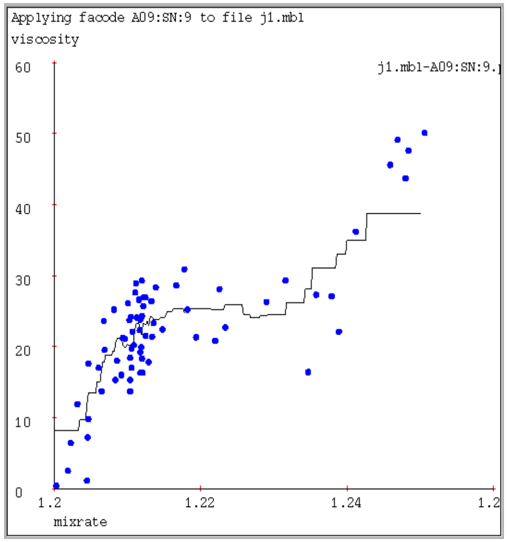


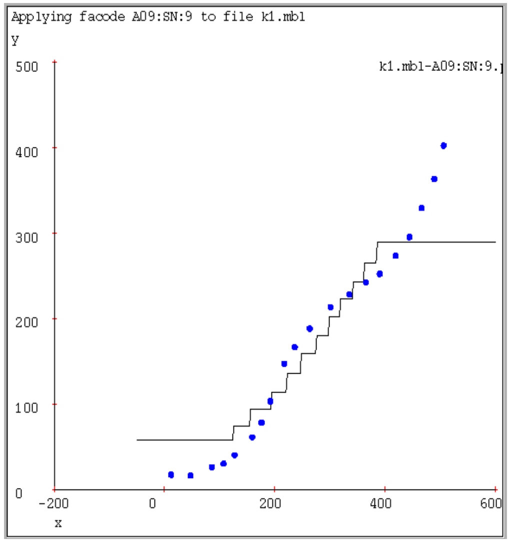
k-Nearest Neighbor

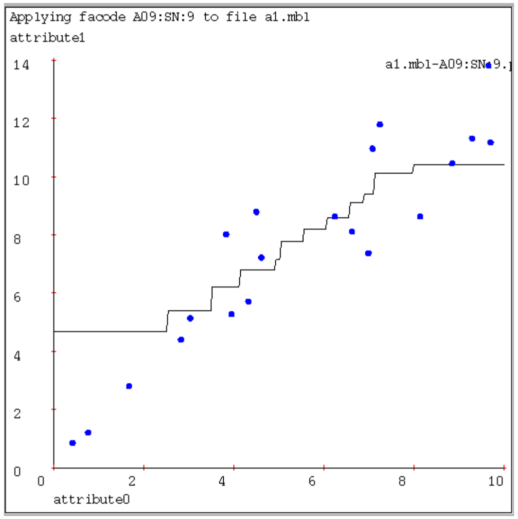
Four things make a memory based learner:

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

k-Nearest Neighbor (here k=9)







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?

Weighted k-NNs

- Neighbors are not all the same

Kernel regression

Four things make a memory based learner:

1. *A distance metric*

   Euclidian (and many more)

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

   All of them

3. *A weighting function (optional)*

   $\boldsymbol{w_i = \exp(-D(x_i, query)^2 / K_w^2)}$

   Nearby points to the query are weighted strongly, far points weakly. The K_W parameter is the **Kernel Width**. Very important.

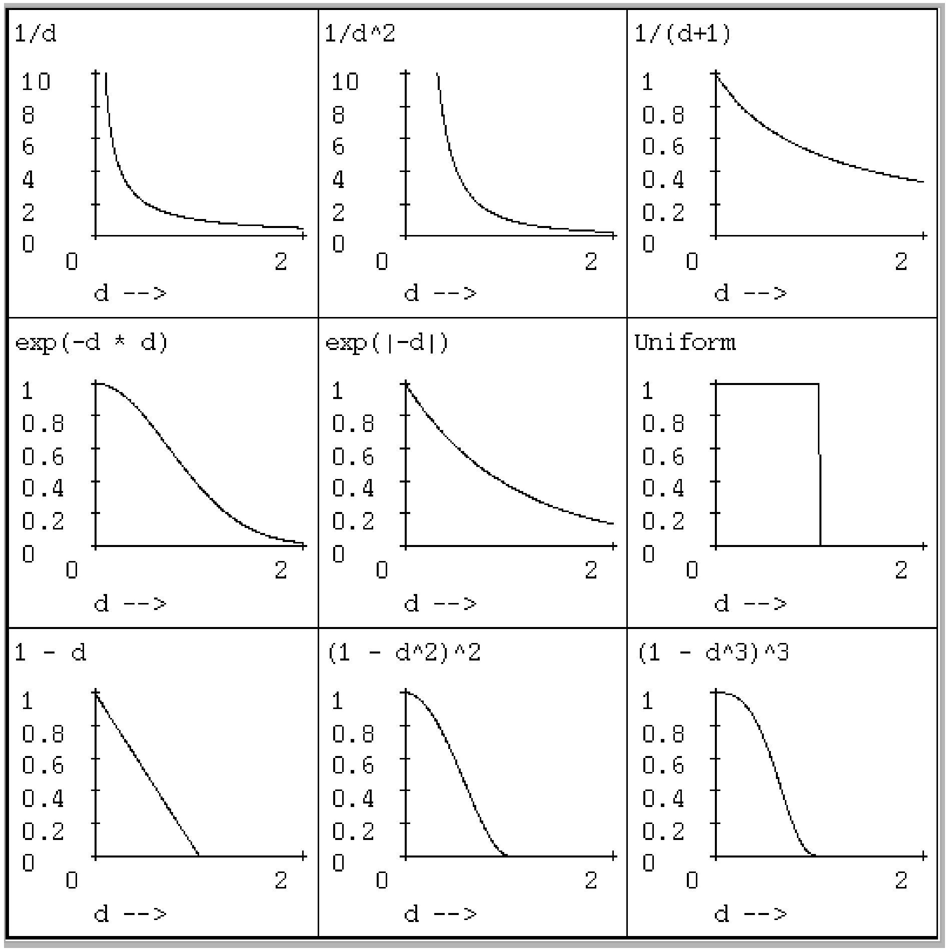
4. *How to fit with the local points?*

   Predict the weighted average of the outputs:

   predict = $\boldsymbol{\Sigma w_i y_i / \Sigma w_i}$

Weighting functions

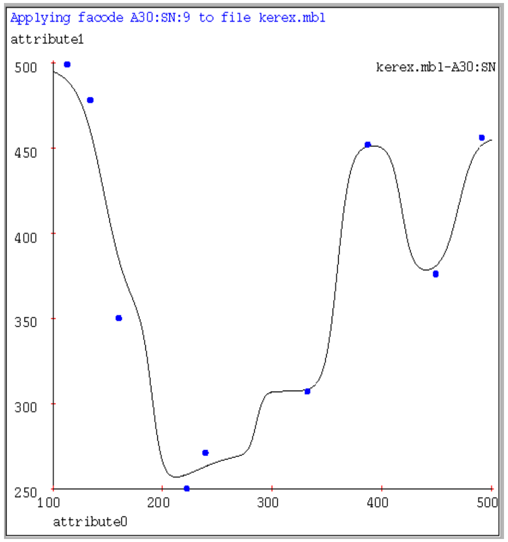
$w_i = exp(-D(x_i, query)^2 / K_w^2)$



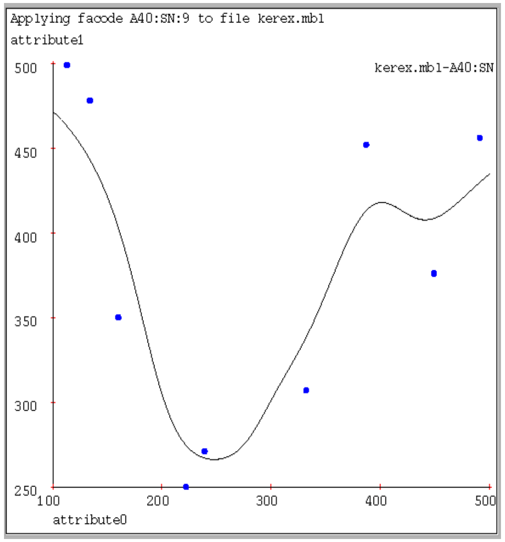
Typically optimize K_w using gradient descent

(Our examples use Gaussian)

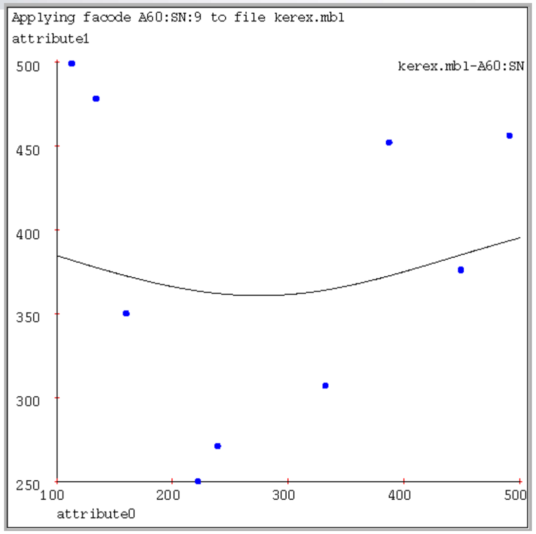
Kernel regression predictions



K_W=10



K_W=20

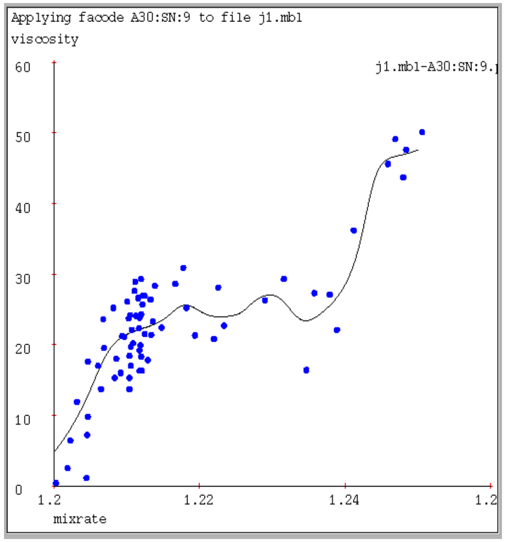


K_W=80

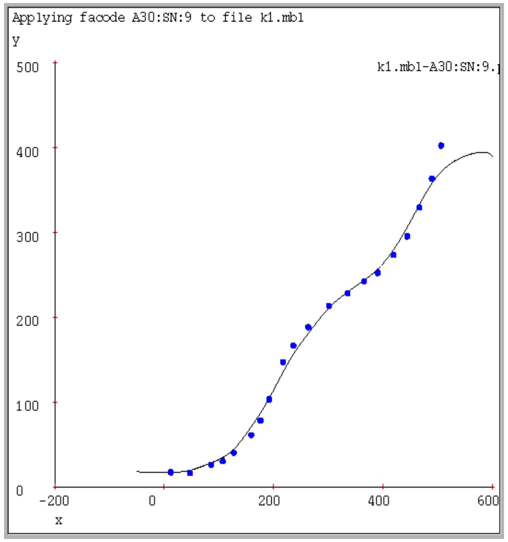
Increasing the kernel width K_w means further away points get an opportunity to influence you.

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

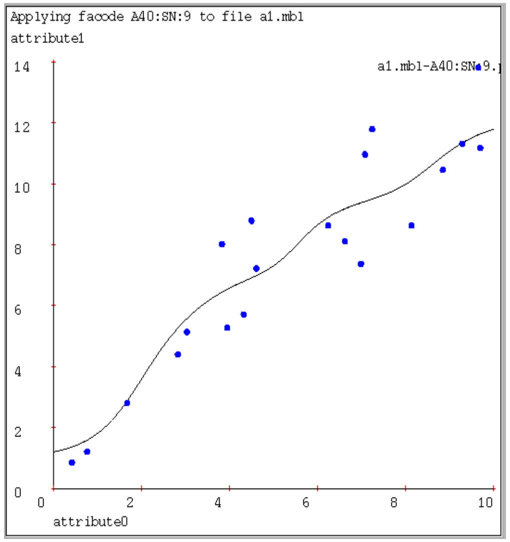
Kernel regression on our test cases



KW=1/32 of x-axis width.



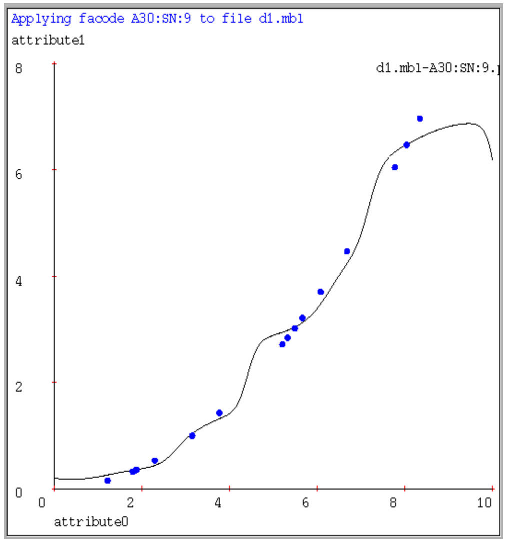
KW=1/32 of x-axis width.



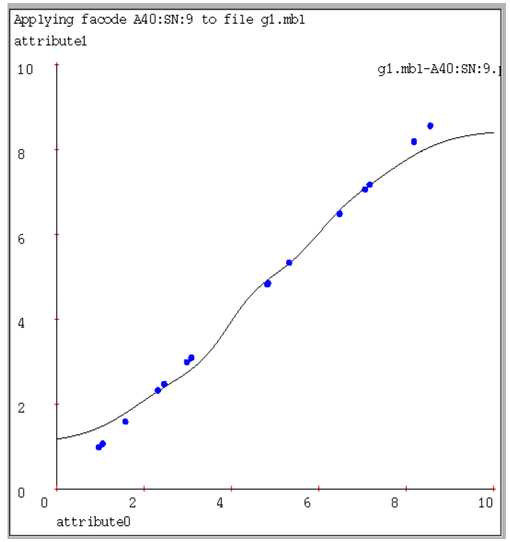
KW=1/16 axis width.

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

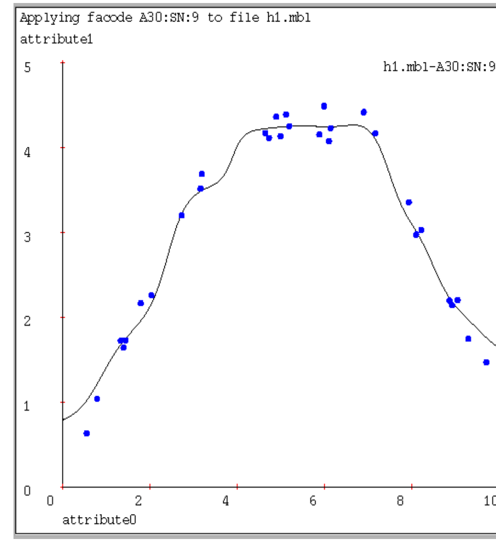
Kernel regression can look bad



KW = Best.



KW = Best.



KW = Best.

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:

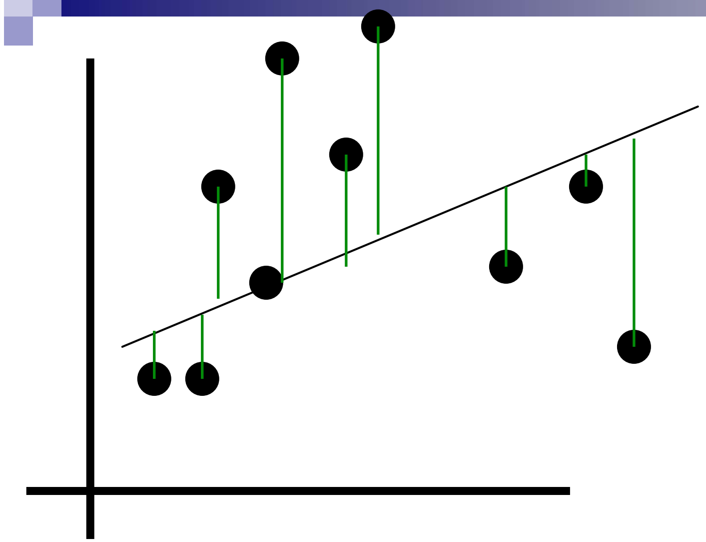
Take a conservative function approximator called LINEAR REGRESSION. Locally weight it.

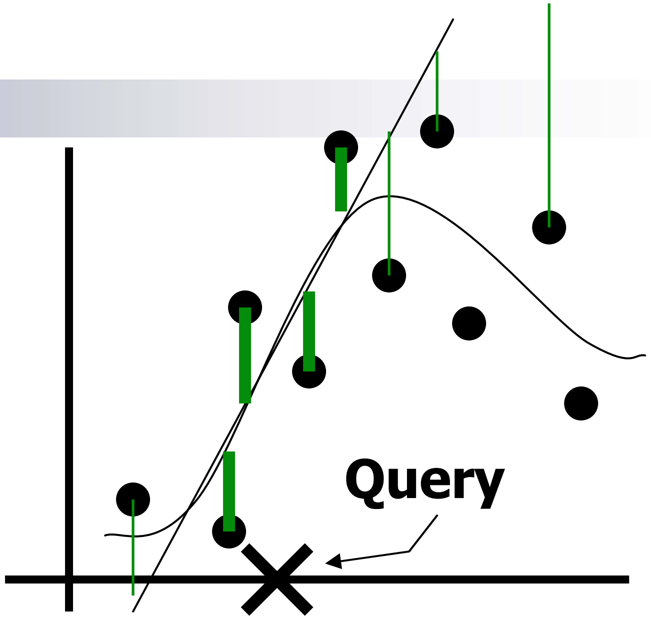
Locally weighted regression

- **Four things make a memory based learner:**
- *A distance metric*
  - **Any**
- *How many nearby neighbors to look at?*
  - **All of them**
- *A weighting function (optional)*
  - **Kernels**
    - ***wi = exp(-D(xi, query)2 / Kw2)***
- *How to fit with the local points?*
  - **General weighted regression:**

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

How LWR works





Linear regression

- Same parameters for all queries

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

Locally weighted regression

- Solve weighted linear regression for each query

$$\hat{\beta} = \left(WX^T WX\right)^{-1} WX^T WY$$

$$W = \begin{pmatrix} w_1 & 0 & 0 & 0 \\ 0 & w_2 & 0 & 0 \\ 0 & 0 & \ddots & 0 \\ 0 & 0 & 0 & w_n \end{pmatrix}$$

Another view of LWR

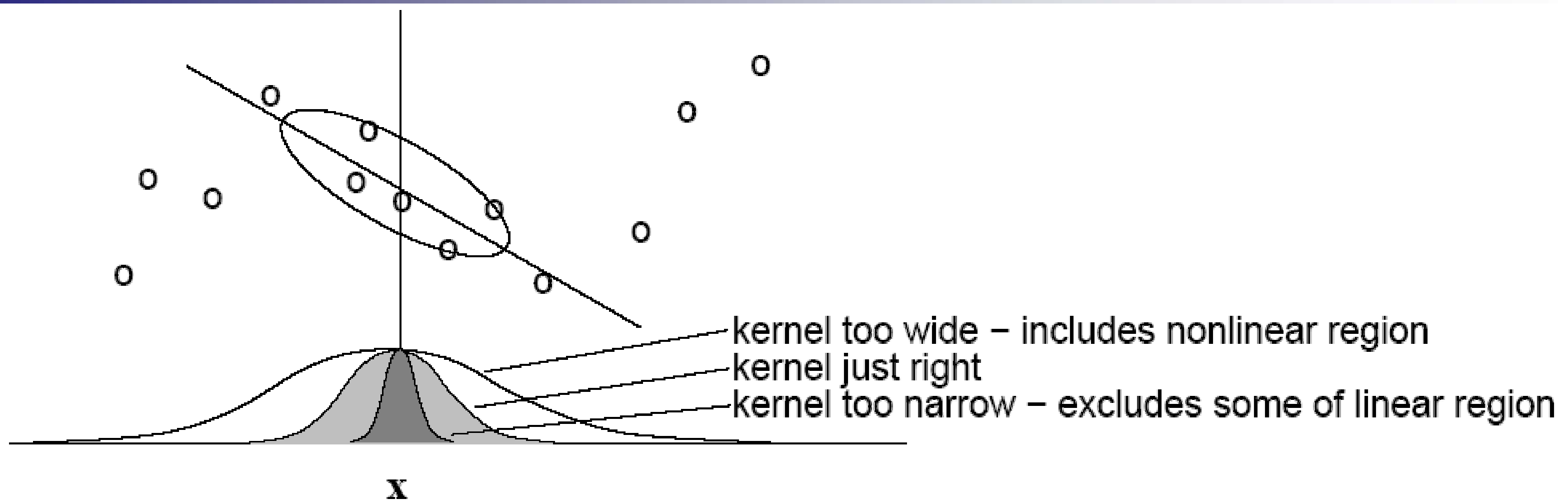
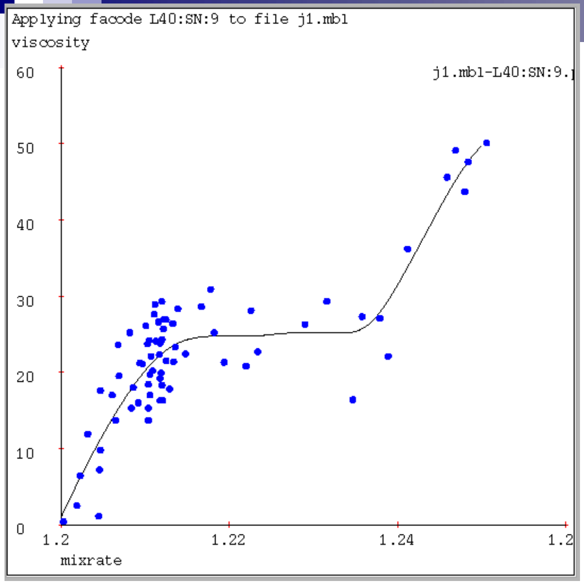
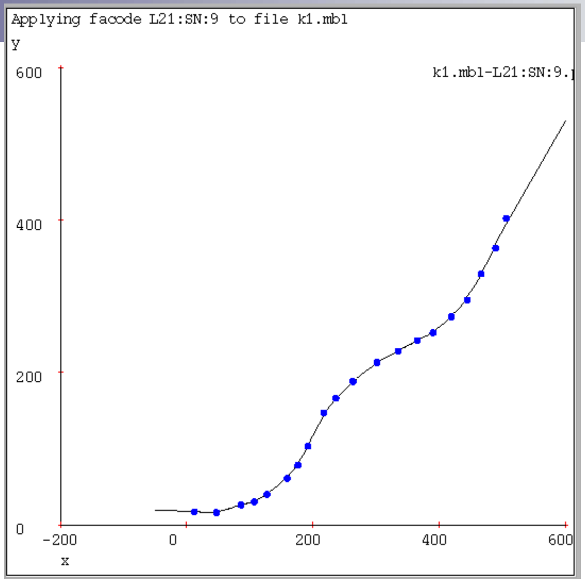


Image from Cohn, D.A., Ghahramani, Z., and Jordan, M.I. (1996) "Active Learning with Statistical Models", JAIR Volume 4, pages 129-145.

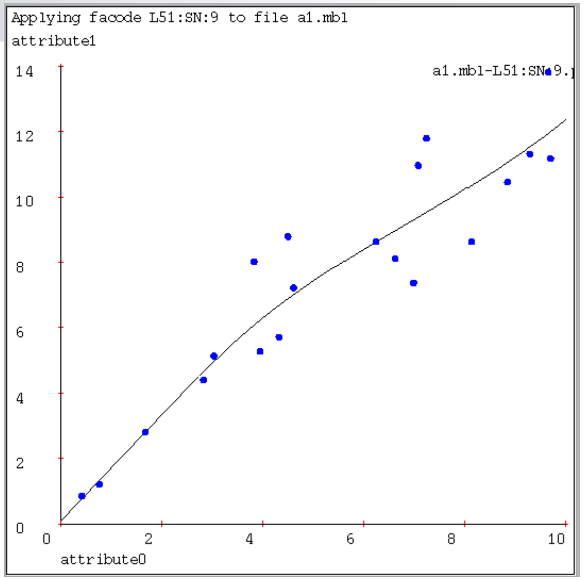
LWR on our test cases



KW = 1/16 of x-axis width.

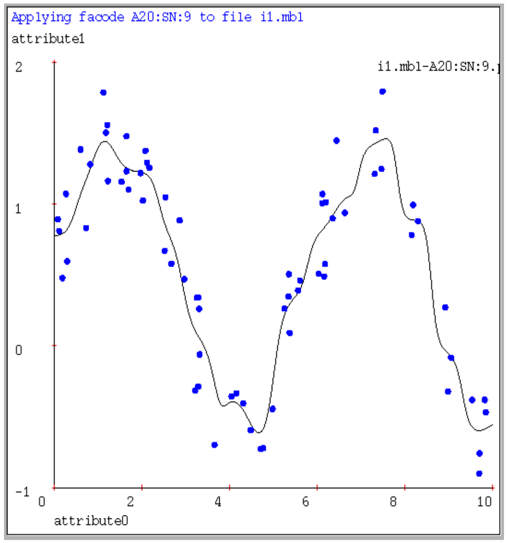


KW = 1/32 of x-axis width.



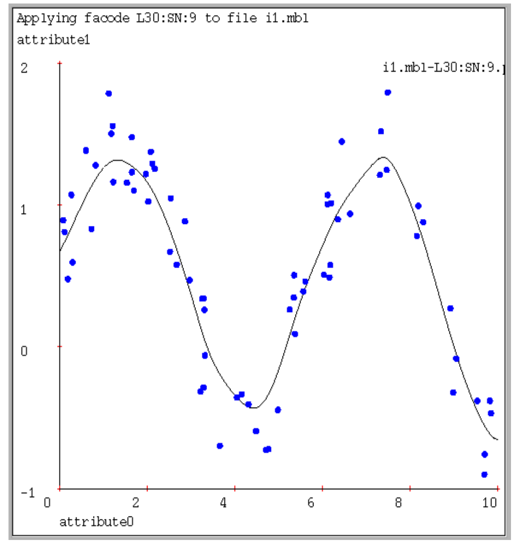
KW = 1/8 of x-axis width.

Locally weighted polynomial regression



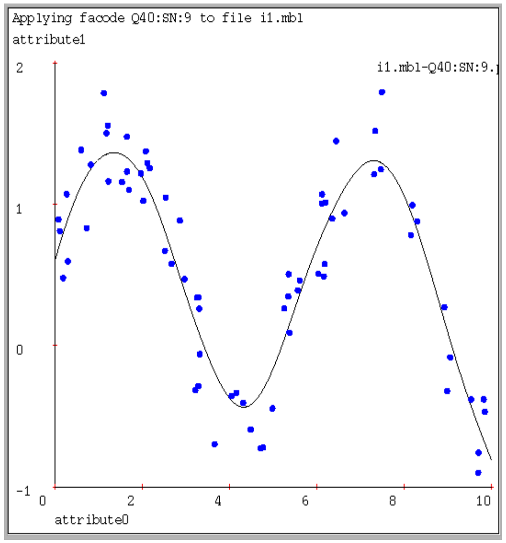
Kernel Regression
Kernel width K_W at optimal level.

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.

Curse of dimensionality for instance-based learning

- Must store and retreve all data!
  - Most real work done during testing
  - For every test sample, must search through all dataset – very slow!
  - We'll see fast methods for dealing with large datasets
- Instance-based learning often poor with noisy or irrelevant features

Curse of the irrelevant feature

What you need to know about instance-based learning

- k-NN
  - Simplest learning algorithm
  - With sufficient data, very hard to beat “strawman” approach
  - Picking k?
- Kernel regression
  - Set k to n (number of data points) and optimize weights by gradient descent
  - Smoother than k-NN
- Locally weighted regression
  - Generalizes kernel regression, not just local average
- Curse of dimensionality
  - Must remember (very large) dataset for prediction
  - Irrelevant features often killers for instance-based approaches

Acknowledgment

- This lecture contains some material from Andrew Moore's excellent collection of ML tutorials:
  - http://www.cs.cmu.edu/~awm/tutorials