Polynomial Curve Fitting

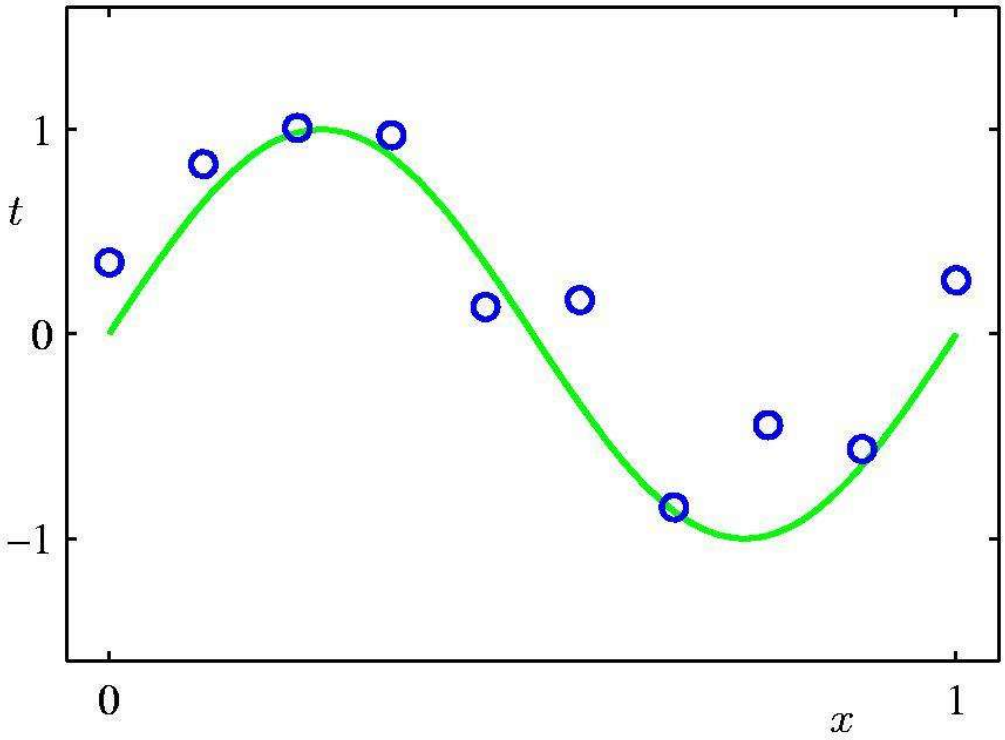
Polynomial Curve Fitting

A Simple Regression Problem

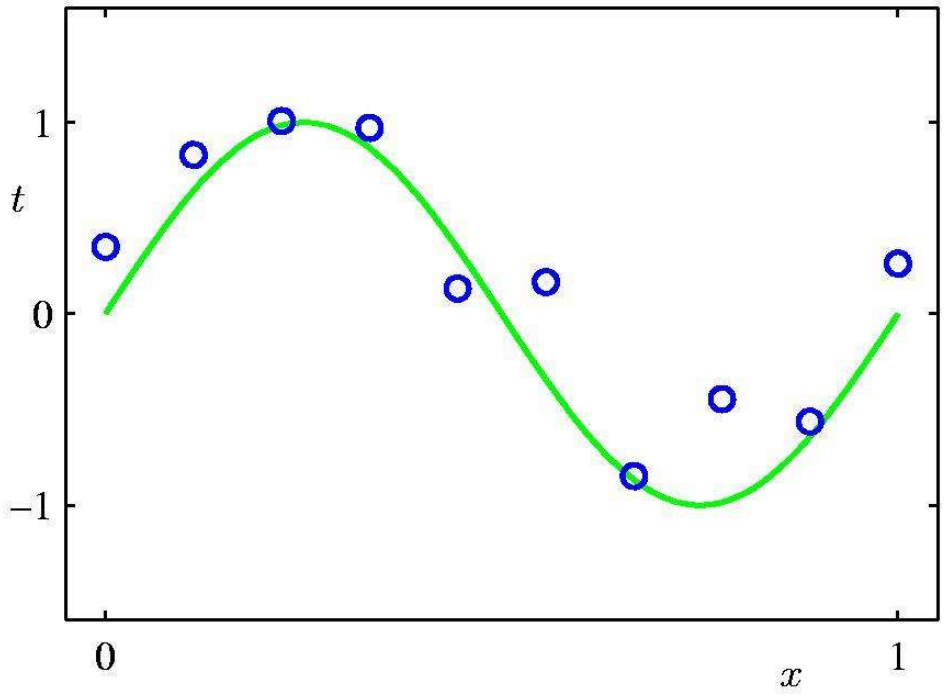
- We observe a real-valued input variable x and we wish to use this observation to predict the value of a real-valued target variable t.
- We use synthetically generated data from the function $\sin(2\pi x)$ with random noise included in the target values.
  - A small level of random noise having a Gaussian distribution
- We have a training set comprising N observations of x, written $x \equiv (x_1, \ldots, x_N)^T$, together with corresponding observations of the values of t, denoted $t \equiv (t_1, \ldots, t_N)^T$.
- Our goal is to predict the value of t for some new value of x,

Polynomial Curve Fitting

- A training data set of N = 10 points, (blue circles),
- The green curve shows the actual function sin(2πx) used to generate the data.
- Our goal is to predict the value of t for some new value of x, without knowledge of the green curve.



Polynomial Curve Fitting

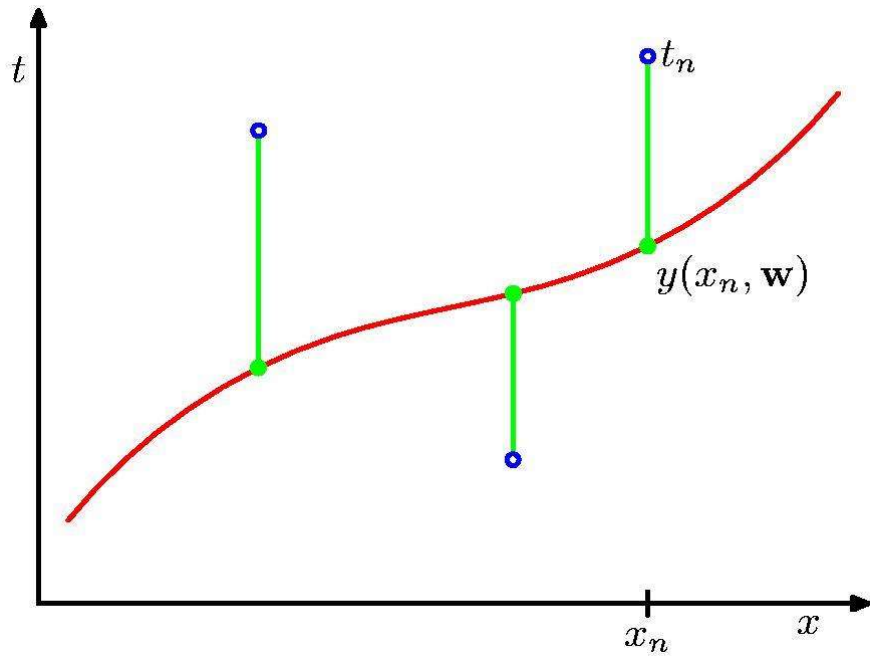


- We try to fit the data using a polynomial function of the form

$$y(x, \mathbf{w}) = w_0 + w_1 x + w_2 x^2 + \ldots + w_M x^M = \sum_{j=0}^{M} w_j x^j$$

Polynomial Curve Fitting

- The values of the coefficients will be determined by fitting the polynomial to the training data.
- This can be done by minimizing an error function that measures the misfit between the function y(x,w), for any given value of w, and the training set data points.
- Error Function: the sum of the squares of the errors between the predictions $y(x_n,w)$ for each data point x_n and the corresponding target values t_n.

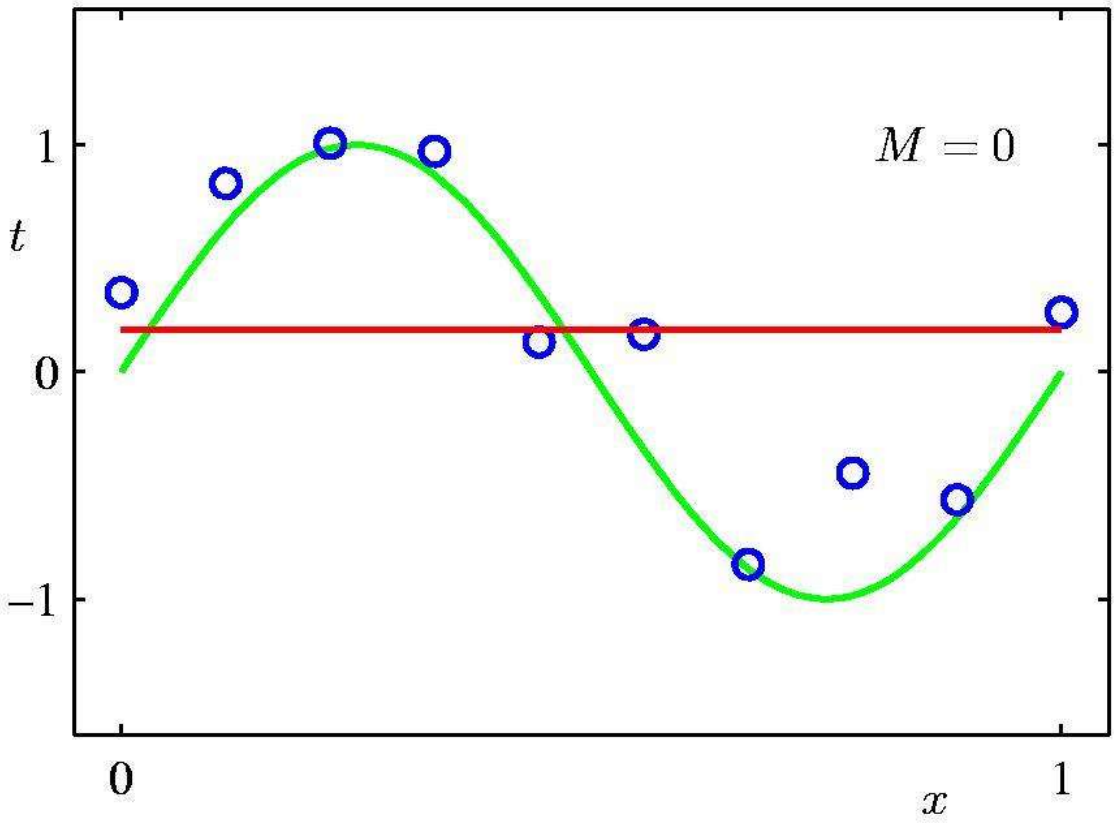


$$E(\mathbf{w}) = \frac{1}{2}\sum_{n=1}^{N}\{y(x_n, \mathbf{w}) - t_n\}^2$$

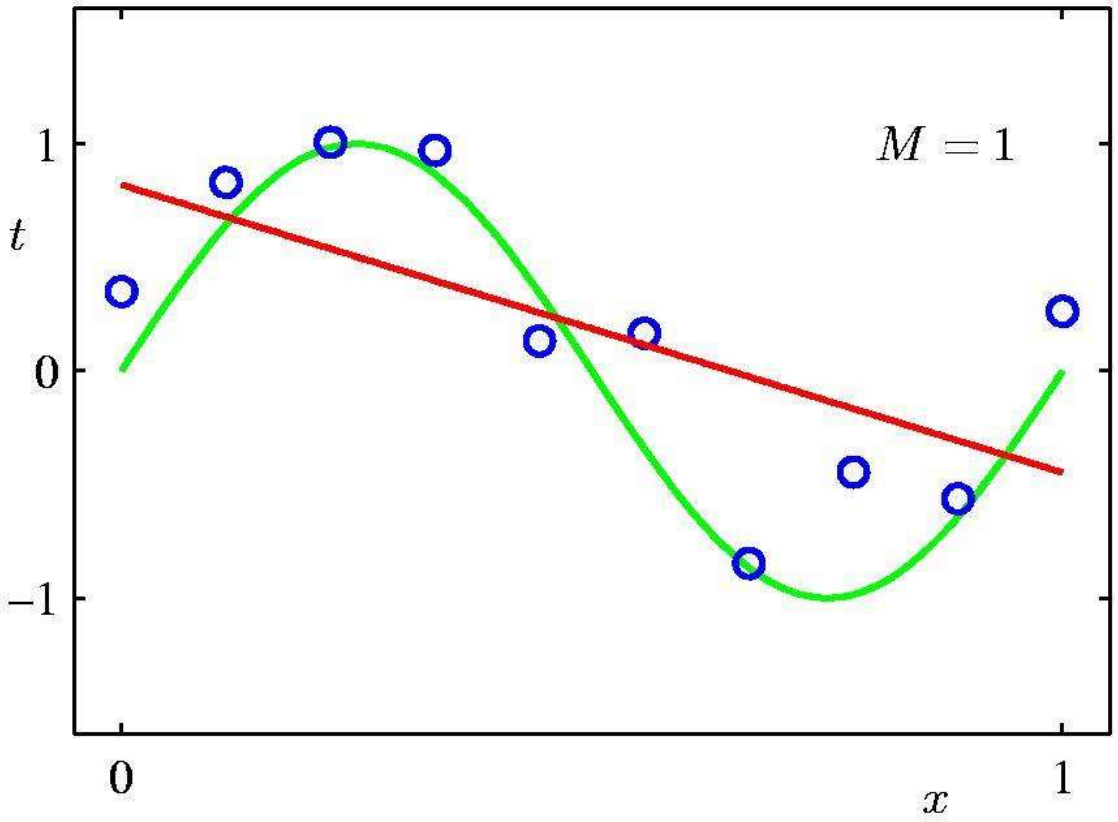
Polynomial Curve Fitting

- We can solve the curve fitting problem by choosing the value of w for which E(w) is as small as possible.
- Since the error function is a quadratic function of the coefficients w, its derivatives with respect to the coefficients will be linear in the elements of w, and so the minimization of the error function has a unique solution, denoted by w*,
- The resulting polynomial is given by the function y(x,w*).
- Choosing the order M of the polynomial ➔ model selection.

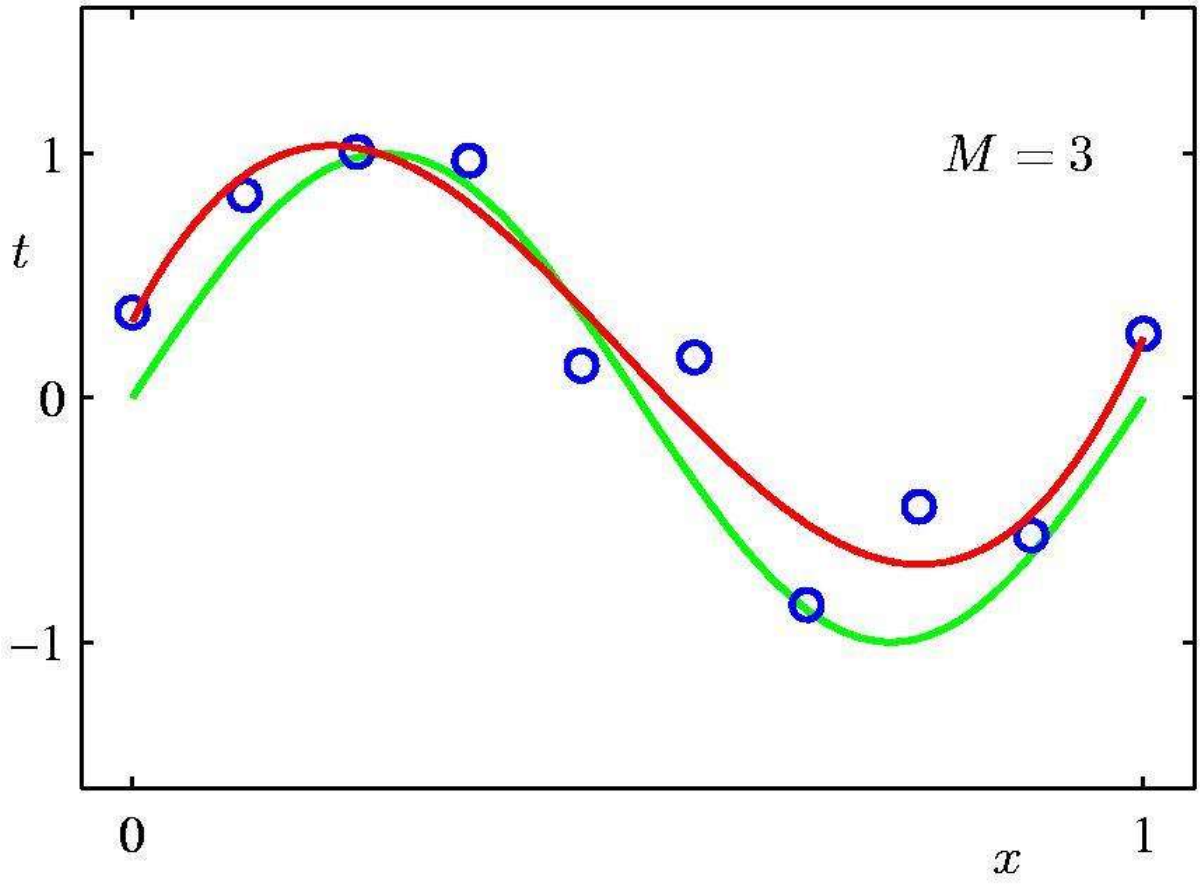
0^{th} Order Polynomial



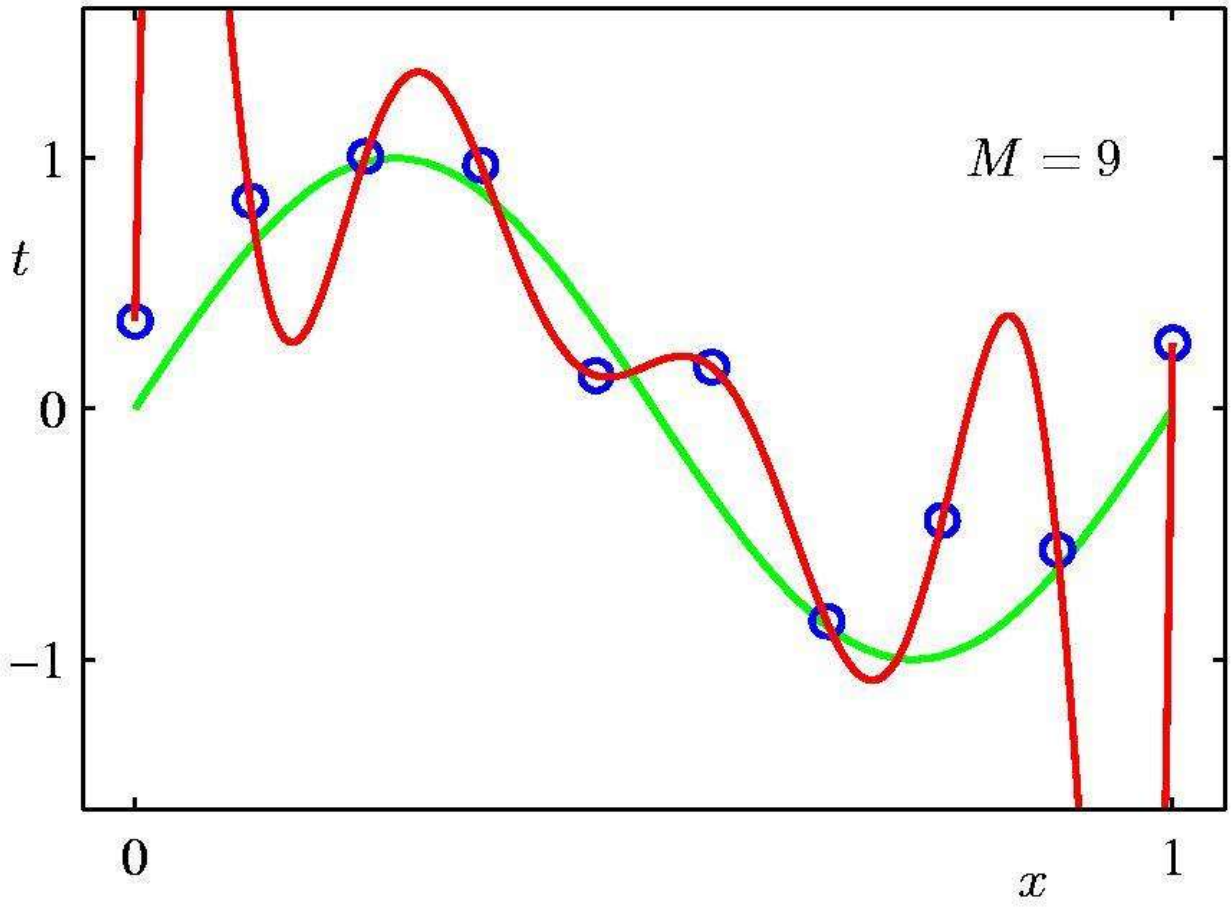
1st Order Polynomial



3rd Order Polynomial

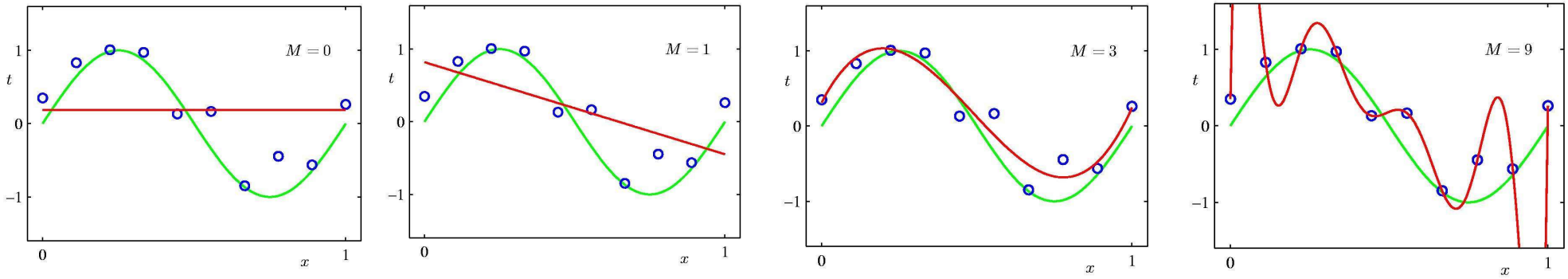


9th Order Polynomial



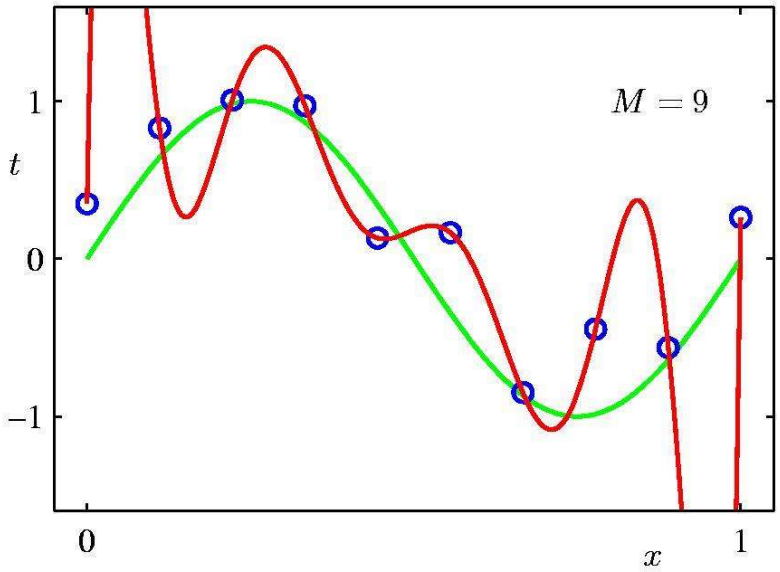
Polynomial Curve Fitting

- The 0^{th} order (M=0) and first order (M=1) polynomials give rather poor fits to the data and consequently rather poor representations of the function $\sin(2\pi x)$.
- The third order (M=3) polynomial seems to give the best fit to the function $\sin(2\pi x)$ of the examples.
- When we go to a much higher order polynomial (M=9), we obtain an excellent fit to the training data.
  - In fact, the polynomial passes exactly through each data point and $E(w^*) = 0$.



Polynomial Curve Fitting

- We obtain an excellent fit to the training data with 9^{th} order.
- However, the fitted curve oscillates wildly and gives a very poor representation of the function $\sin(2\pi x)$.

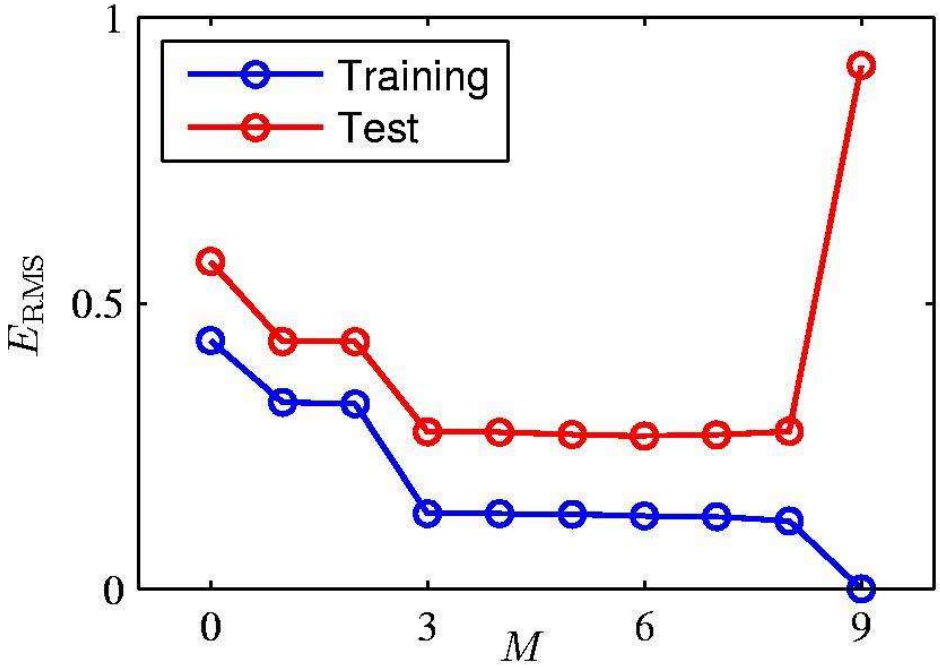


- This behaviour is known as **over-fitting**

Polynomial Curve Fitting

Over-fitting

- We can then evaluate the residual value of E(w*) for the training data, and we can also evaluate E(w*) for the test data set.
- Root-Mean-Square (RMS) Error: $E_{\mathrm{RMS}} = \sqrt{2E(\mathbf{w}^\star)/N}$
  - in which the division by N allows us to compare different sizes of data sets, and the square root ensures that E_{RMS} is measured on the same scale as the target variable t.



Polynomial Curve Fitting

Polynomial Coefficients

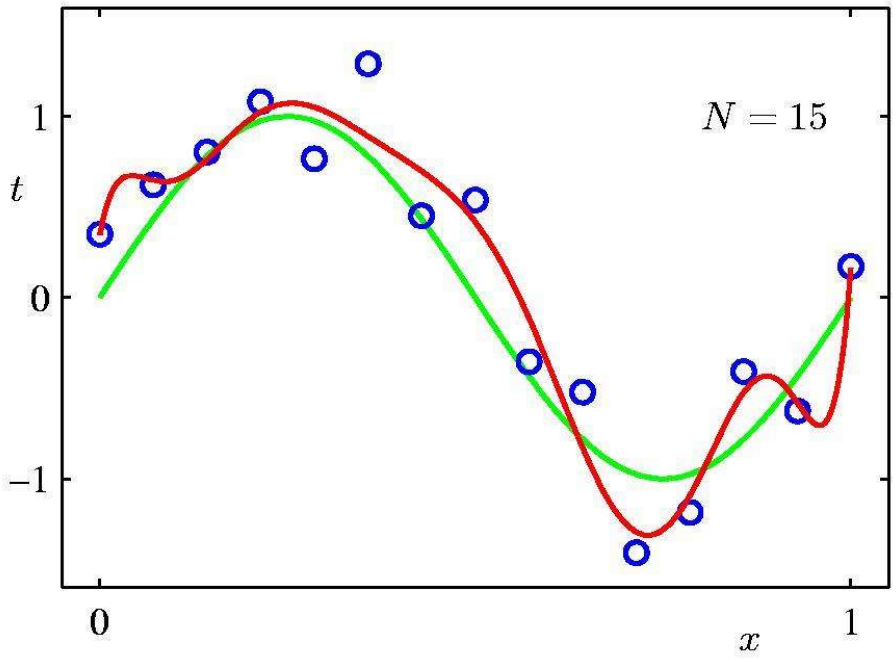
- Magnitude of coefficients increases dramatically as order of polynomial increases.
- Large positive and negative values so that the corresponding polynomial function matches each of the data points exactly, but between data points the function exhibits the large oscillations ➔ over-fitting

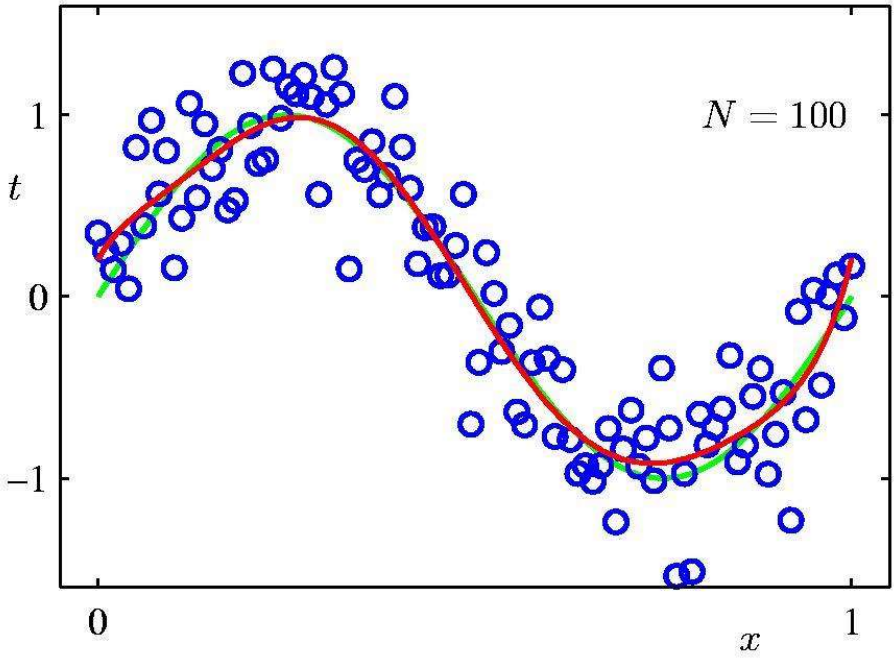
	$M=0$	$M=1$	$M=3$	$M=9$
$w_0^\star$	0.19	0.82	0.31	0.35
$w_1^\star$		-1.27	7.99	232.37
$w_2^\star$			-25.43	-5321.83
$w_3^\star$			17.37	48568.31
$w_4^\star$				-231639.30
$w_5^\star$				640042.26
$w_6^\star$				-1061800.52
$w_7^\star$				1042400.18
$w_8^\star$				-557682.99
$w_9^\star$				125201.43

Polynomial Curve Fitting

- Increasing the size of the data set reduces the over-fitting problem.

- 9th Order Polynomial.





Polynomial Curve Fitting

regularization

- We may wish to use relatively complex and flexible models with data sets of limited size.
- The over-fitting phenomenon can be controlled with **regularization**, which involves adding a penalty term to the error function.

- **Regularization:** Penalize large coefficient values

$$\widetilde{E}(\mathbf{w}) = \frac{1}{2}\sum_{n=1}^{N}\{y(x_n,\mathbf{w}) - t_n\}^2 + \frac{\lambda}{2}\|\mathbf{w}\|^2$$

$$\text{where } \|\mathbf{w}\|^2 \equiv \mathbf{w}^{\mathrm{T}}\mathbf{w} = w_0^2 + w_1^2 + \ldots + w_M^2$$

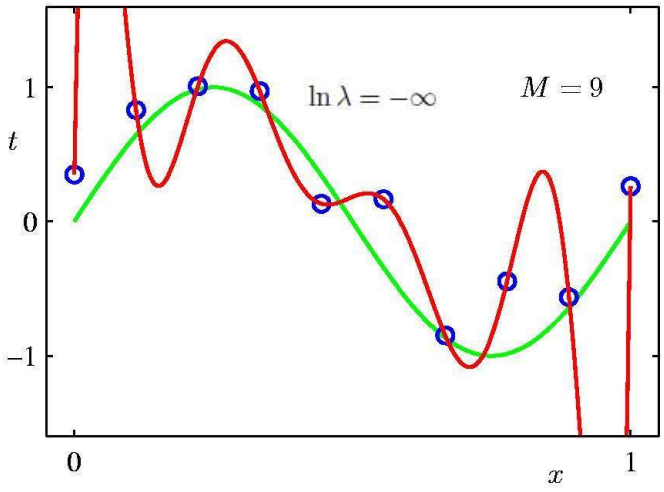
- the coefficient λ governs the relative importance of the regularization term compared with the sum-of-squares error term.

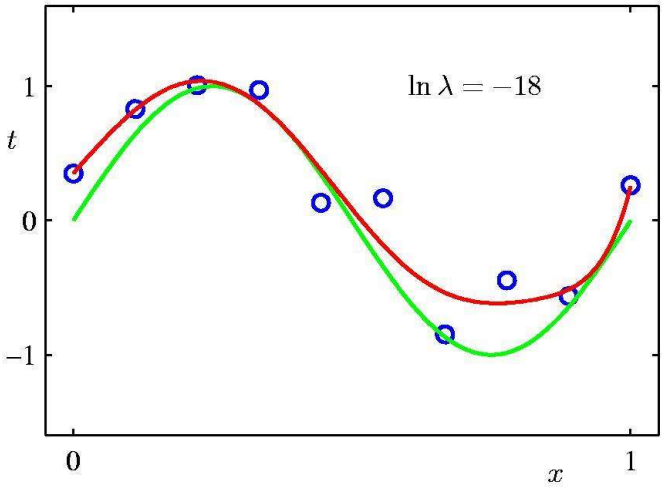
Polynomial Curve Fitting

regularization

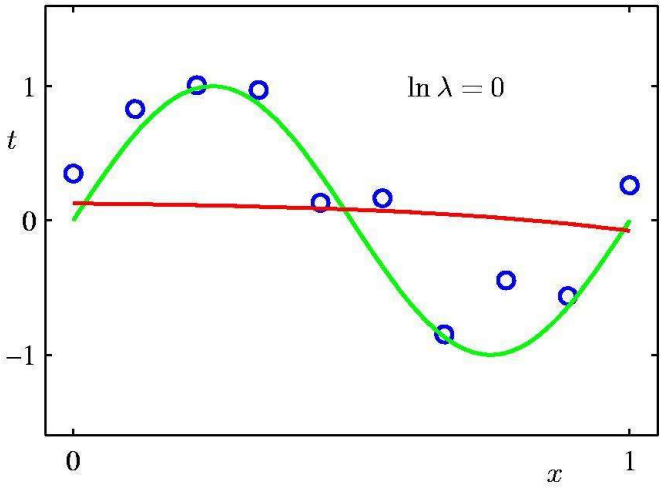
- Plots of M = 9 polynomials fitted to the data set using the regularized error function

no regularization (λ=0)





too much regularization

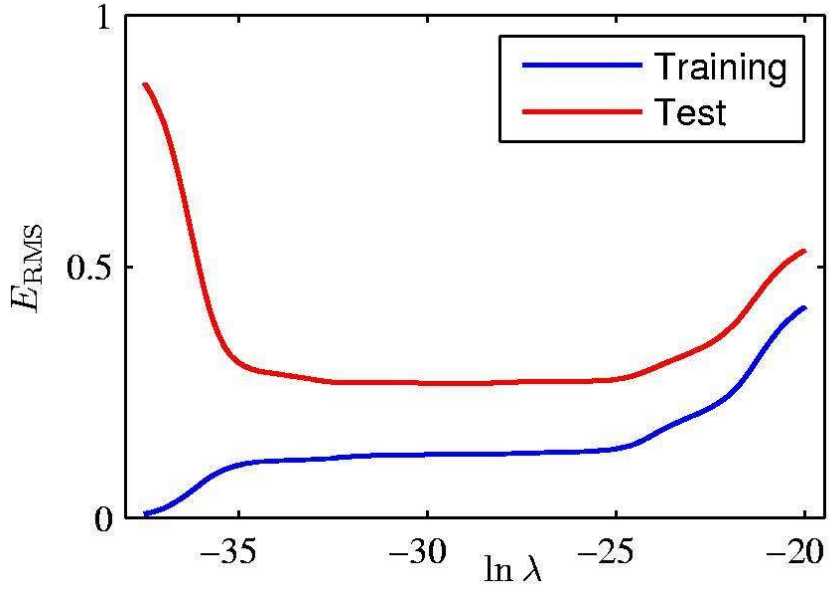


Polynomial Curve Fitting

regularization

- Polynomial Coefficients

	$\ln\lambda = -\infty$	$\ln\lambda = -18$	$\ln\lambda = 0$
$w_0^\star$	0.35	0.35	0.13
$w_1^\star$	232.37	4.74	-0.05
$w_2^\star$	-5321.83	-0.77	-0.06
$w_3^\star$	48568.31	-31.97	-0.05
$w_4^\star$	-231639.30	-3.89	-0.03
$w_5^\star$	640042.26	55.28	-0.02
$w_6^\star$	-1061800.52	41.32	-0.01
$w_7^\star$	1042400.18	-45.95	-0.00
$w_8^\star$	-557682.99	-91.53	0.00
$w_9^\star$	125201.43	72.68	0.01



Graph of the root-mean-square error versus ln λ for the M=9 polynomial.

Linear Basis Function Models

Linear Basis Function Models

- Generally

$$y(\mathbf{x}, \mathbf{w}) = \sum_{j=0}^{M-1} w_j \phi_j(\mathbf{x}) = \mathbf{w}^{\mathrm{T}} \boldsymbol{\phi}(\mathbf{x})$$

where $\phi_j(x)$ are known as **basis functions**.

Linear Basis Function Models

Linear Regression

- The simplest linear model for regression is one that involves a linear combination of the input variables.
- It is often simply known as **linear regression**.

$$y(\mathbf{x}, \mathbf{w}) = w_0 + w_1 x_1 + \ldots + w_D x_D$$

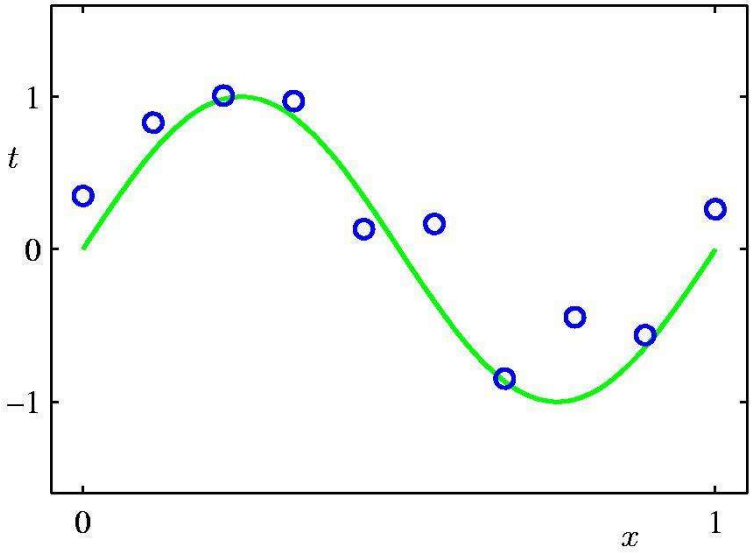
where

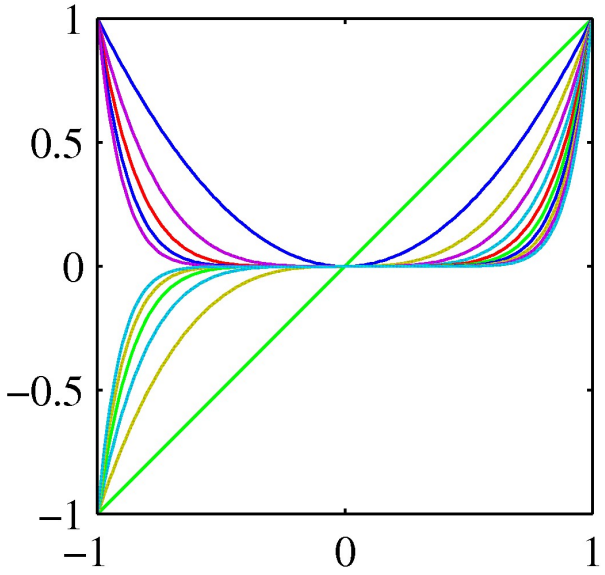
$\phi_0(x) = 1 \quad (=x_0)$

$\phi_j(x) = x_j \quad j>0$

Linear Basis Function Models

Polynomial Curve Fitting: Polynomial basis functions





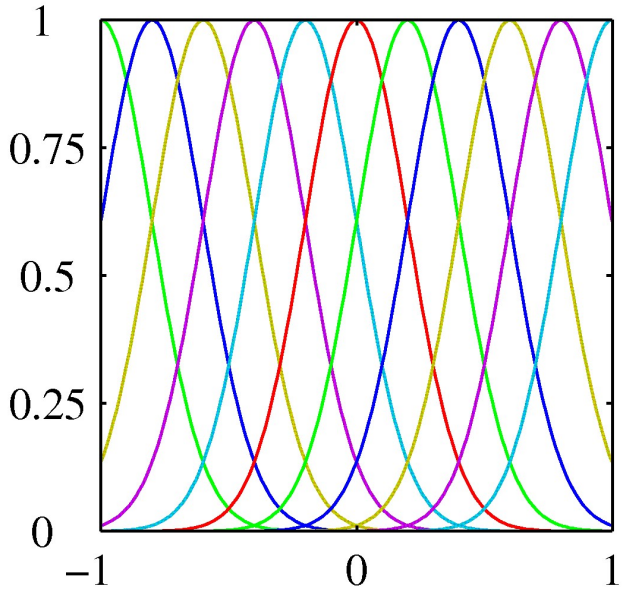
$$y(x, \mathbf{w}) = w_0 + w_1 x + w_2 x^2 + \ldots + w_M x^M = \sum_{j=0}^{M} w_j x^j$$

- $\phi_j(x) = x^j$

Linear Basis Function Models

Gaussian basis functions

$$y(\mathbf{x}, \mathbf{w}) = \sum_{j=0}^{M-1} w_j \phi_j(\mathbf{x}) = \mathbf{w}^{\mathrm{T}} \boldsymbol{\phi}(\mathbf{x})$$



$$\phi_j(x) = \exp\left\{-\frac{(x-\mu_j)^2}{2s^2}\right\}$$

where the μ_j govern the locations of the basis functions in input space, and the parameter s governs their spatial scale.

Linear Basis Function Models

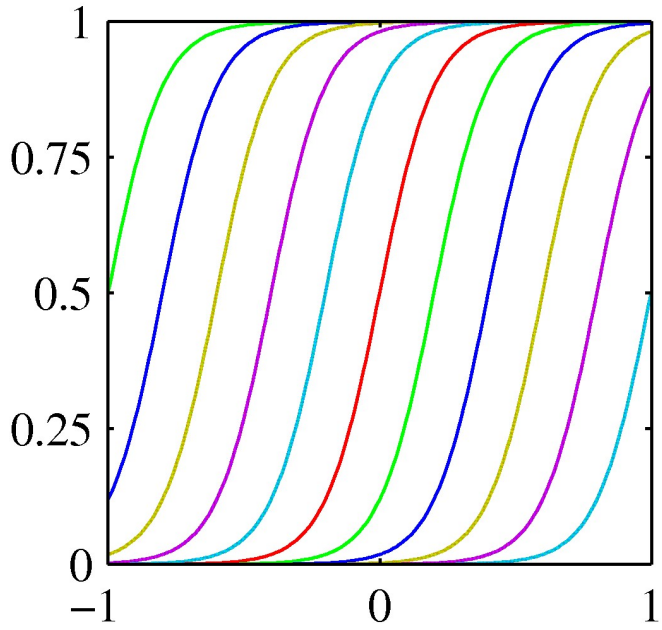
Sigmoidal basis functions

$$y(\mathbf{x}, \mathbf{w}) = \sum_{j=0}^{M-1} w_j \phi_j(\mathbf{x}) = \mathbf{w}^{\mathrm{T}} \boldsymbol{\phi}(\mathbf{x})$$

$$\phi_j(x) = \sigma\left(\frac{x - \mu_j}{s}\right)$$

where σ(a) is the logistic sigmoid function defined by

$$\sigma(a) = \frac{1}{1 + \exp(-a)}.$$



- we can use the 'tanh' function because this is related to the logistic sigmoid by tanh(a) = 2σ(a) − 1,

INSTANCE-BASE LEARNING

INSTANCE-BASE LEARNING

- Instance-based learning methods simply store the training examples instead of learning explicit description of the target function.
  - Generalizing the examples is postponed until a new instance must be classified.
  - When a new instance is encountered, its relationship to the stored examples is examined in order to assign a target function value for the new instance.
- Instance-based learning includes ***nearest neighbor***, ***locally weighted regression*** and ***case-based reasoning*** methods.
- Instance-based methods are sometimes referred to as **lazy** learning methods because they delay processing until a new instance must be classified.
- A key advantage of lazy learning is that instead of estimating the target function once for the entire instance space, these methods can estimate it locally and differently for each new instance to be classified.

k-Nearest Neighbor Learning

- k-Nearest Neighbor Learning algorithm assumes all instances correspond to points in the n-dimensional space $\mathcal{R}^n$
- The nearest neighbors of an instance are defined in terms of Euclidean distance.
- Euclidean distance between the instances $x_i = \langle x_{i1},\ldots,x_{in}\rangle$ and $x_j = \langle x_{j1},\ldots,x_{jn}\rangle$ are:

$$d(x_i, x_j) = \sqrt{\sum_{r=1}^{n} (x_{ir} - x_{jr})^2}$$

- For a given query instance x_q, $f(x_q)$ is calculated the function values of k-nearest neighbor of x_q

k-Nearest Neighbor Learning

- Store all training examples $<x_i, f(x_i)>$
- Calculate $f(x_q)$ for a given query instance x_q using k-nearest neighbor
- **Nearest neighbor: (k=1)**
  - Locate the nearest traing example x_n, and estimate $f(x_q)$ as
  - $f(x_q) \leftarrow f(x_n)$
- **k-Nearest neighbor:**
  - Locate k nearest traing examples, and estimate $f(x_q)$ as
  - If the target function is real-valued, take mean of f-values of k nearest neighbors.

$$f(x_q) = \frac{\sum_{i=1}^{k} f(x_i)}{k}$$

  - If the target function is discrete-valued, take a vote among f-values of k nearest neighbors.

When To Consider Nearest Neighbor

- Instances map to points in R^n
- Less than 20 attributes per instance
- Lots of training data

- **Advantages**
  - Training is very fast
  - Learn complex target functions
  - Can handle noisy data
  - Does not loose any information

- **Disadvantages**
  - Slow at query time
  - Easily fooled by irrelevant attributes

Distance-Weighted kNN

Might want weight nearer neighbors more heavily...

$$\hat{f}(x_q) \leftarrow \frac{\sum_{i=1}^{k} w_i f(x_i)}{\sum_{i=1}^{k} w_i}$$

where

$$w_i \equiv \frac{1}{d(x_q, x_i)^2}$$

and $d(x_q, x_i)$ is distance between x_q and x_i

Note now it makes sense to use *all* training examples instead of just k

k-Nearest Neighbor Classification - Example

A	B	Class
1	1	no
2	1	no
3	2	yes
7	7	yes
8	8	yes

3-Nearest Neighbor Classification of instance <3,3>

A	B	Distance of <3,3>
1	1	$\sqrt{8}$
2	1	$\sqrt{5}$
3	2	$\sqrt{1}$
7	7	$\sqrt{32}$
8	8	$\sqrt{50}$

- First three example are 3 Nearest Neighbors of instance <3,3>.
- Two of them is **no** and one of them is **yes**.
- Majority of classes of its neighbors are **no**, the classification of instance <3,3> is **no**.

Distance Weighted kNN Classification- Example

A	B	Class
1	1	no
2	1	no
3	2	yes
7	7	yes
8	8	yes

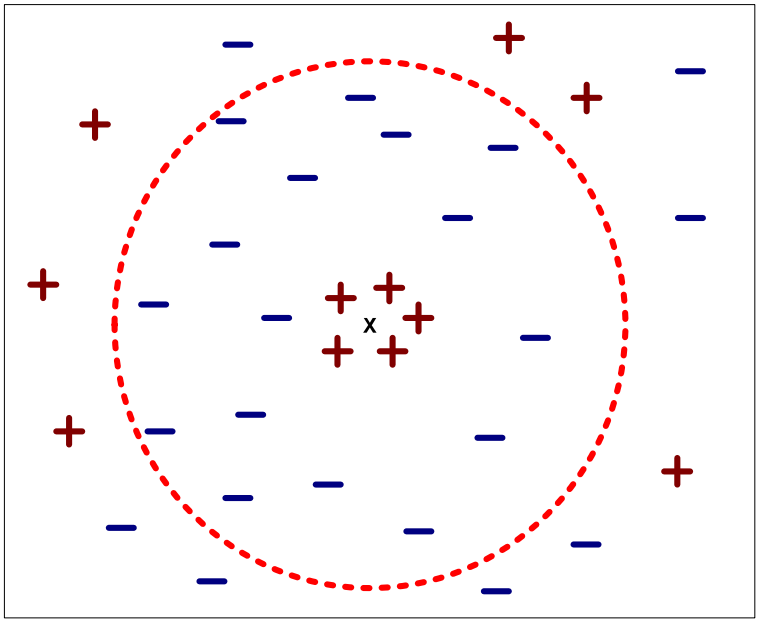
Distance Weighted 3-Nearest Neighbor Classification of instance <3,3>

A	B	Distance of <3,3>
1	1	$\sqrt{8}$
2	1	$\sqrt{5}$
3	2	$\sqrt{1}$
7	7	$\sqrt{32}$
8	8	$\sqrt{50}$

- First three example are 3 Nearest Neighbors of instance <3,3>.
- Weight of **no** = 1/8 + 1/5 = 13/40 Weight of **yes** = 1/1 = 1
- Since 1 > 13/40, the classification of instance <3,3> is **yes**.

k-Nearest Neighbor Classification - Issues

- **Choosing the value of k:**
  - If k is too small, sensitive to noise points
  - If k is too large, neighborhood may include points from other classes.



- **Scaling issues:**
  - Attributes may have to be scaled to prevent distance measures from being dominated by one of the attributes

Curse of Dimensionality

Imagine instances described by 20 attributes, but only 2 are relevant to target function

Curse of dimensionality: nearest nbr is easily mislead when high-dimensional X

One approach:

- Stretch jth axis by weight z_j, where $z_1, \ldots, z_n$ chosen to minimize prediction error
- Use cross-validation to automatically choose weights $z_1, \ldots, z_n$
- Note setting z_j to zero eliminates this dimension altogether

Locally Weighted Regression

- KNN forms local approximation to f for each query point x_q
- Why not form an explicit approximation f(x) for region surrounding x_q

  ➔ Locally Weighted Regression
- **Locally weighted regression** uses nearby or distance-weighted training examples to form this local approximation to f.
- We might approximate the target function in the neighborhood surrounding x, using a linear function, a quadratic function, a multilayer neural network.
- The phrase "**locally weighted regression**" is called
  - ***local*** because the function is approximated based only on data near the query point,
  - ***weighted*** because the contribution of each training example is weighted by its distance from the query point, and
  - ***regression*** because this is the term used widely in the statistical learning community for the problem of approximating real-valued functions.

Locally Weighted Regression

- Given a new query instance x_q, the general approach in locally weighted regression is to construct an approximation f that fits the training examples in the neighborhood surrounding x_q.
- This approximation is then used to calculate the value $f(x_q)$, which is output as the estimated target value for the query instance.

Case-based reasoning

- Instance-based methods
  - lazy
  - classification based on classifications of near (similar) instances
  - data: points in n-dim. space
- Case-based reasoning
  - as above, but data represented in symbolic form
  - new distance metrics required

Lazy & eager learning

- Lazy: generalize at query time
  - kNN, CBR
- Eager: generalize before seeing query
  - regression, ann, ID3, …
- Difference
  - eager *must* create global approximation
  - lazy *can* create many local approximation
  - lazy can represent more complex functions using same H (H = linear functions)