UVA CS 4501 - 001 / 6501 – 007 Introduction to Machine Learning and Data Mining

Lecture 5: Newton's Method and Non-Linear Regression Models

Yanjun Qi / Jane

University of Virginia
Department of
Computer Science

Last Lecture Recap

- Three Ways to train / perform optimization for linear regression models
  - Normal Equation
  - Gradient Descent (GD)
  - Stochastic GD

Linear Regression Models

$$\boxed{f}: X \longrightarrow Y$$

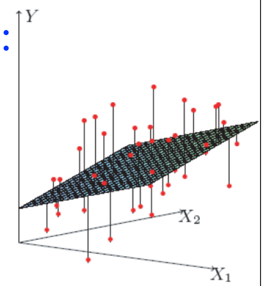
➔ e.g. Linear Regression Models

$$\hat{y} = f(x) = \theta_0 + \theta_1 x^1 + \theta_2 x^2$$

➔ To minimize the cost function:

$$J(\theta) = \frac{1}{2}\sum_{i=1}^{n}(\hat{y}_i(\vec{x}_i) - y_i)^2$$

$\rightarrow \theta$



Method I: normal equations

- Write the cost function in matrix form:

$$\begin{aligned} J(\theta) &= \frac{1}{2}\sum_{i=1}^{n}(\mathbf{x}_i^T\theta - y_i)^2 \\ &= \frac{1}{2}(X\theta - \vec{y})^T(X\theta - \vec{y}) \\ &= \frac{1}{2}(\theta^T X^T X\theta - \theta^T X^T\vec{y} - \vec{y}^T X\theta + \vec{y}^T\vec{y}) \end{aligned}$$

$$\mathbf{X} = \begin{bmatrix} -- & \mathbf{x}_1^T & -- \\ -- & \mathbf{x}_2^T & -- \\ \vdots & \vdots & \vdots \\ -- & \mathbf{x}_n^T & -- \end{bmatrix} \qquad \vec{y} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix}$$

- To minimize $J(\theta)$, take derivative and set to zero:

$$\Rightarrow \boxed{X^T X\theta = X^T\vec{y}}$$

The normal equations

$$\Downarrow$$

$$\theta^* = (X^T X)^{-1} X^T\vec{y}$$

Method II: LR with batch Steepest descent / Gradient descent

$$\theta_t = \theta_{t-1} - \alpha \nabla J(\theta_{t-1})$$

For the t-th epoch

$$\nabla_\theta J = \left[\frac{\partial}{\partial \theta_1} J, \ldots, \frac{\partial}{\partial \theta_k} J\right]^T = -\sum_{i=1}^{n} (y_n - \mathbf{x}_n{}^T \theta)\mathbf{x}_n$$

$$\theta^{t+1} = \theta^t + \alpha \sum_{i=1}^{n} (y_n - \mathbf{x}_n^T \theta^t)\mathbf{x}_n$$

– This is as a **batch** gradient descent algorithm

$$\nabla_\theta J(\theta) = X^T X \theta - X^T Y$$

$$= X^T (X\theta - Y)$$

$$= X^T \left(\underset{n \times p}{\begin{bmatrix} - x_1^T - \\ - x_2^T - \\ \cdots \\ - x_n^T - \end{bmatrix}} \underset{p \times 1}{\theta} - \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix} \right)$$

$$\mathbf{X} = \begin{bmatrix} -- & \mathbf{x}_1^T & -- \\ -- & \mathbf{x}_2^T & -- \\ \vdots & \vdots & \vdots \\ -- & \mathbf{x}_n^T & -- \end{bmatrix}$$

$$= \underset{p \times n}{X^T} \underset{n \times 1}{\begin{bmatrix} x_1^T \theta - y_1 \\ x_2^T \theta - y_2 \\ \cdots \\ x_n^T \theta - y_n \end{bmatrix}} = \begin{bmatrix} | & | & & | \\ x_1 & x_2 & \cdots & x_n \\ | & | & & | \end{bmatrix} \begin{bmatrix} x_1^T\theta - y_1 \\ \vdots \\ x_n^T\theta - y_n \end{bmatrix}$$

$$= \sum_{i=1}^{n} x_i \underbrace{(x_i^T \theta - y_i)}_{1*1}$$

Method III: LR with Stochastic GD ➔

- From the batch steepest descent rule:

$$\theta_j^{t+1} = \theta_j^{t} + \alpha \sum_{i=1}^{n} (y_i - \vec{\mathbf{x}}_i^{T} \theta^t) x_i^j$$

- For a single training point, we have:

$$\theta^{t+1} = \theta^{t} + \alpha (y_i - \vec{\mathbf{x}}_i^{T} \theta^t) \vec{\mathbf{x}}_i$$

  - This is known as the Least-Mean-Square update rule, or the Widrow-Hoff learning rule
  - This is actually a "**stochastic**", "**coordinate**" descent algorithm
  - This can be used as a **on-line** algorithm

Summary: three ways to learn LR

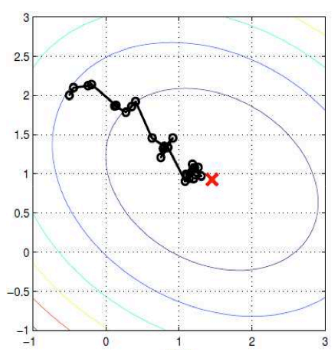
- Normal equations $\theta^* = \left(X^T X\right)^{-1} X^T \vec{y}$
  - Pros: a single-shot algorithm! Easiest to implement.
  - Cons: need to compute pseudo-inverse $(X^T X)^{-1}$, expensive, numerical issues (e.g., matrix is singular ..), although there are ways to get around this …
- GD or Steepest descent $\theta^{t+1} = \theta^{t} + \alpha \sum_{i=1}^{n} (y_n - \mathbf{x}_n^{T} \theta^t) \mathbf{x}_n$
  - Pros: easy to implement, conceptually clean, guaranteed convergence
  - Cons: batch, often slow converging
- Stochastic LMS update rule $\theta_j^{t+1} = \theta_j^{t} + \alpha (y_n - \mathbf{x}_n^{T} \theta^t) x_{n,j}$
  - Pros: on-line, low per-step cost, fast convergence and perhaps less prone to local optimum
  - Cons: convergence to optimum not always guaranteed

Today

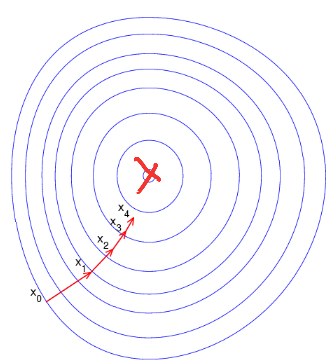
- ❑ More optimization:
  - ❑ Stochastic gradient descent
  - ❑ Newton's method
- ❑ Regression Models Beyond Linear
  - – LR with non-linear basis functions
  - – Locally weighted linear regression
  - – Regression trees and Multilinear Interpolation

Stochastic gradient descent / Online Learning Algorithm

SGD **GD**



versus



Stochastic gradient descent : More variations

- Single-sample:

$$\theta^{t+1} = \theta^t + \alpha (y_i - \vec{x}_i^T \theta^t) \vec{x}_i$$

- Mini-batch:

$$\theta^{t+1} = \theta^t + \alpha \sum_{j=1}^{B} (y_j - \vec{x}_j^T \theta^t) \vec{x}_j$$

e.g. $B = 15$

Stochastic gradient descent

SGD can also be used for offline learning, by repeatedly cycling through the data; each such pass over the whole dataset is called an **epoch**. This is useful if we have **massive datasets** that will not fit in main memory. In this offline case, it is often better to compute the gradient of a **mini-batch** of B data cases. If $B = 1$, this is standard SGD, and if $B = N$, this is standard steepest descent. Typically $B \sim 100$ is used.

Intuitively, one can get a fairly good estimate of the gradient by looking at just a few examples. Carefully evaluating precise gradients using large datasets is often a waste of time, since the algorithm will have to recompute the gradient again anyway at the next step. It is often a better use of computer time to have a noisy estimate and to move rapidly through parameter space.

SGD is often less prone to getting stuck in shallow local minima, because it adds a certain amount of "noise". Consequently it is quite popular in the machine learning community for fitting models such as neural networks and deep belief networks with non-convex objectives.

Nando de Freitas's tutorial slide

Today

- ❑ More optimization:
  - ❑ Stochastic gradient descent
  - ❑ Newton's method
- ❑ Regression Models Beyond Linear
  - – LR with non-linear basis functions
  - – Locally weighted linear regression
  - – Regression trees and Multilinear Interpolation

Yanjun Qi / UVA CS 4501-01-6501-07

Newton's method for optimization

- The most basic second-order optimization algorithm
- Updating parameter with

$$\boldsymbol{\theta}_{k+1} = \boldsymbol{\theta}_k - \mathbf{H}_K^{-1}\mathbf{g}_k$$

Review: Hessian Matrix / d==2 case

- 1st derivative to gradient,
  2nd derivative to Hessian

$$f(x,y)$$

$$g = \nabla f = \begin{pmatrix} \frac{\partial f}{\partial x} \\ \frac{\partial f}{\partial y} \end{pmatrix}$$

$$H = \begin{pmatrix} \frac{\partial^2 f}{\partial x^2} & \frac{\partial^2 f}{\partial x \partial y} \\ \frac{\partial^2 f}{\partial x \partial y} & \frac{\partial^2 f}{\partial y^2} \end{pmatrix}$$

Review: Hessian Matrix

Suppose that $f : \mathbb{R}^n \to \mathbb{R}$ is a function that takes a vector in $\mathbb{R}^n$ and returns a real number. Then the ***Hessian*** matrix with respect to x, written $\nabla_x^2 f(x)$ or simply as H is the $n \times n$ matrix of partial derivatives,

$$\nabla_x^2 f(x) \in \mathbb{R}^{n\times n} = \begin{bmatrix} \frac{\partial^2 f(x)}{\partial x_1^2} & \frac{\partial^2 f(x)}{\partial x_1 \partial x_2} & \cdots & \frac{\partial^2 f(x)}{\partial x_1 \partial x_n} \\ \frac{\partial^2 f(x)}{\partial x_2 \partial x_1} & \frac{\partial^2 f(x)}{\partial x_2^2} & \cdots & \frac{\partial^2 f(x)}{\partial x_2 \partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 f(x)}{\partial x_n \partial x_1} & \frac{\partial^2 f(x)}{\partial x_n \partial x_2} & \cdots & \frac{\partial^2 f(x)}{\partial x_n^2} \end{bmatrix}.$$

Newton's method for optimization

- Making a quadratic/second-order Taylor series approximation

$$\hat{f}_{quad}(\boldsymbol{\theta}) = f(\boldsymbol{\theta}_k) + \mathbf{g}_k^T(\boldsymbol{\theta} - \boldsymbol{\theta}_k) + \frac{1}{2}(\boldsymbol{\theta} - \boldsymbol{\theta}_k)^T \mathbf{H}_k(\boldsymbol{\theta} - \boldsymbol{\theta}_k)$$

the minimum solution of the above right quadratic approximation (quadratic function minimization is easy !)

$$\hat{f}(\theta) = f(\theta_k) + g_k^T(\theta - \theta_k) + \underbrace{\frac{1}{2}(\theta-\theta_k)^T H_k (\theta - \theta_k)}$$

$$\Downarrow$$

$$\frac{1}{2}\left(\theta^T H_k \theta - 2\theta^T H_k \theta_k + \theta_k^T H_k \theta_k\right)$$

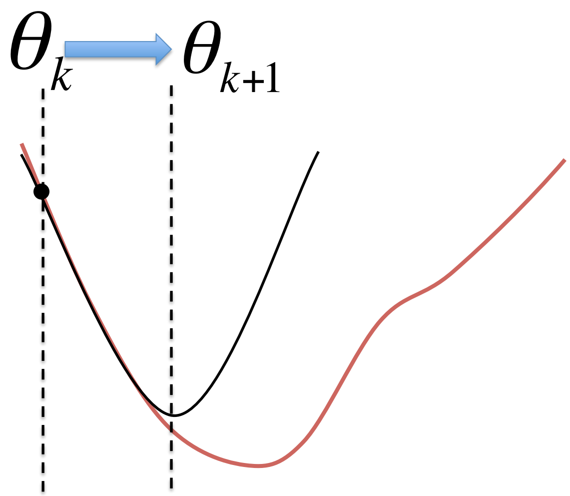
$$\frac{\partial \hat{f}(\theta)}{\partial \theta} = 0 + g_k + \underbrace{\frac{2}{2} H_k \theta - \frac{2}{2} H_k \theta_k}_{\text{see p24 handout}} := 0$$

$$g_k + H_k(\theta - \theta_k) = 0$$

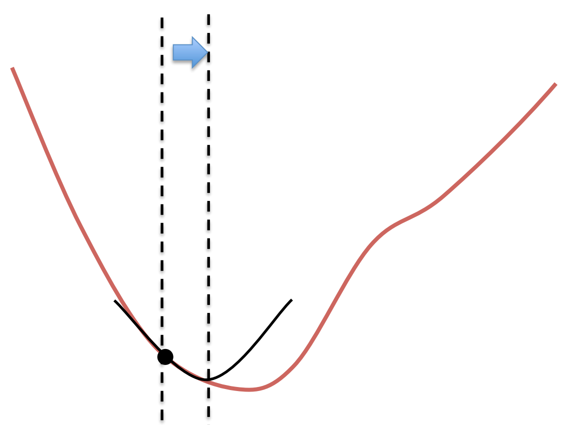
$$\Rightarrow \quad \theta = \theta_k - H_k^{-1} g_k$$

where $\begin{cases} H_k \in \mathbb{R}^{p \times p} \\ g_k \in \mathbb{R}^p \end{cases}$

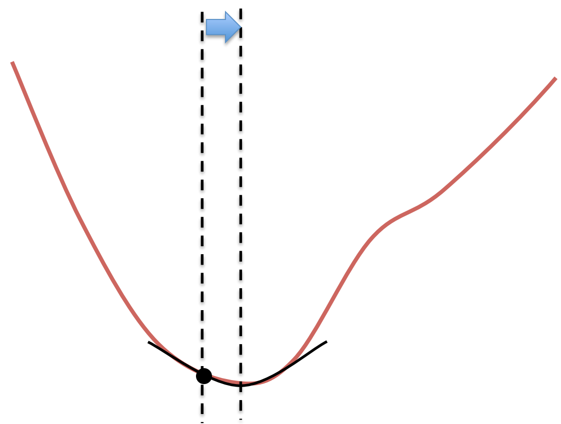
Newton's Method / second-order Taylor series approximation



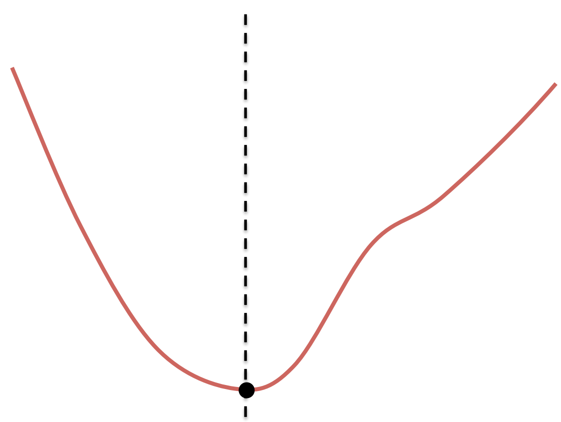
Newton's Method / second-order Taylor series approximation



Newton's Method / second-order Taylor series approximation



Newton's Method / second-order Taylor series approximation



Newton's Method

- At each step:

$$\theta_{k+1} = \theta_k - \frac{f'(\theta_k)}{f''(\theta_k)}$$

$$\theta_{k+1} = \theta_k - H^{-1}(\theta_k)\nabla f(\theta_k)$$

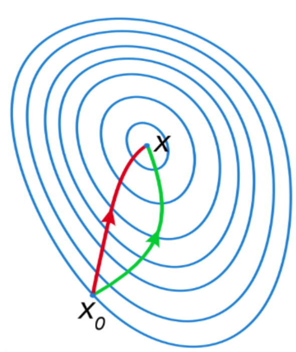
- Requires 1st and 2nd derivatives
- Quadratic convergence
- ➔ However, finding the inverse of the Hessian matrix is often expensive

Comparison

- Newton's method vs. Gradient descent

A comparison of gradient descent (green) and Newton's method (red) for minimizing a function (with small step sizes).

Newton's method uses curvature information to get a more direct route …



$$J(\theta) = \frac{1}{2}(y - X\theta)^T(y - X\theta)$$

$$\nabla J(\theta) = X^T X \theta - X^T \vec{y}$$

$$H = \nabla^2_\theta J(\theta) = X^T X \quad (P24)$$

$$\Rightarrow \theta^t = \theta^t - H^{-1} \nabla f(\theta)$$

$$= \theta^t - (X^T X)^{-1}[X^T X \theta^t - X^T \vec{y}]$$

$$= (X^T X)^{-1} X^T y$$

WHY ???
Normal Eq?

Newton's method for Linear Regression

Today

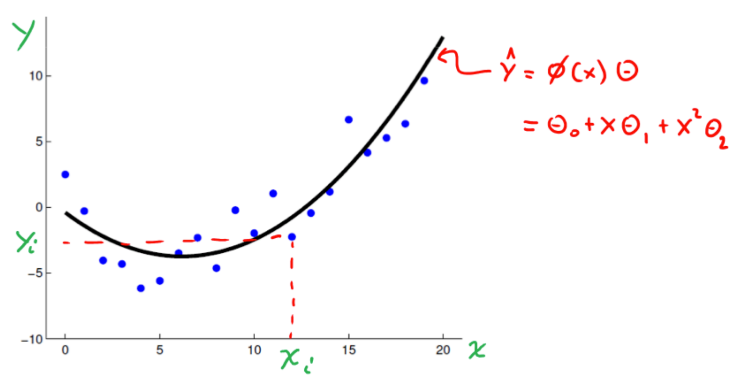
- ❑ More optimization:
- ❑ Regression Models Beyond Linear
  - LR with non-linear basis functions
  - Locally weighted linear regression
  - Regression trees and Multilinear Interpolation

Beyond basic LR

- Linear model is an approximation
- Three ways to moving beyond linearity
  - LR with non-linear basis functions
  - Locally weighted linear regression
  - Regression trees and Multilinear Interpolation (later)

e.g. **polynomial regression**

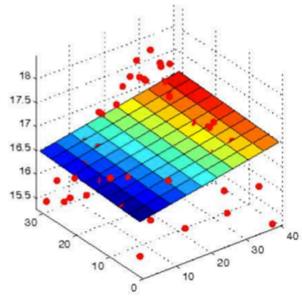
For example, $\phi(x) = [1, x, x^2]$

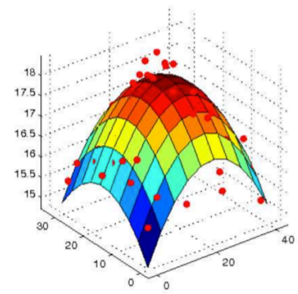


Dr. Nando de Freitas's tutorial slide

e.g. **polynomial regression**

$\phi(\mathbf{x}) = [1, x_1, x_2]$ $\phi(\mathbf{x}) = [1, x_1, x_2, x_1^2, x_2^2]$





KEY: if the bases are given, the problem of learning the parameters is still linear.

LR with non-linear basis functions

- LR does not mean we can only deal with linear relationships

$$y = \theta_0 + \sum_{j=1}^{m} \theta_j \phi(x) = \theta^T \phi(x)$$

- We are free to design (non-linear) features (e.g., basis function derived) under LR

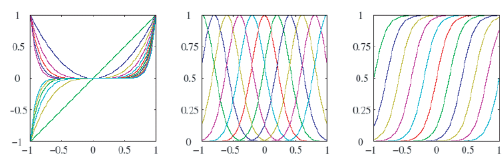
where the $\phi_j(x)$ are fixed basis functions (also define $\phi_0(x) = 1$).

- E.g.: polynomial regression:

$$\phi(x) := \left[1, x, x^2, x^3\right]$$

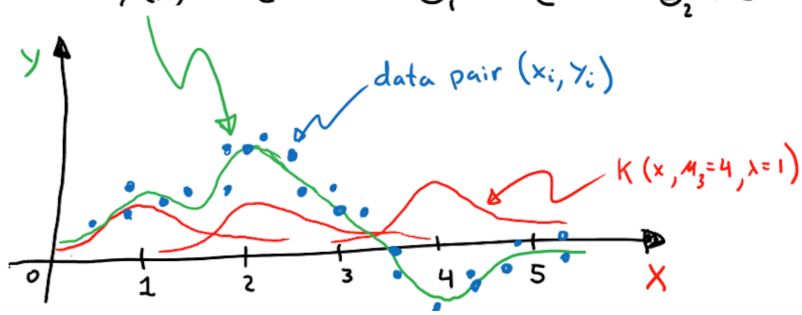
Many Possible Basis functions

- There are many basis functions, e.g.:
  - Polynomial $\varphi_j(x) = x^{j-1}$
  - Radial basis functions $\phi_j(x) = \exp\left(-\frac{(x-\mu_j)^2}{2s^2}\right)$
  - Sigmoidal $\phi_j(x) = \sigma\left(\frac{x-\mu_j}{s}\right)$
  - Splines,
  - Fourier,
  - Wavelets, etc



e.g. nonlinear regression with predefined RBF basis functions

$$\hat{Y}(x) = e^{-\|x-1\|^2}\theta_1 + e^{-\|x-2\|^2}\theta_2 + e^{-\|x-4\|^2}\theta_3$$



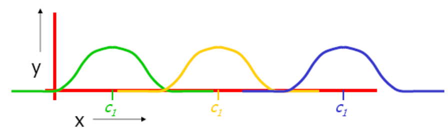
[green curve to fit train data points]

green curve is linear weighted sum of red curves

Dr. Nando de Freitas's tutorial slide

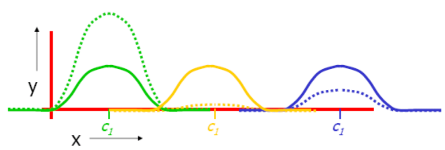
1D and 2D RBFs

- 1D RBF



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

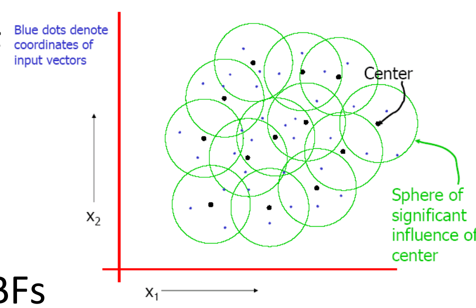
- After fit:



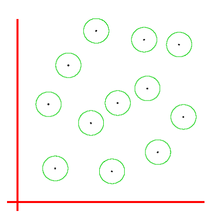
$y^{est} = 2\phi_1(x) + 0.05\phi_2(x) + 0.5\phi_3(x)$

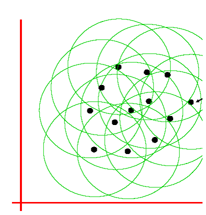
Good and Bad RBFs

- A good 2D RBF



- Two bad 2D RBFs



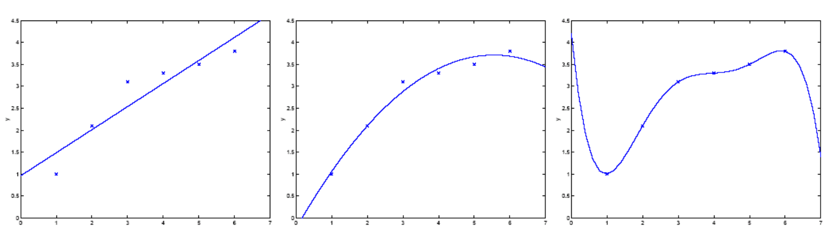


Two main issues:

- Learn the parameter \theta
  - Almost the same as LR, just ➔ X to $\varphi(x)$
  - Linear combination of basis functions (that can be non-linear)

- Choose the model order, e.g. polynomial degree for polynomial regression

Issue: Overfitting and underfitting



$y = \theta_0 + \theta_1 x$ $y = \theta_0 + \theta_1 x + \theta_2 x^2$ $y = \sum_{j=0}^{5} \theta_j x^j$

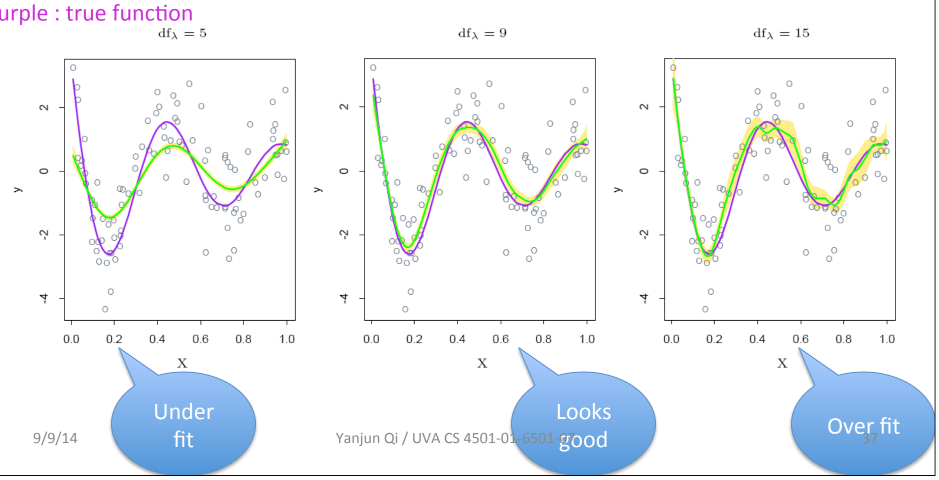
Generalisation: learn function / hypothesis from past data in order to "explain", "predict", "model" or "control" new data examples

K-fold Cross Validation !!!!

Issue: Over-fitting and under-fitting

- Page 159 of “Elements of SL” book, Figure 5.9

Purple : true function



Today

- ❑ More optimization:
- ❑ Regression Models Beyond Linear
  - LR with non-linear basis functions
  - Locally weighted linear regression
  - Regression trees and Multilinear Interpolation (later)

(2) Locally weighted linear regression

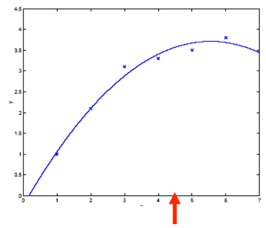
- The algorithm:

  Instead of minimizing $J(\theta) = \frac{1}{2}\sum_{i=1}^{n}(\mathbf{x}_i^T\theta - y_i)^2$

  now we fit θ to minimize $J(\theta) = \frac{1}{2}\sum_{i=1}^{n} w_i(\mathbf{x}_i^T\theta - y_i)^2$

  Where do w_i's come from? $w_i = K(\mathbf{x}_i, \mathbf{x}0) = \exp\left(-\frac{(\mathbf{x}_i - \mathbf{x}0)^2}{2\tau^2}\right)$

  - where $\mathbf{x0}$ is the query point for which we'd like to know its corresponding $\mathbf{y}$



→ Essentially we put higher weights on (errors on) training examples that are close to the query point x0 (than those that are further away from the query)

Locally weighted regression

Locally weighted regression solves a separate weighted least squares problem at each target point x0

$$\min_{\alpha(x_0),\beta(x_0)} \sum_{i=1}^{N} K_\lambda(x_0, x_i)\,[y_i - \alpha(x_0) - \beta(x_0)x_i]^2$$

The estimate is then $\hat{f}(x_0) = \hat{\alpha}(x_0) + \hat{\beta}(x_0)x_0$.

e.g. when for only one feature variable

Define the vector-valued function $b(x)^T = (1, x)$. Let $\mathbf{B}$ be the $N \times 2$ regression matrix with ith row $b(x_i)^T$, and $\mathbf{W}(x_0)$ the $N \times N$ diagonal matrix with ith diagonal element $K_\lambda(x_0, x_i)$. Then

LWR $\quad \hat{f}(x_0) = b(x_0)^T(\mathbf{B}^T\mathbf{W}(x_0)\mathbf{B})^{-1}\mathbf{B}^T\mathbf{W}(x_0)\mathbf{y}$

versus

LR $\quad f(x_q) = (x_q)^T\theta^* = (x_q)^T\left(X^TX\right)^{-1}X^T\vec{y}$

Parametric vs. non-parametric

- Locally weighted linear regression is a **non-parametric** algorithm.
- The (unweighted) linear regression algorithm that we saw earlier is known as a **parametric** learning algorithm
  - because it has a fixed, finite number of parameters (the *theta*), which are fit to the data;
  - Once we've fit the *theta* and stored them away, we no longer need to keep the training data around to make future predictions.
  - In contrast, to make predictions using locally weighted linear regression, we need to keep the entire training set around.
- The term "non-parametric" (roughly) refers to the fact that the amount of stuff we need to keep in order to represent the hypothesis grows with linearly the size of the training set.

Today's Recap

- ❑ More optimization for LR:
  - ❑ Stochastic gradient descent
  - ❑ Newton's method
- ❑ Regression Models Beyond Linear
  - LR with non-linear basis functions
  - Locally weighted linear regression

References

- Big thanks to Prof. Eric Xing @ CMU for allowing me to reuse some of his slides

❑ Prof. Nando de Freitas's tutorial slide