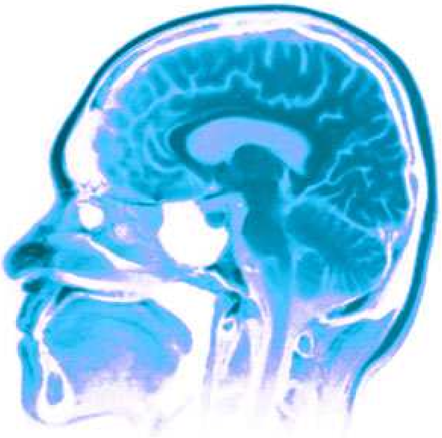
CPSC540



Optimization I

Nando de Freitas

2011

KPM Book Sections: *11.2*

Revision: Gradient vector

- Let $\mathbf{x}$ be an n-dimensional vector, and $f(\mathbf{x})$ a scalar-valued function. The gradient vector of f with respect to $\mathbf{x}$ is the following vector:

$$\nabla_{\mathbf{x}} f(\mathbf{x}) = \begin{bmatrix} \frac{\partial f(\mathbf{x})}{\partial x_1} \\ \frac{\partial f(\mathbf{x})}{\partial x_2} \\ \vdots \\ \frac{\partial f(\mathbf{x})}{\partial x_n} \end{bmatrix}$$

Revision: Hessian matrix

- The **Hessian** matrix of a scalar valued function with respect to $\mathbf{x}$, written $\nabla^2_{\mathbf{x}} f(\mathbf{x})$ or simply as $\mathbf{H}$, is the $n \times n$ matrix of partial derivatives,

$$\nabla^2_{\mathbf{x}} f(\mathbf{x}) = \begin{bmatrix} \frac{\partial^2 f(\mathbf{x})}{\partial x_1^2} & \frac{\partial^2 f(\mathbf{x})}{\partial x_1 \partial x_2} & \cdots & \frac{\partial^2 f(\mathbf{x})}{\partial x_1 \partial x_n} \\ \frac{\partial^2 f(\mathbf{x})}{\partial x_2 \partial x_1} & \frac{\partial^2 f(\mathbf{x})}{\partial x_2^2} & \cdots & \frac{\partial^2 f(\mathbf{x})}{\partial x_2 \partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 f(\mathbf{x})}{\partial x_n \partial x_1} & \frac{\partial^2 f(\mathbf{x})}{\partial x_n \partial x_2} & \cdots & \frac{\partial^2 f(\mathbf{x})}{\partial x_n^2} \end{bmatrix}$$

- We can think of the Hessian as the gradient of the gradient, which can be written as

$$\mathbf{H} = \nabla_{\mathbf{x}}(\nabla^T_{\mathbf{x}} f(\mathbf{x}))$$

Revision: MLE for binary logistic regression

- The gradient and Hessian of $J(\mathbf{w})$ are given by:

$$\begin{aligned} \mathbf{g}(\mathbf{w}) &= \frac{d}{d\mathbf{w}} J(\mathbf{w}) = \sum_i (\pi_i - y_i)\mathbf{x}_i = \mathbf{X}^T(\boldsymbol{\pi} - \mathbf{y}) \\ \mathbf{H} &= \frac{d}{d\mathbf{w}} \mathbf{g}(\mathbf{w})^T = \sum_i (\nabla_{\mathbf{w}} \pi_i)\mathbf{x}_i^T = \sum_i \pi_i(1-\pi_i)\mathbf{x}_i\mathbf{x}_i^T = \mathbf{X}^T \mathrm{diag}(\pi_i(1-\pi_i))\mathbf{X} \end{aligned}$$

- One can show that **H** is positive definite; hence the NLL is **convex** and has a unique global minimum.

- To find this minimum, we will however have to learn a few things about optimization.

PMTK – logistic regression

-
  ```
  winit = randn(D,1);
  options.Display = 'none';
  [wMLE] = minFunc(@(w)LogisticLossSimple(w,X,y), winit, options);
  ```

-
  ```
  function [nll,g,H] = LogisticLossSimple(w,X,y)
  % Negative log likelihood for binary logistic regression
  % w: d*1
  % X: n*d
  % y: n*1, should be -1 or 1

  y01 = (y+1)/2;
  mu = sigmoid(X*w);
  nll = -sum(y01 .* log(mu) + (1-y01) .* log(1-mu));

  if nargout > 1
    g = X'*(mu-y01);
  end

  if nargout > 2
    H = X'*diag(mu.*(1-mu))*X;
  end
  ```

Unconstrained optimization

- We focus on optimization algorithms which can solve ML parameter estimation problems of the following form:

$$\boldsymbol{\theta}^* = \arg\max_{\boldsymbol{\theta}} \log p(\mathcal{D}|\boldsymbol{\theta})$$

- In the optimization community, it is more common to minimize functions than maximize them. We will therefore define our **objective function** as follows:

$$f(\boldsymbol{\theta}) \quad := \quad -\log p(\mathcal{D}|\boldsymbol{\theta})$$

Steepest descent

- One of the simplest optimization algorithms is called **gradient descent** or **steepest descent**. This can be written as follows:

$$\boldsymbol{\theta}_{k+1} = \boldsymbol{\theta}_k - \eta_k \mathbf{g}_k = \boldsymbol{\theta}_k - \eta_k \nabla f(\boldsymbol{\theta})$$

  where k indexes steps of the algorithm, $\mathbf{g}_k = \mathbf{g}(\boldsymbol{\theta}_k)$ is the gradient at step k, and $\eta_k > 0$ is called the **learning rate** or **step size**.

- By **Taylor's theorem**, we have

$$f(\boldsymbol{\theta}_{k+1}) \approx f(\boldsymbol{\theta}_k) + \eta \nabla f(\boldsymbol{\theta}_k)(\boldsymbol{\theta}_{k+1} - \boldsymbol{\theta}_k) = f(\boldsymbol{\theta}_k) - \eta \|\nabla f(\boldsymbol{\theta}_k)\|^2$$

Steepest descent

- One of the simplest optimization algorithms is called **gradient descent** or **steepest descent**. This can be written as follows:

  $$\boldsymbol{\theta}_{k+1} = \boldsymbol{\theta}_k - \eta_k \mathbf{g}_k$$

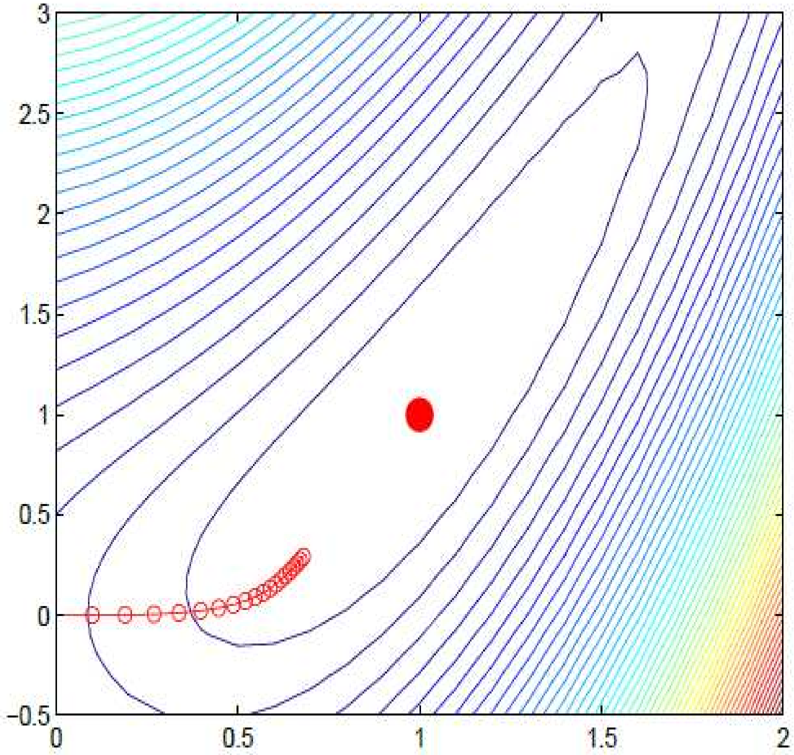
  where k indexes steps of the algorithm, $\mathbf{g}_k = \mathbf{g}(\boldsymbol{\theta}_k)$ is the gradient at step k, and $\eta_k > 0$ is called the **learning rate** or **step size**.

- If the function is convex, gradient descent will in theory always converge to the global minimum.

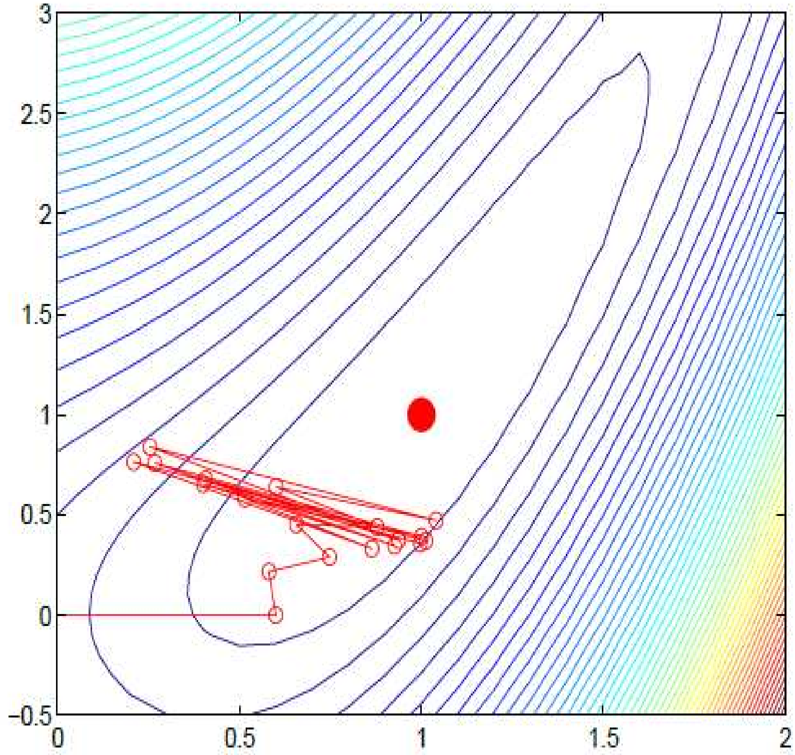
- If the function is non-convex, it will converge to a local minimum, which is a point where the gradient vanishes, $\mathbf{g} = \mathbf{0}$, and the Hessian $\mathbf{H}$ is positive definite, so all sides of the "bowl" point "up hill".

- If $\mathbf{H}$ is only positive semi-definite, we are at a turning or stationary point; such points are usually unstable, so we will generally disregard them.

Step size choice



$\eta = 0.1.$



$\eta = 0.6.$

Line search

- Let us develop a more stable method for picking the step size, so that the method is guaranteed to converge to a local optimum no matter where we start. (This property is called **global convergence**, which should not be confused with convergence to the global optimum!)

- Let us consider the update along a search direction $\mathbf{d}$:

$$\boldsymbol{\theta}_{k+1} = \boldsymbol{\theta}_k + \eta_k \mathbf{d}_k$$

- η_k is assumed to be positive and $\mathbf{d}$ is a descent direction. That is, $\mathbf{g}^T\mathbf{d} < 0$.

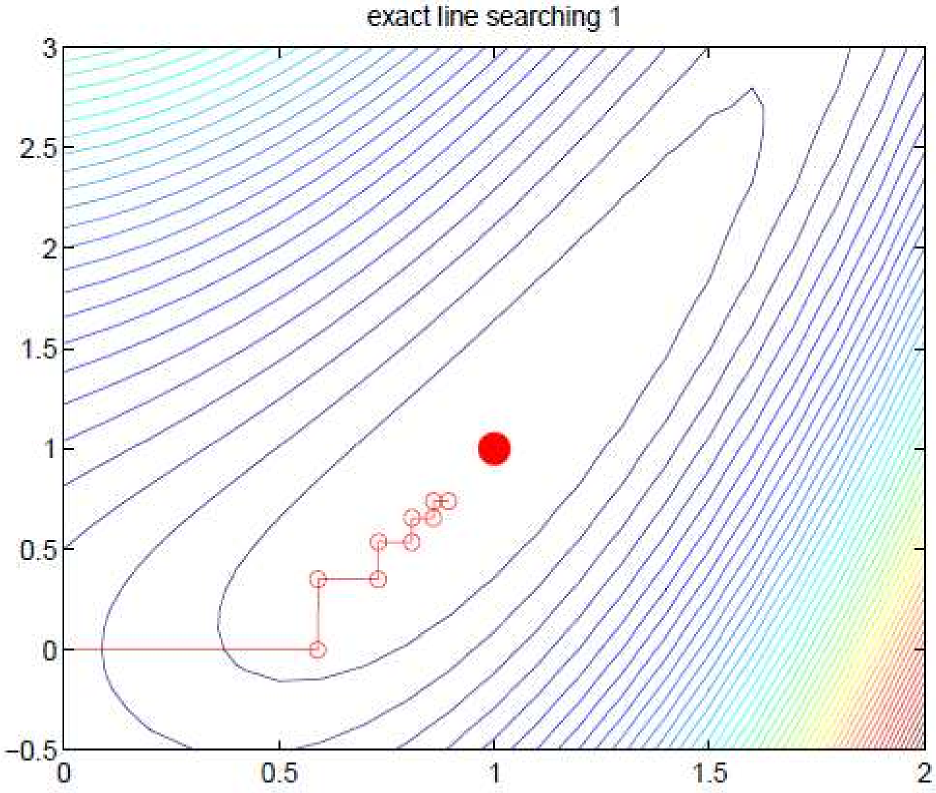
- By Taylor's expansion:

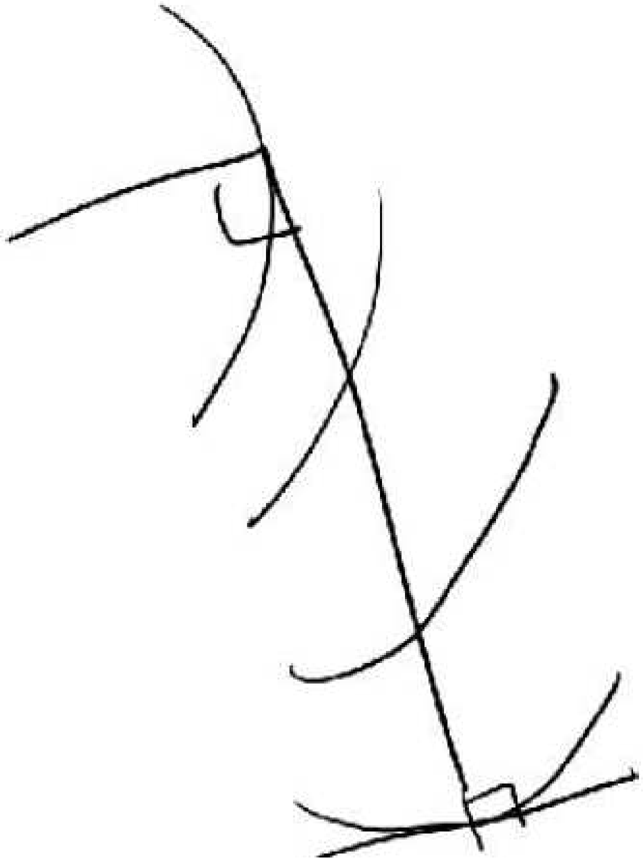
$$f(\boldsymbol{\theta} + \eta\mathbf{d}) \approx f(\boldsymbol{\theta}) + \eta\mathbf{g}^T\mathbf{d}$$

Line search

- $$\boldsymbol{\theta}_{k+1} = \boldsymbol{\theta}_k + \eta_k \mathbf{d}_k$$ $$f(\boldsymbol{\theta} + \eta \mathbf{d}) \approx f(\boldsymbol{\theta}) + \eta \mathbf{g}^T \mathbf{d}$$
- Hence, can pick η to minimize $$\phi(\eta) = f(\boldsymbol{\theta}_k + \eta \mathbf{d}_k)$$ subject to the constraint that the resulting direction is a descent direction (the so-called **Wolfe conditions**). This is called **line minimization** or **line search**.
- This optimization of η can be costly.
- Alternatively, choose an initial step size η. If the step size does lot lead to a reduction in the objective function, then reduce the step size and repeat. This is the intuituition behind the **Armijo rule**.

Line search





Momentum

- One simple heuristic to reduce the effect of zig-zagging is to add a **momentum** term, as follows:

$$\boldsymbol{\theta}_{k+1} \quad = \quad \boldsymbol{\theta}_k + (1 - \mu_k)\eta_k \mathbf{g}_k + \mu_k(\boldsymbol{\theta}_k - \boldsymbol{\theta}_{k-1})$$

  where $0 \leq \mu_k \leq 1$ is the amount of momentum.

- This technique is widely used to train neural networks and other nonlinear models, such as deep belief nets.

- Hinton recommends starting with $\mu_k = 0.5$ and then slowly increasing this to $\mu_k = 0.9$. See also results of Kevin Swersky.

Stochastic gradient descent

- Traditionally machine learning is performed **offline**, which means we have a **batch** of data, and we optimize a cost function of the form

  $$f(\boldsymbol{\theta}) = \frac{1}{N} \sum_{n=1}^{N} f(\boldsymbol{\theta}, \mathbf{x}_n)$$

  where we sum the cost over the N iid training cases.

- The gradient is therefore given by

  $$g(\boldsymbol{\theta}) = \nabla f(\boldsymbol{\theta}) = \frac{1}{N} \sum_{n=1}^{N} \nabla f(\boldsymbol{\theta}, x_n)$$

- In some cases, we can solve $g(\boldsymbol{\theta}) = \mathbf{0}$ in closed form, but in general, we will have to use gradient-based optimizers.

- If we have **streaming data**, we want to perform **online learning**, so we can update our estimates as each new data point arrives rather than waiting until "the end" (which may never occur).

Stochastic gradient descent

- We can use an approach known as **stochastic gradient descent** or **SGD** to solve such problems. The idea is to rewrite the objective and its gradient as an expectation wrt the empirical distribution:

  $$f(\boldsymbol{\theta}) = \mathbb{E}_{\mathbf{x}\sim p_{\text{emp}}}\left[f(\boldsymbol{\theta},\mathbf{x})\right], \;\; g(\boldsymbol{\theta}) = \mathbb{E}_{\mathbf{x}\sim p_{\text{emp}}}\left[\nabla f(\boldsymbol{\theta},\mathbf{x})\right]$$

  We now approximate the gradient with a single sample, corresponding to the most recent observation:

  $$\boldsymbol{\theta}_k = \boldsymbol{\theta}_{k-1} - \eta_k g(\boldsymbol{\theta}_{k-1}, \mathbf{x}_k)$$

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.

- We cannot use line-search to set η_k, so what should we do instead?
- To guarantee convergence, the learning rate must satisfy the following **Robbins-Monro** conditions:

$$\sum_{k=1}^{\infty} \eta_k = \infty \text{ and } \sum_{k=1}^{\infty} \eta_k^2 < \infty$$

  The set of values of η_k over time is called the learning rate **schedule**.
- Various formulas are used, such as $\eta_k = 1/k$, $\eta_k = \frac{\tau}{\tau+k}\eta_0$, or $\eta_k = \eta_0 k^{-\tau}$ for $\alpha \in (\frac{1}{2}, 1)$, where τ and η_0 are tuning parameters. The need to adjust these tuning parameters carefully is one of the main drawback of stochastic optimization.
- If the objective is convex and we know the number of iterations, then we also know, theoretically, the **fixed** value that η must be equal to (Nemirovsky, Juditsky, Lan and Shapiro, 2009).
- It is also possible to use **second order methods** that incorporate Hessian information (Richard H. Byrd, Gillian M. Chin, Will Neveitt and Jorge Nocedal, 2010) and Yurii Nesterov.

Averaging

- To reduce the variance of the estimate, we can average the estimates using

$$\overline{\boldsymbol{\theta}}_k = \sum_{t=1}^{k} \boldsymbol{\theta}_t$$

- This is called **Polyak-Ruppert averaging**. This can be implemented recursively as follows:

$$\overline{\boldsymbol{\theta}}_k = \overline{\boldsymbol{\theta}}_{k-1} - \frac{1}{k}(\overline{\boldsymbol{\theta}}_{k-1} - \boldsymbol{\theta}_k)$$

- The use of this scheme in principle allows one to use a fixed learning rate η_k. The idea is that the $\boldsymbol{\theta}_k$ estimates quickly converge to near the optimum and then wander around it, while $\overline{\boldsymbol{\theta}}_k$ averages out these fluctuations. This suggests that we should not start the averaging process until after a “burn-in” phase.

The LMS algorithm

- As an example of SGD, let us consider how to compute the MLE for linear regression in an online fashion. The online gradient at iteration k is given by

  $$\mathbf{g}_k := \mathbf{g}(\mathbf{w}_k) \approx (\mathbf{w}^T \mathbf{x}_i - y_i)\mathbf{x}_i$$

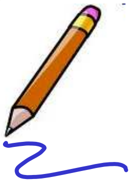
  where $i = k \bmod N$ is the training example to use at iteration k.

- The feature vector $\mathbf{x}_i$ is weighted by the difference between what we predicted, $\hat{y}_i = \mu_i = \mathbf{w}_k^T \mathbf{x}_i$, and the true response, y_i; hence the gradient acts like an *error signal.*

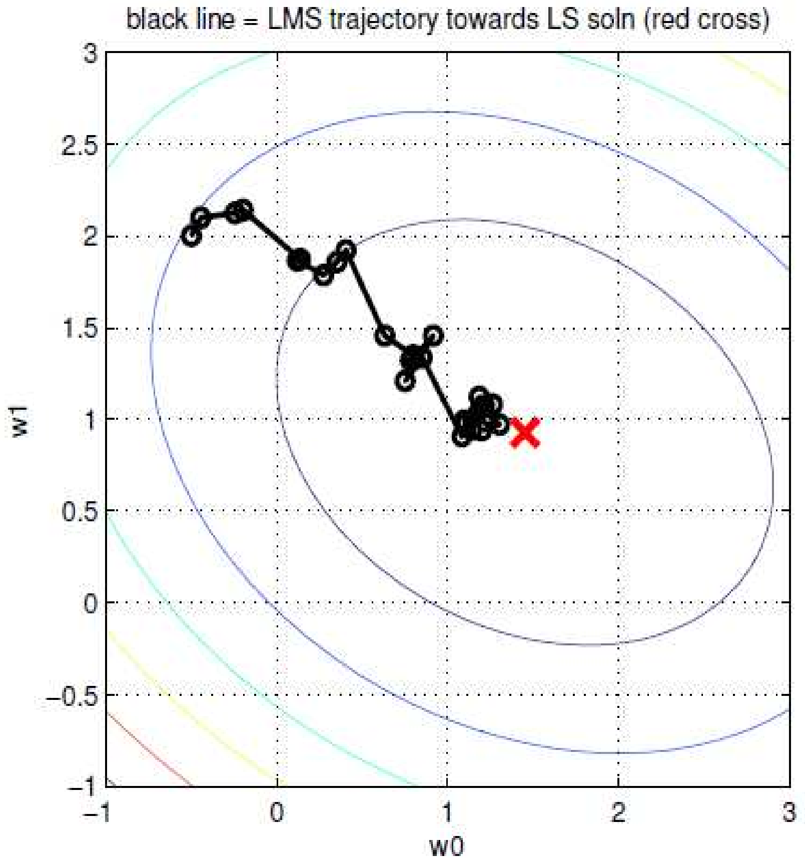
- After computing the gradient, we take a step along it as follows:

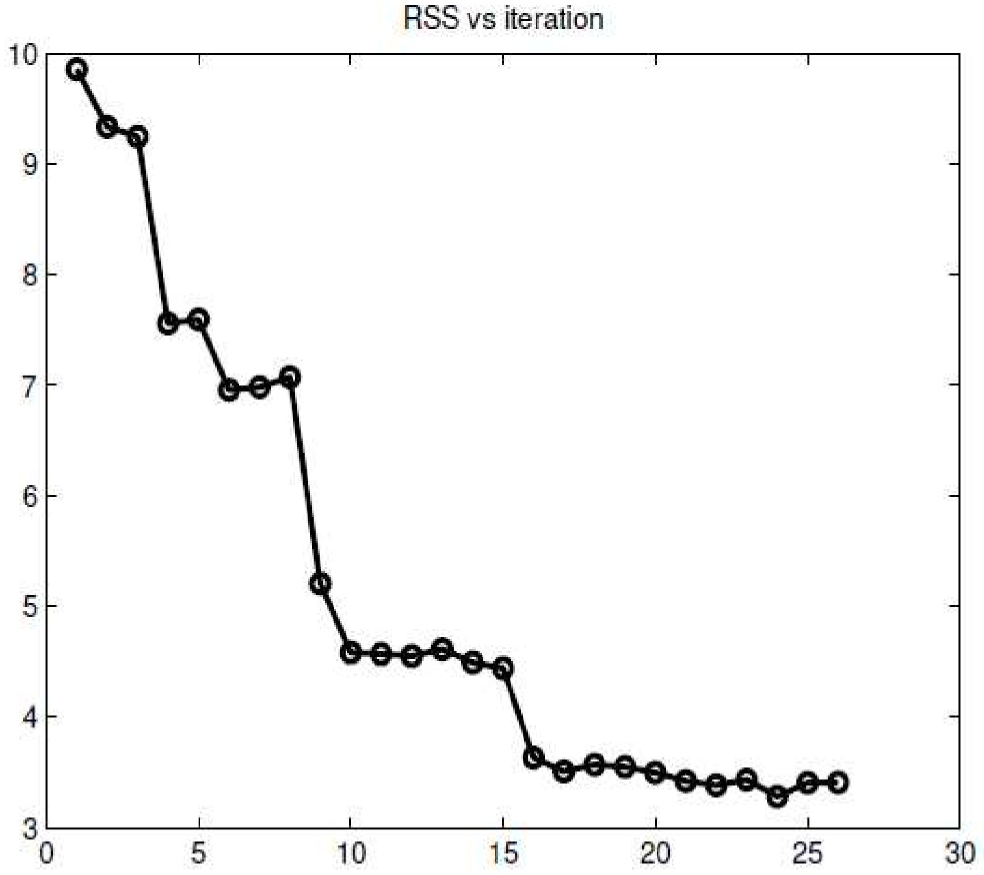
  $$\mathbf{w}_k = \mathbf{w}_{k-1} - \eta_k \mathbf{g}_k = \mathbf{w}_{k-1} - \eta_k (\mu_i - y_i)\mathbf{x}_i$$

  This algorithm is called the **least mean squares** or **LMS** algorithm, and is also known as the **delta rule**, or the **Widrow-Hoff rule**.



The LMS algorithm





The LMS algorithm

```
i = 1; iter = 1; eta = 0.1; sf = 0.999;
while ~done
  xi = X(i,:)’;
  yhat(i) = w’ * xi;
  wold = w;
  w = w + eta * (y(i)-yhat(i)) * xi;
  eta = eta * sf;
  iter = iter + 1;
  i = mod(i,n)+1;
  if norm(w-wold) < 1e-2 || iter > maxIter
    done = true;
  end
end
```

The perceptron algorithm

- Now let us consider how to fit a binary logistic regression model in an online manner. The weight update has the simple form

  $$\mathbf{w}_k = \mathbf{w}_{k-1} - \eta_k \mathbf{g}_i = \mathbf{w}_{k-1} - \eta_k(\mu_i - y_i)\mathbf{x}_i$$

  where $\mu_i = p(y_i = 1|\mathbf{x}_i, \mathbf{w}_k) = \mathbb{E}\left[y_i|\mathbf{x}_i, \mathbf{w}_k\right]$.

- We now consider an approximation to this algorithm. Specifically, let

  $$\hat{y}_i = \arg \max_{y \in \{0,1\}} p(y|\mathbf{x}_i, \mathbf{w})$$

  represent the most probable class label.

- We replace $\mu_i = p(y = 1|\mathbf{x}_i, \mathbf{w}) = \text{sigm}(\mathbf{w}^T \mathbf{x}_i)$ in the gradient expression with $\hat{y}_i$. Thus the approximate gradient becomes

  $$\mathbf{g}_i \approx (\hat{y}_i - y_i)\mathbf{x}_i$$

- It will make the algebra prettier if we assume $y \in \{-1, +1\}$ rather than $y \in \{0, 1\}$. In this case, our prediction becomes

$$\hat{y}_i = \text{sign}(\mathbf{w}^T \mathbf{x}_i)$$

  If $\hat{y}_i y_i = -1$, we made an error, but if $\hat{y}_i y_i = +1$, we guessed right.

- If we predicted correctly, then $\hat{y}_i = y_i$, so the (approximate) gradient is zero and we do not change the weight vector. But if $\mathbf{x}_i$ is misclassified, we update the weights as follows:

  - If $\hat{y}_i = 1$ but $y_i = -1$, the negative gradient is $-(\hat{y}_i - y_i)\mathbf{x}_i = -2\mathbf{x}_i$;
  - if $\hat{y}_i = -1$ but $y_i = 1$, the negative gradient is $-(\hat{y}_i - y_i)\mathbf{x}_i = 2\mathbf{x}_i$.

- We can absorb the factor of 2 into the learning rate η and just write the update, in the case of a misclassification, as

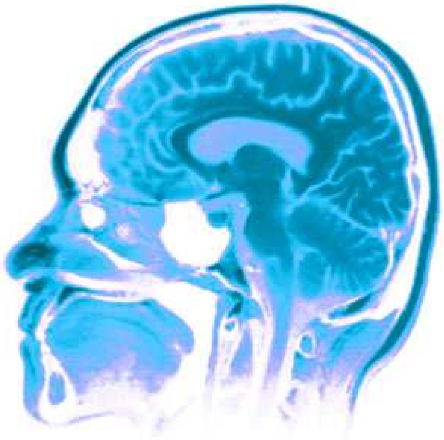
$$\mathbf{w}_k = \mathbf{w}_{k-1} + \eta_k y_i \mathbf{x}_i$$

  Note that we can set $\eta = 1$, since it is only the sign of the weights that matter, not the magnitude. The resulting algorithm is called the **perceptron algorithm**.

The perceptron algorithm

```
function [w,w0] = perceptronFit(X, y)
% X(i,:) is i’th case, y(i) = −1 or +1
labels = y; features = X’;
[n d] = size(X);
w = zeros(d,1);
w0 = 0;
max_iter = 100;
for iter=1:max_iter
  errors = 0;
  for i=1:n
    if ( labels(i) * ( w’ * features(:,i) + w0 ) <= 0 )
      w = w + labels(i) * features(:,i);
      w0 = w0 + labels(i);
      errors = errors + 1;
    end
  end
  if (errors==0), break; end
end
```

Next class



Second order methods and constrained optimization

Nando de Freitas

2011

KPM Book Sections: *11.3 and 30.48*