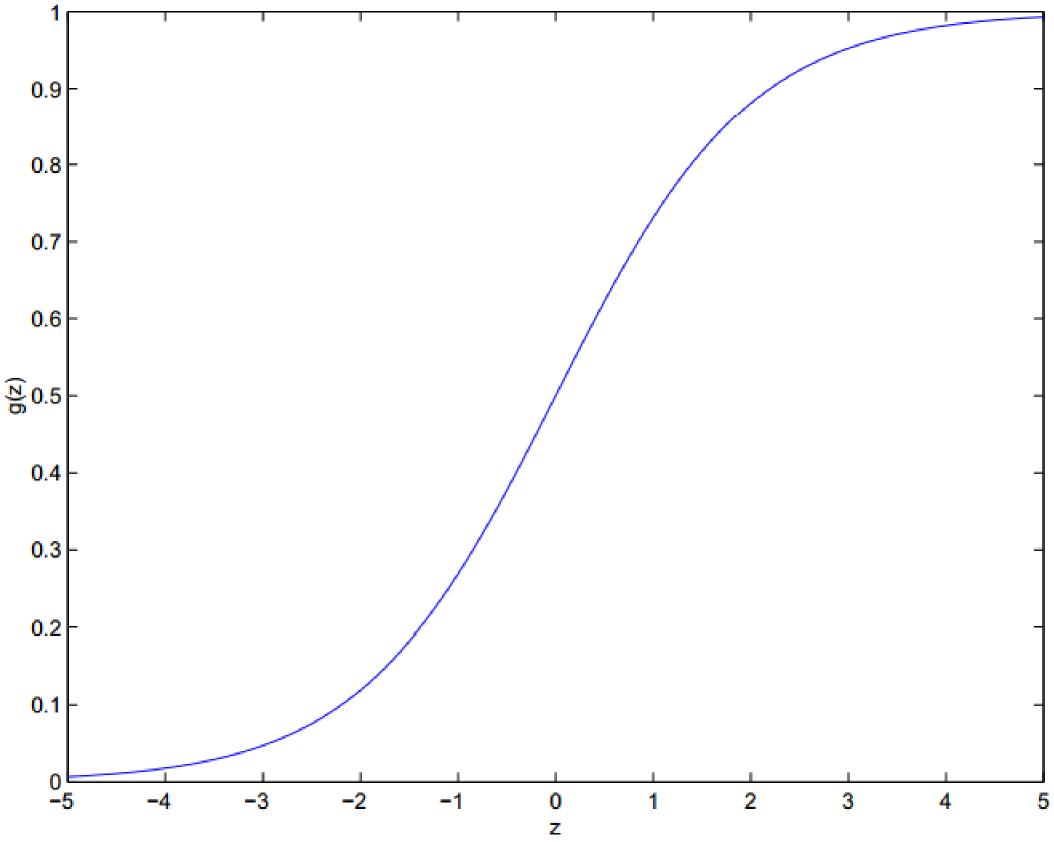
Some Tricks

For efficient implementation

Logistic Regression

- Another popular classification model
- Usual setting
  - Observe data $x_1, ..., x_n \in \mathbb{R}^d$
  - with labels $y_i \in \{-1, +1\}$
- Assume the label probability follows:

$$p(y = 1|x) = g(\langle w, x \rangle) = \frac{1}{1 + \exp(-\langle w, x \rangle)}$$



Analysing further

- Probability for other class

$$\begin{aligned} p(y=-1|x) &= 1 - p(y=1|x) \\ &= 1 - \frac{1}{1+\exp(-\langle w, x\rangle)} \\ &= \frac{\exp(-\langle w, x\rangle)}{1+\exp(-\langle w, x\rangle)} \\ &= \frac{1}{1+\exp(\langle w, x\rangle)} \end{aligned}$$

- Thus, overall we have:

$$p(y|x) = \frac{1}{1+\exp(-y\langle w, x\rangle)}$$

Training LR

- Maximum Likelihood Estimation $$\underset{w}{\text{maximize}} \quad \sum_i \log p(y_i|x_i, w)$$
- Equivalently $$\underset{w}{\text{minimize}} \quad \sum_i \log[1 + \exp(-y_i\langle w, x_i\rangle)]$$
- Add L_2 regularizer $$\underset{w}{\text{minimize}} \quad \sum_i \log[1 + \exp(-y_i\langle w, x_i\rangle)] + \lambda\|w\|^2$$
- Let’s solve this optimization problem in an efficient manner!

Logistic Regression vs SVM

- Recall SVM basically solves

$$\underset{w}{\text{minimize}} \quad \sum_i \max[0, 1 - y_i \langle w, x_i \rangle] + \lambda \|w\|^2$$

- LR basically solves

$$\underset{w}{\text{minimize}} \quad \sum_i \log[1 + \exp(-y_i \langle w, x_i \rangle)] + \lambda \|w\|^2$$

- That is just replace max with softmax!

Gradient Descent to solve LR

- The objective function is:

$$J(w) = \sum_{i=1}^{n} \log \left[1 + \exp\left(-y_i \sum_{j=1}^{d} w_j x_{ij} \right) \right] + \lambda \sum_{j=1}^{d} w_j^2$$

- How to evaluate this?

```
J=0;
for i=1:n
   inner_product = 0;
   for j=1:d
      inner_product = inner_product + w(j)*x(i,j);
   end
   J = J + log( 1 + exp( - y(i)*inner_product ) );
end
for j=1:d
   J = J + lambda*w(j)^2;
end
```

Computing Objective Function

- The objective function is:

$$J(w) = \sum_{i=1}^{n} \log \left[1 + \exp \left(-y_i \sum_{j=1}^{d} w_j x_{ij} \right) \right] + \lambda \sum_{j=1}^{d} w_j^2$$

- How to evaluate this?

Never!

```
J=0;
for i=1:n
   inner_product = 0;
   for j=1:d
      inner_product = inner_product + w(j)*X(i,j);
   end
   J = J + log( 1 + exp( - y(i)*inner_product ) );
end
for j=1:d
   J = J + lambda*w(j)^2;
end
```

Computing Objective Function

- The objective function is:

$$J(w) = \sum_{i=1}^{n} \log \left[1 + \exp \left(-y_i \sum_{j=1}^{d} w_j x_{ij} \right) \right] + \lambda \sum_{j=1}^{d} w_j^2$$

- How to evaluate this?

Not even this!

```
J = 0;
for i=1:n
   J = J + log( 1 + exp( - y(i)*X(i,:)*w ) );
end
J = J + sum(w.^2);
```

Computing Objective Function

- The objective function is:

$$J(w) = \sum_{i=1}^{n} \log \left[1 + \exp \left(-y_i \sum_{j=1}^{d} w_j x_{ij} \right) \right] + \lambda \sum_{j=1}^{d} w_j^2$$

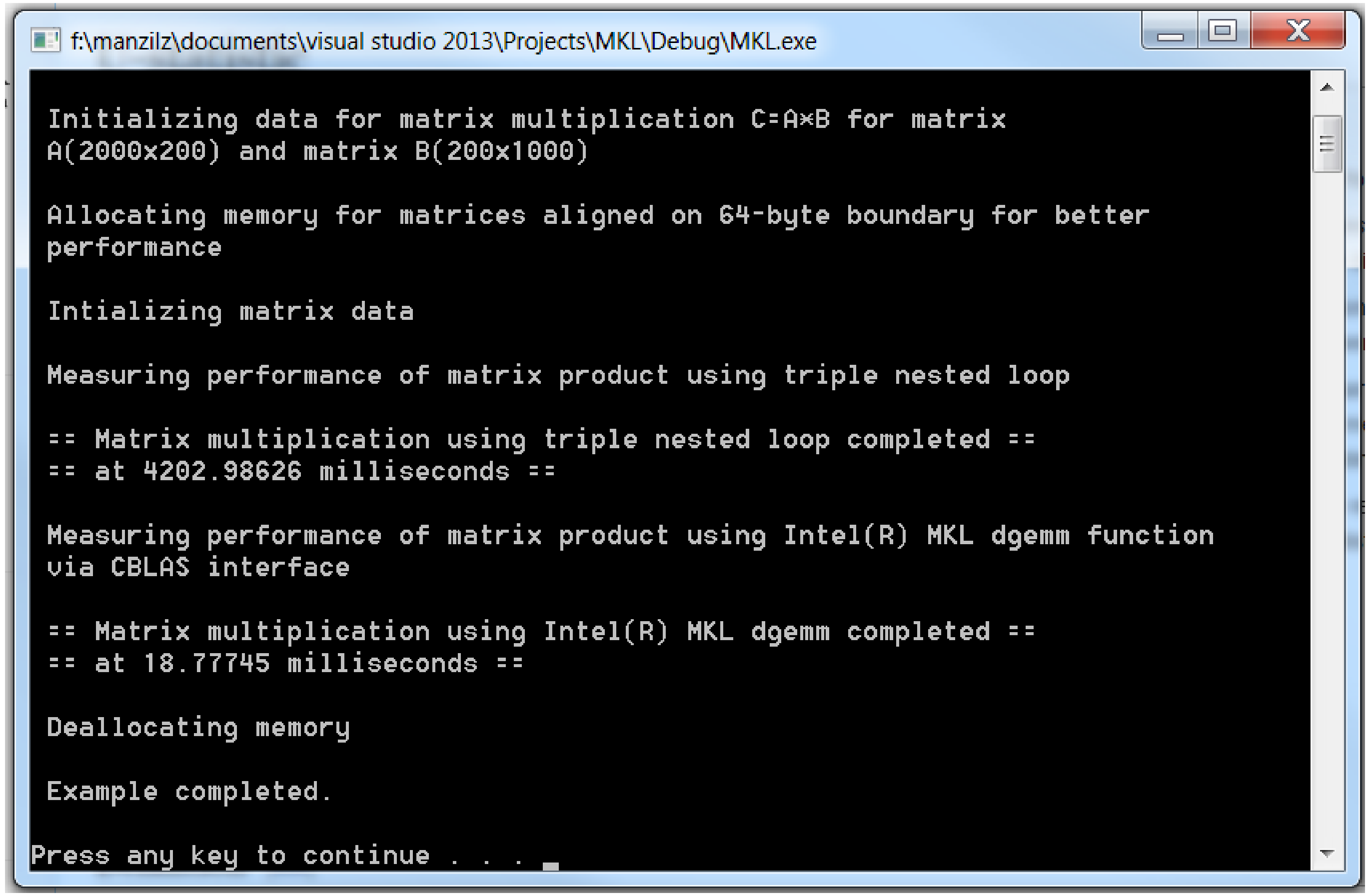
- How to evaluate this?

```
J = sum( log( 1 + exp( - (X*w).*y ) ) ) + lambda*sum(w.^2);
```

  - Short code!
  - Matrix-vector products and summing vectors are highly optimized

Matrix Multiplication

- Never write vector or matrix operations by yourself!
- Always use libraries
  - 100x faster!
- MKL or BLAS maybe intimidating to use directly
- Good News:
  - Matlab already does it for you
  - Eigen as wrapper
    - Almost matlab like API in C++



Exercise: Computing Gradient

- For the gradient descent approach, next thing needed is the gradient!

$$\frac{\partial J(w)}{\partial w_k} = \sum_{i=1}^{n} \frac{y_i x_{ik}}{1 + \exp\left(y_i \sum_{j=1}^{d} w_j x_{ij}\right)} + 2\lambda w_k$$

Exercise: Computing Gradient

- For the gradient descent approach, next thing needed is the gradient!

$$\frac{\partial J(w)}{\partial w_k} = \sum_{i=1}^{n} \frac{y_i x_{ik}}{1 + \exp\left(y_i \sum_{j=1}^{d} w_j x_{ij}\right)} + 2\lambda w_k$$

- Get the entire gradient vector at one go!
- One way using repmat

```
b = ( 1 + exp( (X*w).*y ) ) ) .* y
b = repmat(b,1,5);
g = sum(X./b)’ + 2*lambda*w;
```

Exercise: Computing Gradient

- For the gradient descent approach, next thing needed is the gradient!

$$\frac{\partial J(w)}{\partial w_k} = \sum_{i=1}^{n} \frac{y_i x_{ik}}{1 + \exp\left(y_i \sum_{j=1}^{d} w_j x_{ij}\right)} + 2\lambda w_k$$

- Get the entire gradient vector at one go!
- More memory efficient

```
b = ( 1 + exp( (X*w).*y ) ) ) .* y
g = sum(bsxfun(@rdivide, X,b));
g = g' + 2*lambda*w;
```

Computing Gram Matrices

$$K_{ij} = \exp(-\|x_i - x_j\|^2)$$

```
nsq=sum(X.^2,2);

K=bsxfun(@minus,nsq,(2*X)*X.');
K=bsxfun(@plus,nsq.',K);
K=exp(-K);
```

Algebraic Tricks

- Hopefully if you will solve HW5 bonus and get a multi-variate student t-distribution for the posterior predictive of Normal Inverse Wishart:

PDF of a general $t_\nu(\mathbf{x}; \boldsymbol{\mu}, \boldsymbol{\Sigma}) = \dfrac{\Gamma\left[(\nu+p)/2\right]}{\Gamma(\nu/2)\nu^{p/2}\pi^{p/2}\left|\boldsymbol{\Sigma}\right|^{1/2}\left[1+\frac{1}{\nu}(\mathbf{x}-\boldsymbol{\mu})^T\boldsymbol{\Sigma}^{-1}(\mathbf{x}-\boldsymbol{\mu})\right]^{(\nu+p)/2}}$

- So you need the determinant and inverse of Σ – expensive $O(d^3)$
- Moreover, posterior predictive has to be computed many times for different $\tilde{\mathrm{x}}$

Cholesky Decomposition

- The update in posterior predictive for Σ would be

$$\tilde{\Sigma} = \Sigma_n + \frac{\kappa_n + 1}{\kappa_n}(\tilde{x} - \mu_n)(\tilde{x} - \mu_n)^T$$

- So instead of computing this update:
  - Suppose we have cholesky decomposition of Σ_n
  - Then we calculate only the rank-one update to obtain $\tilde{\Sigma}$

Cholesky Updates

- Suppose A is a positive definite matrix with L as its cholesky decomposition.
- Now if we obtain A' from A by an update of the form

$$A' \quad = A + xx^T$$

- then the cholesky decomposition L' of A' can be obtained by an update operation on L. (Rank 1 update)
- Similarly if we have $A = A'$ - xx^T, then we can perform a Rank1 downdate to get L from L'

Cholesky Update

```
function [L] = cholupdate(L,x)
    p = length(x);
    x = x';
    for k=1:p
        r = sqrt(L(k,k)^2 + x(k)^2);
        c = r / L(k, k);
        s = x(k) / L(k, k);
        L(k, k) = r;
        L(k,k+1:p) = (L(k,k+1:p) + s*x(k+1:p)) / c;
        x(k+1:p) = c*x(k+1:p) - s*L(k, k+1:p);
    end
end
```

- This algorithm is $O(D^2)$!

Nice Properties

- $|A|$ can be computed from L by

$$log(|A|) = 2 * \sum_{i=1}^{D} log(L(i,i))$$

- Now lets try to compute $b^T A^{-1} b$

$$\begin{aligned} b^T A^{-1} b &= b^T (LL^T)^{-1} b \\ &= b^T (L^{-1})^T L^{-1} b \\ &= (L^{-1} b)^T (L^{-1} b) \end{aligned}$$

- Therefore compute $(L^{-1}b)$ and multiply its transpose with itself

Triangular Solver

- $(L^{-1}b)$ is the solution of

$$Lx = b$$

- Remember L is a lower triangular matrix, therefore the above equation can be solved very efficiently using forward substitution!

$$\begin{array}{ccccccc} l_{1,1}x_1 & & & & & = & b_1 \\ l_{2,1}x_1 & + & l_{2,2}x_2 & & & = & b_2 \\ \vdots & & \vdots & \ddots & & & \vdots \\ l_{m,1}x_1 & + & l_{m,2}x_2 & + \cdots + & l_{m,m}x_m & = & b_m \end{array}$$

Miscellaneous Tricks

- Finding the min/max of a matrix of N-d array

```
[MinValue, MinIndex] = min( A(:) );        %find minimum element in A
MinSub = ind2sub(size(A), MinIndex);    %convert MinIndex to subscripts
```

- Try to avoid inverse of a matrix!
  - Typically you only need $\mathrm{x} = A\backslash\mathrm{b}$
  - This invokes appropriate linear solver
  - Much more efficient and numerically stable