Solver, Regressor and Optimizer

Solver	*principle*	*local approximation*	*derivative*	
1.1 Bisection			*no*	
1.2 Newton Raphson method	*Newton method for solver*	*approx fct by tangent*	*1st*	
1.3 Halley method			*1st/2nd*	
1.4 Secant method		*approx fct by secant*	*no*	
1.5 Inverse quadratic interpolation		*approx fct by inv quadratic*	*no*	
1.6 Dekker method and Brent method	*combination of two methods*			
Regressor				
2.1 Random sample concensus			*no*	`(Maths.doc)`
2.2 Gauss Newton method	*Newton method for regressor*	*approx fct by linear / obj by quad*	*1st*	`(Maths.doc)`
2.3 Levenberg Marquardt method	*Newton method for regressor*	*approx fct by linear / obj by quad*	*1st*	`(Maths.doc)`
2.4 Conjugate gradient		*objective is quad by itself*	*1st*	
Optimizer				
3.1 Hill climbing	*hill climbing*		*no*	`(max)`
3.2 Simulated annealing	*hill climbing*		*no*	`(max)`
3.3 Genetic algorithm	*hill climbing*		*no*	`(max)`
3.4 Gradient descent		*approx objective by linear*	*1st*	`(min)`
3.5 Stochastic gradient descent	*mini batch / online training*	*approx objective by linear*	*1st*	`(min)`
3.6 Sequential quadratic programming	*Newton method for optimizer*	*approx objective by quad*	*1st/ 2nd*	`(min)`
3.7 Sequential minimal optimization	*mini batch / online training*	*objective is quad by itself*	*no*	`(max)`

Lagrangian and Karush Kuhn Tucker condition

4.1 Maximization and minimization with equality constraints	*Lagrangian without KKT*
4.2 Maximization and minimization with inequality constraints	*Lagrangian generalized with KKT and active set*
4.3 Duality problem *(which explains why maximize wrt α in SVM)*	

Comparison among 3 solvers

	Newton method	*secant method*	*inverse quadratic*
update equation	$x_{n-1}-f(x_{n-1})/f'(x_{n-1})$	$x_{n-1}-f(x_{n-1})\dfrac{x_{n-1}-x_{n-2}}{f(x_{n-1})-f(x_{n-2})}$	$Ax_{n-1}+Bx_{n-2}+Cx_{n-3}$
dependency	$f(x_{n-1})$ *and* $f'(x_{n-1})$	$f(x_{n-1})$ *and* $f(x_{n-2})$	$f(x_{n-1})$ $f(x_{n-2})$ *and* $f(x_{n-3})$
differentiation?	*yes*	*no*	*no*
approx function	*tangent*	*secant*	*quadratic*$^{-1}$

Solver, regressor and optimizer

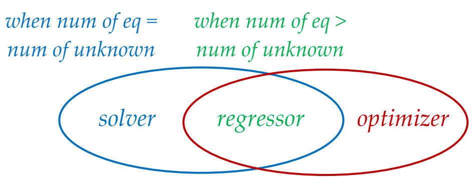
original problem	*convert to solver problem*	*convert to optimizer problem*
solver : solve $AX=B$ *where* $\text{size}(A)=M\times M$	$AX=B$ `(1)`	$\min\limits_X \frac{1}{2}X^TAX-X^TB$ `(1')`
regressor : min error fit $AX=B$ *where* $\text{size}(A)=N\times M$	$A^TAX=A^TB$ `(2)`	$\min\limits_X (AX-B)^T(AX-B)$ `(2')`

proof of equivalence between `(1)` *and* `(1')`

$$\min_X \frac{1}{2}X^TAX-X^TB \quad\rightarrow\quad \frac{\partial}{\partial X}\frac{1}{2}X^TAX-X^TB=0 \quad\rightarrow\quad AX-B=0$$

proof of equivalence between `(2)` *and* `(2')`

$$\min_X (AX-B)^T(AX-B) \quad\rightarrow\quad \frac{\partial}{\partial X}(AX-B)^T(AX-B)=0 \quad\rightarrow\quad A^TAX-A^TB=0$$


when num of eq = num of unknown
when num of eq > num of unknown
solver
regressor
optimizer

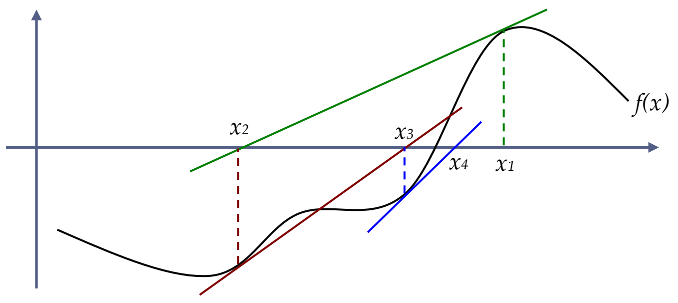
1.1 Bisection method

Please refer to `algorithm.doc` for various versions.

1.2 Newton method (or Newton Raphson method)

1.2a From tangent perspective

Newton method iteratively solves the linear equation that is tangent to the function at the previous estimation.



Newton method solves $f(x) = 0$ with the following updating rule (n is the iteration number) :

$$\begin{aligned} f'(x_{n-1}) &= \frac{y - f(x_{n-1})}{x - x_{n-1}} && \textit{by putting } y = 0\textit{, we then have ...} \\ x_n &= x_{n-1} - f(x_{n-1}) / f'(x_{n-1}) \end{aligned}$$

Disadvantage of Newton method is that analytic derivatives should be available, otherwise it has to be approximated by numerical method, which involves invocation of f twice in each iteration, resulting in slower convergence. Other methods, like secant method and Brent method are developed to limit the number of invocations of function f to once per iteration.

1.2b From Taylor series perspective

Newton method can also be explaint using Taylor series with *1st* term.

$$\begin{aligned} f(x_n) &= f(x_{n-1}) + f'(x_{n-1})(x_n - x_{n-1}) + ... && \textit{by putting } f(x_n) = 0\textit{, we then have ...} \\ x_n &= x_{n-1} - f(x_{n-1}) / f'(x_{n-1}) \end{aligned}$$

1.2c Newton method for multivariate

Newton method can be used in solving multivariate equation system $F(X)$ where $F(X) : \Re^M \to \Re^M$ and $J_{nm} = \frac{\partial F_n}{\partial x_m}$.

$$\begin{aligned} F(X) &= F(X_{t-1}) + J(X - X_{t-1}) && \textit{by putting } F(X) = 0\textit{, we then have ...} \\ X_t &= X_{t-1} - J^{-1} f(X_{t-1}) \end{aligned}$$

1.2d Newton method for multivariate optimization

Newton method can be used in optimization multivariate objective $f(X) : \Re^M \to \Re^1$ and $\nabla f = \left[\frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial x_2}, ... \right]$.

$$\begin{aligned} & \arg\max_X f(X) \textit{ where } f(X) : \Re^M \to \Re^1 \\ \Rightarrow \quad \nabla f(X)^T &= 0 \\ \nabla f(X)^T &= \nabla f(X_{t-1})^T + H(X - X_{t-1}) && \textit{by putting } F(X) = 0\textit{, we then have ...} \\ X_t &= X_{t-1} - H^{-1} \nabla f(X_{t-1})^T \end{aligned}$$

1.3 Halley method

Halley method is generalization of Newton method to second order derivatives. Any root of *f* which is not a root of *f′* is a root of :

$$g(x) = \frac{f(x)}{\sqrt{abs(f'(x))}}$$

$$\Rightarrow \quad g'(x) = \frac{f'(x)}{\sqrt{abs(f'(x))}} - \frac{1}{2}\frac{f(x)f''(x)}{f'(x)\sqrt{abs(f'(x))}} \qquad \textit{since } [abs(f)]^{3/2} = \sqrt{abs(f)[abs(f)]^2} = \sqrt{abs(f)f^2}$$

$$= \frac{2f'(x)f'(x) - f(x)f''(x)}{2f'(x)\sqrt{abs(f'(x))}}$$

Let's apply Newton method on *g*, the final updating equation depends on one previous estimation x_{n-1} only.

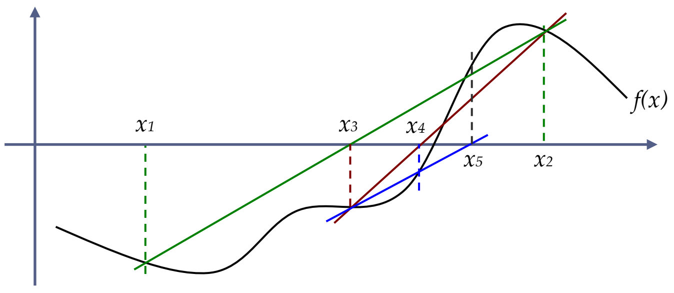
$$\begin{aligned} x_n &= x_{n-1} - g(x_{n-1})/g'(x_{n-1}) \\ &= x_{n-1} - \frac{f(x_{n-1})}{\sqrt{abs(f'(x_{n-1}))}} \Bigg/ \frac{2f'(x_{n-1})f'(x_{n-1}) - f(x_{n-1})f''(x_{n-1})}{2f'(x_{n-1})\sqrt{abs(f'(x_{n-1}))}} \\ &= x_{n-1} - \frac{2f(x_{n-1})f'(x_{n-1})}{2f'(x_{n-1})f'(x_{n-1}) - f(x_{n-1})f''(x_{n-1})} \\ &= x_{n-1} - \frac{2ff'}{2f'f' - ff''}\bigg|_{x=x_{n-1}} \end{aligned}$$

1.4 Secant method

Secant method does not involve derivative, hence it offers a better speed. Yet it does not guarantee a convergence. In contrast to the Newton method, which iteratively solves the tangent to the latest estimation, secant method iteratively solves the secant joining the two latest estimations. Secant equation based on two latest estimations is :

$$y = \frac{f(x_{n-1}) - f(x_{n-2})}{x_{n-1} - x_{n-2}}(x - x_{n-1}) + f(x_{n-1}) \qquad \textit{by putting } y = 0\textit{, we then have ...}$$

$$x_n = x_{n-1} - f(x_{n-1})\frac{x_{n-1} - x_{n-2}}{f(x_{n-1}) - f(x_{n-2})}$$



1.5 Lagrange polynomial and inverse quadratic interpolation

Secant method does not involve derivative, hence it offers a better speed. Apart from iterative solving a tangent or a secant, we can also iterative solving an inverse quadratic (inverse quadratic means that the curve x coordinate is quadratic with respect to y). Prior to the discussion about inverse quadratic, lets introduce the Lagrange polynomial. Given a set of N data points (x_n, y_n) $n \in [1,N]$, the N-1 degree polynomial $y = f(x)$ that passes through all the points can be written as sum of products (each product has degree N-1) :

$$f(x) \quad = \quad \sum_{n=1}^{N}(y_n \prod_{m \neq n} \frac{x-x_m}{x_n-x_m})$$

Lets verify that it does pass through (x_n, y_n) $n \in [1,N]$.

$$\begin{array}{lcl} f(x_n) & = & \left[\begin{array}{l} y_1 \frac{x_n-x_2}{x_1-x_2}\frac{x_n-x_3}{x_1-x_3}\ldots\underbrace{\frac{x_n-x_n}{x_1-x_n}}_{0}\ldots\frac{x_n-x_N}{x_1-x_N} + y_2 \frac{x_n-x_1}{x_2-x_1}\frac{x_n-x_3}{x_2-x_3}\ldots\underbrace{\frac{x_n-x_n}{x_2-x_n}}_{0}\ldots\frac{x_n-x_N}{x_2-x_N} + \ldots \\ + y_n \underbrace{\frac{x_n-x_1}{x_n-x_1}\frac{x_n-x_2}{x_n-x_2}\ldots\frac{x_n-x_{n-1}}{x_n-x_{n-1}}}_{1} + \ldots + y_N \frac{x_n-x_1}{x_N-x_1}\frac{x-x_3}{x_N-x_3}\ldots\underbrace{\frac{x_n-x_n}{x_N-x_n}}_{0}\ldots\frac{x_n-x_{N-1}}{x_N-x_{N-1}} \end{array}\right. \\ & = & y_n \end{array}$$

Therefore it generates a linear function for 2 data points, a quadratic function for 3 data points etc. Inverse quadratic refers to fitting $x = f^{-1}(y)$ with Lagrange polynomial of the 2^{nd} order. Instead of interpolation, inverse quadratic is used as a method for iterative root finding, which depends on 3 latest previous estimations x_{n-1}, x_{n-2} and x_{n-3}, it fits latest estimations with inverse quadratic & update x_n.

$$x \quad = \quad \left[\begin{array}{l} x_{n-1}\left(\frac{y-f(x_{n-2})}{f(x_{n-1})-f(x_{n-2})}\frac{y-f(x_{n-3})}{f(x_{n-1})-f(x_{n-3})}\right)+ \\ x_{n-2}\left(\frac{y-f(x_{n-1})}{f(x_{n-2})-f(x_{n-1})}\frac{y-f(x_{n-3})}{f(x_{n-2})-f(x_{n-3})}\right)+ \\ x_{n-3}\left(\frac{y-f(x_{n-1})}{f(x_{n-3})-f(x_{n-1})}\frac{y-f(x_{n-2})}{f(x_{n-3})-f(x_{n-2})}\right) \end{array}\right. \qquad \textit{by putting } y = 0\textit{, we then have ...}$$

$$x_n \quad = \quad \left[\begin{array}{l} x_{n-1}\left(\frac{f(x_{n-2})}{f(x_{n-1})-f(x_{n-2})}\frac{f(x_{n-3})}{f(x_{n-1})-f(x_{n-3})}\right)+ \\ x_{n-2}\left(\frac{f(x_{n-1})}{f(x_{n-2})-f(x_{n-1})}\frac{f(x_{n-3})}{f(x_{n-2})-f(x_{n-3})}\right)+ \\ x_{n-3}\left(\frac{f(x_{n-1})}{f(x_{n-3})-f(x_{n-1})}\frac{f(x_{n-2})}{f(x_{n-3})-f(x_{n-2})}\right) \end{array}\right.$$

1.6 Dekker method *(1969)* and Brent method *(1973)*

- Dekker method is a combination of bisection and secant method. It is reliable as bisection, while as efficient as secant.
- Brent method is a combination of bisection, secant method and inverse quadratic interpolation.
- Both algorithms are too cumbersome, the details are thus omitted.

2.4 Conjugate gradient [this section is incomplete, please verify the maths]

Conjugate gradient is an effective iterative method for solving system of linear equation $AX=B$, which is not a regression, hence size of A is M by M. For regression problem $AX=B$, when size of A is N by M, we should firstly convert it to linear system by $A^TA=A^TB$.

2.4a Definition of conjugate

Two vectors are said to be conjugate with respect to ***symmetric*** and ***positive definite*** matrix A if :

$$\begin{aligned} u^T Av &= 0 && \textit{if u and v are col matrix (all conjugate vectors are col matrix, in order to make grad descent of X easier)} \\ \langle u,v\rangle_A &= 0 && \textit{for convenience, we denote the conjugate dot product as } \langle u,v\rangle_A \end{aligned}$$

Intuition

Conjugate pair is liked the kernel trick in *SVM*, it is the dot product in transformed space, which is not explicitly given. As A is :

$$\begin{aligned} A &= P^T P && \textit{P stands for projection matrix or linear transformation} \\ u^T Av &= 0 \\ u^T P^T Pv &= 0 \\ (Pu)^T Pv &= 0 \end{aligned}$$

It is equivalent to firstly transform the vectors by matrix P and perform dot product in the new space. Conjugate pairs are therefore orthogonal in the new space, but not in the original space.

2.4b Decomposition into conjugate vectors

Given all conjugate vectors with respect to matrix A, i.e. $Q = \{q_1, q_2, q_3, ..., q_M\}$, where all q_m are Mx1 column matrix.

$$q_m^T A q_{m'} = 0 \qquad \forall m,m' \in [1,M] \text{ and } m \neq m'$$

If we are then given a vector X in $\Re^M$, how can we decompose X into a combination of conjugate vectors? Suppose X is :

$$\begin{aligned} X &= QW \\ &= [q_1 \quad q_2 \quad \cdots \quad q_M] \begin{bmatrix} w_1 \\ w_2 \\ \cdots \\ w_M \end{bmatrix} \\ &= \sum_{m\in[1,M]} w_m q_m \end{aligned}$$

The weight vector can be solved by left-multiplying A on both sides, followed by left-multiplying each q_m on both sides :

$$\begin{aligned} AX &= AQW \\ q_m^T AX &= q_m^T AQW \\ &= w_m q_m^T A q_m && \textit{since } q_m^T A q_{m'} = 0 \textit{ for all } m' \textit{ except when } m = m' \\ w_m &= \frac{q_m^T AX}{q_m^T A q_m} \\ &= \frac{q_m^T B}{q_m^T A q_m} && \textit{where } B = AX \textit{ is a } M\text{x}1 \textit{ column matrix} \end{aligned}$$

2.4c Finding conjugate vectors

This method works like Gram Schmidt orthogonalization in LQ decomposition :

- maintain a subset of conjugate orthogonal vectors
- in each iteration, find one new conjugate vector and add it into the subset
- in each iteration, move in gradient descent direction minus projections on existing conjugate set
- find the residual that cannot be explaint by existing conjugate vectors

Implementation of the algorithm

Please run the algorithm with row scanning ...

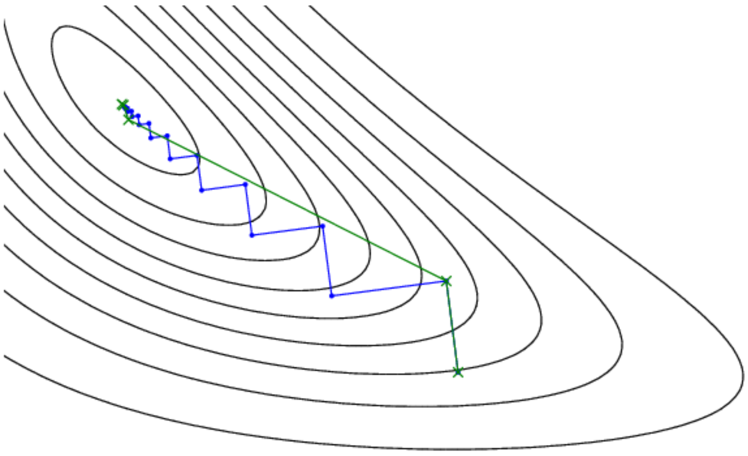
update X	*gradient direction*	*conjugate direction*	*step size*
$X_0 = initial$	$\nabla_X L(X_0) = B - AX_0 = r_0$	$q_0 = r_0$	$w_0 = \frac{q_0^T r_0}{<q_0, q_0>_A}$
$X_1 = X_0 + w_0 q_0$	$\nabla_X L(X_1) = B - AX_1 = r_1$	$q_1 = r_1 - \frac{<q_0, r_0>_A}{<q_0, q_0>_A} q_0$	$w_1 = \frac{q_1^T r_1}{<q_1, q_1>_A}$
$X_2 = X_1 + w_1 q_1$	$\nabla_X L(X_2) = B - AX_2 = r_2$	$q_2 = r_2 - \frac{<q_0, r_2>_A}{<p_0, p_0>_A} q_0 - \frac{<q_1, r_2>_A}{<p_1, p_1>_A} q_1$	$w_2 = \frac{q_2^T r_2}{<q_2, q_2>_A}$
$X_3 = X_2 + w_2 q_2$	$\nabla_X L(X_3) = B - AX_3 = r_3$	$q_3 = r_3 - \frac{<q_0, r_3>_A}{<q_0, q_0>_A} q_0 - \frac{<q_1, r_3>_A}{<q_1, q_1>_A} q_1 - \frac{<q_2, r_3>_A}{<q_2, q_2>_A} q_2$	$w_3 = \frac{q_3^T r_3}{<q_3, q_3>_A}$
$X_4 = X_3 + w_3 q_3$	...	...	

Explanation of conjugate direction column : we need to remove projection of gradient on existing conjugate vectors ...

$$q_m \quad = \quad \nabla_X L(X_m) - \frac{<q_0, \nabla_X L(X_m)>}{<q_0, q_0>_A} q_0 - \frac{<q_1, \nabla_X L(X_m)>_A}{<q_1, q_1>_A} q_1 - ... - \frac{<q_m, \nabla_X L(X_m)>}{<q_{m-1}, q_{m-1}>_A} q_{m-1}$$

Explanation of step size : it seems to be slightly different from part 2.4b. Why that?

Paths of optimization



blue = gradient vs green = conjugate gradient

3.1 Hill climbing

Hill climbing is an optimization in a state graph, in which each node denotes one state, each edge denotes a possible state transition. Our objective is to find a state so that the objective function $f : \text{state} \to \Re$ is maximised. Hill climbing, together with its variants, like stochastic hill climbing, simulated annealing and genetic algorithm, does not depend on its optimization path. Now, consider using hill climbing to solve travelling salesman problem, one city-sequence is a valid state, a swap between two cities generate a new city sequence, that is a new state having an edge connecting to the original state. Hill climbing is a ***greedy local search*** that may stuck in local optimum.

```
state hill_climbing(const graph& graph, const state& init, std::function<double(const state&)>& f)
{
    state x = init;
    value y = f(x);
    bool converge = false;
    while(!converge)
    {
        converge = true;
        std::vector<state> xs = graph.neighbour_states(x); // neighbourhood is defined as states having edge connection
        if (auto [x0,y0] = find_max(xs, f); y0 > y)
        {
            x = x0;
            y = y0;
            converge = false;
        }
    }
    return x;
}
```

Hill climbing can be applied to continuous domain with objective function $f : \Re^M \to \Re^1$, with neighbourhood defined as small range around current state.

```
double hill_climbing(double init, double window_size, std::function<double(double)>& f)
{
    state x = init;
    value y = f(x);
    bool converge = false;
    while(!converge)
    {
        converge = true;
        if (auto [x0,y0] = find_max(x-window_size, x+window_size, f); y0 > y)
        {
            x = x0;
            y = y0;
            converge = false;
        }
    }
    return x;
}
```

When an optimization algorithm is prefixed by the term "stochastic", it means online-training, or equivalently, mini-batch training. Given a dataset with N data points, instead of considering the whole dataset during each iteration, we randomly select a mini-batch (one data point to be extreme) during each iteration. For stochastic hill climbing instead of picking maximum among all neighbours, we randomly pick one out of the better (not the best one) neighbouring states.

```
state hill_climbing(const graph& graph, const state& init, std::function<double(const state&)>& f)
{
    state x = init;
    value y = f(x);
    bool converge = false;
    while(!converge)
    {
        converge = true;
        std::vector<state>  xs = graph.neighbour_states(x);   // can we simplify this part ...
        std::vector<double> ys(xs.size(), 0);                 // ...
        std::transform(xs.begin(), xs.end(), ys.begin(), f);  // ... using range library?
        if (*std::max_element(ys.begin(), ys.end()) > y)
        {
            auto [x0,y0] = random_pick_one_higher_than_threshold(xs, ys, y);
            x = x0;
            y = y0;
            converge = false;
        }
    }
    return x;
}
```

All these hill climbing algorithms ensure objective increases over time. Let's make it less greedy in simulated annealing.

3.2 Simulated annealing

Simulated annealing is a modified hill climbing with a probability of having decrease in objective over time. Probability of decrease in objective is scheduled beforehand, and is decreased over time.

```
double simulated_annealing(double init, double window_size, std::function<double(double)>& f)
{
    state x = init;
    value y = f(x);
    bool converge = false;
    double temp = init_temperature();
    while(!converge)
    {
        converge = true;
        auto [x0,y0] = random_pick_one_within(x-window_size, x+window_size, f);
        if (y0 >  y ||
            y0 <= y && random_01() < exp(-abs(y-y0)/temp)) // as y0 lies further away from y, prob decreases
        {
            x = x0;
            y = y0;
            converge = false;
        }
        temp = temp * 0.99; // we can adopt any schedule for temperature
    }
    return x;
}
```

3.3 Genetic algorithm

Genetic algorithm generalized the simulated annealing to housekeeping a population instead of storing the best candidate in order to avoid stucking in local optimum. Besides, genetic algorithm discretizes any point in the state space as a gene, which is is a string that describes a valid state (or equivalently, a feasible solution). It also defines a connection between two genes (neighbourhood) by crossover and mutation. Suppose we keep a population of N genes, which is an even number. For each iteration, we generate a new set of N genes via 3 steps : natural selection, crossover and mutation.

```
void natural_selection(const std::vector<gene>& gene0, std::vector<gene>& gene1, std::function<double(const gene&)>& f)
{
    // step 1a. calculate objective and sort
    std::vector<std::pair<gene, double>> gene_prob;
    for(const auto& x:gene0) gene_prob.push_back(std::make_pair(x,f(x)));
    std::sort(gene_prob.begin(), gene_prob.end(), [](const auto& x0, const auto& x1) { return x0.second < x1.second; });

    // step 1b. calculate selection probability
    double total = 0;
    for(auto& x:gene_prob) total += x.second;
    for(auto& x:gene_prob) x.second /= total;

    // step 1c. perform natural selection
    for(int n=0; n!=genes0.size(); ++n)
    {
        double p = random_01();
        for(const auto& x:gene_prob)
        {
            if (p <= x.second) { gene1.push_back(x.first); break; }
        }
    }
}

void crossover(const std::vector<gene>& gene1, std::vector<gene>& gene2)
{
    for(int n=0; n!=gene1.size(); n+=2)
    {
        const auto& x = gene[n];
        const auto& y = gene[n+1];
        int crossover_point = rand()%gene[n].size();
        gene2.push_back(gene[n], gene[n+1], m);
        gene2.push_back(gene[n+1], gene[n], m);
    }
}

void mutation(const std::vector<gene>& gene2, std::vector<gene>& gene3)
{
    for(const auto& x:gene2)
    {
        auto y = x;
        for(auto& element:y)
        {
            if (random_01() < small_prob) element.mutate();
        }
        gene3.push_back(y);
    }
}
```

3.4 Gradient descent

Gradient descent is an iterative method to find the minimum of an objective function by following the gradient direction.

$$\arg\min_{X\in\Re^M} f(X) \qquad \rightarrow \qquad X_{t+1} = X_t - sJ^T\Big|_{X=X_t} \qquad where\ \ J = \frac{\partial f(X)}{dX}$$

3.5 Stochastic gradient descent

When the objective function is a sum of *N* observations, stochastic gradient descent can be used to perform online-training, or mini-batch training, in which, gradient of the whole dataset is approximated by gradient of one data point.

$$\arg\min_{X\in\Re^M} \sum_{n=1}^{N} f(A_n \mid X) \qquad \rightarrow \qquad X_{t+1} = X_t - sJ_n^T\Big|_{X=X_t}\ for\ all\ n \qquad where\ \ J_n = \frac{\partial f(A_n \mid X)}{dX}$$

```
para_vec stochastic_graddes(const std::vector<data>& As, const para_vec& X0, std::function<double(const data&, const para_vec&)> f)
{
    para_vec X = X_init;
    while(!converge)
    {
        std::vector<data> Bs = shuffle(As);
        for(const auto& B:Bs)
        {
            para_vec J;
            for(int m=0; m!=para_vec::size; ++m)
            {
                para_vec X_dX = X;
                X_dX[m] += delta;
                J[m] = (f(B, X_dX) - f(B, X)) / delta;
            }
            x -= stepsize * J;
        }
        // determine converge or not
        ...
    }
    return X;
}
```

3.6 Sequential quadratic programming SQP

SQP is a derivative based iterative algorithm for optimization of generic objective function and constraints. SQP combines objective and constraints into a Lagrangian, which is then optimized using Newton method. Newton method for optimization involves both Jacobian and Hessian, which ends up with a linear system. We repeatedly solve the linear system for each parameter and Lagrange multipler, at the same time, update the active-set of constraints, until convergence. Please google *"McCormick Northwestern SQP"*.

It turns out that SQP :

- approximates the objective locally quadratic
- approximates the constraint locally linear

$$\arg\min_{X\in\Re^M} f(X)\ \ such\ that\ \ G(X)=0\ \ and\ \ H(X)\ge 0$$

$X \in \Re^M$	*is a M×1 column matrix*
$f(X):\Re^M \rightarrow \Re$	*is objective (scalar)*
$G(X):\Re^M \rightarrow \Re^{N_E}$	*is a set of N_E equality constraints*
$H(X):\Re^M \rightarrow \Re^{N_I}$	*is a set of N_I inequality constraints*

Step1 Lagrangian

Setting up Lagrangian :

$$L(X,\lambda,\mu) \quad = \quad f(X) - \lambda^T G(X) - \mu^T H(X)$$

where λ is N_E×1 column matrix, μ is N_I×1 column matrix
where G is N_E×1 column matrix, H is N_I×1 column matrix
we adopt Jacobian convention, i.e. multivariate derivative is a row matrix

Step2 Newton method
Next step is to solve :

$$\nabla L(X,\lambda,\mu) \quad = \quad 0 \qquad \textit{both sides are row matrix}$$

$$\rightarrow \quad [\nabla L(X_t,\lambda_t,\mu_t)]^T + \nabla^2 L(X_t,\lambda_t,\mu_t)\begin{bmatrix}\Delta X_t \\ \Delta\lambda_t \\ \Delta\mu_t\end{bmatrix} \quad = \quad 0 \qquad \textit{both sides are column matrix}$$

$$\rightarrow \quad \begin{bmatrix}X_{t+1} \\ \lambda_{t+1} \\ \mu_{t+1}\end{bmatrix} = \begin{bmatrix}X_t \\ \lambda_t \\ \mu_t\end{bmatrix} - [\nabla^2 L(X_t,\lambda_t,\mu_t)]^{-1}[\nabla L(X_t,\lambda_t,\mu_t)]^T$$

$$= \begin{bmatrix}X_t \\ \lambda_t \\ \mu_t\end{bmatrix} - \begin{bmatrix}\partial_{XX}L & \partial_{X\lambda}L & \partial_{X\mu}L \\ \partial_{\lambda X}L & \partial_{\lambda\lambda}L & \partial_{\lambda\mu}L \\ \partial_{\mu X}L & \partial_{\mu\lambda}L & \partial_{\mu\mu}L\end{bmatrix}^{-1}_{X_t,\lambda_t,\mu_t} \begin{bmatrix}[\partial_X L]^T \\ [\partial_\lambda L]^T \\ [\partial_\mu L]^T\end{bmatrix}_{X_t,\lambda_t,\mu_t}$$

$$= \begin{bmatrix}X_t \\ \lambda_t \\ \mu_t\end{bmatrix} - \begin{bmatrix}\partial_{XX}L & -[\partial_X G]^T & -[\partial_X H]^T \\ -\partial_X G & 0 & 0 \\ -\partial_X H & 0 & 0\end{bmatrix}^{-1}_{X_t,\lambda_t,\mu_t} \begin{bmatrix}[\partial_X f - \lambda^T\partial_X G - \mu^T\partial_X H]^T \\ -G \\ -H\end{bmatrix}_{X_t,\lambda_t,\mu_t}$$

- *where* $\partial_X G$ *is* $N_E \times M$ *Jacobian matrix*
- *where* $\partial_X H$ *is* $N_I \times M$ *Jacobian matrix*
- *where* $\partial_{XX} L$ *is* $M \times M$ *Hessian matrix*
- *where* $\partial_{\lambda\mu}L = \partial_\mu[\partial_\lambda L]^T$ *is* $N_E \times N_I$ *Hessian matrix, other 2nd order derivatives are defined in the same way*

Step3 Active set method
Some inequality constraints in H are active (i.e. current solution lies on the constraints), while some are not (i.e. current solution lies within feasible region, therefore these constraints are effectively useless for current iteration). We need to keep track of an active set of inequality contraints *(please read section 4.2 for details)* :

```
std::vector<bool> active_flags(H.size(), true);
while(!converge)
{
    auto [X, lambda, mu] = solve_sqp_linear_system(f, G, H, active_flags, X, lambda, mu);

    // update active set for next iteration
    inactivate_constraint_with_negative_Lagrange_multiplier_mu(active_flags, mu);
      activate_constraint_with_negative_inequality_HX(active_flags, H, X); // check if H(X) >= 0 is fulfilled

    // determine converge or not
    ...
}
return X;
```

3.7 Sequential minimal optimization

Sequential minimal optimization is an efficient online training for large scale constrainted quadratic programming for SVM. In each iteration, it picks one Lagrange multiplier that violates KKT conditions the most, and another Lagrange multiplier randomly, hence effectively reducing the large scale problem into a subproblem with two unknowns, which can be solved analytically.

$$\max_\alpha -\frac{1}{2}\alpha^T H\alpha + l^T\alpha \qquad st \;\; 0 \le \alpha \le C \;\; and \;\; y^T\alpha = 0$$

If α_1 and α_2 are picked, the problem becomes :

$$\max_\alpha -\frac{1}{2}\left(K(x_1,x_1)\alpha_1^2 + 2K(x_1,x_2)\alpha_1\alpha_2 + K(x_2,x_2)\alpha_2^2\right) + \alpha_1 + \alpha_2 + const(\alpha\setminus\{\alpha_1,\alpha_2\}) \qquad st \;\; 0 \le \alpha_{1,2} \le C \;\; and \;\; y_1\alpha_1 + y_2\alpha_2 = const(\alpha\setminus\{\alpha_1,\alpha_2\})$$

$const(\alpha\setminus\{\alpha_1,\alpha_2\})$ *refers to constant depends on all-but-(*α_1 *and* α_2*) Lagrange multipliers*

By substituting one Lagrange multiplier as the other in the objective, we obtain a *1D* quadratic equation.

4.1 Maximization and minimization with equality constraints

4.1a Single equality constraint

We consider both maximization and minimization together to avoid duplication.

$$\arg\max/\min_{X\in\Re^M} f(X) \;\; \textit{such that} \;\; g(X)=0$$

$X\in\Re^M$	*is a M×1 column matrix*
$f(X):\Re^M\to\Re$	*is scalar objective*
$g(X):\Re^M\to\Re$	*is one equality constraint*

Given a feasible intermediate guess X_0, so that $g(X_0)=0$, the necessary and sufficient condition for X_0 being a local optimum is, if we try to vary f by moving ΔX so as to keep $g(X_0+\Delta X)=g(X_0)+\nabla g(X_0)\Delta X=0$, the point X_0 should fulfill the following :

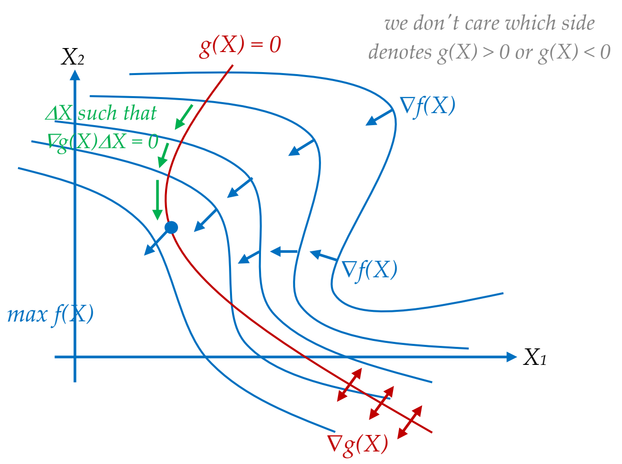
$$\begin{aligned} & g(X_0)=0 \\ \textit{and}\;\; & \nabla f(X_0)\Delta X=0 && \textit{for all } \Delta X \textit{ so that } \nabla g(X_0)\Delta X=0 \\ \textit{and}\;\; & \begin{bmatrix}\Delta X^T\nabla^2 f\Delta X<0 & \textit{for} & \max f \\ \Delta X^T\nabla^2 f\Delta X>0 & \textit{for} & \min f\end{bmatrix} && \textit{for all } \Delta X \textit{ so that } \nabla g(X_0)\Delta X=0 \end{aligned}$$

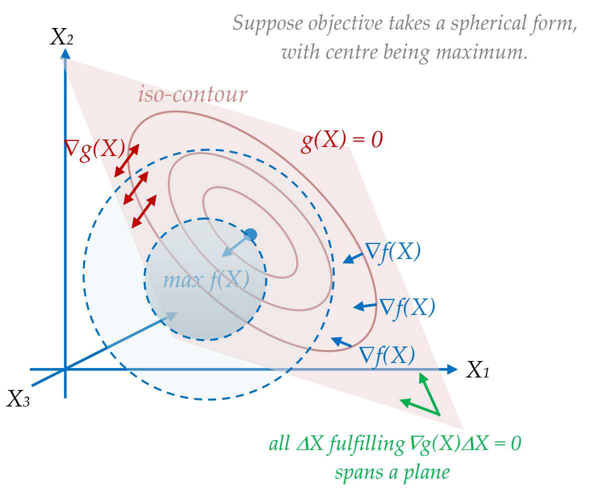
ΔX is column, Jacobian is row

which means that we cannot further optimize f for all possible ΔX that is orthogonal to gradient of g, it happens only when gradient of f is parallel to gradient of g. In other words, there exists a unique non-zero λ such that :

$$\begin{aligned} & g(X_0)=0 \\ \textit{and}\;\; & \nabla f(X_0)=\lambda\nabla g(X_0) \\ \textit{and}\;\; & \begin{bmatrix}\Delta X^T\nabla^2 f\Delta X<0 & \textit{for} & \max f \\ \Delta X^T\nabla^2 f\Delta X>0 & \textit{for} & \min f\end{bmatrix} && \textit{for all } \Delta X \textit{ so that } \nabla g(X_0)\Delta X=0 \end{aligned}$$

ΔX is column, Jacobian is row





Suppose objective takes a spherical form,
with centre being maximum.
X2
iso-contour
∇g(X)
g(X) = 0
∇f(X)
max f(X)
∇f(X)
∇f(X)
X1
X3
all ΔX fulfilling ∇g(X)ΔX = 0
spans a plane

About minimization

Although the above examples are for maximization of f, we can easily turn them into minimization. Let's consider the minimization of $-f$, by flipping the blue arrows, by same reasoning, we end up with the same optimum, as indicated by the solid blue dot.

About Lagrange multiplier

Since g is an equality constraint, we don't care whether gradient of f is in the same direction or in opposite direction to gradient of g, thus as long as $\nabla f \,//\, \nabla g$, Lagrange multipler can take any sign. Besides it is unique, as it is defined by ratio between ∇f and ∇g.

Proof by maths

May we have a mathematical proof of $\nabla f = \lambda \nabla g$? I tried, but it failed, probably because I make the following mistake :

solving for ΔX *such that* $[\nabla f(X_0)\Delta X = 0 \quad \text{and} \quad \nabla g(X_0)\Delta X = 0]$	*is not equivalent to*	$\nabla f(X_0) = \lambda \nabla g(X_0)$
solving for X_0 *such that* $[\nabla f(X_0)\Delta X = 0 \quad \forall \Delta X \quad \text{s.t.} \quad \nabla g(X_0)\Delta X = 0]$	*is equivalent to*	$\nabla f(X_0) = \lambda \nabla g(X_0)$

Its easier to understand by looking at the 3D example. Given a point X_0 lying on an iso-contour on the red plane of $g(X)=0$, with ΔX being a tangent to the iso-contour, then $[X_0, \Delta X]$ pair must fulfill $\nabla f(X_0)\Delta X = 0$ and $\nabla g(X_0)\Delta X = 0$, but definitely not an optimal point.

Proof by geometry

In the 3D example, the spherical objective function intersects the red plane, those iso-contours are the intersection lines. All tangent points to the iso-contour can fulfill the *1st* order requirement $\nabla f(X_0)\Delta X = 0$ and $\nabla g(X_0)\Delta X = 0$, however all the iso-contour points are not optimal, as we can always find a direction ΔX lying on the red plane (i.e. orthogonal to ∇g) having an increase in f value :

if we pick	$\Delta X = proj_{redplane}(\nabla f(X_0))$	
then we have	$\nabla g(X_0)\Delta X = 0$	*since all vectors on red plane are orthogonal to* ∇g
and	$\nabla f(X_0)\Delta X > 0$	*hence we can further optimize* f*, implying that* X_0 *is not optimal*

As the blue sphere contracts, the iso-contour becomes smaller. Eventually the blue sphere touches the red plane (intersecting at one point only), the iso-contour diminished. The projection of gradient of f on the red plane becomes null, meaning that we cannot find one feasible delta move that fulfill the constraint (having $X_0+\Delta X$ lying on the red plane) while having positive $\nabla f(X_0)\Delta X$. In this case, surfaces f and g touch, implying $\nabla f \,//\, \nabla g$, in other words there exists a unique non-zero λ such that $\nabla f = \lambda \nabla g$.

Lagrange formulation

Lets re-state the condition for local maximum or minimuim clearly as, there exists a unique non-zero λ such that :

$$g(X_{opt}) = 0$$

$$\nabla f(X_{opt}) = \lambda \nabla g(X_{opt})$$

$$\begin{cases} \Delta X^T \nabla^2 f \Delta X < 0 & \text{for} \quad \max f \\ \Delta X^T \nabla^2 f \Delta X > 0 & \text{for} \quad \min f \end{cases} \qquad \textit{for all } \Delta X \textit{ so that } \nabla g(X_{opt})\Delta X = 0$$

From now on, I call these three constraints (they follow this order as a convention) :

- feasible constraint
- *1st* order optimality
- *2nd* order optimality

It is more convenient to put it into a Lagrangian formulation.

For Lagrangian $L(X,\lambda) = f(X) - \lambda g(X)$ *there exists a unique non-zero* λ *such that :*

$$\nabla_\lambda L(X_{opt},\lambda_{opt}) = 0 \qquad \textit{which corresponds to } g(X_{opt}) = 0$$

$$\nabla_X L(X_{opt},\lambda_{opt}) = 0 \qquad \textit{which corresponds to } \nabla_X f(X_{opt}) = \lambda \nabla_X g(X_{opt})$$

$$\begin{cases} \Delta X^T \nabla^2 f \Delta X < 0 & \text{for} \quad \max f \\ \Delta X^T \nabla^2 f \Delta X > 0 & \text{for} \quad \min f \end{cases} \qquad \textit{for all } \Delta X \textit{ so that } \nabla_X g(X_{opt})\Delta X = 0$$

4.1b Multiple equality constraints

For multi equality constraint case, we replace *g(X)* by *G(X)* : $\Re^M \rightarrow \Re^{NE}$ and λ becomes a $N_E \times 1$ column matrix. Given a feasible point X_0, then we have to find ΔX so that $X_0+\Delta X$ remains in feasible region. As there are N_E equalities, we have :

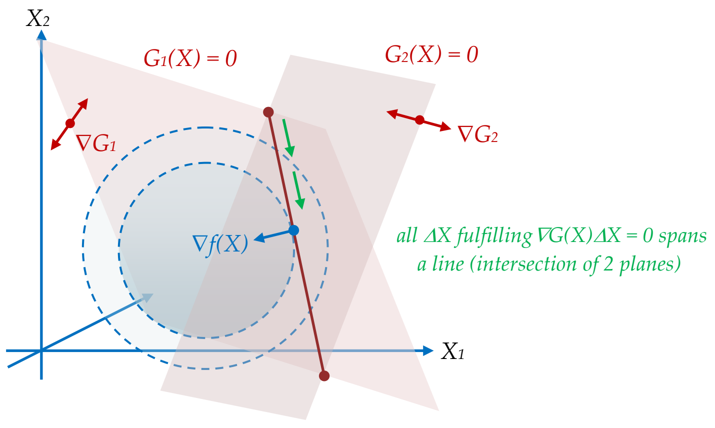
$$\nabla G_n(X_0)\Delta X = 0 \qquad \forall n \in [1, N_E]$$

As ΔX is orthogonal to gradient of each G_n, it must also be orthogonal to the span of all gradients of G. In other words :

$$\lambda^T \nabla G(X_0)\Delta X = 0 \qquad \textit{where } size(\lambda) = N_E \times 1,\ size(G) = N_E \times 1,\ size(\nabla G) = N_E \times M \textit{ and } size(\Delta X) = M \times 1$$

$$\rightarrow \quad \nabla(\lambda^T G(X_0))\Delta X = 0 \qquad \textit{which is liked } \Delta X \textit{ being orthogonal to a combo of constraints}$$

Following the reasoning of single equality constraint, the optimal point is where objective *f* touches the combo of constraints $\lambda^T G(X)$. As shown in the following diagram, the bigger sphere cuts the feasible line at two points. It is then contracted until it touches at one point only. This is the optimal point, on which there exists a unique combination of gradient of *G* equals to gradient of *f*.



X_2
$G_1(X) = 0$
$G_2(X) = 0$
∇G_1
∇G_2
$\nabla f(X)$
all ΔX fulfilling $\nabla G(X)\Delta X = 0$ spans a line (intersection of 2 planes)
X_1

The condition for local maximum or minimuim is, there exists a unique non-zero column matrix λ such that :

$$G(X_{opt}) = 0$$

$$\nabla f(X_{opt}) = \lambda^T \nabla G(X_{opt})$$

$$\begin{bmatrix} \Delta X^T \nabla^2 f \Delta X < 0 & \textit{for} & \max f \\ \Delta X^T \nabla^2 f \Delta X > 0 & \textit{for} & \min f \end{bmatrix} \qquad \textit{for all } \Delta X \textit{ so that } \nabla G(X_{opt})\Delta X = 0$$

Similarly, we can express it as Lagrange formulation (omitted here).

4.2 Maximization and minimization with inequality constraints

Lagrange multiplier can be extended to inequality constraints by KKT and active set (active set is used in `AsmVision::alg::FitAXB`).

4.2a Single inequality constraint

Inequality constraint can be handled by :

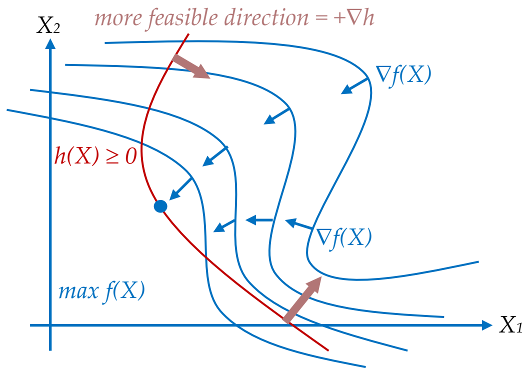
- if the target local optimum lies inside feasible region of an inequality constraint, this constraint is effectively inactive
- if the target local optimum lies outside feasible region of an inequality constraint, this constraint is active
- ► when an inequality constraint is active, it can be regarded as equality constraint, however ...
- ► as only one side of the inequality constraint is feasible (the other is not), thus the sign of Lagrange multipliers matters

Lets consider 4 cases when inequality constraint is active :

- when we maximize *f*, we need a component of $+\nabla f$
- when we minimize *f*, we need a component of $-\nabla f$
- when we move inside feasible side of $h \geq 0$, we need a component of $+\nabla h$
- when we move inside feasible side of $h \leq 0$, we need a component of $-\nabla h$
- when inequality is active, at the optimum point, gradient of *f* is parallel to gradient of *h*, moreover ... the direction matters :
  the gradient towards more optimized f must be ***opposite*** *to the gradient towards more feasible side of h*
  otherwise, it implies that the local optimum already lies within feasible region and the constraint becomes inactive.

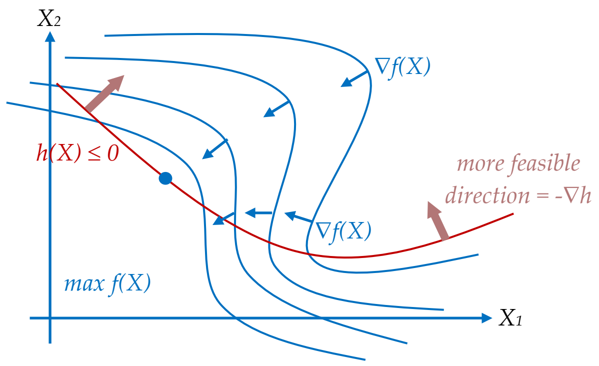
1. $\arg\max_{X\in\Re^M} f(X)$ *such that* $h(X)\geq 0$

grad to max objective $= +\nabla f$
grad to stay feasible $= +\nabla h$
optimality : $\nabla f(X) = -\mu\nabla h(X)$ *where* $\mu > 0$



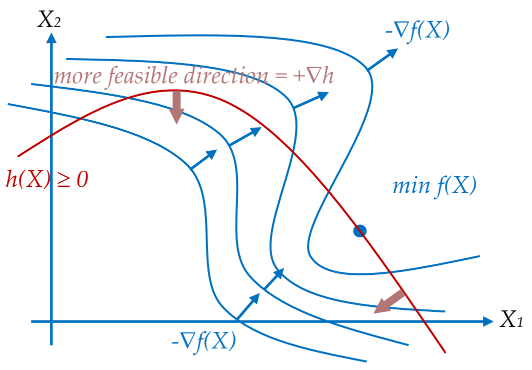
2. $\arg\max_{X\in\Re^M} f(X)$ *such that* $h(X)\leq 0$

grad to max objective $= +\nabla f$
grad to stay feasible $= -\nabla h$
optimality : $\nabla f(X) = -\mu(-\nabla h(X))$ *where* $\mu > 0$



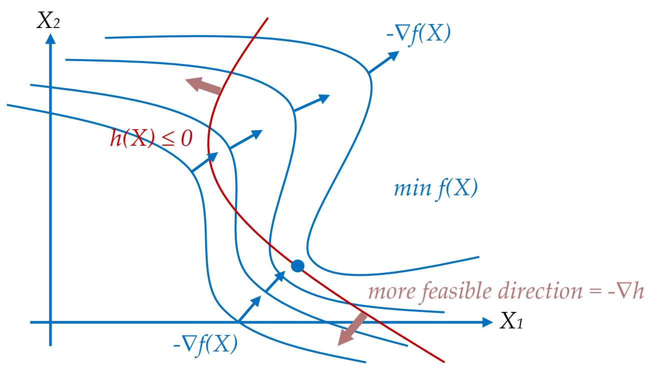
3. $\arg\min_{X\in\Re^M} f(X)$ *such that* $h(X)\geq 0$

grad to max objective $= -\nabla f$
grad to stay feasible $= +\nabla h$
optimality : $-\nabla f(X) = -\mu\nabla h(X)$ *where* $\mu > 0$



4. $\arg\min_{X\in\Re^M} f(X)$ *such that* $h(X)\leq 0$

grad to max objective $= -\nabla f$
grad to stay feasible $= -\nabla h$
optimality : $-\nabla f(X) = -\mu(-\nabla h(X))$ *where* $\mu > 0$



The above diagrams share the same *f*, only *h* is changed. We have $L = \nabla f + \mu\nabla h$ for `(1,4)` and $L = \nabla f - \mu\nabla h$ for `(2,3)`.

The optimality conditions can be summaries as the following. Lets consider case (3), as it is common in SVM formulation :

Case 1 : When local optimum is inside feasible region defined by h(X)≥ 0, then

- *inactive constraint* — *we can consider as* $\mu = 0$
- $h(X_{opt}) > 0$ — *feasible constraint*
- $\nabla f(X_{opt}) = 0$ — *1st order optimality*
- $\Delta X^T \nabla^2 f \Delta X > 0$ — *for all ΔX* — *2nd order optimality*

Case 2 : When local optimum is outside feasible region defined by h(X)≥ 0, then

- *active constraint* — *we can consider as* $\mu > 0$
- $h(X_{opt}) = 0$ — *feasible constraint*
- $\nabla f(X_{opt}) = \mu \nabla h(X_{opt})$ — *where* $\mu > 0$ — *1st order optimality*
- $\Delta X^T \nabla^2 f \Delta X > 0$ — *for all ΔX so that* $\nabla h(X_{opt})\Delta X = 0$ — *2nd order optimality*

By combining both cases together, we can merge the equations in both cases, resulting in KKT condition. The Lagrange formulation for case (3) is :

$$L(X,\mu) = f(X) - \mu h(X)$$

There exists unique set of non-negative μ, such that KKT conditions are fulfilled :

- $h(X_{opt}) \geq 0$ — *feasible constraint*

  $\mu_{opt} \geq 0$

  $\mu_{opt} h(X_{opt}) = 0$
- $\nabla_X L(X_{opt}, \mu_{opt}) = 0$ — *1st order optimality*
- $\Delta X^T \nabla^2 f \Delta X > 0$ — *for all ΔX so that* $\nabla h(X_{opt})\Delta X = 0$ — *2nd order optimality*

4.2b Multiple equality and inequality constraints

Lets consider multiple equality and inequality constraints for case (3).

$$\arg\min_{X \in \Re^M} f(X) \quad such\ that \quad G(X) = 0 \quad and \quad H(X) \geq 0$$

$X \in \Re^M$	*is a M×1 column matrix*
$f(X): \Re^M \to \Re$	*is scalar objective*
$G(X): \Re^M \to \Re^{N_E}$	*is a set of* N_E *equality constraints*
$H(X): \Re^M \to \Re^{N_I}$	*is a set of* N_I *inequality constraints*

The Lagrange formulation is :

$$L(X,\lambda,\mu) = f(X) - \lambda^T G(X) - \mu^T H(X)$$ *where both λ and μ are column matrix*

There exists unique set of positive λ and non-negative μ, such that KKT conditions are fulfilled :

- $G(X_{opt}) = 0$ — *feasible constraint*
- $H(X_{opt}) \geq 0$ — *feasible constraint*

  $\mu_{opt} \geq 0$

  $\mu_{opt,n} H_n(X_{opt}) = 0 \qquad \forall n \in [1, N_I]$
- $\nabla_X L(X_{opt}, \lambda_{opt}, \mu_{opt}) = 0$ — *1st order optimality*
- $\Delta X^T \nabla^2 f \Delta X > 0$ — *for all ΔX so that* $\begin{bmatrix} \nabla G(X_{opt})\Delta X = 0 \\ \nabla H(X_{opt})\Delta X = 0 \end{bmatrix}$ — *2nd order optimality*