Application of Contemporary Search and Optimization Methods to Discover Best Model for Achieving Score Function Goal

U.F. Eze B.Sc. M.Sc. Ph.D, MNCS, MCPN

G. N. Okeudo PhD

Paschal Chinedu
Federal University of Technology, Owerri

Abstract
This research work tries to discuss how some contemporary search and optimization methods are applied in order to discover the best model to solve some problems that involved score function principles. Search and optimization methods are one of the components of data mining algorithms that identify the fundamental rules to determine the values of parameters that reduce general score functions. Different ways of searching for models and pattern such as State-Space Formulation, Simple Greedy Search algorithm, Branch-and-Bound and Systematic Search and Heuristics were explored. Equally, parameter optimization was discussed with a view to analyse the greedy method for optimizing smooth functions, Gradient method of closed form and linear algebra methods, univeriate optimization, optimization parameters and types. Cost functions were minimized and maximized using constant optimization. Expectations maximization (EM) was discussed based on E-step and EM cookbook. Finally, the stochastic search and optimization technique that improves the chances of discovering the global optimum was discussed.

Key words: Search, Optimization, Method, Models and Score Functions

1.0 INTRODUCTION

Search and optimization methods are crucial in practical data mining because there are wider classes of model structures that can be implemented to show knowledge in structured manner as well as stating the principles of how such structures can be scored in terms of how well they matched the observed data. Search and optimization methods pay attention the rules for searching and optimizing our parameters and structures as directed by the available data and present score functions. Score functions numerically express people's preference for one model or pattern over another. Selection of models depends on the type of problem one wants to solve. *Once score function is selected, irrespective of being specific in nature, the next thing is to optimize it with the aim to minimize the score function instead of maximizing it.* Search and optimization methods identify the basic determine the values of parameters that minimizes a general score function.

Practically, there is no significant difference between the discrete parameters and continuous parameters. Discrete parameter are indexed on different classes of models so that each might correspond to trees next to neural networks, another polynomial functions etc. Discrete parameters can also assume only integral values such as all variables to be incorporated in the model. Continuous parameters provide the mean value of a distribution or a parameter vector giving the centers of a set of clusters into which the data set has been partitioned.

Recall, that in data mining algorithms, attention is paid on discovering sets of models, patterns or regions with parameter space. However attention is not paid on single best model, pattern or parameter vector with respect to the selected score functions as obtained in Bayesian averaging methods and in searching for sets of pattern. Obviously, recent algorithms are designed with a view to eliminate multiple passes through the data since the data sets are so numerous in a typical data mining situation. Finally, a lot of problems involve score functions that contain multiple minima (as well as maxima), therefore, the Stochastic method was developed to improve the opportunities of discovering the world-wide optimum instead of local optimum.

1.1 SEARCHING FOR MODELS AND PATTERN

The search background involves some general high level issues of search which embraces the following:

- Simple search strategy for models

This strategy is an exhaustive search where for *every* model in the candidate set, find the best parameters with respect to the score function, and then compare the scores of all the models to find the best.

Pros:

•Guarantee to find the best model with respect to the score function
•May be implemented in a parallel fashion

Cons:

•Re-optimize the parameter for each new model structure
•Face to potentially combinatorial explosion
•Highly inefficient, and most of the time infeasible!

- **Being smart with compromise (I)**

This strategy makes use of the decomposable score function where the score function for a new structure will be an additive function of the score function for the previous structure as well as a term accounting for the change in structure

Pros:

•Easy to obtain the score function value of the current model based on the previous model

Cons:

•Limited to particular score function

- **Being smart with compromise (II)**

This strategy uses approximation for the best parameter ("incremental")

Pros:

- Leaves the existing parameters in the model fixed to the previous values and only optimize the parameters added to the model
- Reduces the number of parameters to be estimated when changing the model structure for a little bit
- Can save up time for searching more candidate model structures

Cons:

•Proven to be suboptimal
•"Error" accumulation problem

- **Being smart with compromise (III)**

This strategy involves Heuristic search which applies some heuristic to narrow down the search space of the model structures

Pros:

•Efficient under combinatorial explosions
•Intuitive, and easy to implement

Cons:

•Lack of mathematically validity
•Might be ineffective under certain unknown situation.

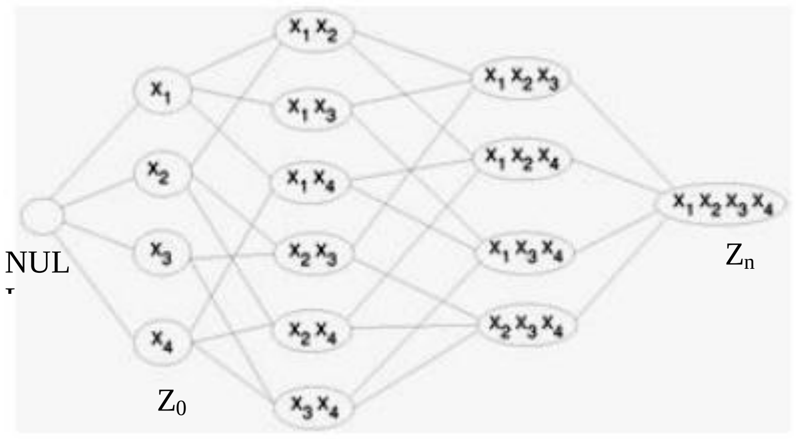
1.2 STATE-SPACE FORMULATION FOR MODEL SEARCH

State-Space Formulation model search deals with discrete space analysis. Model search problem can be viewed as, *one moving through a discrete set of states*

- State space representation
  - Each state corresponds to a particular model in the candidate set
  - Each state can be represented as a vertex in a graph

- Search operators
  - Search operators corresponds to legal "moves" in our search space
  - Can be represented as edges between the (state) vertices in a graph

Let us illustrate search operators capabilities by using four variables x_1, x_2, x_3, x_4 with the initial node being null (no variables).



Z_0 = subset of single variables
Z_n = an individual model structure within the general family.

Fig. 1- A single state-space with four variables

Search operators here are assumed to be directed edges in the state-space graph. Directed edges start from Z_0 to another model structure $Z_{n.}$

1.3. SIMPLE GREEDY SEARCH ALGORITHM

The simple greedy search algorithm involves the following steps

- **Initialize:**

Choose an initial state *M*0, corresponding to a particular model structure *Mk*

- **Iterate:**

Evaluate the score function at all possible adjacent states (as defined by the operators) and ***move to the best one***

- **Stopping criterion:**

Repeat step 2 until no further improvement can be attained in the local score function

- **Multiple restart:**

Repeat steps 1 through 3 from different initial starting points and choose the best solution found.

1.4 SYSTEMATIC SEARCH

- **Search tree**

Instead of following the “single path” to search the “best” at every step, we keep track of multiple models simultaneously. ***Consume huge memories Memory-efficient***

- **Traverse on search tree**

– Blind search
- Breadth-first search
- Depth-first search

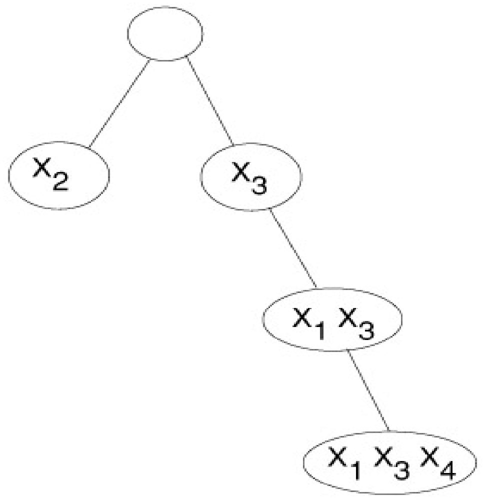


Fig1.1- Simple search tree for the state-space of Fig.1.0

- **Traverse on search tree with heuristics**

▪ Beam search
▪ Keep track of the best models at any point in the search!
▪ *Suboptimal. A trade-off for efficiency!*

- Branch-and-bound

▪ Keep track of the best model structure so far
▪ Calculate *analytically a lower bound* on the best possible score function from a particular branch of the search tree
▪ If the bound is greater than the best score function so far, *prune this branch.*

Demerits

▪ Difficult to find a tight bound
▪ Scalability is limited!

1.5 BRANCH-AND-BOUND ALGORITHMS

A counter-part of the backtracking search algorithm, in the absence of cost criteria, the algorithm traverses a spanning tree of the solution space using the breadth-first approach. That is, a queue is used, and the nodes are processed in first-in-first-out order. If a cost criteria is available, the node to be expanded next (i.e., the branch) is the one with the best cost within the queue. In such a case, the cost function may also be used to discard (i.e., the bound) from the queue nodes that can be determined to be expensive. A priority queue is needed here.

Branch and bound (**BB** or **B&B**) is a general algorithm for finding optimal solutions of various optimization problems, especially in discrete and combinatorial optimization. It consists of a systematic enumeration of all candidate solutions, where large subsets of fruitless candidates are discarded *en masse*, by using upper and lower estimated bounds of the quantity being optimized.
The method was first proposed by A. H. Land and A. G. Doig in 1960 for discrete programming.

General description
For definiteness, we assume that the goal is to find the *minimum* value of a function *f*(*x*), where *x* ranges over some set *S* of *admissible* or *candidate solutions* (the *search space* or *feasible region*). Note that one can find the *maximum* value of *f*(*x*) by finding the minimum of *g*(*x*) = − *f*(*x*). (For example, *S* could be the set of all possible trip schedules for a bus fleet, and *f*(*x*) could be the expected revenue for schedule *x*.)

A branch-and-bound procedure requires two tools. The first one is a *splitting* procedure that, given a set *S* of candidates, returns two or more smaller sets $S_1, S_2, \cdots$ whose union covers *S*. Note that the minimum of *f*(*x*) over *S* is $\min\{v_1, v_2, \ldots\}$, where each v_i is the minimum of *f*(*x*) within S_i. This step is called **branching**, since its recursive application defines a tree structure (the *search tree*) whose *nodes* are the subsets of *S*.
Another tool is a procedure that computes upper and lower bounds for the minimum value of *f*(*x*) within a given subset *S*. This step is called **bounding**.

The key idea of the BB algorithm is: if the *lower* bound for some tree node (set of candidates) *A* is greater than the *upper* bound for some other node *B*, then A may be safely discarded from the search. This step is called **pruning**, and is usually implemented by maintaining a global variable *m* (shared among all nodes of the tree) that records the minimum upper bound seen among all sub regions examined so far. Any node whose lower bound is greater than *m* can be discarded.
The recursion stops when the current candidate set *S* is reduced to a single element; or also when the upper bound for set *S* matches the lower bound. Either way, any element of *S* will be a minimum of the function within *S*.

Effective subdivision
The efficiency of the method depends strongly on the node-splitting procedure and on the upper and lower bound estimators. All other things being equal, it is best to choose a splitting method that provides non-overlapping subsets.

Ideally the procedure stops when all nodes of the search tree are either pruned or solved. At that point, all non-pruned sub regions will have their upper and lower bounds equal to the global minimum of the function. In practice the procedure is often terminated after a given time; at that point, the maximum lower bound and the minimum upper bound, among all non-pruned sections, define a range of values that contains the global minimum. Alternatively, within an overriding time constraint, the algorithm may be terminated when some *error criterion*, such as *(max − min)/(min + max)*, falls below a specified value.

The efficiency of the method depends critically on the effectiveness of the branching and bounding algorithms used; bad choices could lead to repeated branching, without any pruning, until the sub-regions become very small. In that case the method would be reduced to an exhaustive enumeration of the domain, which is often impractically large. There is no universal bounding algorithm that works for all problems, and there is little hope that one will ever be found; therefore the general paradigm needs to be implemented separately for each application, with branching and bounding algorithms that are specially designed for it.

Branch and bound methods may be classified according to the bounding methods and according to the ways of creating/inspecting the search tree nodes. The branch-and-bound design strategy is very similar to backtracking in that a state space tree is used to solve a problem. The differences are that the branch-and-bound method (1) does not limit us to any particular way of traversing the tree and (2) is used only for optimization problems. This method naturally lends itself for parallel and distributed implementations.

CHAPTER TWO

2.0 PARAMETER OPTIMIZATION METHODS

For a given model structure, we can link the parameter for this model structure to the score function, and directly optimize over the score function by using score function specified in terms of a sum of local error function(such as when the training data points exist independently.

$$S(\theta) = \sum_{i=1}^{N} e\left(y(i), \hat{y}_\theta(i)\right)$$

Fig2.1- Simple Score Function

Where yΘ(i) = model's estimate of the target value y(i) in data training

e = an error function in distance measuring between model's prediction and the target.

S(Θ) = optimization solution for a linear optimization problem(ie quadratic function of Θ.

•Optimization can be down by calculating the minimum value directly

$$\mathbf{g}(\theta) = \nabla_\theta S(\theta) = \left(\frac{\partial S(\theta)}{\partial \theta_1}, \frac{\partial S(\theta)}{\partial \theta_2}, \ldots, \frac{\partial S(\theta)}{\partial \theta_d}\right)$$

Where g(Θ) = Gradient function of S for a complex optimization problem (non linear.

2.1 CLOSE FORM AND LINEAR ALGEBRA METHODS

–Close form solution:

Let $\mathbf{g}(\theta) = \nabla_\theta S(\theta) = 0$, solving *d* linear equations (ie gradient g(Θ) is linear in Θ

2.1.1 Greedy method for optimizing smooth functions

Greedy method for optimizing smooth functions takes the following steps

- Initialize

(Randomly) Choose an initial value for the parameter vector

- Iterate:

$$\theta^{i+1} = \theta^i + \underbrace{\boxed{\lambda^i}}_{\text{How much to change the value}} \overbrace{\boxed{\mathbf{v}^i}}^{\text{Determined by Local information}}$$

Where i begin with 0

- Convergence

Repeat 2 until S appears to have attained a local minimum

- Multiple restart

Repeat steps 1 through 3 from different initial starting points and choose the best solution found

2.2 ITERATIVE OPTIMIZATION

- Gradient method

Gradient based optimization strategies iteratively search a minimum of a D dimensional target function $f(\vec{x})$. The target function is thereby approximated by a terminated Taylor series expansion around $\vec{x}_0$:

$$f(\vec{x}_0 + \vec{x}) \approx f(\vec{x}_0) + (\nabla f(\vec{x}_0))^T \vec{x} + \frac{1}{2}\vec{x}^T \nabla^2 f(\vec{x}_0)\vec{x}$$

The actual optimization is performed iteratively. After an initial value $\vec{x}_{i=0}$ was chosen and the target function $f(\vec{x}_i)$ was computed, the following loop tries to achieve a minimum:

1. Check the convergence criterion of $f(\vec{x}_i)$; terminate with $\vec{x}_i$ as the solution if fulfilled.
2. Compute a direction $\vec{p}_i$ and a step width l_i
3. Evaluate the new point $\vec{x}_{i+1} = \vec{x}_i + \vec{p}_i \cdot l_i$; compute the target $f(\vec{x}_{i+1})$ and go to step 1.

To compute the step direction p_i a linear approximation (first order) of the target function can be used:

$$f(\vec{x}_0 + \vec{p}) \approx f(\vec{x}_0) + (\nabla f(\vec{x}_0))^T \vec{p}$$

which results in the step direction:

$$\vec{p} = -\nabla f(\vec{x}_0)$$

This is called the steepest descent method. A second order approach uses a quadratic approximation:

$$\nabla f(\vec{x}_0) + \nabla^2 f(\vec{x}_0)\,\vec{p} = 0$$

This is referred to as Newton's direction method.

To compute the step width l_k the parameter vector of the next iteration is calculated by

$$\vec{x}_{k+1} = \vec{x}_k + \alpha_k \vec{p}_k$$

The parameter α_k denotes the step width and is chosen depending on the used optimization algorithm.

For an analytically given target function the first and second order derivatives can directly be transferred into a computer program, however if no explicit formula can be defined, the target is computed numerically by means of a simulation. In this case approximations for the derivatives are necessary. In the following the approximation of a univariate target function $f(x)$ is given. For multivariate target functions the finite difference approximation is applied for each dimension.

The performance of a gradient based method strongly depends on the initial values supplied. Several optimization runs with different initial values might be necessary if no a priori knowledge (e.g., the result of a process simulation) about the function to optimize can be applied.

2.3 UNIVARIATE OPTIMIZATION

❖ The Newton-Raphson Method

–Based on Taylor series expansion

$$g(\theta^s) \approx g(\theta^*) + (\theta^s - \theta^*)g'(\theta^*)$$

– Since $g(\theta^s) = 0$ then, $\theta^s \approx \theta^* - \frac{g(\theta^*)}{g'(\theta^*)}$

Second order method

– So the update rule is $\theta^{i+1} = \theta^i - \frac{g(\theta^i)}{g'(\theta^i)}$

Pros:

•**Convergence rate is quadratic if close to the solution**

Cons:

•**May not converge at all if far from the solution**

❖ The Gradient descent Method

–The update rule is:

$$\theta^{i+1} = \theta^i - \lambda g(\theta^i)$$

•Momentum-Based methods:

–Accelerate the convergence of gradient descent by adding momentum term.

$$\theta^{i+1} = \theta^i + \Delta^i$$

$$\Delta^i = -\lambda g(\theta^i) + \mu \Delta^{i-1}$$

Considering the history of the path information

Accelerate the convergence in low curvature region.

❖ Bracketing Method

–Finding a bracket that provably contains the extremum of the function

2.4 OPTIMIZATION PARAMETERS

An *optimization parameter* (or a decision variable, in the terms of optimization) is a model parameter to be optimized. For example, the number of nurses to employ during the morning shift in an emergency room may be an optimization parameter in a model of a hospital. The OptQuest Engine searches through possible values of optimization parameters to find optimal parameters. It is possible to have more than one optimization parameter.

Only a parameter of the main active object class of the optimization experiment can be an optimization parameter. So, in order to perform optimization, you must have at least one parameter in this active object class. If you need to optimize parameters of embedded objects, you should use parameter propagation.

The dimension of the search area depends on the number of optimization parameters. Each new parameter expands the search area, thus slowing down the optimization. If you have N optimization parameters, their ranges form the N-dimensional square search area. Obviously, that area must be wide enough to contain the optimal point. However, the wider the range is, the more time is needed to find the optimum in the search area. On the other hand, suggested parameter values located near the optimal value can shorten the time it takes to find the optimal solution.

2.5 OPTIMIZATION PARAMETER TYPES

During the optimization process, the parameter's value is changed in accordance to its type within an interval, specified by lower and upper bounds. There are the following types of optimization parameters:

- Continuous parameter
- Discrete parameter
- Design parameter

Continuous parameter can take any value from the interval. The parameter precision determines the minimal value continuous parameters can change.
Discrete parameter is represented by a finite set of decisions with essential direction: the parameter influences the objective like a numeric parameter, but can take values from the specified set only. It begins at a lower bound and increments by a step size up to an upper bound.

Sometimes the range and step are exactly defined by the problem; but generally you will have to choose them. If you specify the step for the parameter, only the discrete points will be involved in the optimization, so it will be impossible to determine optimal parameter value more precisely than defined by the step. So, if you are not sure what the step should be, choose the **Continuous** rather than the **Discrete** parameter type.

Design parameter is represented by a finite set of decisions, where there is no clear sense of direction. Value of design parameter represents an alternative but not a quantity. It begins at a lower bound and increments by a step size up to an upper bound. Values order is inconsequential. Using design parameters you can model choosing the best alternative from the catalogue, where the choices are not in a specific order. For example, a design parameter, which can take values 0 or 1 (min=0, max=1, step=1) may represent a choice between: a model has some element or has not.

Defining Optimization Parameters
Optimization parameters are defined in the **Parameters** table on the **General** properties page of the optimization experiment. The table already lists all parameters of the root active object class. By default, all of them are *fixed*, i.e. they are not varied by optimization process.

To make parameter a decision variable

1. Select the optimization experiment in the **Projects** view.
2. On the **General** page of the **Properties** view, go to the row of the **Parameters** table containing the parameter you want make a decision variable.
3. Click the **Type** field and choose the type of the optimization parameter other than **fixed**. Depending on the type of the parameter, the list of possible values may vary: **design**, **int**, **discrete** for *integer* parameters; **continuous** and **discrete** for *double*, etc.
4. Specify the range for the parameter. Enter the parameter's lower bound in the **Min** field and the parameter's upper bound in the **Max** field.
5. For **discrete** and **design** parameters, specify the parameter step in the **Step** field.
6. Suggest the initial value for the parameter in the **Suggested** field. Initially, the value is set to the parameter's default value, but you can enter any other value here.

2.6 CONSTRAINT OPTIMIZATION

In constraint satisfaction, **constraint optimization** (also called *constrained optimization*) seeks for a solution maximizing or minimizing a cost function.

Definition
A constraint optimization problem can be defined as a regular constraint satisfaction problem in which constraints are weighted and the goal is to find a solution maximizing the weight of satisfied constraints.

Alternatively, a constraint optimization problem can be defined as a regular constraint satisfaction problem augmented with a number of "local" cost functions. The aim of constraint optimization is to find a solution to the problem whose cost, evaluated as the sum of the cost functions, is maximized or minimized. The regular constraints are called *hard constraints*, while the cost functions are called *soft constraints*. These names illustrate that hard constraints are to be necessarily satisfied, while soft constraints only express a preference of some solutions (those having a high or low cost) over other ones (those having lower/higher cost).

A general constrained optimization problem may be written as follows:

$$\begin{array}{lll} \max & f(\mathbf{x}) & \\ \text{subject to} & g_i(\mathbf{x}) = c_i & \text{for } i = 1, \cdots, n \quad \text{Equality constraints} \\ & h_j(\mathbf{x}) \leq d_j & \text{for } j = 1, \cdots, m \quad \text{Inequality constraints} \end{array}$$

Where $\mathbf{x}$ is a vector residing in a n-dimensional space, $f(\mathbf{x})$ is a scalar valued objective function, $g_i(\mathbf{x}) = c_i \text{ for } i = 1, \cdots, n$ and $h_j(\mathbf{x}) \leq d_j \text{ for } j = 1, \cdots, m$ are constraint functions that need to be satisfied.

2.6.1 SOLUTION METHODS

- **Branch and bound**

Constraint optimization can be solved by branch and bound algorithms. These are backtracking algorithms storing the cost of the best solution found during execution and use it for avoiding part of the search. More precisely, whenever the algorithm encounters a partial solution that cannot be extended to form a solution of better cost than the stored best cost, the algorithm backtracks, instead of trying to extend this solution.

Assuming that cost is to be maximized, the efficiency of these algorithms depends on how the cost that can be obtained from extending a partial solution is evaluated. Indeed, if the algorithm can backtrack from a partial solution, part of the search is skipped. The lower the estimated cost, the better the algorithm, as a lower estimated cost is more likely to be lower than the best cost of solution found so far.

On the other hand, this estimated cost cannot be lower than the effective cost that can be obtained by extending the solution, as otherwise the algorithm could backtrack while a solution better than the best found so far exists. As a result, the algorithm requires an upper bound on the cost that can be obtained from extending a partial solution, and this upper bound should be as small as possible.

- **First-choice bounding functions**

One way for evaluating this upper bound for a partial solution is to consider each soft constraint separately. For each soft constraint, the maximal possible value for any assignment to the unassigned variables is assumed. The sum of these values is an upper bound because the soft constraints cannot assume a higher value. It is exact because the maximal values of soft constraints may derive from different evaluations: a soft constraint may be maximal for $x = a$ while another constraint is maximal for $x = b$.

- **Russian doll search**

This method runs a branch-and-bound algorithm on *n* problems, where *n* is the number of variables. Each such problem is the subproblem obtained by dropping a sequence of variables $x_1, \ldots, x_i$from the original problem, along with the constraints containing them. After the problem on variables $x_{i+1}, \ldots, x_n$ is solved, its optimal cost can be used as an upper bound while solving the other problems,

In particular, the cost estimate of a solution having $x_{i+1}, \ldots, x_n$ as unassigned variables is added to the cost that derives from the evaluated variables. Virtually, this corresponds on ignoring the evaluated variables and solving the problem on the unassigned ones, except that the latter problem has already been solved. More precisely, the cost of soft constraints containing both assigned and unassigned variables is estimated as above (or using an arbitrary other method); the cost of soft constraints containing only unassigned variables is instead estimated using the optimal solution of the corresponding problem, which is already known at this point.

- **Bucket elimination**

The bucket elimination algorithm can be adapted for constraint optimization. A given variable can be indeed removed from the problem by replacing all soft constraints containing it with a new soft constraint. The cost of this new constraint is computed assuming a maximal value for every value of the removed variable. Formally, if *x* is the variable to be removed, $C_1, \ldots, C_n$are the soft constraints containing it, and $y_1, \ldots, y_m$are their variables except *x*, the new soft constraint is defined by:

$$C(y_1 = a_1, \ldots, y_n = a_n) = \max_a \sum_i C_i(x = a, y_1 = a_1, \ldots, y_n = a_n)$$

Bucket elimination works with an (arbitrary) ordering of the variables. Every variable is associated a bucket of constraints; the bucket of a variable contains all constraints having the variable has the highest in the order. Bucket elimination proceeds from the last variable to the first. For each variable, all constraints of the bucket are replaced as above to remove the variable. The resulting constraint is then placed in the appropriate bucket.

CHAPTER THREE

3.0 OPTIMIZATION WITH MISSING DATA SUCH AS EXPECTATION MAXIMIZATION ALGORITHM

Engineering optimization relies routinely on deterministic computer based design evaluations, typically comprising geometry creation, mesh generation and numerical simulation. Simple optimization routines tend to stall and require user intervention if a failure occurs at any of these stages. This motivated us to develop an optimization strategy based on surrogate modelling, which penalizes the likely failure regions of the design space without prior knowledge of their locations. A Gaussian process based design improvement expectation measure guides the search towards the feasible global optimum.

New areas of research or the early stages of a design process require a geometry model, with coupled analysis codes, which encompasses a wide range of possible designs. In such situations the analysis code is likely to fail to return results for some configurations due to a variety of reasons. Problems may occur at any stage of the design evaluation process: (i) the geometry described by a parameter combination may be infeasible, (ii) an automated mesh generator may not be able to cope with a particularly complex or corrupt geometry or (iii) the simulation may not converge, either due to mesh problems or the automated solution setup being unable to cope with certain unusual, yet geometrically feasible, configurations.

In an ideal world, a seamless parameterization would ensure that the optimizer could visit wide ranging geometries and move between them without interruption.

In reality, however, the uniform, flawless coverage of the design space by a generic geometry model and simulation procedure is fraught with difficulties. In all but the most trivial or over-constrained cases the design evaluation can fail in certain areas of the search space. Of course, if these infeasible areas are rectangular, they can simply be avoided by adjusting the bounds of the relevant variables, but this is rarely the case. Most at times such regions will have complex, irregular, and even disconnected shapes, and are much harder to identify and avoid. In the presence of such problems it is difficult to accomplish entirely automated optimization without narrowing the bounds of the problem to the extent that promising designs are likely to be excluded. This negates the possible benefits of global optimization routines, and the designer may well choose to revert to a more conservative local optimization around a known good design.

Although industry and academia direct considerable effort at robust geometry creation and mesh generation, it is unlikely that truly fault free automated design evaluation will be achieved. Rather than tackling problems with the individual design evaluation components, we address, or rather circumvent, the problem of failed design evaluations via the medium of surrogate based optimization methodology. This approach allows us to cope with regions of infeasible designs without knowing where they will occur. Although gradient-free search methods, for example a genetic algorithm (GA), can complete a global search of a domain with infeasible regions, the large number of direct calls to the computer code will render such methods impractical when using time consuming simulations, as shown when we compare the performance of our surrogate based methods with a GA in §4.

3.1 THE EXPECTATION MAXIMIZATION ALGORITHM

The Expectation-Maximization (EM) algorithm is used for finding MLE of parameters in probabilistic models that depends on unobserved under variables.
•It is an iterative procedure altering between Expectation step and Maximization step

- **E step**: computes an expectation of the log likelihood w.r.t. the current estimate of the distribution for the hidden variables

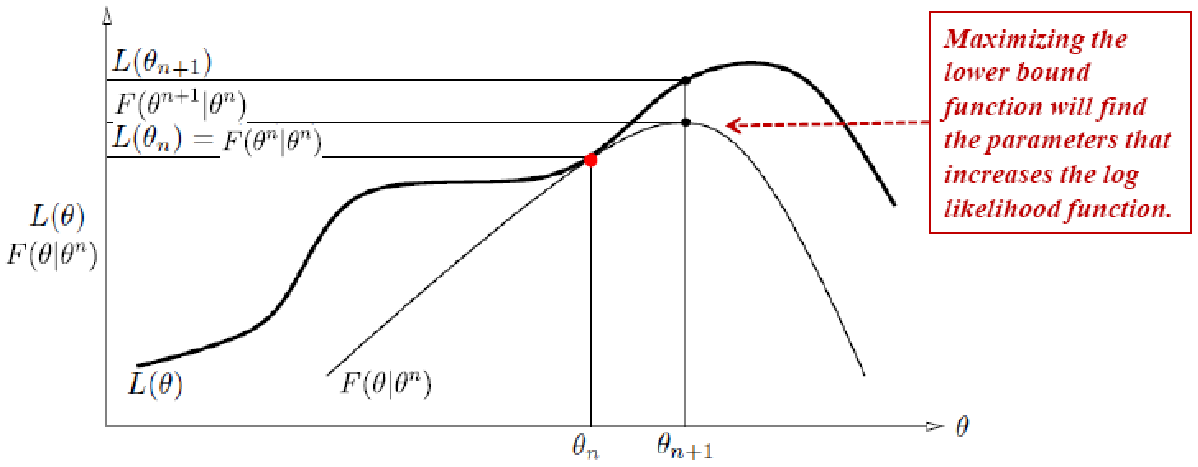
$$F(\theta|\theta^k) = \mathbb{E}_{Z|D,\theta^k}[\log L(\theta; D, Z)]$$

- **M step**: computes the parameters which maximize the expected log likelihood found on the E step

$$\theta^{k+1} \arg\max_{\theta} F(\theta|\theta^k)$$

3.2 WHY EXPECTATION MAXIMIZATION WORKS? (I)

•Idea behind EM
–Find a **lower bound** of log likelihood function
–At each step, optimize the lower bound w.r.t. one set of unknown parameters by fixing other set of parameters at **their current values**
–Iterates until the process **converges**


$L(\theta_{n+1})$
$F(\theta^{n+1}|\theta^n)$
$L(\theta_n) = F(\theta^n|\theta^n)$
$L(\theta)$
$F(\theta|\theta^n)$
$L(\theta)$
$F(\theta|\theta^n)$
θ_n
θ_{n+1}
θ
Maximizing the lower bound function will find the parameters that increases the log likelihood function.

Encodes the basic idea of EM with some mathematics

•The lower bound:

$$L(\theta) = \log \sum_{Z} p(D, Z|\theta) = \log \sum_{Z} Q(Z) \frac{p(D, Z|\theta)}{Q(Z)}$$

Jessen's inequality

$$\geq \sum_{Z} Q(Z) \log \frac{p(D, Z|\theta)}{Q(Z)} = F(Q, \theta)$$

•Iterative optimization:

$$\text{From k-th rounds to the (k+1)-th rounds} \quad \begin{cases} Q^{k+1} = \arg\max_{Q} F\left(Q, \theta^{k}\right) \\ \theta^{k+1} = \arg\max_{\theta} F\left(Q^{k+1}, \theta\right) \end{cases}$$

•Convergence:
–*F* value will be increased during the any successive rounds
–*F* is bounded from the top by *L* which has an maximum value

3.3 THE EXPECTATION MAXIMIZATION COOKBOOK
To use EM, the following issues should be determined:
–What is the log likelihood function? (EM can apply to any likelihood function)
–What are the model parameters and what are the hidden variables?
–What is the expectation in the E step, and how to compute it?
–What is supposed to be maximized based on the expectation, and how to compute it?

3.5 ONLINE AND SINGLE- SCAN ALGORITHMS
•In many online applications
–Data come in stream
–Limited space for storing training data
–Need quick response for a given input.

•We can only receive one data point, make use of it and then discard it.

•Stochastic approximation would be applied
–Reform the "batch" score function to **"instantaneous" score function** which only depends on the current example.
–**Optimize** the instantaneous score function (e.g., take a gradient descent)
–The average instantaneous score function should **asymptotically approach to the batch score function** to guarantee the approximation is good.

CHAPTER FOUR

4.0 STOCHASTIC SEARCH AND OPTIMIZATION TECHNIGUE

Stochastic search and optimization techniques are used in a vast number of areas, including aerospace, medicine, transportation, and finance, to name but a few. Whether the goal is refining the design of a missile or aircraft, determining the effectiveness of a new drug, developing the most efficient timing strategies for traffic signals, or making investment decisions in order to increase profits, stochastic algorithms can help researchers and practitioners devise optimal solutions to countless real-world problems.

Introduction to Stochastic Search and Optimization: Estimation, Simulation, and Control is a graduate-level introduction to the principles, algorithms, and practical aspects of stochastic optimization, including applications drawn from engineering, statistics, and computer science. The treatment is both rigorous and broadly accessible, distinguishing this text from much of the current literature and providing students, researchers, and practitioners with a strong foundation for the often-daunting task of solving real-world problems.

The text covers a broad range of today's most widely used stochastic algorithms, including:

- Random search
- Recursive linear estimation
- Stochastic approximation
- Simulated annealing
- Genetic and evolutionary algorithms
- Machine (reinforcement) learning
- Model selection
- Simulation-based optimization
- Markov chain Monte Carlo
- Optimal experimental design

Stochastic optimization (SO) methods are optimization algorithms which incorporate probabilistic (random) elements, either in the problem data (the objective function, the constraints, etc.), or in the algorithm itself (through random parameter values, random choices, etc.), or in both. The concept contrasts with the deterministic optimization methods, where the values of the objective function are assumed to be exact, and the computation is completely determined by the values sampled so far.

4.1 METHODS FOR STOCHASTIC FUNCTIONS

Partly-random input data arise in such areas as real-time estimation and control, simulation-based optimization where Monte Carlo simulations are run as estimates of an actual system, and problems where there is experimental (random) error in the measurements of the criterion. In such cases, knowledge that the function values are contaminated by random "noise" leads naturally to algorithms that use statistical inference tools to estimate the "true" values of the function and/or make statistically optimal decisions about the next steps. Methods of this class include:

- stochastic approximation (SA), by Robbins and Monro (1951)
    - stochastic gradient descent
    - finite-difference SA by Kiefer and Wolfowitz (1952)
    - simultaneous perturbation SA by Spall (1992)

Randomized search methods

On the other hand, even when the data set consists of exact measurements, it is sometimes beneficial to deliberately introduce randomness into the search process as a means of speeding convergence and making the algorithm less sensitive to modelling errors. Further, the injected randomness may provide the necessary impetus to move away from a local solution when searching for a global optimum. Indeed, this randomization principle is known to be a simple and effective way to obtain algorithms with *almost certain* good performance uniformly across many data sets, for many sorts of problems. Stochastic optimization methods of this kind include:

- simulated annealing by S. Kirkpatrick, C. D. Gelatt and M. P. Vecchi (1983)
    - adaptive simulated annealing
- quantum annealing
- cross-entropy method by Rubinstein and Kroese (2004)
- random search by Anatoly Zhigljavsky (1991)
- basin hopping technique a.k.a. Monte Carlo with minimization
- stochastic tunneling
- parallel tempering a.k.a. replica exchange
- stochastic hill climbing
- Harmony search
- swarm algorithms
    - ant colony optimization
    - particle swarm optimization
    - Bees algorithm
    - Firefly algorithm
    - stochastic diffusion search
    - evolutionary algorithms
    - genetic algorithms by JOHN H. HOLLAND (1975)
    - CMA-ES (covariance matrix adaptation evolution strategy) by Hansen et al.

REFERENCES

- Dechter, R. (2003). *Constraint Processing*. Morgan Kaufmann. ISBN 1-55860-890-7.
- David Hand, Heikki Mannila, Padhraic Symith (2001). Principles of Data Mining.Eastern Economy Edition (c) 2001
- Forrester, A., So´Bester, A. & Keane, A. (2006). Optimization with Missing Data. *Computational Engineering and Design Group, School of Engineering Sciences, University of Southampton, Southampton* SO17 1BJ, UK. Available online at (Accessed: March 20, 2011)

- Li, M. (2010). Data Mining: Chapter 8: Search and Optimization Methods.
  Department of Computer Science and Technology, Nanjing University. Available online a
  (Accessed: March 20, 2011)

- Link: http://www.cse.ohio-state.edu/~gurari/course/cis680/cis680Ch20.html

- Link: http://www.mutah.edu.jo/userhomepages/CS252/branchandbound.html

- Link: http://en.wikipedia.org/wiki/Branch_and_bound