Training Support Vector Machines with Particle Swarms

deur

Ulrich Paquet

Voorgelê as 'n deel van die vereistes vir die graad

Magister Scientiae

in die Fakulteit Ingenieurswese, Bouomgewing, en Inligtingstegnologie

Universiteit van Pretoria

November 2003

Die finansiële ondersteuning van die *National Research Foundation* (NRF) tot hierdie navorsing word hierdeur erken. Opinies en gevolgtrekkings in hierdie verhandeling is dié van die outeur, en kan nie noodwendig aan die NRF toegeskryf word nie.

Training Support Vector Machines with Particle Swarms

by

Ulrich Paquet

Submitted in partial fulfilment of the requirements for the degree

Magister Scientiae

in the Faculty of Engineering, Built Environment and Information Technology

University of Pretoria

November 2003

The financial assistance of the National Research Foundation (NRF) towards this research is hereby acknowledged. Opinions expressed in this thesis and conclusions arrived at, are those of the author and not necessarily to be attributed to the National Research Foundation.

Training Support Vector Machines with Particle Swarms

deur

Ulrich Paquet

Opsomming

Deelswerms kan met gemak gebruik word om 'n funksie, wat beperk word deur 'n stel lineêre beperkings, te optimeer. 'n "Lineêre Deelswermoptimeerder" en 'n "Konvergente Lineêre Deelswermoptimeerder" word ontwikkel om sulke beperkte funksies te optimeer. As die hele swerm aanvanklik slegs uit geldige oplossings bestaan, dan kan die swerm die beperkte funksie optimeer sonder om ooit weer die stel beperkings te oorweeg. Om 'n Ondersteuningsvektormasjien te leer moet 'n beperkte kwadratiese programmeringsprobleem opgelos word, en die Konvergente Lineêre Deelswermoptimeerder voldoen aan die behoeftes van 'n optimeringsmetode vir Ondersteuningsvektormasjiene. Deelswerms is intuïtief en maklik om te implementeer, en word aangebied as 'n alternatief tot huidige metodes om Ondersteuningsvektormasjiene te leer.

Studieleier: Prof. A.P. Engelbrecht
Departement Rekenaarwetenskap
Degree: Magister Scientiae

Training Support Vector Machines with Particle Swarms

by

Ulrich Paquet

Abstract

Particle swarms can easily be used to optimise a function with a set of linear equality constraints, and a "Linear Particle Swarm Optimiser" and "Converging Linear Particle Swarm Optimiser" is developed to optimise such constrained functions. It is shown that if the entire swarm of particles is initialised to consist of only feasible solutions, then the swarm can optimise the constrained objective function without ever again considering the set of constraints. Training a Support Vector Machine requires solving a constrained quadratic programming problem, and the Converging Linear Particle Swarm Optimiser ideally fits the needs of an optimisation method for Support Vector Machine training. Particle swarms are intuitive and easy to implement, and is presented as an alternative to current numeric Support Vector Machine training methods.

Supervisor: Prof. A.P. Engelbrecht
Department of Computer Science
Degree: Magister Scientiae

Preface

The question that originally spurred the research in this thesis was - "can a Particle Swarm Optimiser be used to train a Support Vector Machine, and to what extent will it be successful?"

Training a Support Vector Machine (SVM) involves solving a quadratic programming problem, with a single linear constraint, and a set of non-negativity constraints. At first this problem seemed trivial - the objective function that needs to be minimised is convex, and the Particle Swarm Optimiser (PSO) will not be trapped in any local minima.

The difficulty in the problem arose with developing a method to handle the linear constraint. This has led to the development of the Linear (and Converging Linear) PSO algorithms (LPSO and CLPSO), which both have unique properties needed not only for handling the single linear constraint, but any set of (feasible) linear constraints. The non-negativity constraints have led to the extension of both new Particle Swarm algorithms to include cases when constraints appear as boxed constraints. With the addition of slack variables to an optimisation problem with linear constraints, it becomes possible to solve any of these problems.

The main contributions made by this thesis are therefore:

1. An algorithm for SVM training, which is based on analysis of a method for decomposing the SVM quadratic programming problem.

2. The development of LPSO for general optimisation problems, and a proof of a condition on the initial swarm guaranteeing that any point in the search space can be reached.

3. A proof that LPSO is ideally suited for linearly constrained optimisation, with a precondition needed for LPSO to always satisfy linear equality constraints.

4. The extension of LPSO to CLPSO to preclude premature convergence, and a proof that CLPSO will at least converge to a local minimum.

5. The addition of a method to LPSO and CLPSO needed for inequality constraint handling, and the implementation of CLPSO with inequality constraints as an optimiser in SVM training.

In a sense this thesis has delivered more than its original aim. The new PSO algorithms will probably be of greater importance to further milestones in the Swarm Intelligence community, than its application in SVM training.

Chapter 1 puts SVMs under the magnifying glass, and sets the main optimisation problem (a quadratic programming problem) that forms the backbone of this thesis. SVM training has unique problems of its own, primarily because the training problem shows quadratic growth as the training set size increases. Methods for SVM training are discussed in *Chapter 2*, and a training algorithm, based on standard methods of decomposing the main optimisation problem into subproblems, are used to construct a correct training algorithm. *Chapter 3* introduces PSO as an optimisation algorithm, and discusses some of the recent advancements to the PSO method itself. The PSO is extended to handle constrained problems in *Chapter 4*, and LPSO and CLPSO are developed. This extension includes a rigorous analysis of the newly developed algorithms. The successes and failures of LPSO and CLPSO are empirically shown in *Chapter 5*. It is also shown how the CLPSO can be used to train SVMs from a very large character recognition dataset. Finally, *Chapter 6* provides an overview, and gives some thoughts for further research.

Many people have contributed to the successful completion of this thesis. Foremost, I am greatly indebted to professor Andries Engelbrecht for introducing me to the world of artificial intelligence, and for his patient guidance throughout my research.

Ulrich Paquet
Pretoria, South Africa
June 2003

Commit thy works unto the LORD, and thy thoughts shall be established. Proverbs 16:3

Contents

List of Figures

List of Tables

Chapter 1

Support Vector Machines

This chapter discusses the development and basic theoretic building blocks of Support Vector Machines. An overview of both linear and non-linear Support Vector Machines is given from the viewpoint of pattern recognition. Kernel methods are introduced, and the chapter concludes with the Support Vector Machine training problem that will play a key role in the chapters to follow.

1.1 Introduction to Support Vector Machines

Support Vector Machines (SVMs) are a young and important addition to the machine learning toolbox. Having been formally introduced at the 1992 Workshop on Computational Learning Theory [6], SVMs have proved their worth. In the following decade there has been a remarkable growth in both the theory and practice of these learning machines. The original treatments of Support Vector Machines (SVMs) are due to [6, 13, 21, 56, 58].

Traditionally, a SVM is a learning machine for two-class classification problems, and learns from a set of examples. The algorithm aims to do a separation between the two classes by creating a linear decision surface between them. This surface is, however, not created in input space, but rather in a very high-dimensional feature space. Because the feature space is non-linearly related to the input space, the resulting model is non-linear. Special properties of the decision surface ensures high generalisation abilities of SVMs.

Although the Support Vector (SV) algorithm appears to be a linear algorithm in a high-dimensional space, no computations are done in that high-dimensional space. All computations are performed directly in input space by making use of kernel functions. Due to the use of Kernel Methods (KMs), a seemingly complex algorithm for non-linear pattern recognition or regression can be implemented and analysed as a simple linear algorithm. KMs are very modular. Any kernel function can be used with any kernel-based learning algorithm, and any kernel-based learning algorithm can work with any kernel function. By combining simple kernels to complex ones, the kernel functions themselves can also be derived in a modular way.

The SV algorithm makes use of "support vectors" to define the decision surface. Support vectors are a subset of the training patterns, or training vectors. These patterns can be called the most informative, and it is this subset of informative patterns that define the architecture of a SVM. If all non-support vector training patterns (the "uninformative" patterns) are removed, and the SVM retrained, the solution will be exactly the same.

SVMs are popular due to two main reasons. Firstly, an important characteristic of SVMs is its mathematical tractability and geometric interpretation. The SV algorithm is based on very theoretical and intuitive ideas. Secondly, SVMs have shown to be accurate in practical applications, with successes in fields as diverse as text categorisation, bioinformatics and machine vision.

The algorithm holds learning theory in one hand, and practice in the other. Statistical learning theory can be used to identify factors needed for certain algorithms to learn successfully. Complex models and algorithms – such as neural networks – are often needed for practical real-world applications. These models are, however, hard to analyse theoretically. SVMs construct models that are complex enough, with the advantage that the models are relatively simple to analyse mathematically. These models include a large class of neural networks, radial basis function (RBF) networks, and special cases of polynomial classifiers.

SVMs have become an increasingly popular alternative to neural networks. In comparison to neural networks, SVMs have only a small number of tuneable parameters. The SV algorithm also defines the architecture of the learning machine. The SVM training process is characterised by solving a convex quadratic programming problem. The solutions to the training problem are global, and usually unique [9]. A great benefit of SVM training is the absence of local minima (or maxima), and the learning parameters converge monotonically toward the solution.

Applications and theory of SVMs have been extended far beyond basic classification tasks to handle pattern recognition, regression, operator inversion, density estimation, and novelty detection. For pattern recognition, SVMs have been successfully applied in the areas of isolated handwritten digit recognition [8, 13, 44, 45], speaker identification [42], text categorisation [23], face detection in images [39] and object recognition [4]. In the case of regression estimation problems, SVMs have been compared to benchmark time series prediction tests [34, 36]. SVMs have also been used for density estimation [59] and ANOVA decomposition [51].

Although the SV algorithm is firmly rooted in statistical learning theory, learning theory is not included in this work. An excellent explanation can be found in [56, 57]. This chapter focuses on the creation of SVMs: The basic idea behind pattern recognition is explained, which is used in constructing an optimal hyperplane and linear SVMs for linearly separable data. The linear SVM is then adapted to handle nonseparable classification problems. Finally, SVMs are extended to non-linear classification models by the use of kernel functions.

1.2 Pattern recognition

By observing their environment, machines can learn to distinguish interesting patterns. These patterns can be any entity that can be given a name, for example a handwritten character or word, a fingerprint, a face, or a speech signal. After learning, the machine should be able to make intelligent decisions about the categories of similar patterns – this process is called pattern recognition.

Pattern recognition algorithms can be divided into two principal groups. Identifying a pattern as a member of a predefined class is called *supervised* learning and classification. If the algorithm learns classes of patterns based on a measure of similarity, the process is called *unsupervised* learning, or clustering. Unsupervised classification assigns a pattern to one of these determined classes. A SVM is an example of supervised classification, learning from example patterns with class labels.

For a given pattern recognition problem, the objective is to estimate a function $f : \mathbb{R}^N \to \{\pm 1\}$ using a finite set of training data. The training data set consists of a total of l N-dimensional patterns $\mathbf{x}_i$ and their respective class labels y_i, i.e.

$$\{\mathbf{x}_1, y_1\}, \ldots, \{\mathbf{x}_l, y_l\} \in \mathbb{R}^N \times \{\pm 1\} \tag{1.1}$$

If a new pattern $\{\mathbf{x}, y\}$ is generated from the same underlying probability density function $P(\mathbf{x}, y)$ as the training data, then f should correctly classify this example – that is, $f(\mathbf{x}) = y$.

1.3 Linear Support Vector Machines

When training data is linearly separable, a separating hyperplane (a hyperplane that separates the positive from the negative examples) of the form

$$\mathbf{w} \cdot \mathbf{x} + b = 0 \tag{1.2}$$

can be fitted to correctly classify training patterns. Here the vector $\mathbf{w}$ is normal to the hyperplane, and defines its orientation. This hyperplane is shown in Figure 1.1. From equation (1.2), a decision function

$$f(\mathbf{x}) = \mathrm{sign}(\mathbf{w} \cdot \mathbf{x} + b). \tag{1.3}$$

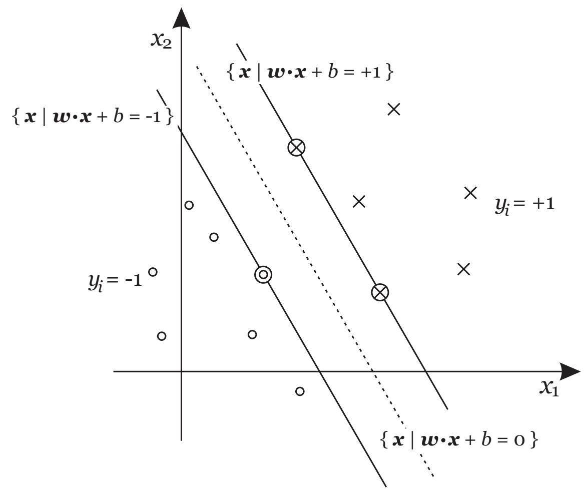


FIGURE 1.1: An example of a classification problem in two dimensions, with the support vectors encircled.

can be derived, with f classifying both positive ($y_i = +1$) and negative ($y_i = -1$) patterns.

Let d_+ (d_-) be the shortest distance from the separating hyperplane to the closest positive (negative) example, then the margin of the hyperplane is defined as the sum $d_+ + d_-$. An optimal hyperplane for a linearly seperable set of training data is here defined as the linear decision function with maximal margin between the vectors of the two classes, as is shown in Figures 1.2(a) and 1.2(b). The support vector algorithm will construct this optimal separating hyperplane.

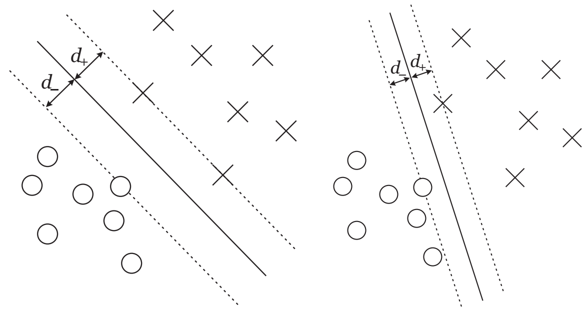
It was shown in [55] that the optimal hyperplane will have good generalisation abilities, and only a relatively small amount of training data is needed to construct this plane. The set of margin-determining training vectors are called the *support vectors*. It was also shown that if the training vectors are separated without errors by an optimal hyperplane, the expected value of the probability of committing an error on a test example is bounded by the ratio between the expected number of support vectors and the number of training vectors:

$$E[P(\text{error})] \leq \frac{E[\text{number of support vectors}]}{\text{number of training vectors}} \tag{1.4}$$

The bound given in equation (1.4) does not explicitly contain the dimensionality of the space of separation. If the optimal hyperplane can thus be constructed from a small number of support vectors relative to the training set size, the generalisation ability will be high, even in an infinite-dimensional space.

Assume all training data satisfy

$$\begin{aligned} \mathbf{w} \cdot \mathbf{x}_i + b &\geq +1 \quad \text{for} \quad y_i = +1 \\ \mathbf{w} \cdot \mathbf{x}_i + b &\leq -1 \quad \text{for} \quad y_i = -1 \end{aligned} \tag{1.5}$$



(a) Classification with a large margin

(b) Classification with a small margin

FIGURE 1.2: Constructing an optimal hyperplane

as shown in Figure 1.1. This can be combined into a single set of equalities:

$$y_i(\mathbf{w} \cdot \mathbf{x}_i + b) - 1 \geq 0 \qquad i = 1, \ldots, l \tag{1.6}$$

where l is the training set size.

To find the optimal separating hyperplane, it is necessary to maximise the margin $d_+ + d_-$. Suppose $\mathbf{x}_1$ and $\mathbf{x}_2$ with $y_1 = +1$ and $y_2 = -1$ are positive and negative points closest to the hyperplane. For maximal separation, the hyperplane should be as far away as possible from each of them. By letting $||\cdot||$ be the l_2 norm of a vector, get

$$\begin{aligned}
\mathbf{w} \cdot \mathbf{x}_1 + b &= +1 \\
\mathbf{w} \cdot \mathbf{x}_2 + b &= -1 \\
\Rightarrow \mathbf{w} \cdot (\mathbf{x}_1 - \mathbf{x}_2) &= +2 \\
\Rightarrow \frac{\mathbf{w}}{||\mathbf{w}||} \cdot (\mathbf{x}_1 - \mathbf{x}_2) &= \frac{2}{||\mathbf{w}||}
\end{aligned}$$

Maximising the margin is equivalent to maximising $\frac{2}{||\mathbf{w}||}$, which is in turn the same as solving

$$\min_{\mathbf{w},b} \quad \frac{1}{2}||\mathbf{w}||^2 \tag{1.7}$$

subject to the constraints in (1.6). Constructing the optimal hyperplane is therefore a convex quadratic problem.

A standard optimisation technique, Lagrange multipliers [19], is used in constructing this optimal hyperplane. There are two main reasons for doing this. The first is that the constraints in (1.6) will be replaced by constraints on the Lagrange multipliers themselves, which will be much easier to handle. The second reason is that, in the Lagrangian reformulation, the training data will only appear as dot products between vectors. This is the crucial property that allows generalisation to the non-linear case. The Lagrange multipliers, $\alpha_i \geq 0$, are introduced for each of the constraints in (1.6) to get the following Lagrangian:

$$L(\mathbf{w}, b, \boldsymbol{\alpha}) = \frac{1}{2}||\mathbf{w}||^2 - \sum_{i=1}^{l} \alpha_i (y_i(\mathbf{w} \cdot \mathbf{x}_i + b) - 1) \tag{1.8}$$

The objective is to minimise (1.8) with respect to $\mathbf{w}$ and b, under the requirement that the derivatives of the Lagrangian with respect to all the α_i vanish. This must be subject to the constraint that the Lagrange multipliers α_i remain non-negative.

Since all the constraints are linear and thus convex, their intersection is also convex. Because the objective function is also convex, the problem is a convex quadratic programming problem. Thus it is possible to *equivalently* solve the dual optimisation problem of maximising (1.8), such that the gradient of L with respect to $\mathbf{w}$ and b vanishes, and requiring that $\alpha_i \geq 0$. That is,

$$\frac{\partial}{\partial b} L(\mathbf{w}, b, \boldsymbol{\alpha}) = 0, \qquad \frac{\partial}{\partial \mathbf{w}} L(\mathbf{w}, b, \boldsymbol{\alpha}) = 0 \tag{1.9}$$

and thus

$$\sum_{i=1}^{l} y_i \alpha_i = 0, \qquad \mathbf{w} = \sum_{i=1}^{l} \alpha_i y_i \mathbf{x}_i \tag{1.10}$$

By substituting (1.10) into (1.8), the dual form of the optimisation problem is derived. Determine

$$\max_{\boldsymbol{\alpha}} \; W(\boldsymbol{\alpha}) = \sum_{i=1}^{l} \alpha_i - \frac{1}{2} \sum_{i=1}^{l} \sum_{j=1}^{l} \alpha_i \alpha_j y_i y_j \mathbf{x}_i \cdot \mathbf{x}_j \tag{1.11}$$

subject to

$$\alpha_i \geq 0, \quad i = 1, \ldots, l \qquad \text{and} \qquad \sum_{i=1}^{l} \alpha_i y_i = 0 \tag{1.12}$$

Thus, by solving the dual optimisation problem, the coefficients α_i are obtained. These coefficients are then used to calculate $\mathbf{w}$ from equation (1.10). The vector $\mathbf{w}$ will be a solution to problem (1.7). The decision function from equation (1.3) can be rewritten as

$$f(\mathbf{x}) = \text{sign}\left(\sum_{i=1}^{l} y_i \alpha_i \mathbf{x} \cdot \mathbf{x}_i + b \right) \tag{1.13}$$

The decision surface of (1.13) is determined by the l Lagrange multipliers α_i. These multipliers are either zero or positive. The subset of zero multipliers will have no effect on the decision function, and can be omitted. It is the subset of positive multipliers that

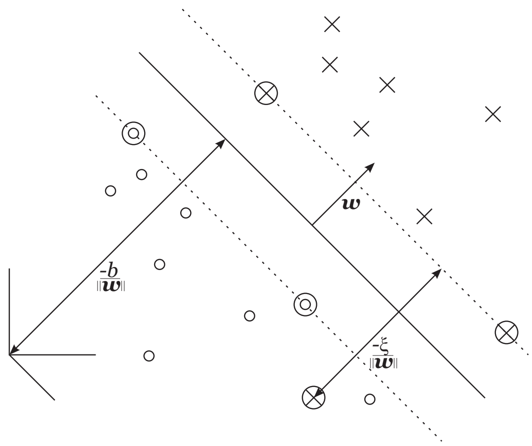


FIGURE 1.3: An example of a linear separating hyperplane for the non-separable case.

influences the classification, and their corresponding training vectors are called the *support vectors*.

The ideas presented in this section only handle separable data. Real data are usually non-separable data, and some examples might violate (1.6). In the following section, SVMs are extended to handle misclassifications.

1.4 Soft margin hyperplanes

In many cases it is impossible to separate the training data without errors, as illustrated in Figure 1.3. If separation by a hyperplane is impossible, the margin between patterns of the two classes becomes arbitrarily small, and the constrained dual Lagrangian (1.11) will grow arbitrarily large.

In this case the separation of the training set can be done with a minimal number of errors (misclassifications), by relaxing the constraints given in (1.6). Here the notion of "soft margin classifiers" are introduced. Add l nonnegative slack variables ξ_i to relax the hard-margin constraints:

$$y_i(\mathbf{w} \cdot \mathbf{x}_i + b) \geq 1 - \xi_i, \qquad \xi_i \geq 0, \qquad i = 1, \dots, l \tag{1.14}$$

Thus for an error to occur, the value of ξ_i must exceed one. It is clear that $\sum_i \xi_i$ is an upper bound on the number of training errors. The natural way to assign an extra cost for errors is to change to objective function to be minimised from (1.7), to solving

$$\min_{\mathbf{w},b,\xi} \quad \frac{1}{2}||\mathbf{w}||^2 + C\sum_{i=1}^{l} \xi_i \tag{1.15}$$

Here $C > 0$ is an arbitrarily chosen – and problem dependent – parameter. A larger value of C assigns a *greater* penalty to errors, since it constrains $\sum_i \xi_i$ to a smaller value. A smaller C allows $\sum_i \xi_i$ to be larger. The functional in (1.15) describes (for sufficiently large C) the problem of constructing a separating hyperplane which minimises the sum of deviations, ξ, of training errors and maximises the margin for the correctly classified vectors [13].

The problem in (1.15) is also a convex quadratic programming problem. Since the values of ξ_i do not appear in the dual Lagrangian, (1.11) must again be maximised subject to

$$0 \leq \alpha_i \leq C, \quad i = 1, \ldots, l \qquad \text{and} \qquad \sum_{i=1}^{l} \alpha_i y_i = 0 \tag{1.16}$$

A crucial property of the quadratic programming problem in (1.11, 1.12) and the decision function $f(\mathbf{x}) = \text{sign}(\sum_i y_i \alpha_i \mathbf{x} \cdot \mathbf{x}_i + b)$ is that they depend only on dot products between patterns. It is this property that allows generalisation to the non-linear case.

1.5 Non-linear Support Vector Machines

A set of linear classifiers, as presented in the method above, is often not rich enough for more diverse classification problems. What is needed is a method that handles non-linear classification equally well. Linear SVMs can very easily be generalised to include these non-linear decision functions: Boser *et al* [6] showed that the so-called *kernel trick* [1] can accomplish this generalisation. Notice that the training patterns only appear in the form of dot products $\mathbf{x}_i \cdot \mathbf{x}_j$ in equations (1.11, 1.13). A non-linear transformation can be done on the set of input vectors to a higher dimensional space (where the dot product is defined), and the linear separation can be done in this higher dimensional space. The data are thus mapped into some other dot product space – a *feature space* – $\mathcal{F}$ via the non-linear map

$$\Phi : \mathbb{R}^N \to \mathcal{F} \tag{1.17}$$

The only requirement on $\mathcal{F}$ is that it is equipped with the dot product operator. No assumptions are made on the dimensionality of $\mathcal{F}$; it can possibly be an infinite-dimensional space. For a given training data set, the SVM is now constructed in $\mathcal{F}$ instead of $\mathbb{R}^N$, i.e. using the set of examples

$$\{\Phi(\mathbf{x}_1), y_1\}, \ldots, \{\Phi(\mathbf{x}_l), y_l\} \in \mathbb{R}^N \times \{\pm 1\} \tag{1.18}$$

From this mapped set of examples a decision function in $\mathcal{F}$ has to be estimated. Intuitively, the difficulty of constructing a decision function in input space should grow with the dimension of the patterns. Statisticians call this difficulty the *curse of dimensionality*

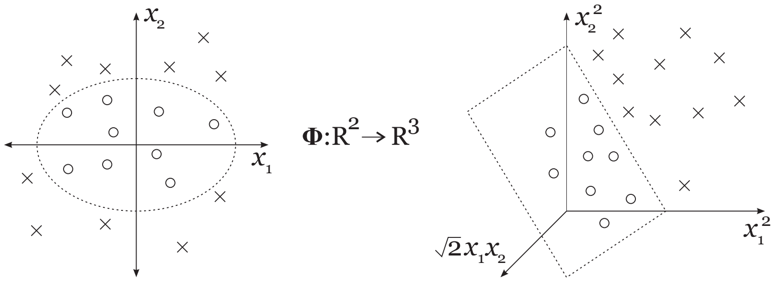


FIGURE 1.4: An example of two-dimensional classification. The three-dimensional feature space is defined by monomials x_1^2, $\sqrt{2}x_1x_2$, and x_2^2, where a linear decision surface is constructed. This construction corresponds to a non-linear ellipsoidal decision boundary in $\mathbb{R}^2$.

– a function of dimension N needs exponentially many patterns to sample the space properly. Considering the curse of dimensionality, mapping to a higher dimensional feature space seems like a dubious idea.

The contrary can, however, be true. Statistical learning theory shows that learning in $\mathcal{F}$ can be simpler if functions of a lower complexity are used. It is the complexity of the function class, not the dimensionality, that matters. The richness of a powerful function class is then introduced by the mapping Φ.

This idea can be understood by considering a simple class of decision rules, namely linear classifiers. Consider the toy example in Figure 1.4, where the training vectors are two-dimensional. A complicated non-linear decision surface is needed to separate the training examples in input space. By defining the mapping

$$\begin{aligned} \Phi : \mathbb{R}^2 &\rightarrow \mathbb{R}^3 \\ (x_1, x_2)^T &\mapsto (x_1^2, \sqrt{2}x_1x_2, x_2^2)^T \end{aligned} \tag{1.19}$$

a *linear* hyperplane separates the mapped training vectors in a three-dimensional feature space. The feature space is defined by the second order monomials x_1^2, $\sqrt{2}x_1x_2$, and x_2^2. This construction corresponds to a non-linear ellipsoidal decision boundary [35].

In the above example, both the statistical complexity and the algorithmic complexity of the learning machine were controlled. The statistical complexity was controlled by using a simple linear hyperplane classifier. Using a three-dimensional feature space kept the algorithmic complexity low.

A technical problem arises in real-world problems, since the algorithmic complexity cannot be kept low. Patterns may be images of 16×16 pixels, a 256-dimensional input space. When fourth order monomials are used as mapping Φ, the feature space would contain all the fourth order products of 256 pixels, and its dimension will be $\binom{4+256-1}{4} \approx 200$ million.

In 1992 it was shown that the problem of treating such high-dimensional spaces could be overcome [6]. Instead of making a non-linear transformation of the input vectors followed by dot products with support vectors in the feature space $\mathcal{F}$, the order of operations is interchanged. A comparison is first done between two vectors in input space (for example by taking their dot product or some distance measure), and then a non-linear transformation of the value of the result is made. The comparison and transformation is done by a *kernel function.*

In the toy example of Figure 1.4, the computation of a dot product between two feature space vectors can be reformulated in terms of a kernel function k:

$$\begin{aligned}
\Phi(\mathbf{x}_i)\cdot\Phi(\mathbf{x}_j) &= \begin{pmatrix} x_{i1}^2 \\ \sqrt{2}x_{i1}x_{i2} \\ x_{i2}^2 \end{pmatrix} \cdot \begin{pmatrix} x_{j1}^2 \\ \sqrt{2}x_{j1}x_{j2} \\ x_{j2}^2 \end{pmatrix} \\
&= x_{i1}^2 x_{j1}^2 + 2x_{i1}x_{i2}x_{j1}x_{j2} + x_{i2}^2 x_{j2}^2 \\
&= \left(\begin{pmatrix} x_{i1} \\ x_{i2} \end{pmatrix} \cdot \begin{pmatrix} x_{j1} \\ x_{j2} \end{pmatrix} \right)^2 \\
&= (\mathbf{x}_i \cdot \mathbf{x}_j)^2 \\
&= k(\mathbf{x}_i, \mathbf{x}_j)
\end{aligned} \tag{1.20}$$

Training a non-linear SVM thus requires the computation of dot products $\Phi(\mathbf{x}_i)\cdot\Phi(\mathbf{x}_j)$ in the feature space $\mathcal{F}$, and can be reduced by defining a suitable kernel function, k, such that

$$k(\mathbf{x}_i, \mathbf{x}_j) = \Phi(\mathbf{x}_i)\cdot\Phi(\mathbf{x}_j) \tag{1.21}$$

The question, which function k corresponds to a dot product in some feature space $\mathcal{F}$, arises. In other words, how can a map Φ be constructed such that k computes the dot product in the space Φ maps to? This has been dealt with by [6, 56], and the answer is seen from Mercer's theorem of functional analysis [14]:

To guarantee that there exits a mapping Φ and an expansion

$$k(\mathbf{u}, \mathbf{v}) = \Phi(\mathbf{u})\cdot\Phi(\mathbf{v}) = \sum_i \Phi(\mathbf{u})_i \Phi(\mathbf{v})_i \tag{1.22}$$

it is necessary and sufficient that the condition

$$\iint k(\mathbf{u}, \mathbf{v}) g(\mathbf{u}) g(\mathbf{v})\, d\mathbf{u}\, d\mathbf{v} \geq 0 \tag{1.23}$$

be valid for all g for which

$$\int g^2(\mathbf{u})\, d\mathbf{u} < \infty \tag{1.24}$$

As an example, consider the toy example of Figure 1.4, with the kernel defined in equation (1.20), and $\mathbf{x}$ a two-dimensional vector. To show that Mercer's condition is satisfied for $k(\mathbf{x}_i, \mathbf{x}_j) = (\mathbf{x}_i \cdot \mathbf{x}_j)^2$, it must be shown that

$$\iint (\mathbf{x}_i \cdot \mathbf{x}_j)^2 g(\mathbf{x}_i) g(\mathbf{x}_j)\, d\mathbf{x}_i\, d\mathbf{x}_j \geq 0 \tag{1.25}$$

hold for all g with finite L_2 norm, i.e. g must satisfy equation (1.24). Expanding and factorising the left-hand side of the above inequality gives the needed result.

$$\begin{aligned}
& \iint (x_{i1}^2 x_{j1}^2 + 2x_{i1}x_{i2}x_{j1}x_{j2} + x_{i2}^2 x_{j2}^2) g(\mathbf{x}_i) g(\mathbf{x}_j)\, d\mathbf{x}_i\, d\mathbf{x}_j \\
= & \int x_{i1}^2 g(\mathbf{x}_i)\, d\mathbf{x}_i \int x_{j1}^2 g(\mathbf{x}_j)\, d\mathbf{x}_j + 2\int x_{i1}x_{i2} g(\mathbf{x}_i)\, d\mathbf{x}_i \cdots \\
& \cdots \int x_{j1}x_{j2} g(\mathbf{x}_j)\, d\mathbf{x}_j + \int x_{i2}^2 g(\mathbf{x}_i)\, d\mathbf{x}_i \int x_{j2}^2 g(\mathbf{x}_j)\, d\mathbf{x}_j \\
= & \Big(\int x_{i1}^2 g(\mathbf{x}_i)\, d\mathbf{x}_i \Big)^2 + 2 \Big(\int x_{i1}x_{i2} g(\mathbf{x}_i)\, d\mathbf{x}_i \Big)^2 + \Big(\int x_{i2}^2 g(\mathbf{x}_i)\, d\mathbf{x}_i \Big)^2 \\
\geq & \ 0
\end{aligned} \tag{1.26}$$

In many specific cases it is not as easy to check Mercer's condition, since equation (1.23) must hold for every g with finite L_2 norm. Mercer's condition does give information on whether some kernel computes a dot product in some feature space, but it does not tell what the mapping Φ or the space $\mathcal{F}$ is.

When a kernel function does not comply with Mercer's condition, training data may exist that give rise to an indefinite Hessian matrix in the dual Lagrangian (1.11). The objective function can become arbitrarily large, and the quadratic programming problem will have no solution. Many training sets can still result in a positive semi-definite Hessian, and a SVM's constrained objective function can be maximised. The results, however, will not have the usual geometric interpretation of support vectors.

By definition of the kernel function $k(\mathbf{x}_i, \mathbf{x}_j) = \Phi(\mathbf{x}_i) \cdot \Phi(\mathbf{x}_j)$, the SVM decision function becomes

$$\begin{aligned}
f(\mathbf{x}) &= \operatorname{sign}\Big(\sum_{i=1}^{l} y_i \alpha_i \Phi(\mathbf{x}) \cdot \Phi(\mathbf{x}_i) + b \Big) \\
&= \operatorname{sign}\Big(\sum_{i=1}^{l} y_i \alpha_i k(\mathbf{x}, \mathbf{x}_i) + b \Big)
\end{aligned} \tag{1.27}$$

The architecture of the above decision function defines the architecture of the SVM, as shown in Figure 1.5. Examples of kernel functions most commonly used in pattern recognition problems are:

$$k(\mathbf{x}_i, \mathbf{x}_j) = (\mathbf{x}_i \cdot \mathbf{x}_j + 1)^p \tag{1.28}$$

$$k(\mathbf{x}_i, \mathbf{x}_j) = e^{-||\mathbf{x}_i - \mathbf{x}_j||^2/2\sigma^2} \tag{1.29}$$

$$k(\mathbf{x}_i, \mathbf{x}_j) = \tanh(\kappa \mathbf{x}_i \cdot \mathbf{x}_j - \delta) \tag{1.30}$$

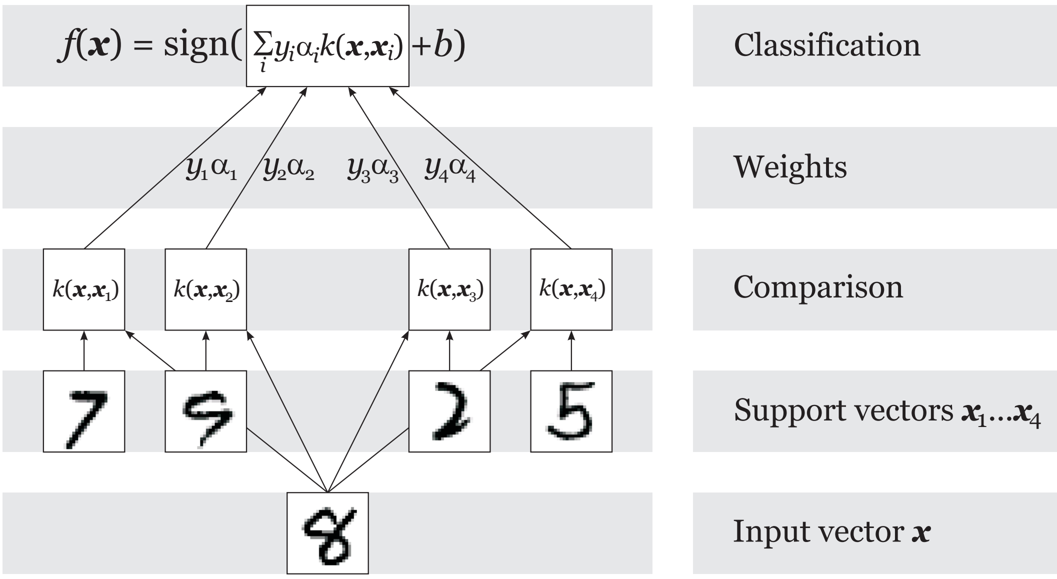
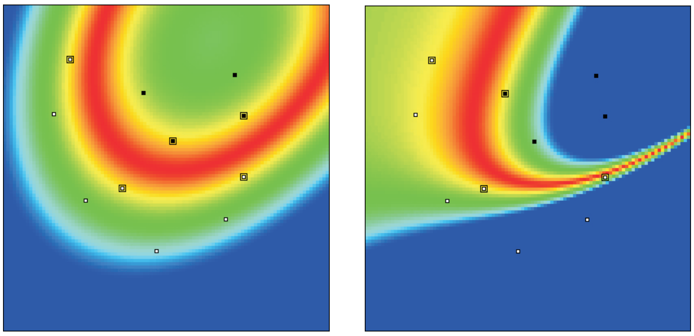


FIGURE 1.5: Architecture of a Support Vector Machine: The input vector $\mathbf{x}$ and the support vectors $\mathbf{x}_i$ (in this example optical digits) are non-linearly mapped (by Φ) into a feature space $\mathcal{F}$, where dot products between their mapped representations are computed. By the use of the kernel k, these two steps are in practice combined. The results are linearly combined by weights α_i found by solving a quadratic program. The linear combination is then fed into a decision function f, which determines the classification of $\mathbf{x}$.



(a) A Gaussian kernel $e^{-||\mathbf{x}_i - \mathbf{x}_j||^2}$.

(b) A polynomial kernel $(\mathbf{x}_i \cdot \mathbf{x}_j + 1)^5$.

FIGURE 1.6: Classifying with different kernel functions. The support vectors, with nonzero α_i, are shown with a double outline, and define the decision boundaries between the two classes.

Equation (1.28) results in a classifier that is a polynomial of degree p. Equation (1.29) results in a Gaussian radial basis function classifier, while (1.30) gives a particular kind of two-layer sigmoidal neural network. Figures 1.6(a) and 1.6(b) show the decision boundaries arising from both Gaussian radial basis function and polynomial kernels. More sophisticated kernels, like kernels generating splines or Fourier expansions, can be found in [43, 50, 57].

1.6 Concluding

This chapter presented the SVM optimisation problem: In training a non-linear SVM, the following quadratic problem needs to be maximised:

$$W(\boldsymbol{\alpha}) = \sum_{i=1}^{l} \alpha_i - \frac{1}{2} \sum_{i=1}^{l} \sum_{j=1}^{l} \alpha_i \alpha_j y_i y_j k(\mathbf{x}_i, \mathbf{x}_j) \tag{1.31}$$

subject to

$$0 \leq \alpha_i \leq 0, \quad i = 1, \ldots, l, \qquad \text{and} \qquad \sum_{i=1}^{l} \alpha_i y_i = 0 \tag{1.32}$$

By constructing a matrix Q such that $(Q)_{ij} = y_i y_j k(\mathbf{x}_i, \mathbf{x}_j)$ the problem at hand is to find

$$\begin{aligned} \max_{\boldsymbol{\alpha}} W(\boldsymbol{\alpha}) &= \boldsymbol{\alpha}^T \mathbf{1} - \frac{1}{2} \boldsymbol{\alpha}^T Q \boldsymbol{\alpha} \\ \text{subject to} \quad \boldsymbol{\alpha}^T \mathbf{y} &= 0 \\ \boldsymbol{\alpha} &\geq \mathbf{0} \\ C\mathbf{1} - \boldsymbol{\alpha} &\geq \mathbf{0} \end{aligned} \tag{1.33}$$

The following chapter is devoted to solving the above linearly constrained quadratic programming problem (1.33). The problem often involves a matrix with an extremely large number of entries, which make off-the-shelf optimisation packages unsuitable. Several SVM training methods are presented, and a detailed decomposition method of solving (1.33) is discussed.

Chapter 2

Support Vector Machine Training Methods

An overview of current methods of Support Vector Machine training is given in this chapter. The method of decomposing the training problem into subproblems is discussed in detail, and includes conditions for optimality of the training problem, methods for selecting good subproblems, and different optimisations to the decomposition algorithm itself. The chapter concludes with a complete Support Vector Machine training algorithm.

2.1 Introduction to Support Vector Machine training methods

Training a Support Vector Machine (SVM) involves solving a linearly constrained quadratic optimisation problem. The SVM fits a decision function to a labelled set of l training patterns, which correspond to the total of l free parameters in the optimisation problem. The quadratic programming (QP) problem, from chapter one, is to find

$$\begin{aligned} \max_{\boldsymbol{\alpha}} \; W(\boldsymbol{\alpha}) &= \boldsymbol{\alpha}^T \mathbf{1} - \frac{1}{2}\boldsymbol{\alpha}^T Q \boldsymbol{\alpha} \\ \text{subject to} \quad \boldsymbol{\alpha}^T \mathbf{y} &= 0 \\ \boldsymbol{\alpha} &\geq \mathbf{0} \\ C\mathbf{1} - \boldsymbol{\alpha} &\geq \mathbf{0} \end{aligned} \tag{2.1}$$

In the QP problem, the objective function – the function to be maximised – depends on the α_i quadratically, while the parameters α_i only appear linearly in the constraints. Q is an l by l matrix that depends on both a kernel function of the training inputs, and their respective labels: $(Q)_{ij} = y_i y_j k(\mathbf{x}_i, \mathbf{x}_j)$.

The QP problem is equivalent to finding the maximum of a bowl-shaped objective function. The search for the maximum occurs in l dimensions, and is constrained to lie inside a hypercube and on a hyperplane. Due to the definition of the kernel function, the matrix Q always gives a convex QP problem. The convexity of the optimisation problem implies that every local maximum is also a global maximum [19]. A global maximum means that there is no other point inside the feasible region at which the objective function takes a higher value. When Q is positive definite, the objective function will be bowl-shaped; when Q is positive semi-definite, the objective function will have flat-bottomed troughs. The objective function will never be saddle-shaped. Thus there exists a unique maximum or a connected set of maximums. Certain optimality conditions – the Karush-Kuhn-Tucker (KKT) conditions [19] – give conditions determining whether the constrained maximum has been found.

The SVM QP problem is simple and well understood; yet solving the QP problem for real-world cases can prove to be very difficult. Analytic solutions are possible when the number of training patterns is very small, or when the data is separable and it is known beforehand which vectors will be support vectors. In most real-world cases, numeric solutions are called for. Small problems can be solved with general-purpose optimisation packages that solve linearly constrained convex QPs. Larger problems, however, bring about difficulties in both the size and density of Q.

The matrix Q has a dimension equal to the number of training examples. A training set of 60,000 vectors gives rise to a matrix Q with 3.6 billion elements, which does not fit into the memory of a standard computer. For large learning tasks, off-the-shelf optimisation packages and techniques for general quadratic programming quickly become intractable in their memory and time requirements.

In general $(Q)_{ij}$ is nonzero, which makes Q completely dense. Most mathematical approaches either assume that Q is sparse (i.e. most $(Q)_{ij}$ are zero), or are only suitable for small problems.

Since standard QP techniques cannot easily be used to train SVMs with several thousands of examples, a number of other approaches have been invented. These algorithms allow for fast convergence and small memory requirements, even on large problems.

2.1.1 Chunking

The chunking algorithm is based on the fact that the non-support vectors play no role in the SVM decision boundary. If they are removed from the training set of examples, the SVM solution will be exactly the same.

Chunking was first suggested by V. Vapnik in [55]. The large QP problem is broken down into a number of smaller problems:

A QP routine is used to optimise the Lagrangian on an arbitrary subset of data. After this optimisation, the set of nonzero α_i (the current support vectors) are retained, and all other data points ($\alpha_i = 0$) are discarded. At every subsequent step, chunking solves the QP problem that consists of all nonzero α_i, plus some of the α_i that violates the KKT conditions. These are in general the worst M violations, for some value of M. After optimising the subproblem, data points with $\alpha_i = 0$ are again discarded. This procedure is iterated until the KKT conditions are met, and the margin is maximised. Solving each subproblem still requires a numeric quadratic optimiser.

The size of the subproblem varies, but tends to grow with time. At the last step, chunking has identified and optimised all the nonzero α_i, which correspond to the set of all the support vectors. Thus the overall QP problem is solved.

Although this technique of reducing the Q matrix's dimension from the number of training examples to approximately the number of support vectors makes it suitable to large problems, a limitation still exists. The number of support vectors may exceed the maximal number of parameters α_i that the quadratic optimiser can handle, and even the reduced matrix may not fit into memory.

2.1.2 Decomposition

Decomposition methods solve a sequence of smaller QP problems, and are similar in spirit to chunking. The difference from chunking is in the size of the subproblems: the size remains fixed.

Decomposition methods were introduced in 1997 by E. Osuna *et al.* [40]. The large QP problem is broken down into a series of smaller subproblems, and a numeric QP optimiser solves each of these problems. It was suggested that one vector be added and one removed from the subproblem at each iteration, and that the size of the subproblems should be kept fixed. The motivation behind this method is based on the observation that as long as at least one α_i violating the KKT conditions is added to the previous subproblem, each step reduces the objective function and maintains all of the constraints. In this fashion the sequence of QP subproblems will asymptotically converge. For faster practical convergence, researchers use different unpublished heuristics to add and delete multiple examples.

While the strategy used in chunking takes advantage of the fact that the expected number of support vectors is small (< 3000), decomposition allows for training arbitrarily large data sets.

Another decomposition method was introduced by T. Joachims in [24]. Joachim's method is based on the gradient of the objective function. The idea is to pick α_i for the QP subproblem such that the α_i form the steepest possible direction of ascent on the objective function, where the number of nonzero elements in the direction is equal to the size of the QP subproblem. As in Osuna's method, the size of the subproblem remains fixed.

2.1.3 Sequential Minimal Optimisation

The most extreme case of decomposition is Sequential Minimal Optimisation (SMO) – where the smallest possible optimisation problem is solved at each step [41]. Due to the fact that the α_i must obey the linear equality constraint, the smallest set of α_i that can be optimised at each step is two. At every step, SMO chooses two α_i to jointly optimise, finds the optimal values for these α_i, and updates the SVM to reflect these changes.

SMO avoids numerical QP optimisation and large matrix storage entirely: if the two chosen α_i are optimised and the rest of the parameters α_i kept fixed, it derives an analytic solution which is executed in a few numerical operations. The method therefore consists of a heuristic step for finding the best pair of parameters to optimise, and the use of an analytic expression to ensure the objective function increases monotonically. Because the smallest possible subproblem is optimised at each iteration of the algorithm, SMO solves more subproblems than other methods of decomposition. Optimising each subproblem, however, is so fast that the overall QP problem can be solved quickly. Due to the decomposition of the QP problem and its speed, SMO is probably the method of choice for training SVMs [10].

In this chapter a decomposition algorithm based on the ideas of [24] is discussed. This algorithm makes no assumption on the expected number of support vectors, and allows training arbitrary large data sets. In constructing the algorithm, conditions for optimality, decomposition and optimality conditions on the working set are discussed. Finally, a complete training algorithm is presented.

2.2 Conditions for optimality

In this section, conditions for optimality of a solution $\boldsymbol{\alpha}$ to problem (2.1) are introduced. Since Q is a positive semi-definite matrix (the kernel function used is positive definite), and the constraints are linear, the Karush-Kuhn-Tucker (KKT) conditions [19] are necessary and sufficient for optimality.

The KKT multipliers are introduced by letting μ be the associated multiplier of $\boldsymbol{\alpha}^T\mathbf{y} = 0$, $\boldsymbol{\pi}^T = (\pi_1, \ldots, \pi_l)$ be the associated multiplier of $-\boldsymbol{\alpha} \leq \mathbf{0}$, and $\boldsymbol{\upsilon}^T = (\upsilon_1, \cdots, \upsilon_l)$ be the associated multiplier of $\boldsymbol{\alpha} - C\mathbf{1} \leq \mathbf{0}$. The following KKT conditions must then hold for optimality:

$$
\begin{aligned}
\nabla W(\boldsymbol{\alpha}) - \nabla\boldsymbol{\upsilon}^T(\boldsymbol{\alpha} - C\mathbf{1}) - \nabla\boldsymbol{\pi}^T(-\boldsymbol{\alpha}) - \nabla\mu(\boldsymbol{\alpha}^T\mathbf{y}) &= \mathbf{0} \\
\Rightarrow \nabla W(\boldsymbol{\alpha}) - \boldsymbol{\upsilon} + \boldsymbol{\pi} - \mu\mathbf{y} &= \mathbf{0} \qquad (2.2) \\
\boldsymbol{\upsilon}^T(\boldsymbol{\alpha} - C\mathbf{1}) &= 0 \qquad (2.3) \\
\boldsymbol{\pi}^T\boldsymbol{\alpha} &= 0 \qquad (2.4) \\
\boldsymbol{\upsilon} &\geq \mathbf{0} \qquad (2.5)
\end{aligned}
$$

$$\boldsymbol{\pi} \geq \mathbf{0} \tag{2.6}$$

The Lagrange multipliers α_i can have three possible values: The value of α_i can be at zero, at the upper bound C, or somewhere in the interval $(0, C)$. By defining the classifier function

$$f^\star(\mathbf{x}) = \sum_{i=1}^{l} y_i \alpha_i k(\mathbf{x}, \mathbf{x}_i) + b \tag{2.7}$$

similar to (1.27), each of these cases are now considered and expanded separately.

Case 1: $0 < \alpha_i < C$

Consider a single value of α_i, i.e. the Lagrange multiplier associated with some input vector i. Then, from equation (2.2),

$$1 - (Q\boldsymbol{\alpha})_i - \upsilon_i + \pi_i - \mu y_i = 0$$

Since this case examines α_i from the interval $(0, C)$, the term $(\boldsymbol{\alpha} - C\mathbf{1})_i$ from (2.3) must be non-zero and negative. For equations (2.3) and (2.5) to hold, υ_i must be equal to zero.

By using a similar argument, conditions (2.4) and (2.6) imply that π_i can only be zero. This gives

$$1 - (Q\boldsymbol{\alpha})_i - \mu y_i = 0 \tag{2.8}$$

Because the equation

$$y_i f^\star(\mathbf{x}_i) = y_i \Big(\sum_{j=1}^{l} y_j \alpha_j k(\mathbf{x}_i, \mathbf{x}_j) + b \Big) = 1 \tag{2.9}$$

holds when $0 < \alpha_i < C$, and given that

$$\begin{aligned} (Q\boldsymbol{\alpha})_i &= \sum_{j=1}^{l} y_i y_j \alpha_j k(\mathbf{x}_i, \mathbf{x}_j) \\ &= y_i \sum_{j=1}^{l} y_j \alpha_j k(\mathbf{x}_i, \mathbf{x}_j) \\ &= y_i \big(f^\star(\mathbf{x}_i) - b \big) \end{aligned}$$

equation (2.8) can be rewritten, and simpifies as

$$\begin{aligned} 1 - (Q\boldsymbol{\alpha})_i - \mu y_i &= 1 - y_i \big(f^\star(\mathbf{x}_i) - b \big) - \mu y_i \\ &= 1 - 1 + y_i b - \mu y_i = 0 \end{aligned}$$

From this the value of b is equal to the KKT multiplier μ, i.e.

$$\mu = b \tag{2.10}$$

Case 2: $\alpha_i = C$

As in the previous case, consider equation (2.2) for a single Lagrange multiplier α_i at the upper bound C:

$$1 - (Q\boldsymbol{\alpha})_i - \upsilon_i + \pi_i - \mu y_i = 0$$

Because $\alpha_i = C$, conditions (2.4) and (2.6) imply that π_i must be equal to zero. Then,

$$1 - (Q\boldsymbol{\alpha})_i - \upsilon_i - \mu y_i = 0 \tag{2.11}$$

Equation (2.5) specifies that $\upsilon_i \geq 0$, and thus

$$\begin{aligned} 1 - (Q\boldsymbol{\alpha})_i - \mu y_i &\geq 0 \\ 1 - y_i\big(f^\star(\mathbf{x}_i) - b\big) - by_i = 1 - y_i f^\star(\mathbf{x}_i) &\geq 0 \end{aligned}$$

Thus for a value of $\alpha_i = C$ to meet the KKT conditions, it must be true that

$$y_i f^\star(\mathbf{x}_i) \leq 1 \tag{2.12}$$

Case 3: $\alpha_i = 0$

In the case of $\alpha_i = 0$, equation (2.2) becomes

$$1 - (Q\boldsymbol{\alpha})_i - \upsilon_i + \pi_i - \mu y_i = 0$$

Conditions (2.3) and (2.5), with $\alpha_i = 0$, imply that $\upsilon_i = 0$. Therefore,

$$1 - (Q\boldsymbol{\alpha})_i + \pi - \mu y_i = 0 \tag{2.13}$$

Using similar reasoning as the above case of $\alpha_i = C$, it can be shown that a value of $\alpha_i = 0$ meets the KKT conditions if

$$y_i f^\star(\mathbf{x}_i) \geq 1 \tag{2.14}$$

Concluding on the KKT conditions

From the three cases presented above, a solution $\boldsymbol{\alpha}$ of problem (2.1) is an optimal solution if the following relations hold for each α_i:

$$\begin{aligned} \alpha_i = 0 &\;\Rightarrow\; y_i f^\star(\mathbf{x}_i) \geq 1 \\ 0 < \alpha_i < C &\;\Rightarrow\; y_i f^\star(\mathbf{x}_i) = 1 \\ \alpha_i = C &\;\Rightarrow\; y_i f^\star(\mathbf{x}_i) \leq 1 \end{aligned} \tag{2.15}$$

If, for some given stage in the process of training a SVM, all Lagrange multipliers meet the KKT conditions, an optimal solution to (2.1) is found and SVM training can stop.

Computing the value of the threshold b

A value for the threshold b is needed for (2.7), and can be computed for each of the support vectors. From (2.9),

$$b_i = y_i - \sum_{j=1}^{l} y_j \alpha_j k(\mathbf{x}_i, \mathbf{x}_j) \tag{2.16}$$

The average of these values is taken as the value for b.

2.3 A decomposition method

Decomposition methods break the large QP problem down to a series of smaller subproblems, and these subproblems are optimised to improve the objective function.

In the process of decomposition, a subset of variables is chosen for optimisation. The original set of Lagrange multiplier variables is divided into two sets, called B and N. Set B is called the "working set," and is created by picking q sub-optimal variables from all l α_i. The working set of variables is optimised while keeping the remaining variables (set N) constant. After subset B is optimised, it is "put back" into the original set and a new working set is selected for optimisation.

Since it is known when a solution $\boldsymbol{\alpha}$ is an optimal solution (the solution satisfies all KKT conditions), the problem can be decomposed and optimised until these conditions are met with an adequate tolerance. The general decomposition algorithm is summarized as follows:

Algorithm 2.1 - General decomposition algorithm

1. While the optimality conditions (2.15) are violated
   (a) Select q variables for the working set B. The remaining $l - q$ variables are fixed at their current values.
   (b) Decompose the problem and solve the quadratic program subproblem, i.e. optimise $W(\boldsymbol{\alpha})$ on B.
2. Terminate and return $\boldsymbol{\alpha}$.

Concerns of the above algorithm are the creation of KKT criteria for knowing when the working set B is optimised, and methods of picking the optimal working set.

Firstly, however, it is necessary to rewrite equation (2.1) as a function that is only dependent on the working set. Let $\boldsymbol{\alpha}$ be split into two sets $\boldsymbol{\alpha}_B$ and $\boldsymbol{\alpha}_N$. If $\boldsymbol{\alpha}$, $\mathbf{y}$ and Q are appropriately rearranged, one has

$$\boldsymbol{\alpha} = \begin{bmatrix} \boldsymbol{\alpha}_B \\ \boldsymbol{\alpha}_N \end{bmatrix}, \qquad \mathbf{y} = \begin{bmatrix} \mathbf{y}_B \\ \mathbf{y}_N \end{bmatrix}, \qquad Q = \begin{bmatrix} Q_{BB} & Q_{BN} \\ Q_{NB} & Q_{NN} \end{bmatrix}$$

Since only $\boldsymbol{\alpha}_B$ is being optimised for the subproblem, W is rewritten from equation (2.1) in terms of $\boldsymbol{\alpha}_B$ to give

$$\begin{aligned} W(\boldsymbol{\alpha}_B) = {} & \left(\boldsymbol{\alpha}_B^T \mathbf{1} + \boldsymbol{\alpha}_N^T \mathbf{1}\right) - \frac{1}{2}\Big(\boldsymbol{\alpha}_B^T Q_{BB} \boldsymbol{\alpha}_B + \boldsymbol{\alpha}_B^T Q_{BN} \boldsymbol{\alpha}_N \\ & + \boldsymbol{\alpha}_N^T Q_{NB} \boldsymbol{\alpha}_B + \boldsymbol{\alpha}_N^T Q_{NN} \boldsymbol{\alpha}_N\Big) \end{aligned} \tag{2.17}$$

If terms that do not contain $\boldsymbol{\alpha}_B$ are dropped, the optimisation problem remains essentially the same. Also, since Q is a symmetric matrix, with $Q_{BN} = Q_{NB}^T$, the problem reduces to finding

$$\begin{aligned} \max_{\boldsymbol{\alpha}_B} \; W(\boldsymbol{\alpha}_B) &= \boldsymbol{\alpha}_B^T \mathbf{1} - \frac{1}{2}\boldsymbol{\alpha}_B^T Q_{BB} \boldsymbol{\alpha}_B - \boldsymbol{\alpha}_B^T Q_{BN} \boldsymbol{\alpha}_N \\ \text{subject to} \qquad \boldsymbol{\alpha}_B^T \mathbf{y}_B + \boldsymbol{\alpha}_N^T \mathbf{y}_N &= 0 \\ \boldsymbol{\alpha}_B &\geq \mathbf{0} \\ C\mathbf{1} - \boldsymbol{\alpha}_B &\geq \mathbf{0} \end{aligned} \tag{2.18}$$

With $|B| \ll |N|$, the term $\boldsymbol{\alpha}_B^T Q_{BN} \boldsymbol{\alpha}_N$ consumes the majority of computing time when determining $W(\boldsymbol{\alpha}_B)$. As a performance optimisation, define a vector $\mathbf{q}_{BN} = Q_{BN}\boldsymbol{\alpha}_N$ in the following way:

$$(q_{BN})_i = y_i \sum_{j \in N} \alpha_j y_j k(\mathbf{x}_i, \mathbf{x}_j) \tag{2.19}$$

The vector $\mathbf{q}_{BN}$ is computed once at the start of every subset optimisation. The complexity of the optimisation problem then becomes proportional to the size of the working set, independent of l. Given that l can be very large and that $q = |B|$ will be relatively small, it is a vast improvement. The optimisation problem becomes equivalent to finding

$$\begin{aligned} \max_{\boldsymbol{\alpha}_B} \; W(\boldsymbol{\alpha}_B) &= \boldsymbol{\alpha}_B^T \mathbf{1} - \frac{1}{2}\boldsymbol{\alpha}_B^T Q_{BB} \boldsymbol{\alpha}_B - \boldsymbol{\alpha}_B^T \mathbf{q}_{BN} \\ \text{subject to} \qquad \boldsymbol{\alpha}_B^T \mathbf{y}_B + \boldsymbol{\alpha}_N^T \mathbf{y}_N &= 0 \\ \boldsymbol{\alpha}_B &\geq \mathbf{0} \\ C\mathbf{1} - \boldsymbol{\alpha}_B &\geq \mathbf{0} \end{aligned} \tag{2.20}$$

2.3.1 Optimality of the working set

The optimisation problem in (2.20) has one particularly useful property: one can computationally determine if a solution is an optimal solution. This gives a stopping criterion for optimising the working set B.

The decomposed problem (2.20) consists of a convex objective function (since matrix Q_{BB} is positive semi-define), and linear constraints. The KKT conditions are thus necessary and sufficient for optimality.

The KKT conditions must hold for each element in $\boldsymbol{\alpha}_B$, and by again considering the possible values of $(\alpha_B)_i$, as in Section 2.2, the conditions are:

$$\begin{aligned}(\alpha_B)_i = 0 &\Rightarrow (Q_{BB}\boldsymbol{\alpha}_B)_i + (q_{BN})_i + \mu(y_B)_i \geq 1 \\ 0 < (\alpha_B)_i < C &\Rightarrow (Q_{BB}\boldsymbol{\alpha}_B)_i + (q_{BN})_i + \mu(y_B)_i = 1 \\ (\alpha_B)_i = C &\Rightarrow (Q_{BB}\boldsymbol{\alpha}_B)_i + (q_{BN})_i + \mu(y_B)_i \leq 1 \end{aligned} \tag{2.21}$$

When the Lagrange multiplier α_i lies between zero and C, the value of μ can be computed with

$$\mu = (y_B)_i\big(1 - (Q_{BB}\boldsymbol{\alpha}_B)_i - (q_{BN})_i\big)$$

The value of μ, as it appears in the above KKT conditions (2.21), can be taken as the average of μ computed for each i where $0 < (\alpha_B)_i < C$.

Apart from the optimality conditions described here, a method for selecting good or optimal working sets – a decomposition algorithm – is needed. Such a method will choose the working set B, while the KKT conditions presented here determines the termination criteria on optimising B.

2.3.2 Selecting the working set

One of the most important issues in a decomposition algorithm is the selection of the working set. The working set selected plays a major role in the speed of the SVM training algorithm. Selecting working sets at random causes the training algorithm (Algorithm 2.1) to converge very slowly, while continually selecting optimal variables causes the training algorithm to cycle. A method for selecting approximately optimal working sets is presented below.

The decomposition method presented in this section is due to [24, 38]. It works on the classical method of feasible directions, proposed in the optimisation theory by [61]. If Ω is a feasible region of a general constrained problem, then a vector $\mathbf{d}$ is a feasible direction at the point $\boldsymbol{\alpha}$ in Ω, if there exists a $\tilde{\lambda}$ such that $\boldsymbol{\alpha} + \lambda\mathbf{d}$ lies in Ω for all $0 \leq \lambda \leq \tilde{\lambda}$.

The main idea of the method of feasible directions is to start with an initial feasible solution, and to find the optimal solution by making steps along feasible directions. At each iteration of a feasible directions algorithm, the optimal feasible direction (the direction giving the largest rate of increase of the objective function) is found. The algorithm then aims to maximise the objective function along this direction, by making a line search to determine a step length along the feasible direction. The solution is moved by "stepping" along the feasible direction to the better solution found. The algorithm terminates when no feasible directions can be found which improve the objective function.

The optimal feasible direction of a general constrained optimisation problem of the form

$$\text{Maximise } f(\boldsymbol{\alpha}) \qquad \text{subject to } A\boldsymbol{\alpha} \leq \mathbf{b}$$

is found by solving the direction finding linear program

$$\text{Maximise } \nabla f^T \mathbf{d} \qquad \text{subject to } A\mathbf{d} \leq \mathbf{0}, \;\; ||\mathbf{d}||_2 \leq 1$$

SVM training solves a constrained quadratic optimisation problem, therefore the method of feasible directions is directly applicable to training a SVM. Finding the optimal feasible direction when solving the SVM problem (2.1) can be stated as

$$\begin{aligned} \text{Maximise} \quad & \nabla W(\boldsymbol{\alpha})^T \mathbf{d} \\ \text{subject to} \quad & \mathbf{y}^T \mathbf{d} = 0 \\ & d_i \geq 0 \quad \text{if } \alpha_i = 0 \\ & d_i \leq 0 \quad \text{if } \alpha_i = C \\ & ||\mathbf{d}||_2 \leq 1 \end{aligned} \tag{2.22}$$

Optimisation problem (2.22) is a full-scale linear program of dimension l, which is computationally expensive to solve at every iteration of the decomposition method of SVM training. An approximate solution to this problem, which can be obtained in linear time, was proposed by T. Joachims [24].

A requirement is added to (2.22), specifying that only q components of $\mathbf{d}$ be non-zero. The variables corresponding to these q non-zero components are included in the working set. Since this only gives an approximation to (2.22), $\mathbf{d}$ is only used to identify B, and not as a search direction. Instead of doing a line search on $\mathbf{d}$, the optimum solution is found in the *entire* subspace spanned by the non-zero components of $\mathbf{d}$.

By specifying that only q components of $\mathbf{d}$ be non-zero, the problem becomes intractable. This problem of intractability is overcome by letting d_i be equal to either -1, 0 or $+1$, such that the Lagrange multipliers α_i corresponding to $d_i = \pm 1$ are included in B. An approximation of (2.22) is thus found by

$$\begin{aligned} \text{Maximise} \quad & \nabla W(\boldsymbol{\alpha})^T \mathbf{d} \\ \text{subject to} \quad & \mathbf{y}^T \mathbf{d} = 0 \\ & d_i \geq 0 \qquad \text{if } \alpha_i = 0 \\ & d_i \leq 0 \qquad \text{if } \alpha_i = C \\ & d_i \in \{-1, 0, 1\} \\ & |\{d_i : d_i \neq 0\}| = q \end{aligned} \tag{2.23}$$

$\mathbf{g}^T =$	-1	-2	+3	+4	-4	+5	+1	0	-2	+5
$\mathbf{y}^T =$	+1	+1	-1	-1	+1	-1	+1	+1	-1	+1
$\mathbf{d}^T =$	0	0	0	+1	-1	+1	0	0	0	+1
$y_i d_i =$	0	0	0	-1	-1	-1	0	0	0	+1

(a) Selecting the four largest values of $|g_i|$, and setting each corresponding d_i to the sign of g_i, maximises $\mathbf{g}^T\mathbf{d}$, but the equality constraint $\mathbf{y}^T\mathbf{d} = 0$ is not met.

$\mathbf{g}^T =$	-1	-2	+3	+4	-4	+5	+1	0	-2	+5
$\mathbf{y}^T =$	+1	+1	-1	-1	+1	-1	+1	+1	-1	+1
$y_i g_i =$	-1	-2	-3	-4	-4	-5	+1	0	+2	+5
$\mathbf{d}^T =$	0	0	0	+1	0	+1	0	0	-1	+1
$y_i d_i =$	0	0	0	-1	0	-1	0	0	+1	+1

(b) Selecting the two smallest and largest $y_i g_i$, and respectively letting d_i be of opposite and similar sign to y_i, $\mathbf{g}^T\mathbf{d}$ is maximised such that the equality constraint $\mathbf{y}^T\mathbf{d} = 0$ is also met.

FIGURE 2.1: Selecting a working set of size four.

From this approximation the question arises: how is the direction $\mathbf{d}$ determined? Firstly, assume that the constraints $\mathbf{y}^T\mathbf{d} = 0$, $d_i \geq 0$ if $\alpha_i = 0$, and $d_i \leq 0$ if $\alpha_i = C$, are all absent. Also, to simplify the notation used, let the shorthand $\mathbf{g} = \nabla W(\boldsymbol{\alpha})$ denote the directional derivative of W. With the equality and inequality constraints absent, the maximum of the objective function is achieved by selecting q points with the highest values of $|g_i|$. Then d_i will take the value of $\text{sign}(g_i)$.

As an example, consider Figure 2.1(a), with q equal to four. The four largest values of $|g_i|$ are chosen ($|g_4| = 4$, $|g_5| = 4$, $|g_6| = 5$ and $|g_{10}| = 5$), and each corresponding d_i is set to the sign of g_i. In this way $\mathbf{g}^T\mathbf{d}$ is maximised.

The first remark that can be made about the example in Figure 2.1(a), is that the equality constraint $\mathbf{y}^T\mathbf{d} = 0$ is being violated. For $\mathbf{y}^T\mathbf{d}$ to be equal to zero, the number of elements with sign matches between d_i and y_i must be equal to the number of elements with sign mismatches between d_i and y_i. This means that if a working set of size q is selected, with q being even, each number must be equal to $\frac{q}{2}$. The working set can thus be selected by making two passes over the data. A "forward pass" will select $\frac{q}{2}$ sign mismatches, while a "backward pass" will select $\frac{q}{2}$ sign matches. To implement selection of the working set, let γ_k denote the largest contribution to the objective function $\mathbf{g}^T\mathbf{d}$ by some point k, subject to the equality constraint $\mathbf{y}^T\mathbf{d} = 0$. The two passes over the data, each selecting $\frac{q}{2}$ variables, are expanded in the following way:

"Forward pass"

The forward pass attempts to select $\frac{q}{2}$ variables such that $y_k d_k$ is negative. This implies that the signs of y_k and d_k must be different in maximising $\mathbf{g}^T\mathbf{d}$. To maximise $\mathbf{g}^T\mathbf{d}$, the minimum g_i is chosen when d_i is negative, while the maximum g_i is selected when d_i is positive, i.e.

$$\begin{array}{llll} y_k = 1 & \Rightarrow d_k = -1 & \Rightarrow \gamma_k = \min_{i:y_i=1}(g_i) & \Rightarrow \gamma_k = \min_{i:y_i=1}(y_i g_i) \\ y_k = -1 & \Rightarrow d_k = 1 & \Rightarrow \gamma_k = \max_{i:y_i=-1}(g_i) & \Rightarrow \gamma_k = \min_{i:y_i=-1}(y_i g_i) \end{array}$$

If the subscripts are combined, the largest contribution to the objective function (with y_k and d_k having different signs), subject to the equality constraint, is

$$\gamma_k = \min_i (y_i g_i) \tag{2.24}$$

"Backward pass"

The backward pass over the data selects a total of $\frac{q}{2}$ variables, such that $y_k d_k$ is positive. Thus the signs of y_k and d_k must be the same in maximising $\mathbf{g}^T\mathbf{d}$, i.e.

$$\begin{array}{llll} y_k = 1 & \Rightarrow d_k = 1 & \Rightarrow \gamma_k = \max_{i:y_i=1}(g_i) & \Rightarrow \gamma_k = \max_{i:y_i=1}(y_i g_i) \\ y_k = -1 & \Rightarrow d_k = -1 & \Rightarrow \gamma_k = \min_{i:y_i=-1}(g_i) & \Rightarrow \gamma_k = \max_{i:y_i=-1}(y_i g_i) \end{array}$$

If the subscripts are combined, the largest contribution to the objective function (with y_k and d_k having the same signs), subject to the equality constraint, is

$$\gamma_k = \max_i (y_i g_i) \tag{2.25}$$

The working set is thus selected based on the equations (2.24, 2.25) defined above. The example of Figure 2.1(a) selected an optimal but useless working set, since it does not include the equality constraint.

In Figure 2.1(b) the two smallest and largest $y_i g_i$ ($y_4 g_4 = -4$, $y_6 g_6 = -5$, $y_9 g_9 = +2$ and $y_{10} g_{10} = +5$) are selected, such that the example correctly meets the equality constraint $\mathbf{y}^T\mathbf{d} = 0$.

It is clear that the quantity $y_i g_i$ gives an indication of an element's contribution to the objective function subject to the equality constraint. This quantity is used to select the working set, by *sorting* the data elements according to $y_i g_i$ and selecting the top and bottom $\frac{q}{2}$.

Accounting for the inequality constraints in (2.23) then becomes a trivial task – when selecting the top and bottom Lagrange multiplier variables α_i from the sorted list, a variable is skipped if the inequality constraints are violated. Thus variables are skipped if $d_i = -y_i$ (or in the case of the backward pass, if $d_i = y_i$) violates $d_i \geq 0$ if $\alpha_i = 0$, and $d_i \leq 0$ if $\alpha_i = C$. Consider the forward pass: if $d_i = -y_i$, then variables should be chosen when $-y_i \geq 0$ if $\alpha_i = 0$, and $-y_i \leq 0$ if $\alpha_i = C$. These conditions hold when $y_i = -1$ and $\alpha_i = 0$, or when $y_i = 1$ and $\alpha_i = C$. A similar argument on the backward pass states that variables should be chosen when $y_i = 1$ and $\alpha_i = 0$, or when $y_i = -1$ and $\alpha_i = C$.

The decomposition algorithm, which selects variables with a forward and backward pass over the data, is implemented below:

Algorithm 2.2 - Decomposition algorithm

1. Let L be a list of all Lagrange multipliers.
2. While the optimality conditions (2.15) are violated
   (a) sort L by $y_i g_i$ in increasing order
   (b) select $\frac{q}{2}$ samples from the front of L such that
      - $0 < \alpha_i < C$ or
      - $\big(y_i = -1$ and $\alpha_i = 0\big)$ or $\big(y_i = 1$ and $\alpha_i = C\big)$

   (c) select $\frac{q}{2}$ samples from the back of L such that
      - $0 < \alpha_i < C$ or
      - $\big(y_i = 1$ and $\alpha_i = 0\big)$ or $\big(y_i = -1$ and $\alpha_i = C\big)$

   (d) optimise the newly selected working set
3. Terminate and return $\boldsymbol{\alpha}$.

2.3.3 Shortcuts and optimisations to the decomposition algorithm

The speed of the decomposition algorithm is hampered by many redundant computations. This section discusses some of these performance bottlenecks, and ways minimise additional computations.

Let t define a certain iteration in Algorithm 2.2. At time t, a number of factors consume the algorithm's execution time: Its efficiency greatly depends on the amount of time taken to compute the vector $\mathbf{g} = \nabla W(\boldsymbol{\alpha}^{(t)})$ and matrices Q_{BB} and Q_{BN}. Its speed is also influenced by the time taken to compute the KKT conditions at each iteration, since it too requires the kernel matrix.

Due to the approach taken by the decomposition method, the quantities $\mathbf{g} = \nabla W(\boldsymbol{\alpha}^{(t)})$ (needed for selecting the working set) and $y_i f^\star(\mathbf{x}_i)$ (needed for KKT conditions), can be defined using knowledge of only q rows of the Hessian Q. These q rows correspond to the q elements in the current working set.

For this purpose, define a vector $\mathbf{s}^{(t)}$, that is computed directly after working set selection, and is stored throughout the training iteration:

$$s_i^{(t)} = \sum_{j=1}^{l} \alpha_j^{(t)} y_j k(\mathbf{x}_i, \mathbf{x}_j) = \sum_{j \in B} \alpha_j^{(t)} y_j k(\mathbf{x}_i, \mathbf{x}_j) + \sum_{j \in N} \alpha_j^{(t)} y_j k(\mathbf{x}_i, \mathbf{x}_j) \tag{2.26}$$

As $\boldsymbol{\alpha}^{(t)}$ is refined, the objective function $W(\boldsymbol{\alpha}^{(t)})$ is increased by each iteration of the decomposition method. The best vector $\boldsymbol{\alpha}^{(t)}$ found in iteration t is therefore used as the vector $\boldsymbol{\alpha}^{(t+1)}$, which the decomposition method uses to select a working set for iteration $t+1$. The vector $\boldsymbol{\alpha}^{(t+1)}$ is therefore the vector that maximises $W(\boldsymbol{\alpha}^{(t)})$ over the working set B from iteration t, i.e.

$$W(\boldsymbol{\alpha}^{(t+1)}) = \max_B \ W(\boldsymbol{\alpha}^{(t)}) \tag{2.27}$$

and

$$\begin{aligned} W(\boldsymbol{\alpha}^{(t)}) &= (\boldsymbol{\alpha}^{(t)})^T\mathbf{1} - \frac{1}{2}(\boldsymbol{\alpha}^{(t)})^T Q\boldsymbol{\alpha}^{(t)} \\ &= \sum_{i=1}^{l} \alpha_i^{(t)} - \frac{1}{2}\sum_{i=1}^{l}\sum_{j=1}^{l} \alpha_i^{(t)}\alpha_j^{(t)} y_i y_j k(\mathbf{x}_i, \mathbf{x}_j) \\ &= \sum_{i=1}^{l} \alpha_i^{(t)} - \frac{1}{2}\sum_{i=1}^{l} \alpha_i^{(t)} y_i \sum_{j=1}^{l} \alpha_j^{(t)} y_j k(\mathbf{x}_i, \mathbf{x}_j) \\ &= \sum_{i=1}^{l} \alpha_i^{(t)} - \frac{1}{2}\sum_{i=1}^{l} \alpha_i y_i s_i^{(t)} \end{aligned} \tag{2.28}$$

When a vector $\boldsymbol{\alpha}^{(t)}$ has been found that maximises $W(\boldsymbol{\alpha}^{(t)})$ over the working set B, the starting vector for the next iteration – which is also the best solution $\boldsymbol{\alpha}$ found thus far – is updated with $\boldsymbol{\alpha}^{(t+1)} \leftarrow \boldsymbol{\alpha}^{(t)}$. Because $\boldsymbol{\alpha}$ is updated, the value of $\mathbf{s}$ must also be updated. Since only the value of $\boldsymbol{\alpha}_B$, or the working set of variables, has changed from time t to time $t+1$, $\mathbf{s}$ is updated with

$$s_i^{(t+1)} = s_i^{(t)} + \sum_{j\in B} \big(\alpha_j^{(t+1)} - \alpha_j^{(t)}\big) y_j k(\mathbf{x}_i, \mathbf{x}_j) \tag{2.29}$$

Many optimisations can be implemented using definition (2.26) and simple update (2.29) of vector $\mathbf{s}$. At the start of training of a new working set, the value of $\mathbf{q}_{BN}$ from (2.19) is computed with

$$(q_{BN})_{i\in B}^{(t)} = y_i\Big(s_i^{(t)} - \sum_{j\in B} \alpha_j^{(t)} y_j k(\mathbf{x}_i, \mathbf{x}_j)\Big) \tag{2.30}$$

The derivative of W at time t (needed for selecting an optimal working set) is easily determined from $\mathbf{s}$, i.e.

$$\begin{aligned} \nabla W(\boldsymbol{\alpha}^{(t)})_i &= 1 - \frac{1}{2}\cdot 2y_i \sum_{j=1}^{l} \alpha_j^{(t)} y_j k(\mathbf{x}_i, \mathbf{x}_j) \\ &= 1 - y_i s_i^{(t)} \end{aligned} \tag{2.31}$$

By using $\mathbf{s}$, the value of the threshold b (2.16) is rewritten for each support vector as

$$\begin{aligned} b_i^{(t)} &= y_i - \sum_{j=1}^{l} y_j \alpha_j^{(t)} k(\mathbf{x}_i, \mathbf{x}_j) \\ &= y_i - s_i^{(t)} \end{aligned} \tag{2.32}$$

The value of $b^{(t)}$ is taken as the average over all the $b_i^{(t)}$ of all support vectors i.

Finally, the KKT optimality conditions specified in (2.15) are also rewritten in terms of $\mathbf{s}$, and are computed in linear time. A solution $\boldsymbol{\alpha}^{(t)}$ of (2.1) is an optimal solution if the following relations hold for each $\alpha_i^{(t)}$:

$$\begin{aligned} \alpha_i^{(t)} = 0 &\Rightarrow y_i(s_i^{(t)} + b^{(t)}) \geq 1 \\ 0 < \alpha_i^{(t)} < C &\Rightarrow y_i(s_i^{(t)} + b^{(t)}) = 1 \\ \alpha_i^{(t)} = C &\Rightarrow y_i(s_i^{(t)} + b^{(t)}) \leq 1 \end{aligned} \tag{2.33}$$

2.4 The training algorithm

Almost all necessary tools are now gathered to create a SVM training algorithm.

In this chapter the Karush-Kuhn-Tucker conditions have been used to specify whether and optimal solution has been found and the training algorithm can terminate. A method was developed to decompose the SVM problem into more workable subproblems. Optimisations to reduce the number of computations were also introduced.

Finally, the detailed training algorithm is presented:

Algorithm 2.3 - SVM training algorithm

1. Pick an initial vector $\boldsymbol{\alpha}^{(0)}$

2. Compute the initial value of $\mathbf{s}^{(0)}$:

$$s_i^{(0)} = \sum_{j=1}^{l} \alpha_j^{(0)} y_j k(\mathbf{x}_i, \mathbf{x}_j).$$

3. Compute the initial value of b with

$$b^{(0)} = \frac{1}{SVs} \sum_{i \in SVs} (y_i - s_i^{(0)}),$$

   where SVs is the total number of current support vectors.

4. Let L be a list of all l Lagrange multipliers α_i.

5. While the Karush-Kuhn-Tucker conditions in (2.33) are not met

(a) Let $\mathbf{g} \in \mathbb{R}^l$ be defined by

$$g_i = \nabla W(\boldsymbol{\alpha}^{(t)})_i = 1 - y_i s_i^{(t)}.$$

(b) Sort L by $y_i g_i$ in increasing order.

(c) Select $\frac{q}{2}$ samples from the front of L such that

- $0 < \alpha_i^{(t)} < C$ or
- $\big(y_i = -1 \text{ and } \alpha_i^{(t)} = 0\big)$ or $\big(y_i = 1 \text{ and } \alpha_i^{(t)} = C\big)$

(d) Select $\frac{q}{2}$ samples from the back of L such that

- $0 < \alpha_i^{(t)} < C$ or
- $\big(y_i = 1 \text{ and } \alpha_i^{(t)} = 0\big)$ or $\big(y_i = -1 \text{ and } \alpha_i^{(t)} = C\big)$

(e) After selection of the elements $\boldsymbol{\alpha}_B$ in the working set B, compute the Hessian matrix Q_{BB}.

(f) Determine the vector $\mathbf{q}_{BN}$ with

$$(q_{BN})_{i \in B}^{(t)} = y_i \Big(s_i^{(t)} - \sum_{j \in B} \alpha_j^{(t)} y_j k(\mathbf{x}_i, \mathbf{x}_j)\Big).$$

(g) Re-optimise the working set, using

$$W(\boldsymbol{\alpha}_B) = \boldsymbol{\alpha}_B^T \mathbf{1} - \frac{1}{2}\boldsymbol{\alpha}_B^T Q_{BB} \boldsymbol{\alpha}_B - \boldsymbol{\alpha}_B^T \mathbf{q}_{BN},$$

and constraints defined in (2.20). Replace the optimised $\boldsymbol{\alpha}_B$ into $\boldsymbol{\alpha}^{(t)}$ to get $\boldsymbol{\alpha}^{(t+1)}$.

(h) Update the vector $\mathbf{s}^{(t+1)}$ with

$$s_i^{(t+1)} = s_i^{(t)} + \sum_{j \in B} (\alpha_j^{(t+1)} - \alpha_j^{(t)}) y_j k(\mathbf{x}_i, \mathbf{x}_j).$$

(i) Recompute the value of b with

$$b^{(t+1)} = \frac{1}{SVs} \sum_{i \in SVs} (y_i - s_i^{(t+1)}).$$

(j) Increase time t with $t := t + 1$.

6. Terminate and return $\boldsymbol{\alpha}$.

There is one tool needed to complete the SVM training algorithm, and that is a routine to optimise the working set, i.e. a routine that can solve (2.20). The following chapter introduces Particle Swarm Optimisation (PSO) as a general optimisation method. Since (2.20) is a problem with linear and boxed constraints, PSO is adapted to handle linear equality and inequality constraints, and the working set can be optimised using PSO, and the SVM trained.

Chapter 3

Particle Swarm Optimisation

Particle Swarm Optimisation is discussed as an algorithm for optimising unconstrained problems. The chapter looks into standard topologies used in the algorithm, and touches on a number of improvements to Particle Swarm Optimisation.

3.1 Introduction to unconstrained optimisation

Numerical optimisation techniques have their application in many fields, including natural science, engineering, finance, medicine and telecommunications. The objective of such techniques is to assign values from a given domain to a set of parameters such that a specific function is optimised. The function that is minimised or maximised (optimised) is called the objective function, and it depends on a set of solution-defining variables. Let $\mathbf{x} = (x_1, x_2, \ldots x_n)^T \in \mathbb{R}^n$ represent the domain of the objective function, or the optimisation (solution-defining) variable. Let f, the function that needs to be optimised, assign values from $\mathbb{R}^n$ to $\mathbb{R}$ such that $f : \mathbb{R}^n \to \mathbb{R}$.

For minimisation problems, the ideal is to find a global minimum $\mathbf{x}^\star$ such that

$$f(\mathbf{x}^\star) \leq f(\mathbf{x}), \quad \forall \mathbf{x} \in \mathbb{R}^n \tag{3.1}$$

For some applications, a local minimum $\mathbf{x}_L^\star$ on a domain $L \subset \mathbb{R}^n$ is an acceptable solution. In such cases

$$f(\mathbf{x}_L^\star) \leq f(\mathbf{x}), \quad \forall \mathbf{x} \in L \tag{3.2}$$

In both cases, finding a global minimum or a local minimum, the search space can be unconstrained or constrained by a set of constraints. This chapter focuses on Particle Swarm Optimisation (PSO) for unconstrained optimisation, the constrained case is examined in detail in Chapter 4.

Traditionally, numerical optimisation techniques have mainly been developed from the operations research community [18, 37]. The past decade has witnessed an increase in contributions from the artificial intelligence community, most notably from the evolutionary computing field [3]. Recently, PSO has been introduced as a successful technique for numerical optimisation [16, 25, 27]. Other recent methods for optimisation include artificial immune systems, differential evolution, memetic algorithms and scatter search [12].

3.2 Introduction to Particle Swarm Optimisation

Many efficient optimisation algorithms can be constructed from the study of ants working as a colony, birds migrating in a flock toward some destination, or fish swimming in a school. While the individual behaviour of an organism may seem inefficient, the collective effort of individuals inside a swarm can become complex and intelligent [5].

One such a method is Particle Swarm Optimisation (PSO), originally introduced by Kennedy and Eberhart [25]. PSO represents an optimisation method where individuals, called particles, collaborate as a population, or swarm, to reach a collective goal, for example minimising an n-dimensional function f.

Each particle is n-dimensional, and is a potential minimum of f. A particle has memory of the best solution that it has found, called its *personal best*. The particles fly through the search space with a velocity, which is dynamically adjusted according to its personal best and the best solution found by a neighbourhood of particles.

There is thus a sharing of information that takes place. Particles profit from the discoveries and previous experience of other particles during the exploration and search for lower objective function values.

There exist a great number of schemes in which this information sharing can take place. One of two sociometric principles is usually implemented [27], with more recent tolopogies investigated in [26, 28]. The first, called *gbest* (global best), conceptually connects all the particles in the population to one another. Thus each particle is influenced by the very best performance of the entire population. The second, called *lbest* (local best), creates a neighbourhood for each individual comprising itself and its k nearest neighbours in the population. Neighbourhoods are usually determined using particle indices, although topological neighbourhoods have also been used [52].

PSO differs from traditional optimisation methods, in that a population of potential solutions are used in the search. The direct fitness information instead of function derivatives or other related knowledge is used to guide the search. This search is based on probabilistic, rather than deterministic, transition rules.

Let i indicate a particle's index in the swarm. Then

$$S = \{\mathbf{p}_1, \mathbf{p}_2, \ldots, \mathbf{p}_s\}$$

is a swarm of s particles. In PSO each of the s particles has a current position

$$\mathbf{p}_i = (p_{i1}, p_{i2}, \ldots, p_{in})^T$$

and fly through the n-dimensional search space $\mathbb{R}^n$ with a current velocity

$$\mathbf{v}_i = (v_{i1}, v_{i2}, \ldots, v_{in})^T,$$

which is dynamically adjusted according to its own previous best solution

$$\mathbf{z}_i = (z_{i1}, z_{i2}, \ldots, z_{in})^T$$

and the current best solution $\hat{\mathbf{z}}$ of the entire swarm (*gbest*), or the particle's neighbourhood (*lbest*).

At iteration time t of the PSO algorithm, the velocity and particle updates are specified seperately for each dimension j of the velocity and particle vectors. A particle $\mathbf{p}_i$ will interact and move according to the following equations [25]:

$$v_{ij}^{(t+1)} = v_{ij}^{(t)} + c_1 r_1^{(t)} [z_{ij}^{(t)} - p_{ij}^{(t)}] + c_2 r_2^{(t)} [\hat{z}_j^{(t)} - p_{ij}^{(t)}] \tag{3.3}$$

$$p_{ij}^{(t+1)} = v_{ij}^{(t+1)} + p_{ij}^{(t)} \tag{3.4}$$

Equation (3.3) takes three terms into consideration to calculate the velocity of particle i: the particle's previous velocity, the distance between the particle and its personal best, and the distance between the particle and the best solution found by its neighbourhood, which may be the entire swarm.

The stochastic nature of the algorithm is determined by $r_1^{(t)}, r_2^{(t)} \sim \mathit{UNIF}(0,1)$, two uniform random numbers between zero and one. In the second and third terms these numbers are scaled by acceleration coefficients c_1 and c_2, where $0 \leq c_1, c_2 \leq 2$. Coefficient c_1 has been called the *cognitive learning rate* [2], since it scales the second term in (3.3), the term that defines the particle's movement in the direction of its personal best. In the same way, c_2 is called the *social learning rate*, scaling the influence of the neighbourhood's best solution on the particle.

After determining particle i's velocity, it moves toward its new position, as shown in (3.4).

At iteration time t of the PSO algorithm, the personal best of each particle is compared to its current performance. The personal best $\mathbf{z}_i^{(t)}$ is set to the better performance, i.e.

$$\mathbf{z}_i^{(t)} = \begin{cases} \mathbf{z}_i^{(t-1)} & \text{if } f(\mathbf{p}_i^{(t)}) \geq f(\mathbf{z}_i^{(t-1)}) \\ \mathbf{p}_i^{(t)} & \text{if } f(\mathbf{p}_i^{(t)}) < f(\mathbf{z}_i^{(t-1)}) \end{cases} \tag{3.5}$$

The definition of a particle's neighbourhood determines the vector $\widehat{\mathbf{z}}$, the best solution found by either the entire swarm or the particle's neighbourhood. Information sharing takes place through the neighbourhood - the most common, *gbest* and *lbest*, are discussed below. More recent tolopogies are investigated in [26, 28].

3.2.1 Global best (*gbest*)

The global best (*gbest*) PSO conceptually connects all the particles in the population to one another, so that each particle is influenced by the very best performance of the entire population. The global best particle pulls all particles towards itself, and particles move in its direction. If the global best is not updated regularly, the entire swarm may converge to it, resulting in premature convergence.

The global best $\widehat{\mathbf{z}}^{(t)}$ is set to the position of the particle with the best performance within the swarm, i.e.

$$\begin{aligned}\widehat{\mathbf{z}}^{(t)} &\in \{\mathbf{z}_1^{(t)}, \mathbf{z}_2^{(t)}, \ldots, \mathbf{z}_s^{(t)}\} \;\big|\; f(\widehat{\mathbf{z}}^{(t)}) \\ &= \min\{f(\mathbf{z}_1^{(t)}), f(\mathbf{z}_2^{(t)}), \ldots, f(\mathbf{z}_s^{(t)})\}\end{aligned} \tag{3.6}$$

3.2.2 Local best (*lbest*)

The *lbest* (local best) version of the PSO creates a neighbourhood for each individual comprising itself and its k nearest neighbours in the population. Neighbourhoods are usually determined using particle indices, although topological neighbourhoods have also been used [26]. Assuming that particle indices wrap around at s, let N_i be the neighbourhood of particle i.

$$N_i = \{\mathbf{z}_{i-k}^{(t)}, \mathbf{z}_{i-k+1}^{(t)}, \ldots, \mathbf{z}_i^{(t)}, \ldots, \mathbf{z}_{i+k-1}^{(t)}, \mathbf{z}_{i+k}^{(t)}\} \tag{3.7}$$

The neighbourhood best $\widehat{\mathbf{z}}_{N_i}^{(t)}$ at time t is defined as the best solution in particle i's neighbourhood:

$$\widehat{\mathbf{z}}_{N_i}^{(t)} \in N_i \;\big|\; f(\widehat{\mathbf{z}}_{N_i}^{(t)}) = \max\{f(\mathbf{z}_j^{(t)})\} \qquad \forall\, \mathbf{z}_j \in N_i \tag{3.8}$$

It is possible to let the neighbourhood size k be equal to zero, in which case each particle $\mathbf{p}_i$ only compares its current position with its own best position $\mathbf{z}_i^{(t)}$, and no information sharing takes place. A neighbourhood size of k equal to the swarm size s is equivalent to the *gbest* version of the PSO.

It was shown by [16, 48] that, although *lbest* is slower in convergence than *gbest*, *lbest* results in better solutions and searches a larger part of the search space.

3.2.3 The PSO algorithm

All that is left to complete from the above sections is the PSO algorithm itself. The definition of a particle's personal and global or local best position was defined. Using these best positions to determine each particle's velocity, the swarm of particles can successfully traverse the search space, looking for an optimum solution to a problem. The standard PSO algorithm, used to minimise a function

$$f : \mathbb{R}^n \to \mathbb{R} \tag{3.9}$$

is presented below:

Algorithm 3.1 - Particle Swarm Optimisation

1. Set the iteration number t to zero, and initialise the swarm S of n-dimensional particles $\mathbf{p}_i^{(0)}$: each component $p_{ij}^{(0)}$ of $\mathbf{p}_i^{(0)}$ is randomly initialised to a value in the initial domain of the swarm, an interval $[p_{min}, p_{max}]$. Since the particles are already randomly distributed, the velocities of particles are initialised to the zero vector $\mathbf{0}$.

2. Evaluate the performance $f(\mathbf{p}_i^{(t)})$ of each particle.

3. Compare the personal best of each particle to its current performance, and set $\mathbf{z}_i^{(t)}$ to the better performance, as shown in (3.5).

4. Use (3.6) to set the global best $\widehat{\mathbf{z}}^{(t)}$ to the position of the particle with the best performance within the entire swarm (*gbest*). When a *lbest* PSO is implemented, equation (3.8) is used to set the neighbourhood best $\widehat{\mathbf{z}}_{N_i}^{(t)}$ for each particle i.

5. Change the velocity vector for each particle according to equation (3.3).

6. Move each particle to its new position, according to equation (3.4).

7. Let $t := t + 1$.

8. Go to step 2, and repeat until convergence or $t = t_{max}$.

The algorithm has converged if the difference between the best solution found over a specified number of iterations remains within a certain bound. The algorithm iterates until either one of two conditions is met: the algorithm has converged, or the maximum number of iterations t_{max} have been reached.

3.2.4 Improvements

A number of methods have been proposed to improve the convergence and probability of convergence of the standard PSO algorithm, and are discussed in this section. Apart from changes to the PSO update equation (3.3), most of these methods make no changes to the PSO algorithm itself.

Maximum velocity

The probability of particles leaving the current search space can be reduced by clamping the velocity updates – the velocity update vectors in the first term of (3.3) can be restricted by specifying upper and lower bounds v_{max} and $-v_{max}$ on $v_{ij}^{(t)}$. If $v_{ij}^{(t)}$ is greater than v_{max}, then $v_{ij}^{(t)}$ is set to v_{max}. Similary, if $v_{ij}^{(t)}$ is smaller than $-v_{max}$, then $v_{ij}^{(t)}$ is set to the value of $-v_{max}$. The value of v_{max} is usually a function of the range of the problem. If the range of each component p_{ij} of particle $\mathbf{p}_i$ is between -10 and 10, v_{max} will be proportional to 10.

Inertia weight

The previous velocity in the first term of (3.3) can be scaled with an intertia weight w, i.e.

$$v_{ij}^{(t+1)} = wv_{ij}^{(t)} + c_1 r_1^{(t)}[z_{ij}^{(t)} - p_{ij}^{(t)}] + c_2 r_2^{(t)}[\hat{z}_j^{(t)} - p_{ij}^{(t)}] \tag{3.10}$$

The inertia weight was introduced to improve the rate of convergence of the PSO algorithm [47], and determines how much the velocity at time t should influence the velocity at time $t+1$. A large inertia weight causes the PSO to explore larger parts of the search space, while a smaller inertia weight results in exploitation of a smaller and more focussed region of the search space. An inertia weight of one results in an update equation equivalent to (3.3).

It is possible, through careful selection of the inertia weight, to create a balance between local and global exploration abilities, and therefore create a faster rate of convergence. The balance can be achieved with a linearly decreasing inertia weight

$$w = w_{max} - \frac{t}{t_{max}}(w_{max} - w_{min}) \tag{3.11}$$

where w_{max} is the initial (starting) inertia weight, and w_{min} is the final weight. The values t_{max} and t respectively indicate the maximum and current iteration number. Setting w_{max} to 0.9 and w_{min} to 0.4 has been shown to give good conversion, independent of the problems tested [46, 47].

Constriction coefficient

Maurice Clerc has introduced a *constriction factor* to PSO, which improves PSO's ability to control velocities [11]. The constriction factor analytically chooses values for w, c_1 and c_2 such that control is allowed over the dynamical characteristics of the particle swarm, including its exploration versus exploitation abilities. Clamping the velocities is not necessary when a constriction coefficient χ is used in (3.3), changing the velocity update to

$$v_{ij}^{(t+1)} = \chi\big(v_{ij}^{(t)} + c_1 r_1^{(t)}[z_{ij}^{(t)} - p_{ij}^{(t)}] + c_2 r_2^{(t)}[\widehat{z}_j^{(t)} - p_{ij}^{(t)}]\big) \tag{3.12}$$

with

$$\chi = \frac{2}{|2 - \varphi - \sqrt{\varphi^2 - 4\varphi}|} \tag{3.13}$$

and $\varphi = c_1 + c_2$, $\varphi > 4$.

As the value of φ tends to 4 (from above), the value of χ tends to 1 (from below), and the particle's velocity is almost not damped at all; as φ grows larger, χ tends to zero, and the particle's velocity is more strongly damped. A correct choice of the constriction factor makes velocity clamping unnecessary, although it was found that χ, combined with constraints on v_{max}, significantly improved PSO performance [17].

Guaranteed Convergence Particle Swarm Optimiser

The PSO described in this chapter, including the versions with an inertia weight (3.10) and constriction factor (3.12), all have a probability of converging prematurely. This can be clearly seen by considering the case when a particle's position and personal best coincide with the global best. The velocity of the particle will only depend on $v_{ij}^{(t)}$ (or $wv_{ij}^{(t)}$ or $\chi v_{ij}^{(t)}$), and if it is close to zero, or the position of the global best does not change, the particle will 'catch up' with the global best. This does not mean that the swarm has converged to a minimum, but merely that it has converged prematurely to the global best.

Van den Berg has introduced the Guaranteed Convergence PSO (GCPSO) [53, 54], which defines a different velocity update for the global best particle. If τ is the index of the global best particle, such that $\mathbf{z}_\tau = \widehat{\mathbf{z}}$, then the new velocity update ensures that a point is sampled from the support of a probability measure containing $\widehat{\mathbf{z}}$ or close to $\widehat{\mathbf{z}}$:

$$v_{\tau,j}^{(t+1)} = -p_{\tau,j}^{(t)} + \widehat{z}_j^{(t)} + wv_{\tau,j}^{(t)} + \rho^{(t)}(1 - 2r_2^{(t)}) \tag{3.14}$$

The value ρ is a scaling factor used to generate a random sample space with ρ as its side lengths, with $r_2^{(t)}$ again being uniformly distributed between zero and one. In essence the velocity update resets the particle's position to that of the global best, and adds the current search direction. To this result a random vector from $\rho^{(t)}(1-2r_2^{(t)})$ is added. By combining equations (3.4) and (3.14), the position of the new particle will be

$$p_{\tau,j}^{(t+1)} = \hat{z}_j^{(t)} + wv_{\tau,j}^{(t)} + \rho^{(t)}(1-2r_2^{(t)}) \tag{3.15}$$

The size of the random search volume is changed by expanding ρ when better function evaluations are successfully found. The sampling volume is decreased when no improvements to the function evaluation is found over time; the smaller volume increases the probability of choosing a variable that gives a better objective function value. If a failure in decreasing the objective function is equivalent to $f(\hat{\mathbf{z}}^{(t)}) = f(\hat{\mathbf{z}}^{(t-1)})$, then the value of $\rho^{(t)}$ is adapted after each iteration of the GCPSO algorithm with

$$\rho^{(t+1)} = \begin{cases} 2\rho^{(t)} & \text{if } \#s > s_c \\ \frac{1}{2}\rho^{(t)} & \text{if } \#f > f_c \\ \rho^{(t)} & \text{otherwise} \end{cases} \tag{3.16}$$

The terms $\#s$ and $\#f$ respectively denote the number of consecutive successes and failures, with s_c and f_c being threshold parameters. To ensure the correctness of (3.16), $\#f$ is set to zero if $\#s$ increases from iteration t to iteration $t+1$ of the algorithm. In a similar fashion, $\#s$ is set to zero when $\#f$ increases. A rigorous analysis of GCPSO can be found in [53].

3.3 Concluding

The basic PSO algorithm was discussed in this chapter, and a (by no means exhaustive) number of improvements were shown. In particular, this chapter has focused on improvements to the PSO that are relevant to the rest of this thesis. The GCPSO is of particular interest, since it will be the basis for development of the Converging Linear PSO in Chapter 4. The interested reader is referred to [7, 27, 53], the proceedings of the *Particle Swarm Optimization Workshop (2001)*, and the proceedings of the *IEEE Swarm Intelligence Symposium (2003)* for a thorough treatment of research in Particle Swarm Optimisers.

An overview of unconstrained optimisation was given, but it will only serve as a platform from which PSO will be extended to optimise constrained problems. The following chapter takes care of this extension, by examining and analysing a method of linear constraint handling. Inequality constraints are also taken care of, and finally we not only have a PSO that can train Support Vector Machines, but can also optimise general problems with both linear equality and inequality constraints.

Chapter 4

Constrained Particle Swarm Optimisation

The standard Particle Swarm Optimiser is unable to easily optimise functions bound by a set of linear equality or inequality constraints. The objective of this chapter is to present two new algorithms, the Linear Particle Swarm Optimiser (LPSO) and the Converging Linear Particle Swarm Optimiser (CLPSO), designed specifically with constrained optimisation in mind. The properties and convergence of these new algorithms are carefully analysed; a proof for a set of initial conditions on LPSO, a proof of both algorithms' ability to search within the constrained space, and a convergence proof for CLPSO, is given.

4.1 Introduction to constrained optimisation

Optimisation problems have the goal of finding the best value of some function. These types of problems are generally composed of three parts: an *objective function* that needs to be optimised (minimised or maximised), a set of solution-defining *variables* on which the objective function depends, and a set of *constraints* that restricts feasible values of these variables. Constraints can be of two types: *equality* constraints specify that a function of the variables must be equal to a constant, while *inequality* constraints specify that a certain function of the variables must be greater than or equal to (or less than or equal to) a constant.

4.1.1 Terminology

Let $\mathbf{x} = (x_1, x_2, \ldots, x_n)^T \in \mathbb{R}^n$ represent the solution-defining variable. This vector $\mathbf{x} \in \mathbb{R}^n$ is called the optimisation variable. The function that needs to be optimised is defined as

$f : \mathbb{R}^n \rightarrow \mathbb{R}$, and is commonly called the objective function or cost function. Let the set of functions $g_i : \mathbb{R}^n \rightarrow \mathbb{R}$ define the inequality constraint functions, giving a set of inequalities $g_i(\mathbf{x}) \leq 0$. Defining $h_i(\mathbf{x}) = 0$ as equality constraints, the functions $h_i : \mathbb{R}^n \rightarrow \mathbb{R}$ are the equality constraint functions. If there are no constraints, the problem is called an unconstrained problem, as was examined in Chapter 3.

Using the above notation, a general optimisation problem can be stated as

$$\begin{aligned} \text{Minimise} \quad & f(\mathbf{x}) \\ \text{Subject to} \quad & g_i(\mathbf{x}) \leq 0, \ i = 1 \ldots k \\ & h_i(\mathbf{x}) = 0, \ i = 1 \ldots m \end{aligned} \tag{4.1}$$

to describe the problem of finding an optimisation variable $\mathbf{x}$ that minimises $f(\mathbf{x})$ over all values of $\mathbf{x}$ that satisfy the conditions $g_i(\mathbf{x}) = 0$, $i = 1 \ldots k$, and $h_i(\mathbf{x}) = 0$, $i = 1 \ldots m$.

The maximum of $f(\mathbf{x})$ can be found by minimising $-f^{\star}(\mathbf{x})$. In a similar way, an inequality constraint function $g_i(\mathbf{x}) \geq 0$ can be written in the standard form with $-g_i(\mathbf{x}) \leq 0$.

The domain Ω of the constrained optimisation problem is the set of $\mathbf{x}$-values for which the objective and all constraint functions are defined. If $dom(g_i)$ denotes the set of $\mathbf{x}$-values for which $g_i(\mathbf{x}) \leq 0$, and $dom(h_i)$ denotes the set of $\mathbf{x}$-values for which $h_i(\mathbf{x}) = 0$, then Ω is the intersecion of these domains.

$$\Omega = \bigcap_{i=1}^{k} dom(g_i) \ \cap \ \bigcap_{i=1}^{m} dom(h_i) \tag{4.2}$$

A point $\mathbf{x} \in \Omega$ is feasible if it satisfies the constraints $g_i(\mathbf{x}) \leq 0$ and $h_i(\mathbf{x}) = 0$.

If Ω is non-empty, there exists at least one feasible point, and the problem is feasible. If no feasible point exists, the problem is infeasible. The domain is the set of all feasible points, called the feasible set. The problem of determining whether the problem is feasible or not is called the feasibility problem. The feasibility problem determines if the inequalities are consistent, and if so, finds a point that satisfies them. It is written as

$$\begin{aligned} \text{Find} \quad & \mathbf{x} \\ \text{Subject to} \quad & g_i(\mathbf{x}) \leq 0, \ i = 1 \ldots k \\ & h_i(\mathbf{x}) = 0, \ i = 1 \ldots m \end{aligned} \tag{4.3}$$

If $\mathbf{x}$ is feasible and $g_i(\mathbf{x}) = 0$, the constraint $g_i(\mathbf{x}) \leq 0$ is *active* at $\mathbf{x}$. If $g_i(\mathbf{x}) < 0$, the constraint $g_i(\mathbf{x}) \leq 0$ is *inactive*. The equality constraint $h_i(\mathbf{x}) = 0$ is active at all feasible points. Redundant constraints are constraints that are implied by other constraints, and deleting them will not change the set of feasible solutions.

The optimal or minimal value $f^{\star}$ of problem (4.1) is defined as

$$f^\star = \inf\{f(\mathbf{x}) \mid g_i(\mathbf{x}) \le 0,\ i = 1 \dots k, \ \text{ and } \ h_i(\mathbf{x}) = 0,\ i = 1 \dots m\} \tag{4.4}$$

The values of $f^\star$ are allowed to take on the extended values $\pm\infty$. If the problem is infeasible, and the standard convention that the infimum of the empty set is $+\infty$ is used, the optimal value $f^\star$ is equal to $+\infty$.

It is also possible that the problem is unbounded from below, such that $f^\star = -\infty$. A problem unbounded from below contains a sequence of feasible points $\{\mathbf{x}_j\}_{j \ge 1}$ with $f(\mathbf{x}_j) \to -\infty$ as $j \to \infty$.

For $\mathbf{x} \in \mathbb{R}^n$ to be an optimal point, it must be feasible and have $f(\mathbf{x}) = f^\star$. The problem may contain more than one feasible $\mathbf{x}$ that minimises $f(\mathbf{x})$, and the set of these optimal points is denoted by

$$\begin{aligned} \mathcal{X} = \{\mathbf{x} \mid g_i(\mathbf{x}) \le 0,\ i = 1 \dots k, \ \text{ and} \\ h_i(\mathbf{x}) = 0,\ i = 1 \dots m, \ \text{ and } \ f(\mathbf{x}) = f^\star\} \end{aligned} \tag{4.5}$$

If $\mathcal{X}$ is not empty, the optimal value can be found and the problem is solvable. If $\mathcal{X}$ is empty, an optimal value can not be found.

An approximate solution to the problem is very often sufficient if a numeric method is used to find it. This approximate solution must lie within an error margin $\epsilon > 0$ from the true solution. The solution $\mathbf{x}$ with $f(\mathbf{x}) \le f^\star + \epsilon$ is called ϵ-suboptimal and the set of all ϵ-suboptimal points is called the ϵ-suboptimal set for the problem.

A *local* solution to the optimisation problem is a feasible point $\mathbf{x}$ which will minimise f on the set of nearby feasible solutions within a certain radius from $\mathbf{x}$. A feasible point $\mathbf{x}$ is thus locally optimal if there is an $R > 0$ such that

$$\begin{aligned} f(\mathbf{x}) = \inf\{f(\mathbf{z}) \mid g_i(\mathbf{z}) \le 0,\ i = 1 \dots k, \ \text{ and} \\ h_i(\mathbf{z}) = 0,\ i = 1 \dots m, \ \text{ and } \ \|\mathbf{z} - \mathbf{x}\| \le R\} \end{aligned} \tag{4.6}$$

The term 'globally optimal' is used for 'optimal' to distinguish between 'locally optimal' and 'optimal.'

4.1.2 Expressing problems in the standard form

Optimisation problem (4.1) is referred to as a standard form optimisation problem. The convention is chosen that the right-hand side of the inequality and equality constraints are zero. This can always be arranged by subtracting any nonzero right-hand side: for example, the equality constraint $h_i^{(1)}(\mathbf{x}) = h_i^{(2)}(\mathbf{x})$ is written as $h_i(\mathbf{x}) = 0$, where $h_i(\mathbf{x}) = h_i^{(1)}(\mathbf{x}) - h_i^{(2)}(\mathbf{x})$. In a similar way inequalities of the form $g_i(\mathbf{x}) \ge 0$ are expressed as $-g_i(\mathbf{x}) \le 0$.

4.1.3 Slack variables

Problem (4.1),

$$\begin{aligned} \text{Minimise} \quad & f(\mathbf{x}) \\ \text{Subject to} \quad & g_i(\mathbf{x}) \leq 0, \ i = 1 \dots k \\ & h_i(\mathbf{x}) = 0, \ i = 1 \dots m \end{aligned}$$

can be rewritten so that all inequalities involve only a single variable, instead of an entire function $g_i(\mathbf{x})$. These single variables are called *slack variables* and replaces each inequality constraint with an equality constraint, and a non-negativity constraint. There is one slack variable s_i associated with each original inequality constraint $g_i(\mathbf{x}) \leq 0$. Optimisation problem (4.1) is rewritten as

$$\begin{aligned} \text{Minimise} \quad & f(\mathbf{x}) \\ \text{Subject to} \quad & g_i(\mathbf{x}) + s_i = 0, \ i = 1 \dots k \\ & h_i(\mathbf{x}) = 0, \quad i = 1 \dots m \\ & s_i \geq 0, \qquad i = 1 \dots k \end{aligned} \tag{4.7}$$

where the variables are $\mathbf{x} \in \mathbb{R}^n$ and $\mathbf{s} \in \mathbb{R}^k$. This problem has $n + k$ variables, k inequality constraints (the non-negativity constraints on s_i), and $k + m$ equality constraints.

The problem is equivalent to the original standard form problem. If $(\mathbf{x}, \mathbf{s})$ is feasible for the above problem, then $\mathbf{x}$ is feasible for the original problem, since $s_i = -g_i(\mathbf{x}) \geq 0$. Conversely, if $\mathbf{x}$ is feasible for the original problem, then $(\mathbf{x}, \mathbf{s})$ is feasible for the above problem, where $s_i = -g_i(\mathbf{x})$. Similarly, $\mathbf{x}$ is optimal for the original problem if and only if $(\mathbf{x}, \mathbf{s})$ is optimal for the above problem, where $s_i = -g_i(\mathbf{x})$.

4.1.4 Convex optimisation

A convex optimisation problem is one of the form

$$\begin{aligned} \text{Minimise} \quad & f(\mathbf{x}) \\ \text{Subject to} \quad & g_i(\mathbf{x}) \leq 0, \ i = 1 \dots k \\ & A\mathbf{x} = \mathbf{b}, \ \ A \in \mathbb{R}^{m \times n} \ \text{ and } \ \mathbf{b} \in \mathbb{R}^m \end{aligned} \tag{4.8}$$

where both f and $g_1, \dots, g_k$ are convex functions. The convex problem has three additional requirements compared to the general standard form problem (4.1): the objective is convex, the inequality constraint functions are convex, and the equality constraint functions $h_i(\mathbf{x}) = \mathbf{a}_i^T \mathbf{x} - b_i$ are affine.

The additional requirements give rise to an important property: the feasible set of a convex optimisation problem is convex, since it is the domain of the problem

$$\Omega = \bigcap_{i=1}^{k} dom(g_i) \ \cap \ \{\mathbf{x} \mid A\mathbf{x} = \mathbf{b}\} \tag{4.9}$$

which is a convex set, with k (convex) sublevel sets $\{\mathbf{x} \mid g_i(\mathbf{x}) \leq 0\}$ and m hyperplanes $\{\mathbf{x} \mid \mathbf{a}_i^T\mathbf{x} = b_i\}$.

An important property of convex optimisation problems is that any locally optimal point is also globally optimal. To see this, suppose that (feasible) $\mathbf{x}$ is locally optimal for a convex optimisation problem, and

$$\begin{aligned} f(\mathbf{x}) = \inf\{f(\mathbf{z}) \bigm| g_i(\mathbf{z}) \leq 0,\ i = 1\ldots k,\ \text{ and}\\ \mathbf{a}_i^T\mathbf{x} = b_i,\ i = 1\ldots m,\ \text{ and }\ \|\mathbf{z}-\mathbf{x}\| \leq R\} \end{aligned} \tag{4.10}$$

for some $R > 0$. Now suppose that $\mathbf{x}$ is not globally optimal, such that there is a feasible $\mathbf{y}$ with $f(\mathbf{y}) < f(\mathbf{x})$. As a result $\|\mathbf{y}-\mathbf{x}\| > R$, since otherwise $f(\mathbf{x}) \leq f(\mathbf{y})$. Consider the point $\mathbf{z}$ given by

$$\mathbf{z} = (1-\alpha)\mathbf{x} + \alpha\mathbf{y}, \quad \alpha = \frac{R}{2\|\mathbf{y}-\mathbf{x}\|} \tag{4.11}$$

It follows that $\|\mathbf{z}-\mathbf{x}\| = \frac{R}{2} < R$, and by convexity of the feasible set, $\mathbf{z}$ is feasible. By convexity of f it follows that

$$f(\mathbf{z}) \leq (1-\alpha)f(\mathbf{x}) + \alpha f(\mathbf{y}) < f(\mathbf{x}) \tag{4.12}$$

which contradicts (4.10). Hence there exists no feasible $\mathbf{y}$ with $f(\mathbf{y}) < f(\mathbf{x})$, and it follows that $\mathbf{x}$ is globally optimal.

4.1.5 Duality

Consider the standard optimisation problem (4.1), with a non-empty domain Ω, also called a 'primal problem.' The constraints in (4.1) can be introduced to the objective function by augmenting it with a weighted sum of the constraint functions. Let the vector $\boldsymbol{\mu} \in \mathbb{R}^k$ be associated with the set of k inequality constraints, and $\boldsymbol{\lambda} \in \mathbb{R}^m$ be associated with the set of m equality constraints. These vectors are called the Lagrange multiplier vectors, and define the Lagrangian $L : \mathbb{R}^n \times \mathbb{R}^k \times \mathbb{R}^m \to \mathbb{R}$ associated with (4.1) as

$$L(\mathbf{x}, \boldsymbol{\mu}, \boldsymbol{\lambda}) = f(\mathbf{x}) + \sum_{i=1}^{k} \mu_i g_i(\mathbf{x}) + \sum_{i=1}^{m} \lambda_i h_i(\mathbf{x}) \tag{4.13}$$

The dual problem associated with the primal problem is written as

$$\begin{aligned} &\text{Maximise } L(\mathbf{x}, \boldsymbol{\mu}, \boldsymbol{\lambda}) \text{ with respect to } \boldsymbol{\mu} \text{ and } \boldsymbol{\lambda} \\ &\text{Subject to} \quad \boldsymbol{\mu} \geq 0 \end{aligned} \tag{4.14}$$

If the problem (4.1) is convex, then the solution of the primal problem is the vector $\mathbf{x}^\star$ of the saddle point $(\mathbf{x}^\star, \boldsymbol{\mu}^\star, \boldsymbol{\lambda}^\star)$ of (4.13) such that

$$L(\mathbf{x}^\star, \boldsymbol{\mu}, \boldsymbol{\lambda}) \leq L(\mathbf{x}^\star, \boldsymbol{\mu}^\star, \boldsymbol{\lambda}^\star) \leq L(\mathbf{x}, \boldsymbol{\mu}^\star, \boldsymbol{\lambda}^\star) \tag{4.15}$$

The vector $\mathbf{x}^\star$ that solves the primal problem, as well as the two Lagrange multiplier vectors $\boldsymbol{\mu}$ and $\boldsymbol{\lambda}$, can be found by solving the min-max problem

$$\min_{\mathbf{x}} \max_{\boldsymbol{\mu}, \boldsymbol{\lambda}} L(\mathbf{x}, \boldsymbol{\mu}, \boldsymbol{\lambda}) \tag{4.16}$$

4.1.6 Equality-constrained optimisation

The final optimisation problem introduced, is the problem of minimising f under a set of linear equality constraints. This problem is written as

$$\begin{aligned} &\text{Minimise} \quad f(\mathbf{x}) \\ &\text{Subject to } A\mathbf{x} = \mathbf{b},\ A \in \mathbb{R}^{m \times n} \ \text{ and } \ \mathbf{b} \in \mathbb{R}^m \end{aligned} \tag{4.17}$$

and will from the basis of PSO developments to follow. In the following sections, a PSO algorithm is developed to successfully handle the above equality constrained optimisation problem. The method is extended to handle inequality constraints as well.

4.2 Linear Particle Swarm Optimisation

The Particle Swarm Optimisation (PSO) algorithm discussed in Chapter 3 is an algorithm suited for unconstrained optimisation. This section introduces a new PSO algorithm, the Linear Particle Swarm Optimiser, that is specifically developed with linear constraints in mind. The update equations for a particle's velocity and position, with inertia weight w, is repeated here:

$$v_{ij}^{(t+1)} = wv_{ij}^{(t)} + c_1 r_1^{(t)}[z_{ij}^{(t)} - p_{ij}^{(t)}] + c_2 r_2^{(t)}[\hat{z}_j^{(t)} - p_{ij}^{(t)}] \tag{4.18}$$

$$p_{ij}^{(t+1)} = v_{ij}^{(t+1)} + p_{ij}^{(t)} \tag{4.19}$$

Traditionally, the above velocity and position update steps are specified separately for each dimension of a particle, as is done in [25, 27, 48, 53]. If the random numbers $r_1^{(t)}$ and $r_2^{(t)}$

are rather kept constant for all vector dimensions, the velocity updates are calculated as a linear combination of position and velocity vectors.

$$\mathbf{v}_i^{(t+1)} = w\mathbf{v}_i^{(t)} + c_1 r_1^{(t)}[\mathbf{z}_i^{(t)} - \mathbf{p}_i^{(t)}] + c_2 r_2^{(t)}[\widehat{\mathbf{z}}^{(t)} - \mathbf{p}_i^{(t)}] \tag{4.20}$$

$$\mathbf{p}_i^{(t+1)} = \mathbf{v}_i^{(t+1)} + \mathbf{p}_i^{(t)} \tag{4.21}$$

The above approach has the advantage that the flight of particles is defined by standard linear operations on vectors. The guaranteed movement of particles through subspaces and subsets becomes possible, as stated in Theorem 4.1 (to follow). The PSO algorithm using update equations (4.20, 4.21) is referred to as a "Linear Particle Swarm Optimiser" (LPSO), due to the way the update equations are formulated. The LPSO algorithm, used to minimise a function

$$f : \mathbb{R}^n \to \mathbb{R} \tag{4.22}$$

is presented below:

Algorithm 4.1 - Linear Particle Swarm Optimisation (LPSO)

1. Set the iteration number t to zero, and randomly initialise the swarm S of n-dimensional particles $\mathbf{p}_i^{(0)}$ to a value in the initial domain of the swarm. Initialise all velocity vectors $\mathbf{v}_i = \mathbf{0}$.

2. Evaluate the performance $f(\mathbf{p}_i^{(t)})$ of each particle.

3. Compare the personal best of each particle to its current performance, and set $\mathbf{z}_i^{(t)}$ to the better performance:

$$\mathbf{z}_i^{(t)} = \begin{cases} \mathbf{z}_i^{(t-1)} & \text{if } f(\mathbf{p}_i^{(t)}) \geq f(\mathbf{z}_i^{(t-1)}) \\ \mathbf{p}_i^{(t)} & \text{if } f(\mathbf{p}_i^{(t)}) < f(\mathbf{z}_i^{(t-1)}) \end{cases} \tag{4.23}$$

4. Set the global best $\widehat{\mathbf{z}}^{(t)}$ to the position of the best performance in the swarm:

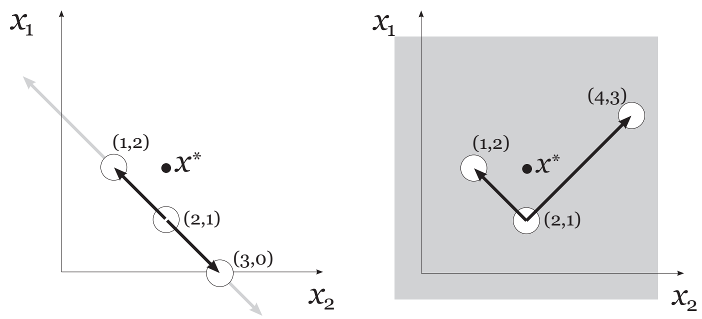
$$\begin{aligned} \widehat{\mathbf{z}}^{(t)} &\in \{\mathbf{z}_1^{(t)}, \mathbf{z}_2^{(t)}, \ldots, \mathbf{z}_s^{(t)}\} \mid f(\widehat{\mathbf{z}}^{(t)}) \\ &= \min\{f(\mathbf{z}_1^{(t)}), f(\mathbf{z}_2^{(t)}), \ldots, f(\mathbf{z}_s^{(t)})\} \end{aligned} \tag{4.24}$$

5. Change the velocity vector for each particle according to equation (4.20).

6. Move each particle to its new position, according to equation (4.21).

7. Let $t := t + 1$.

8. Go to step 2, and repeat until convergence or $t = t_{max}$.



(a) $S^\star$ spanning $\mathbb{R}^1$. (b) $S^\star$ spanning $\mathbb{R}^2$.

FIGURE 4.1: Comparing the possible search spaces resulting from different initial swarms in LPSO, with $\mathbf{v}_i^{(0)} = \mathbf{0}$.

The algorithm has converged if the difference between the best solution found over a specified number of iterations remains within a certain bound. The algorithm iterates until either one of two conditions is met: the algorithm has converged, or the maximum number of iterations t_{max} have been reached. In essence the convergence and stopping conditions are therefore the same as for the standard PSO.

4.2.1 Criteria on the initial swarm

If a PSO is considered in the traditional sense, with random numbers $r_1^{(t)}$ and $r_2^{(t)}$ generated for each dimension in a particle's update equations (4.18, 4.19), any point in the search space can possibly be reached with a swarm of arbitrary size. It is even possible to reach any point in the search space with a swarm of size two [27].

This generalisation does not work for the LPSO, where the update equations (4.20) and (4.21) are in fact linear combinations of position and velocity vectors. The initial swarm will thus influence which positions can and cannot be found.

In fact, if all velocities are initialised to zero (like in the LPSO algorithm above), only positions in the span of the set of vectors created by subtracting the initial global best $\widehat{\mathbf{z}}^{(0)}$ from each initial position vector, will be found. A similar idea is true if the initial velocities are non-zero, where the initial velocity vectors are added to the previous set of vectors (created by subtracting the global best $\widehat{\mathbf{z}}^{(0)}$ from each initial position vector) to span the set of possible solutions found.

Consider the example illustrated in Figures 4.1(a) and 4.1(b), and say $f(\mathbf{x})$ is minimized at a point (or vector) $\mathbf{x}^\star$. If the LPSO algorithm is able to find $\mathbf{x}^\star$, vector $\mathbf{x}^\star$ should be decomposable into a linear combination of the initial velocity vectors.

It is easy to see from Figure 4.1(a) that a swarm with initial population $\{(1,2),\ (2,1),\ (3,0)\}$

will never be able to reach $\mathbf{x}^\star = (2,2)$. This is due to the way the particles are moved with velocities which are initialised to the zero vector. $\mathbf{v}_1^{(t)}$ and $\mathbf{v}_2^{(t)}$ will cause the particles to fly on a straight line, since all possible velocities will be of the form $\alpha[(1,2)-(2,1)] = \alpha(-1,1)$, with $\alpha \in \mathbb{R}$. All personal and global bests will also lie on this line, and thus searches will be in $\mathbb{R}^1$ and not in $\mathbb{R}^2$. If $\widehat{\mathbf{z}}^{(0)} = (2,1)$, then the set of vectors

$$\{(1,2)-\widehat{\mathbf{z}}^{(0)},\ (2,1)-\widehat{\mathbf{z}}^{(0)},\ (3,0)-\widehat{\mathbf{z}}^{(0)}\} = \{(-1,1),\ (0,0),\ (1,-1)\}$$

as shown in Figure 4.1(a), will only span $\mathbb{R}^1$. The optimal value $\mathbf{x}^\star$ at $(2,2)$ can not be reached. In comparison, Figure 4.1(b) shows that the set of vectors

$$\{\mathbf{p}-\widehat{\mathbf{z}}^{(0)} \mid \mathbf{p} \in S^{(0)}\} = \{(-1,1),\ (0,0),\ (2,2)\}$$

from the initial swarm $S^{(0)} = \{(1,2),\ (2,1),\ (4,3)\}$ will span $\mathbb{R}^2$, and $\mathbf{x}^\star$ at $(2,2)$ can possibly be reached.

This leads us to a first important theorem, which makes the following assumptions from Algorithm 4.1 (LPSO):

1. $\mathbf{v}_i^{(0)} = \mathbf{0}$
2. $\mathbf{z}_i^{(0)} = \mathbf{p}_i^{(0)}$

Theorem 4.1

If f needs to be optimized in $\mathbb{R}^n$, a swarm of s particles $S^{(0)} = \{\mathbf{p}_1^{(0)},\ \mathbf{p}_2^{(0)},\ \ldots,\mathbf{p}_s^{(0)}\}$ will be able to find the optimal value $\mathbf{x}^\star$ if and only if there exists a subset $S^\star \subseteq S^{(0)} - \widehat{\mathbf{z}}^{(0)} = \{\mathbf{p}-\widehat{\mathbf{z}}^{(0)} \mid \mathbf{p} \in S^{(0)}\}$ that forms a basis for $\mathbb{R}^n$.

Proof

Say the optimal value $\mathbf{x}^\star$ can be found. Then $\mathbf{x}^\star$ can be written as some $\mathbf{p}_k^{(0)} \in S^{(0)}$ plus a linear combination of the set of initial velocity vectors

$$\left\{w\mathbf{v}_i^{(0)} + c_1 r_1^{(0)}[\mathbf{z}_i^{(0)} - \mathbf{p}_i^{(0)}] + c_2 r_2^{(0)}[\widehat{\mathbf{z}}^{(0)} - \mathbf{p}_i^{(0)}] \mid i = 1\ldots s\right\}$$

Since $\mathbf{v}_i^{(0)} = \mathbf{0}$, $\mathbf{z}_i^{(0)} = \mathbf{p}_i^{(0)}$ and $-c_2 r_2^{(0)}$ is a non-zero scalar, $\mathbf{x}^\star - \mathbf{p}_k^{(0)}$ can be written as a linear combination of

$$S^{(0)} - \widehat{\mathbf{z}}^{(0)} = \left\{\mathbf{p}_1^{(0)} - \widehat{\mathbf{z}}^{(0)},\ \mathbf{p}_2^{(0)} - \widehat{\mathbf{z}}^{(0)},\ \ldots,\mathbf{p}_s^{(0)} - \widehat{\mathbf{z}}^{(0)}\right\}$$

Because any vector $\mathbf{x}^\star - \mathbf{p}_k^{(0)} \in \mathbb{R}^n$ can be written as a linear combination of $S^{(0)} - \widehat{\mathbf{z}}^{(0)}$, it is true that $S^{(0)} - \widehat{\mathbf{z}}^{(0)}$ spans $\mathbb{R}^n$. Thus there exits a subset $S^\star \subseteq S^{(0)} - \widehat{\mathbf{z}}^{(0)}$ of linearly independent vectors that also spans $\mathbb{R}^n$. This subset $S^\star$ will form a basis for $\mathbb{R}^n$.

To prove the converse, assume $S^\star \subseteq S^{(0)} - \widehat{\mathbf{z}}^{(0)}$ forms a basis for $\mathbb{R}^n$. Then any vector in $\mathbb{R}^n$ can be written as a linear combination of $S^{(0)} - \widehat{\mathbf{z}}^{(0)}$. The optimal value can thus be written as $\mathbf{p}_k^{(0)} \in S^{(0)}$ plus some linear combination of $S^{(0)} - \widehat{\mathbf{z}}^{(0)}$, and can thus be reached.

$\square$

Since one of the particles will be the global best particle with $\mathbf{p}_i^{(0)} - \widehat{\mathbf{z}}^{(0)} = \mathbf{0}$, the set of vectors $S^{(0)} - \widehat{\mathbf{z}}^{(0)}$ will contain the zero vector, and so $S^{(0)}$ needs to contain a minimum of $n+1$ vectors for $S^{(0)} - \widehat{\mathbf{z}}^{(0)}$ to span $\mathbb{R}^n$, namely

$$\inf |S^{(0)}| = n + 1 \tag{4.25}$$

To explore the case when initial velocities are non-zero, consider the LPSO update equations (4.20) and (4.21). Assuming that the initial personal best $\mathbf{z}_i^{(0)}$ is set to $\mathbf{p}_i^{(0)}$, two vectors play a role in particle i's update equations: the initial velocity vector $\mathbf{v}_i^{(0)}$ and the difference between the initial global best $\widehat{\mathbf{z}}^{(0)}$ and the initial position $\mathbf{p}_i^{(0)}$. It follows that the set of vectors

$$\left\{\mathbf{v}_1^{(0)},\, \widehat{\mathbf{z}}^{(0)} - \mathbf{p}_1^{(0)},\, \mathbf{v}_2^{(0)},\, \widehat{\mathbf{z}}^{(0)} - \mathbf{p}_2^{(0)},\, \ldots, \mathbf{v}_s^{(0)},\, \widehat{\mathbf{z}}^{(0)} - \mathbf{p}_s^{(0)}\right\}$$

must span $\mathbb{R}^n$, and the minimum swarm size for LPSO of $S^{(0)}$ will be $\lceil \frac{n}{2} \rceil + 1$.

4.3 Equality-constrained optimisation

The LPSO algorithm lends itself perfectly to solving equality-constrained optimisation problems, as was discussed in Section 4.1.6. This section summarises current methods from the Evolutionary Computing and PSO fields, and discusses and proves the usefulness of LPSO to equality-constrained optimisation.

4.3.1 Current methods

Many methods for constraint handling have been proposed in the Evolutionary Computation field [32]. These can be broadly classified into *penalty*, *repair* and *constraint-preserving* methods.

Penalty methods add a penalty to the objective function to decrease the quality of infeasible solutions [20, 22, 32]. While penalty methods are very popular, they do not guarantee a solution where no constraints are violated, since the search space still includes infeasible solutions, and success depends on the penalty method.

Repair methods apply operators to move an infeasible solution closer to the feasible space of solutions [30, 60]. Operators designed to 'correct' infeasible solutions are usually

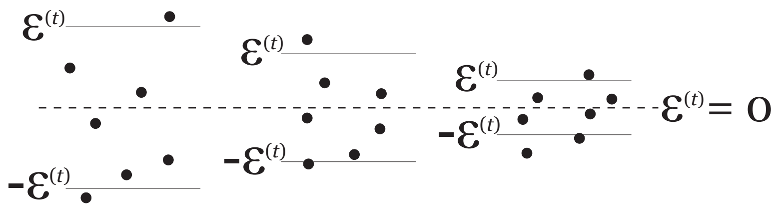


FIGURE 4.2: Progressive reduction of the feasible domain.

computationally intensive. Not all constraints can be easily implemented to be corrected by these operators, which must be tailored to the particular problem [15].

Constraint-preserving methods (feasible solutions methods) reduce the search space by ensuring that all candidate solutions at all times satisfy the constraints [32]. Solutions are initialised within the feasible domain, and transformations of candidate solutions are such that the resulting solutions still lie within the feasible domain.

Hamida and Schoenauer introduced a hybrid approach for Evolutionary Algorithms to handle equality constraints [22]. In this approach, equalities $h_j(\mathbf{x}) = 0$ are written as double inequalities $-\varepsilon^{(t)} \leq h_j(\mathbf{x}) \leq \varepsilon^{(t)}$. The idea is to start, for each equality, with a large feasible domain, and so tolerate high violation degrees. This domain is then gradually reduced along evolution, in order to bring it as close as possible to a null measure feasible domain, as illustrated in Figure 4.2. The value of ε is progressively reduced with the aim of reaching $0 \leq h_j(\mathbf{x}) \leq 0$.

Feasible solutions methods, on the other hand, are based on transforming feasible individuals into other feasible individuals. In the Evolutionary Algorithm sense, it is done by operators that are closed on the feasible part of the search space. These methods assume linear constraints only and a feasible starting point, or a feasible initial population [32].

Michalewicz and Janikow developed a genetic algorithm called *Genocop*, named after "GEnetic algorithm for Numerical Optimisation for COnstrained Problems" [31]. The approach, focusing on linear constrains, firstly eliminates the equalities in the set of constraints, and secondly uses carefully designed 'genetic' operators that guarantee to keep all 'chromosomes' of the genetic algorithm within the constrained space.

Shi and Krohling developed a method using two co-evolving PSOs, and duality from Section 4.1.5, to solve a constrained optimisation problem [49]. The min-max problem (4.16) is solved by evolving two simultaneous PSOs. The first PSO freezes the Lagrange multipliers $\boldsymbol{\mu}$ and $\boldsymbol{\lambda}$, and minimises the Lagrangian $L(\mathbf{x}, \boldsymbol{\mu}, \boldsymbol{\lambda})$ over $\mathbf{x}$. The second PSO freezes the variable vector $\mathbf{x}$, and maximises $L(\mathbf{x}, \boldsymbol{\mu}, \boldsymbol{\lambda})$ over the Lagrange multipliers $\boldsymbol{\mu}$ and $\boldsymbol{\lambda}$. However, if the optimisation problem is non-convex, the solution of the primal and dual problems do not coincide. In this case a penalty, determined by the inequality and equality constraint functions, is added to the Lagrangian.

The LPSO falls in the constraint-preserving class of constraint handling algorithms. Linear constraints are assumed, and if the initial swarm contains only feasible starting points, transitions to new solutions through velocity updates ensure feasible solutions to be generated.

4.3.2 PSO for equality-constrained optimisation

Let the objective be to find the minimum of some function $f(\mathbf{x})$, where $\mathbf{x} \in \mathbb{R}^n$, subject to a set of linear constraints,

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + \ldots + a_{1n}x_n &= b_1 \\ a_{21}x_1 + a_{22}x_2 + \ldots + a_{2n}x_n &= b_2 \\ &\cdots \\ a_{m1}x_1 + a_{m2}x_2 + \ldots + a_{mn}x_n &= b_m \end{aligned}$$

or

$$\begin{bmatrix} a_{11} & a_{12} & \ldots & a_{1n} \\ a_{21} & a_{22} & \ldots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \ldots & a_{mn} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_m \end{bmatrix}$$

It is assumed that the problem is feasible, or the solution set for the linear constraints is non-empty. Then, in simple terms, the problem is defined as

$$\begin{aligned} &\text{Minimise} \quad f(\mathbf{x}), \quad \mathbf{x} \in \mathbb{R}^n \\ &\text{Subject to } A\mathbf{x} = \mathbf{b},\ A \in \mathbb{R}^{m \times n} \text{ and } \mathbf{b} \in \mathbb{R}^m \end{aligned} \tag{4.26}$$

It can be said that f needs to be optimised in the hyperplane C, the set of particular solutions of the linear system $A\mathbf{x} = \mathbf{b}$. That is,

$$C = \{\mathbf{x} \mid A\mathbf{x} = \mathbf{b}\}$$

defines the set of feasible solutions to (4.26), and each point in C will be a feasible point. Figure 4.3 illustrates a one-dimensional hyperplane (or line) C that constrains two-dimensional solutions $\mathbf{x} = (x_1, x_2)$.

The approach presented below flies the swarm through the set of feasible solutions, in this case hyperplane C.

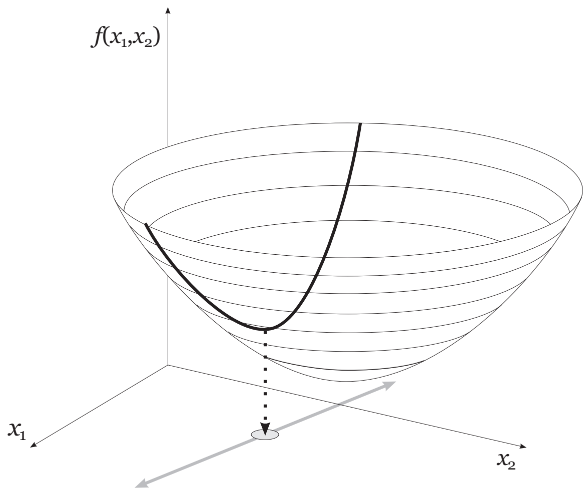


FIGURE 4.3: Minimising f under a linear equality constraint.

Feasible directions

Given a feasible point $\mathbf{x}$ (a particle's position, for instance), it will be necessary to fly from $\mathbf{x}$ to other feasible points. This can be done with feasible directions. Let

$$H = \{\mathbf{x} \mid A\mathbf{x} = \mathbf{0}\}$$

define the set of solutions of the homogeneous system $A\mathbf{x} = \mathbf{0}$. H is a subspace of $\mathbb{R}^n$, and since H is closed under vector addition and scalar multiplication, it is also a vector space. If $\mathbf{c}_0$ is any element of C, then H is defined by C minus some offset $\mathbf{c}_0$, or the set of vectors $C - \mathbf{c}_0 = \{\mathbf{c} - \mathbf{c}_0 \mid \mathbf{c} \in C\}$.

If $\mathbf{x}$ is feasible and $\mathbf{h} \in H$, the point $\mathbf{x} + \alpha\mathbf{h}$ is also feasible for every value of α, since $A(\mathbf{x} + a\mathbf{h}) = A\mathbf{x} + \alpha A\mathbf{h} = \mathbf{b} + \alpha\mathbf{0} = \mathbf{b}$. Any move from a feasible point along $\mathbf{h}$ will produce another feasible point. Any nonzero direction $\mathbf{h} \in H$ is called a *feasible direction* for the constraints $A\mathbf{x} = \mathbf{b}$ in (4.26).

If the initial swarm is feasible, and the particles fly with only feasible directions as their velocity vectors, then the swarm will stay within the search space. This is summarized in Theorem 4.2, which can be proved by a simple inductive argument:

Theorem 4.2

If all initial velocity vectors $\mathbf{v}_i^{(0)}$ are solutions to the homogeneous system $A\mathbf{x} = \mathbf{0}$, and all initial particles $\mathbf{p}_i^{(0)}$ lie in the hyperplane defined by $A\mathbf{x} = \mathbf{b}$, then for any time t

$$\begin{array}{ll} \textit{I)} & A\mathbf{v}_i^{(t)} = \mathbf{0} \\ \textit{II)} & A\mathbf{p}_i^{(t)} = \mathbf{b} \\ \textit{III)} & A\mathbf{z}_i^{(t)} = \mathbf{b} \\ \textit{IV)} & A\widehat{\mathbf{z}}^{(t)} = \mathbf{b} \end{array}$$

i.e. the swarm will fly through the hyperplane defined by the constraints.

Proof

Without losing generality, subscript i, denoting a specific particle in the swarm, is dropped.

Basis step:

I) $\mathbf{v}^{(0)} = \mathbf{0}$ (by initialisation) is the trivial solution to $A\mathbf{x} = \mathbf{0}$

II) $\mathbf{p}^{(0)}$ is initialised on the hyperplane $A\mathbf{x} = \mathbf{b}$

III) $\mathbf{z}^{(0)} = \mathbf{p}^{(0)}$

$\Rightarrow A\mathbf{z}^{(0)} = \mathbf{b}$

IV) $\widehat{\mathbf{z}}^{(0)} \in \{\mathbf{z}_1^{(0)}, \mathbf{z}_2^{(0)}, \ \ldots, \mathbf{z}_s^{(0)}\}$

$\mid f(\widehat{\mathbf{z}}^{(0)}) = \min\{\ f(\mathbf{z}_1^{(0)}), f(\mathbf{z}_2^{(0)}),\ \ldots, f(\mathbf{z}_s^{(0)})\}$

$\Rightarrow A\widehat{\mathbf{z}}^{(0)} = \mathbf{b}$

Inductive step:

Suppose $A\mathbf{v}^{(k)} = \mathbf{0}$, $A\mathbf{p}^{(k)} = \mathbf{b}$, $A\mathbf{z}^{(k)} = \mathbf{b}$ and $A\widehat{\mathbf{z}}^{(k)} = \mathbf{b}$. Then

I)
$$\begin{aligned} & A\mathbf{v}^{(k+1)} \\ &= A\big(w\mathbf{v}^{(k)} + c_1 r_1^{(k)}[\mathbf{z}^{(k)} - \mathbf{p}^{(k)}] + c_2 r_2^{(k)}[\widehat{\mathbf{z}}^{(k)} - \mathbf{p}^{(k)}]\big) \\ &= wA\mathbf{v}^{(k)} + c_1 r_1^{(k)}(A\mathbf{z}^{(k)} - A\mathbf{p}^{(k)}) + c_2 r_2^{(k)}(A\widehat{\mathbf{z}}^{(k)} - A\mathbf{p}^{(k)}) \\ &= wA\mathbf{v}^{(k)} + c_1 r_1^{(k)}(\mathbf{b} - \mathbf{b}) + c_2 r_2^{(k)}(\mathbf{b} - \mathbf{b}) \\ &= w\mathbf{0} + c_1 r_1^{(k)}\mathbf{0} + c_2 r_2^{(k)}\mathbf{0} \\ &= \mathbf{0} \end{aligned}$$

II)
$$\begin{aligned} A\mathbf{p}^{(k+1)} &= A(\mathbf{v}^{(k+1)} + \mathbf{p}^{(k)}) \\ &= A\mathbf{v}^{(k+1)} + A\mathbf{p}^{(k)} \\ &= \mathbf{0} + \mathbf{b} \\ &= \mathbf{b} \end{aligned}$$

III)
$$\begin{aligned} A\mathbf{z}^{(k+1)} &= \begin{cases} A\mathbf{z}^{(k)} & \text{if } f(\mathbf{p}^{(k+1)}) \geq f(\mathbf{z}^{(k)}) \\ A\mathbf{p}^{(k+1)} & \text{if } f(\mathbf{p}^{(k+1)}) < f(\mathbf{z}^{(k)}) \end{cases} \\ &= \mathbf{b} \end{aligned}$$

IV) $\widehat{\mathbf{z}}^{(k+1)} \in \{\mathbf{z}_1^{(k+1)}, \mathbf{z}_2^{(k+1)},\ \ldots, \mathbf{z}_s^{(k+1)}\} \mid f(\widehat{\mathbf{z}}^{(k+1)})$

$= \min\{f(\mathbf{z}_1^{(k+1)}), f(\mathbf{z}_2^{(k+1)}),\ \ldots, f(\mathbf{z}_s^{(k+1)})\}$

$\Rightarrow A\widehat{\mathbf{z}}^{(k+1)} = \mathbf{b}$

$\square$

This shows that the swarm will fly through the solution hyperplane C defined by the set of feasible solutions.

Change of PSO for constrained optimisation

It is clear from the above that, if the swarm is initialised to a set of feasible solutions, all solutions found will be feasible. However, this does not mean that the optimum solution can be found.

Theorem 4.1 provides a condition on the initial swarm that guarantees that any point inside the search space can be found. This search space was $\mathbb{R}^n$. With the given constraints, the search space will be some hyperplane inside $\mathbb{R}^n$. The initial swarm can be chosen such that any point in this hyperplane can be found.

By definition, any direction $\mathbf{h}$ satisfying $A\mathbf{h} = \mathbf{0}$ lies in the null space of A. If the rank of A is r, let

$$S^\star = \{\mathbf{p}_1^{(0)} - \widehat{\mathbf{z}}^{(0)},\ \mathbf{p}_2^{(0)} - \widehat{\mathbf{z}}^{(0)},\ \ldots,\ \mathbf{p}_{n-r}^{(0)} - \widehat{\mathbf{z}}^{(0)}\}$$

denote a generic set of $n - r$ linearly independent vectors, such that $A(\mathbf{p}_i^{(0)} - \widehat{\mathbf{z}}^{(0)}) = A\mathbf{p}_i^{(0)} - A\widehat{\mathbf{z}}^{(0)} = \mathbf{b} - \mathbf{b} = \mathbf{0}$. This implies that $S^\star$ forms a basis for the $n - r$ dimensional null space of A. $S^\star$ provides a convenient way to represent all feasible points. Given any point $\mathbf{p}^{(0)}$ such that $A\mathbf{p}^{(0)} = \mathbf{b}$, every feasible point can be written as $\mathbf{p}^{(0)}$ plus some linear combination of $S^\star$.

For constrained optimisation, f is optimised in an $n - r$ dimensional hyperplane inside $\mathbb{R}^n$, with $r = rank(A)$. Thus a swarm of s particles $S^{(0)} = \{\mathbf{p}_1^{(0)},\ \mathbf{p}_2^{(0)},\ \ldots,\ \mathbf{p}_s^{(0)}\}$ will be able to find the optimal value if and only if there exists a subset $S^\star \subseteq S^{(0)} - \widehat{\mathbf{z}}^{(0)} = \{\mathbf{p} - \widehat{\mathbf{z}}^{(0)} \mid \mathbf{p} \in S^{(0)}\}$ that forms a basis for $\mathbb{R}^{n-r}$. In this case the minimum swarm size will be

$$\inf |S^{(0)}| = n - r + 1 \tag{4.27}$$

If the whole swarm is thus initialised to lie within the hyperplane $A\mathbf{x} = \mathbf{b}$, and $S^\star \subseteq S^{(0)} - \widehat{\mathbf{z}}^{(0)}$ defines a basis for $\mathbb{R}^{n-r}$, then f can be optimised in the standard way. It is due to this property that Linear Particle Swarm Optimisation is ideally suited to solving these kinds of optimisation problems.

Initialising particles within the search plane

The next task is to find a way to initialise such a swarm with s particles. Most importantly, all particles should lie within the search plane. This can be done by reducing the augmented matrix $[A|\mathbf{b}]$ to row-echelon form $[A'|\mathbf{b}']$ with Gauss-Jordan reduction, and choosing vectors in the hyperplane by using this matrix, as summarized below:

Algorithm 4.2 - Initialising particles within the search plane

1. Reduce the augmented matrix $[A|\mathbf{b}]$ to transform the coefficient matrix A of the given constraints to row-echelon form. The number of pivots in this form will be equal to r, the rank of A.

$$[A|\mathbf{b}] \sim [A'|\mathbf{b}'] = \left[\begin{array}{cccccccc|c} 1 & 0 & \dots & 0 & a'_{1r+1} & \dots & a'_{1n} & b'_1 \\ 0 & 1 & \dots & 0 & a'_{2r+1} & \dots & a'_{2n} & b'_2 \\ \vdots & \vdots & \ddots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \dots & 1 & a'_{rr+1} & \dots & a'_{rn} & b'_r \\ 0 & 0 & \dots & 0 & 0 & \dots & 0 & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \dots & 0 & 0 & \dots & 0 & 0 \end{array}\right]$$

(No pivots appear after column r)

2. Use $[A'|\mathbf{b}']$ to generate a total of $n-r$ linearly independent random vectors such that $A\mathbf{p}_i^{(0)} = \mathbf{b}$ for $i = 1 \dots n-r$.

   A random vector $\mathbf{p} = (p_1, p_2, \dots, p_n)^T$ satisfying $A\mathbf{p} = \mathbf{b}$ can be constructed by choosing values for p_k randomly, with $k = r+1, \dots, n$ ($k \in$ *non-pivot columns*). Now for each $j = 1, \dots, r$ ($j \in$ *pivot columns*), let

$$p_j = b'_j - \sum_{k=r+1}^{n} a'_{jk} p_k$$

3. Generate one more vector $\mathbf{p}_{n-r+1}^{(0)} = \sum_{i=1}^{n-r} \frac{1}{n-r}\mathbf{p}_i^{(0)}$. Now,

$$A\mathbf{p}_{n-r+1}^{(0)} = A\sum_{i=1}^{n-r} \frac{1}{n-r}\mathbf{p}_i^{(0)} = \sum_{i=1}^{n-r} \frac{1}{n-r}A\mathbf{p}_i^{(0)} = \sum_{i=1}^{n-r} \frac{1}{n-r}\mathbf{b} = \mathbf{b}$$

   This vector is a combination of all other vectors and is linearly dependent on all vectors $1, \dots, n-r$. Any $S^{(0)} - \widehat{\mathbf{z}}^{(0)}$ will form a basis for $\mathbb{R}^{n-r}$, since subtracting any choice of $\widehat{\mathbf{z}}^{(0)}$ will give a linearly independent set.

4. Choose the initial positions of particles $n-r+2$ to s at random by using the method described in Step 2 to create a swarm of size s.

4.3.3 Overcoming premature convergence

The LPSO algorithm (Algorithm 4.1) discussed above has one property that is very disadvantageous, and that is the possibility of premature convergence.

If $\mathbf{v}^{(0)}$ is initialised to $\mathbf{0}$ and the position of the global best particle does not change, searches will continue on lines connecting each particle with the global best. So the whole hyperplane is not searched, but only lines.

In another scenario, consider $\mathbf{p}_i = \mathbf{z}_i = \widehat{\mathbf{z}}$, where velocity updates will depend only on the value of $w\mathbf{v}_i^{(t)}$, as discussed in [53, 54]. If a particle's current position coincides with the global best position, the particle will only move away from this point if its previous velocity and w are non-zero. Premature convergence will occur when previous velocities are close to zero, and particles stop moving once they catch up with the global best particle.

To overcome this premature convergence, the Guaranteed Convergence Particle Swarm Optimiser (GCPSO) was developed [53, 54]. In this algorithm, the velocity update for the global best particle is changed to force it to search for a better solution in an area around the position of that particle. A convergence proof for the GCPSO, and results to substantiate its success can be found in [53, 54].

The GCPSO cannot be used as given in [53, 54], since unconstrained random adjustments may generate infeasible solutions. A variation is necessary because particles cannot be altered with any random vector, but only with feasible directions. The new algorithm, referred to as Converging LPSO (CLPSO), ensures that the constraints from equation (4.26) are still met.

Let τ be the index of the global best particle, then

$$\mathbf{z}_\tau = \widehat{\mathbf{z}} \tag{4.28}$$

Change the velocity update equation (4.20) for the global best particle τ, so that

$$\mathbf{v}_\tau^{(t+1)} = -\mathbf{p}_\tau^{(t)} + \widehat{\mathbf{z}}^{(t)} + \rho^{(t)}\boldsymbol{v}^{(t)} \tag{4.29}$$

where $\rho^{(t)}$ is a scaling factor and $\boldsymbol{v}^{(t)} \sim \mathit{UNIF}(-1,1)^n$ with the property that $A\boldsymbol{v}^{(t)} = \mathbf{0}$, or $\boldsymbol{v}^{(t)}$ lies in the null space of A. The vector $\boldsymbol{v}^{(t)}$ can be constructed from the reduced augmented matrix $[A'|\mathbf{b}']$, with A in row-echelon form. Such a method is described in Step 2 of Section 4.3.2. Now,

$$\begin{aligned}
A\mathbf{v}_\tau^{(t+1)} &= A\big(-\mathbf{p}_\tau^{(t)} + \widehat{\mathbf{z}}^{(t)} + \rho^{(t)}\boldsymbol{v}^{(t)}\big) \\
&= -A\mathbf{p}_\tau^{(t)} + A\widehat{\mathbf{z}}^{(t)} + \rho^{(t)}A\boldsymbol{v}^{(t)} \\
&= -\mathbf{b} + \mathbf{b} + \mathbf{0} \\
&= \mathbf{0}
\end{aligned}$$

and so the swarm will still fly through the hyperplane as described in Theorem 4.2. Since

$$\begin{aligned}
\mathbf{p}_\tau^{(t+1)} &= \mathbf{v}_\tau^{(t+1)} + \mathbf{p}_\tau^{(t)} \\
&= \widehat{\mathbf{z}}^{(t)} + \rho^{(t)}\boldsymbol{v}^{(t)}
\end{aligned}$$

the new position of the global best particle will be its personal best $\widehat{\mathbf{z}}^{(t)}$, with a random vector $\rho^{(t)}\boldsymbol{v}^{(t)}$ from the null space of A added. It is *only* the global best particle that is moved with the above velocity update (4.29), all other particles in the swarm are still moved with the original equations (4.20) and (4.21).

Removal of initial conditions for CLPSO

Adding random vectors to the algorithm changes the initial conditions: Theorem 4.1 is based on LPSO which does not make any allowance for random changes to particle positions. Since $\boldsymbol{v}^{(t)}$ is random, the condition that some $S^\star \subseteq S^{(0)} - \widehat{\mathbf{z}}^{(0)}$ that defines a basis for $\mathbb{R}^{n-r}$ (with $rank(A) = r$) should exist, can be dropped for CLPSO.

4.3.4 Proof of convergence for CLPSO

To prove the convergence of CLPSO to at least a local minimum, a more general condition for convergence of a random search algorithm is first discussed and proved. Consider the following problem and conceptual algorithm:

P *Given a measurable function $f : \mathbb{R}^n \to \mathbb{R}$ and $\mathcal{S} \subseteq \mathbb{R}^n$. We seek a point $\mathbf{x} \in \mathcal{S}$ which at least finds a local minimum of f on $\mathcal{S}$ or yields an approximation of a local minimum of f on $\mathcal{S}$.*

Algorithm 4.3 - Conceptual algorithm

1. Find $\mathbf{x}^{(0)} \in \mathcal{S}$ and set $k = 0$
2. Generate $\boldsymbol{\xi}^{(k)}$ from $(\mathbb{R}^n, \mathcal{B}, \mu_k)$
3. $\mathbf{x}^{(k+1)} = D(\mathbf{x}^{(k+1)}, \boldsymbol{\xi}^{(k)})$, choose μ_{k+1}, $k := k + 1$, go to step 1

The probability space $(\mathbb{R}^n, \mathcal{B}, \mu_k)$ is such that $\mathcal{B}$ is the σ-field of Borel subsets of $\mathbb{R}^n$, and μ_k is a probability measure on $\mathcal{B}$ such that $\mu_k(\mathbb{R}^n) = 1$. The algorithm starts with an initial solution $\mathbf{x}^{(0)}$, and at each iteration a possible new solution $\boldsymbol{\xi}^{(k)}$ is generated from $(\mathbb{R}^n, \mathcal{B}, \mu_k)$. The function D, explained below, maps $\mathcal{S} \times \mathbb{R}^n$ to $\mathcal{S}$.

It is sufficient to show that if the random search algorithm satisfies two conditions – the *algorithm condition* and the *convergence condition* – then it will at least converge to a local minimum. Each of these condisions are presented below.

Algorithm condition The mapping $D : \mathcal{S} \times \mathbb{R}^n \to \mathcal{S}$ should satisfy $f(D(\mathbf{x}, \boldsymbol{\xi})) \leq f(\mathbf{x})$ and if $\boldsymbol{\xi} \in \mathcal{S}$, then $f(D(\mathbf{x}, \boldsymbol{\xi})) \leq f(\boldsymbol{\xi})$.

Let M_k be the support of μ_k, the smallest closed subset of $\mathbb{R}^n$ with measure of one. Almost all random search algorithms are adaptive, implying that μ_k depends on the solutions $\mathbf{x}^{(0)}, \ldots, \mathbf{x}^{(k-1)}$ generated by the previous iterations of the algorithm. The μ_k are then viewed as conditional probability measures. Let m be the Lebesgue measure of a set. The search method discussed here is a called a *local search method*, which means that the μ_k with bounded support M_k have, for all except a possibly finite k, $m(\mathcal{S} \cap M_k) < m(\mathcal{S})$. Methods called *global search methods* have $m(\mathcal{S} \cap M_k) = m(\mathcal{S})$ for all k.

To avoid having to search for an element in a set of null measure, the search will be for the essential infimum of f. This assures that, for a pathological case like $f(x) = x^2$ for $x \neq 0$, and $f(x) = -1$ for $x = 0$, the true minimum at -1 need not be found, but simply an x for which $f(x)$ is arbitrarily close to zero. Thus the search for the infimum will be replaced by a search for the essential infimum. Define the minimum of f on $\mathcal{S}$ as

$$\alpha = \text{ess inf } f = \sup\{z : f(\mathbf{x}) \geq z \text{ a.e.}\}$$

and assume that α is finite.[1]

Since the nature of the search is for the essential infimum and therefore may preclude the actual minimum, it is necessary to establish convergence to a small region of values surrounding the minimum. Let the *optimality region* for the (global) minimum be defined as

$$R_{\epsilon,0} = \{\mathbf{x} \in \mathcal{S} : f(\mathbf{x}) < \alpha + \epsilon\}$$

Function f has an essential local minimum at $\mathbf{c}_i \in \mathcal{S}$ if there exists an n-dimensional interval $I_i \subseteq \mathcal{S}$ around $\mathbf{c}_i$, such that $f(\mathbf{c}_i) \leq f(\mathbf{x})$ a.e. for all $\mathbf{x} \in I_i$. For each of the (possibly infinite) local minima $\mathbf{c}_i$ with $i \geq 1$, define the optimality region (that is sufficient for the search algorithm to find) as

$$R_{\epsilon,i} = \{\mathbf{x} \in I_i : f(\mathbf{x}) < f(\mathbf{c}_i) + \epsilon\}$$

Now let $R_\epsilon = \bigcup_i R_{\epsilon,i}$ be the optimality region for problem $\mathbf{P}$.

Convergence condition *Sufficient condition for convergence to at least a local minimum (of a local search algorithm): For any $\mathbf{x}^{(k)} \in \mathcal{S}$, there exists a $\gamma > 0$ and a $0 < \eta \leq 1$ such that*

$$\mu_k\big(f(\mathbf{x}^{(k+1)}) \leq f(\mathbf{x}^{(k)}) - \gamma \text{ or } \mathbf{x}^{(k)} \in R_\epsilon\big) \geq \eta \tag{4.30}$$

[1]Thus the minimum is defined as the supremum of all z values such that f is greater than or equal to z *almost everywhere* (a.e.), i.e. everywhere except possibly on some null set. Letting $\alpha = -\infty$ will not alter the spirit of the algorithm, if a very large negative value is taken as a sufficient 'approximation' of $-\infty$.

Proof Take the complement of (4.30) to get

$$\mu_k\big(f(\mathbf{x}^{(k+1)}) > f(\mathbf{x}^{(k)}) - \gamma \text{ and } \mathbf{x}^{(k)} \notin R_\epsilon\big) \leq 1 - \eta$$

for all $\gamma > 0$.

From the definition of D, $f(\mathbf{x}^{(k+1)}) > f(\mathbf{x}^{(k)}) - \gamma$ for all $\gamma > 0$ is not possible, and so

$$\mu_k(\mathbf{x}^{(k)} \notin R_\epsilon) \leq 1 - \eta$$

Let $\{\mathbf{x}^{(k)}\}_{k\geq 0}$ be the sequence generated by D. Therefore it needs to be shown that $\lim_{k\to\infty} P(\mathbf{x}^{(k)} \in R_\epsilon) = 1$. Define A to be the event that $\mathbf{x}^{(k)} \in R_\epsilon$ before iteration p. Then,

$$\begin{aligned} P(A) &= 1 - P(\bar{A}) \\ &= 1 - \prod_{i=0}^{p-1} \mu_i(\mathbf{x}^{(i)} \notin R_\epsilon) \\ &\geq 1 - (1-\eta)^p \end{aligned}$$

and so $P(A) \to 1$ as $p \to +\infty$.

To complete the proof, consider the case when $\mathbf{x}^{(p)} \in R_\epsilon$, and $\mu_p(f(\mathbf{x}^{(p+1)}) \leq f(\mathbf{x}^{(p)}) - \gamma) > 0$. Then there is a positive probability that $\mathbf{x}^{(p+1)} \notin R_\epsilon$, and if that is the case, the above argument assures us that $\mathbf{x}^{(k)}$ will converge to R_ϵ once again. From the definition of R_ϵ and D, this will be to a local or possibly global minimum less than $\mathbf{x}^{(p)}$. (When $\mu_p(f(\mathbf{x}^{(p+1)}) \leq f(\mathbf{x}^{(p)}) - \gamma) = 0$, the sequence will remain in R_ϵ at a local or the global minimum.)

□

To prove that CLPSO converges at least to a local minimum, and does not stagnate and converge prematurely, it needs to be shown that both the *algorithm condition* and the *convergence condition* defined above will hold. Let $\mathcal{S} = \mathbb{R}^n$.

Algorithm condition The global best $\widehat{\mathbf{z}}^{(t)}$ is set to the position of the best performance in the swarm, i.e.

$$\begin{aligned} &\widehat{\mathbf{z}}^{(t)} \in \{\mathbf{z}_1^{(t)}, \mathbf{z}_2^{(t)}, \ldots, \mathbf{z}_s^{(t)}\} \;\big|\; f(\widehat{\mathbf{z}}^{(t)}) \\ &= \max\{f(\mathbf{z}_1^{(t)}), f(\mathbf{z}_2^{(t)}), \ldots, f(\mathbf{z}_s^{(t)})\} \end{aligned}$$

and

$$\mathbf{z}_i^{(t)} = \begin{cases} \mathbf{z}_i^{(t-1)} & \text{if } f(\mathbf{p}_i^{(t)}) \geq f(\mathbf{z}_i^{(t-1)}) \\ \mathbf{p}_i^{(t)} & \text{if } f(\mathbf{p}_i^{(t)}) < f(\mathbf{z}_i^{(t-1)}) \end{cases}$$

The above update equations imply that the *algorithm condition* holds.

Convergence condition Particle update equations are

$$\mathbf{p}_i^{(t+1)} = \mathbf{p}_i^{(t)} + w\mathbf{v}_i^{(t)} + c_1 r_1^{(t)}[\mathbf{z}_i^{(t)} - \mathbf{p}_i^{(t)}] + c_2 r_2^{(t)}[\widehat{\mathbf{z}}^{(t)} - \mathbf{p}_i^{(t)}]$$

and for the global best particle

$$\mathbf{p}_\tau^{(t+1)} = \widehat{\mathbf{z}}^{(t)} + \rho^{(t)}\boldsymbol{v}^{(t)}$$

Sampling a new point (that might be better than $\widehat{\mathbf{z}}^{(t)}$) will be done for each of s particles, and thus we will define M_t, the support for μ_t at iteration t, as the set from which each of these s values can be picked. For each particle $\mathbf{p}_i$ (except for the global best particle τ) define $M_{t,i}$ as the convex hull defined by $\big(\mathbf{p}_i^{(t)}\big)$, $\big(\mathbf{p}_i^{(t)} + w\mathbf{v}_i^{(t)}\big)$, $\big(\mathbf{p}_i^{(t)} + c_1[\mathbf{z}_i^{(t)} - \mathbf{p}_i^{(t)}]\big)$, and $\big(\mathbf{p}_i^{(t)} + c_2[\widehat{\mathbf{z}}^{(t)} - \mathbf{p}_i^{(t)}]\big)$.

Since $r_1^{(t)}, r_2^{(t)} \sim \mathit{UNIF}(0,1)$, the new particle $\mathbf{p}_i^{(t+1)}$ will lie within $M_{t,i}$. Also define $M_{t,\tau}$ as the n-dimensional hypercube with sides of length $\rho^{(t)}$, centered at $\widehat{\mathbf{z}}^{(t)}$. Let

$$M_t = \bigcup_{i=1}^{s} M_{t,i}$$

be the support of probability measure μ_t. Since $M_{t,\tau} \subseteq M_t$ a point arbitrarily close to $\widehat{\mathbf{z}}^{(t)}$ can be chosen, and hence there is always a $\gamma > 0$ and $0 < \eta \leq 1$ such that

$$\mu_t\big(f(\widehat{\mathbf{z}}^{(t+1)}) \leq f(\widehat{\mathbf{z}}^{(t)}) - \gamma \text{ or } \widehat{\mathbf{z}}^{(t)} \in R_\epsilon\big) \geq \eta$$

□

4.4 Inequality-constrained optimisation

Inequality-constrained optimisation problems can be reduced to problems involving only non-negativity constraints on a set of variables. In Section 4.1.3 the notion of slack variables, where a standard optimisation problem is converted to one where all inequalities involve only a single variable, was introduced. The LPSO, and consequently the CLPSO as well, are expanded to handle non-negativity constraints on a set of variables. As the aim of the CLPSO is (in the context of this thesis) to solve a SVM's constrained optimisation problem, the method explained below focuses on box constraints of the form $a \leq x_j \leq b$. These constraints force the particles to only fly inside a n-dimensional hypercube, but the method developed will work equally well if no upper bound on the variables existed.

Consider the way a particle $\mathbf{p}_i$ is being updated:

$$\mathbf{p}_i^{(t+1)} = \mathbf{v}_i^{(t+1)} + \mathbf{p}_i^{(t)} \tag{4.31}$$

In the above equation, it is also assumed that $\mathbf{p}_i^{(t)}$ lies inside the problem's feasible region Ω. That is, inside the n-dimensional hypercube. For notational convenience, the subscript i will be dropped. That is,

$$\mathbf{p}^{(t)} = (p_1^{(t)}, p_2^{(t)}, \ldots, p_n^{(t)})^T \tag{4.32}$$

For the above particle, for all values $p_j^{(t)}$ it will be true that $a \leq p_j^{(t)} \leq b$. However, when the velocity vector $\mathbf{v}^{(t+1)}$ is added, it may become true that a value of $p_j^{(t+1)}$ may violate these constraints.

In this case, the velocity vector needs to be scaled so that all values $p_j^{(t+1)}$ will fall inside the constraints. To scale the velocity vector, a scale factor is computed for each $p_j^{(t+1)}$ that lies outside of the constraints. This factor will scale the vector element to lie exactly on the bound. Since the scale factor of one element may scale other elements to lie outside of the bounds, the minimum of all these scale factors are taken to scale the velocity vector. Using this simple technique, the movement of the particles are restricted to the hypercube.

As an example, let $a = 0$ and $b = 2$ such that $0 \leq p_j^{(t)} \leq 2$ in the following position vector, and consider the addition of a velocity vector:

$$\begin{array}{rcccccccl} \mathbf{p}^{(t)} = (& \frac{1}{8} & \frac{1}{8} & \frac{6}{8} & 0 & 0 & \frac{7}{8} & \frac{1}{8} &)^T \\ \mathbf{v}^{(t+1)} = (& 0 & 0 & -\frac{8}{8} & 0 & 0 & \frac{10}{8} & \frac{18}{8} &)^T \\ \mathbf{p}^{(t+1)} = (& \frac{1}{8} & \frac{1}{8} & -\frac{2}{8} & 0 & 0 & \frac{17}{8} & \frac{19}{8} &)^T \\ & & & < 0 & & & > 2 & > 2 & \end{array}$$

It is clear that the new particle lies outside the $[0,2]^7$ hypercube. For scaling, a value δ needs to be found such that $\mathbf{p}^{(t+1)} = \delta \mathbf{v}^{(t+1)} + \mathbf{p}^{(t)}$ will lie inside these constraints. This δ must be chosen such that $p_3^{(t+1)}$, which is smaller than $a = 0$, will now satisfy $p_3^{(t+1)} \geq 0$. The value of δ must also enforce $p_6^{(t+1)} \leq 2$ and $p_7^{(t+1)} \leq 2$.

Continuing the example, δ is computed for each violating dimension. The value of $p_3^{(t+1)}$ is $-\frac{2}{8}$, but it should ideally be 'cut' to lie within its closest boundary, zero. Substituting zero for $p_3^{(t+1)}$ gives the scaling factor δ with which the velocity vector should be scaled to achieve this ideal value:

$$\begin{aligned} p_3^{(t+1)} &= \delta_3 v_3^{(t+1)} + p_3^{(t)} \\ \delta_3 &= \big(p_3^{(t+1)} - p_3^{(t)}\big)/v_3^{(t+1)} \\ &= \big(0 - \frac{6}{8}\big)/\big(-\frac{8}{8}\big) = \frac{6}{8} \end{aligned} \tag{4.33}$$

Similarly, the value for $p_6^{(t+1)}$ is $\frac{17}{8}$, but should ideally be scaled down to two, to lie within its closest border:

$$\begin{aligned} \delta_6 &= \big(p_6^{(t+1)} - p_6^{(t)}\big)/v_6^{(t+1)} \\ &= \big(2 - \frac{7}{8}\big)/\big(\frac{10}{8}\big) = \frac{9}{10} \end{aligned} \tag{4.34}$$

The value for $p_7^{(t+1)}$ is $\frac{19}{8}$, but should also be scaled down to two, to lie within its closest border:

$$\begin{aligned}\delta_7 &= \big(p_7^{(t+1)} - p_7^{(t)}\big)/v_7^{(t+1)} \\ \delta_7 &= \big(2 - \frac{1}{8}\big)\ \big(\frac{18}{8}\big) = \frac{15}{18}\end{aligned} \tag{4.35}$$

From these possible scale values that were computed in (4.33), (4.34), and (4.35), the smallest δ is chosen to scale the velocity vector with. Thus the value of δ will be $\frac{6}{8}$. Multiplying δ with $\mathbf{v}^{(t+1)}$ and updating the particle gives a new position $\mathbf{p}^{(t+1)}$ that lies exactly within the constraints.

$$\begin{aligned}\mathbf{p}^{(t)} &= (\ \tfrac{1}{8}\ \tfrac{1}{8}\ \ \tfrac{6}{8}\ \ 0\ 0\ \tfrac{7}{8}\ \ \tfrac{1}{8}\ )^T \\ \delta\mathbf{v}^{(t+1)} &= (\ 0\ \ 0\ -\tfrac{6}{8}\ 0\ 0\ \tfrac{15}{16}\ \tfrac{27}{16}\ )^T \\ \mathbf{p}^{(t+1)} &= (\ \tfrac{1}{8}\ \tfrac{1}{8}\ \ 0\ \ 0\ 0\ \tfrac{29}{16}\ \tfrac{29}{16}\ )^T\end{aligned}$$

From the above example, an algorithm to keep a swarm of particles within an n-dimensional hypercube $[a,b]^n$, can be generalised.

Algorithm 4.4 - Satisfying inequality constraints

1. Determine the new position that a particle will fly to (but do not move it there)

$$\mathbf{p}^{(t+1)} = \mathbf{v}^{(t+1)} + \mathbf{p}^{(t)}$$

2. For each dimension j in the new position that lies outside $[a,b]^n$, compute a scaling factor δ_j

$$\begin{aligned}\delta_j &= \big(a - p_j^{(t)}\big)/v_j^{(t+1)} \text{ if } p_j^{(t+1)} < a \\ \delta_j &= \big(b - p_j^{(t)}\big)/v_j^{(t+1)} \text{ if } p_j^{(t+1)} > b\end{aligned}$$

   Note that, since $p_j^{(t)} \in [a,b]$ and $p_j^{(t+1)} \notin [a,b]$, the value of δ will always be positive.

3. Set $\delta = \min\{\delta_j \mid p_j^{(t+1)} \notin [a,b]\}$

4. Finally, move the particle to the new position with

$$\mathbf{p}^{(t+1)} = \delta\mathbf{v}^{(t+1)} + \mathbf{p}^{(t)}$$

   to lie within the constrained hypercube $[a,b]^n$.

The method described above in Algorithm 4.3 is used and experimentally verified as part of the CLPSO used for training Support Vector Machines.

It is now possible to fly the swarm such that both linear and bounded constraints are always met. However, the above approach of ‘cutting against the borders’ induces a new hurdle that the LPSO has to overcome.

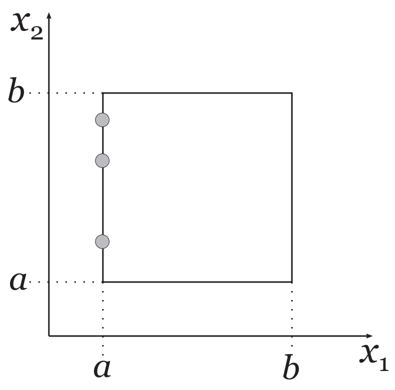


FIGURE 4.4: Particles becoming a linear combination of each other.

The LPSO requires that the set of vectors created by subtracting the particle's current position from the global best solution vector, together with the swarm's set of velocity vectors, must span the entire search space. If all particles are 'cut' against a single constraint (say a in $a \leq p_j \leq b$, as shown in Figure 4.4), the particle positions may all become linear combinations of each other, and if the global best also lies on the specific constraint, the property of spanning the search space will be lost. This problem can be remedied by randomly scattering the swarm, or adding a random vector to each particle to move its current position to the inside of the box constraints, when no improvement is made in the objective function for a fixed number of iterations.

Due to the way the global best particle is moved in CLPSO, a random vector is always added to a position in the swarm. The random vector ensures that, with a probability greater than zero for each iteration, that the global best particle will be moved away from the bound to be inside (a, b).

4.5 Concluding

In this chapter the original form of the PSO algorithm was extended to solving constrained optimisation problems. Two new PSO algorithms were developed. The Linear PSO (LPSO) makes it possible to traverse a search space as a hyperplane, and conditions for LPSO to reach any point within the search space were rigorously analysed. LPSO does however make allowance for premature convergence. To remedy the problem of premature convergence, the Converging LPSO (CLPSO) was developed. A formal proof of CLPSO convergence was given. Finally, a method of handling inequality (box) constraints was presented.

Experimental results follow in the next chapter, and illustrate LPSO and CLPSO on a number of problems, as well as their performance as an optimiser in Support Vector Machine training.

Chapter 5

Experimental results

The purpose of the following chapter, presenting experimental results, is twofold: The convergence of Linear PSO (LPSO) and Converging LPSO (CLPSO) is tested, and the CLPSO is implemented as the constrained optimisation algorithm that is used in training a Support Vector Machine (SVM).

Experimental results are shown to illustrate the differences between the LPSO and the CLPSO in linearly minimising constrained functions. The minima found by these two PSOs are compared for correctness against the minima found by a genetic algorithm implementation, called Genocop II.

As a conclusion, the CLPSO is used in the SVM training algorithm, defined in Section 2.4. The algorithm is empirically compared against two standard SVM training methods, namely decomposition and sequential minimal optimisation.

5.1 Linear Particle Swarm Optimiser

5.1.1 Experimental results

In order to test the performance of LPSO and CLPSO to minimising problems constrained by a set of linear constraints $A\mathbf{x} = \mathbf{b}$, let

$$A = \begin{bmatrix} 0 & -3 & -1 & 0 & 0 & 2 & -6 & 0 & -4 & -2 \\ -1 & -3 & -1 & 0 & 0 & 0 & -5 & -1 & -7 & -2 \\ 0 & 0 & 1 & 0 & 0 & 1 & 3 & 0 & -2 & 2 \\ 2 & 6 & 2 & 2 & 0 & 0 & 4 & 6 & 16 & 4 \\ -1 & -6 & -1 & -2 & -2 & 3 & -6 & -5 & -13 & -4 \end{bmatrix}, \quad \mathbf{b} = \begin{bmatrix} 3 \\ 0 \\ 9 \\ -16 \\ 30 \end{bmatrix} \tag{5.1}$$

Defining matrix A and vector $\mathbf{b}$ in the above way gives a set of constraints for testing ten-dimensional functions.

In all experiments the inertia weight w was set to 0.7, while the values of c_1 and c_2 were set to 1.4. The choice is due to [17], where it is shown that parameter settings close to these ($w = 0.7298$ and $c_1 = c_1 = 1.49618$) give acceptable results. The value of $\rho^{(t)}$ was kept constant at 1.

The correctness of the results are tested against those found by *Genocop II*, a genetic algorithm for optimising constrained problems [31]. Experiments on *Genocop II* are done in a twofold manner:

1. A good minimum is needed against which comparisons can be made. In each case a good minimum for each constrained function was found by evolving the genetic algorithm with a population size of 100, for a total of 4000 generations.
2. For purposes of comparison with LPSO and CLPSO, *Genocop II* was also evolved with the same number of chromosomes (particles) and generations (iterations) as LPSO and CLPSO.

In the following experimental results, the 'good minimum' found by *Genocop II* (with a population size of 100 and after 4000 generations) is indicated first. After the good minimum is shown, the simulations used for comparison with LPSO and CLPSO are discussed.

Test 1

The first function tested, f_1, is a second order polynomial (parabolic) function. For purposes of testing the free dimensions were randomly initialised in the interval $[-100, 100]$. The problem is defined as

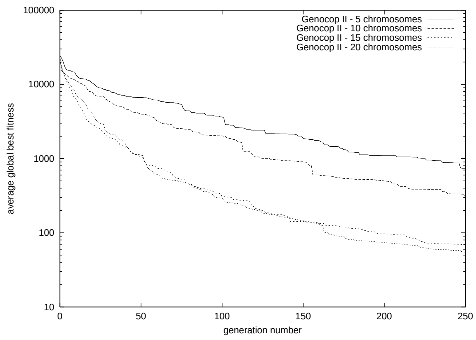
$$\begin{aligned} \text{Minimise } & f_1(\mathbf{x}) = \textstyle\sum_i x_i^2, \ \mathbf{x} \in \mathbb{R}^{10} \\ \text{Subject to } & \quad A\mathbf{x} = \mathbf{b} \end{aligned} \tag{5.2}$$

where A and $\mathbf{b}$ are defined in equation (5.1).

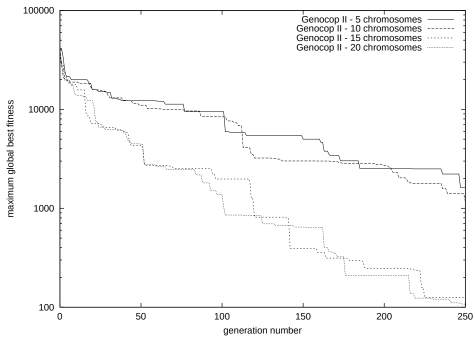
Genocop II The best solution found by *Genocop II*, with a population size of 100 and 4000 generations, was $f_1(\mathbf{x}^\star) = 32.137$ with

$$\mathbf{x}^\star = (0.567, -0.487, 1.736, -1.181, -3.404, 3.357, 0.9, -1.795, -0.528, 0.075)^T$$

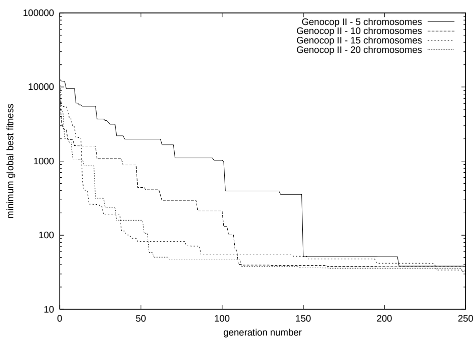
Genocop II was evolved for a total of 250 generations, for population sizes of 5, 10, 15, and 20 chromosomes. The average convergence over 100 simulations is shown in Figure 5.1(a). The average is determined over the best fitness values at a specific generation, over all simulations. The maximum and minimum values over all simulations are computed in a similar fashion, and are shown in Figures 5.1(b) and 5.1(c) respectively. The decreasing



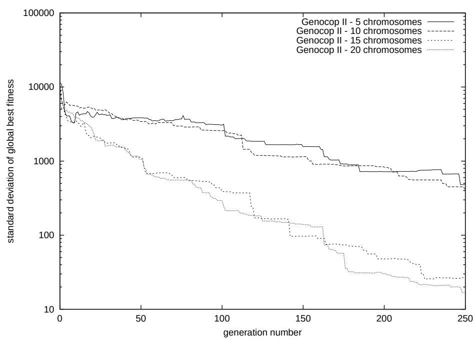
(a) Average



(b) Maximum



(c) Minimum

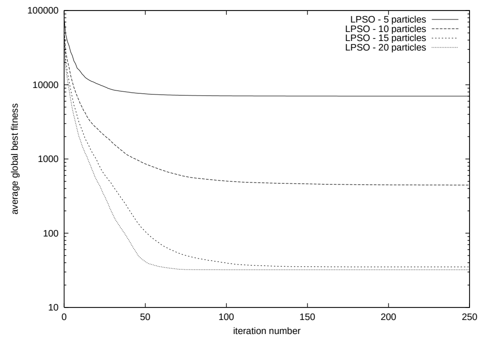


(d) Standard Deviation

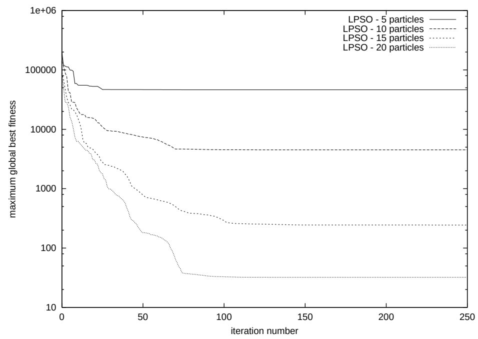
FIGURE 5.1: Results of 100 *Genocop II* simulations on the constrained parabola f_1 defined in equation (5.2).

TABLE 5.1: Results of 100 *Genocop II* simulations on the constrained parabola f_1 defined in equation (5.2), after 250 generations. ('chromosomes' is abbreviated as chrms.)

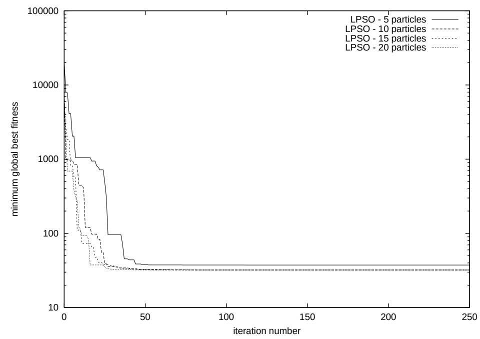
Genocop II	5 chrms.	10 chrms.	15 chrms.	20 chrms.
Average	739.438	304.884	69.154	54.846
Maximum	1.626×10^3	1.168×10^3	124.820	107.584
Minimum	38.322	37.612	33.837	32.544
Standard Deviation	840.279	387.746	26.749	16.939



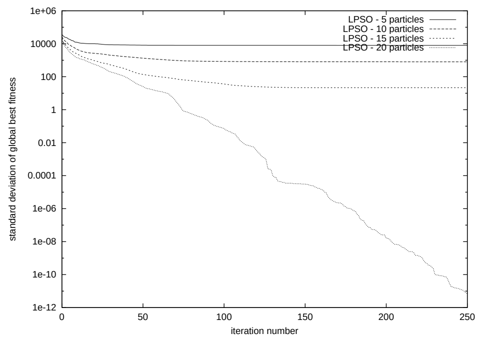
(a) Average



(b) Maximum



(c) Minimum



(d) Standard Deviation

FIGURE 5.2: Results of 100 simulations of LPSO on the constrained parabola f_1 defined in equation (5.2).

maximum (worst) performances give a clear indication that the genetic algorithm converges for all simulations. The standard deviation of the best fitness values over 100 simulations is shown in Figure 5.1(d). These results are summarised in Table 5.1, and are compared to the PSO under the CLPSO results.

LPSO Figure 5.2 shows the convergence of LPSO over 250 iterations, or time steps, of the LPSO algorithm. The results are taken from a total of 100 simulations on swarm sizes of 5, 10, 15, and 20. The average at a specific iteration is computed over the 100 *gbest* values at that specific iteration number, and the averages over all iterations are illustrated in Figure 5.2(a). The maximums and minimums are computed in a similar way, with the maximum being the largest of the 100 *gbest* values at a specific iteration, and the minimum being the smallest of the 100 *gbest* values at a specific iteration. This is shown in Figures 5.2(b) and 5.2(c). The standard deviation of all the LPSO's *gbest* values at a certain time

(Figure 5.2(d)), shows the similarity in the convergence of the 100 swarms.

The average LPSO results for each set of simulations are completely different, and illustrates how the swarms of particles catch up with the global best particle, and converges to a sub-optimal solution. Only a swarm of 20 particles were able to converge to the optimal solution during each simulation. This is illustrated by comparing the standard deviations of each set of simulations's *gbest* values. After 250 iterations (time steps), the standard deviation the 20-particle LPSO's *gbest* is only 7.176×10^{-12}, implying that all swarms converged to the optimal solution. The standard deviations of swarms with 5, 10, and 15 particles are substantially larger, implying that the swarms converged to different solutions, with the variance in convergence increasing as the swarm size decreases. This is the expected result, due to particles catching up and converging to the global best solution [53]. The results after 250 iterations are shown in Table 5.2.

The large average *gbest* of 7.034×10^3 for a swarm of 5 particles – compared to the averages of 10, 15, and 20 particles – is also expected. The minimum number of particles needed to ensure that the swarm spans the entire search space, is $\inf |S^{(0)}| = n - r + 1 = 10 - 5 + 1 = 6$ (refer to equation (4.27)). Consequently, a swarm with 5 particles cannot possibly span the entire search space, which explains the large average *gbest*.

CLPSO The results of CLPSO over 250 time steps are shown in Figure 5.3, with the averages, maximums, minimums, and standard deviations computed in the same way as was done with the LPSO above.

The CLPSO simulations (for 5, 10, 15, and 20 particles) all converged on average to the minimum, or a value close to it. The minimum solution found was

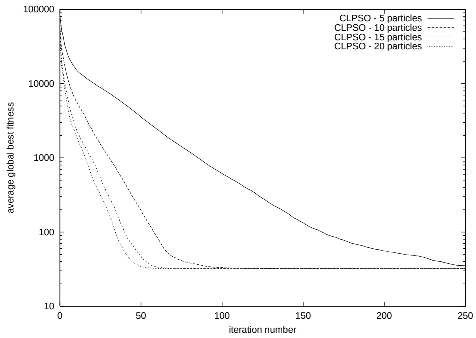
$$\mathbf{x}^\star = (0.566, -0.485, 1.738, -1.181, -3.402, 3.357, 0.9, -1.795, -0.528, 0.074)^T$$

with

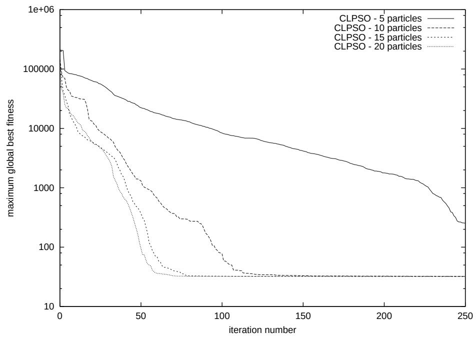
$$f_1(\mathbf{x}^\star) = 32.137$$

The rate of convergence is higher for larger swarms. Figure 5.3(a) shows how the speed of convergence increases as the swarm size grows from 5 to 10, 15, and 20 particles. The standard deviations in Table 5.2 show that there is a very small variance in the *gbest* found by each swarm in the different sets of simulations, indicating that all swarms were close to or at the minimum solution after 250 time steps.

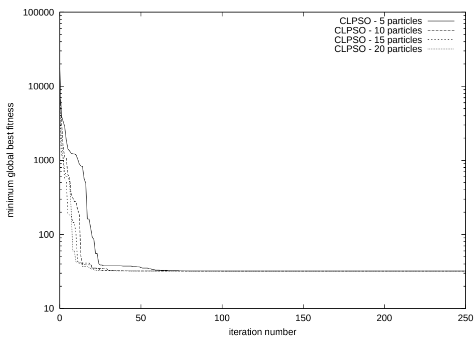
Since the initial condition (refer to equation (4.27)) on a swarm is dropped for the CLPSO, a swarm of 5 particles also searched the entire search space and found the minimum. This can be seen by comparing the average and minimum of a 5-particle swarm in Table 5.2. The difference between LPSO and CLPSO can be clearly seen when Figures 5.2(a) and



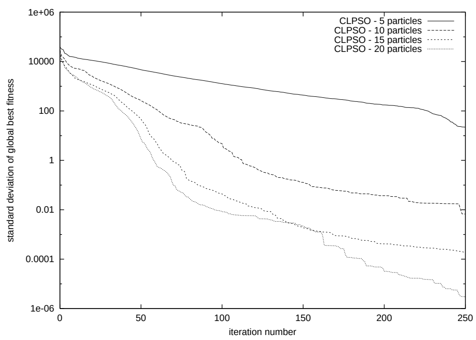
(a) Average



(b) Maximum



(c) Minimum



(d) Standard Deviation

FIGURE 5.3: Results of 100 simulations of CLPSO on the constrained parabola f_1 defined in equation (5.2).

TABLE 5.2: Results of 100 LPSO and CLPSO simulations on the constrained parabola f_1 defined in equation (5.2), after 250 iterations.

LPSO	5 particles	10 particles	15 particles	20 particles
Average	7.034×10^3	445.316	35.071	32.137
Maximum	4.630×10^4	4.505×10^3	244.077	32.137
Minimum	37.420	32.137	32.137	32.137
Standard Deviation	8.007×10^3	803.006	21.500	7.176×10^{-12}
CLPSO	**5 particles**	**10 particles**	**15 particles**	**20 particles**
Average	35.197	32.139	32.137	32.137
Maximum	252.826	32.183	32.138	32.137
Minimum	32.138	32.137	32.137	32.137
Standard Deviation	22.132	6.689×10^{-3}	1.832×10^{-4}	3.016×10^{-6}

5.3(a) are compared. The CLPSO converges on average to the minimum; the LPSO shows premature convergence for smaller swarm sizes, since when the global best does not improve over a large number of iterations, the swarm catches up with it. Note that the probability of finding better solutions increase with LPSO swarm size, and thus the probability of convergence also increases.

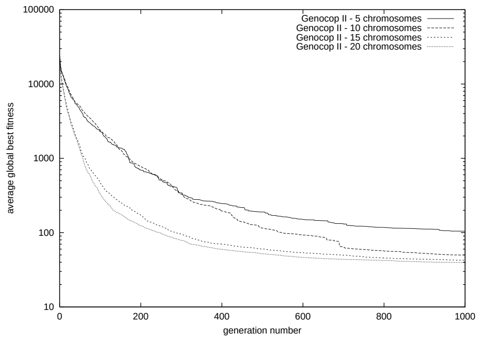
In comparison to *Genocop II*, the CLPSO has a substantially smaller standard deviation of *gbest* values at iteration 250. This is due to the fact that CLSPO has already converged, while *Genocop II* has, for the larger part of simulations, not yet converged to the minimum.

Test 2

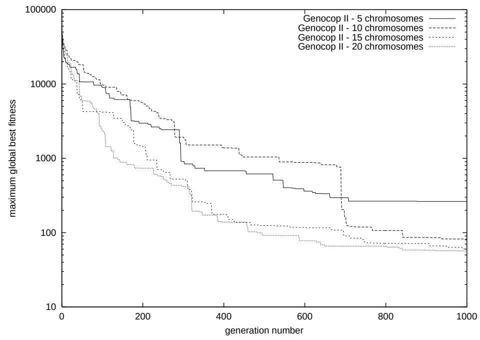
Function f_2 is a quadratic function similar to those commonly found in quadratic programming problems. This function was chosen because it is also similar to the dual Lagrangian optimised in SVM training. Again, the free dimensions were randomly initialised in the interval $[-100, 100]$. The problem is defined as

$$\begin{array}{ll} \text{Minimise} & f_2(\mathbf{x}) = \sum_i \sum_j e^{-(x_i - x_j)^2} x_i x_j + \sum_i x_i,\ \mathbf{x} \in \mathbb{R}^{10} \\ \text{Subject to} & A\mathbf{x} = \mathbf{b} \end{array} \tag{5.3}$$

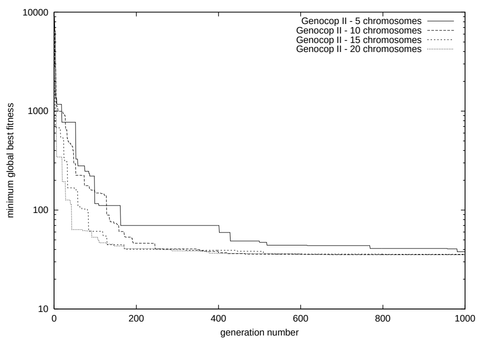
where A and $\mathbf{b}$ are defined in equation (5.1).



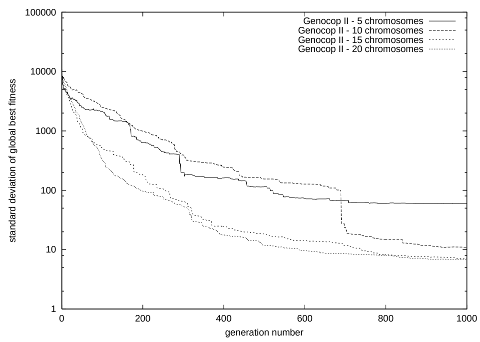
(a) Average



(b) Maximum



(c) Minimum



(d) Standard Deviation

FIGURE 5.4: Results of 100 *Genocop II* simulations on the constrained quadratic function f_2 defined in equation (5.3).

TABLE 5.3: Results of 100 *Genocop II* simulations on the constrained quadratic function f_2 defined in equation (5.3), after 1000 generations. ('chromosomes' is abbreviated as chrms.)

Genocop II	5 chrms.	10 chrms.	15 chrms.	20 chrms.
Average	104.192	49.945	42.393	39.500
Maximum	262.656	82.221	60.110	56.613
Minimum	37.939	35.393	35.772	35.410
Standard Deviation	59.873	10.996	6.861	6.785

Genocop II The best solution found by *Genocop II*, with a population size of 100 and 4000 generations, was $f_2(\mathbf{x}^\star) = 35.377$ with

$$\mathbf{x}^\star = (0.076, -0.28, 0.446, -0.373, -3.956, 3.762, 1.119, -1.865, -0.539, 0.178)^T$$

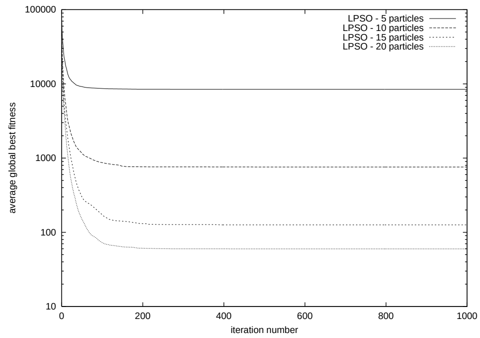
Genocop II was evolved for a total of 1000 generations, for population sizes of 5, 10, 15, and 20 chromosomes. The averages, maximums, minimums and standard deviations over 100 simulations are shown in Figure 5.4, and are computed in the same way as Test 1. Again, these results are summarised in Table 5.3, and are compared to the PSO under the CLPSO results.

LPSO The results of LPSO over 1000 time steps are shown in Figure 5.5, with the averages, maximums, minimums, and standard deviations computed in the same way as explained in Test 1 above.

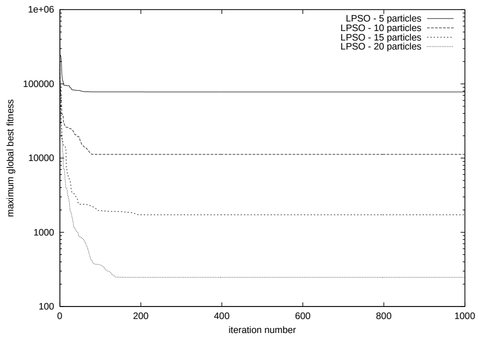
The averages, maximums and standard deviations illustrate the same behaviour as the results in Test 1 (optimising the constrained f_1 with LPSO). It is worthwhile to note that the minimum found by the LPSO, as seen in Figure 5.5(c) and Table 5.4, is the true minimum, except for the 5-particle case. This again illustrates that the LPSO's 5 particles do not span the entire search space, which is 6-dimensional.

CLPSO The results of CLPSO over 1000 time steps are shown in Figure 5.6, with the averages, maximums, minimums, and standard deviations computed in the same way as explained in Test 1 above.

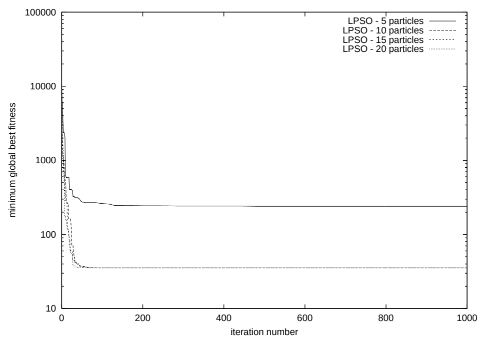
It is clear from Figure 5.6(a) that, after 1000 iterations, the CLSPO is still converging. After 2000 iterations (not shown in the figures), the average *gbest* values were 76.677 for 5 particles, 66.084 for 10 particles, 56.731 for 15 particles, and 39.537 for 20 particles. The averages after 2000 iterations are all smaller than the averages at 1000 generations, shown in Table 5.4.



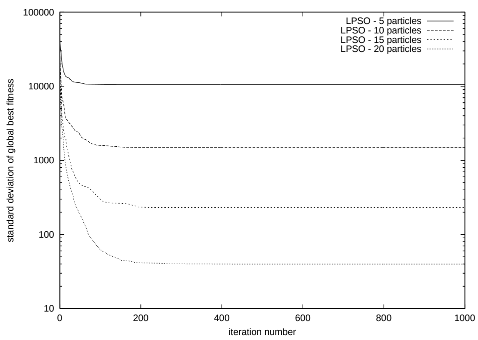
(a) Average



(b) Maximum

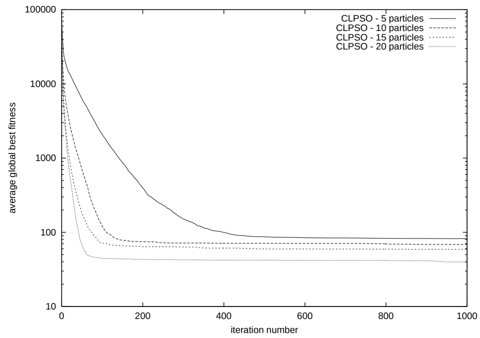


(c) Minimum

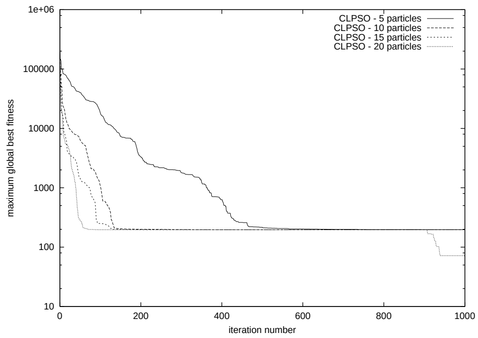


(d) Standard Deviation

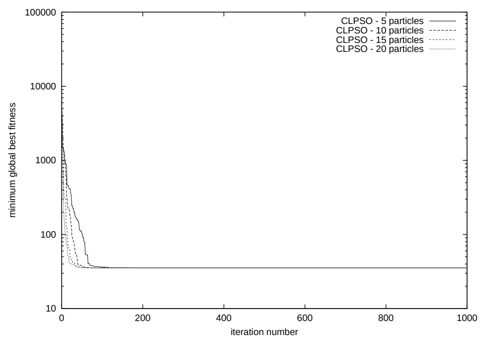
FIGURE 5.5: Results of 100 simulations of LPSO on the constrained quadratic function f_2 defined in equation (5.3).



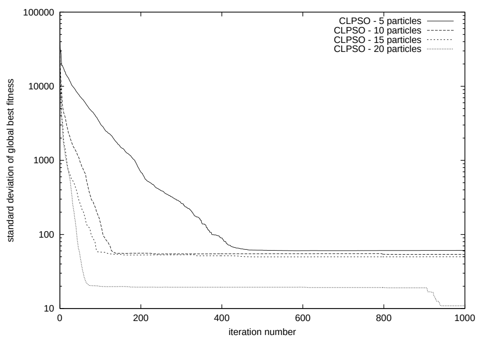
(a) Average



(b) Maximum



(c) Minimum



(d) Standard Deviation

FIGURE 5.6: Results of 100 simulations of CLPSO on the constrained quadratic function f_2 defined in equation (5.3).

TABLE 5.4: Results of 100 LPSO and CLPSO simulations on the constrained quadratic function f_2 defined in equation (5.3), after 1000 iterations.

LPSO	5 particles	10 particles	15 particles	20 particles
Average	8.463×10^3	758.525	125.727	59.762
Maximum	7.793×10^4	1.123×10^4	1.719×10^3	246.905
Minimum	240.101	35.400	35.377	35.377
Standard Deviation	1.051×10^4	1.496×10^3	231.095	39.831
CLPSO	**5 particles**	**10 particles**	**15 particles**	**20 particles**
Average	82.077	68.570	59.001	39.832
Maximum	197.389	196.067	196.065	71.380
Minimum	35.377	35.377	35.377	35.377
Standard Deviation	60.959	53.865	49.957	10.887

Table 5.4 illustrates the average, maximum, minimum, and standard deviation of the *gbest* convergence of 100 simulations of swarms with 5, 10, 15, and 20 particles, after 1000 time steps. The minimum *gbest* was

$$f_2(\mathbf{x}^\star) = 35.377$$

at

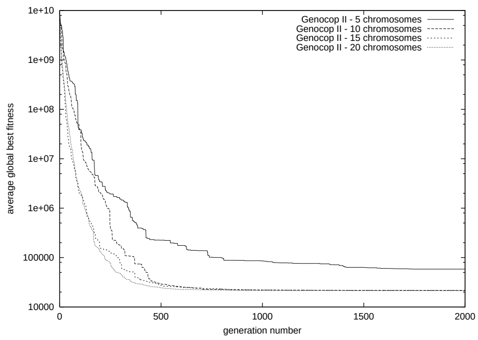
$$\mathbf{x}^\star = (0.076, -0.281, 0.445, -0.373, -3.956, 3.762, 1.12, -1.865, -0.538, 0.178)^T$$

If the average minimum values found in Figures 5.4(a) and 5.6(a) are compared, CLPSO shows a faster rate of convergence than *Genocop II*. The standard deviation after 1000 iterations or generations is smaller for *Genocop II* (compare Tables 5.3 and 5.4), indicating greater consistency in convergence between the different simulations.

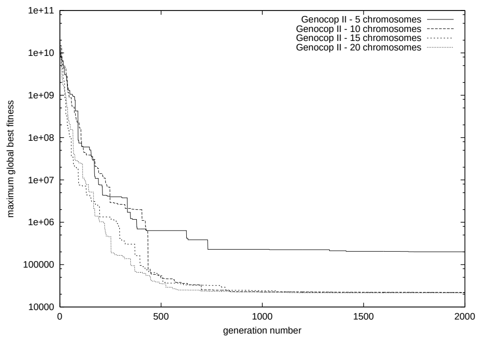
Test 3

The third function tested, f_3, is a Rosenbrock function in ten dimensions. The constrained f_3 differs from both f_1 and f_2 because it is not a convex function. The free dimensions were randomly initialised in the interval $[-100, 100]$. The problem is defined as

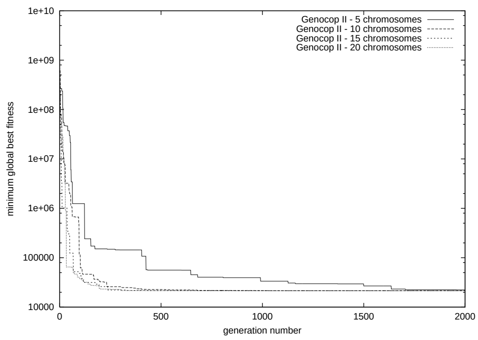
$$\begin{aligned} &\text{Minimise} \quad f_3(\mathbf{x}) = \textstyle\sum_{i=1}^{n-1}(100(x_{i+1} - x_i^2)^2 + (1 - x_i)^2),\ \mathbf{x} \in \mathbb{R}^{10} \\ &\text{Subject to} \qquad\qquad A\mathbf{x} = \mathbf{b} \end{aligned} \tag{5.4}$$



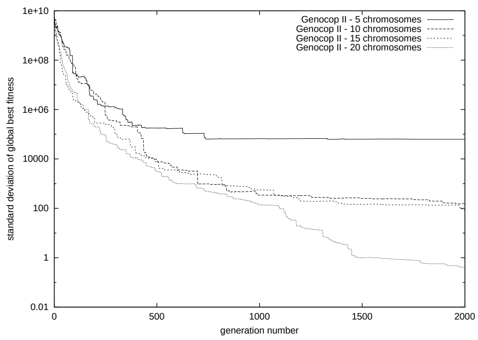
(a) Average



(b) Maximum



(c) Minimum



(d) Standard Deviation

FIGURE 5.7: Results of 100 *Genocop II* simulations on the constrained Rosenbrock function f_3 defined in equation (5.4).

where A and $\mathbf{b}$ are defined in equation (5.1).

Genocop II The best solution found by *Genocop II*, with a population size of 100 and 4000 generations, was $f_3(\mathbf{x}^\star) = 21485.361$ with

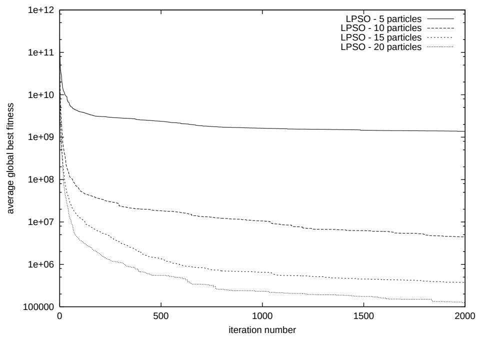
$$\mathbf{x}^\star = (0.84, -1.516, 2.359, -0.669, -3.352, 2.991, 1.053, -1.949, -0.273, -0.028)^T$$

Genocop II was evolved for a total of 2000 generations, for population sizes of 5, 10, 15, and 20 chromosomes. The averages, maximums, minimums and standard deviations over 100 simulations are shown in Figure 5.7, and are computed in the same way as Test 1. Again, these results are summarised in Table 5.5, and are compared to the PSO under the CLPSO results.

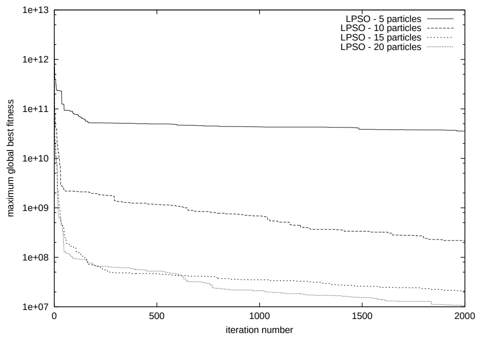
LPSO The results of LPSO over 2000 time steps are shown in Figure 5.8, with the averages, maximums, minimums, and standard deviations computed in the same way as explained in

TABLE 5.5: Results of 100 *Genocop II* simulations on the constrained Rosenbrock function f_3 defined in equation (5.4), after 2000 generations. ('chromosomes' is abbreviated as chrms.)

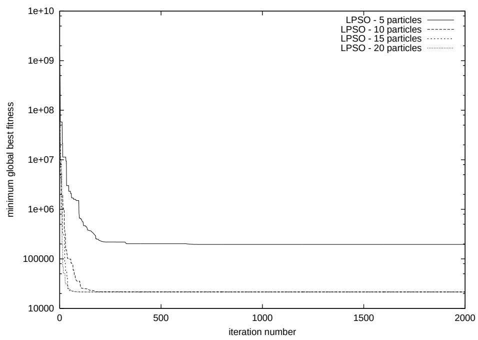
Genocop II	5 chrms.	10 chrms.	15 chrms.	20 chrms.
Average	58249.328	21630.020	21546.332	21485.714
Maximum	2.005×10^5	22030.988	21836.797	21486.646
Minimum	22334.971	21490.840	21487.098	21485.363
Standard Deviation	62513.767	154.443	85.311	0.400



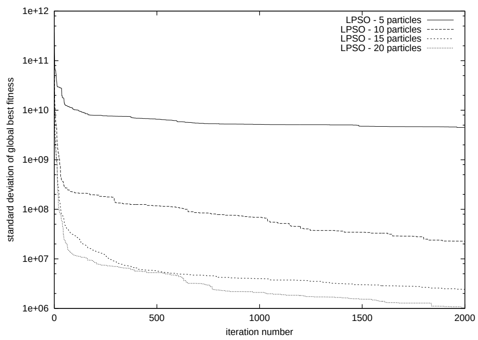
(a) Average



(b) Maximum

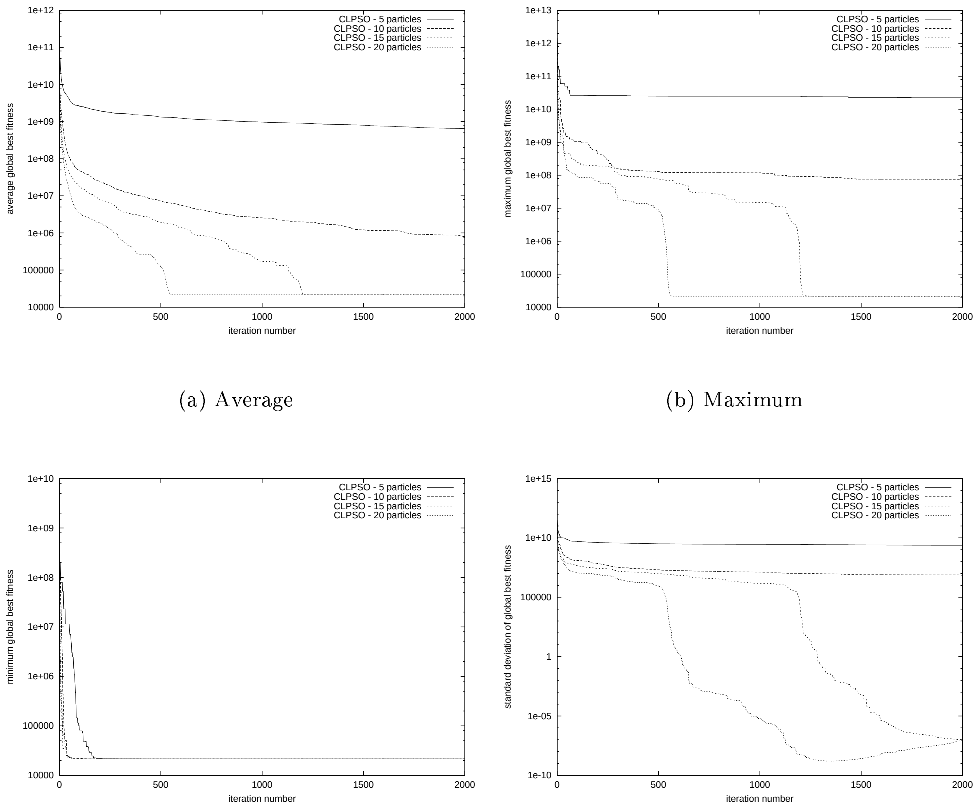


(c) Minimum



(d) Standard Deviation

FIGURE 5.8: Results of 100 simulations of LPSO on the constrained Rosenbrock function f_3 defined in equation (5.4).



(a) Average (b) Maximum

(c) Minimum (d) Standard Deviation

FIGURE 5.9: Results of 100 simulations of CLPSO on the constrained Rosenbrock function f_3 defined in equation (5.4).

Test 1 above.

The LPSO swarms of 10, 15, and 20 particles managed to find the minimum value of the function, or came close to it at iteration number 2000. After 2000 iterations, the LPSO swarms were still finding better solutions, as is illustrated in Figure 5.8(a).

CLPSO The results of CLPSO over 2000 time steps are shown in Figure 5.9, with the averages, maximums, minimums, and standard deviations computed in the same way as explained in Test 1 above.

The 20-particle CLSPO consistently converged to the minimum, as can be seen from Figure 5.9 and Table 5.6. Figure 5.9(a) also shows that the average fitness decreases dramatically to the minimum after the swarm has converged below a certain fitness level. It also shows that the swarms of 5 and 10 particles did not stagnate, but are still converging at the 2000$^{\text{th}}$ iteration. The sudden and complete convergence when the swarm decreases

TABLE 5.6: Results of 100 LPSO and CLPSO simulations on the constrained Rosenbrock function f_3 defined in equation (5.4), after 2000 iterations.

LPSO	5 particles	10 particles	15 particles	20 particles
Average	1.375×10^9	4.444×10^6	3.710×10^5	1.260×10^5
Maximum	3.556×10^{10}	2.177×10^8	2.054×10^7	1.045×10^7
Minimum	1.955×10^5	21554.158	21483.373	21485.925
Standard Deviation	4.485×10^9	2.278×10^7	2.407×10^6	1.043×10^6
CLPSO	**5 particles**	**10 particles**	**15 particles**	**20 particles**
Average	6.522×10^8	7.446×10^5	21485.305	21485.305
Maximum	2.233×10^{10}	7.112×10^7	21485.305	21485.305
Minimum	21485.306	21485.305	21485.305	21485.305
Standard Deviation	2.395×10^9	7.120×10^6	9.834×10^{-8}	9.401×10^{-8}

below a specific fitness value is confirmed by the standard deviations of Figure 5.9(d), where the variance in gbest for swarms of 20 particles becomes close to zero.

The raise in variance after a good minimum was found (see Figure 5.9(d)), can be attributed to the random search performed by CLPSO. As minutely better minimums are found, the *gbest* values will start to differ slightly, causing a rise in standard deviation in the order of 10^{-7}.

The CLPSO found

$$\mathbf{x}^\star = (0.84, -1.514, 2.359, -0.67, -3.352, 2.991, 1.053, -1.949, -0.274, -0.028)^T$$

after 2000 time steps. The value of f_3 at $\mathbf{x}^\star$ was

$$f_3(\mathbf{x}^\star) = 21485.305$$

The average best fitness of *Genocop II* is substantially better than that of both LPSO and CLPSO for small population or swarm sizes (see Figures 5.7(a) and 5.9(a)). This can be ascribed to a greater amount of mutation on the chromosomes (particles), and therefore a greater diversity in the solutions tested. The better convergence for small population or swarm sizes is supported by the difference in the standard deviations over the simulations, shown in Tables 5.5 and 5.6. For larger populations or greater swarm sizes, *Genocop II* and CLPSO have very similar performance.

5.1.2 LPSO and CLPSO Convergence characteristics

Some remarks, all confirming the theoretical properties needed for LPSO and CLPSO to successfully converge to a minimum, can be made from the above experimental results.

1. A swarm of 5 particles is smaller than the minimum swarm size, as derived in equation (4.27), of

$$\inf |S^{(0)}| = n - r + 1 = 10 - 5 + 1 = 6$$

   Thus LPSO will not cover all search dimensions, and results can be expected to be suboptimal. Indeed, the average *gbest* (minimum) of the constrained f_1 in equation (5.2) was at 7.034×10^3 after 250 time steps, while CLPSO managed to find an average *gbest* of 35.197 with five particles. The comparison is shown in Table 5.2. With five particles and $t = 1000$, LPSO's average best for f_2 defined in (5.3) was 8.463×10^3, while CLPSO's best f_2 was 82.007 (see Table 5.4).

2. As can be clearly seen in Figures 5.2, 5.5, and 5.8, the swarm catches up with the global best particle before reaching a minimum to cause premature convergence. This problem is overcome by CLPSO, as the empirical results in Figures 5.3, 5.6, and 5.9 illustrate.

5.2 Support Vector Machine Training

After showing the convergence and properties of the newly developed LPSO and CLPSO, the CLPSO algorithm will be implemented in training SVMs. This section illustrates the success and simplicity of the method, and also discusses some bottlenecks that have to be overcome to make the algorithm practically competitive.

5.2.1 Implementing the SVM training algorithm

Two issues remain to be resolved in implementing the SVM training algorithm described in Section 2.4. Both issues consist of finding feasible vectors: The first is to find an initial feasible solution $\boldsymbol{\alpha}$ for the algorithm to start with. The second is, given a working set B, to initialise the swarm of particles that is going to optimise B, such that the swarm is feasible.

Finding an initial feasible solution $\boldsymbol{\alpha}$

To resolve the first issue, a feasible solution that satisfies the linear constraint $\boldsymbol{\alpha}^T\mathbf{y} = 0$, with constraints $0 \leq \alpha_i \leq C$ also met, is needed at the start of the decomposition algorithm. The initial solution is constructed in the following way:

Let c be some real number between 0 and C, and γ some positive integer less than both the number of positive examples ($y_i = +1$) and negative examples ($y_i = -1$) in the training set. Randomly pick a total of γ positive examples, and γ negative examples, and initialise their corresponding α_i to c. By setting all other α_i to zero, the initial solution will be feasible.

The value 2γ gives the total number of initial support vectors, and since these initial support vectors are a randomly chosen guess, it is suggested that the value of γ be kept small.[1]

Initialising a feasible swarm of particles

To resolve the second issue, consider the constrained optimisation problem solved by the CLPSO, repeated here for convenience:

$$\max_{\boldsymbol{\alpha}_B} W(\boldsymbol{\alpha}_B) = \boldsymbol{\alpha}_B^T\mathbf{1} - \frac{1}{2}\boldsymbol{\alpha}_B^T Q_{BB}\boldsymbol{\alpha}_B - \boldsymbol{\alpha}_B^T Q_{BN}\boldsymbol{\alpha}_N \tag{5.5}$$

subject to

$$\begin{aligned} \boldsymbol{\alpha}_B^T\mathbf{y}_B + \boldsymbol{\alpha}_N^T\mathbf{y}_N &= 0 \\ \boldsymbol{\alpha}_B &\geq \mathbf{0} \\ C\mathbf{1} - \boldsymbol{\alpha}_B &\geq \mathbf{0} \end{aligned} \tag{5.6}$$

In optimising the q-dimensional subproblem, CLPSO requires that all particles be initialised such that $\boldsymbol{\alpha}_B^T\mathbf{y}_B + \boldsymbol{\alpha}_N^T\mathbf{y}_N = 0$ is met. This is done as follows:

1. Set each particle in the swarm to the q-dimensional vector $\boldsymbol{\alpha}_B$.
2. Add a random q-dimensional vector $\boldsymbol{\delta}$ satisfying $\mathbf{y}_B^T\boldsymbol{\delta} = 0$ to each particle, under the condition that the particle will still lie in the hypercube $[0, C]^q$.

Initialising the swarm in this way ensures that the initial swarm lies in the set of feasible solutions $P = \{\mathbf{p} \mid A\mathbf{p} = -\boldsymbol{\alpha}_N^T\mathbf{y}_N\}$, allowing the flight of the swarm to be defined by feasible directions.

5.2.2 Practical concerns and improvements

A number of practical issues need to be addressed to implement the algorithm numerically. One issue is on deciding when a solution is 'optimal enough,' and the Karush-Kuhn-Tucker

[1] In reality, *any* non-zero feasible vector can be used as an initial solution; choosing γ positive and negative examples only gives a simple way of constructing such a vector. Larger values of γ imply a larger set of initial support vectors, and the training algorithm simply spends extra time in removing the non-support vectors from this set.

conditions are adapted to be correct within an error threshold from the true conditions. The SVM training algorithm presented in Chapter 2 assumes infinite precision arithmetic. Since machine numbers allow only finite accuracy, the problem of error accumulation and round-off errors is addressed. A strategy is also given to optimise the dot product between two sparse vectors.

An approximation to the optimality conditions

The Karush-Kuhn-Tucker conditions (2.33) that define the stopping criteria for the training algorithm, specify that an $\alpha_i^{(t)}$ between zero and C must imply that $y_i(s_i^{(t)} + b^{(t)})$ should be exactly equal to one. In practice this is not always possible, and a small positive error ϵ on the KKT conditions will be tolerated to allow the algorithm to terminate. The value of ϵ close to 0.01 or 0.02 will typically give a very accurate optimisation [24]. The practical KKT conditions are therefore

$$\begin{aligned} \alpha_i^{(t)} = 0 &\Rightarrow y_i(s_i^{(t)} + b^{(t)}) > 1 - \epsilon \\ 0 < \alpha_i^{(t)} < C &\Rightarrow 1 - \epsilon < y_i(s_i^{(t)} + b^{(t)}) < 1 + \epsilon \\ \alpha_i^{(t)} = C &\Rightarrow y_i(s_i^{(t)} + b^{(t)}) < 1 + \epsilon \end{aligned} \tag{5.7}$$

Error accumulation and round-off errors

The nature of the constrained LPSO algorithm allows for division and multiplication by very large and very small real numbers. This can give rise to numerical precision problems. One of the constraints on the SVM optimisation problem is that the sum of all $y_i \alpha_i$ must be equal to zero. It may be true that, due to rounding errors, this sum can shift from zero. To solve this problem, a check is done to determine

$$error = \sum_{i=1}^{l} y_i \alpha_i$$

To reset the sum to zero, one of the zero Lagrange multipliers α_i is set to the absolute value of *error*. If *error* is positive, an α_i corresponding to a negative example y_i is randomly chosen. If the opposite is true and *error* is negative, an α_i corresponding to a positive example y_i is randomly chosen. As optimisation continues, this adjusted Lagrange multiplier will be picked for reoptimisation, with the equality constraint holding.

The update is done when *error* rises above a certain threshold; in the experiments presented here, *error* was in the order of 10^{-6}. In practice this update rarely happens, but can occur.

Optimising the dot product between two sparse vectors

The time taken to compute the dot product between two sparse vectors can be greatly optimised if all multiplications with zero are simply ignored. The dot product between two n-dimensional vectors $\mathbf{x}_i$ and $\mathbf{x}_j$ is defined as

$$\mathbf{x}_i \cdot \mathbf{x}_j = x_{i1}x_{j1} + x_{i2}x_{j2} + \ldots + x_{in}x_{jn}$$

Since a sparse vector contains many zero elements, many multiplications will be with zero and therefore unnecessary. The following algorithm is adapted from [41], and scans through both vectors to compute the dot product:

```
/* Array x1, with length n1, is an array that stores only
   xi's nonzero components. The original positions of these
   components in vector xi is stored in array id1. Arrays
   x2 and id2 with size n2 is used to store sparse vector xj.
*/
p1 = 0, p2 = 0, dot = 0
while (p1 < n1 && p2 < n2)
{
   a1 = id1[p1], a2 = id2[p2]
   if (a1 == a2)
   {
      dot += x1[p1]*x2[p2]
      p1++, p2++
   }
   else if (a1 > a2)
      p2++
   else
      p1++
}
```

5.2.3 Experimental results

The SVM training algorithm presented in Section 2.4 was tested on the MNIST dataset [33]. The influence of different working set sizes, as well as the scalability of the approach, is examined. Finally, the training results are compared to two other algorithms, a decomposition method and the method of sequential minimal optimisation.

FIGURE 5.10: A few examples from the MNIST dataset.

The MNIST dataset

The MNIST database is an optical character dataset, and consists of a training set of 60,000 handwritten digits [33]. This database is a subset of a larger set available from the National Institute of Standards Bureau (NIST). As shown in Figure 5.10, the examples are 28 by 28 pixel grey-level images. This is equivalent to each example being a 784-dimensional vector. Each pixel value corresponds to an integer in the range 0 (white) to 255 (black). It is a common database for benchmarking learning techniques and pattern recognition methods.

Training the SVM

For training a SVM on the MNIST dataset, the character '8' was chosen to represent the set of positive examples, while the remaining digits defined the negative examples. Training was done with a polynomial kernel of degree five:

$$k(\mathbf{x}_i, \mathbf{x}_j) = (\mathbf{x}_i \cdot \mathbf{x}_j + 1)^5 \tag{5.8}$$

Due to the size of the dot product between two images, raised to the fifth power, the pixel

TABLE 5.7: Influence of different working set sizes on the first 20,000 elements of the MNIST dataset

Working set size	Working set selections	Time	SVs
4	8,782	02:17:43	1,631
6	8,213	03:11:40	1,637
8	7,502	03:51:24	1,639
10	10,023	06:27:06	1,648
12	9,667	07:26:23	1,652

values were scaled to the range $[0, 0.1]$. This gives Lagrange multipliers α_i that are easier for the CLPSO to handle. (The kernel function of two unscaled black images would be $(784 \times 255^2 + 1)^5$, while the kernel function of the scaled versions gives a more practical $(784 \times 0.01 + 1)^5 \approx 835$).

For an optimal solution to be found in the following PSO experiments, the KKT conditions in equation (5.7) needed to be satisfied within an error threshold of $\epsilon = 0.02$. Optimisation of the working set terminated when the KKT conditions on the working set were met with an error of 0.001, or when the swarm has optimised for a hundred iterations.

The following parameters defined the experimental CLPSO: By letting $\gamma = 10$, a total of 20 initial support vectors were chosen to start the algorithm. The swarm size s used in each experiment was 10, while the inertia weight w was set to 0.7. The acceleration coefficients c_1 and c_2 were both set to 1.4 [53]. Since the objective function is constrainted by a set of box constraints, the velocity vectors were not clamped. For each experiment the upper bound C was kept at 100.0 (a commonly used upper bound in SVM training).

The PSO training algorithm was written in Java, and does not make use of caching and shrinking methods to optimize its speed. The sparsity of input data is used to speed up the evaluation of kernel functions. All experiments were preformed on a 1.00 GHz AMD Duron processor.

Experimental results show successful and accurate training on the MNIST database. The influence of different working set sizes on the CLPSO training algorithm, its scalability, as well as its relation to other SVM training algorithms, were examined.

TABLE 5.8: Scalability: training on the MNIST dataset

MNIST elements	PSO Working set selections	PSO time	PSO SVs	SMO time	SMO SVs	SVM^{light} time	SVM^{light} SVs
10,000	3,898	00:29:49	1,022	00:01:29	1,032	00:02:02	1,034
20,000	8,782	02:17:43	1,631	00:06:14	1,647	00:10:43	1,641
30,000	12,428	04:50:11	1,988	00:13:22	2,012	00:23:04	2,001
40,000	15,725	08:14:26	2,353	00:22:46	2,355	00:41:09	2,367
50,000	22,727	15:05:09	2,728	01:46:38	2,740	01:31:48	2,726
60,000	25,914	20:54:15	3,025	04:38:11	3,043	08:01:05	3,026

Influence of working set sizes

Experiments on different working set sizes were done on the first 20,000 elements of the MNIST database. Results are shown in Table 5.7, and indicate that a working set of size $q = 4$ gives the fastest convergence time and fewest support vectors. A working set of size 2 can be solved analytically, as is true in the case of Sequential Minimal Optimisation (SMO). The results in Table 5.7 are not necessarily an indication of the speed of the PSO on the working set, as selection of the working set also burdens the speed of the algorithm (the $\frac{q}{2}$ greatest and least values of $y_i \nabla W(\boldsymbol{\alpha})_i$ need to be selected from a list of thousands).

Scalability of the PSO approach

Scalability of the PSO algorithm was tested by training on the first 10,000, 20,000, etc. examples from the MNIST dataset, as shown in Table 5.8. In each case a working set of size 4 was used. The experimental results indicate that the PSO training algorithm shows quadratic scalability, and scales as $\sim l^{2.1}$ (with l being the training set size).

Comparison to other algorithms

In Table 5.8, the PSO approach is compared to SMO and a decomposition method, SVM^{light} [24]. WinSVM was developed by C. Longbin [29] from the SVM^{light} source code, and was used as an implementation of SMO. Unlike these methods, the current PSO algorithm does not make use of caching and shrinking to optimise its speed.

Results similar to Table 5.7 indicate that SVM^{light} gives the fastest rate of convergence with a working set size $q = 8$, which is used in Table 5.8's comparison.

Experimental results show SMO scaling as $\sim l^{2.8}$, and SVM^{light} scaling as $\sim l^{3.0}$. Both these algorithms are substantially faster than training a SVM with PSO on the MNIST dataset, but the PSO approach shows better scaling abilities ($\sim l^{2.1}$). Due to the fact that the PSO training algorithm starts with a very small set of possible support vectors, with all other α_i set to zero, the PSO method consistently finds a few support vectors less than the other approaches.

The main drawback from the current PSO approach is its slow performance times, but from this initial study many optimisations can be implemented on both decomposition and PSO methods.

5.3 Concluding

The success of the CLPSO in optimising linearly constrained functions was experimentally illustrated in this chapter. The necessity to change the LPSO to a locally converging algorithm was also shown.

It was shown that a PSO can be used to train a SVM. Some properties of LPSO make it particularly useful to solve the SVM constrained QP problem. The PSO algorithm is simple to implement, and does not require any background of numerical methods. Accurate and scalable training results were shown on the MNIST dataset, with the PSO algorithm finding fewer support vectors and better scalability than other approaches. Although the algorithm is simple, its speed poses a practical bottleneck, which can be improved from this initial study.

Chapter 6

Conclusion and Future Work

This thesis aimed to answer the question - "can a Particle Swarm Optimiser be used to train a Support Vector Machine, and to what extent will it be successful?"

The research conducted for this thesis stood on two pillars. The first pillar was Support Vector Machines (SVMs) and algorithms to train them, and a decomposition-training algorithm was developed based on similar algorithms. The second pillar was Particle Swarm Optimisation (PSO), which is implemented as the optimisation method in the SVM training algorithm.

Concluding on the second pillar, it was shown that particle swarms can easily be used to optimise a function with equality constraints of the form $A\mathbf{x} = \mathbf{b}$. A variation of PSO, the "Linear Particle Swarm Optimiser" (LPSO), was introduced to optimise these types of problems, and conditions for the LPSO to be able to find any point in the feasible search space, was developed. There is a positive probability that LPSO can converge prematurely. The problem of LPSO's premature convergence was overcome by creating a "Converging LPSO" (CLSPO). A proof was given to show that CLPSO will at least converge to a local minimum. An important property of the two new PSO algorithms is that, if the whole swarm is initialised to lie within the hyperplane $A\mathbf{x} = \mathbf{b}$, then the swarm can optimise the objective function without having to worry about the set of constraints. This property was formally proved, and shows that LPSO and CLPSO are ideally suited to solving equality-constrained optimisation problems. The success of CLPSO (and premature convergence of LPSO) in optimising linearly constrained functions was experimentally illustrated. The experimental results were compared to results achieved with *Genocop II*, a genetic algorithm for constrained optimisation. Experimental results show a general similarity in convergence between *Genocop II* and CLPSO.

To conclude on the first pillar, it was shown that a PSO could be used to train a SVM. Some properties of CLPSO make it particularly useful to solve the SVM constrained quadratic programming problem, and it was used in the decomposition algorithm to solve the SVM's constrained subproblems. The CLPSO algorithm is simple to implement, and does not require any background of numerical methods. Accurate and scalable training results were shown on the MNIST dataset.

Although the CLPSO algorithm is simple, its speed in SVM training poses a practical bottleneck. Future research may include improvement to the speed of the algorithm by improving the CLPSO, and the cashing of kernel evaluations can be implemented.

Further research can also explore the possibility of parallel training of SVMs. Instead of selecting a single working set for optimisation, a number of working sets can be selected and optimised in parallel. If the working sets are distinct, the subproblems will be independent of each other, making this method a strong candidate for further investigation.

The standard methods of improving the original PSO can also be implemented on both LPSO and CLPSO. There is also scope for a proper analysis of CLPSO in the context of random search algorithms.

Finally, many interesting constrained problems are waiting to be solved!

Publications derived from this thesis

U. Paquet and A.P. Engelbrecht. "Training support vector machines with particle swarms," in *Proceedings of the International Joint Conference on Neural Networks*, Portland, Oregon, 2003.

U. Paquet and A.P. Engelbrecht. "Particle swarms for equality-constrained optimization," submitted to *IEEE Transactions on Evolutionary Computation*.

U. Paquet and A.P. Engelbrecht. "A new particle swarm optimizer for linearly constrained optimization," submitted to *The Congress on Evolutionary Computation*, Canberra, Australia, 2003.

Bibliography

[1] M. Aizerman, E. Braverman, and L. Rozoner. "Theoretical foundations of the potential function method in pattern recognition learning," in *Automation and Remote Control*, volume 25, pages 821-837, 1964.

[2] P.J. Angeline. "Evolutionary optimization versus particle swarm optimization: philosophy and performance differences," in *Evolutionary Programming VII: Proceedings of the Seventh Annual Conference on Evolutionary Programming*. 1998

[3] T. Bäck, D.B. Fogel, and Z. Michalewicz. *Handbook of Evolutionary Computation*. Oxford University Press, New York NY, 1997.

[4] V. Blanz, B. Schölkopf, H. Bülthoff, C. Burges, V. Vapnik, and T. Vetter. "Comparison of view-based object recognition algorithms using realistic 3d models," in C. von der Malsburg, W. von Seelen, J.C. Vorbrüggen, and B. Sendhoff, editors, *Artificial Neural Networks – ICANN '96*, pages 47-52, Berlin, 1996. Springer Lecture Notes in Computer Science, Volume 1112.

[5] E. Bonabeau, M. Dorigo, and G. Theraulaz. *Swarm Intelligence: From natural to artificial systems*. Oxford University Press, 1999.

[6] B.E. Boser, I.M. Guyon, and V.N. Vapnik. "A training algorithm for optimal margin classifiers," in D. Haussler, editor, *Proceedings of the Fifth Annual ACM Workshop on Computational Learning Theory*, pages 144-152, Pittsburgh, PA, 1992. ACM Press.

[7] R. Brits. *Niching particle swarm optimizers*. PhD Thesis, Department of Computer Science, University of Pretoria, 2003.

[8] C.J.C. Burges and B. Schölkopf. "Improving the accuracy and speed of support vector learning machines," in M. Mozer, M. Jordan, and T. Petsche, editors, *Advances in Neural Information Processing Systems 9*, pages 375-381, Cambridge, MA, 1997. MIT Press.

[9] C. Burges and D. Crisp. "Uniqueness of the SVM solution," in NIPS, 12, 2000.

[10] C. Campbell. "Algorithmic approaches to training support vector machines: a survey," in *Proceedings of the Eigth European Symposium On Artificial Neural Networks*, pages 27-36, Bruges, Belgium, 2000.

[11] M. Clerc and J. Kennedy. "The particle swarm – explosion, stability, and convergence in a multidimensional complex space," in *IEEE Transactions on Evolutionary Computation*, volume 6, number 1, pages 58-73, 2002.

[12] D. Corne, M. Dorigo, and F. Glover (editors). *New Ideas in Optimization*. McGraw-Hill, 1999.

[13] C. Cortes and V. Vapnik. "Support vector networks," in *Machine Learning*, volume 20, pages 273-297, 1995.

[14] R. Courant and D. Hilbert. *Methods of Mathematical Physics*. Interscience, New York, 1953.

[15] L. Davis. *Genetic Algorithms and Simulated Annealing*. Pitman, London, 1987.

[16] R.C. Eberhart, R.W. Dobbins, and P. Simpson. *Computational Intelligence PC Tools*. Academic Press, 1996.

[17] R.C. Eberhart and Y. Shi. "Comparing inertia weights and constriction factors in particle swarm optimization," in *Proceedings of the Congress on Evolutionary Computation*, pages 84-88. 2000.

[18] R. Fletcher. *Practical Methods of Optimization, Volume 1*. John Wiley and Sons, Inc., 1980.

[19] R. Fletcher. *Practical Methods of Optimization*. John Wiley and Sons, Inc., 2nd edition, 1987.

[20] D.E. Goldberg. *Genetic Algorithms in Search Optimization and Machine Learning*. Addison-Wesley, 1989.

[21] I. Guyon, B. Boser, and V. Vapnik. "Automatic capacity tuning of very large VC-dimension classifiers," in S.J. Hanson, J.D. Cowan, and C.L. Giles, editors, *Advances in Neural Information Processing Systems*, volume 5, pages 147-155. Morgan Kaufmann, San Mateo, CA, 1993.

[22] S.B. Hamida and M. Schoenauer. "ASHEA: New results using adaptive segregational handling," in *IEEE World Congress on Computational Intelligence, Proceedings of the Congress on Evolutionary Computing*. Honolulu, Hawaii, 2002.

[23] T. Joachims. “Text categorization with support vector machines,”Technical report, LS VIII Number 23, University of Dortmund, 1997.

[24] T. Joachims, “Making large-scale SVM learning practical,” in *Advances in Kernel Methods – Support Vector Learning*, B. Schölkopf, C.J.C Burges, and A.J. Smola, editors, pages 169-184. MIT Press, Cambridge, MA, 1999.

[25] J. Kennedy and R.C. Eberhart. “Particle swarm optimization,” in *Proceedings of the IEEE International Conference on Neural Networks, IV*, pages 1942-1948. 1995.

[26] J. Kennedy. “Small worlds and mega minds: effects of neighborhood topology on particle swarm performance,” in *Proceedings of the Congress of Evolutionary Computation, Washington DC, USA*, pages 1931-1938. 1999.

[27] J. Kennedy, R.C. Eberhart, and Y. Shi. *Swarm Intelligence*. Morgan Kaufmann Publishers, 2001.

[28] J. Kennedy and R. Mendes. “Population structure and particle swarm performance,” in *IEEE World Congress on Computational Intelligence, Proceedings of the Congress on Evolutionary Computing*. Honolulu, Hawaii, 2002.

[29] C. Longbin. `http://liama.ia.ac.cn/PersonalPage/lbchen/`, Institute of Automation, Chinese Academy of Sciences (CASIA).

[30] Z. Michalewicz. *Genetic Algorithms + Data Structures = Evolution Programs*. Springer Verlag, 1996.

[31] Z. Michalewicz and C.Z. Janikow. “GENOCOP: a genetic algorithm for numerical optimization problems with linear constraints,” in *Communications of the ACM*, volume 39, article no. 175. 1996.

[32] Z. Michalewicz and M. Schoenauer. “Evolutionary algorithms for constrained parameter optimization Problems,” in *Evolutionary Computation*, volume 4, pages 1-32. 1996.

[33] MNIST Optical Character Database at AT&T Research, `http://yann.lecun.com/exdb/mnist/`.

[34] S. Mukherjee, E. Osuna, and F. Girosi. “Nonlinear prediction of chaotic time series using a support vector machine,” in *Neural Networks for Signal Processing VII – Proceedings of the 1997 IEEE Workshop*, J. Principe, L. Gile, N. Morgan, and E. Wilson, editors, pages 511-520, New York, 1997. IEEE Press.

[35] K.-R. Müller, S. Mika, G. Rätsch, K. Tsuda, and B. Schölkopf. “An introduction to kernel-based learning algorithms,” in *IEEE Transactions on Neural Networks*, volume 12, number 2, pages 181-202, 2001.

[36] K.-R. Müller, A. Smola, G. Rätsch, B. Schölkopf, J. Kohlmorgen, and V.N. Vapnik. "Predicting time series with support vector machines," in *Artificial Neural Networks – ICANN '97*, W. Gerstner, A. Germond, M. Hasler, and J.-D. Nicoud, editors, pages 999-1004, Berlin, 1997. Springer Lecture Notes in Computer Science, Volume 1327.

[37] J. Nocedal and S.J. Wright. *Numerical Optimization*. Springer Verlag, 1999.

[38] P. Laskov, "Feasible direction decomposition algorithms for training support vector machines," in *Machine Learning, Volume 46*, N. Cristianini, C. Campbell, and Chris Burges, editors, pages 315-349, 2002.

[39] E. Osuna, R. Freund, and F. Girosi. "Training support vector machines: an application to face detection," in *IEEE Conference on Computer Vision and Pattern Recognition*, pages 130-136, 1997.

[40] E. Osuna, R. Freund, and F. Girosi, "An improved training algorithm for support vector machines," in *Neural Networks for Signal Processing VII – Proceedings of the 1997 IEEE Workshop*, J. Principe, L. Gile, N. Morgan, and E. Wilson, editors, pages 276-285. IEEE, New York, 1997.

[41] J. Platt, "Fast training of support vector machines using sequential minimal optimization," in *Advances in Kernel Methods – Support Vector Learning*, B. Schölkopf, C.J.C Burges, and A.J. Smola, editors, pages 185-208. MIT Press, Cambridge, MA, 1999.

[42] M. Schmidt. "Identifying speaker with support vector networks," in *Interface '96 Proceedings*, Sydney, 1996.

[43] B. Schölkopf, *Support vector learning*. Oldenbourg Verlag, Munich, 1997.

[44] B. Schölkopf, C. Burges, and V. Vapnik. "Extracting support data for a given task," in U.M. Fayyad and R. Uthurusamy, editors, *Proceedings of the First International Conference on Knowledge Discovery and Data Mining*. AAAI Press, Menlo Park, CA, 1995.

[45] B. Schölkopf, C. Burges, and V. Vapnik. "Incorporating invariances in support vector learning machines," in C. von der Malsburg, W. von Seelen, J.C. Vorbrüggen, and B. Sendhoff, editors, *Artificial Neural Networks – ICANN '96*, pages 47-52, Berlin, 1996. Springer Lecture Notes in Computer Science, Volume 1112.

[46] Y. Shi and R.C. Eberhart. "Parameter selection in particle swarm optimization," in *Proceedings of the Seventh Annual Conference on Evolutionary Programming*, pages 591-600. New York, 1998.

[47] Y. Shi and R.C. Eberhart. "A modified particle swarm optimizer," in *Proceedings of the IEEE Congress on Evolutionary Computation*, pages 69-73. Piscataway, NJ, 1998.

[48] Y. Shi and R.C. Eberhart. “Empirical study of particle swarm optimization,” in *Proceedings of the IEEE Congress on Evolutionary Computation*, volume 3, pages 1945-1950. 1999.

[49] Y. Shi and R.A. Krohling. “Co-evolutionary particle swarm optimization to solve min-max problems,” in *IEEE World Congress on Computational Intelligence, Proceedings of the Congress on Evolutionary Computing*. Honolulu, Hawaii, 2002.

[50] A.J. Smola, *Learning with kernels*. Ph.D. thesis, Technische Universität Berlin, 1998.

[51] M.O. Stitson, A. Gammerman, V. Vapnik, V. Vovk, C. Watkins, and J. Weston. “Support vector ANOVA decomposition,” Technical report, Royal Holloway College, Report number CSD-TR-97-22, 1997.

[52] P.N. Suganthan. “Particle swarm optimiser with neighbourhood operator.” in *Proceedings of the IEEE Congress on Evolutionary Computation*, pages 1958-1962. Piscataway, NJ, 1999.

[53] F. van den Bergh. *An analysis of particle swarm optimizers*. PhD Thesis, Department of Computer Science, University of Pretoria, 2002.

[54] F. van den Bergh and A.P. Engelbrecht. “A locally convergent particle swarm optimiser,” accepted for *IEEE conference on Systems, Man, and Cybernetics*. Tunisia, 2002.

[55] V. Vapnik. *Estimation of Dependences Based on Empirical Data [in Russian]*, Nauka, Moscow, 1979. (English translation: Springer Verlag, New York, 1982).

[56] V. Vapnik. *The Nature of Statistical Learning Theory*. Springer Verlag, New York, 1995.

[57] V. Vapnik. *Statistical Learning Theory*. Wiley, New York, 1998.

[58] V. Vapnik and A. Chervonenkis. *Theory of Pattern Recognition [in Russian]*, Nauka, Moscow, 1974. (German Translation: W. Wapnik & A Tscherwonenkis, *Theorie der Zeichenerkennung*. Akademie-Verlag, Berlin, 1979).

[59] J. Weston, A. Gammerman, M.O. Stitson, V. Vapnik, V. Vovk, and C. Watkins. “Density estimation using support vector machines,” Technical report, Royal Holloway College, Report number CSD-TR-97-23, 1997.

[60] D. Whitley, V.S. Gordon, and K. Mathias. “Lamarcian evolution, the baldwin effect and function optimization,” in Y. Davidor, H-P Schwefel, and R. Männer, editors, *Proceedings of the Third Conference on Parallel Problem Solving from Nature*. Springer, 1996.

[61] G. Zoutendijk. *Methods of Feasible Directions*. Elsevier, Amsterdam, 1970