# Evolutionary Optimization Of Support Vector Machines

Fred Gruber
*University of Central Florida*

# EVOLUTIONARY OPTIMIZATION OF SUPPORT VECTOR MACHINES

By

FRED KARL GRUBER
B.S. Technological University of Panama, Panama, 2003

A thesis submitted in partial fulfillment of the requirements
for the degree of Master of Science
in the Department of Industrial Engineering and Management Systems
in the College of Engineering and Computer Science
at the University of Central Florida
Orlando, Florida

Fall Term
2004

Major Professor: Luis Rabelo

## ABSTRACT

Support vector machines are a relatively new approach for creating classifiers that have become increasingly popular in the machine learning community. They present several advantages over other methods like neural networks in areas like training speed, convergence, complexity control of the classifier, as well as a stronger mathematical background based on optimization and statistical learning theory. This thesis deals with the problem of model selection with support vector machines, that is, the problem of finding the optimal parameters that will improve the performance of the algorithm. It is shown that genetic algorithms provide an effective way to find the optimal parameters for support vector machines. The proposed algorithm is compared with a backpropagation Neural Network in a dataset that represents individual models for electronic commerce.

*To my family*

## ACKNOWLEDGMENTS

My sincere gratitude to Dr. Luis Rabelo for supporting me during the duration of my Master's studies, for proposing such a remarkable topic for my thesis, and for his constant support and guidance.

Special thanks to Dr. José Sepúlveda for his advice and support.

Thanks also to Dr. Christopher Geiger for participating in the committee of my thesis defense and for his suggestions for improving my thesis.

# TABLE OF CONTENTS

# LIST OF TABLES

# LIST OF FIGURES

# CHAPTER 1. INTRODUCTION

This chapter provides a brief overview of the main topics considered in this thesis. First, we will start by briefly discussing learning algorithms in general and their relation with simulation modeling. Then we will introduce a very specific learning algorithm known as Support Vector Machines and how we intend to improve them.

## 1.1 Learning Algorithms

A *learning algorithm* is a program that creates a function, classifier, or solution from a given data set called the training set. A special type of learning algorithm is the so called *supervised learning* in which the training set is pairs of the form $(\mathbf{x}_1, y_1), (\mathbf{x}_2, y_2), \ldots, (\mathbf{x}_n, y_n)$ where $\mathbf{x}_n$ is a vector and $y_n$ is a number assigned to vector $\mathbf{x}_n$. If the value $y_n$ comes from a finite set of possible values, then we are dealing with a classification problem. If $y_n$ can take any real number, then we are dealing with a regression problem. In this thesis, we will be working with binary classification only.

The main objective of learning algorithms for classification is to create a function that will correctly classify data points that are not used in its construction. This is called the *generalization ability* and it is the performance measure used in comparing and selecting algorithms. The generalization ability is usually measured by the percentage of correct classifications or the percentage of misclassifications in the test set. More about this is discussed in Section 2.4.

Learning algorithms and artificial intelligence in general, have a lot of similarities to simulation. In simulation we use computers to create a model of an existing natural or artificial system in order to analyze its behavior, estimate certain characteristics, and perhaps to try to find optimal parameters that will improve certain measures of performance. In artificial intelligence, we try to model knowledge (Fishwich and Modejeski, 1991) or to simulate certain aspects of human intelligent behavior (Paul Fishwich and Modejeski, 1991), therefore, we deal with more abstract systems. In both cases, we need input and output data used to create models and validate them, e.g., arrival of customers, service time, time waiting in system, and so on. Also in both cases, there is a lot of uncertainty involved in the data that we use; therefore, there is a need of statistical tools to arrive to trustworthy conclusions.   There are differences though: in learning algorithms, the model is created from the data alone; we have a black box that for certain inputs produces certain outputs, while in simulation we create the model from our understanding of an existing system and the data available.  Another important difference is the objective of both techniques.  With learning algorithms, we are interesting in discovering the true relationship between the inputs and the outputs while, in simulation, we may be more interested in improving the original system in terms of certain measures of performances. Also, sometimes there are used complementary to each other.  For instance, simulation is sometimes used to evaluate a learning algorithm.  Despite their differences, the fusion of both techniques has created new types of simulations like, for instances, Knowledge-based simulation—which represents a natural extension of simulation that try to go beyond the classical *what-if* question answered by conventional simulation (Rothenberg, 1991; Fishwich and Modejeski, 1991)—and agent based simulation –which uses the concept of agents (Wooldridge, 2002) to model complex systems.  In this thesis, the experimental data is a collection of 125 datasets—from an experiment conducted by Ryan (1999)—that represent the likes and dislikes of 125 individuals with respect to certain images. The images are produced by the variation of several parameters: blur, color, brightness, density, pointalization (size of the points that make the drawing), saturation, and background (see Section 4.1 for a definition of this parameters). Therefore, in this case the system is the abstract subjective preferences of a person (see Section 4.1).

## 1.2 <u>Support Vector Machines</u>

Support vector machines are a relatively new approach for creating classifiers that have become increasingly popular in the machine learning community. They present several advantages over other methods like neural networks in several areas like training speed, convergence, controlling the complexity of the classifier, as well as a stronger mathematical background based on statistical learning theory. The theory behind support vector machines is described in Section 2.3.

Nevertheless, like most learning algorithms they present certain problems. One of them is related to the parameters that control the behavior and that ultimately determine how good the resulting classifier is. The simplest way to find good parameter values is using an exhaustive search, i.e., trying all possible combinations but this method is impractical as the number of parameters increases. The problem of finding the right parameters values to improve the performance is called the model selection problem. This problem has been explored by several researchers but in most cases the methods proposed were limited to certain restrictive condition (see CHAPTER 3).

## 1.3 Model Selection with Genetic Algorithms

This thesis deals with the problem of finding the optimal parameters of support vector machines. The main contribution of this thesis is to show that genetic algorithms provide an effective way to find the optimal parameters for support vector machines and to propose a possible implementation. Several variations of the basic genetic algorithm are compared and the one with the fastest convergence is selected. In addition, it is shown that using a convex sum of two kernels (a function such that $K\left(\mathbf{x}_i, \mathbf{x}_j\right) = \left\langle \Phi\left(\mathbf{x}_i\right) \cdot \Phi\left(\mathbf{x}_j\right) \right\rangle$ where $\Phi$ is an arbitrarily mapping function and $\mathbf{x}$ is a vector. See Section 4.2.2 for more information about kernel) provides an effective kernel for classification problems and not only for regression as is previously tested in Smits and Jordaan (2002). The algorithm is tested in a data set that consists of information on 125 subjects from a study conducted by Ryan (1999) and previously used for comparing several learning algorithms in Rabelo (2001). The proposed algorithm is compared with a backpropagation Neural Network in a dataset that represents individual models for electronic commerce (CHAPTER 5).

## 1.4 Synopsis of Thesis

CHAPTER 2 contains the theoretical background of learning algorithms, support vector machines, generalization performance measures, and some important statistical test for comparing algorithms. CHAPTER 3 presents literature related to the research that has been done regarding model selection with support vector machines and the integration of genetic algorithms with support vector machines. CHAPTER 4 presents our implementation, case study, and some experiments to help us decide on the best parameters of the genetic algorithm for this particular application.  CHAPTER 5 compares the proposed algorithms with a fixed architecture SVM and with a backpropagation neural network with different number of hidden nodes.  Finally, CHAPTER 6 contains the conclusions, contribution, and future research directions.

# CHAPTER 2. THEORETICAL BACKGROUND

This chapter provides the theoretical background behind the two main techniques used in this thesis: genetic algorithms and support vector machines. In addition, we will provide some basic concepts of learning algorithms for pattern recognition, generalization error estimates, and some basic statistical tests that we use.

## 2.1 Genetic Algorithms

Evolutionary computation is a search and optimization technique inspired in natural evolution. The various evolutionary models that have been proposed and studied are usually referred as evolutionary algorithms (EAs) and they share similar characteristics (Bäck et. al, 2000):

- the use of a population of individuals or possible solutions,
- the creation of new solutions or individual by means of random process that model biological crossover and mutation, and
- a fitness function that assign a numeric value to each individual in the population. A selection process will favor those individual with a higher fitness function. The fitness function represents the environment in which the individuals live.

Genetic algorithms (GAs) (see Figure 1) are evolutionary algorithms first proposed by Holland in 1975 (Holland, 1975) and they initially had three distinguishing features (Bäck et. al, 2000):

- the individuals are represented by bitstrings, i.e., strings of 0's and 1's of fixed length,

- the individuals are selected for reproduction according to proportional selection, and
- the primary method of producing variation is crossover. In addition, mutation of newly-generated offspring induced variability in the population.

GAs have been through a lot of changes and the difference with other EAs has started to blur. Nevertheless, most GAs implementation follow certain common elements (Goldberg, 1989; Mitchell ,1998):

- they work with a representation or coding of the parameters,
- they search a populations of individuals,
- selection is according to a fitness function only, and
- they use probabilistic transition rules.

The search for a solution implies a compromise between two contradictory requirements: *exploitation* of the best available solution, and robust *exploration* of the search space. Exploitation is referred to the search of similar solutions and it is closely related to the crossover operator while exploration involves a global search and it is related to the mutation operator. If the solutions are overexploited, a *premature convergence* of the search procedure may occur. This means that the search stops progressing and the procedure eventually ends with a suboptimal solution. If emphasis is given to the exploration, the information already available may be lost and the convergence of the search process could become very slow.

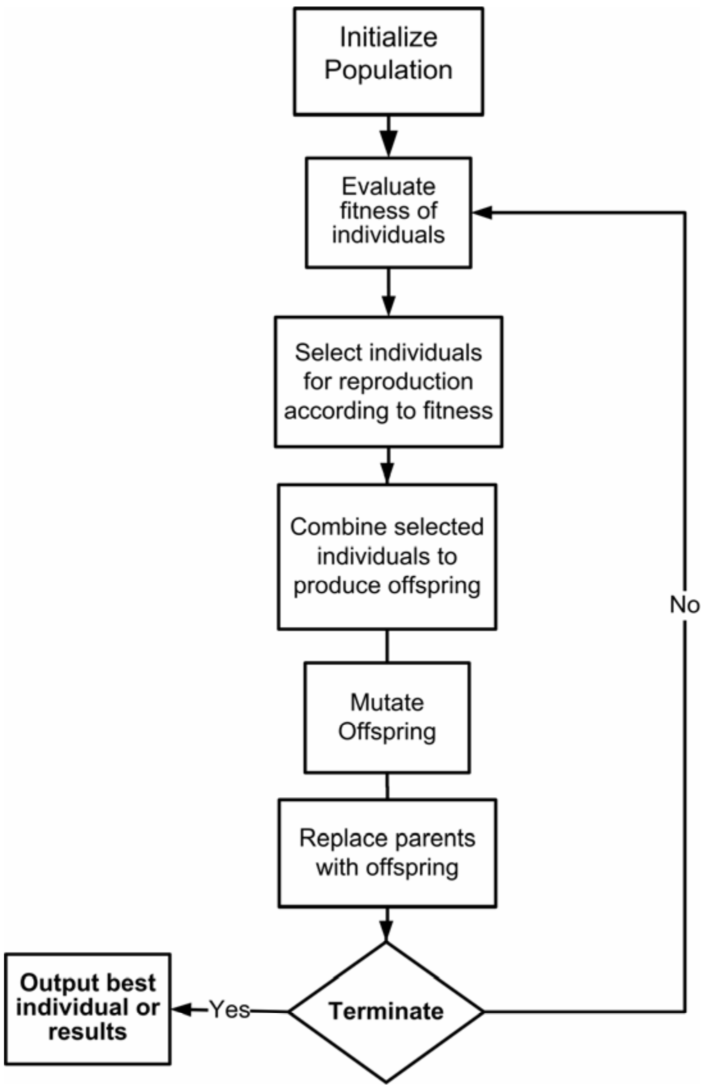

Initialize Population
Evaluate fitness of individuals
Select individuals for reproduction according to fitness
Combine selected individuals to produce offspring
Mutate Offspring
Replace parents with offspring
Terminate
Yes
No
Output best individual or results

**Figure 1. Simple Genetic Algorithm.**

Probably, the most important characteristics of genetic algorithms are the robustness—they tend to solve a wide domain of problems with relatively efficiency—and the flexibility—they do not require any especial information about the problem (e.g. derivatives, etc) besides the fitness function. Thanks to these characteristics, they have been applied to a great variety of problems.

### 2.1.1 Elements of Genetic Algorithms

#### 2.1.1.1 <u>Representation</u>

In the simple GA introduced by Holland, the individuals were represented by a string of bits. Certain number of bits represents a particular attribute or variable:

$$\underbrace{1\,0\,0\,1\,0}_{\text{var1}}\,\underbrace{1\,0\,1\,0\,1\,1\,1\,0}_{\text{var2}}\,\underbrace{1\,0\,1}_{\text{var3}}$$

Depending on the problem, each of these strings can be transformed into integers, decimals, and so on.

Usually the initial population is selected at random; every bit has an equal chance of being a ‘0’ or a ‘1’. For each individual, a fitness value is assigned according to the problem. The selection of the parents that will generate the new generation will depend on this value.

Another popular representation in GAs is the floating point representation: each gene in the individual represents a variable. This type of representation has been successfully used in optimization problems (Michalewicz and Janikow, 1996; Goldberg, 1991). It is important to note, though, that real-coded genetic algorithms require specialized operators.

### 2.1.1.2 Selection

There are different ways to select the parents. In the *fitness proportionate selection* method every individual is selected for crossover a number of times proportional to his fitness. It is usually implemented with the *roulette-wheel* sampling (also called Montecarlo Selection algorithm in Dumitrescu et al. (2000)): each solution occupies an area of a circular roulette wheel that is proportional to the individual's fitness. The roulette wheel is spun as many times as the size of the population. This method of selection has several drawbacks. During in the start of the algorithm if there are individuals that have a relatively large fitness function they will be selected many times which could cause a premature convergence due to lack of diversity. Later on the simulation run when most individuals have similar fitness function every individual will have roughly the same probability of being selected. Also, it is not compatible with negative values and it only works with maximization problems.

In *Tournament selection,* $k$ individuals are selected at random from the population. In the *deterministic Tournament selection*, the fitter of the $k$ individuals is selected. In the *nondeterministic* version, the fitter individual is selected with certain probability. Tournament selection is becoming a popular selection method because it does not have the problems of fitness-proportionate selection and because it is adequate for parallel implementations (Bäck et. al., 2000).

Other selections methods include rank-based selection, Boltzman selection, steady state selection, sigma scaling and others. For a complete survey of the different selection methods the reader can refer to Bäck et. al. (2000) and Mitchell (1998).

#### 2.1.1.3 Operators

There are two main types of operators Bäck et. al. (2000): unary, e.g., mutation and higher order operators, e.g., crossover.

Crossover involves two or more individuals that are combined together to form one or more individual. The simplest crossover type is one point crossover:

```
Parent1:    1 0 1 1 0 0 0 1 0 1 0 0 0 1
Parent2:    1 0 1 0 1 0 1 0 0 1 1 1 1 1
                      ↑
                  One point
                  crossover

Child 1:     1 0 1 1 0 0 1 0 0 1 1 1 1 1
Child 2:     1 0 1 0 1 0 0 1 0 1 0 0 0 1
```

This operator has an important shortcoming: positional bias—the bits in the extremes are always exchanged. This type of crossover is rarely used in practice (Bäck et. al 2000).

Two-point crossover is a variation of the previous operator:

```
Parent1:    1 0 1 1 0 0 0 1 0 1 0 0 0 1
Parent2:    1 0 1 0 1 0 1 0 0 1 1 1 1 1
                ↑           ↑
                  Two point
                  crossover

Child 1:     1 0 1 1 0 0 0 1 0 1 0 0 0 1
Child 2:     1 0 1 0 1 0 1 0 0 1 1 1 1 1
```

Other types include $n$-point crossover or uniform crossover. In uniform crossover, a mask determines which parent will provide each bit. For instance, one child could be formed by selecting the bit from parent1 if the corresponding bit in the mask is a 1 and selecting the bit from parent 2 if the bit in the mask is a 0. Another child could be formed by doing the inverse.

| | |
|---|---|
| Parent1 | 1 0 1 1 0 0 0 1 0 1 0 0 0 1 |
| Parent2 | 1 0 1 0 1 0 1 0 0 1 1 1 1 1 |
| Mask | 1 1 1 0 0 0 1 1 0 0 0 1 1 0 |
| | |
| Child 1 | 1 0 1 0 1 0 0 1 0 1 1 0 0 1 |
| Child 2 | 1 0 1 1 0 0 1 0 0 1 0 1 1 1 |

There is no clear "best crossover" and the performance of the GA usually depends on the problem and the other parameters as well.

Crossover is not limited to two parents, though. There have been experimental results pointing out that multiparent crossover, e.g., six parent diagonal crossover, have better performance than the one-point crossover (see Eiben, 2002 and references therein).

In the one-child version of the diagonal crossover, if there are $n$ parents, there will be $n-1$ crossover points and one child (see Figure 2)

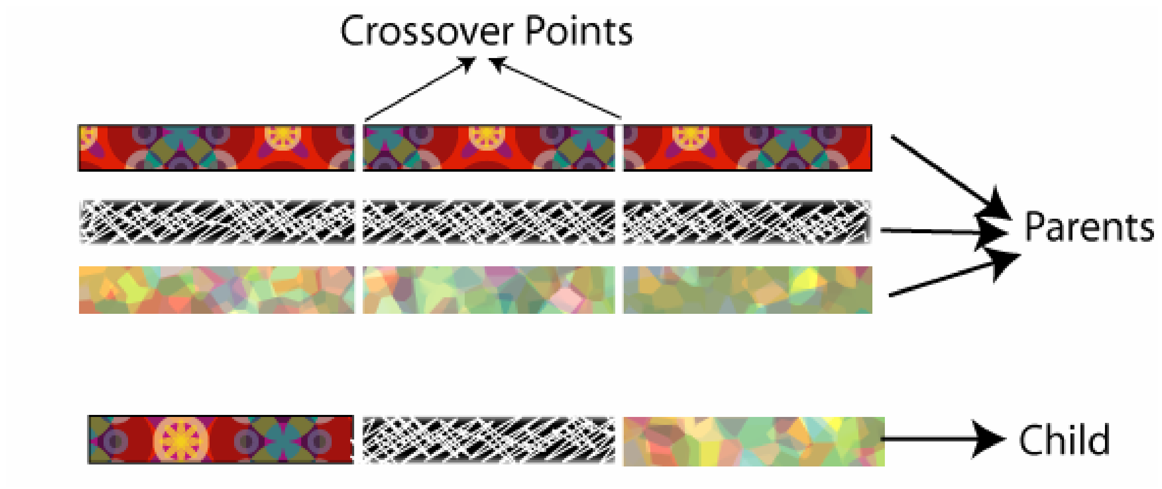


**Figure 2. Diagonal crossover with one child.**

In GAs, crossover is the main operator of variation, while mutation plays a reduced role. The simplest type of mutation is flipping a bit at each gene position with a predefined probability. Some studies have shown that varying the mutation rate can improve significantly the performance rate when compared with fixed mutation rates (see Thierens, 2002).

There are three main approaches to varying the mutation rate (Thierens, 2002):

- Dynamic parameter control in which the mutation rate is a function of the generations.
- Adaptive parameter control in which the mutation rate is modified according to a measure of how well the search is going.
- Self-adaptive parameter control in which the mutation rate is evolved together with the variables that are being optimized.

An example of a dynamic mutation rate is tested in Bäck and Schütz (1996) where the mutation rate depended on the generation according to

$$p_t = \left( 2 + \frac{n-2}{T-1} \cdot t \right)^{-1}$$

where $t$ is the current generation and $T$ is the maximum number of generations.

In the adaptive methodology, the goodness of the search is evaluated and the mutation rate, and sometimes also the crossover rate, is modified accordingly. One technique that is found to produce good results in Vasconcelos et al. (2001) measured the "genetic diversity" of the search according to the ratio of the average fitness to the best fitness or $gdm$. A value of $gdm$ close to 1 implies that all individuals have the same genetic code (or the same fitness) and the search is converging. To avoid premature convergence, it is necessary to increase exploration (by increasing the mutation rate) and to reduce the exploitation (by reducing the crossover rate). For the contrary, if the $gdm$ falls below a lower limit the crossover rate is increased and the mutation rate reduced.

In the self-adaptive methodology, several bits are added to each individual that will represent the mutation rate for that particular individual. This way the mutation rate evolves with each individual. This technique is investigated by Bäck and Schütz (1996).

Another important variation is elitism in which the best individual is copied to the next generation without modifications. This way the best solution is never lost (see, for example, Xiangrong and Fang, 2002).

## 2.2 Learning algorithms for binary classification

In binary classification problems, we are given a set of pattern vectors $\mathbf{x}_1, \dots, \mathbf{x}_m$ with associated classes $y_1, y_2, \dots, y_m \in \{1, -1\}$ called the training set. From these input vectors we try to find a function $f(\mathbf{x})$ that will minimize the number of classification errors in patterns vectors that are not in the training set. This error in this testing set is called the generalization error or expected risk and can be defined mathematically as (Vapnik, 1999)

$$R(f) = \int L(f(\mathbf{x}), y) dP(\mathbf{x}, y) \tag{1}$$

where $L(\bullet)$ is a loss function that measures the disagreement between the predicted value of the algorithm $f(\mathbf{x})$ and the correct (or desired) value $y$ and $P(\mathbf{x}, y)$ is the unknown joint probability distribution from which the patterns vectors are being generated. The goal is to find a function $f(\mathbf{x})$ that minimizes the risk $R(f)$.

The problem is that the risk cannot be minimized directly since the joint probability $P(\mathbf{x}, y)$ is unknown. Therefore, an induction principle is needed to achieve this goal indirectly. One induction principle that may be used for large sample sizes is the empirical risk minimization (ERM) induction principle where the empirical risk is given by

$$R_{emp}(f) = \frac{1}{m}\sum_{i=1}^{m} L\left(f(\mathbf{x}_i), y_i\right) \tag{2}$$

The empirical error measures the average error in the training set.

The expected risk is bounded by the empirical risk according to the following equation (Vapnik, 1999):

$$R(f) \leq R_{emp}(f) + \phi(f) \tag{3}$$

where the second term is the confidence interval. The confidence interval depends on the complexity of the type of functions being used (Vapnik, 1999).

For large sample sizes, Statistical Learning theory (Vapnik, 1999) tells us that there are functions that will minimize the expected risk and for which the empirical risk will asymptotically converge to the expected risk. In other words, the expected risk $R(f)$ (Eq. 1) can be approximated by the empirical risk $R_{emp}(f)$ (Eq. 2). In fact, as Vapnik (1999) points out, several classical methods for solving specific learning problems are based on the empirical risk such as the least squares and the maximum likelihood methods.

Nevertheless, this may not be the case for small sample sizes. If the selected classification function is too complex, the empirical risk may be minimized to zero even though the confidence interval may be large, which implies that the errors in the testing set could be considerably high. This is known as overfitting. In order to avoid overfitting, it is required to keep the functions simple. On the other hand, if the functions are very simple, it will be difficult to minimize the empirical risk. Therefore, we need a tradeoff between the complexity of the decision function and how close we approximate the training set. This search for this tradeoff is known as the Structural Risk Minimization principle (Vapnik, 1999).

There are two approaches that can be used to minimize the bound (Eq. 3) (Vapnik, 1999):

- The confidence interval is fixed by selecting a particular set of functions while minimizing the training set. Neural networks implement this approach. First, an architecture is selected (feedforward or recurrent; single layer or multilayer; how many layers; how many nodes per hidden layer, and so on) that will define the confidence interval or complexity of the decision functions. Then, we minimize the error in the training set.

- Or, the empirical risk is fixed while minimizing the confidence interval, e.g. support vector machines. In this case, we need to control the tradeoff between the complexity of the set of decision functions and the number of classification errors in the training set.

## 2.3 Support Vector Classification

In support vector classification, the pattern vectors $\mathbf{x}_1, \dots, \mathbf{x}_m$ are first mapped into a high-dimensional space, also known as feature space, by means of a predefined mapping function $\Phi(\bullet)$. In this-high dimensional space, we try to find a linear decision function

$$f_{\boldsymbol{\omega},b}(\mathbf{x}) = \operatorname{sign}\left(\left\langle \mathbf{w} \cdot \Phi\left(\mathbf{x}\right)\right\rangle + b\right) \tag{4}$$

that will classify correctly the pattern vectors. In the Eq. 4, $\mathbf{w}$ is an orthogonal vector to the hyperplane

$$\left\langle \mathbf{w} \cdot \Phi\left(\mathbf{x}\right)\right\rangle + b = 0 \tag{5}$$

(see Figure 3 for an hyperplane in two dimensions) and $b$ is a threshold. The reason for the mapping is to transform any dataset into a linearly separable dataset so that it can be classified correctly by a linear function which is a simple function for which the bounds of the expected risk can be calculated and minimized.

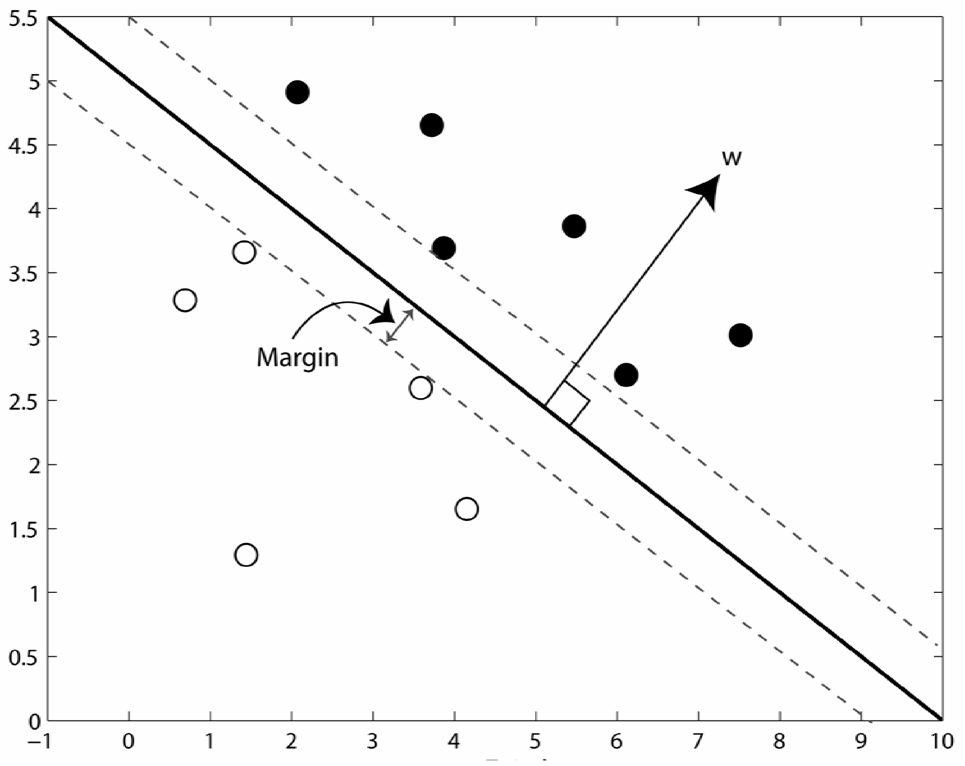


**Figure 3. Hyperplane in two dimensions.**

For hyperplanes, the complexity term or confidence interval in Eq. 3 is a function of the margin (see Figure 3) which can be measure by the length of the vector $\mathbf{w}$ (Vapnik, 1999; Muller et al., 1997; Muller et al., 2001). Therefore, we can rewrite Eq. 3 as

$$R(f) \leq \sum_{i=1}^{m} L(f(\mathbf{x_i}) - y_i) + \lambda \|\mathbf{w}\|^2 \qquad \textbf{(6)}$$

where $L$ is a lost function, $\lambda$ is a regularization constant, and $m$ is the number of training points (Muller et al., 1999).

There is certain degree of freedom in selecting the hyperplane related with the fact that we can multiply both $\mathbf{w}$ and $b$ by the same non-zero constant without affecting the decision function. To find a unique solution, the canonical hyperplane is introduced:

$(\mathbf{w}, b)$ is a canonical form of the hyperplane $\langle \mathbf{w} \cdot \Phi(\mathbf{x}) \rangle + b = 0$ with respect to $\mathbf{x}_1, \ldots, \mathbf{x}_m$ if it is scaled so that

$$\min_{i=1,\ldots,m} \left| \langle \mathbf{w} \cdot \Phi(\mathbf{x}_i) \rangle + b \right| = 1,$$

which means that the point closest to the hyperplane has a distance of $1/\|\mathbf{w}\|$ (Schölkopf, B. and A.J. Smola, 2002). This last quantity is known as the margin of the classifier.

By using canonical hyperplanes, the condition that the empirical risk is zero (perfect classification) for the function in Eq. 4 can be expressed as:

$$y_i \left( \langle \mathbf{w} \cdot \Phi(\mathbf{x}_i) \rangle + b \right) \geq 1, \; i = 1, \ldots, m \qquad \textbf{(7)}$$

The hyperplane that satisfies Eq. 7 and at the same time minimize $\mathbf{w}$ is called the *maximal margin hyperplane* or *optimal hyperplane.*

In order to find this optimal hyperplane we need to solve the following optimization problem:

$$\begin{aligned} &\text{Minimize} \quad \frac{1}{2} \langle \mathbf{w} \cdot \mathbf{w} \rangle \\ &\text{subject to} \quad y_i \left( \langle \mathbf{w} \cdot \Phi(\mathbf{x}_i) \rangle + b \right) \geq 1, \; i = 1, \ldots, m \end{aligned} \qquad \textbf{(8)}$$

One way to solved this constrained optimization problem is by using the generalized Lagrangian

$$L(\mathbf{w}, b, \alpha) = \langle \mathbf{w} \cdot \mathbf{w} \rangle - \sum_{i=1}^{m} \alpha_i \left( y_i \left( \langle \mathbf{w} \cdot \Phi(\mathbf{x}_i) \rangle + b \right) - 1 \right) \qquad \textbf{(9)}$$

where $\alpha_i$ are the Lagrange multipliers.

We need to minimize $L(\mathbf{w}, b, \alpha)$ subject to $\alpha_i \geq 0$. Eq. 9 is called a convex quadratic programming problem because the objective function is convex and have a quadratic dependence on the $\alpha$ and the constraints are linear and, therefore, also forms a convex set (Burges, 1998).

Mathematically, a function is convex if (Bazaraa et al., 1994)

$$f[\lambda \mathbf{x}_1 + (1-\lambda)\mathbf{x}_2] \leq \lambda f(\mathbf{x}_1) + (1-\lambda) f(\mathbf{x}_2)$$

where $\mathbf{x}_1$ and $\mathbf{x}_2$ are elements of a convex set and $\lambda \in [0,1]$ (see Figure 4 and Figure 5).

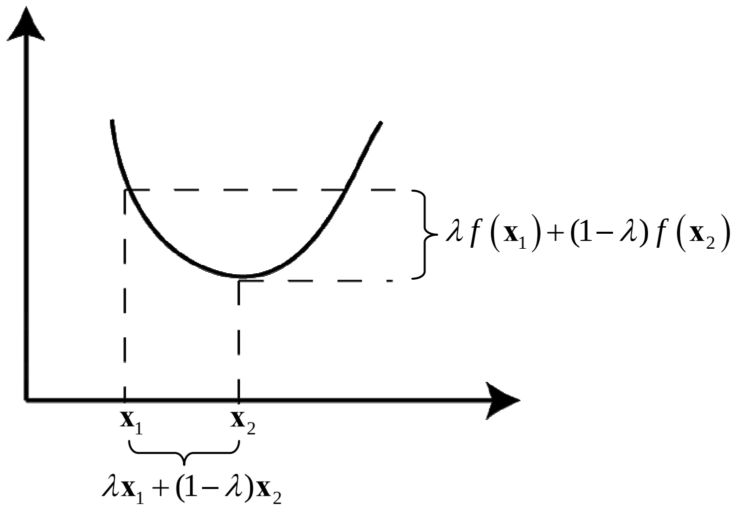

$\lambda f(\mathbf{x}_1) + (1-\lambda) f(\mathbf{x}_2)$
$\mathbf{x}_1$
$\mathbf{x}_2$
$\lambda \mathbf{x}_1 + (1-\lambda)\mathbf{x}_2$

**Figure 4. A convex function.**

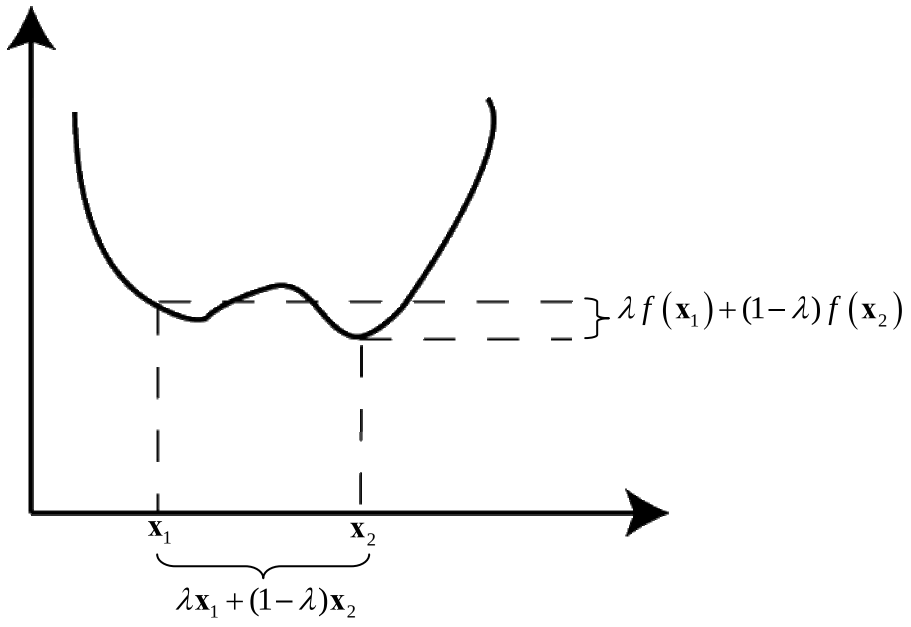


**Figure 5. A non-convex function.**

A set is convex if for each $\mathbf{x}_1$ and $\mathbf{x}_2$ that belongs to that set, the line segment $\lambda \mathbf{x}_1 + (1-\lambda)\mathbf{x}_2$ also belongs to that set. In other words, any line segment that we can draw between two points of that set is inside the set (see Figure 6).

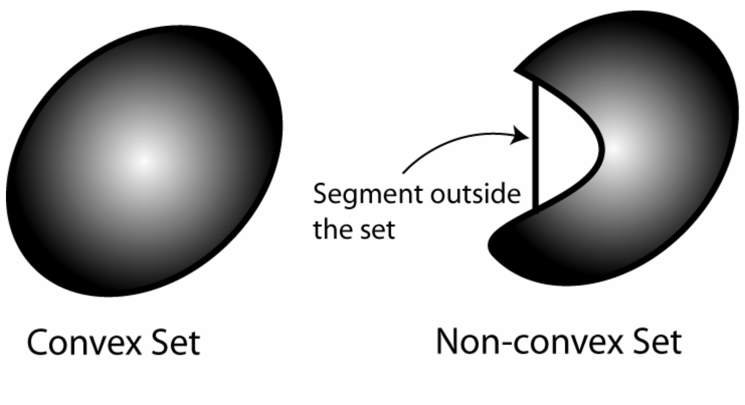


**Figure 6. Convex and non-convex sets.**

For convex problems, the Kuhn-Tucker Theorem (KKT) gives us the necessary and sufficient conditions of optimality (Burges, 1998; Cristianini and Shawe-Taylor, 2000):

$$\begin{aligned}
&\frac{\partial L(\mathbf{w},b,\boldsymbol{\alpha})}{\partial \mathbf{w}} = \mathbf{w} - \sum_{i=1}^{m} y_i \alpha_i \mathbf{x}_i = 0, \\
&\frac{\partial L(\mathbf{w},b,\boldsymbol{\alpha})}{\partial b} = \sum_{i=1}^{m} y_i \alpha_i = 0 \\
&y_i\left(\left\langle \mathbf{w}\cdot\Phi\left(\mathbf{x}_i\right)\right\rangle + b\right) \geq 1 \qquad i = 1,\ldots,m \\
&\alpha_i\left(y_i\left(\left\langle \mathbf{x}_i\cdot\mathbf{w}\right\rangle + b\right) - 1\right) = 0 \quad \forall i \\
&\alpha_i \geq 0 \qquad\qquad\qquad\qquad \forall i
\end{aligned} \tag{10}$$

From the first two equations, we have that

$$\mathbf{w} = \sum_{i=1}^{m} y_i \alpha_i \mathbf{x}_i$$

with

$$\sum_{i=1}^{m} y_i \alpha_i = 0\,.$$

Replacing the primal variables in the Lagrangian Eq (9), one obtains the dual as follows:

$$\begin{aligned}
L(\mathbf{w},b,\alpha) &= \frac{1}{2}\sum_{i,j=1}^{m} y_i y_j \alpha_i \alpha_j \left\langle \Phi\left(\mathbf{x}_i\right)\cdot\Phi\left(\mathbf{x}_j\right)\right\rangle - \sum_{i=1}^{m}\alpha_i\left(y_i\sum_{j=1}^{m} y_j\alpha_j\left\langle\Phi\left(\mathbf{x}_j\right)\cdot\Phi\left(\mathbf{x}_i\right)\right\rangle + y_i b - 1\right) \\
&= \frac{1}{2}\sum_{i,j=1}^{m} y_i y_j \alpha_i \alpha_j \left\langle \Phi\left(\mathbf{x}_i\right)\cdot\Phi\left(\mathbf{x}_j\right)\right\rangle - \sum_{i=1}^{m}\sum_{j=1}^{m}\alpha_i\alpha_j y_i y_j\left\langle\Phi\left(\mathbf{x}_i\right)\cdot\Phi\left(\mathbf{x}_j\right)\right\rangle - b\sum_{i=1}^{m}\alpha_i y_i + \sum_{i=1}^{m}\alpha_i \\
&= \sum_{i=1}^{m}\alpha_i - \frac{1}{2}\sum_{i,j=1}^{m} y_i y_j \alpha_i \alpha_j \left\langle \Phi\left(\mathbf{x}_i\right)\cdot\Phi\left(\mathbf{x}_j\right)\right\rangle
\end{aligned}$$

Therefore, the dual programming problem of the primal (Eq. 8) can be stated as

$$\begin{aligned} &\text{Maximize } \sum_{i=1}^{m}\alpha_i - \frac{1}{2}\sum_{i,j=1}^{m} y_i y_j \alpha_i \alpha_j \left\langle \Phi\left(\mathbf{x}_i\right)\cdot\Phi\left(\mathbf{x}_j\right)\right\rangle \\ &\text{subject to } \sum_{i=1}^{m} y_i\alpha_i = 0, \\ &\qquad \alpha_i \geq 0, i = 1,\dots,m, \end{aligned} \tag{11}$$

where expression $K\left(\mathbf{x}_i,\mathbf{x}_j\right) = \left\langle \Phi\left(\mathbf{x}_i\right)\cdot\Phi\left(\mathbf{x}_j\right)\right\rangle$ is known as the kernel. If this optimization problem is strictly convex, there is only one global solution; if not, there could be several, equally good, solutions.

The third equation in (10) is known as the Karush-Kuhn-Tucker complementary condition and it provides insights about the form of the solution. This condition states that the optimal solutions must satisfy

$$\alpha_i\left(y_i\left(\langle \mathbf{x}_i\cdot\mathbf{w}\rangle + b\right) - 1\right) = 0\,,$$

which implies that only the points that are closest to the hyperplane (with margin of 1 since we are using canonical hyperplanes) will have $\alpha_i > 0$. These points are called *support vectors.* All other points will have $\alpha_i = 0$. This property is called *sparseness.* As Cristianini and Shawe-Taylor (2000) stated, each Lagrange multiplier gives a measure of how important is a given training point in finding the classifying function. Clearly, a training point with $\alpha_i = 0$ is not important and can be eliminated without altering the final decision function.

By using the KKT complementary condition, $b$ is determined by averaging (Schölkopf, B. and A.J. Smola, 2002)

$$b = y_j - \sum_{i=1}^{m} y_i\alpha_i K(\mathbf{x}_j,\mathbf{x}_i) \tag{12}$$

over all points with $\alpha_j > 0$.

Finally, the decision function Eq. (4) may be rewritten in terms of the kernel and the Lagrange multiplier as

$$f(\mathbf{x}) = \operatorname{sign}\left(\sum_{i=1}^{m} \alpha_i y_i K\left(\mathbf{x}, \mathbf{x}_i\right) + b\right)$$

or

$$f(\mathbf{x}) = \operatorname{sign}\left(\sum_{i \in sv} \alpha_i y_i K\left(\mathbf{x}, \mathbf{x}_i\right) + b\right). \tag{13}$$

This decision function only depends on the support vectors. It is not needed to keep the rest of the input vectors once the training completes. All the information required to find the optimal hyperplane lies in the support vectors. Figure 7 depicts the decision function in terms of a one-layer network or *Perceptron*.

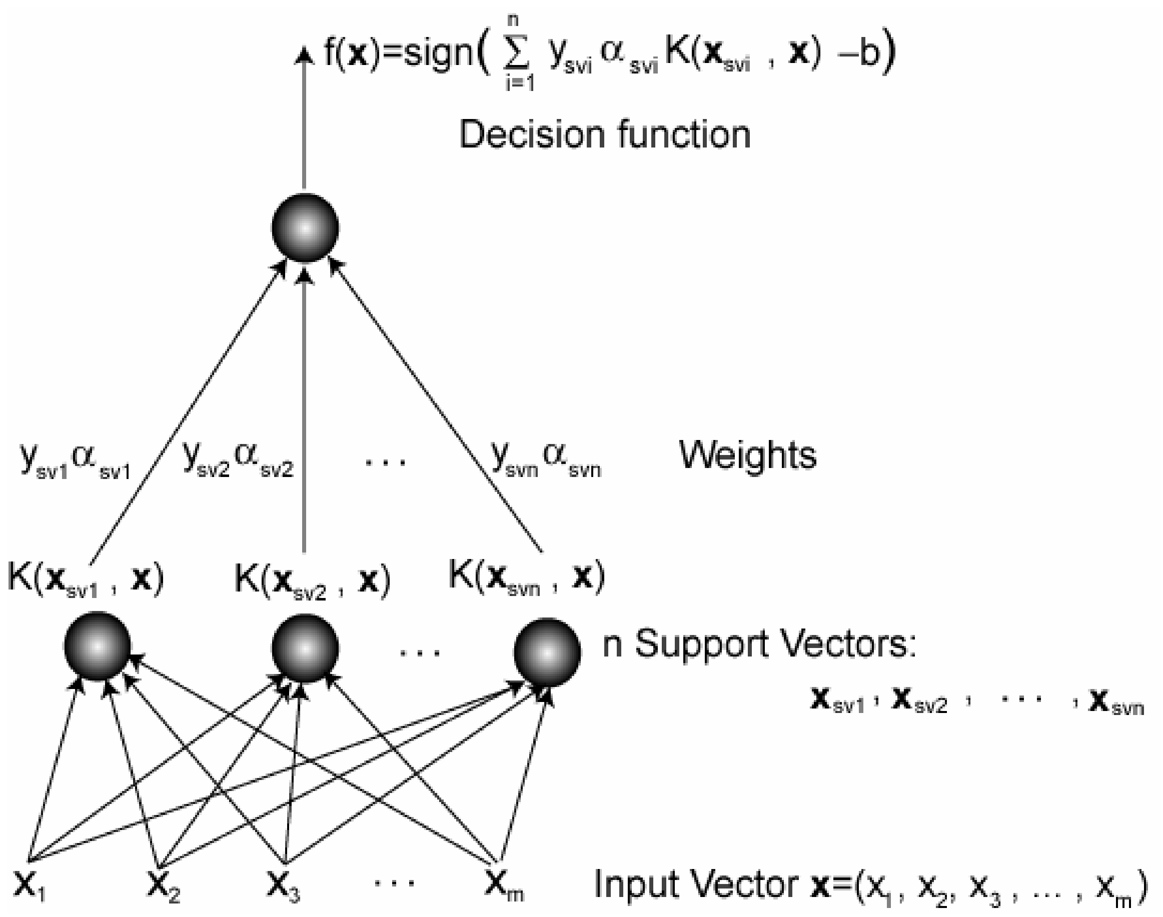

$f(\mathbf{x})=\text{sign}\left(\sum_{i=1}^{n} y_{svi}\,\alpha_{svi}\,K(\mathbf{x}_{svi}, \mathbf{x}) - b\right)$
Decision function
$y_{sv1}\alpha_{sv1}$
$y_{sv2}\alpha_{sv2}$
. . .
$y_{svn}\alpha_{svn}$
Weights
$K(\mathbf{x}_{sv1}, \mathbf{x})$
$K(\mathbf{x}_{sv2}, \mathbf{x})$
$K(\mathbf{x}_{svn}, \mathbf{x})$
n Support Vectors:
$\mathbf{x}_{sv1}, \mathbf{x}_{sv2}, \dots, \mathbf{x}_{svn}$
$x_1$ $x_2$ $x_3$ . . . $x_m$
Input Vector $\mathbf{x}=(x_1, x_2, x_3, \dots, x_m)$

**Figure 7. Network representation of the SVM architecture (based on drawing from Vapnik (1999)).**

### 2.3.1 Soft Margin Classifiers

So far, we have assumed that there exists a hyperplane capable of separating the data correctly, i.e., we assume that the training data is linearly separable in the feature space. Nonetheless, if the data is noisy or have outliers, we may not be able to find a solution or we may tend to overfit it. Furthermore, in the previous section, we stated the need of a control variable that will determine the tradeoff between how well the training points are approximated and how complex is the decision function.

To ignore noisy data and outliers, the constraint in the primal optimization problem is modified to include a slack variable $\xi_i$:

$$\begin{aligned} &\text{Minimize} \quad \frac{1}{2}\langle \mathbf{w}\cdot\mathbf{w}\rangle + C\sum_{i=1}^{m}\xi_i \\ &\text{subject to} \quad y_i\left(\langle \mathbf{w}\cdot\Phi(\mathbf{x}_i)\rangle + b\right) \geq 1-\xi_i,\ i=1,\ldots,m, \\ &\qquad\qquad\quad \xi_{\mathrm{i}} \geq 0, i=1,\ldots,m, \end{aligned} \tag{14}$$

where a penalty term has been added in the objective function and where $C$ is the penalty value for the slack variables selected a priori. Choosing a particular value of $C$ corresponds to selecting the complexity of the function (by selecting a $\|\mathbf{w}\|$ value) and then minimizing the slack variables $\xi_i$ (see also Cristianini and Shawe-Taylor, 2000). Usually, we will try different values of $C$ in order to find the optimal one for a particular problem.

The corresponding dual can be found following a procedure similar to the one used for the maximal margin classifier case.

The corresponding Lagrangian is:

$$L(\mathbf{w},b,\boldsymbol{\xi},\boldsymbol{\alpha},\mathbf{r}) = \frac{1}{2}\langle \mathbf{w}\cdot\mathbf{w}\rangle + C\sum_{i=1}^{m}\xi_i - \sum \alpha_i\left(y_i\langle \mathbf{x}_i\cdot\mathbf{w}\rangle + b - 1 + \xi_i\right) - \sum_{i=1}^{m} r_i\xi_i$$

with $\alpha_i, r_i \geq 0$. $r_i$ is an extra Lagrange multiplier to take into account the extra constraint. To find the dual problem, we need to apply the KKT conditions:

$$
\begin{aligned}
\frac{\partial L(\mathbf{w},b,\boldsymbol{\xi},\boldsymbol{\alpha},\mathbf{r})}{\partial \mathbf{w}} &= \mathbf{w} - \sum_{i=1}^{m} y_i \alpha_i \mathbf{x}_i = 0, \\
\frac{\partial L(\mathbf{w},b,\boldsymbol{\xi},\boldsymbol{\alpha},\mathbf{r})}{\partial \xi_i} &= C - \alpha_i - r_i = 0, \\
\frac{\partial L(\mathbf{w},b,\boldsymbol{\xi},\boldsymbol{\alpha},\mathbf{r})}{\partial b} &= \sum_{i=1}^{m} y_i \alpha_i = 0.
\end{aligned}
\tag{15}
$$

From the second expression in Eq. 15 and noting that $r_i \geq 0$ it is easy to show that $\alpha_i \leq C$. From the KKT complementary conditions we obtain

$$
\begin{aligned}
&\alpha_i \left( y_i \left\langle \mathbf{x}_i \cdot \mathbf{w} \right\rangle + b - 1 + \xi_i \right) = 0, i = 1, \ldots, m, \\
&r_i \xi_i = 0, i = 1, \ldots, m.
\end{aligned}
$$

Using the second equation, it is clear that $\xi_i \neq 0$ only when $r_i = 0$ and $\alpha_i = C$.

Therefore, the dual problem of the primal (14) is

$$
\begin{aligned}
&\text{Maximize } \sum_{i=1}^{m} \alpha_i - \frac{1}{2} \sum_{i,j=1}^{m} y_i y_j \alpha_i \alpha_j K\left(\mathbf{x}_i, \mathbf{x}_j\right) \\
&\text{subject to } \sum_{i=1}^{m} y_i \alpha_i = 0, \\
&\qquad 0 \leq \alpha_i \leq C, i = 1, \ldots, m.
\end{aligned}
\tag{16}
$$

This formulation is usually known as the box constraint (Cristianini and Shawe-Taylor, 2000) because of the bounds in the values of $\alpha_i$. It is interesting to note that the only difference between Eq. 11 and Eq. 15 is the upper bound on $\alpha_i$.

### 2.3.2 Training of the SVM

The training of the SVM consists on solving the quadratic programming problem given in Eq. 16 which can be rewritten as

$$\begin{aligned} &\text{Minimize } \frac{1}{2}\boldsymbol{\alpha}^T\mathbf{Q}\boldsymbol{\alpha} - \mathbf{e}^T\boldsymbol{\alpha} \\ &\text{subject to } \mathbf{y}^T\boldsymbol{\alpha} = 0 \\ &\qquad 0 \le \alpha_i \le C \text{ for } u = 1,\dots,m \end{aligned} \qquad \mathbf{17}$$

where $\mathbf{Q}$ is a matrix with elements $q_{ij} = K(\mathbf{x}_i, \mathbf{x}_j)y_i y_j$, and $\mathbf{e}$ is a vector of ones.

Another way of seeing the convexity of this problem is by studying the properties of the matrix $\mathbf{Q}$ (Platt, 1998):

- Matrix Q may be positive definitive which means that it have a unique minimum or it may be positive semidefinite which means it have a set of equivalent minima.
- As mentioned before, there is an optimality condition that describe this minima and it is given by the Karush Kuhn Tucker theorem.

Because of these properties, there are no local minima and there are efficient ways to find the solution. The solution of Eq. 17 could be obtained simply by using gradient descent, conjugate gradient and several other methods. However, it is important to note that there are as many unknown variables $\alpha$ as training points. For some real-world problems the matrix $\mathbf{Q}$ could be huge. For example, with 10,000 training points, we would need a matrix with 100 million points. For these situations, we need methods capable of using subsets of the matrix at a time instead of using the complete matrix.

There are two ways to solve this optimization problem (Platt, 1998):

1. Using specialized structures. Kaufman (1998) shows that when the matrix $\mathbf{Q}$ is quadratic and the training vectors are available, there is no need to store the matrix $\mathbf{Q}$ when doing matrix-vector multiplication with $\mathbf{Q}$. A pseudo-algorithm for this is found in Kaufman (1998).

2. Decomposition methods. These methods are based on the fact that if we knew in advance which points are support vectors, the problem could be simplified. Therefore, many strategies have been developed to somehow guess the support vectors and restrict the training to this subset. In other words, several smaller optimization problems are solved instead of one big problem. Cristianini and Shawe-Taylor (2000) considers the sequential minimal optimization (SMO) algorithm a decomposition method taken to the extreme since a subset of just two points is optimized at each iteration. This provides the advantage that the optimization of two data points can be solved analytically. This means that quadratic optimizers are no longer needed. Several heuristics are used to determine which two points should be used at each iteration.

For detailed implementation issues, the reader can refer to Cristianini and Shawe-Taylor (2000), Schölkopf and Smola (2002), Kaufman (1998), Platt (1998), and Burges(1998).

### 2.3.3 More about Kernels

Since support vector machines are linear classifiers, it is necessary to map the input vectors with a nonlinear mapping in order to learn non-linear relations. The resulting vectors are usually called *features.* The problem with using mappings is that the dimensionality usually increases, degrading the computational performance of the algorithm. For example, using a complete polynomial kernel of the form

$$k(u,v) = \left(\langle u \cdot v \rangle + c\right)^p$$

the feature space would have a dimensionality of (Herbrich, 2001; Cristianini and Shawe-Taylor, 2000)

$$\binom{n+p}{p},$$

where $n$ is the dimensionality of the input vectors. Input vectors with a dimensionality of 10 transformed by a polynomial kernel of degree 3 would have a feature of dimensionality 286.

A kernel is a function $K$, such that for all $\mathbf{x},\ \mathbf{z}$

$$K(\mathbf{x},\mathbf{z}) = \left\langle \Phi\left(\mathbf{x}\right) \cdot \Phi\left(\mathbf{z}\right) \right\rangle .$$

Kernels are symmetric (Cristianini and Shawe-Taylor, 2000), i.e.,

$$K(\mathbf{x},\mathbf{z}) = \left\langle \phi(\mathbf{x}) \cdot \phi(\mathbf{z}) \right\rangle = \left\langle \phi(\mathbf{z}) \cdot \phi(\mathbf{x}) \right\rangle = K(\mathbf{z},\mathbf{x})$$

and follow the Cauchy-Schwarz inequality

$$\begin{aligned} K(\mathbf{x},\mathbf{z})^2 &= \left\langle \phi(\mathbf{x}) \cdot \phi(\mathbf{z}) \right\rangle^2 \le \left\| \phi(\mathbf{x}) \right\|^2 \left\| \phi(\mathbf{z}) \right\|^2 \\ &= \left\langle \phi(\mathbf{x}) \cdot \phi(\mathbf{x}) \right\rangle \left\langle \phi(\mathbf{z}) \cdot \phi(\mathbf{z}) \right\rangle = K(\mathbf{x},\mathbf{x}) K(\mathbf{z},\mathbf{z}) \end{aligned} .$$

By using a kernel we do not need to work with the features directly. The kernel maps the input data *implicitly* into a feature space avoiding the computational problems mentioned above. This is possible because in the dual representation the algorithm has been expressed in terms of inner products.

There are several ways to find kernels (Herbrich, 2001; Schölkopf and Smola (2002)):

1. choosing a mapping $\phi$ that will explicitly gives us a kernel $K$, e.g, the polynomial kernel,

2. choosing a kernel $K$ which implicitly corresponds to a fixed mapping $\phi$, e.g., the Radial Basis Function (RBF) kernel
3. combining several simple kernels to form a more complicated kernel. It is possible to create kernels by using the following properties:
   a. $K(\mathbf{x},\mathbf{z}) = c_1 K_1(\mathbf{x},\mathbf{z}) + c_2 K_2(\mathbf{x},\mathbf{z})$ for $c_1, c_2 \geq 0$
   b. $K(\mathbf{x},\mathbf{z}) = c \cdot K_1(\mathbf{x},\mathbf{z})$ for all $c \in \mathbb{R}$
   c. $K(\mathbf{x},\mathbf{z}) = K_1(\mathbf{x},\mathbf{z}) \cdot K_2(\mathbf{x},\mathbf{z})$

Two of the most commonly used kernels are the Polynomial and the Radial Basis function kernel.

### 2.3.3.1 Polynomial Kernel

For some applications like visual pattern recognition (Schölkopf and Smola, 2002), most information is contained in monomials of degree $d$. So the input vectors are first mapped into monomials of degree $d$,

$$(x_1, x_2) \rightarrow (x_1{}^2, x_2{}^2, x_1 x_2).$$

Using this mapping, if the inputs are in $N$ dimensions, the feature space will have a dimensionality of

$$\binom{d+N-1}{d}.$$

It is possible to construct a polynomial kernel that will accomplish this mapping implicitly (for a proof see p. 27 of Schölkopf and Smola, 2002):

$$K(\mathbf{x},\mathbf{z}) = \langle \mathbf{x} \cdot \mathbf{z} \rangle^d .$$

The complete polynomial kernel, i.e., a kernel that will map the input vectors to the space of all monomials up to degree $d$, can be constructed similarly:

$$K(\mathbf{x},\mathbf{z}) = \langle \mathbf{x}\cdot\mathbf{z} + c\rangle^{d}$$

Figure 8 shows a toy example of an application of the polynomial kernel.

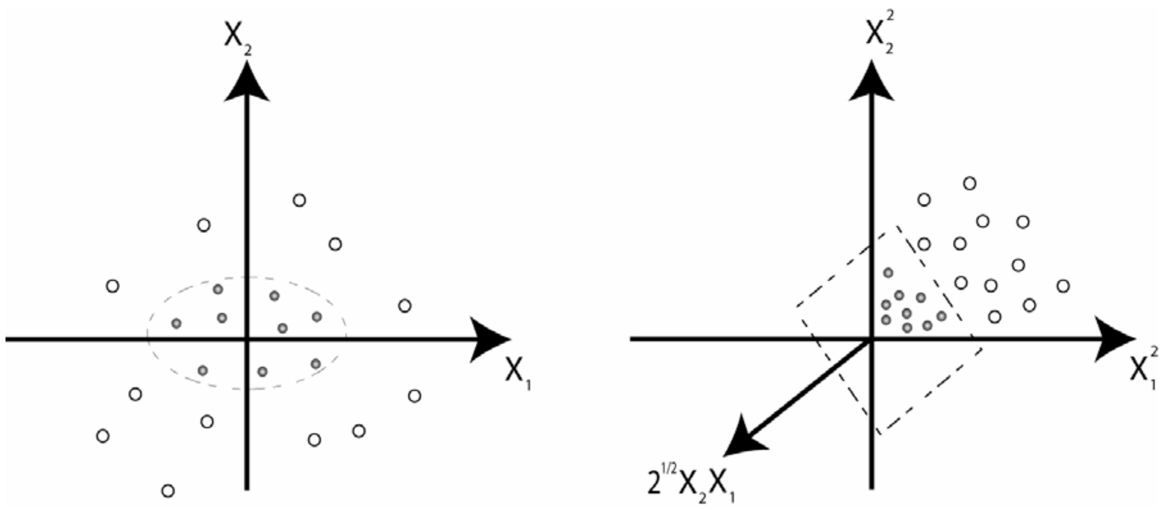


**Figure 8. Toy Example showing the importance of the polynomial kernel in allowing a linear decision function to classify input vectors with nonlinear relations. (based on drawing from Schölkopf and Smola, 2002).**

### 2.3.3.2 <u>Gaussian Radial Basis Function Kernel</u>

Radial basis function (RBF) kernels are those that are a function of a distance measure (Schölkopf and Smola, 2002):

$$K(\mathbf{x},\mathbf{z}) = f(d(\mathbf{x},\mathbf{z}))$$

where $d(\cdot)$ is a metric.

A popular example of a RBF kernel is the Gaussian kernel:

$$K(\mathbf{x},\mathbf{z}) = e^{\left(-\frac{\|\mathbf{x}-\mathbf{z}\|^2}{2\sigma^2}\right)}$$

or

$$K(\mathbf{x},\mathbf{z}) = e^{\left(-\gamma\|\mathbf{x}-\mathbf{z}\|^2\right)}$$

Gaussian RBF kernels have several important properties:

- Since $K(\mathbf{x},\mathbf{x}) = 1$ each mapped input vector have a unit length, $\|\Phi(\mathbf{x})\| = 1$.
- If all input vectors are different, the mapped points $\Phi(x_1), \Phi(x_2), \ldots, \Phi(x_m)$ are linearly independent.
- The feature space have infinite dimension.
- RBF kernels are shift invariant: $\|(\mathbf{x}+\mathbf{a})-(\mathbf{z}+\mathbf{a})\|^2 = \|\mathbf{x}-\mathbf{z}\|^2$
- If we assume that all input vector are different and that $l_1 > l_2 + 1 > 2$ where $l_1$ and $l_2$ are the number of input vectors belonging to class 1 and 2, respectively, then for any value of C and $\sigma^2$ (or $\gamma$) there is a unique solution (Keerthi and Lin, 2003).

The parameter $\gamma$ controls the smooth level of the kernel as seen in Figure 9.

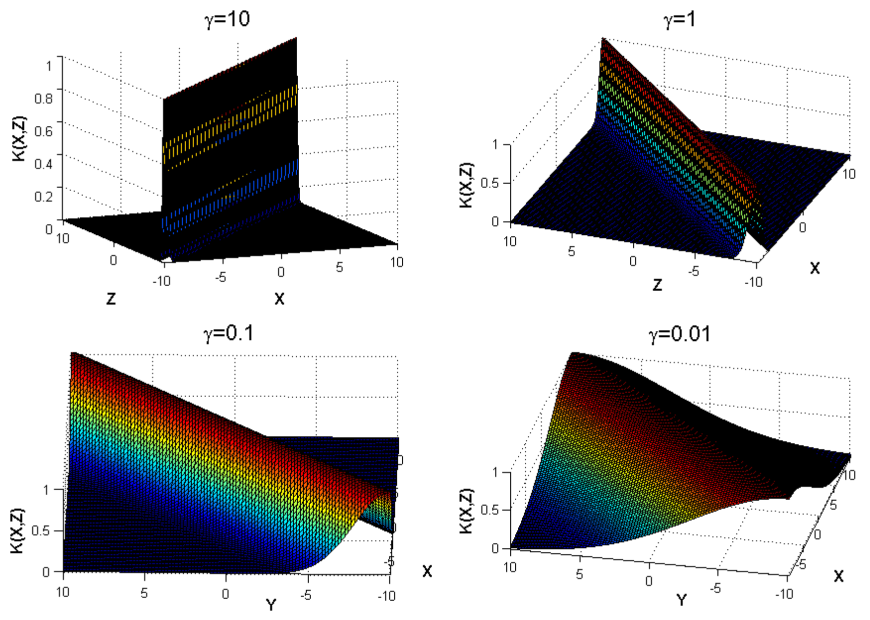


**Figure 9. The parameter $\gamma$ of the Gaussian RBF Kernels control the smoothing of the function. In this case the input points are 1-dimensional points.**

Table 1 summarizes the properties of the kernels previously mentioned.

**Table 1. Summary of some important kernels.**

| Name | Kernel function | Feature Space Dimension |
|---|---|---|
| **pth degree polynomial** | $K(\mathbf{x},\mathbf{z}) = \langle \mathbf{x}\cdot\mathbf{z}\rangle^d$ | $\binom{d+N-1}{d}$ |
| **Complete or inhomogeneus polynomial** | $K(\mathbf{x},\mathbf{z}) = \langle \mathbf{x}\cdot\mathbf{z}+c\rangle^d$ | $\binom{d+N}{d}$ |
| **Gaussian RBF** | $K(\mathbf{x},\mathbf{z}) = e^{\left(-\frac{\lVert\mathbf{x}-\mathbf{z}\rVert^2}{2\sigma^2}\right)}$ | $\infty$ |

## 2.4 More on the Generalization Error

As it was seen in the previous section there are several parameters that need to be set before starting the training like, for instance, the type of kernel, the kernel-specific parameters, and the $C$ value. The generalization performance of the SVM depends on the value of these parameters. For this reason, we need a way to measure the generalization error of the learning algorithm in order to compare and select the best model. There are several theoretical bounds on the generalization error of SVMs but they are usually the worst case scenario (Joachims, 1999) and their accuracy is still a subject of research (Cristianini and Shawe-Taylor, 2000). In any case, the estimate of the performance error should have a low bias and a low variance.

The generalization error estimation technique that we should use depends on the amount of data available. If we have a lot of data points we can simply set aside part of it for testing purposes and use the rest as training. Usually, however, this is not the case so we need to use resampling techniques to estimate this error.

Widely used techniques for estimating the generalization error of any learning algorithms includes the hold-out testing, Montecarlo Crossvalidation, k-fold crossvalidation, leave one out and, stratified crossvalidation.

It is also important to note that there is a *minimum error rate* also known as Bayes's optimal error rate which is a fixed but unknown quantity (Martin and Hirschberg, 1996). This error exists due to errors in the dataset or insufficient data. An optimal classifier would have an error rate equal to Bayes's error rate.

### 2.4.1 Hold-Out Testing

In this method, also called single validation estimate (Chapelle et al., 2002) or test sample estimation (Kohavi, 1995), the data is partitioned in two mutually exclusive subsets: the training set and the testing set (also called the validation set or the holdout set). Usually 2/3 of the original data is used for training and 1/3 for testing. If the testing set is $\{\mathbf{x}'_i, \mathbf{y}'_i\}_{1 \le i \le p}$ for SVMs the estimate is

$$T = \frac{1}{p} \sum_{i=1}^{p} \Psi(-y_i f(\mathbf{x}'_i))$$

where $\Psi$ is the step function: $\Psi(x) = 1$ when $x > 0$ and $\Psi(x) = 0$ otherwise. This is a pessimistic estimator because only a portion of the dataset is given to the learning machine for training. A variation, called the random subsampling (also called Monte-Carlo Crossvalidation in Lendasse et al. (2003)), consists on randomly splitting the data into training a testing set $k$ times and the performance of each holdout is averaged.

### 2.4.2 Cross-validation and Leave One out

In *k*-fold cross-validation the dataset is divided into *k* subsets of approximately the same size. The learning algorithm is then trained on $k-1$ subsets and the resulting decision function is tested on the remaining subset. Each subset is left out once so that it is required to do *k* trainings. The average performance of those k trainings is the $k$ -fold cross validation estimate.

In complete crosssvalidation all possible combinations, $\binom{n}{n/k}$, of observations are used as testing sets and the average performance is the estimate. This is clearly a very expensive procedure. Instead, we can get a Monte-Carlo estimate if we repeat the *k*-fold crossvalidation using several random divisions of the data. Another variation is the stratified cross-validation (Kohavi, 1995), where the folds contain approximately the same proportions of each class as in the original set. Kohavi (1995), Weiss (1991), and Weiss and Indurkhya (1994) reported good results using stratified crossvalidation as compared with other resampling methods.

The leave-one-out method is a *n*-fold crossvalidation where *n* is the total number of examples. In this case all points are tested once. If we have $n$ training points we will have to do $n$ training using $n-1$ training points. Therefore this is also a computationally expensive process. There are, however, theoretical results that indicate that this estimator gives a good approximation for the generalization error (Joachims, 1999; Chapelle et al., 2002) even though it has a high variability (Joachims, 1999; Kohavi, 1995).

One interesting variation of the *k*-fold crossvalidation is proposed by Burman (1989). In his study, he concludes that in order to reduce the bias and variance certain correction terms should be added to the crossvalidation estimate.

### 2.4.3 SVM Specific Generalization Estimates

Several generalization performances estimates specific for SVMs have been proposed. Vapnik (1999) shows theoretically that a bound for the leave-one-out estimate for hard margin support vector machines is

$$T = \frac{N_{SV}}{n}$$

where $N_{SV}$ is the number of support vectors after training and $n$ is the total number of training points. Joachims (1999) extends this algorithm for the soft margin version of the algorithm. The error is defined in terms of the $\alpha$ and the $\xi$:

$$Err_{\xi\alpha}{}^{n} = \frac{d}{n} \text{ with } d = \left|\left\{ i : \left( \rho\alpha_i R^2{}_{\Delta} + \xi_i \right) \geq 1 \right\}\right|$$

where $\rho = 2$ and $R^2{}_{\Delta}$ is an upper bound on $K(\mathbf{x},\mathbf{x}) - K(\mathbf{x},\mathbf{x}')$ for all $\mathbf{x},\mathbf{x}'$. The proof for this bound can be found in Joachims (1999).

The advantage of this bound to leave-one-out is that we can calculate it with just one training of the SVM in contrast to doing $n$ trainings in leave-one-out or 10 trainings per repetition for repeated 10-fold crossvalidation. In the next chapter, we perform experiments to determine if this bound is predictive when changing parameters.

## 2.5 Statistical tests for comparing classifiers

While SVMs are deterministic algorithms, the methods for estimating their generalization performance, e.g. crossvalidation, depends on random partitions of the data. In addition, the genetic algorithms that we will use to optimize the SVM, uses random variables extensively. Therefore, it is important to apply statistical tests to find significant results.

A primary concern in this research is model selection. We want to find the best classifier for our particular dataset. This means that we need a valid way to compare two classifiers constructed using limited number of observations.

As noted by Dietterich (1988), there are many sources of random variations when comparing classifiers:

- The random selection of the test data used to evaluate each classifier. If the test set is not representative of the real population, we may be mislead to think that one classifier is better than the other when in reality they may be just as good.
- The random selection of the training set used to create the classifier. In SVMs deleting a support vector will affect the performance of the decision function. Support vectors are the points that compress the information about the relationship between input and output vectors.
- The randomness of the algorithm. This is not the case for SVMs but it is an important issue with neural networks.
- Random classification errors, outliers, etc. SVMs try to overcome these problems by allowing errors in the training. The number of allowable errors depends on the value of $C$.

In order to make statistically-sound decisions, we need to use statistical tests to select among models. In this thesis, two particular tests will be used extensively: *paired-t confidence interval* and *selecting the best of k systems*.

### 2.5.1 Paired-t Confidence Interval

In the paired-t test we have $X_{i1}, X_{i2}, \ldots, X_{in}$ independent and identically distributed (IID) observations from system $i$ with expected value $\mu_i = E(X_{ij})$. We want to find a confidence interval for $\mu_1 - \mu_2$. Lets define $Z_j = X_{1j} - X_{2j}$, for $j = 1, 2, \ldots, n$. Then

$$\overline{Z}(n) = \frac{\sum_{j=1}^{n} Z_j}{n}$$

and

$$\widehat{Var}[\overline{Z}(n)]^\wedge = \frac{\sum_{j=1}^{n}\left[Z_j - \overline{Z}_n\right]^2}{n(n-1)}$$

and we can form the approximate $100(1-\alpha)$ percent confident interval

$$\overline{Z}(n) \pm t_{n-1,i-\alpha/2}\sqrt{\widehat{Var}\left[\overline{Z}(n)\right]}^\wedge$$

If the $Z_j$'s are normally distributed, the confidence interval is exact. Otherwise, according to the Central Limit Theorem the coverage probability will be near $1-\alpha$ for large $n$ (Law and Kelton, 2000). Many statistical books consider $n \geq 30$ a large sample (Mendenhall and Sincich, 1995).

### 2.5.2 Selecting the Best of k Systems (Law and Kelton, 2000)

Let $\mu_{i_l}$ be the $l$th smallest of the $\mu_i$'s, so that $\mu_{i_1} \leq \mu_{i_2} \leq \ldots \leq \mu_{i_k}$. We would like to select the system with the smallest expected response (or larger). Let "CS" denote the correct selection. We would like to make the CS with a given probability. In addition, if $\mu_{i_2}$ and $\mu_{i_1}$ are very close we might not care if we select either one and, therefore, we want to avoid doing unnecessary replications.

We want $P(CS) \geq P^*$ provided that $\mu_{i_2} - \mu_{i_1} \geq d^*$ where $P^* > 1/k$ and the difference $d^* > 0$ are both specified by the analyst.

This process involves two stages of sampling. In the first-stage sampling, we make $n_0 \geq 2$ replications of each system and define

$$\overline{X}_i^{(1)}(n_0) = \frac{\sum_{j=1}^{n_0} X_{ij}}{n_0}$$

and

$$S_i^2(n_0) = \frac{\sum_{j=1}^{n_0} [X_{ij} - \overline{X}_i^{(1)}(n_0)]^2}{n_0 - 1}$$

for $i = 1, 2, \ldots, k$. Then we compute the total sample size $N_i$ needed for system $i$ as

$$N_i = \max\left\{ n_0 + 1, \left\lceil \frac{h_1^2 S_i^2(n_0)}{(d^*)^2} \right\rceil \right\}$$

where $\lceil \bullet \rceil$ is the smallest integer that is greater than or equal to the real number $\bullet$, and $h_1$ is a constant from a table and it is a function of $k$, $P^*$, and $n_0$. Then we make $N_i - n_0$ more replications of system $i$ and obtain the second-stage sample means

$$\overline{X}_i^{(2)}(N_i - n_0) = \frac{\sum_{j=n_{0+1}}^{N_i} X_{ij}}{N_i - n_0}.$$

Then define the weights (see p. 568 and p. 575 of Law and Kelton, 2000)

$$W_{i1} = \frac{n_0}{N_i}\left[ 1 + \sqrt{1 - \frac{N_i}{n_0}\left( 1 - \frac{(N_i - n_0)(d^*)^2}{h_i^2 S_i^2(n_0)} \right)} \right]$$

and $W_{i2} = 1 - W_{i1}$, for $i = 1, 2, \ldots, k$. Finally, the weighted sample means are

$$\widetilde{X}_i(N_i) = W_{i1}\overline{X}_i^{(1)}(n_0) + W_{i2}\overline{X}_i^{(2)}(N_i - n_0).$$

The system with the smallest $\widetilde{X}_i(N_i)$ is the CS with a probability of at least $P^*$.

## 2.6 Summary

SVM combines several fields like optimization theory from Operations Research, kernels from integral operator theory, generalization bounds from statistical learning theory, and the learning methodology from machine learning. It presents several theoretical and practical advantages over other learning techniques like neural networks. First, instead of minimizing the empirical risk as neural network do, SMVs follow the structural risk minimization principle to minimize the bound on the generalization ability of the algorithm which implies that they are particularly suited for problems with limited data. Second, we can directly control the tradeoff between the approximation to the training set and the complexity of the approximating function. Third, in other to train a neural network we usually need to solve a nonlinear non-convex optimization problem which presents the danger of getting trapped in a local solution which implies a loss in performance. To train a SVM we need to solve a convex quadratic programming problem for which efficient solution exists and which is usually sparse. Fourth, there are far less parameters to set with SVM than with neural networks.

# CHAPTER 3. LITERATURE SURVEY

This chapter presents a literature survey of previous work regarding support vector machines and model selection and of the uses of genetic algorithms to improve support vector machines.

## 3.1 Support vector machines and model selection

As explained in the previous chapter, support vector machines have several parameters that affect their performance and that need to be selected in advance. These parameters include the penalty value $C$, the kernel type, and the kernel specific parameters. While for some kernels, like the Gaussian RBF kernel, there is only one parameter to set ($\gamma$), other more complicated kernels may need an increasing number of parameters. The usual way to find good values for these parameters is to train different support vector machines –each one with a different combination of parameter values– and compare their performance on a test set or by using other generalization estimates like leave one out or crossvalidation. Nevertheless, an exhaustive search of the parameter space is time consuming and ineffective especially for more complicated kernels. For this reason several researchers have proposed methods to deal with this problem.

Cristianini and Shawe-Taylor et al. (1999) shows that the margin of a support vector machine with Gaussian Radial Basis Function kernel is a smooth function of the kernel parameter $\sigma$ $(1/\gamma)$. This implies that the bound on the generalization error is also smooth in $\sigma$. Using this result they propose a kernel selection procedure that makes use of the Kernel-Adatron training algorithm (Frieβ et al, 1998):

1. Initialize $\sigma$ to a very small value
2. Train the SV with the Kernel-Adatron algorithm
3. If the margin is maximized, then
    a. Observe the validation error
    b. Increase the kernel parameter: $\sigma \leftarrow \sigma + \delta\sigma$

   else go to step 2.
4. Stop when a predetermined value of $\sigma$ is reached.

The previous procedure will find the maximal margin hyperplane for a small value of $\sigma$ and then the hyperplane is kept at the maximal margin by continually adjusting the $\alpha$ values while the kernel-parameter is increased. This procedure is quite effective if we only have one parameter to set. Furthermore, the kernel type is selected arbitrarily. In Cristianini and Shawe-Taylor et al. (1999) the experiments were done using a Gaussian kernel. While the authors reported a speedup in the convergence when using this technique, it is not clear how to find the penalty value $C$ or other parameters of the kernel using this procedure

Chapelle et al. (2002) propose a method of using gradient descent to set the kernel parameters $\boldsymbol{\theta}$ (a vector containing the penalty value $C$ and the kernel specific parameters) of a support vector machine:

1. Initialize $\boldsymbol{\theta}$ to some value.
2. Using the standard SVM algorithm, find the maximum of the quadratic form $W$:

$$\alpha^0(\boldsymbol{\theta}) = \arg\max W(\boldsymbol{\alpha}, \boldsymbol{\theta}) .$$

3. Update the parameters $\boldsymbol{\theta}$ such that the generalization error estimate is minimized.

   This is typically achieved by a gradient step.

4. Go to step 2 or stop when the minimum of the generalization error is reached.

Using this method requires finding how the generalization error varies with the parameter vector $\boldsymbol{\theta}$, i.e., the kernel has to be differentiable with respect to the vector of parameters in order to use the gradient descent technique. Furthermore, it is not clear if the generalization error is a convex function of the parameter vector. If it is not, there is no guarantee that the solution obtained is a global solution (see for example Keerthi, S. and Lin C.-J., 2003; Xuefeng and Fang, 2002). Also, the kernel is not selected automatically but it is assigned arbitrarily.

Shao and Cherkassky (1999) propose a completely different approach to dealing with finding the optimal parameters. They describe an extension of the SVM method based on using several kernels at the same time with different scales called the Multi-Resolution Support Vector Machine (M-SVM). They develop this variation of SVMs as an extension to wavelet-based multi-resolution analysis. The training of M-SVM will find not only the $\alpha$ values but also the kernel-specific parameters. They experiment with two Radial Basis Function kernels in several simulated regression problem. The results show that this combination of kernels is able to reduce the prediction error of the SVM. Nevertheless, this method requires adding *m*-fold free variables for *m* kernels used in the optimization formulation which increases the computational complexity. Even more important, now we need to arbitrarily set one penalty value $C$ for each kernel.

Another approach for improving regression performance by using a mixture of kernels was proposed by Smits and Jordaan (2002). They use a convex combination of a polynomial and a RBF kernel:

$$K_{mix} = \rho K_{poly} + (1-\rho)K_{rbf}$$

In this approach one extra parameter—that requires setting—is added. Also there is a need to find optimal values for $C$ and $\varepsilon$. The technique is tested on a real-life industrial data set and presents improved interpolation and extrapolation results as compared with using individual kernels.

Ali and Smith (2003) propose a method based on Bayesian inference on the training set to find the optimal degree for a polynomial kernel. They find that for datasets that are strongly non-normal the performance of the model found with their method is better than that obtained by using arbitrarily assigned degrees or even using a Gaussian RBF kernel with arbitrarily assigned parameters. With this approach, we can only find the optimal degree of a polynomial kernel. It will not allow us to find parameters of other important kernels like, for instance, a RBF kernel. Also, it is not clear what value of $C$ is used for the experiments and how this value is selected.

## 3.2 Genetic algorithms and learning algorithms

For many years now, genetic algorithms have been used together with neural networks. There have been different ways to integrate genetic algorithms and neural networks: they have been use to find the weights (training), to determine the architecture, for input feature selection, weight initialization, among other uses. A thorough review can be found in Yao (1999). Recently, researchers have been looking into the combination of support vector machines with genetic algorithms.

Few researchers have tried integrating SVMs with genetic algorithms. There are basically two types of integrations of SVM and GA. The most common one consists on using the GA to select a subset of the possible variables reducing the dimensionality of the input vector for the training set of the SVM or selecting a subset of the input vectors that are more likely to be support vectors (Sepúlveda-Sanchis et al., 2002; Zhang et al., 2001; Xiangrong and Fang, 2002; Chen, 2003). A second type of integration found in the literature is using a GA for finding the optimal parameters for the SVM (Quang et al, 2002; Xuefeng and Fang, 2002).

In Sepúlveda-Sanchis et al. (2002), a genetic algorithm is designed to find a subset of variables with the highest influence on predicting the risk of acute unstable angina. The fitness function of this GA is based on several information criteria like Mallow's $C_p$ criterion, Akaike's Information Criteria (AIC), and the Maximum Description Length (MDL) (see references in Sepúlveda-Sanchis et al., 2002). The genetic algorithm was able to find a subset of 5 variables out of the 75 original variables. Those variables were then used to train a support vector machine to predict the mortality of patients with unstable angina.

A similar approach is proposed in Xiangrong and Fang (2002). In this case the fitness function is based on a previous work by Zhang et al. (2001), where a method called the Center Distance Ratio Method is introduced. This method allows them to find those input vectors that are closest to the margin and that, therefore, are more likely to be support vectors. While they so not present any experimental results, it seems conceivable that this particular combination of GAs with SVM could produce good results.

GAs have also been applied to a slightly different problem: reducing the dimensionality of the data. This problem is of particular relevance for bioinformatics, e.g., gene selection where the data is usually limited and the dimensionality is high.

Chen (2003) uses a combination of GAs with a resampling technique known as Bootstrap to select those genes that are needed to be able to discriminate between cancer and normal cells. The algorithm is tested with two datasets: a colon cancer dataset and a leukemia dataset. In both cases, the dataset is split in a training and a testing set. Each individual of the population of GAs became a possible subset of 5 genes. The fitness function is based on training a SVM with the training set using only the subset of genes defined by the individual and then measuring the misclassification on the testing set. Using this procedure, it is found that several subsets could classify all observations from the testing set correctly.

Xuefeng and Fang (2002) present a variation to the method proposed by Chapelle et al. (2002), where the gradient descent is replaced by a genetic algorithm. The advantage of this approach lies in that now the kernel does not need to be differentiable and that the solution have a better chance of being a global solution.

Finally, Quang et al. (2002) used GAs to find the different parameters of a SVM with a mixture of kernels. In addition, they use two penalty values, one for each class. The main difference in the approach that we use lies in that the fitness function used by Quang et al. (2002) is based on estimating the generalization error of the SVM with the $\xi\alpha$ -estimator proposed by Joachims (1999) because of the great efficiency of the method. However, this estimate was not compared to other well tested methods for estimating the generalization error like crossvalidation or repeated holdout testing. In fact, research by Duan et al. (2003) have found that the $\xi\alpha$ gives estimates close to the test error for small values of $C$. However, when the $C$ value increases, this estimate differs a lot from the test error. This will tend to mislead the search of the GA.

## 3.3 Summary

In summary, from this literature survey it is clear that several researchers have been devising ways to automatically find the best parameters for SVMs. In this thesis, we will propose another approach that makes use of ten-fold crossvalidation, genetic algorithms, and support vector machines with a mixture of kernel for pattern recognition. The experiments are done using a dataset that represents model of individuals for electronic commerce applications. This particular combination of techniques has not been integrated and applied to e-commerce datasets before as far as we know.

# CHAPTER 4. IMPLEMENTATION AND RESULTS

This chapter presents our genetic algorithm implementation to automatically find the optimal parameters of a support vector machine with a mixture of Gaussian and polynomial kernel for pattern recognition. We start by describing the dataset used for all experiments. Immediately after we describe the different parts of the genetic algorithm and present the Unified Modeling Language (UML) diagrams of the main classes. Finally, we will run some initial experiments to decide the operators for the genetic algorithm.

## 4.1 Dataset

All experiments use data from the study conducted by Ryan (1999) that contains information on 125 subjects. A web site is used for this experiment, where 648 images are shown sequentially to each subject (all of the images are saved using a JPG quality of 5). The response required from the individuals is their preference for each image (1: Yes, 0: No).

The images varied on seven attributes (features) with some specific levels:

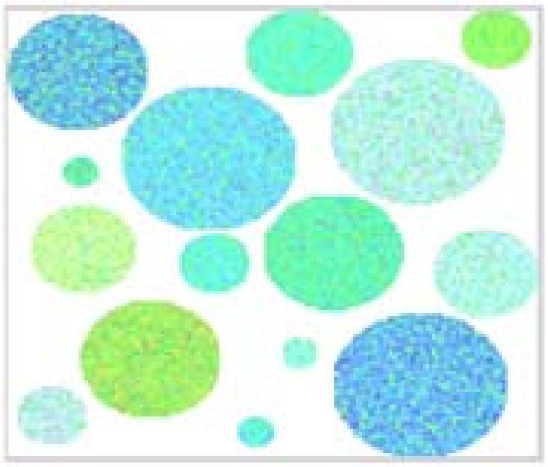
- Density – Describes the number of circles in an image (3 levels).
- Color Family – Describes the hue of the circles (3 levels).
- Pointalization – Describes the size of the points that make the individual circles (3 levels).
- Saturation – Describes the strength of the color within the circles (3 levels).
- Brightness – Describes the amount of light in the circles themselves (4 levels).

- Blur – Describes the crispness of the circles (2 levels).
- Background – Describes the background color of the image (3 levels).

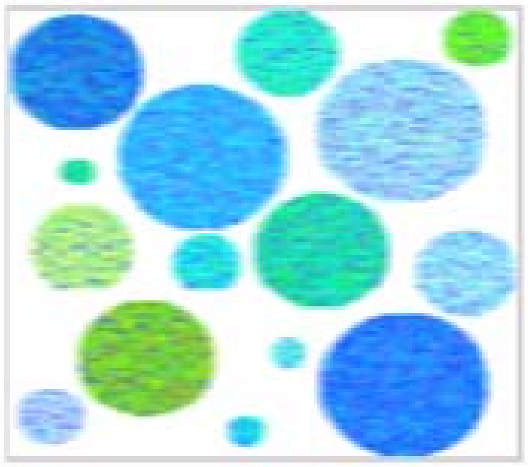
**Table 2. Features used to generate the 624 images (Rabelo 2000).**

| | **Attribute** | **Level 1** | **Level 2** | **Level 3** | **Level 4** |
|---|---|---|---|---|---|
| **1** | **Density** | X3 | X2 | X1 | -- |
| **2** | **Cold vs. Warm** | Cold: blue, green | purples | Warm: red, orange | -- |
| **3** | **Pointalized** | 5 | 15 | 50 | -- |
| **4** | **Saturation** | 50 | 0 | -- | -- |
| **5** | **Light/Dark** | 50 | -- | -- | -25 |
| **6** | **Motion blur** | 0 | 10 | -- | -- |
| **7** | **BKG** | Black | Gray | White | -- |

For example, Figure 10 to Figure 12 show examples of some of the images.

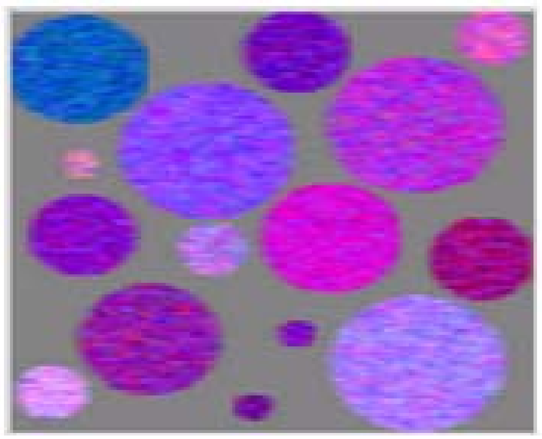


**Density**: Level 1 **Cold vs Warm**: Level 1
**Pointalized**: Level 1 **Saturation**: Level 1
**Light/Dark**: Level 1 **Motion blur**: Level 1
**BKG**: Level 3

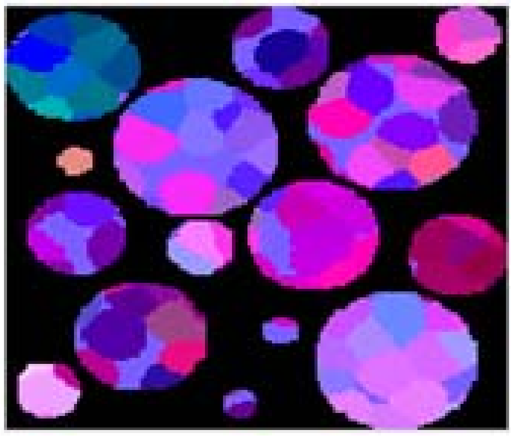


**Density**: Level 1 **Cold vs Warm**: Level 1
**Pointalized**: Level 1 **Saturation**: Level 1
**Light/Dark**: Level 2 **Motion blur**: Level 2
**BKG**: Level 3

**Figure 10. Images with features 1111113 and 1111223 respectively.**

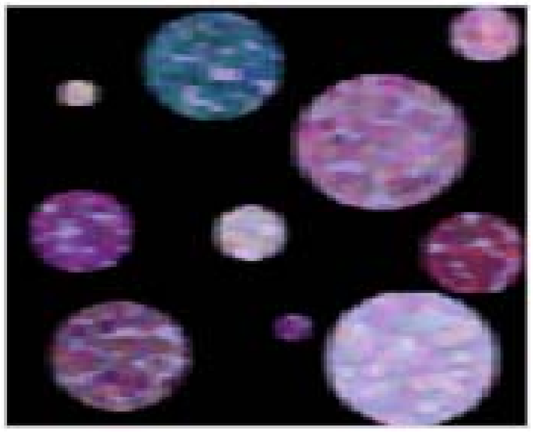


**Density**: Level 1 **Cold vs Warm**: Level 2
**Pointalized**: Level 1 **Saturation**: Level 1
**Light/Dark**: Level 3 **Motion blur**: Level 2
**BKG**: Level 2

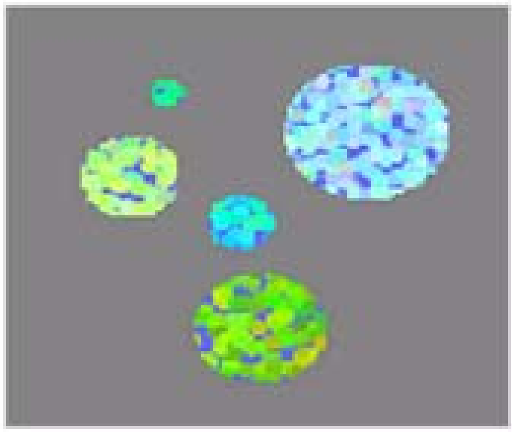


**Density**: Level 1 **Cold vs Warm**: Level 2
**Pointalized**: Level 3 **Saturation**: Level 1
**Light/Dark**: Level 3 **Motion blur**: Level 1
**BKG**: Level 1

**Figure 11. Images with features 1211323 and 1231311 respectively.**



**Density**: Level 2 **Cold vs Warm**: Level 2
**Pointalized**: Level 2 **Saturation**: Level 3
**Light/Dark**: Level 3 **Motion blur**: Level 2
**BKG**: Level 1



**Density**: Level 3 **Cold vs Warm**: Level 1
**Pointalized**: Level 2 **Saturation**: Level 1
**Light/Dark**: Level 2 **Motion blur**: Level 1
**BKG**: Level 2

**Figure 12.** Images with features 2223321 and 3121212 respectively.

The response of each individual is an independent dataset. Rabelo (2001) compares the performance of several learning algorithms using this dataset.

## 4.2 Implementation of the Genetic Algorithm

All programs are written in C++ and compile in Visual C++ .NET. The support vector training is based on a modified version of LIBSVM (Chang and Lin, 2001). An object-oriented methodology is followed in the creation of all programs. Interfaces provided the skeleton for most operators like the fitness function, selection, mutation, and crossover. As a result, the population class could use any fitness function implementing the corresponding interface. In the same way, any mutation or crossover operator implementing the corresponding interface could act on the population.

### 4.2.1 Representation

The LIBSVM program is modified to include a mixture of Gaussian and polynomial kernel:

$$p \cdot e^{-\gamma \|\mathbf{u}-\mathbf{v}\|^2} + (1-p) \cdot \left( \langle \mathbf{u} \cdot \mathbf{v} \rangle + r \right)^d$$

Keerthi and Lin (2003) find that when a Gaussian RBF kernel is used for model selection, there is no need to consider the linear kernel since it behaves as a linear kernel for certain values of the parameters $C$ and $\gamma$.

Each individual is represented as a binary string that encoded five variables (see Figure 13):

- The first 16 bits represents the cost or penalty value, C. It is scaled from 0.01 to 1000.
- The next 16 bits represents the width of the Gaussian kernel, $\gamma$, scaled from 0.0001 to 1000.
- The next 2 bits represents 4 possible values for the degree $d$ : from 2 to 5
- The next 16 bits represents the $\rho$ parameter which controls the percentage of polynomial and Gaussian kernel. It was scaled from 0 to 1.
- Finally, the last parameter is the $r$ value, which determines whether we use a complete polynomial or not.

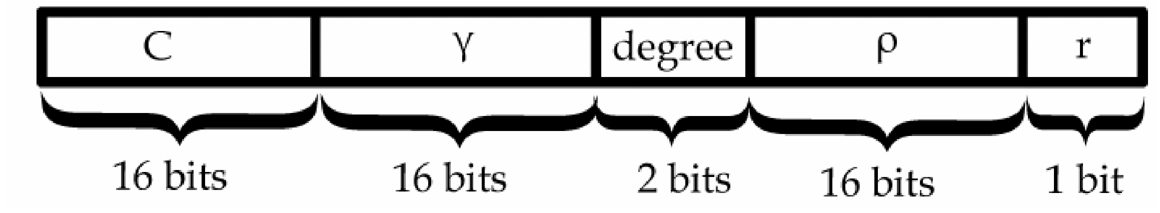


**Figure 13. Representation of parameters.**

The binary code $s_i$ that represents each variable is transformed to an integer according to the expression

$$m = \sum_{i=0}^{N-1} s_i 2^i$$

where $N$ is the number of bits. This integer value is then scaled to a real number in the interval $[a,b]$ according to

$$x = a + m\frac{b-a}{2^N - 1}$$

The precision depends on the range and the number of bits:

$$\varepsilon = \frac{b-a}{2^N - 1}.$$

The population of individuals is implemented by a class called BinPopulation.h (see Figure 14). Individuals are stored in a vector of strings while their corresponding fitness is stored in a vector of doubles. The population keeps a pointer to the fitness function interface, which is needed to calculate the fitness of each individual. Since all fitness functions implement this interface, the population class can point to any fitness function. The crossover operation requires selecting the parents. Therefore the population class also keeps a pointer to the selection interface which allows it to use any selection technique that implements this interface.

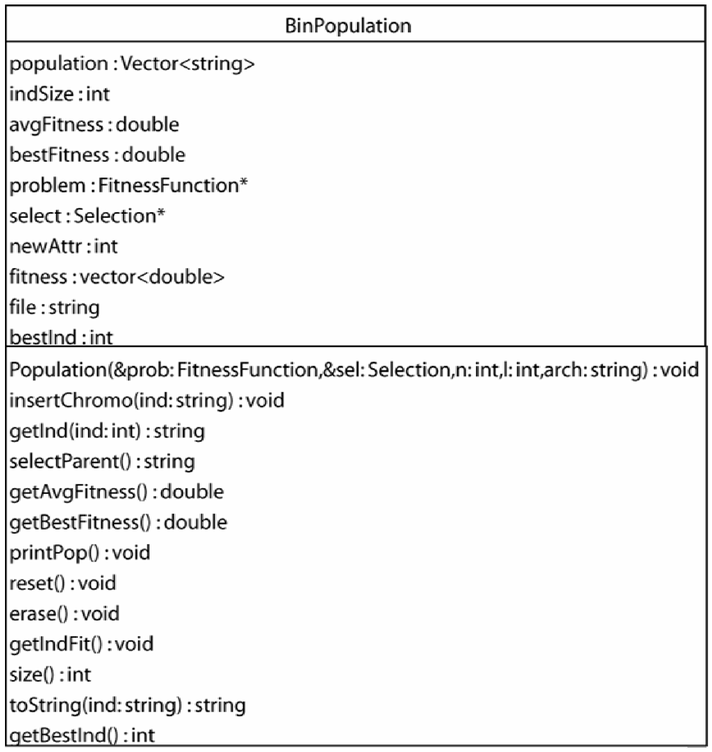


**Figure 14. UML diagram of the Population class.**

### 4.2.2 Fitness Function

The objective function is probably the most important part of the genetic algorithms since it is problem-dependent.

We need a way to measure the performance or quality of the possible solutions. As indicated previously, there are several methods that try to estimate the generalization error of a classifier. Contrary to other applications of genetic algorithms, the objective function in this problem is a random variable with associated variance and it is computationally expensive since it involves training a learning algorithm. In order to decide which method to use, several experiments are run to find the estimator with the lowest variance.

A class, Xvali, split the data according to the different techniques (see Figure 15).

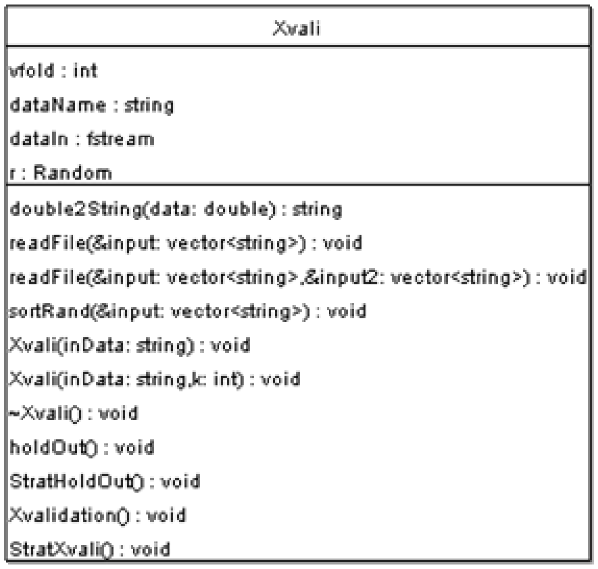

Xvali
vfold : int
dataName : string
dataIn : fstream
r : Random
double2String(data: double) : string
readFile(&input: vector<string>) : void
readFile(&input: vector<string>,&input2: vector<string>) : void
sortRand(&input: vector<string>) : void
Xvali(inData: string) : void
Xvali(inData: string,k: int) : void
~Xvali() : void
holdOut() : void
StratHoldOut() : void
Xvalidation() : void
StratXvali() : void

**Figure 15. UML diagram of the Crossvalidation class used to test the different generalization estimates.**

LIBSVM with $C = 10$, Gaussian RBF kernel, and $\gamma = 0.1$ was used as a classifier. A sample of 1000 observations where taking from each method:

- Hold out
- Stratified hold out
- 10 fold crossvalidation
- 10 fold stratified crossvalidation

- 10 fold modified crossvalidation
- 10 fold modified stratified crossvalidation

The results are summarized in Table 3.

**Table 3. Mean and standard deviation of the several estimates of the generalization performance obtained from a sample of 1000 observations.**

| Technique | Mean (%) | Standard Deviation (%) |
|---|---|---|
| **10 fold Stratified Modified Crossvalidation** | 86.830 | 0.461 |
| **Modified Crossvalidation** | 86.791 | 0.463 |
| **Stratified Crossvalidation** | 86.681 | 0.486 |
| **Crossvalidation** | 86.617 | 0.496 |
| **5 fold Stratified Modified Crossvalidation** | 86.847 | 0.540 |
| **5 fold Stratified Crossvalidation** | 86.567 | 0.609 |
| **5 fold Crossvalidation** | 86.540 | 0.629 |
| **Stratified hold out** | 86.215 | 1.809 |
| **Hold out** | 86.241 | 1.977 |

The hold out technique had the highest standard deviation. Stratifying the method, i.e., keeping the same ratio between classes in the training and testing set, slightly reduced the standard deviation. All crossvalidation estimates had a significantly lower standard deviation than the hold out technique.

Since there is really no statistically significant difference in the standard deviation between the different crossvalidation techniques we use one of the most common: 10-fold crossvalidation.

The fitness function is a component of the class population that evaluates an individual. It returns the generalization estimate according to the 10-fold crossvalidation.

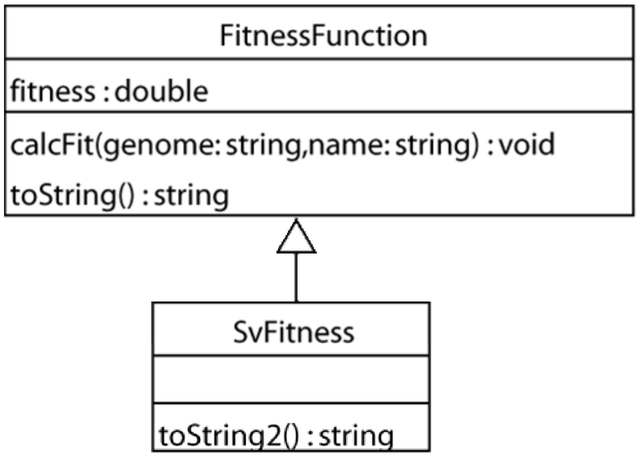


**Figure 16. UML diagram of the Fitness function class.**

### 4.2.2.1 <u>An efficient generalization estimate vs. 10-fold Crossvalidation</u>

As mentioned in the previous chapter, there have been some leave-one-out approximations that are specific to SVM and are very efficient. Because of this efficiency, if they would produce a good approximation of the generalization error, they would make an excellent fitness function. For this reason, we compare the estimate obtained using this method with the one obtained using 10-fold crossvalidation. The estimate for the $\xi\alpha$ technique is calculated using Joachims's SVMlight. The 10-fold crossvalidation is repeated 50 times and the average is calculated. The results for different values of $C$ is shown in Table 4 for dataset ind2 and a Gaussian RBF kernel with $\gamma = 0.1$.

**Table 4. Comparison of two generalization estimates.**

| C | $\xi\alpha$ Estimate (% f errors) | 10-fold crossvalidation (% of errors) |
|---|---|---|
| **0.01** | 25.62 | 25.613 |
| **0.1** | 24.07 | 23.19122 |
| **0.5** | 41.36 | 13.77713 |
| **1** | 38.27 | 12.92339 |
| **2** | 36.11 | 12.98201 |
| **5** | 33.49 | 12.89836 |
| **10** | 32.72 | 13.44506 |
| **25** | 30.4 | 14.42876 |
| **50** | 30.71 | 15.32877 |
| **100** | 30.86 | 16.214 |
| **500** | 30.4 | 17.35171 |
| **1000** | 28.7 | 18.61872 |

From these results it seems that for small values of $C$ the estimate is very close to the crossvalidation estimate. However, as the $C$ increases the estimate starts to deviate. According to the $\xi\alpha$ estimate the best model is the one with $C = 0.1$ while according to 10-fold crossvalidation estimate the best model has $C = 5$.

Clearly, the $\xi\alpha$ requires more research in order to understand these deviations.

### 4.2.3 Crossover

Several crossover operators are tested: one point, two point, uniform, and multiparent diagonal. Each class implemented the Crossover interface (see Figure 17).

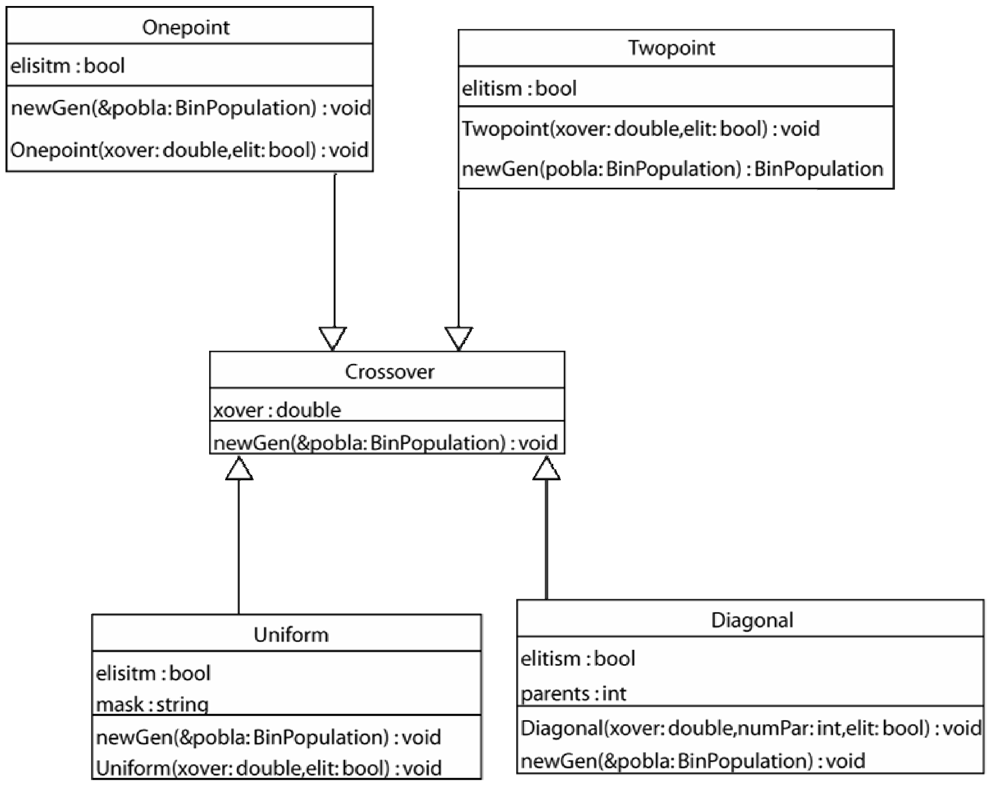

Onepoint
elisitm : bool
newGen(&pobla: BinPopulation) : void
Onepoint(xover: double,elit: bool) : void
Twopoint
elitism : bool
Twopoint(xover: double,elit: bool) : void
newGen(pobla: BinPopulation) : BinPopulation
Crossover
xover : double
newGen(&pobla: BinPopulation) : void
Uniform
elisitm : bool
mask : string
newGen(&pobla: BinPopulation) : void
Uniform(xover: double,elit: bool) : void
Diagonal
elitism : bool
parents : int
Diagonal(xover: double,numPar: int,elit: bool) : void
newGen(&pobla: BinPopulation) : void

**Figure 17. UML diagrams of the Crossover operators.**

### 4.2.4 Mutation

Two mutation operators implement the Mutation interface (see Figure 18). SimpleMut is a simple mutation with fixed mutation probability. DynaMut is a mutation with a dynamic rate of mutation that depends on the generation according to the equation:

$$p_t = \left( 2 + \frac{n-2}{T-1} \cdot t \right)^{-1}$$

In addition, SimpleMut serves as the base for other techniques for varying the mutation rate: a self-adaptation method and a feedback mechanism based on the genetic diversity.

The self adaptation method consists on adding 16 bits to each individual in order to obtain a probability $p$. From this value the mutation rate is obtained according to the following equation (Bäck and Schütz ,1996):

$$p' = \left( 1 + \frac{1-p}{p} \cdot e^{-\gamma \cdot N(0,1)} \right)^{-1}$$

where $\gamma$ is the rate that controls the adaptation speed and $N(0,1)$ is a random normal number with mean 0 and standard deviation 1. The normal random variable is generated according to the Box and Muller method (see, for example, Law and Kelton 2000 p 465)

The feedback mechanism was based on calculating the genetic diversity of the population $AvgFitness / BestFitness$. If the genetic diversity falls below a particular level the mutation rate is increased and the crossover rate is reduced. The contrary happens if the genetic diversity becomes bigger than a given value. The problem is to find those critical values. Clearly, it will depend on the problem.

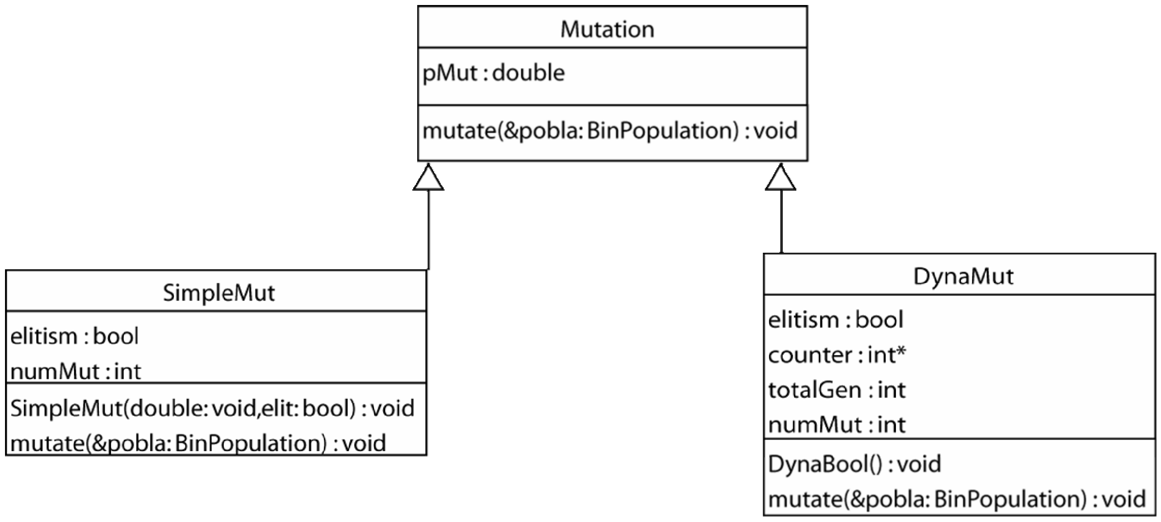

Mutation
pMut : double
mutate(&pobla: BinPopulation) : void
SimpleMut
elitism : bool
numMut : int
SimpleMut(double: void,elit: bool) : void
mutate(&pobla: BinPopulation) : void
DynaMut
elitism : bool
counter : int*
totalGen : int
numMut : int
DynaBool() : void
mutate(&pobla: BinPopulation) : void

**Figure 18. UML diagrams of the Mutation operators.**

### 4.2.5 Selection

The deterministic $k$ Tournament selection is implemented as seen in Figure 19.

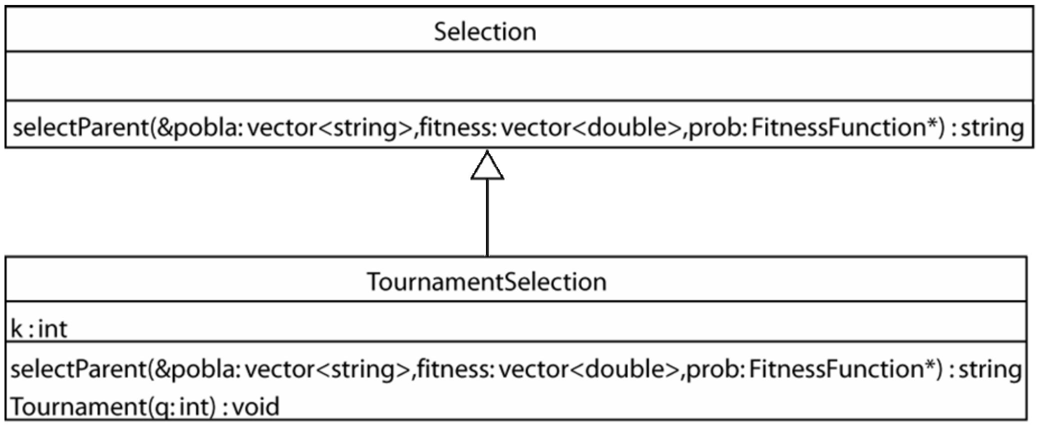

Selection
selectParent(&pobla: vector<string>,fitness: vector<double>,prob: FitnessFunction*) : string
TournamentSelection
k : int
selectParent(&pobla: vector<string>,fitness: vector<double>,prob: FitnessFunction*) : string
Tournament(q: int) : void

**Figure 19. UML diagram of the Deterministic Tournament selection class.**

## 4.3 Comparison of Variations of Genetic Algorithms

To select the operators with the best performance (e.g., faster convergence of the genetic algorithm) from the different possibilities, we repeat the algorithm 30 times with different random initial solution. With each replication, we obtain an independent estimate for the best generalization ability at each generation.

At the start of each replication, the dataset is randomly split in the ten subsets required by the 10 fold crossvalidation. Using the same split during the whole run allows us to study the effect of the different variations without being affected by randomness, i.e., one particular model will always have the same performance throughout the run of the genetic algorithm. At the same time, since we are doing 30 replications –each with a different random split— we can get a good idea of the average performance as a function of the generation for each of the variations of the genetic algorithm. Figure 20 summarizes this process in an activity diagram.

Table 5 lists the different parameters of the genetic algorithm.

**Table 5. Parameters of the genetic algorithm used for testing the different variations.**

| Paremeter | Value |
|---|---|
| **Population** | 10 |
| **Generations** | 20 |
| **Prob. of crossover** | 0.95 |
| **Prob. of mutation** | 0.05 |
| **Fitness function** | 10 fold crossvalidation |

| | |
|---|---|
| **Selection** | 2-Tournament selection |
| **Crossover types** | One point, two point, uniform, diagonal with 4 parents |
| **Mutation type** | Fixed rate, dynamic rate, self adaptive rate, feedback |
| **Other** | Elisitm, no elitism |

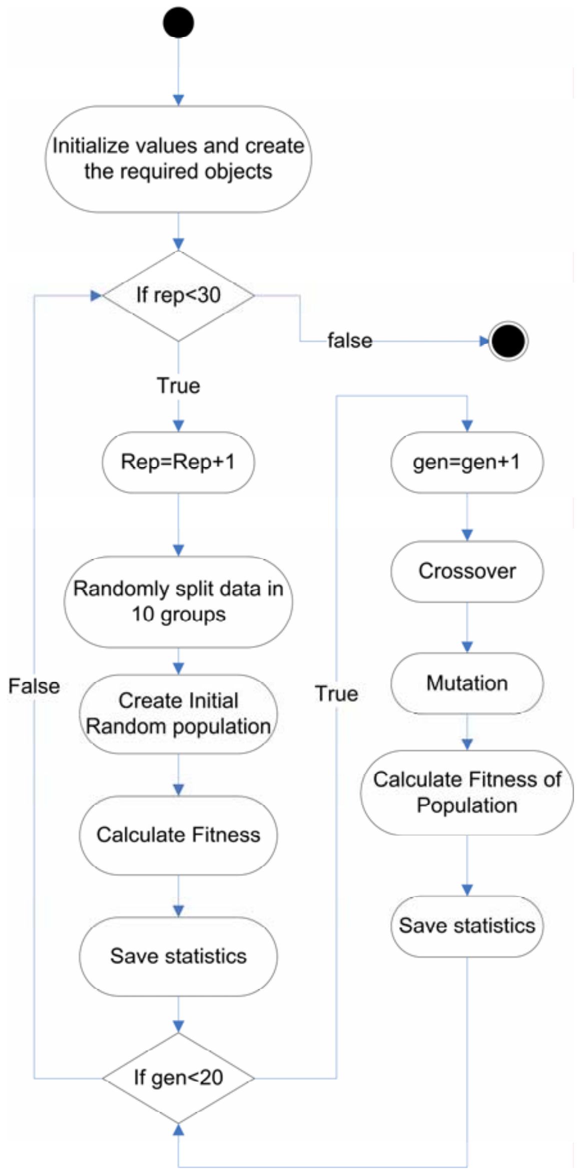

Initialize values and create the required objects
If rep<30
false
True
Rep=Rep+1
Randomly split data in 10 groups
False
Create Initial Random population
Calculate Fitness
Save statistics
If gen<20
True
gen=gen+1
Crossover
Mutation
Calculate Fitness of Population
Save statistics

**Figure 20. Overview of the genetic algorithm.**

### 4.3.1 Experiments

To study the behavior of the genetic algorithms as a function of the different parameters, we repeat the experiment 30 times and calculated the average for each generation. A subset of 215 points is used for the experiments. This subset was obtained in a stratified manner (the proportion of individuals of class 1 to class -1 was kept equal to the original dataset) from individual number 2. The reduction of the number of points is done to reduce the processing time.

In most cases, we are interested in comparing the performance measures at the 20th generation the genetic algorithms using different parameters. This comparison is made using several statistical tests like 2 sample t test and best of k systems (Law and Kelton, 2000).

#### 4.3.1.1 Effect of the elitist strategy

Figure 21 shows the effect of elitism when the genetic algorithms uses a one point crossover with crossover rate of 0.95 and simple mutation with mutation rate of 0.05.

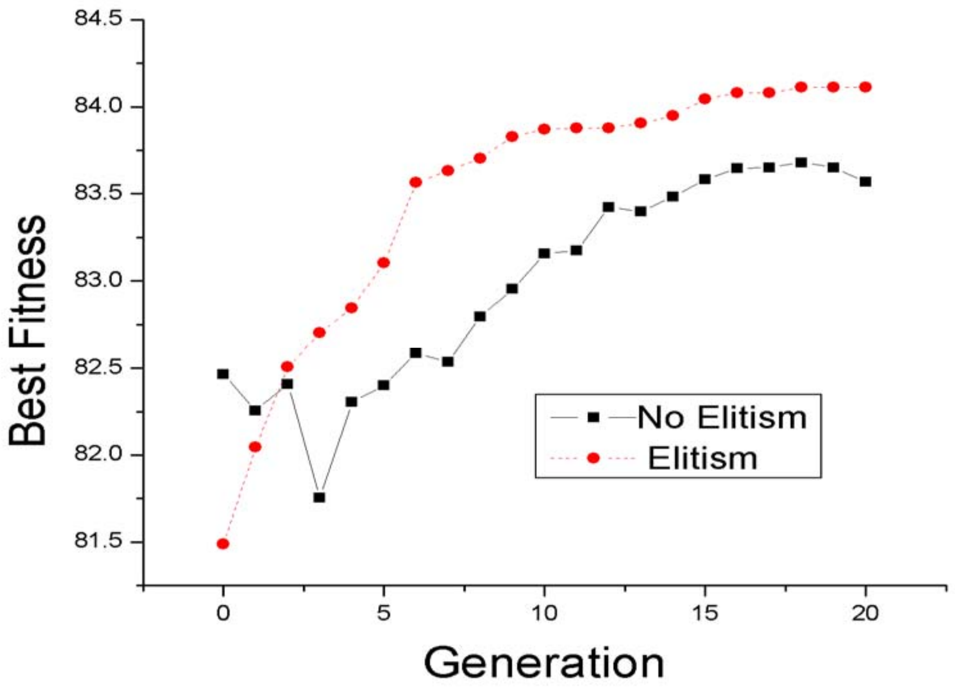


**Figure 21. Effect of elitism in the best fitness per generation.**

We use simple elitism, i.e., the best parent is passed unmodified to the next generation. As it is shown in Figure 21, by not using elitism there is a risk of losing good individuals, which may also increase the number of generations needed to find a good solution.

A two sample t-test shows that, at generation 20, the average best fitness of the elitism GA is significantly higher at the 0.1 level with a p-value of 0.054 and a lower limit for the 90% confidence interval of 0.542557.

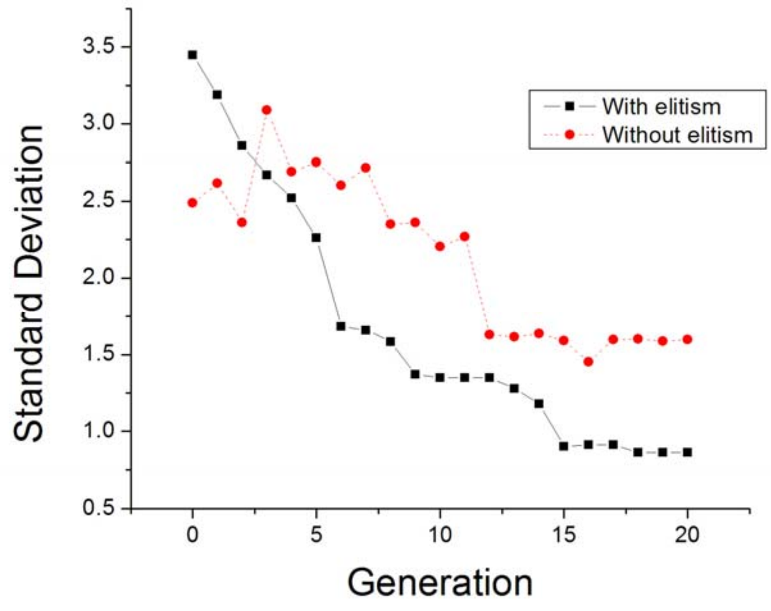


**Figure 22. Standard deviation of the average best fitness for elitism vs. not elitism.**

Figure 22 shows the standard deviation of the two genetic algorithms as a function of the generation. Another advantage of using the elitist strategy is that as the generation increases the standard deviation decreases. The standard deviation of the genetic algorithm with elitist strategy is significantly lower at the 20th generation at the 0.1 level in the F test for two variances and the Bonferroni confidence interval (see Figure 23).

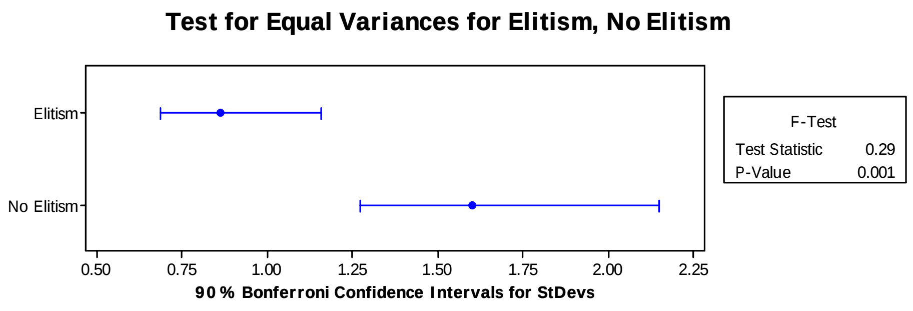


**Figure 23. Test for Equal Variances for elitism vs. not elitism.**

#### 4.3.1.2 <u>Effect of Crossover type</u>

Four crossover types are tested: one point, two points, uniform, and a 4-parents diagonal as defined in Section 3.1.1. The comparison is shown in Figure 24 and Figure 25.

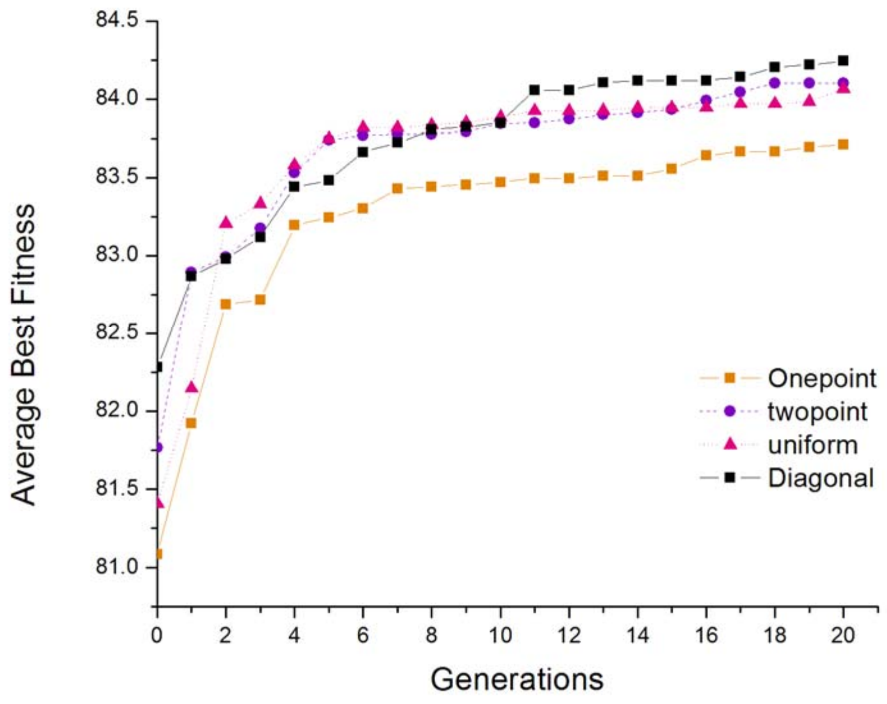


**Figure 24. Effect of the different crossover type on the fitness function.**

**Table 6. Average and Variance in the 20th generation as a function of the crossover type.**

| Crossover Type | Average | Variance |
|---|---|---|

|  |  |  |
|---|---|---|
| **Diagonal** | 84.24481 | 1.015474 |
| **Twopoint** | 84.10167 | 0.456379 |
| **Uniform** | 84.06692 | 1.105777 |
| **Onepoint** | 83.71069 | 1.593839 |

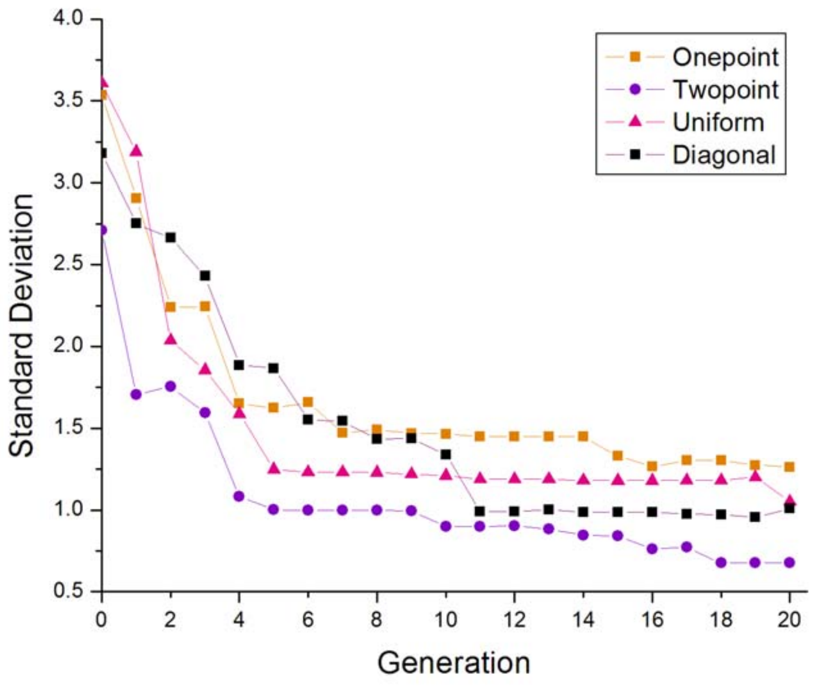


**Figure 25. Effect of the different crossover type on the standard deviation.**

The 4-parent diagonal crossover has the highest fitness function at the 20th generation; however, it has a higher standard deviation than the two-points crossover (see Figure 25 and Table 6). In order to make a decision we use a technique found in Law and Kelton (2000) for finding the best of *k* systems (see also previous chapter). With this methodology, we select the diagonal crossover as the best for this particular problem. Also, with 90% confidence, the expected performance of the diagonal crossover will be no less than 84.24481-0.5 (in other words, $P^* = 90\%$, $d^* = 0.5$). The assumptions required by this test are normality and independence. Normality is tested using Anderson-Darling test as implemented by Minitab (see Table 7). Each genetic algorithm is run using an independent stream of random numbers.

**Table 7. Normality test (Anderson-Darling).**

| Crossover Type | p-value | Decision |
|---|---|---|
| **Onepoint** | 0.22 | Normal at 0.05 level |
| **Twopoint** | 0.249 | Normal at 0.05 level |
| **Uniform** | 0.391 | Normal at 0.05 level |
| **Diagonal** | 0.176 | Normal at 0.05 level |

#### 4.3.1.3 Effect of varying mutation rates

Four ways to set the mutation rate are tested: fixed mutation rate, dynamically adapted, self adaptation, and feedback. The other parameters are kept constant: diagonal crossover with 4 parents, crossover rate of 0.95 and tournament selection. For the fixed mutation rate, the probability of mutation is set to 0.05. The behavior of the average best fitness as a function of the generation is shown in Figure 26. Figure 27 shows the behavior of the standard deviation.

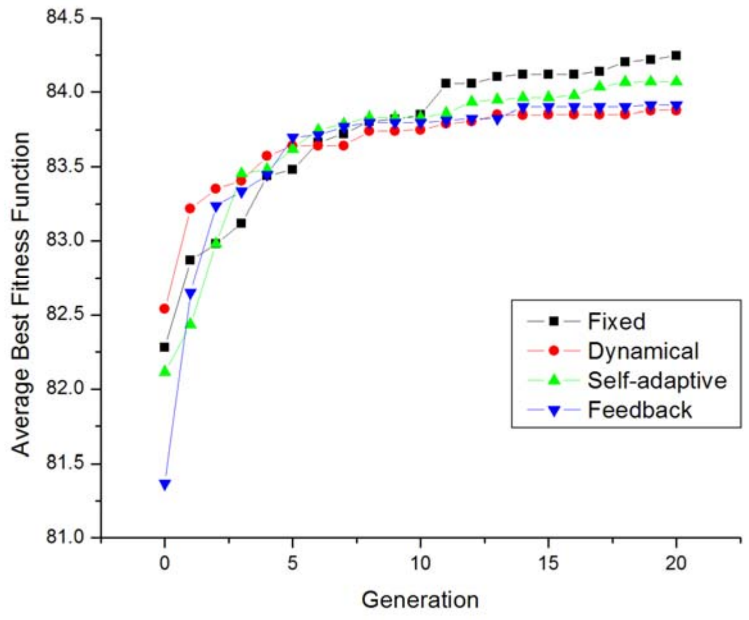


**Figure 26. Effect of mutation rate adaptation.**

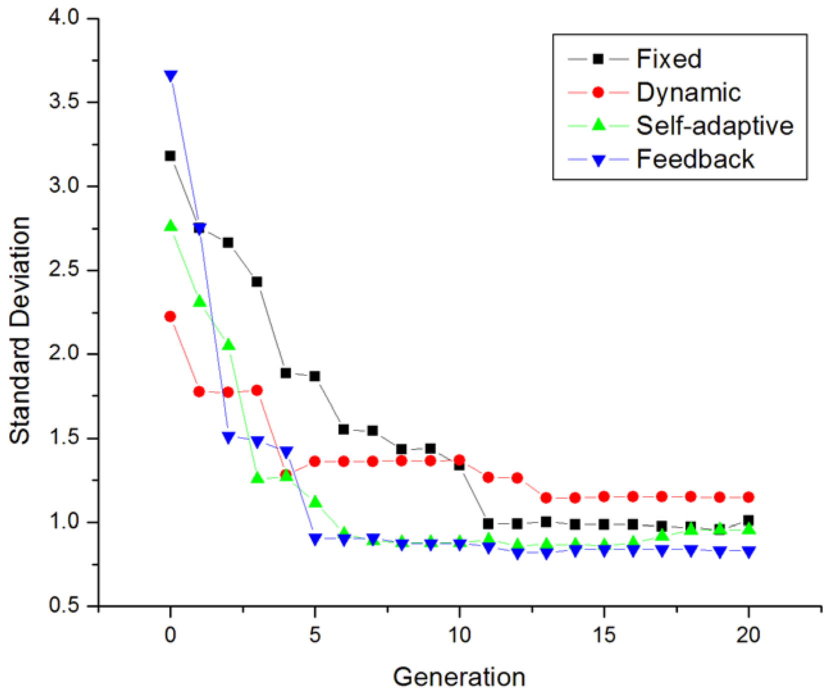


**Figure 27. Standard deviation of the best fitness per generation.**

Again, to select between the different techniques we use the select the best of *k* system methodology to choose among the different techniques with the best performance at the 20th generation. The selected method is the fixed mutation rate.

The assumption of normality is tested with Anderson Darling test (see Table 8).

**Table 8. Normality test for the 30 replications at the 20th generation.**

| **Crossover Type** | **p-value** | **Decision** |
|---|---|---|
| **Fixed** | 0.176 | Normal at 0.05 level |
| **Dynamic** | 0.291 | Normal at 0.05 level |
| **Uniform** | 0.255 | Normal at 0.05 level |
| **Diagonal** | 0.379 | Normal at 0.05 level |

## 4.4 Genetic Algorithm and Support Vector Machines

From the previous experiments, we select the parameters shown in Table 9.

**Table 9. Parameters in the final genetic algorithm.**

| **Parameters** | **Value** |
|---|---|
| **Population** | 10 |

| | |
|---|---|
| **Generations** | 20 |
| **Prob. of crossover** | 0.95 |
| **Prob. of mutation** | 0.05 |
| **Fitness function** | 10-fold crossvalidation |
| **Selection** | 2-Tournament selection |
| **Crossover types** | Diagonal with 4 parents |
| **Mutation type** | Fixed rate |
| **Others** | Elitist strategy |

The activity diagram of the final genetic algorithm is shown in Figure 28. The most important difference between this final model and the one used in the previous section is related to the random split of the data. Instead of using only one split of the data, every time the fitness of the population is calculated, we use a different random split (see Figure 29). As a result, all individuals at a particular generation are measured under the same conditions. Using only one random split throughout the whole run of the GA carries the danger that the generalization error estimate for one particular model may be higher than for other models because of the particular random selection and not because it was really better in general. Using a different random split before calculating the fitness of every individual carries the same danger: an apparent difference in performance may be due to the particular random order and not due to the different value of the parameters. While repeating the estimate several times and getting an average would probably improve the estimate, the increase in computational requirements makes this approach prohibitive. For example, if we have 10 individuals and we use 10 fold crossvalidation we would have to do 100 trainings per generation. If in addition, we repeat every estimate 10 times to get an average we would have to do 1000 trainings.  Clearly, for real world problems this is not a good solution.

Using the same random split in each generation has an interesting analogy with natural evolution.  In nature the environment (represented by a fitness function in GAs) is likely to vary with time, however, at any particular time all individuals are competing under the same conditions.

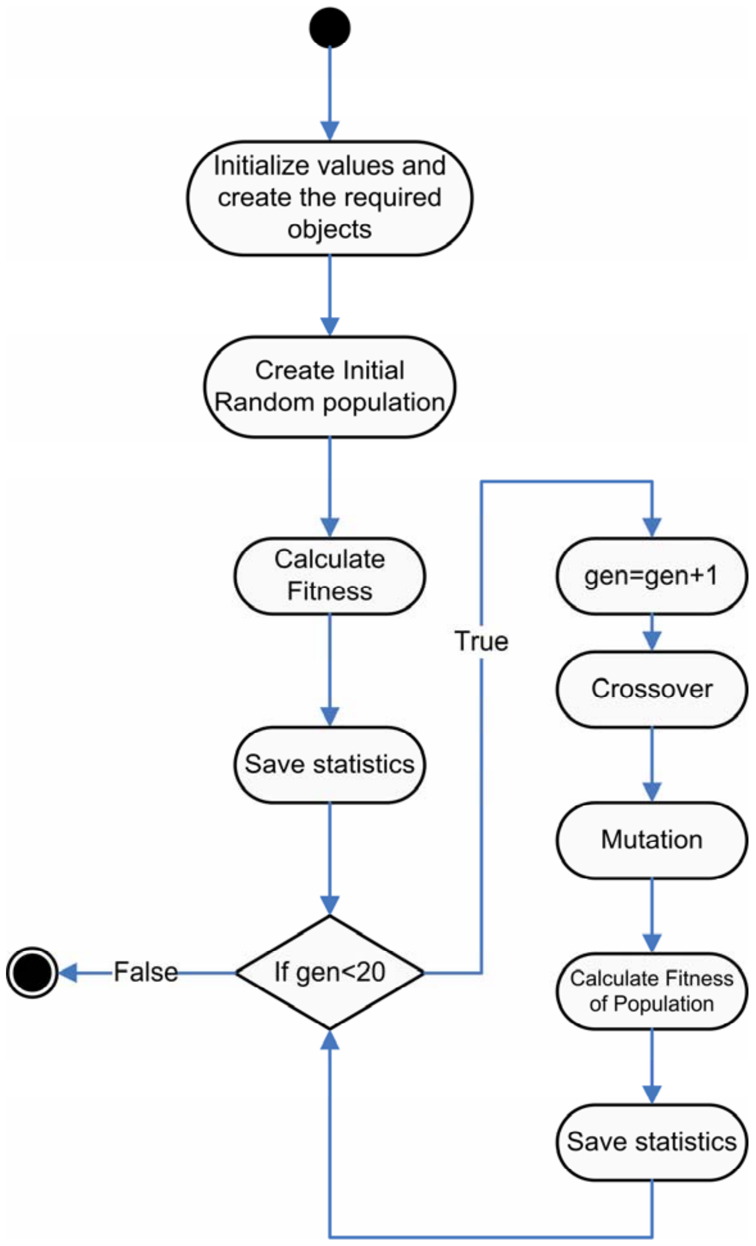

Initialize values and create the required objects
Create Initial Random population
Calculate Fitness
Save statistics
If gen<20
False
True
gen=gen+1
Crossover
Mutation
Calculate Fitness of Population
Save statistics

**Figure 28. Final genetic algorithm.**

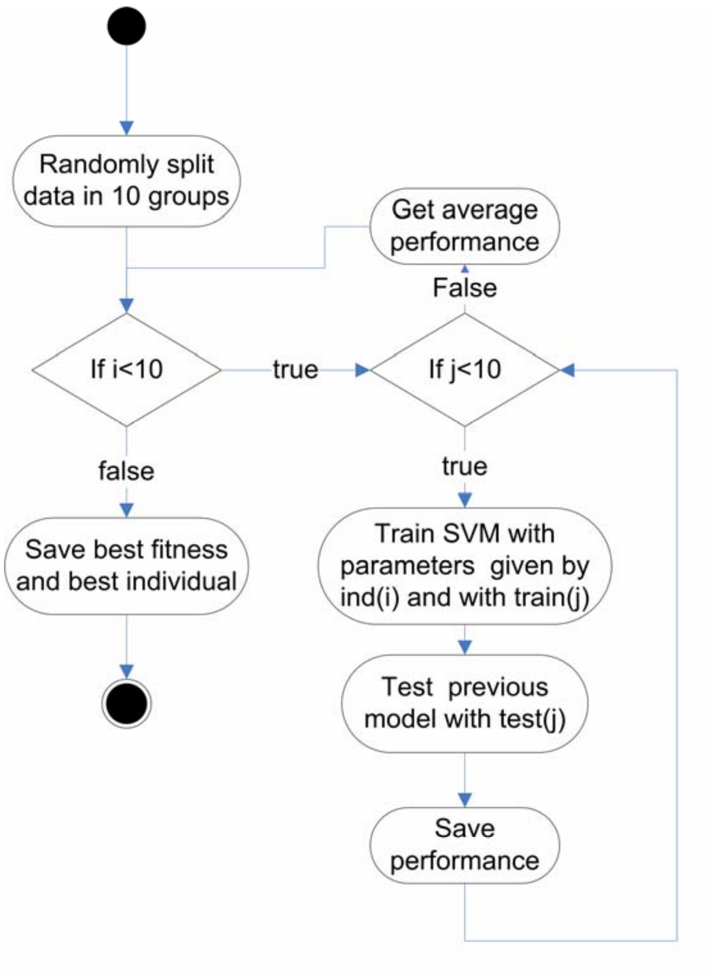

Randomly split data in 10 groups
Get average performance
False
If i<10
true
If j<10
false
true
Save best fitness and best individual
Train SVM with parameters given by ind(i) and with train(j)
Test previous model with test(j)
Save performance

**Figure 29. Calculation of the fitness of the population.**

# CHAPTER 5. EXPERIMENTAL RESULTS

In this chapter, we compare the proposed algorithm with the conventional way of setting the parameters: assigning arbitrarily the kernel, kernel parameters, and penalty value $C$. Also, we compare the algorithm with backpropagation neural networks with different number of hidden nodes.

## 5.1 Comparison with the conventional approach

The experiments are performed with selected individuals of the previously mentioned case study. The individuals were selected according to the worst performance as reported in Rabelo (2000). All 648 data points were use in the experiments.

The generalization performance of the model constructed by the GA was then compared with the performance of a model constructed by arbitrarily selecting the kernel and the kernel parameters. This method of selecting the model will be referred to from now as the *conventional* way. In order to compare the different models the 10-fold crossvalidation was repeated 50 times using the same stream random numbers. This is akin to the common random number technique (Law and Kelton, 2000) to reduce variance. Additionally, the best model from the conventional method was compared with the model created by the GA by a paired t test to determine if the difference was significant.

The model created by the genetic algorithms had the parameters shown in Table 10

**Table 10. Best model found by the genetic algorithm.**

| Dataset | $\gamma$ | $C$ | Degree | p | r |
|---|---|---|---|---|---|
| **Ind7** | 451.637 | 959.289 | 2 | 0.682536 | 1 |
| **Ind10** | 214.603 | 677.992 | 2 | 0.00968948 | 1 |
| **Ind100** | 479.011 | 456.25 | 2 | 0.428016 | 1 |

Interestingly, for 2 datasets (ind7 and ind100) the chosen kernel was a mixture of Gaussian and polynomial kernel.

For the conventional method the kernel is arbitrarily set to Gaussian and the penalty value $C$ was set to 50 while the kernel width $\gamma$ is varied to 0.1, 0.5, 1, 10, and 50. The average generalization error after the 50 replications for 3 individuals from the case study is shown in Table 11 and Table 12 and the Tufte's boxplot (Tufte, 1983) are shown in Figure 30-Figure 32. Notice that in this case, we are comparing percentage of misclassification instead of percentage of correct classifications like in the plots of the performance of the different genetic algorithms as in the previous section.

**Table 11. Performance of models created using the conventional method.**

| Kernel width ($\gamma$) | Ind7 | Ind10 | Ind100 |
|---|---|---|---|
| **0.1** | 23.9168 | 24.3358 | 24.1783 |
| **0.5** | 30.5086 | 29.8396 | 30.4063 |
| **1** | 29.0546 | 28.4365 | 29.2966 |
| **10** | 30.3981 | 46.2980 | 38.2692 |

| | | | |
|---|---|---|---|
| **50** | 30.3981 | 46.2980 | 38.2692 |

**Table 12. Performance of model created using the genetic algorithm.**

| | **Ind7** | **Ind10** | **Ind100** |
|---|---|---|---|
| **GA** | 22.0025 | 21.8491 | 21.9937 |

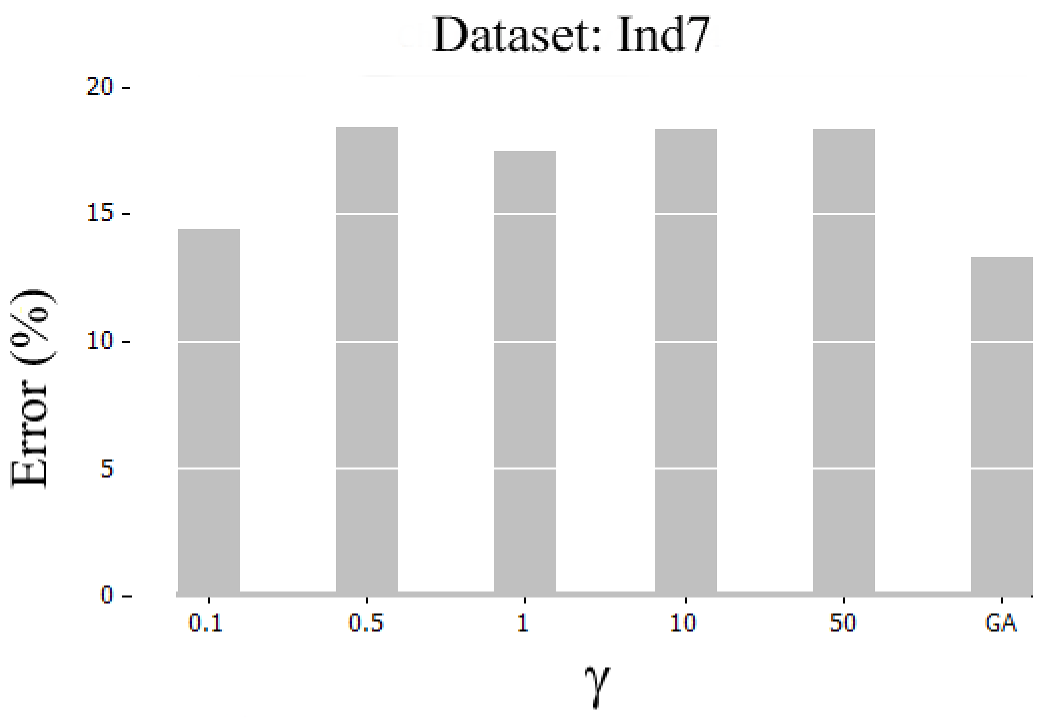


**Figure 30. Average performance of the different models for dataset Ind7.**

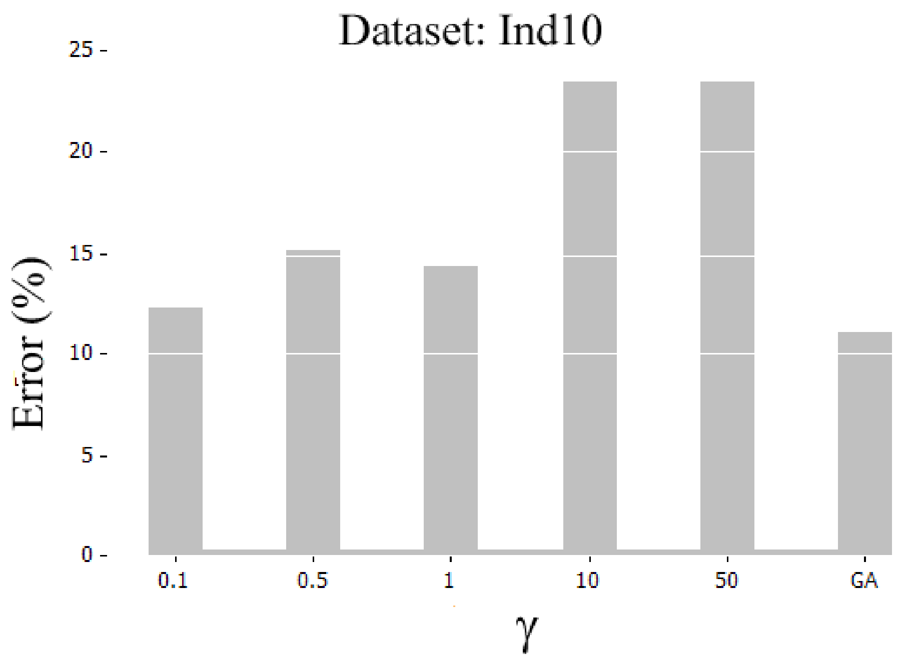


**Figure 31. Average performance of the different models for dataset Ind10.**

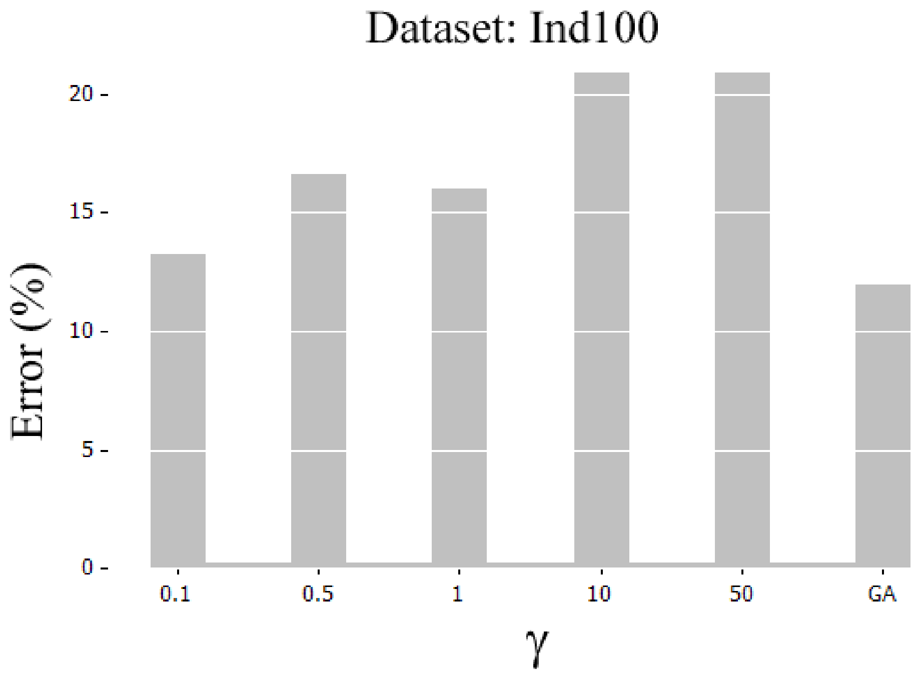


**Figure 32. Average performance of the different models for dataset Ind100.**

The results of a paired t-test of the difference between the performance of best model using the conventional method and the model constructed by the genetic algorithms show that the difference in performance is statistically significant at the 95% level (see Table 13).

**Table 13. Paired t-test of the conventional best model vs the genetic algorithms.**

| **Dataset** | **95% C.I.** **Upper bound** | **p-value** |
|---|---|---|
| **Ind7** | -1.67608 | 0.000 |
| **Ind10** | -2.24952 | 0.000 |
| **Ind100** | -1.97312 | 0.000 |

These experiments show that using genetic algorithms are an effective way to find the optimal parameters for support vector machines.  This method will become particularly important as more complex kernels with more parameters are designed.

## 5.2 Comparison with Backpropagation Neural Networks

In this section, our proposed algorithm is compared to a more traditional approach to learning algorithms: a backprogation neural network (NN).

In order to reduce the variance, 30 pairs of training and testing set were generated for each dataset and then stored (see Figure 33).  At every replication, the dataset is randomly split in training and testing set keeping the proportion of class 1 to class 2 approximately equal to the proportion in the original dataset (stratified holdout). The same training and testing set is used to train and test the SVM with the parameters selected by the GA and the backpropagation neural networks with different number of hidden nodes.

These experiments allow us to use a paired t- test and a 95% confidence interval to determine if there is a statically significant difference between the performances.

The backpropagation NN is trained using the Levenberg-Marquardt algorithm. This algorithm is designed to approach second-order training speed. It approximates the Hessian matrix $\mathbf{H}$ by using the Jacobian matrix $\mathbf{J}$ (which contains the first derivatives of the network errors with respect to the weights) as follows (Demuth and Beale, 1998):

$$\mathbf{H} = \mathbf{J}^T\mathbf{J}$$

And the gradient can be computed as

$$\mathbf{J}^T\mathbf{e}$$

where $\mathbf{e}$ is a vector of the neural network errors. Therefore, the update is modified to be:

$$\mathbf{W}(t+1) = \mathbf{W}(t) - \left[\mathbf{J}^T\mathbf{J} + \beta\mathbf{I}\right]^{-1}\mathbf{J}^T\mathbf{e},$$

where $\beta$ is a constant that is decreased or increased depending on the performance function. Levenberg-Marquardt is considered one of the fastest algorithms for training backpropagation neural networks.

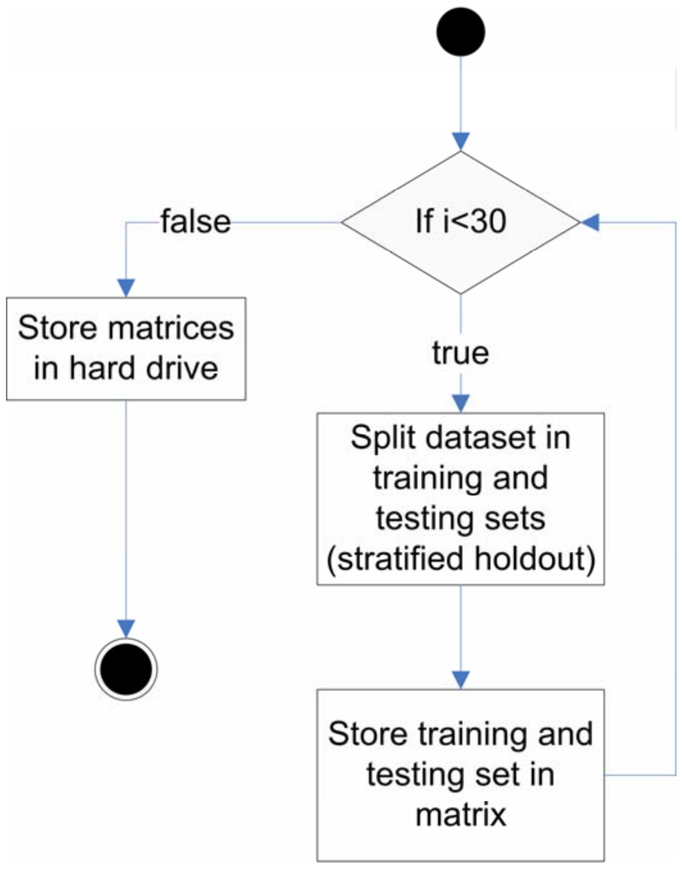

If i<30
false
Store matrices in hard drive
true
Split dataset in training and testing sets (stratified holdout)
Store training and testing set in matrix

**Figure 33. Comparison between the a backpropagation neural network and the proposed algorithm.**

Table 14 shows the mean and standard deviation while Table 15 shows the results from the paired t-test using 10% of the dataset for training. In all cases, the mean test error is lower with the SVM than with the NN (as indicated by the negative test statistics, t-value and Figure 34 -Figure 37). Furthermore, for most cases this difference is statistically significant at the 95% confidence level as indicated by the negative upper bound and the p-value. Only when tested with datasets ind1 the difference is not significant at the 0.05 level (but it was significant at the 0.1 level).

**Table 14. Mean test error with 10% of the dataset (approximately 65 points) used in training.**

| Dataset | | Ind1 | Ind7 | Ind10 | Ind100 |
|---|---|---|---|---|---|
| **SVM** | **Average** | 4.47685 | 34.02635 | 32.53859 | 30.79474 |
| | **Std dev** | 2.481708 | 4.052861 | 3.476982 | 3.61084 |
| **NN 1 node** | **Average** | 7.890222 | 53.43643 | 47.82161 | 46.70097 |
| | **Std dev** | 13.20988 | 21.16148 | 22.86714 | 23.94577 |
| **NN 2 nodes** | **Average** | 8.793595 | 55.75601 | 53.15037 | 47.22127 |
| | **Std dev** | 6.993879 | 17.69439 | 13.72225 | 18.43694 |
| **NN 3 nodes** | **Average** | 8.702118 | 54.78236 | 52.03544 | 47.61578 |
| | **Std dev** | 5.041017 | 13.45559 | 11.10692 | 14.49721 |
| **NN 4 nodes** | **Average** | 10.68039 | 56.67238 | 56.62664 | 47.54718 |
| | **Std dev** | 7.799044 | 9.136175 | 9.696914 | 11.34729 |
| **NN 5 nodes** | **Average** | 12.96169 | 57.44559 | 56.98685 | 52.57862 |
| | **Std dev** | 9.102082 | 8.561429 | 9.839618 | 11.39469 |

**Table 15. Paired t test for the difference between test errors for the SVM vs. the NN with 10% of the dataset (approximately 65 points) used in training.**

| Dataset | 95% Upper bound | p-value | T-value |
|---|---|---|---|
| | SVM vs. best NN | SVM vs. best NN | SVM vs. best NN |
| **Ind1** | 0.85380 | 0.092 | -1.36 |
| **Ind7** | -12.8299 | 0.000 | -5.01 |
| **Ind10** | -8.6208 | 0.000 | -3.90 |
| **Ind100** | -8.6573 | 0.000 | -3.73 |

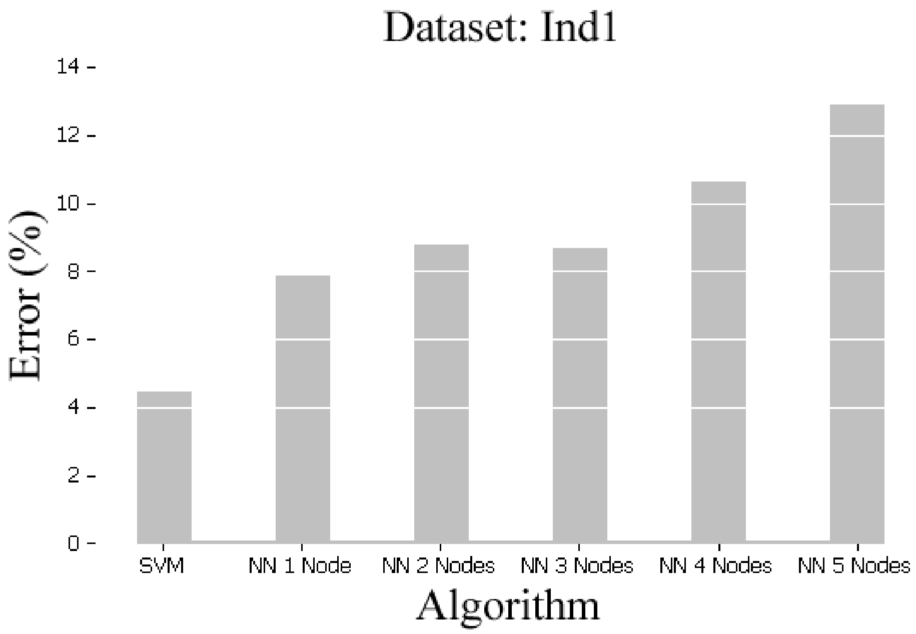


**Figure 34. SVM vs. NN for different architectures of the NN and training with 10% of dataset ind1.**

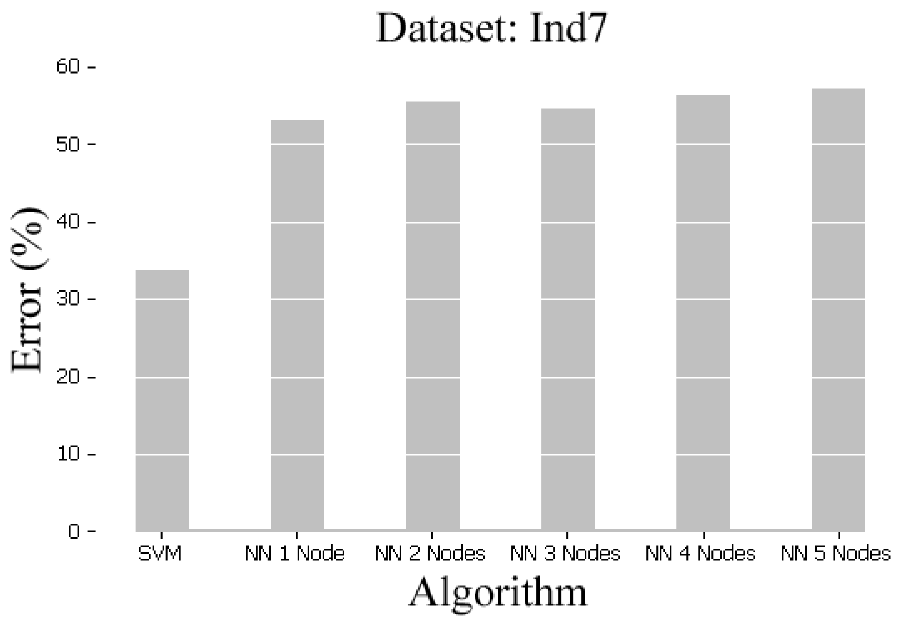


**Figure 35. SVM vs. NN for different architectures of the NN and training with 10% of dataset ind7.**

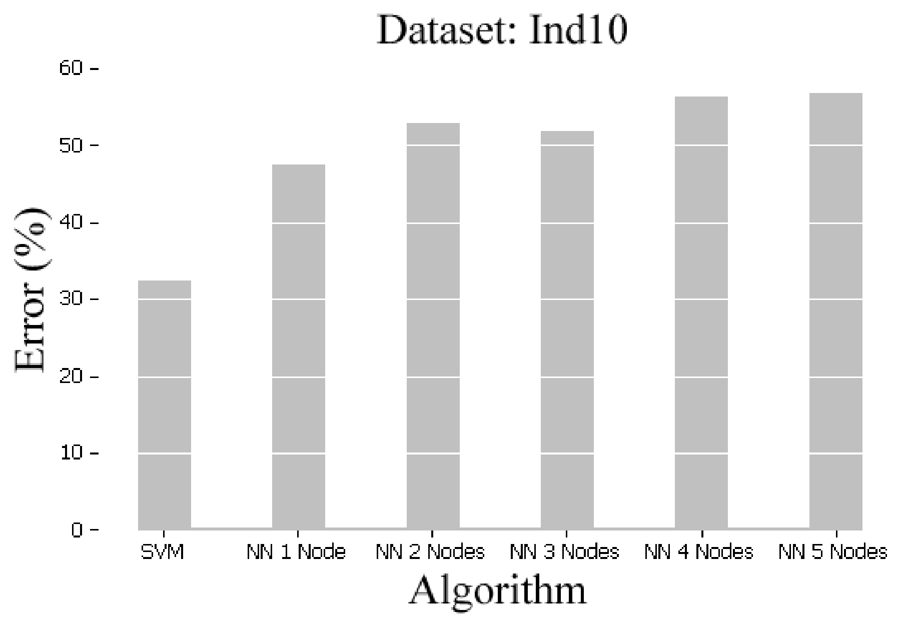


**Figure 36. SVM vs. NN for different architectures of the NN and training with 10% of dataset ind10.**

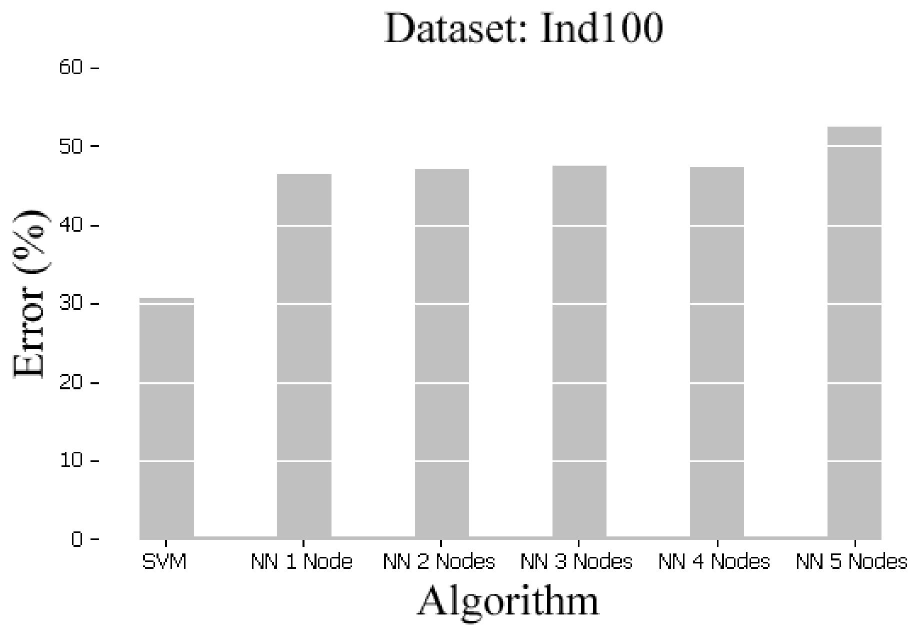


**Figure 37. SVM vs. NN for different architectures of the NN and training with 10% of dataset ind100.**

Table 16 shows the means and standard deviation while Table 17 shows statistical test results when using 20% of the dataset for training. Again, the results show that the SVM produces a lower average testing error and in all cases this difference is statistically significant at the 0.05 level (see Figure 38 to Figure 41). These results agree with the results found in Rabelo (2000), where SVM are compared with several types of NN and it is found that for small datasets the results are significantly better when using SVMs.

**Table 16. Mean test error with 20% of the dataset (approximately 130 points) used in training**

| Dataset | | Ind1 | Ind7 | Ind10 | Ind100 |
|---|---|---|---|---|---|
| **SVM** | **Average** | 4.009023 | 28.96196 | 28.97684 | 28.1789 |
| | **Std dev** | 1.84101 | 4.00994 | 3.542786 | 3.437897 |
| **NN 1 node** | **Average** | 4.98713 | 54.73888 | 48.81595 | 37.98584 |
| | **Std dev** | 2.434057 | 18.52352 | 26.15919 | 16.19415 |
| **NN 2 nodes** | **Average** | 6.563708 | 51.88266 | 51.6731 | 42.87644 |
| | **Std dev** | 2.429254 | 14.73741 | 11.70238 | 13.79561 |
| **NN 3 nodes** | **Average** | 6.801804 | 52.69505 | 50.46976 | 46.80824 |
| | **Std dev** | 2.792366 | 13.96284 | 11.38751 | 12.31946 |
| **NN 4 nodes** | **Average** | 8.912484 | 58.04643 | 52.50322 | 48.55212 |
| | **Std dev** | 4.730942 | 13.19836 | 7.515809 | 11.66712 |
| **NN 5 nodes** | **Average** | 9.536683 | 58.82657 | 54.4659 | 48.79664 |
| | **Std dev** | 3.931462 | 5.374914 | 6.901339 | 8.419554 |

**Table 17. Paired t test for the difference between test errors for the SVM vs. the NN with 20% of the dataset (approximately 130 points)**

| Dataset | 95% Upper bound | p-value | T-value |
|---|---|---|---|
| | SVM vs. best NN | SVM vs. best NN | SVM vs. best NN |
| **Ind1** | -0.089702 | 0.036 | -1.87 |
| **Ind7** | -19.6691 | 0.000 | -7.17 |
| **Ind10** | -11.4120 | 0.000 | -4.00 |
| **Ind100** | -4.93208 | 0.001 | -3.42 |

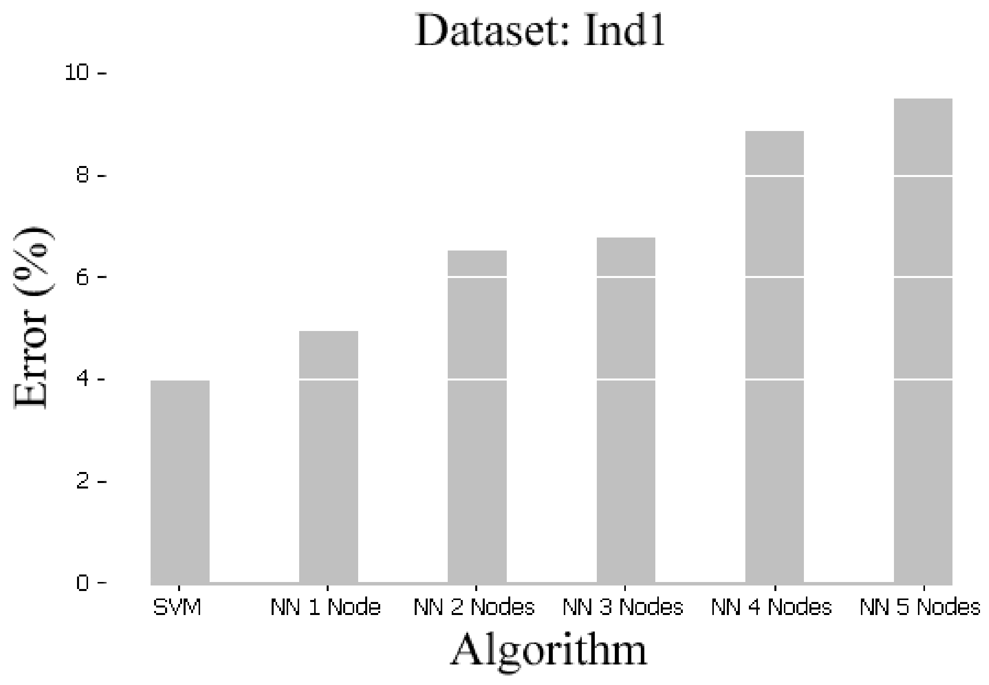


**Figure 38. SVM vs. NN for different architectures of the NN and training with 20% of dataset ind1.**

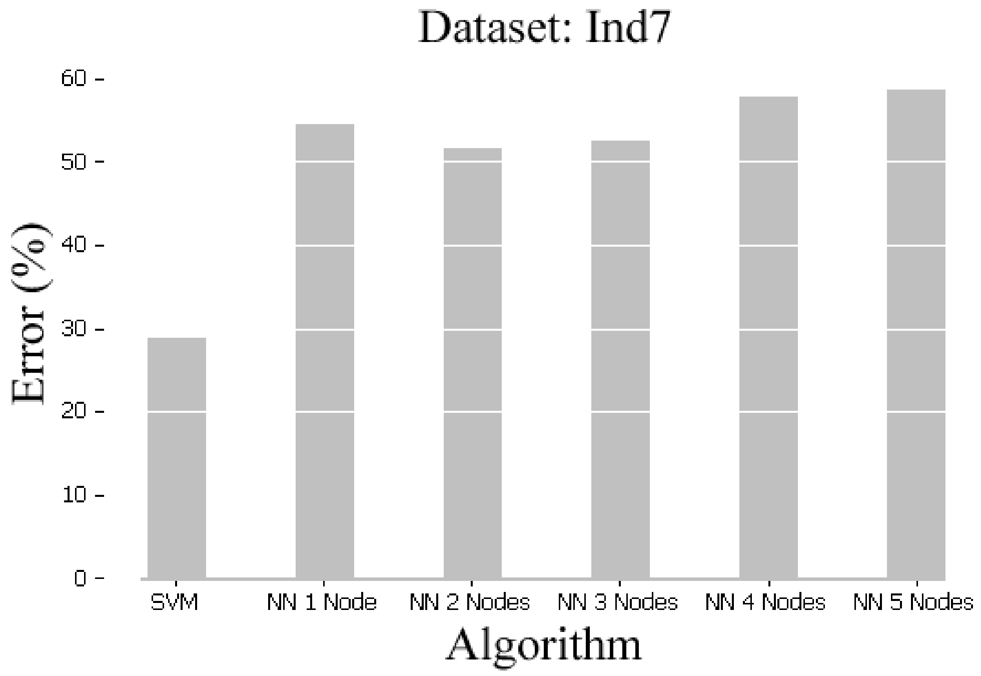


**Figure 39. SVM vs. NN for different architectures of the NN and training with 20% of dataset ind7.**

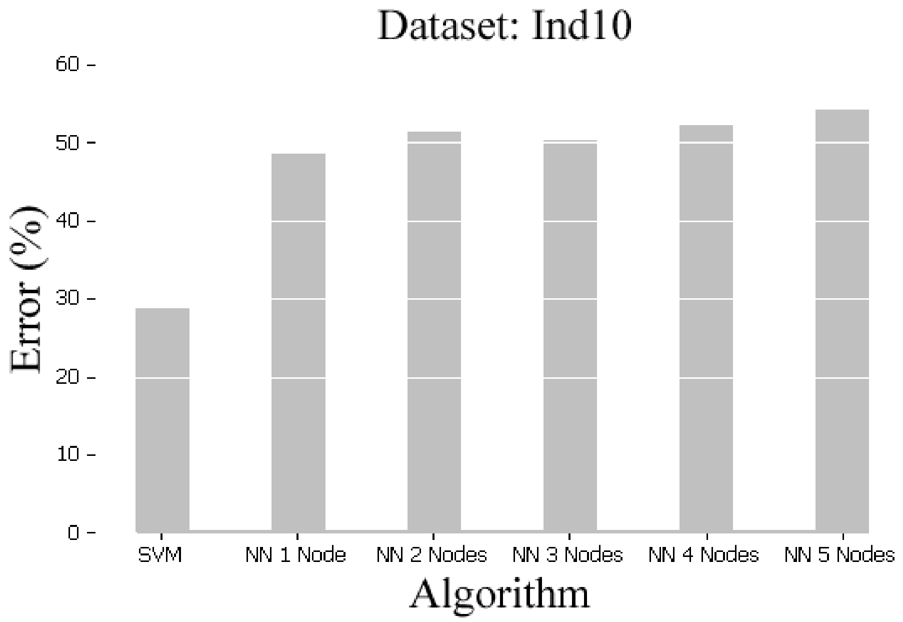


**Figure 40. SVM vs. NN for different architectures of the NN and training with 20% of dataset ind10**

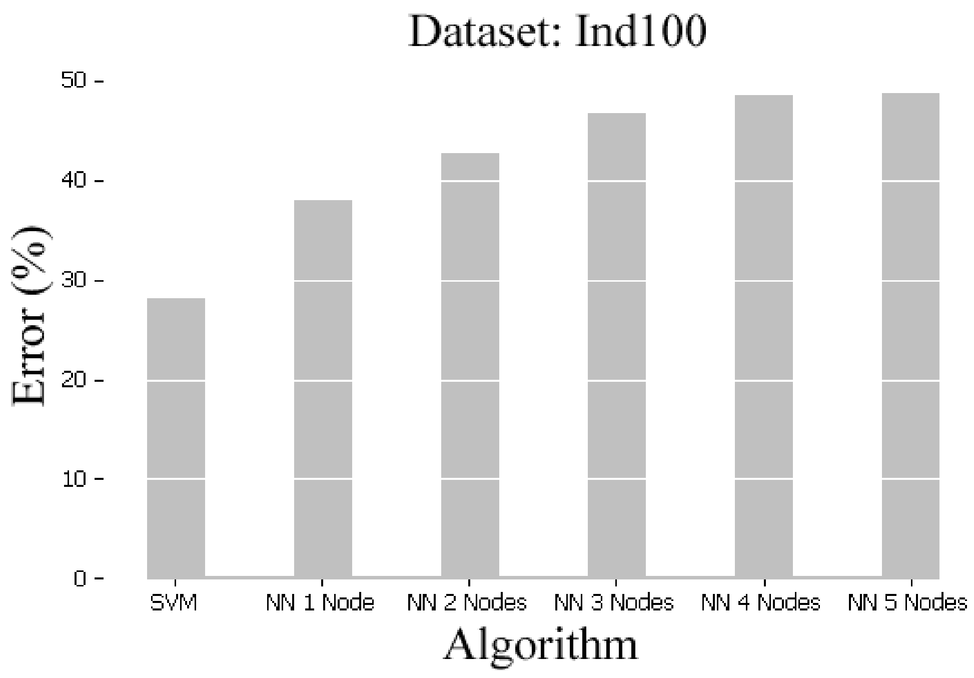


**Figure 41. SVM vs. NN for different architectures of the NN and training with 20% of dataset ind100.**

In both cases the parameters used in the experiments are shown in Table 18 and Table 19.

**Table 18. Parameters for the GA.**

| Parameters | Value |
|---|---|
| **Population** | 10 |
| **Generations** | 5 |
| **Prob. of crossover** | 0.95 |

|  |  |
|---|---|
| **Prob. of mutation** | 0.05 |
| **Fitness function** | 10-fold crossvalidation |
| **Selection** | Tournament selection |
| **Crossover types** | Diagonal with 4 parents |
| **Mutation type** | Fixed rate |
| **Others** | Elitist strategy |

**Table 19. Parameters for NN.**

| **Parameters** | **Value** |
|---|---|
| **Training** | Levenberg-Marquardt |
| **Epochs** | 300 |
| **Transfer Function** | Tan-sigmoid/purelin |
| **Hidden nodes** | 1-5 |
| **Layers** | 2 |

## 5.3 A final experiment

One last experiment was meant to test the algorithm in a more realistic situation. We randomly split the data in half: training and testing. Then, we take a sample of 10, 25, 50, 100, and 324 points from the training set. Finally, we train the NN and the SVM with those subset 30 times (see Figure 42).

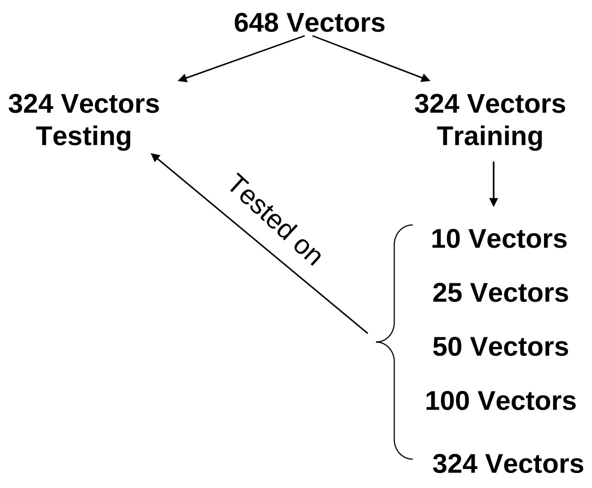


**Figure 42. Final experiments.**

The results are shown in Table 20. In all cases, the SVM had a lower mean classification error and a smaller standard deviation. It is important to notice that for big sample sizes the difference in performance between SVM and NN tend to become smaller. For instance, the difference in performance between SVM and the best NN model for a 10 points training set was 10.7% and for 324 points it was 4.85%.

**Table 20. Mean and standard deviation of the percentage of incorrect classification for the final experiments.**

| Dataset | | 10 | 25 | 50 | 100 | 324 |
|---|---|---|---|---|---|---|
| SVM | Mean | 31.18 | 18.91 | 20.97 | 13.85 | 8.87 |
| | Std Dev | 1.94 | 6.98 | 3.22 | 2.25 | 1.63 |
| NN 1 node | Mean | 43.67 | 25.80 | 41.62 | 19.22 | 13.72 |
| | Std Dev | 19.26 | 18.55 | 11.45 | 10.23 | 11.78 |
| NN 2 nodes | Mean | 41.88 | 27.08 | 36.31 | 21.91 | 14.53 |

| | | | | | | |
|---|---|---|---|---|---|---|
| | **Std Dev** | 17.67 | 15.47 | 12.80 | 10.44 | 7.30 |
| **NN 3 nodes** | **Mean** | 44.34 | 30.25 | 39.91 | 22.90 | 16.98 |
| | **Std Dev** | 13.10 | 14.03 | 13.26 | 9.79 | 8.78 |
| **NN 4 nodes** | **Mean** | 44.04 | 28.88 | 44.60 | 23.901 | 17.28 |
| | **Std Dev** | 11.51 | 14.94 | 12.64 | 7.98 | 6.11 |
| **NN 5 nodes** | **Mean** | 44.43 | 32.28 | 45.98 | 31.48 | 18.51 |
| | **Std Dev** | 11.29 | 13.62 | 12.08 | 12.54 | 5.45 |

Rabelo (2000) compared a SVM with optimal width selected by a GA with several other techniques like Fuzzy Art Map (FAM), backpropagation NN, FAM with voting schemes, frequency random generator (Fran), and Pure Random Generator (Pran). A reproduction of the results for one dataset is shown in Table 21.

**Table 21. Results for dataset ind2 from Rabelo (2001).**

| **Training File** | **SVM** | **FAM** | **Voting** | **BP** | **Pran** | **Fran** |
|---|---|---|---|---|---|---|
| **10** | 76.2% | 77.8% | 78.4% | 78.1% | 49.1% | 72.8% |
| **25** | 76.2% | 75.9% | 75.0% | 76.2% | 53.1% | 71.6% |
| **50** | 77.2% | 76.2% | 74.1% | 76.5% | 46.6% | 63.9% |
| **100** | 76.2% | 72.2% | 79.6% | 75.3% | 52.2% | 60.5% |
| **324** | 80.9% | 81.2% | 83.3% | 84.0% | 49.4% | 62.0% |

It is important to note a couple of issues about these last experiments. First, the random splits are not the same as used in Rabelo (2001). Second, in Table 21 the reported results are one observation and not a mean.

## 5.4 Processing times considerations

The previous experiments showed that 10-fold crossvalidation is an adequate estimate of the generalization error and has allowed us to guide the genetic algorithm to a good set of parameters for a SVM. Nevertheless, there is an important drawback: the processing time. Every time we use crossvalidation, we need to train the SVM ten times. Furthermore, in case of the genetic algorithm, we need to repeat the process for each individual of the population. For instance, for a population of 10 individuals, we need to do 100 trainings at each generation. The experiments were done on a Athlon XP 2500+ computer with 1.84 GHz and 256 MB of RAM. With a training set of 130 points each generation required approximately 7 seconds. From the experiments, it seems that 5 generation is enough to get a good answer. Therefore, every run of the GA requires at least 35 seconds. In contrast, a run of the neural network required approximately 3 seconds. Clearly, there is a need to find a more efficient estimate of the generalization error to replace the 10 fold crossvalidation.

# CHAPTER 6. CONCLUSIONS AND FURTHER RESEARCH

This chapter presents the summary of the work done, the main results and contributions, and future directions of research.

## 6.1 Summary of Work and Results

In this thesis, we explore and propose a way to use genetic algorithms to optimize the parameters of a SVM. Currently, the proposed algorithm makes use of 10-fold crossvalidation as its fitness function. However, the whole algorithm has been designed to take advantage of interfaces (abstract classes) to facilitate the testing of other fitness functions and operators. Several types of crossover and mutation for the genetic algorithm are implemented and compared. It is seen that a diagonal crossover with 4 parents and a fixed mutation rate provided the best performance. Also, 10-fold crossvalidation is compared with an efficient estimate of the generalization error known as the $\xi\alpha$ and it is found that this last estimator was biased for high values of $C$.

The SVM engine is based on a C++ version of LIBSVM (Chang and Lin, 2001). This implementation is modified to include a kernel that is a mixture of Gaussian and polynomial kernels. As a result, the genetic algorithm has the flexibility to decide on one or the other or a mix of both kernels. Additionally, this is a more complicated kernel that requires more parameters to set and it better shows the significance of having an automated technique to find the optimal values.

The results of the experiments shows significant improvement over using a SVM with fixed architecture and over other learning algorithms like Neural Networks trained with the Levenberg-Marquardt algorithm and with different architectures.

Finally, we should state that this improvement in performance comes with the price of an increased processing time. This drawback will not be an issue once an efficient and unbiased estimate of the performance of SVMs is found.

## 6.2 Contributions

In this thesis, a genetic algorithm was designed for the purpose of finding the optimal parameters for a SVM trained on an e-commerce dataset. Several types of crossover and mutation operators were implemented and tested. These experiments indicated that a diagonal crossover with 4 parents and simple mutation with fixed rate had the best performance in terms of convergence time and variance.

While the importance of using GAs for finding optimal parameters might not seem so great for SVM with simple kernels like a Gaussian RBF with only one parameter to set, as applications continue to appear and new, more complicated kernels (and likely with more parameters) are designed for specific problems, this need will become apparent. For this reason a kernel which is a mixture of RBF and complete polynomial kernel was used in the experiments. This kernel was previously tested in regression problems by other researchers and also seems to be providing good results for classification problems.

It was also shown that 10 fold crossvalidation is a good estimator of the generalization performance of support vector machines and it allowed us to guide the genetic algorithm to good values for the parameters of the SVM. In addition, we explored the possibility of using the efficient bound to leave-one-out known as $\xi\alpha$. The experiments indicated that this estimator tended to diverge with respect to the 10 crossvalidation estimate for high values of $C$.

## 6.3 Further Research Issues

- Further research is needed in developing more efficient methods to estimate the generalization error of support vector machines. Several researchers have proposed algorithms; however, whether they are predictive so that we can use them for model selection is still an open question. Preliminary experiments comparing the $\xi\alpha$ with crossvalidation seems to indicate that this estimate is not suited for parameter selection using the tested e-commerce dataset.
- Another possible area of research is the possibility of using genetic algorithms to improve the current training methods for support vector machines. For instance, genetic algorithms may be adequate to replace the current heuristics used to determine which two points to optimize at each iteration of the SMO algorithm.
- The same methodology used here to find the optimal parameters for support vector classification can be applied to regression problems without many difficulties.
- It would also be interesting to compare the current algorithms with other machine learning techniques like decision trees, fuzzy ART map, RBF Neural Network, and others.

# REFERENCES

S. Ali and K. A Smith, "Automatic parameter selection for polynomial kernel," in *Proc. of the IEEE International Conference on Information Reuse and Integration*, 2003

T. Bäck and M. Schütz, “Intelligent Mutation Rate Control in Canonical Genetic Algorithms,” *in Proc. ISMIS*, 1996, pp. 158-167.

T. Baeck, D. Fogel, Z. Michalewicz, *Evolutionary Computation 1: Basic Algorithms and Operators*, Institute of Physics, 2000.

M. Bazaraa, H. Shetty and C. M. Sherali, *Nonlinear Programming: Theory & Applications*. Wiley, 1994

C.J.C. Burges, “A Tutorial on Support Vector Machines for Pattern Recognition,” *Data Mining and Knowledge Discovery*, vol. 2 (2), pp. 121-167, June 1998.

P. Burman, “A comparative study of ordinary cross-validation, v-fold cross validation and the repeated learning testing methods,” *Biometrika* vol. 76 (3), pp. 503-514, 1989.

O. Chapelle, V. Vapnik, O. Bousquet and S. Mukherjee, “Choosing Multiple Parameters for Support Vector Machines,” *Machine Learning*, vol. 46(1), pp. 131-159, 2002.

X. Chen, “Gene selection for cancer classification using Bootstrapped Genetic Algorithms and Support Vector Machines,” in *Proc. Computational Systems Bioinformatic*s, 2003.

Chih-Chung Chang and Chih-Jen Lin, LIBSVM : a library for support vector machines, 2001. Software available at http://www.csie.ntu.edu.tw/~cjlin/libsvm.

N. Cristianini, C. Campbell and J. S. Taylor, “Dynamically Adapting Kernels in Support Vector Machines,” in *Advances in Neural Information Processing Systems*, vol. 11, M. Kearns, S. Solla and D. Cohn Ed., MIT Press, 1999, pp. 204-210.

N. Cristianini, J. Shawe-Taylor, *An Introduction to Support Vector Machines*, Cambridge, MA: University Press, 2000.

H. Demuth and M. Beale, Neural Network Toolbox (MATLAB), The MATH WORKS Inc., 1998.

T.G. Dietterich, “Approximate statistical tests for comparing supervised classification learnign algorithms,” *Neural Computation*, vol. 10(7), 1988, pp. 1895-1924.

K. Duan, S.S. Keerthi and A.N. Poo, “Evaluation of simple performance measures for tuning SVM hyperparameters,” *Neurocomputing* vol. 51 pp. 41-59, April 2003.

D. Dumitrescu, B. Lazzerini, L.C. Jain and A. Dumitrescu, *Evolutionary Computation.* CRC Press, 2000.

A.E. Eiben, “Multiparent Recombination in evolutionary computing,” in *Advances in Evolutionary Computing*, A. Ghosh and S. Tsutsui, Eds., Natural Computing Series, Springer, 2002, pp. 175-192.

P. A. Fishwick and R. B. Modjeski, Ed., *Knowledge Based Simulation: Methodology and Application*. Springer Verlag, 1991.

T. Frieß, N. Cristianini and C. Campbell, “The Kernel-Adatron: a Fast and Simple Learning Procedure for Support Vector Machines,” in *Proceedings of the Fifteenth International Conference on Machine Learning*, J. Shavlik Ed, 1998 pp. 188-196.

H. Frohlich, O. Chapelle, B. Scholkopf, “Feature selection for support vector machines by means of genetic algorithm,” in *Proc. 15th IEEE International Conference on Tools with Artificial Intelligence*, pp. 142-148, 2003.

D. E. Goldberg, "Real-coded Genetic Algorithms, Virtual Alphabets, and Blocking," in: *Complex Systems*, Vol. 5, Complex Systems Publications, Inc., 1991, pp. 139-167.

D. E. Goldberg, *Genetic Algorithms in Search, Optimization and Machine Learning*. Boston, MA: Addison-Wesley, 1989.

R. Herbrich, *Learning Kernel Classifiers Theory and Algorithms.* The MIT Press 2001.

Holland, J. H., *Adaptation in Natural and Artificial Systems*. University of Michigan Press, Ann Arbor 1975.

T. Joachims, “Estimating the Generalization Performance of a SVM Efficiently,” University of Dortmund, LS-8 Report 25, 1999.

L. Kaufman, “Solving the Quadratic Programming Problem Arising in Support Vector Classification,” *Advances in Kernel Methods—Support Vector Learning*, MIT Press, 1998.

S. S. Keerthi, C.-J. Lin, “Asymptotic Behaviors of Support Vector Machines with Gaussian Kernel,” *Neural Computation Archive*, vol. 15 (7), July 2003.

R. Kohavi. ”A study of crossvalidation and bootstrap for accuracy estimation and model selection,” in *Proc. of the Fourteenth International Joint Conference on Artificial Intelligence*, Morgan Kaufman Publishers, Inc., pp. 1137-1143, 1995.

A. M. Law and W. D. Kelton, *Simulation  Modeling and Analysis*. 3rd edition,  McGraw-Hill, 2000 .

A. Lendasse, V. Wertz and M. Verleysen, “Model Selection with Cross-Validations and Bootstraps - Application to Time Series Prediction with RBFN Models,” in *Proc. ICANN*, 2003, pp. 573-580.

J.K. Martin and D.S. Hirschberg, “Small sample statistics for classification error rates, I: Error rate measurements,” *Tech. Rpt.* 96-21, ICS Dept., UC Irvine, 1996.

M. Mitchell, *An Introduction to Genetic Algorithms*, Bradford Books, 1998.

W. Mendenhall and T. Sincich, *Statistics for engineering and the sciences*. 4th edition, Englewood Cliffs, NJ: Prentice Hall, 1995.

Z. Michalewicz and C. Z. Janikow, *Genetic Algorithms + Data Structures = Evolution Programs.* Springer-Verlag, 1996.

J. Platt, “How to implement SVMs,” *IEEE Intelligent Systems*, July/August 1998.

A. T. Quang, Q.-L. Zhang and X. Li, “Evolving support vector machine parameters,” in *Proc 2002 International Conference on Machine Learning and Cybernetics*, 2002.

L. Rabelo, What intelligent agent is smarter? A comparison, *MS Thesis*, M.I.T., 2001.

J. Rothenberg, “Tutorial: artificial intelligence and simulation,” in *Proc.. of the 21st conference on Winter simulation*, p.33-39, December 04-06, 1989, Washington, D.C., United States

K. Ryan, Success Measures of Accelerated Learning Agents for e-Commerce, *MS Thesis*, M.I.T., September 1999.

B. Schölkopf and A.J. Smola, *Learning with Kernels*, Cambridge, MA: MIT Press, 2002.

J. Sepúlveda-Sanchis, G. Camps, E. Soria, S. Salcedo, C. Bousoño, G. Sánz and J. Marrugat J., “Support Vector Machines and Genetic Algorithms for Detecting Unstable Angina,” *Computers in Cardiology*, IEEE Computer Society Press, Menphis (U.S.A.) 2002.

X. Shao, V. Cherkassky, “Multi-Resolution Support Vector Machine,” in *Proc. IJCNN* vol. 2, 1999, pp. 1065-1070.

G. F. Smits and E. M. Jordaan, Improved SVM Regression using Mixtures of Kernels, in *Proc. IJCNN* vol. 3, 2002, pp. 2785-2790.

D. Thierens, “Adaptive mutation rate control schemes in genetic algorithms,” in *Proc. 2002 IEEE World Congress on Computational Intelligence: Congress on Evolutionary Computation*, 2002,  pp. 980-985.

E.R. Tufte, *The Visual Display of Quantitative Information.* Cheshire, Conn: Graphic Press, 1983 pp. 1667-1689.

V. N. Vapnik, *The Nature of Statistical Learning Theory*. New York: Springer-Verlag, 1999.

J. A. Vasconcelos, J. A. Ramírez, R.H.C. Takahashi and R.R. Saldanha, “Improvements in genetic algorithms,” *IEEE Transactions on Magnetics*, vol. 37 (5), pp.3414-3417, September, 2001.

S. M. Weiss, “Small sample error rate estimation for k nearest neighbor classifiers,” *IEEE Transactions on Pattern Analysis ad Machine Intelligence,* vol. 13(3), pp. 285-289, 1991.

S. M. Weiss and N. Indurkhya, “Decision tree pruning: Biased or optimal,” in *Proc. 12th national conference on artificial intelligence,* AAAI Press and MIT Press, pp. 626-632, 1994.

M. Wooldridge, *An Introduction to Multiagents Systems*. Chichester, England: John Wiley & Sons, 2002.

Z. Xiangrong and L. Fang, “A pattern Classification Method based on GA and SVM”, *in Proc. iCSP* 2002.

L. Xuefeng, L. Fang,”Choosing multiple parameters for SVM based on genetic algorithm”, in Proc. 6th International Conference on Signal Processing, vol. 1, 2002, pp.117-119.

X. Yao, “Evolving artificial neural networks,” in *Proc. of the IEEE*, vol. 87(9), pp.1423-1447, September 1999.

L. Zhang; W. Zhou and L.C. Jiao, “Pre-Extracting Support Vectors For Support Vector Machine,” *Electronic Transaction*, vol. 3, pp. 383-386, 2001.