THE IMPLEMENTATION OF SUPPORT VECTOR MACHINES USING THE SEQUENTIAL MINIMAL OPTIMIZATION ALGORITHM

by
Ginny Mak

Supervisors: Professor G. Ratzer, Professor H. Vangheluwe

April 2000

School of Computer Science
McGill University, Montreal, Canada

A Master's Project Submitted in Partial Fulfilment of Requirements for
the Master of Science Degree

Preface

> ***I seem to have been only like a boy playing on the seashore, and diverting myself in now and then finding a smoother pebble or a prettier shell than ordinary, whilst the great ocean of truth lay all undiscovered before me.***

Issac Newton, 1642-1727

Throughout human history, great scientists and philosophers inspired us with their discoveries as well as their wit and wisdom. From the very beginning when the idea of implementing Support Vector Machines using the Sequential Minimal Optimization Algorithm germinated, to the time this project report was completed, I not only gained insight and understanding about SVM but personally experienced the truths some of these great men and women passed on to us. As I was evaluating this project and the written report, I discovered that during the process of developing the research project and writing the individual chapters, there is a philosophical truth which I can and did learn from each chapter. I thought it would be a good idea to share this wisdom with my readers by putting them at the beginning of each chapter in the report.

The readers will encounter the terms Lagrange multiplier(s) and Lagrangian function in this report a lot. They were named after the greatest mathematician of the eighteen century, Joseph Louis Lagrange (1736-1813) whose biography can be found in the book “A Short Account of the History of Mathematics” (4th edition, 1908) by W.W. Rouse Ball. It is my intention that the contributions of our great scientists and mathematicians will never be forgotten. I hope the readers find this added dimension as meaningful as I do.

Ginny MF. Mak

Montreal, April 2000

Acknowledgments

Many individuals have contributed to this project report with their advice, interest, time and support. They are the wind beneath my wings.

I thank God for making everything possible.

I would like to thank Professor Hans Vangheluwe who supervised my project. His technical advice, ideas and constructive criticism contributed to the success of this report. I am deeply indebted to him for taking the time to review the drafts in such detail.

I thank Professor Gerald Ratzer, my second supervisor, who critiqued the drafts and offered valuable suggestions concerning the content and layout of the report.

This work would not have been possible without the broad vision of both Professors Vangheluwe and Ratzer who, in supervising this project, supported me in my exploration of a new research area.

I want to express my deep gratitude to my cousin Leslie in Oregon for her financial support during these two years at McGill. I also thank my cousin for her sense of humor, encouragement and above all her philosophy in life - that life is not just about grades and financial success, but about being yourself, pursuing what you enjoy most and being compassionate to others. Her loaning me her laptop computer has made working on this project much easier.

I thank Professor Mary Seitz of Oregon who not only taught me writing skills but above all the ability to think critically and analytically through the classical and controversial articles we discussed in class. Many thanks to Professor Joseph Albert, Professor Cynthia Brown and my friend John May, all of Portland, Oregon for offering me their advice and words of encouragement. I also thank Professor Andrew Tolmach of Portland, Oregon for convincing me of the importance of mathematics in the realm of computer science; Professor Ken Mock currently with Washington State University for introducing me to the excellent SVM tutorial written by Christopher Burges of Bell Laboratories, which sparked my interest in Support Vector Machine.

Additionally, I want to thank Masumi Aizawa of Tochigi, Japan; Amanda Tran of Columbia, Maryland; Mary Rutt of Portland, Oregon; Elizabeth and Keith Man of Calgary, Alberta, for their warm and caring friendship which has always been an invaluable part of my life.

Finally I want to express my gratitude to the late Emile Hayhoe of Toronto, originally from Montreal and a McGill alumnus, who was like a mother to me. My fond memory of her kindness, open-mindedness and understanding helped me make the decision to do my graduate studies at McGill University.

Contents

List of Figures

List of Tables

The beginning is the most important part of the work.

— Plato 427-347BC —

1 Introduction

Support vector machines, SVMs, are a machine learning technique, which is based on the Structural Risk Minimization Principle [20]. The purpose of this project is to implement a support vector machine on a personal computer using John Platt's Sequential Minimal Optimization Algorithm so that a better understanding of the theory behind SVM can be gained and the report of the project can serve as an introduction of SVM to readers who are not familiar with the subject. This report consists of six sections. In the second section, the theoretical overview of SVM is given followed by a description of how SVM works in different cases of binary pattern recognition and a broad survey of some recent applications of SVMs. The third section describes the Sequential Minimial Optimization Method, SMO, which is one of the many methods to speed up SVM implementation. The fourth section details the software implementation of this project describing the design, the modules making up the software, and the data structures. The results of tests conducted using artificial data and benchmarks together with their analysis are presented in the fifth section. Finally, we conclude with a summary of the overall project results and the issues learned in the implementation in the last section.

Support Vector Machines are a relatively recent machine learning technique. There are still many unexplored areas and unanswered questions of both theoretical and practical nature in this field. While the basic theory behind SVM is not hard to understand, it does require some knowledge of optimization using Lagrange multipliers as well as basic matrix theory. A review of the basic mathematics needed to understand the theoretical underpinnings of SVM is included in the appendix.

I cannot teach anybody anything.
I can only make them think.

The beginning of wisdom is
a definition of terms.

— Socrates —

2

Support Vector Machines

2.1 Theory

Machine learning is a search problem. Given a task of classifying some data into a number of classes, a computer program has to search for an hypothesis, which can correctly predict the class label y for an input data $\mathbf{x}$, from the hypothesis space.

Let say we have a set of data drawn from an unknown distribution $P(\mathbf{x},y)$[1]

$$(\mathbf{x_1},y_1),\ldots,(\mathbf{x_l},y_l)\ ,\ \mathbf{x_i}\in\mathbb{R}^n\ ,\ y_i\in\{-1,1\}.$$

We are also given a set of decision functions, or hypothesis space [12]

$$\{f_\lambda : \lambda\in\Lambda\}.$$

where Λ (an index set) is a set of abstract parameters, not necessarily vectors. f_λ is also called a hypothesis.

$$f_\lambda : \mathbb{R}^n \longrightarrow \{-1,+1\}.$$

The set of functions f_λ could be a set of Radial Basis Functions or a multi-layer neural network. In the latter case, the set Λ is the set of weights of the neural network.

For a given function f_λ, the expected risk (the test error) $R(\lambda)$ is the possible average error committed by f_λ on the unknown example drawn randomly from the sample distribution $P(\mathbf{x},y)$

$$R(\lambda)=\int \frac{1}{2}|f_\lambda(\mathbf{x})-y|dP(\mathbf{x},y).$$

$R(\lambda)$ is a measure of how well a hypothesis f_λ predicts the correct label y from an input $\mathbf{x}$.

Since the sample distribution P is unknown, we have to use inductive principles to minimize the risk. This is achieved by sampling the data and computing a stochastic approximation to the actual risk. This approximation is called empirical risk $R_{emp}(\lambda)$ (training error)

$$R_{emp}(\lambda)=\frac{1}{l}\sum_{i=1}^{l}|f_\lambda(\mathbf{x_i})-y_i|,$$

where l is the number of samples.

[1] The joined probability of the combination of $\mathbf{x},y$ happen.

The Empirical Risk Minimization Principle

If R_{emp} converges in probability to R, the expected risk, then the minimum of R_{emp} may converge to the minimum of the expected risk R. The Empirical Risk Minimization principle allows us to make inferences based on the data set only if the the convergence holds, otherwise the principle is not consistent [12]. Vapnik and Chervonenkis showed that the convergence to the minimum of R holds if and only if the convergence in probability of R_{emp} to R is replaced by a uniform convergence in probability (Theory of Uniform Convergence in probability).

$$\lim_{l\to\infty} P\{\sup_{\lambda\in\Lambda}(R(\lambda) - R_{emp}(\lambda)) > \varepsilon\} = 0 \qquad \forall\, \varepsilon > 0.$$

Moreover, the necessary and sufficient condition for consistency of the Empirical Risk Minimization principle is the finiteness of the VC-dimension h of the hypothesis space H [12]. The VC-dimension is further explained in the next section.

2.2 The VC-dimension

The VC-dimension or Vapnik Chervonenkis dimension of a set of functions is the maximum number h of vectors $\mathbf{z_1}, \ldots, \mathbf{z_h}$ that can be separated into two classes in all 2^h possible ways, by using functions in the set. In other words, VC dimension is the maximum number of vectors that can be shattered by the set of functions[2]. In Figure 2.1, each dashed line separates three vectors into two classes. For three vectors, the maximum possible ways to separate them into two classes by a function which is a straight line is 2^3. Hence the VC-dimension of a set of straight lines is 3. On the other hand, Figure 2.2 shows that no straight line can shatter four vectors into two classes. If for any n, there exists a set of n vectors which can be shattered by the set of functions $\{f_\lambda, \lambda \in \Lambda\}$, then the VC dimension of that set of functions is equal to infinity [20].

The ability of a set of functions (a hypothesis space) to shatter a set of instances is closely related to the inductive bias of a hypothesis space [10]. The VC-dimension of a hypothesis space is a measure of the complexity or expressiveness of that hypothesis space. The larger the VC-dimension , the larger the capacity (expressiveness) of a learning machine to learn without any training error. An unbiased hypothesis space is one which can represent every possible concept definable over the instance space $\mathbf{X}$ and is therefore very expressive [10]. It can shatter the whole instance space $\mathbf{X}$. However, a learning machine with an unbiased hypothesis space cannot generalize well. In order that such a learning machine can converge to a final target function, it has to be presented with every instance in the instance space $\mathbf{X}$ as a training example. Christopher Burges wrote in his "A tutorial on support vector machines for pattern recognition" that

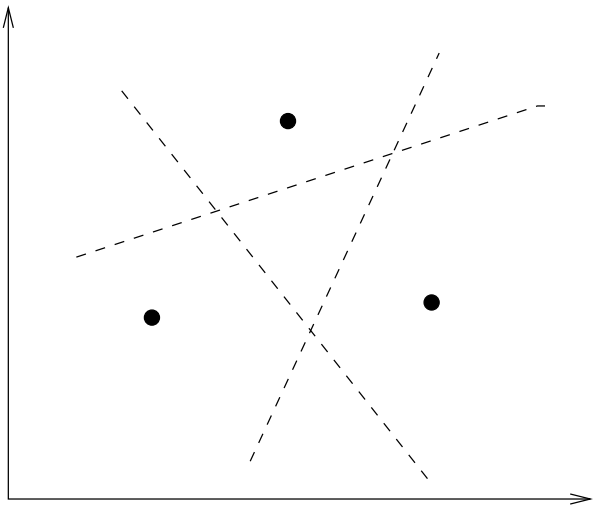
> A machine with too much capacity is like a botanist with a photographic memory who, when presented with a new tree, concludes that it is not a tree because it has a different number of leaves from anything she has seen before; a machine with too little capacity is like the botanist's lazy brother, who declares that if it's green, it's a tree. Neither can generalize well.

The theory of uniform convergence in probability developed by Vapnik and Chervonenkis gives an upper bound of the expected risk, R, with a probability of $1-\psi$ as

$$R(\lambda) \leq R_{emp}(\lambda) + \sqrt{\frac{h\left(\ln\frac{2l}{h}+1\right) - \ln\frac{\psi}{4}}{l}}\,, \qquad \forall\lambda \in \Lambda \tag{2.1}$$

where h is the VC dimension of f_λ and l is the number of data. The second term of the right hand side of equation 2.1 is called the VC-confidence.

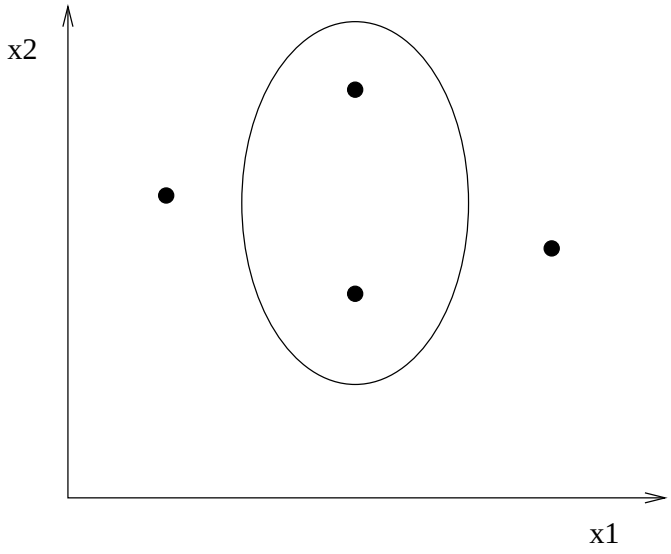
[2] There are 2^n ways to label a set of n points in two classes. If for each labelling of the set of n points, a member of a set of function can be found which can correctly label the points, then the set of points is shattered by that set of function.



f is a line in a plane. The VC dim. of {f} is 3, since they shatter 3 vectors.

(The Nature of Statistical Learning Theory, Vapnik, 1995).

Figure 2.1: VC dimension



No straight line can shatter four vectors.

(The Nature of Statistical Learning Theory, Vapnik, 1995).

Figure 2.2: Shattering 4 vectors

To get a small R, we need a small R_{emp} and a small ratio of $\frac{h}{l}$ at the same time. h depends on the set of functions which the learning machine can implement. R_{emp} depends on the f_λ chosen by the learning machine and it decreases with increasing h. We have to choose an optimal h especially when l, the number of data, is small in order to get good performance. This is where Structural Risk Minimization, an improved induction principle, comes in.

2.2.1 Structural Risk Minimization

Let $S = \{ f_\lambda : \lambda \in \Lambda \}$ be a set of functions and Λ_k be a subset of Λ, and

$$S_k = \{ f_\lambda : \lambda \in \Lambda_k \}.$$

We define a structure of nested subset

$$S_1 \subset S_2 \subset \ldots \subset S_n \subset \ldots \tag{2.2}$$

such that the corresponding VC dimension of each subset S_n satisfies the condition

$$h_1 \leq h_2 \leq \ldots \leq h_n \leq \ldots$$

Each choice of a structure S produces a learning algorithm. For a given set of l examples $\mathbf{z_1}, \ldots, \mathbf{z_l}$, Structural Risk Minimization principle chooses the function $f_{\lambda_l^n}$ in the subset $\{ f_\lambda : \lambda \in \Lambda_n \}$ for which the second term of equation 2.1 is minimal[17].

However, implementing SRM can be difficult because the VC dimension of S_n could be hard to compute. Even if we can compute h_n of S_n, finding

$$min \left(R_{emp}(\lambda) + \sqrt{\frac{h_n}{l}} \right).$$

among h_n's is hard[3].

Support Vector Machines, SVM, are able to achieve the goal of minimizing the upper bound of $R(\lambda)$ by minimizing a bound on the VC dimension h and $R_{emp}(\lambda)$ at the same time during training. The structure on which SVMs are based is a set of separating hyperplanes.

Let X be a N-dimensional instance space. We have a set of vectors $\{\mathbf{x_1}, \mathbf{x_2}, \ldots, \mathbf{x_l}\}$ where $x_i \in X$. Each hyperplane $\mathbf{w}.\mathbf{x} - b = 0$ will correspond to a canonical pair (unique pair) $(\mathbf{w}, b)$ if we put a constraint that

$$min|\mathbf{w}.\mathbf{x_i} - b| = 1, \qquad \text{i = 1,...,l.} \tag{2.3}$$

This means that the point closest to the hyperplane has a distance of $1/\|\mathbf{w}\|$. The VC-dimension of the canonical hyperplanes is $N+1$[20]. For Structural Risk Minimization to be applicable, we need to construct sets of hyperplanes of varying VC-dimension so that we can minimize the VC-dimension and the empirical risk simultaneously. This is achieved by adding a constraint on $\mathbf{w}$ in the following way.

Let R be the radius of the smallest ball $B_{\mathbf{x_1},\ldots,\mathbf{x_l}}$ which contains $\mathbf{x_1}, \mathbf{x_2}, \ldots, \mathbf{x_l}$

$$B_{\mathbf{x_1},\ldots,\mathbf{x_l}} = \{\mathbf{x} \in X : \|\mathbf{x} - \mathbf{a}\| < R\}, \mathbf{a} \in \mathbf{X}.$$

We also define a decision function $f_{\mathbf{w},b}$ so that[1]

$$f_{\mathbf{w},b} : B_{\mathbf{x_1},\ldots,\mathbf{x_l}} \longrightarrow \{\pm 1\},$$

$$f_{\mathbf{w},b} = sgn((\mathbf{w}.\mathbf{x}) - b).$$

[3] We ignore the log factor in equation 2.1

The possibility of introducing a structure on the set of hyperplanes is based on the result of Vapnik[20] that the set of canonical hyperplanes

$$\left\{ f_{\mathbf{w},b} = sgn(\mathbf{w}.\mathbf{x} - b) \;\; | \|\mathbf{w}\| \leq A \right\} \tag{2.4}$$

has a VC-dimension h, which satisfies the following bound[12]

$$h \leq min\{\lceil R^2 A^2 \rceil, N\} + 1. \tag{2.5}$$

Removing the constraint $\|\mathbf{w}\| \leq A$ gives us a set of functions whose VC-dimension is $N+1$ where N is the dimensionality of X. By adding the constraint $\|\mathbf{w}\| \leq A$, we can get VC-dimensions that are much smaller then N and hence allow us to work in a very high dimensional space[17].

Geometrically speaking, the distance from a point $\mathbf{x}$ to the hyperplane defined with a $(\mathbf{w}, b)$ is

$$d(\mathbf{x}; \mathbf{w}, b) = \frac{|\mathbf{w}.\mathbf{x} - b|}{\|\mathbf{w}\|}.$$

Equation 2.3 tells us that the closest distance between the canonical hyperplane $(\mathbf{w}, b)$ and the data points is $\frac{1}{\|\mathbf{w}\|}$. Since $\|\mathbf{w}\| \leq A$, the closest distance between the hyperplane and the data points must be greater than $\frac{1}{A}$ (see equation 2.4). The constrained set of hyperplanes will have a distance of at lease $\frac{1}{A}$ from the data points. This is equivalent to putting spheres of radius $\frac{1}{A}$ around each data point and consider only those hyperplanes that do not intersect any of the spheres[12].

If the set of training examples are linearly separable, then the SVM which is based on Structural Risk Minimization is to find among the canonical hyperplanes the one with the minimum norm ($\|\mathbf{w}\|^2$) because a small norm gives a small VC-dimension h (see equation 2.5). In the next section, we will show that minimizing $\|\mathbf{w}\|^2$ is equivalent to maximizing the distance between the two convex hulls of the two classes of data, measured along a line perpendicular to the separating hyperplane[12].

2.3 How SVM works

We will illustrate how SVMs work by describing the three different cases in binary pattern classification.

Linearly Separable

The data to be classified are linearly separable.

The general equation of a plane in n-dimensions is $\mathbf{w} \cdot \mathbf{x} = b$ where $\mathbf{x}$ is a $n \times 1$ vector, $\mathbf{w}$ is the normal to the hyperplane and b is a (scalar) constant. Of all the points on the plane, one has minimum distance d_{min} to the origin

$$d_{min} = \frac{|b|}{\|\mathbf{w}\|}.$$

Training patterns $(\mathbf{x_1}, y_1)$,...,$(\mathbf{x_l}, y_l)$ are given where $\mathbf{x_i}$ is a d-dimensional vector and

$$y_i = 1 \qquad \text{if } \mathbf{x_i} \text{ is in class A,}$$

$$y_i = -1 \qquad \text{if } \mathbf{x_i} \text{ is in class B.}$$

If the data are linearly separable, then there exist a d-dimensional vector $\mathbf{w}$ and a scalar b such that

$$\mathbf{w} \cdot \mathbf{x_i} - b \geq 1 \qquad \text{if } y_i = 1; \qquad \mathbf{w} \cdot \mathbf{x_i} - b \leq 1 \qquad \text{if } y_i = -1. \tag{2.6}$$

In compact form, equation 2.3 can be written as

$$y_i(\mathbf{w} \cdot \mathbf{x_i} - b) \geq 1 \tag{2.7}$$

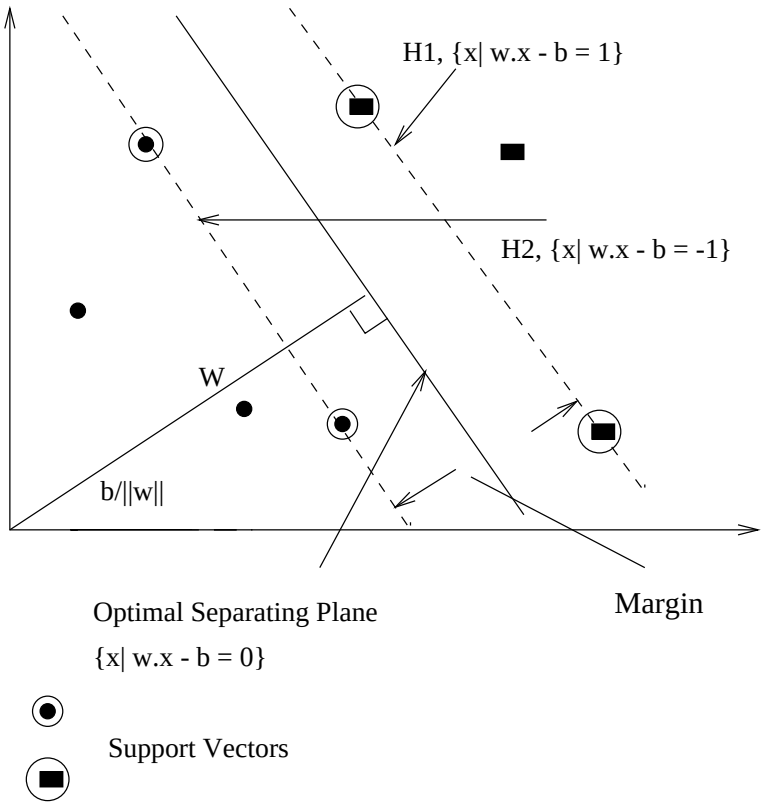


Figure 2.3: Optimal separating plane and support vectors

or

$$-(y_i(\mathbf{w}\cdot\mathbf{x_i}-b)-1)\leq 0.$$

$(\mathbf{w},b)$ defines the hyperplane separating the two classes of data. The equation of the hyperplane is $\mathbf{w}\cdot\mathbf{x}-b=0$ where $\mathbf{w}$ is normal to the plane, b is the minimum distance from the origin to the plane. In order to make each decision surface $(\mathbf{w},b)$ unique, we normalize the perpendicular distance from the origin to the separating hyperplane by dividing it by $\|\mathbf{w}\|$, giving the distance as $\frac{|b|}{\|\mathbf{w}\|}$.
As depicted in Figure 2.3, the perpendicular distance from the origin to hyperplane $H_1 : \mathbf{w}\cdot\mathbf{x_i}-b=1$ is $\frac{|1+b|}{\|\mathbf{w}\|}$

The perpendicular distance from the origin to hyperplane $H_2 : \mathbf{w}\cdot\mathbf{x_i}-b=-1$ is $\frac{|b-1|}{\|\mathbf{w}\|}$. The support vectors are defined as the training points on H_1 and H_2. Removing any points not on those two planes would not change the classification result, but removing the support vectors will do so. The margin, the distance between the two hyperplanes H_1 and H_2 is $\frac{2}{\|\mathbf{w}\|}$. The margin determines the capacity of the learning machine which in turn determines the bound of the actual risk - the expected test error $R(\lambda)$ (see equation 2.1). The wider the margin the smaller is h, the VC-dimension of the classifier. Therefore our goal is to maximize $\frac{2}{\|\mathbf{w}\|}$ which is equivalent to minimizing $\frac{\|\mathbf{w}\|^2}{2}$. Now we can formulate our problem as

Minimize

$$f=\frac{\|\mathbf{w}\|^2}{2}$$

subject to

$$g_i=-(y_i(\mathbf{w}\cdot\mathbf{x_i}-b)-1)\leq 0\,,\qquad \text{i = 1,...,}l$$

where l is the number of training examples.

This problem can be solved by using standard Quadratic Programming techniques. However, using the Lagrangian method to solve the problem makes it easier to extend it to the non-linear separable case presented in the next section. To get acquainted with the subject of Lagrange multipliers and Karush-Kuhn-Tucker (KKT) conditions, "Introduction to operational research" by J. G. Eiker and M. Kupferschmid [5] is a good reference. A review of Lagrangian Method and Karush-Kuhn-Tucker conditions is included in the appendix.

The **Lagrangian function** of this problem is

$$\begin{aligned} L_p(\mathbf{w}, b, \Lambda) &= f(\mathbf{x}) + \sum_{i=1}^{l} \lambda_i g_i(\mathbf{x}) \\ &= \frac{\mathbf{w}\cdot\mathbf{w}}{2} - \sum_{i=1}^{l} \lambda_i (y_i(\mathbf{w}\cdot\mathbf{x_i} - b) - 1) \\ &= \frac{\mathbf{w}\cdot\mathbf{w}}{2} - \sum_{i=1}^{l} \lambda_i y_i \mathbf{w}\cdot\mathbf{x_i} + \sum_{i=1}^{l} \lambda_i y_i b + \sum_{i=1}^{l} \lambda_i \end{aligned}$$

where $\Lambda = (\lambda_1, \lambda_2, \ldots, \lambda_l)$ is the set of Lagrange multipliers of the training examples.

The **KKT conditions** for this problem is
Gradient condition:

$$\frac{\partial L_p}{\partial \mathbf{w}} = \mathbf{w} - \sum_{i=1}^{l} \lambda_i y_i \mathbf{x_i} = \mathbf{0}, \tag{2.8}$$

where $\frac{\partial L_p}{\partial \mathbf{w}} = \left(\frac{\partial L_p}{\partial w_1}, \frac{\partial L_p}{\partial w_2}, \ldots, \frac{\partial L_p}{\partial w_d}\right)$

$$\frac{\partial L_p}{\partial b} = \sum_{i=1}^{l} \lambda_i y_i = 0, \tag{2.9}$$

$$\frac{\partial L_p}{\partial \lambda_i} = g_i(\mathbf{x}) = 0. \tag{2.10}$$

Orthogonality condition:

$$\lambda_i g_i = -\lambda_i (y_i(\mathbf{w}\cdot\mathbf{x_i} - b) - 1) = 0 \qquad i = 1, \ldots l. \tag{2.11}$$

Feasibility condition:

$$-y_i(\mathbf{w}\cdot\mathbf{x_i} - b) + 1 \leq 0 \qquad i = 1, \ldots l. \tag{2.12}$$

Non-negativity condition:

$$\lambda_i \geq 0 \qquad i = 1, \ldots l. \tag{2.13}$$

The stationary point of the Lagrangian determines the solution to the optimization problem. Substituting equations 2.8 and 2.9 into the right hand side of the Lagrangian function reduces the function into the dual function with λ_i as the dual variable. The dual problem after the substitution is:
Maximize

$$L_D = \sum_{i=1}^{l} \lambda_i - \frac{1}{2} \sum_{i=1}^{l} \sum_{i=1}^{l} \lambda_i \lambda_j y_i y_j \mathbf{x_i} \cdot \mathbf{x_j} \tag{2.14}$$

subject to

$$\sum_{i=1}^{l} \lambda_i y_i = 0$$

$$\lambda_i \geq 0 \qquad i = 1, \ldots, l.$$

We can find all the λ's by solving equation 2.14 with Quadratic Programming. $\mathbf{w}$ can then be obtained from equation 2.8

$$\mathbf{w} = \sum_{i=1}^{l} \lambda_i y_i \mathbf{x_i}.$$

λ_i is > 0 at the point which is a support vector and the corresponding constraint is active. It is zero at the point which is not a support vector and the corresponding constraint is inactive.
We can calculate b using equation 2.11

$$\lambda_i (y_i (\mathbf{w} \cdot \mathbf{x_i} - b) - 1) = 0 \qquad i = 1, \ldots, l \tag{2.15}$$

by choosing the $\mathbf{x_i}$ with nonzero λ.
The class of an input data $\mathbf{x}$ is then determined by

$$class(\mathbf{x}) = sign(\mathbf{w} \cdot \mathbf{x} - b) \tag{2.16}$$

In practice because of numerical implementaton, the mean value of b is found by averaging all the b's computed using equation 2.15.

Figure 2.4 shows the result of training a set of data which are linearly separable with a C parameter of 4.5. C is a parameter which can be imagined as a penalty factor. The larger C is the narrower the margin and there will be less training error. A smaller C gives a wider margin with more of the outliers included. The figure was produced with the interactive Python program **demo.py** (see Appendix B for the interactive window of the GUI and a brief description of the program). Margin support vectors are circled with dark circles and bound support vectors are circled with light circles. Bound support vectors are those vectors whose Lagrange multipliers equal the C parameter. These bound support vectors lie closer to the separating plane than the margin support vectors. In some cases they lie on the other side of the separating plane and are training errors (misclassified points). The middle line is the optimal separating plane found by the linear SVM. The two light lines are lines joining the margin support vectors.

2.3.1 Linearly Nonseparable Case

If a linearly separating hyperplane does not exist, we could still search for a linear decision plane by introducing a set of variables ξ which measure the degree of violation of the constraints for a linearly separable case - $y_i(\mathbf{w} \cdot \mathbf{x_i} - b) - 1 \geq 0$. (See equation 2.7).

The problem is then formulated as follows:

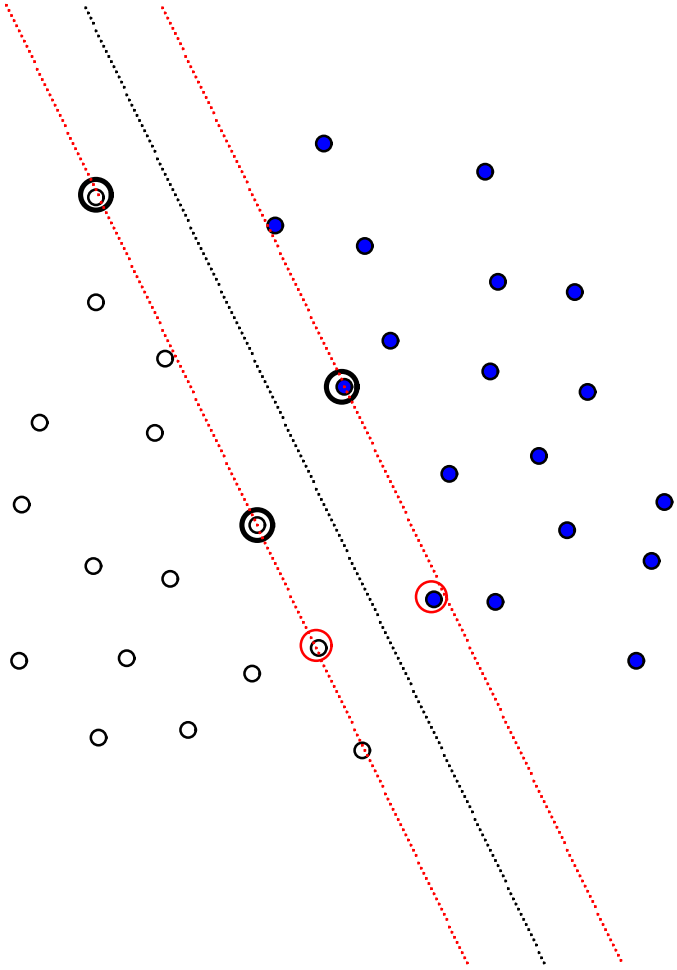


Figure 2.4: Linearly separable data, trained with C = 4.5

Minimize

$$f(\mathbf{w},\Xi) = \frac{1}{2}\|\mathbf{w}\|^2 + C\sum_{i=1}^{l}\xi$$

subject to

$$y_i(\mathbf{w}\cdot\mathbf{x_i} - b) \geq 1 - \xi_i\,, \qquad \text{i=1,...,l}$$
$$\xi_i \geq 0\,, \qquad \text{i=1,...,}l$$

where C is determined by the user and $\Xi = (\xi_1,\ldots,\xi_l)$. One can imagine C as a penalty for errors. A small C maximizes the margin and the hyperplane is less sensitive to the outliers in the training data. A large C minimizes the number of misclassified points.

Now the Lagrangian function is:

$$L_p = \frac{1}{2}\|\mathbf{w}\|^2 + C\sum_{i=1}^{l}\xi_i - \sum_{i=1}^{l}\lambda_i(y_i(\mathbf{x_i}\cdot\mathbf{w} - b) - 1 + \xi) - \sum_{i=1}^{l}\mu_i\xi_i$$

where μ_i are the Lagrangian variables introduced by the constraint $\xi_i \geq 0$

The **KKT conditions** for the problem are:

Gradient condition:

$$\frac{\partial L}{\partial \mathbf{w}} = \mathbf{w} - \sum_{i=1}^{l}\lambda_i y_i \mathbf{x_i} = \mathbf{0} \tag{2.17}$$

where $\frac{\partial L}{\partial \mathbf{w}} = \left(\frac{\partial L}{\partial w_1}, \frac{\partial L}{\partial w_2}, \ldots, \frac{\partial L}{\partial w_d}\right)$

$$\frac{\partial L}{\partial b} = \sum_{i=1}^{l}\lambda_i y_i = 0, \tag{2.18}$$

$$\frac{\partial L}{\partial \xi_i} = C - \lambda_i - \mu_i = 0, \tag{2.19}$$

$$\frac{\partial L}{\partial \lambda_i} = -(y_i(\mathbf{x_i}\cdot\mathbf{w} - b) - 1 + \xi_i). \tag{2.20}$$

Orthogonality condition:

$$\lambda_i(y_i(\mathbf{x_i}\cdot\mathbf{w} - b) - 1 + \xi_i) = 0\,, \qquad i = 1,\ldots,l. \tag{2.21}$$

Feasibility condition:

$$(y_i(\mathbf{x_i}\cdot\mathbf{w} - b) - 1 + \xi_i) \geq 0\,, \qquad i = 1,\ldots,l. \tag{2.22}$$

Nonnegativity condition:

$$\xi_i \geq 0\,, \qquad i = 1,\ldots,l \tag{2.23}$$
$$\lambda_i \geq 0\,, \qquad i = 1,\ldots,l \tag{2.24}$$
$$\mu_i \geq 0\,, \qquad i = 1,\ldots,l \tag{2.25}$$

$$\mu_i \xi_i = 0 \,, \qquad i = 1, \ldots, l \tag{2.26}$$

Substituting equations 2.17 and 2.18 into the right side of the Lagrangian function, we obtain the following dual problem with the dual Lagrangian variables as before.

Maximize

$$L_D = \sum_{i=1}^{l} \lambda_i - \frac{1}{2} \sum_{i,j=1}^{l} \lambda_i \lambda_j y_i y_j \mathbf{x_i} \cdot \mathbf{x_j}$$

subject to:

$$0 \leq \lambda_i \leq C \,,$$

$$\sum_{i=1}^{l} \lambda_i y_i = 0.$$

The solution for $\mathbf{w}$ is found by equation 2.17 which describes one of the KKT condition:

$$\mathbf{w} = \sum_{i=1}^{l} \lambda_i y_i \mathbf{x_i}$$

Again b can be found by averaging over all the training examples' b values, which are calculated by using the following two KKT conditions:

$$\lambda_i (y_i (\mathbf{w} \cdot \mathbf{x_i} - b) - 1 + \xi_i) = 0$$

$$(C - \lambda_i) \xi_i = 0$$

The above equations also indicate that $\xi_i = 0$ if $\lambda_i < C$. Therefore b can be averaged over only those examples in which $0 \leq \lambda_i < C$.

If $\lambda_i < C$, then $\xi_i = 0$. The support vectors lie at a distance of $\frac{1}{\|\mathbf{w}\|}$ from the separating hyperplane. These are *margin support vectors*. When $\lambda_i = C$, the support vectors are misclassified points if $\xi_i > 1$. When $0 < \xi \leq 1$ the support vectors are classified correctly but are closer than the $\frac{1}{\|\mathbf{w}\|}$ from the hyperplane. These are bound support vectors. See Figure 2.5. Figure 2.6 and Figure 2.7 show the results of training by means of the **demo.py** program on data which are not linearly separable. The training in Figure 2.6 was done with a C parameter of 4.0 while that in Figure 2.7 was done with a C parameter of 10.5. Margin support vectors are circled with dark circles and bound support vectors are circled with grey circles. One can see the margin narrows with increase of C and there are less bound support vectors which are support vectors with their Lagrange multipliers equal C. Notice the change in orientation of the optimal separating plane.

2.3.2 Non linear decision surface

In most classfication cases, the separating plane is non-linear. However, the theory of SVM can be extended to handle those cases as well. The core idea is to map the input data $\mathbf{x}$ into a feature space of a higher dimension (a Hilbert space of finite or infinite dimension) [14] and then perform linear separation in that higher dimensional space.

$$\mathbf{x} \longrightarrow \varphi(\mathbf{x}),$$

$$\mathbf{x} = (x_1, x_2, \ldots, x_n),$$

$$\varphi(\mathbf{x}) = (\varphi_1(\mathbf{x}), \varphi_2(\mathbf{x}), , \ldots, \varphi_n(\mathbf{x}), \ldots),$$

where $\varphi_n(\mathbf{x})$ are some real functions. There is an optimal separating hyperplane in a higher dimension, which corresponds to a nonlinear separating surface in input space. A very simple example to illustrate this concept is to visualize

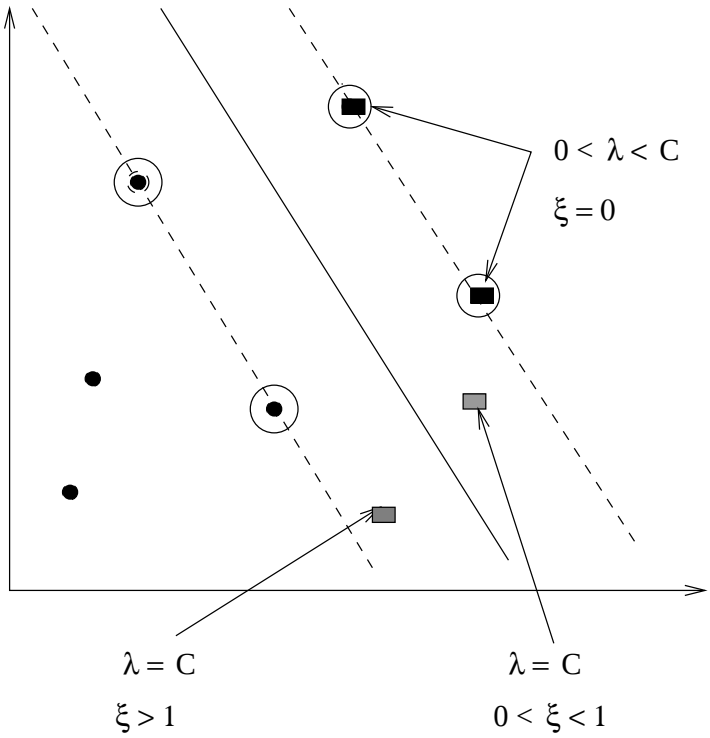


Figure 2.5: Marginal and bound support vectors

separating a set of data in a 2-dimensional space whose decision surface is a circle. See Figure 2.8. We can map each data point (x_1, x_2) into a 3 dimensional feature space.

Data inside the circle are mapped to points on the surface of the lower sphere whereas the ones outside the circle are mapped to points on the surface of the upper sphere. The decision surface is linear in the 3-dimensional sphere.

The solution of the SVM has the same form as the linear separating case. See equation 2.16.

$$\text{Class of data } = \text{ sign } (\varphi(\mathbf{x}) \cdot \mathbf{w} - b) = \text{ sign } \left(\sum_{i=1}^{l} y_i \lambda_i \varphi(\mathbf{x_i}) \cdot \varphi(\mathbf{x}) - b \right). \tag{2.27}$$

We do not need to know the function φ in order to do our calculation. A function K called *kernel function* with the properties that

$$K(\mathbf{x}, y) \equiv \varphi(\mathbf{x}).\varphi(y)$$

will simplify equation 2.27 to

$$sign \left(\sum_{i=1}^{l} y_i \lambda_i K(\mathbf{x_i}, \mathbf{x}) - b \right).$$

We can find kernels, which identify certain families of decision surfaces. Mercer's Theorem [2] gives a criterion for deciding if a kernel can be written in the form $\varphi(\mathbf{x}).\varphi(\mathbf{y})$.

Mercer's Theorem

$$K(\mathbf{x}, \mathbf{y}) = \varphi(\mathbf{x}) \cdot \varphi(\mathbf{y}) \text{ iff}$$

$$K(\mathbf{x}, \mathbf{y}) = K(\mathbf{y}, \mathbf{x}) \text{ and}$$

$$\int \int K(\mathbf{x}, \mathbf{y}) f(\mathbf{x}) f(\mathbf{y}), d\mathbf{x} d\mathbf{y} \geq 0 , \qquad f \in L^2$$

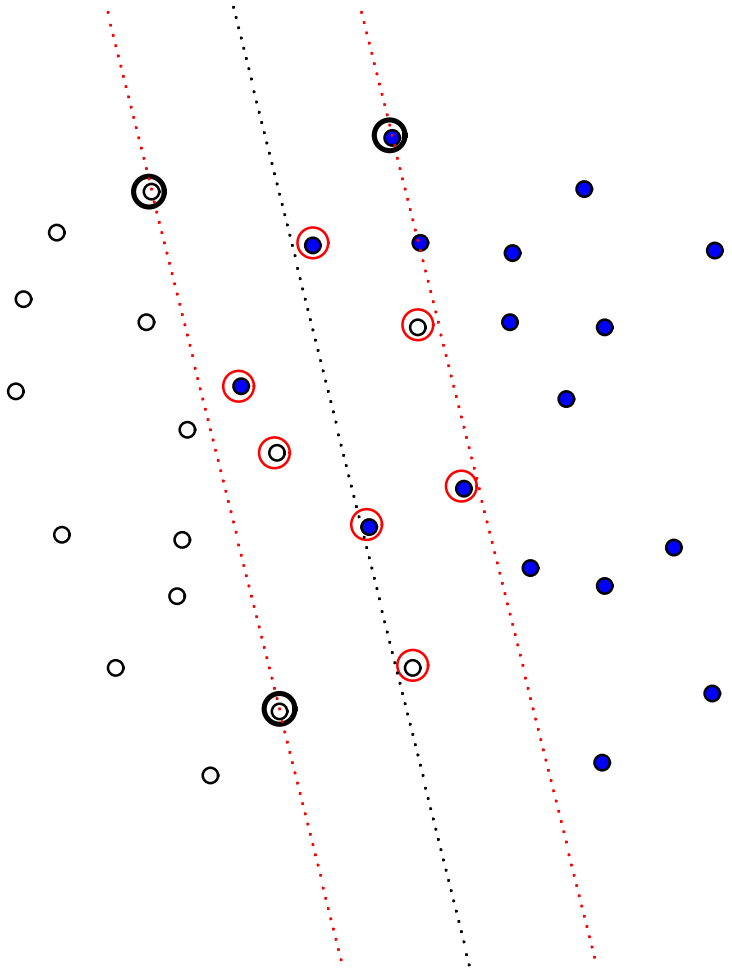


Figure 2.6: Nonlinearly separable data , C = 4.0

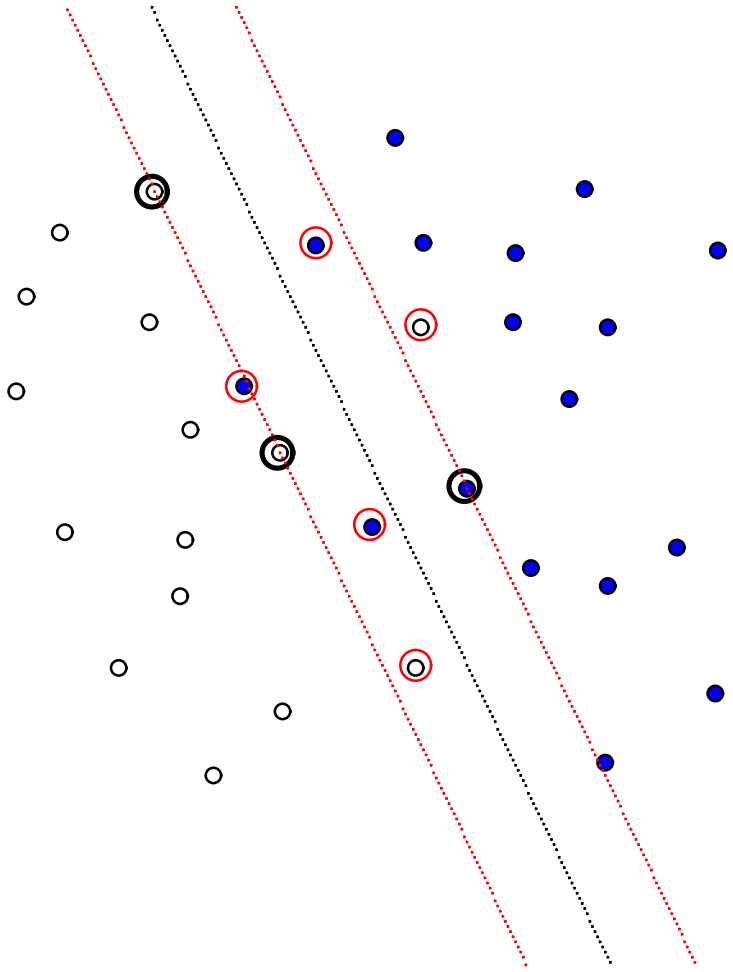


Figure 2.7: Nonlinearly separable data, C = 10.5

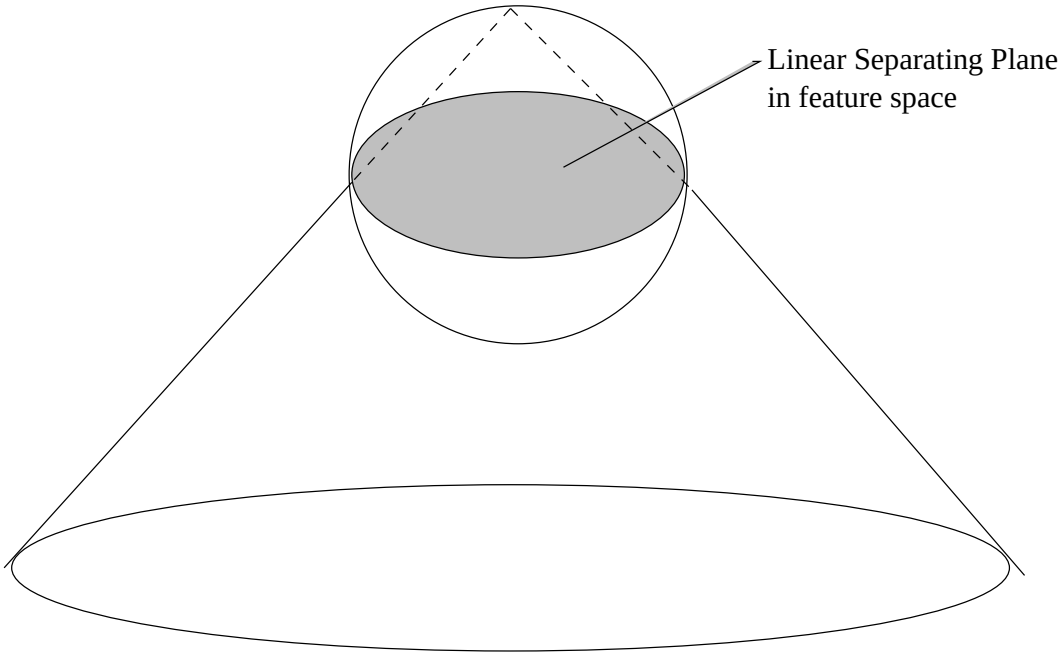


Figure 2.8: Linear separation in a high dimensional space

The following are two examples of kernel functions.
Example 1:

$$\mathbf{x} = (x_1, x_2)$$
$$\varphi(\mathbf{x}) = (1, \sqrt{2}x_1, \sqrt{2}x_2, x_1^2, x_2^2, \sqrt{x}_1 x_2)$$

$$\begin{aligned} \varphi(\mathbf{x}) \cdot \varphi(\mathbf{y}) &= 1 + 2x_1y_1 + 2x_2y_2 + 2x_1^2y_1^2 + 2x_1y_1x_2y_2 \\ &= (1 + x_1y_1 + x_2y_2)^2 \\ &= (1 + x.y)^2 \end{aligned}$$

$$K(\mathbf{x}, \mathbf{y}) = \varphi(\mathbf{x}).\varphi(\mathbf{y}) = (1 + \mathbf{x} \cdot \mathbf{y})^2$$

K is a finite polynomial kernel, which satisfies Mercer's Theorem. The separating surface in the data input space is a polynomial surface of degree 2.

Example 2:

$$K(\mathbf{x}, \mathbf{y}) = exp\left(\frac{-\|\mathbf{x} - \mathbf{y}\|^2}{2\sigma^2}\right), \qquad \sigma \in R.$$

The Gaussian kernel satisfies Mercer's Theorem and it is infinite.

Using a kernel function, equation 2.14 can be written as

$$W(\lambda) = \sum_{i=1}^{l} \lambda_i - \frac{1}{2} \sum_{i=1}^{l} \sum_{i=1}^{l} y_i y_j K(\mathbf{x_i} \cdot \mathbf{x_j}) \lambda_i \lambda_j.$$

Figure 2.9 and Figure 2.10 show the training results using a polynomial kernel of degree 2 and a C parameter of 15 and 30 respectively on nonlinearly separable data.

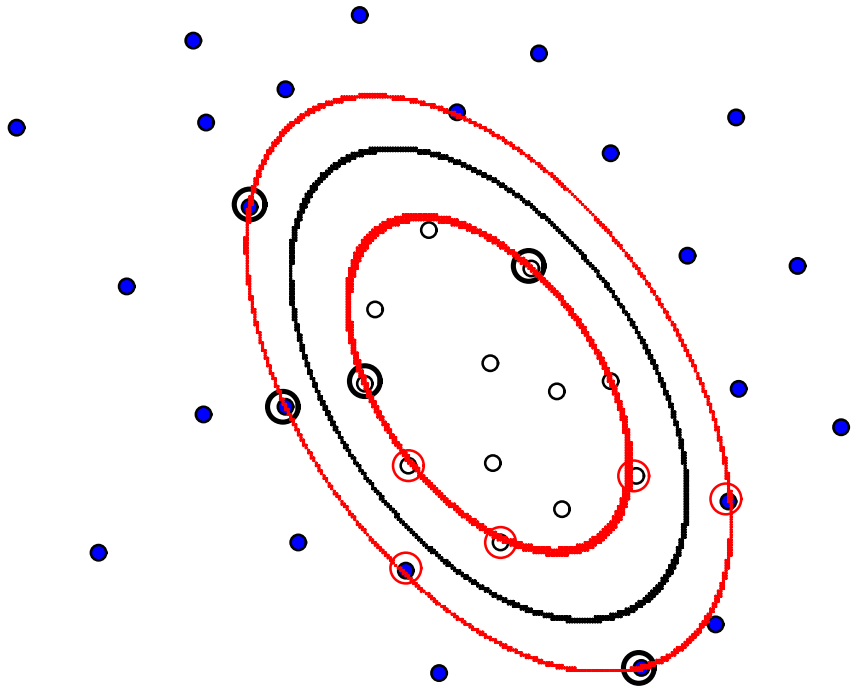


Figure 2.9: Nonlinearly separable data: polynomial kernel of degree 2, C = 15

Figure 2.11 and Figure 2.12 show the training results using a polynomial kernel of degree 2 and a C parameter of 25 and 50 respectively.
Figure 2.13 and Figure 2.14 show the training results using a radial basis function kernel with a C parameter of 2.5 and 6.5 and a variance of 1.
In all of the figures, it is noticed how the number of bound support vectors decreases as the separating margin narrows with the increase of C parameter. The orientation of the separating plane changes as well. All results are produced using the interactive Python tool **demo.py**.

2.4 Applications of SVM

Increasingly scientists and engineers are attracted to SVM to solve many real-life problems because of its sound theoretical foundation and proven state-of-the-art performance in many applications. In classification problems, the classifier only needs to calculate the inner product between two vectors of the training data. The generalization ability of SVM only depends on its VC-dimension and not on the dimension of the data. The following is a broad survey of applications of SVM in different fields.

When electrons and positrons collide, they generate a huge amount of energy and particles. Monte Carlo methods are used to simulate these collision events in a detector which measures the physical properties of the particles generated from the collision events. The goal of the research was to classify these events according to the quark-flavour they

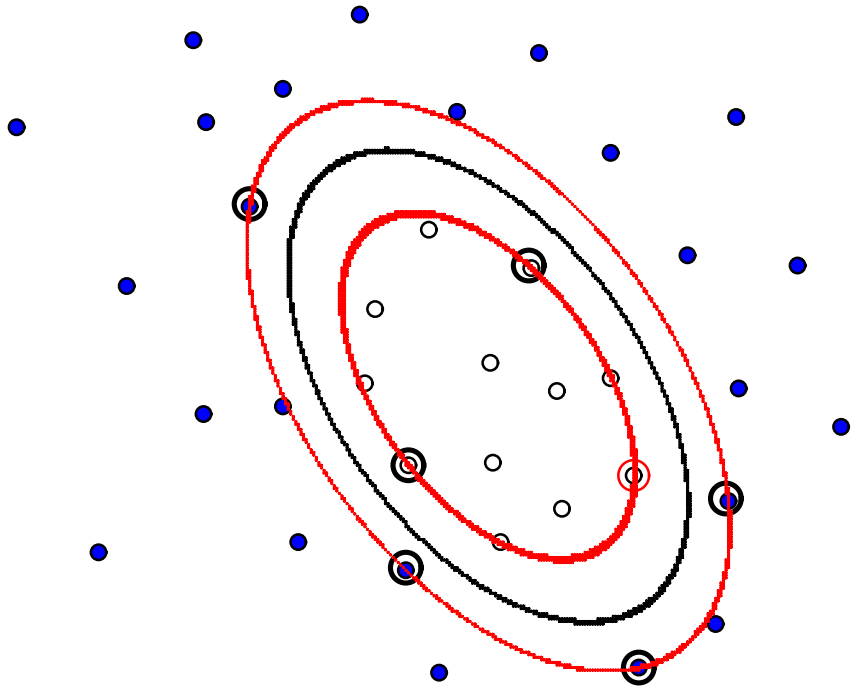


Figure 2.10: Nonlinearly separable data: polynomial kernel of degree 2, C = 30

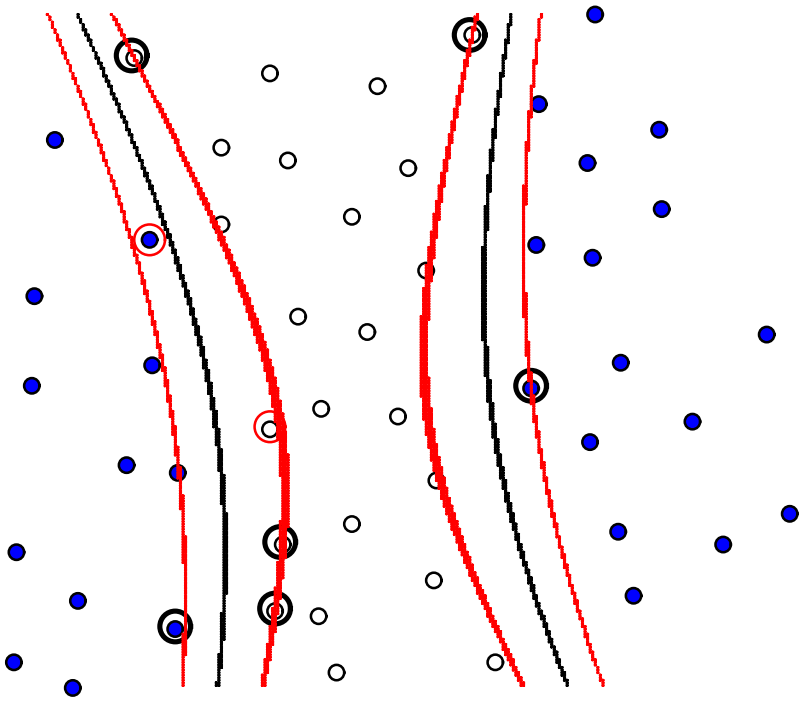


Figure 2.11: Nonlinearly separable data: polynomial kernel of degree 2, C = 25

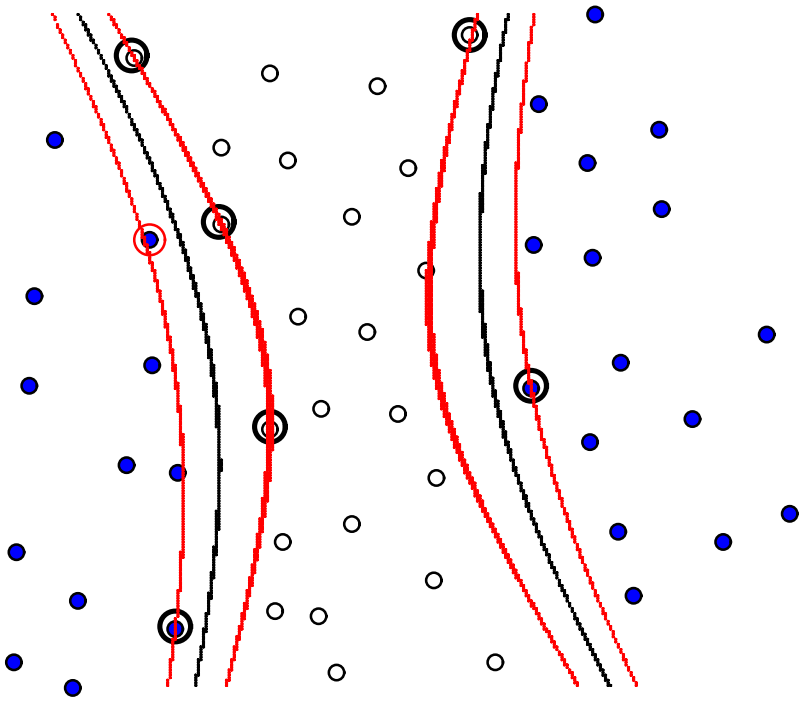


Figure 2.12: Nonlinearly separable data: polynomial kernel of degree 2, C = 50

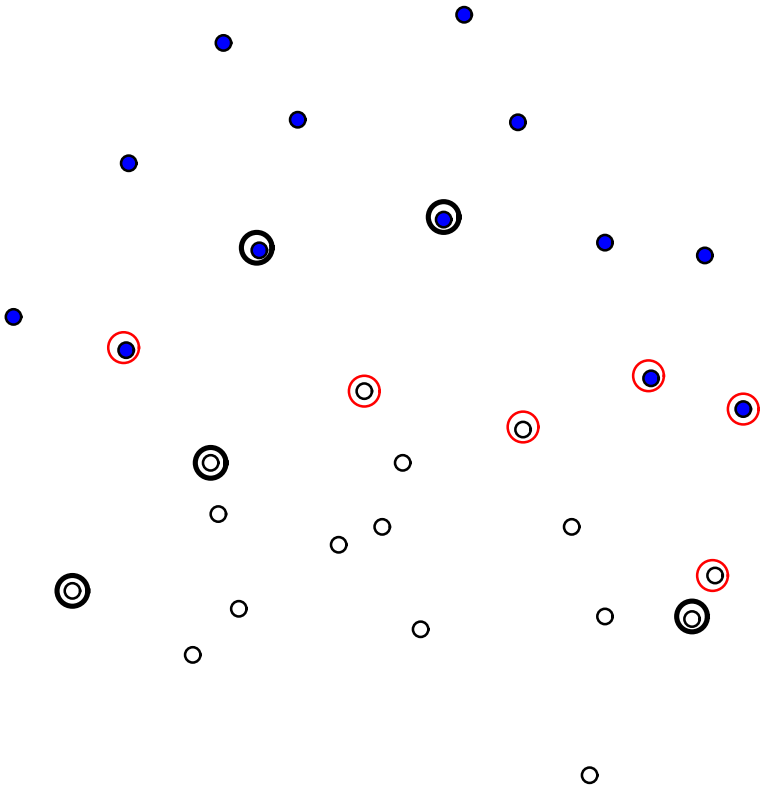


Figure 2.13: Nonlinearly separable data: radial basis function sigma2 = 1, C = 2.5

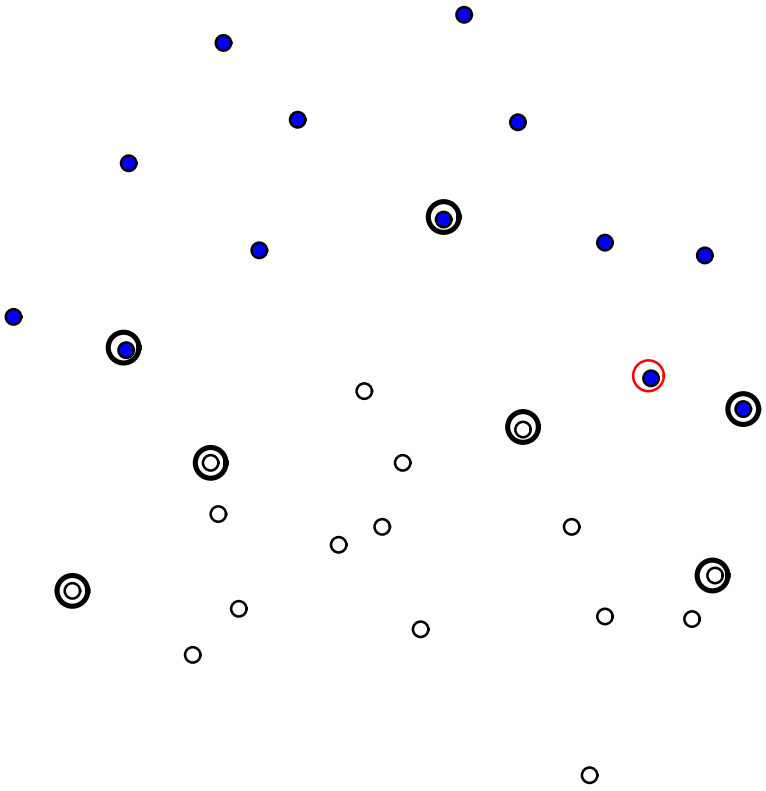


Figure 2.14: Nonlinearly separable data: radial basis function sigma2 = 1, C = 6.5

originate from and identify the particles in these events[6]. The quark-flavour problem involves patterns of 3x100k in size. SVMs(RBF kernel) were used in solving this classification problem.

Face detection as a computer vision task has many potential applications such as human-computer interfaces, surveillance systems, census systems and cancer growth detection. An arbitrary image (digitized video or scanned photograph) is used as an input to a detection system which determines if there is any human face in the image and returns an encoding of those faces' locations if the detection is positive. Face detection, like most object-detection problems which is to differentiate a certain class of objects from all other patterns and objects of other classes, is different from object recognition which is to differentiate between elements of the same class[6]. Object-detection is a hard problem because it is hard to parameterize analytically the significant pattern variations. In face-detection problems, the sources of the pattern variations are facial appearance, expression, presence or absence of common structural features, like glasses or a moustache, shadows, etc.[12]. In the face detection research done within the Center for Biological and Computational Learning at MIT, researchers apply SVMs to the classification step of the detection system and get results which are as well as other state-of-the-art systems. SVMs have also been applied to a great variety of 3-D object recognition problems and produce excellent recognition rates[15].

A research group in Germany applied SVMs to an engine knock detection system for combustion engine control[6]. A large database with different engine states (2000 and 4000 rounds per minute; non-knocking, borderline knocking and hard-knocking) were collected. The problem is highly non-linear. Different approaches were used including SVM, MLP nets and Adaboost. SVM outperformed all other approaches significantly.

SVMs have been applied to many biomedical researches. When extracting protein sequences from nucleotide sequences, the translation initiation sites (regions where the encoding proteins start) have to be recognized. SVMs are used to recognize these sites. SVMs perform better than neural network[21]. SVMs have also been applied to breast cancer diagnosis and prognosis problem The dataset used is the Wisconsin breast cancer dataset containing 699 patterns with 10 attributes for a binary classification of malignant or benign cancer. The system which uses Adatron SVM has 99.48% success rate, compared to 94.2% (CART), 95.9% (RBF), 96% (linear discriminant), 96.6% (Backpropagation network)[6]. Researchers at the University of California, Santa Cruz has used SVMs with a linear kernel, polynomial kernel and RBF kernel to functionally classify genes using gene expression data from DNA microarray hybridization experiments. SVMs outperformed all other classifiers when they are provided with a specifically designed kernel to deal with very imbalanced data[6].

SVM has been generalized to regression and been applied to the problems of estimating real-valued functions. The Boston housing problem predicts house prices from socio-economic and environmental factors, such as crime rate, nitric oxide concentration, distance to employment centers and age of a property[6]. SVMs outperform the baseline system (bagging) on the Boston housing problem[6][3]. Bagging is a technique of combining a multiple of classifiers (an ensemble), say decision trees and neural networks, and producing a single classifier. Generally, the resulting classifier is more accurate than any of the individual classifier in the ensemble[11].

Closely related to regression problems are time series prediction and dynamic reconstruction of a chaotic system. The essence of the problem is to reconstruct the dynamics of an unknown system, given a noisy time-series representing the evolution of one variable of the system with time. The rebuilt model should be as close as possible to the original system in terms of its invariants. Researchers report excellent performance result with SVM and its effectiveness with this type of problems[6][9].

The continuing growth of the internet increases the demand of text categorization tools to help us manage all the electronic text information available. We need fast and efficient search engines, text filters to remove junk mail, information routing/pushing, classifiers for saving files, emails and URLs. Text categorization is a task of assigning natural language texts to one or more predefined categories based on their contents. The task has certain characteristics which make SVMs especially suitable for it. Text categorization has to deal with data with many features (usualy more than

10,000) because each stemmed word[4] is a feature. The document vector of a text document is sparse, i.e. it contains few non-zero entries. A text document has 'dense' concept-very few irrelevant features. Most text categorization problems are linearly separable. Experiments of researchers[4][7] affirm the suitability of SVMs to text classification tasks. They are robust and consistently give good performance; they outperform existing methods substantially and significantly[7].

[4]The word stem is derived from the occurance form of a word by removing case and flection information[7]. Hence "computes", "computing" and "computer" are all mapped to the same stem "comput".

The mere formulation of a problem is far more often essential than its solution, which may be merely a matter of mathematical or experiement skill

— Albert Einstein, 1879-1955 —

3

Implementation of SVM Using Sequential Minimal Optimization

Solving a quadratic programming problem is slow and requires a lot of memory as well as in depth knowledge of numerical analysis. Sequential Minimal Optimization does away with the need for quadratic programming.

3.1 How SMO works

The QP problem for training an SVM is[1]:
Maximize

$$W(\lambda) = \sum_{i=1}^{l} \lambda_i - \frac{1}{2} \sum_{i=1}^{l} \sum_{i=1}^{l} y_i y_j K(\mathbf{x_i} \cdot \mathbf{x_j}) \lambda_i \lambda_j \tag{3.1}$$

subject to :

$$0 \leq \lambda_i \leq C\,, \qquad i = 1, \ldots, l, \tag{3.2}$$

$$\sum_{i=1}^{l} y_i \lambda_i = 0. \tag{3.3}$$

As shown in chapter 2, the SVM overview section, the KKT conditions which are the necessary and sufficient conditions for a point to be optimal in the following equation

$$L_p = \frac{1}{2}\|\mathbf{w}\|^2 + C\sum_{i=1}^{l} \xi_i - \sum_{i=1}^{l} \lambda_i (y_i(\mathbf{x_i} \cdot \mathbf{w} - b) - 1 + \xi) - \sum_{i=1}^{l} \mu_i \xi_i$$

are:

$$\lambda_i (y_i(\mathbf{w} \cdot \mathbf{x_i} - b) - 1 + \xi_i) = 0 \qquad \text{and} \qquad (C - \lambda_i)\xi_i = 0$$

They can be rewritten as:

$$\lambda_i (y_i f(\mathbf{x_i}) - 1 + \xi_i) = 0 \qquad \text{and } (C - \lambda_i)\xi_i = 0$$

where $f(\mathbf{x_i}) = (\mathbf{w} \cdot \mathbf{x_i} - b)$.
When $\lambda_i = 0$, then $\xi_i = 0$, and $y_i f(\mathbf{x_i}) \geq 1$
When $0 < \lambda_i < C$, then $\xi_i = 0$, and $y_i f(\mathbf{x_i}) - 1 = 0$

[1] This chapter is based on J. Platt's paper[13]. We will use most of the notations used by Platt.

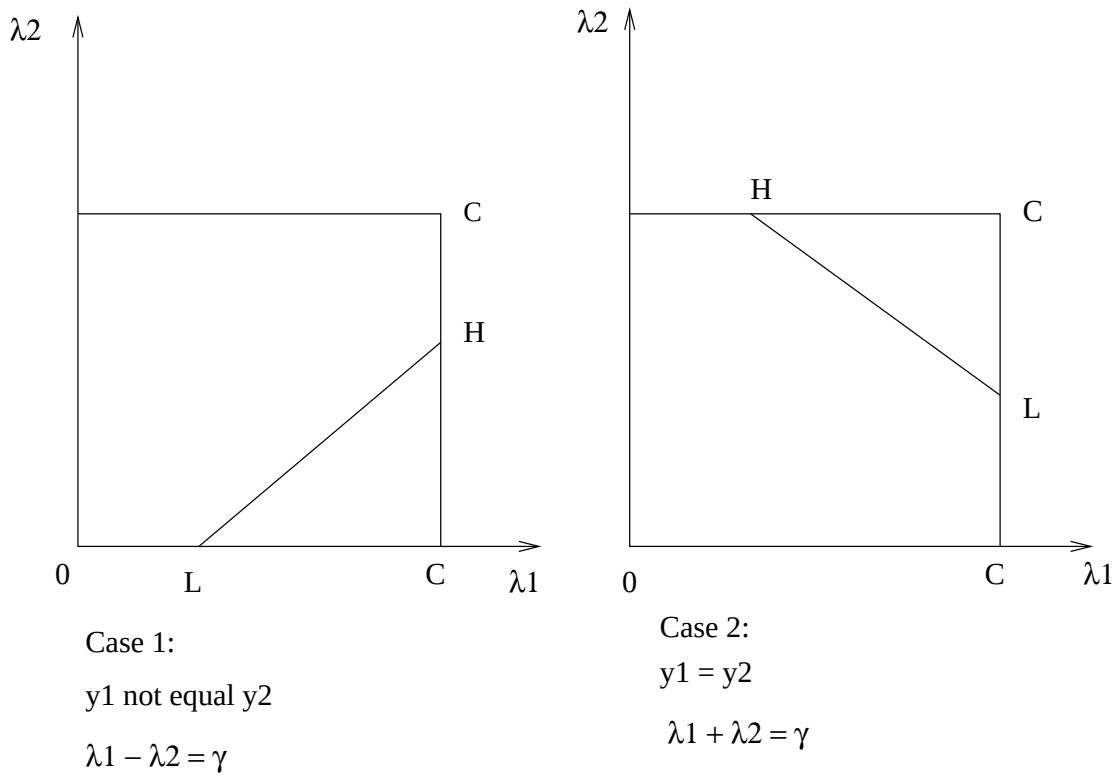


Figure 3.1: Two cases of optimizations

When $\lambda_i = C$, then $\xi_i \neq 0$, and $y_i f(\mathbf{x_i}) \leq 1$

These KKT conditions can be evaluated one example at a time and when all the λ's obey the KKT condition, then W reaches its maximum.

SMO solves the optimization problem in the SVM technique without any extra matrix storage by decomposing the problem into small subproblems. The smallest subproblem in this case is one which optimizes two Lagrange multipliers because of the equality constraint.

$$\sum_{i=1}^{l} \lambda_i y_i = 0 \ .$$

At each step, SMO picks two Lagrange multipliers to optimize jointly and update the SVM at the end of each optimization. For such a small problem, analytical method can be used instead of QP to solve for the two λ's.

3.1.1 How to solve for the two λ's analytically

The constraint of each λ is

$$0 \leq \lambda \leq C$$

Hence the two λ's lie within a boxed area. See Figure 3.1

The two λ's must fulfil the equality constraint $\sum_{i=1}^{l} \lambda_i y_i = 0$. Call these two λ's λ_1 and λ_2 then

$$\lambda_1 y_1 + \lambda_2 y_2 + \sum_{i=3}^{l} \lambda_i y_i = 0$$

or

$$\lambda_1 y_1 + \lambda_2 y_2 = \gamma \qquad \text{where } \gamma = -\sum_{i=3}^{1} \lambda_i y_i$$

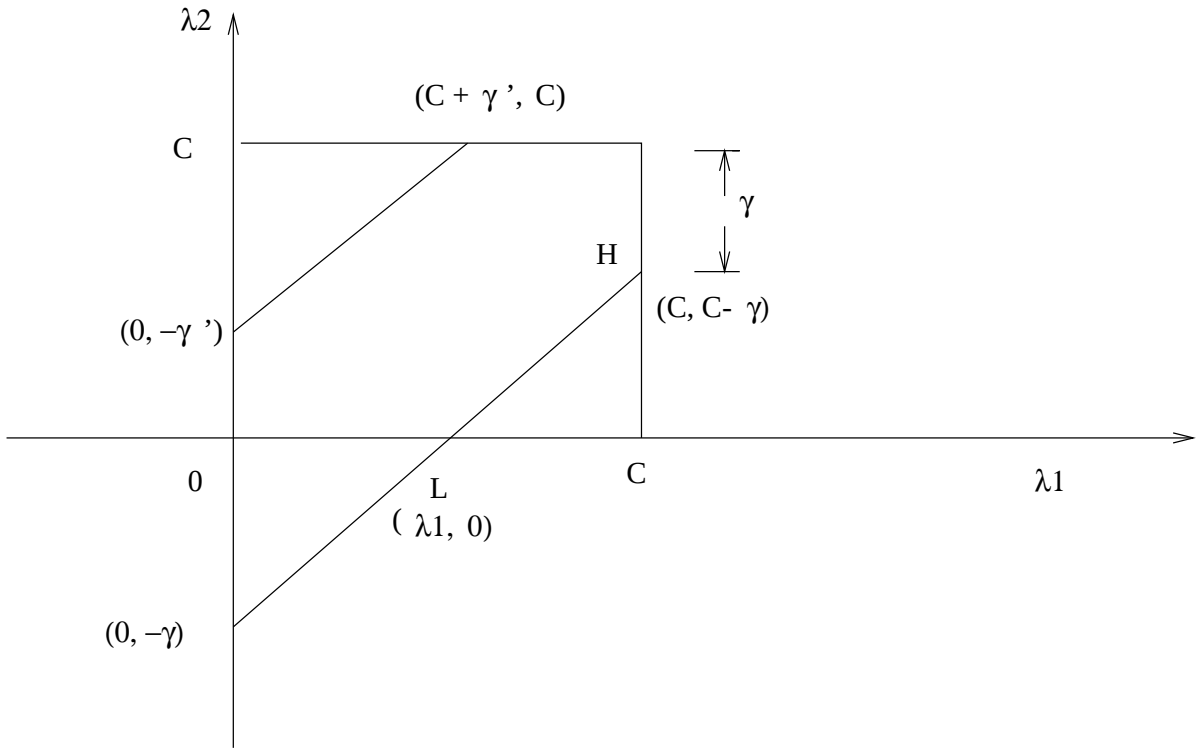


Figure 3.2: Case 1

There are two cases to consider (remember, $y_i \in \{1,-1\}$):
Case 1: $y_1 \neq y_2$ then

$$\lambda_1 - \lambda_2 = \gamma. \tag{3.4}$$

Case 2: $y_1 = y_2$ then

$$\lambda_1 + \lambda_2 = \gamma. \tag{3.5}$$

Let $s = y_1 y_2$, then equations 3.4 and 3.5 can be written as:

$$\lambda_1 + s\lambda_2 = \gamma \tag{3.6}$$

and $\gamma = \lambda_1^{old} + s\lambda_2^{old}$ before optimization.

The bound constraint equation 3.2 requires that the two Lagrange multipliers lie within the box and the equality constraint equation 3.3 requires that they lie on a diagonal line. The end points of the diagonal line can be expressed as follows:

Case 1: $y_1 \neq y_2$: (See Figure 3.2)

$$\lambda_1^{old} - \lambda_2^{old} = \gamma.$$

L (λ_2 at the lower end point) is:

$$max(0,-\gamma) = max(0,\lambda_2^{old} - \lambda_1^{old}). \tag{3.7}$$

H (λ_2 at the higher end point) is:

$$min(C,C-\gamma) = min(C,C+\lambda_2^{old} - \lambda_1^{old}). \tag{3.8}$$

Equation 3.7 and 3.8 are used in the code implementation.

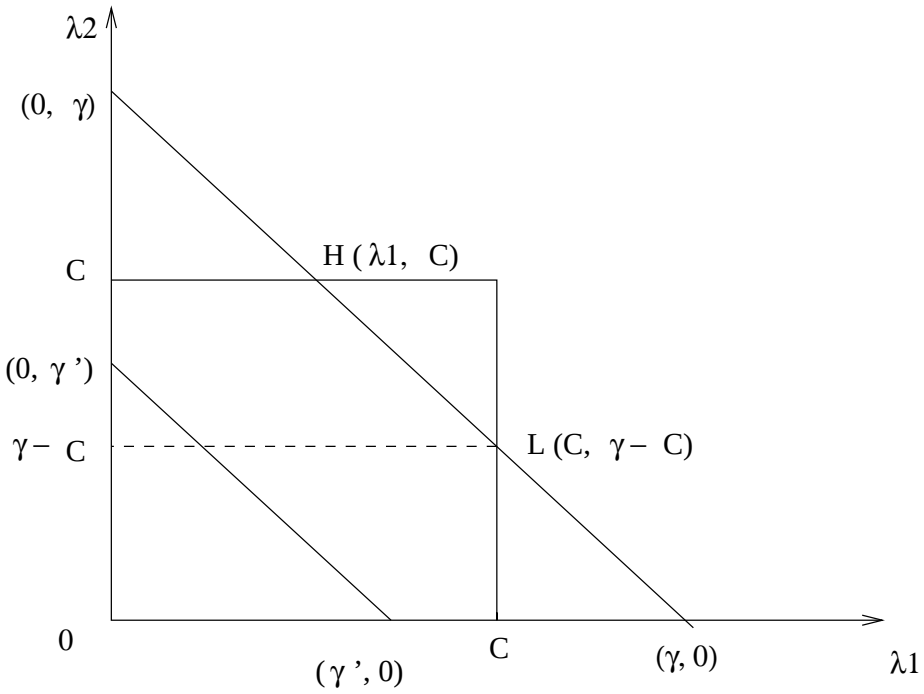


Figure 3.3: Case 2

Case 2: $y_1 = y_2$: (See Figure 3.3)

$$\lambda_1^{old} + \lambda_2^{old} = \gamma$$

L (λ_2 at the lower end point) is:

$$max(0, \gamma - C) = max(0, \lambda_1^{old} + \lambda_2^{old} - C). \tag{3.9}$$

H (λ_2 at the higher end point) is:

$$min(C, \gamma) = min(C, \lambda_1^{old} + \lambda_2^{old}). \tag{3.10}$$

Equation 3.9 and 3.10 are used in the code implementation.

Equation 3.1 for $W(\lambda)$ can be written as:

$$W = \sum_{i=1}^{l} \lambda_i - \frac{1}{2} \sum_{i=1}^{l} \sum_{j=1}^{l} y_i y_j k_{ij} \lambda_i \lambda_j$$

where $k_{ij} = K(\mathbf{x_i}, \mathbf{x_j})$ and $\lambda = (\lambda_1, \lambda_2, \ldots, \lambda_l)$.

Putting W as a function of λ_1, λ_2 by setting $s = y_1 y_2$ and $v_i = \sum_{j=3}^{l} y_j \lambda_j k_{ij}$ we get

$$\begin{aligned} W(\lambda_1, \lambda_2) \quad = \quad & \lambda_1 + \lambda_2 + \sum_{i=3}^{l} \lambda_i - \frac{1}{2} \sum_{i=1}^{2} \sum_{j=1}^{2} y_i y_j k_{ij} \lambda_i \lambda_j \\ & - \frac{1}{2} \sum_{i=1}^{2} \sum_{j=3}^{l} y_i y_j k_{ij} \lambda_i \lambda_j - \frac{1}{2} \sum_{i=3}^{l} \sum_{j=1}^{2} y_i y_j k_{ij} \lambda_i \lambda_j \\ & - \frac{1}{2} \sum_{i=3}^{l} \sum_{j=3}^{l} y_i y_j k_{ij} \lambda_i \lambda_j \end{aligned}$$

$$
\begin{aligned}
&= \lambda_1+\lambda_2+\sum_{i=3}^{l}\lambda_i-\frac{1}{2}k_{11}\lambda_1^2-\frac{1}{2}k_{22}\lambda_2^2-sk_{12}\lambda_1\lambda_2 \\
&\quad -\frac{1}{2}\sum_{j=3}^{l}y_1y_jk_{1j}\lambda_1\lambda_j-\frac{1}{2}\sum_{j=3}^{l}y_2y_jk_{2j}\lambda_2\lambda_j \\
&\quad -\frac{1}{2}\sum_{i=3}^{l}y_iy_1k_{i1}\lambda_i\lambda_1-\frac{1}{2}\sum_{i=3}^{l}y_iy_2k_{i2}\lambda_i\lambda_2 \\
&\quad -\frac{1}{2}\sum_{i=3}^{l}\sum_{j=3}^{l}y_iy_jk_{ij}\lambda_i\lambda_j \\
&= \lambda_1+\lambda_2+\sum_{i=3}^{l}\lambda_i-\frac{1}{2}k_{11}\lambda_1^2-\frac{1}{2}k_{22}\lambda_2^2-sk_{12}\lambda_1\lambda_2-y_1\lambda_1v_1-y_2\lambda_2v_2 \\
&\quad -\frac{1}{2}\sum_{i=3}^{l}\sum_{j=3}^{l}y_iy_jk_{ij}\lambda_i\lambda_j \\
&= \lambda_1+\lambda_2-\frac{1}{2}k_{11}\lambda_1^2-\frac{1}{2}k_{22}\lambda_2^2-sk_{12}\lambda_1\lambda_2-y_1\lambda_1v_1-y_2\lambda_2v_2 \\
&\quad +W_{constant}
\end{aligned}
\tag{3.11}
$$

where $W_{constant}=\sum_{i=3}^{l}\lambda_i-\frac{1}{2}\sum_{i=3}^{l}\sum_{j=3}^{l}y_iy_jk_{ij}\lambda_i\lambda_j$.

Since $\lambda_1=\gamma-s\lambda_2$, we can rewrite W as a function of λ_2 by substituting $\gamma-s\lambda_2$ into equation 3.11. Therefore

$$
\begin{aligned}
W(\lambda_2) &= \gamma-s\lambda_2+\lambda_2-\frac{1}{2}k_{11}(\gamma-s\lambda_2)^2-\frac{1}{2}k_{22}\lambda_2^2-sk_{12}(\gamma-s\lambda_2)\lambda_2 \\
&\quad -y_1v_1(\gamma-s\lambda_2)-y_2v_2\lambda_2+W_{constant} \\
&= \gamma-s\lambda_2+\lambda_2-\frac{1}{2}k_{11}\gamma^2+k_{11}s\gamma\lambda_2-\frac{1}{2}k_{11}\lambda_2^2-\frac{1}{2}k_{22}\lambda_2^2 \\
&\quad -sk_{12}\gamma\lambda_2+k_{12}\lambda_2^2-y_1v_1\gamma+y_1v_1s\lambda_2 \\
&\quad -y_2v_2\lambda_2+W_{constant}
\end{aligned}
\tag{3.12}
$$

The first derivative of W w.r.t. λ_2 is

$$
\begin{aligned}
\frac{\partial W}{\partial\lambda_2} &= -s+1+s\gamma k_{11}-k_{11}\lambda_2-k_{22}\lambda_2-s\gamma k_{12}+2k_{12}\lambda_2+sy_1v_1-y_2v_2 \\
&= -s+1+sk_{11}(\gamma-s\lambda_2)-k_{22}\lambda_2+k_{12}\lambda_2-sk_{12}(\gamma-s\lambda_2)+y_2(v_1-v_2)
\end{aligned}
\tag{3.13}
$$

where $sy_1=y_1y_2y_1=y_1^2y_2=y_2$.

The second derivative of W w.r.t. λ_2 is :

$$
\frac{\partial^2W}{\partial\lambda_2^2}=2k_{12}-k_{11}-k_{22}=\eta \tag{3.14}
$$

Equation 3.14 is used in the code implementation.

Now we are ready to calculate the constrained maximum of the objective function W (equation 3.1) while allowing only two Lagrange multipliers to change. We first define E_i as the error on the ith training example

$$E_i = u_i - y_i,$$

where u_i is the output of the SVM with input $\mathbf{x_i}$ and y_i is the class label of $\mathbf{x_i}$.
From equation 2.27 we get

$$\begin{aligned} u_1 &= \sum_{j=1}^{l} y_j \lambda_j k_{1j} - b \\ &= \sum_{j=3}^{l} y_j \lambda_j k_{1j} + y_1 \lambda_1 k_{11} + y_2 \lambda_2 k_{12} - b \\ &= v_1 - b + y_1 \lambda_1 k_{11} + y_2 \lambda_2 k_{12} \end{aligned}$$

or

$$v_1 = u_1 + b - y_2 \lambda_2 k_{12} - s y_2 \lambda_1 k_{11} \tag{3.15}$$

Similarly,

$$\begin{aligned} u_2 &= \sum_{j=1}^{l} y_j \lambda_j k_{2j} - b \\ &= \sum_{j=3}^{l} y_j \lambda_j k_{j2} + y_1 \lambda_1 k_{12} + y_2 \lambda_2 k_{22} - b \\ &= v_2 - b + y_1 \lambda_1 k_{12} + y_2 \lambda_2 k_{22} \end{aligned}$$

or

$$v_2 = u_2 + b - y_1 \lambda_1 k_{12} - y_2 \lambda_2 k_{22} \tag{3.16}$$

At the maximal point of W, equation 3.13 $\frac{\partial W}{\partial \lambda_2}$ is zero. Hence

$$\lambda_2 (k_{11} + k_{22} - k_{12}) = s(k_{11} - k_{12})\gamma + y_2(v_1 - v_2) + 1 - s \tag{3.17}$$

Substituting equations 3.15 and 3.16 into equation 3.17, we get

$$\begin{aligned} (-\eta)\lambda_2^{new} &= s\gamma(k_{11} - k_{12}) + y_2(u_1 + b - u_2 - b) - y_2^2 \lambda_2^{old} k_{12} \\ &\quad - s\lambda_1^{old} k_{11} + s\lambda_1^{old} k_{12} + \lambda_2^{old} k_{22} + y_2^2 - y_1 y_2 \\ &= s(\lambda_1^{old} + s\lambda_2^{old})(k_{11} - k_{12}) - \lambda_2^{old} k_{12} + s\lambda_1^{old} k_{12} - s\lambda_1^{old} k_{11} \\ &\quad + \lambda_2^{old} k_{22} + y_2(u_1 - u_2 + y_2 - y_1) \\ &= s\lambda_1^{old} k_{11} - s\lambda_1^{old} k_{12} + \lambda_2^{old} k_{11} - \lambda_2^{old} k_{12} - \lambda_2^{old} k_{12} \\ &\quad + s\lambda_1^{old} k_{12} - s\lambda_1^{old} k_{11} + \lambda_2^{old} k_{22} + y_2(u_1 - y_1 - u_2 + y_2) \\ &= \lambda_2^{old}(-2k_{12} + k_{11} + k_{22}) + y_2(E_1 - E_2) \\ &= (-\eta)\lambda_2^{old} + y_2(E_1 - E_2). \end{aligned}$$

$$\lambda_2^{new} = \lambda_2^{old} - \frac{y_2(E_1 - E_2)}{\eta}. \tag{3.18}$$

Equation 3.18 is used in the code implemenation.
Once we find λ_2^{new} which is at the unconstrained maximum, we have to constrain it to within the ends of the diagonal line segment. See Figure 3.1

$$\lambda_2^{new,clipped} = \begin{pmatrix} H & \text{if } \lambda_2^{new} \geq H \\ \lambda_2^{new} & \text{if } L < \lambda_2^{new} < H \\ L & \text{if } \lambda_2^{new} \leq L \end{pmatrix}. \tag{3.19}$$

Equation 3.19[13] is used in the code implementation.
To compute the constrained λ_1 we make use of the equation:

$$\lambda_1^{new} + s\lambda_2^{new,clipped} = \lambda_1^{old} + s\lambda_2^{old},$$

which gives us

$$\lambda_1^{new} = \lambda_1^{old} + s(\lambda_2^{old} - \lambda_2^{new,clipped}). \tag{3.20}$$

Equation 3.20[13] is used in the code implementation.
When we evaluate η, the second derivative of W, using equation 3.14, we might find that it is evaluated to zero when there is more than one example in our training example having the same input vector $\mathbf{x}$. In this case we will find another example and re-optimize the first multiplier with a new λ_2. Under some unusual circumstances, η is not negative. Then, SMO evaluates the objective function at each end of the line segment (see Figure 3.2 and Figure 3.3) and uses the Lagrange multipliers at the end point, which yields the highest value of the objective function as the new λ's. This is how we can evaluate the objective functions at the end points.

Let L_1 be the value of λ_1 at the lower end point of the line segment.
Let H_1 be the value of λ_1 at the higher end point of the line segment.
Let L_2 be the value of λ_2 at the lower end point of the line segment.
Let H_2 be the value of λ_2 at the higher end point of the line segment.
Both L_2 and H_2 can be found from the equations pair 3.7, 3.8 or 3.9, 3.10 depending on the value of y_1 and y_2.
Then, by using equation 3.20, we have

$$L_1 = \lambda_1 + s(\lambda_2 - L_2), \tag{3.21}$$

$$H_1 = \lambda_1 + s(\lambda_2 - H_2). \tag{3.22}$$

Equations 3.21 and 3.22 are used in the code implementation.

$$u_1 = y_1\lambda_1 k_{11} + y_2\lambda_2 k_{12} + \sum_{j=3}^{l} \lambda_j y_j k_{1j} - b.$$

$$u_1 - y_1 = E_1.$$

Hence

$$\begin{aligned} y_1 &= u_1 - E_1 \\ &= y_1\lambda_1 k_{11} + y_2\lambda_2 k_{12} + \sum_{j=3}^{l} \lambda_j y_j k_{1j} - b - E_1. \end{aligned}$$

After multipling both sides by y_1 and using $s = y_1 y_2$ we have

$$1 = \lambda_1 k_{11} + y_1 y_2 \lambda_2 k_{12} + y_1 \sum_{j=3}^{l} \lambda_j y_j k_{1j} - y_1 b - y_1 E_1.$$

Rearranging the terms, we have

$$y_1 \sum_{j=3}^{l} \lambda_j y_j k_{1j} - 1 = y_1(E_1 + b) - \lambda_1 k_{11} - s\lambda_2 k_{12},$$

or

$$y_1 v_1 - 1 = y_1(E_1 + b) - \lambda_1 k_{11} - s\lambda_2 k_{12}. \tag{3.23}$$

Similarly we get:

$$y_2 \sum_{j=3}^{l} \lambda_j y_j k_{2j} - 1 = y_2(E_2 + b) - \lambda_2 k_{22} - s\lambda_1 k_{12},$$

or

$$y_2 v_2 - 1 = y_2(E_2 + b) - \lambda_2 k_{22} - s\lambda_1 k_{12}. \tag{3.24}$$

Let us define

$$f_1 = y_1 \sum_{j=3}^{l} \lambda_j y_j k_{1j} - 1 = y_1 v_1 - 1 \tag{3.25}$$

and

$$f_2 = y_2 \sum_{j=3}^{l} \lambda_j y_j k_{2j} - 1 = y_2 v_2 - 1. \tag{3.26}$$

Substituting equations 3.23, 3.24 into 3.27 and 3.28 respectively we get

$$f_1 = y_1(E_1 + b) - \lambda_1 k_{11} - s\lambda_2 k_{12} \tag{3.27}$$

and

$$f_2 = y_2(E_2 + b) - \lambda_2 k_{22} - s\lambda_1 k_{12}. \tag{3.28}$$

When we evaluate the objective function $W(\lambda_1, \lambda_2)$ at each end point of the line segment for finding the highest value, we only need to evaluate those terms involving the variable λ_1 and λ_2 in equation 3.11 because the rest of the terms do not change during optimization. Let us call this function which involves only the terms λ_1 and λ_2, $w(\lambda_1, \lambda_2)$.
We are interested in the change in the objective function (equation 3.11) at the end points(L_1, L_2)

$w(L_1, L_2) = L_1 + L_2 - \frac{1}{2}L_1^2 k_{11} - \frac{1}{2}L_2^2 k_{22} - sL_1 L_2 k_{12} - L_1 y_1 v_1 - L_2 y_2 v_2.$

Since $y_1 v_1 = f_1 + 1$ (equation 3.25) and $y_2 v_2 = f_2 + 1$ (equation 3.26), we can rewrite $w(L_1, L_2)$ at the lower end as:

$$\begin{aligned} w(L_1, L_2) &= L_1 + L_2 - \frac{1}{2}L_1^2 k_{11} - \frac{1}{2}L_2^2 k_{22} - sL_1 L_2 k_{12} \\ &\quad -L_1(f_1 + 1) - L_2(f_2 + 1) \\ &= -\frac{1}{2}L_1^2 k_{11} - \frac{1}{2}L_2^2 k_{22} - sL_1 L_2 k_{12} - L_1 f_1 - L_2 f_2. \end{aligned} \tag{3.29}$$

Similarly, we obtain $w(H_1, H_2)$ at the higher end point as:

$$w(H_1, H_2) = -\frac{1}{2}H_1^2 k_{11} - \frac{1}{2}H_2^2 k_{22} - sH_1 H_2 k_{12} - H_1 f_1 - H_2 f_2. \tag{3.30}$$

Equation 3.29 and 3.30 are used in the code implementation.
If $w(L1, L2)$ and $w(H_1, H_2)$ are the same within a small round-off error of ε, then the joint optimization makes no progress. We will skip these two λ's and find another pair of multipliers to optimize.

3.2 Choice of Lagrange multipliers for optimization

The SMO algorithm is based on the evaluation of the KKT conditions. When every multiplier fulfils the KKT conditions of the problem, the algorithm terminates. The KKT conditions are verified to within ε. Platt mentions in his paper that ε is typically in the range of 10^{-2} to 10^{-3}. Two heuristics are used to choose the multipliers for optimization.

The outer loop of the algorithm first iterates over the entire set of training examples deciding whether an example violates the KKT condition. If it does, then that example is chosen for optimization immediately. A second example is found using the second choice heuristic and then the two multipliers are jointly optimized. At the end of the optimization, the SVM is updated and the algorithm resumes iterating over the training examples looking for KKT violator. After one pass through the training set, the outer loop only iterates over those examples whose Lagrange multipliers are non-bound, i.e. they are within the open interval $(0,C)$. The KKT condition is checked. If the multiplier violates the KKT conditioon, then it is eligible for joint optimization. The outer loop repeats checking violation of the KKT condition over the non-bound examples until they all obey the KKT condition within ε. Then the outer loop goes over the entire training set again. It will alternate between the entire training set and the non-bound set until all the λ's obey the KKT conditions within ε. This is the first choice heuristic for choosing the first multiplier.

The SMO algorithm uses another heuristic to choose the second multiplier λ_2. The heuristic is based on maximizing the step that can be taken during joint optimization. Equation 3.14 is used at this step. We want to choose the maximum possible step size by having the biggest value of $\|E_1 - E_2\|$ in equation 3.18. A cached error value E is kept for every non-bound training example from which example can be chosen for maximizing the step size. If E_1 is positive, then the example with the minimum error E_2 is chosen. If E_1 is negative, then the example with the largest error E_2 is chosen. There are cases when there is no positive progress; for instance when both input vectors are identical. This can be avoided by not choosing the example with its error E_2 equals E_1. SMO uses a hierarchy of choices in choosing the second multiplier. If there is no positive progress, the algorithm will iterate through the non-bound example starting at a random position. If none of the non-bound example make positive progress, then the algorithm starts at a random position in the entire training set and iterates through the entire set in finding the λ_2 that will make positive progress in joint optimization. The randomness in choosing the starting position is to avoid bias towards examples stored at the beginning of the training set. In very extreme degenerative cases, a second multiplier which can make positive progress cannot be found. In this case, we will skip the first multiplier we found and start with another multiplier. I modified the hierarchy of the second choice heuristic in my implementation for efficiency. This modification will be described in section 4.

3.3 Updating the threshold b

Since E, the error on an example *i*, is the output of the SVM on example *i* - target of example *i*, we need to know the value of *b* for updating the value of the error cache at the end of optimization. Therefore, after each optimization, we will re-evaluate *b*.
Let u_1 = output of the SVM with the old λ_1 and λ_2

$$u_1 = \lambda_1^{old} y_1^{old} k_{11} + \lambda_2^{old} y_2 k_{12} + \sum_{j=3}^{l} \lambda_j y_j k_{1j} - b^{old} \ . \tag{3.31}$$

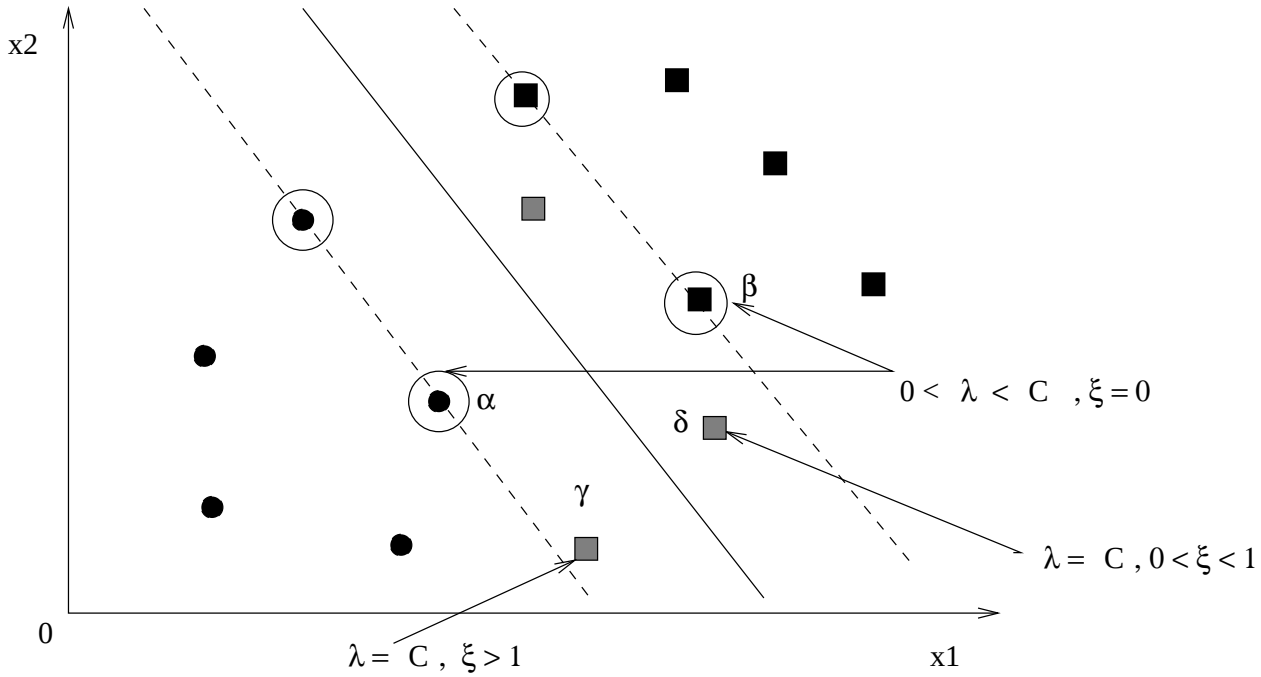
$$u_1 - E_1 = y_1 \ . \tag{3.32}$$

If the new λ_1 is not at the bounds, then the output of the SVM after optimization on example 1 will be y_1, its label value. See Figure 3.4.
Therefore

$$y_1 = \lambda_1^{new} y_1 k_{11} + \lambda_2^{new,clipped} y_2 k_{12} + \sum_{j=3}^{l} \lambda_j y_j k_{1j} - b_1. \tag{3.33}$$

Substituting equation 3.31, 3.33 into equation 3.32, we get

$$b_1 = E_1 + b^{old} + y_1(\lambda_1^{new} - \lambda_1^{old})k_{11} + y_2(\lambda_2^{new,clipped} - \lambda_2^{old})k_{12}. \tag{3.34}$$



The support vectors α and β give the same threshold *b*, the distance the optimal separating hyperplane is from the origin. Point γ and δ give threshold *b1* and *b2* respectively. They are error points. *b* is somewhere between *b1* and *b2*.

Figure 3.4: Threshold b when both λ's are bound

Similarly we can obtain the equation for b_2 such that the output of the SVM after optimization is y_2 when λ_2 is not at bounds.

$$b_2 = E_2 + b^{old} + y_1(\lambda_1^{new} - \lambda_1^{old})k_{12} + y_2(\lambda_2^{new,clipped} - \lambda_2^{old})k_{22}. \tag{3.35}$$

When both b_1 and b_2 are equal, they are valid[13]. See Figure 3.4.

When both new multipliers are at the bounds and if L is not equal to H, then the threshold values that are within the closed interval $[b_1, b_2]$ are all consistent with the KKT conditions[13]. In this case we use $b^{new} = \frac{b_1+b_2}{2}$ as the new threshold (see Figure 3.4). If one multiplier is at bound and the other is not, then the b value calculated using the non-bound multiplier is used as the new updated threshold.
Equations 3.34 and 3.35 are used in the code implementation for updating the threshold b after optimization.

3.4 Updating the error cache

When a Lagrange multiplier is non-bound after being optimized, its cached error is zero. The stored errors of other non-bound multipliers not involved in joint optimization are updated as follows.

$$E_k^{new} - E_k^{old} = u_k^{new} - u_k^{old}. \tag{3.36}$$

$$E_k^{new} = E_k^{old} + u_k^{new} - u_k^{old}. \tag{3.37}$$

For any *k*th example in the training set, the difference between its new SVM output value and its old SVM output

value, $u_k^{new} - u_k^{old}$ is due to the change in λ_1, λ_2 and the change in the threshold b.

$$u_k^{new} - u_k^{old} = y_1\lambda_1^{new}k_{1k} + y_2\lambda_2^{new}k_{2k} - b^{new} - y_1\lambda_1^{old}k_{1k} - y_2\lambda_2^{old}k_{2k} + b^{old}. \tag{3.38}$$

Substituting equation 3.37 into equation 3.36, we have

$$E_k^{new} = E_k^{old} + y_1(\lambda_1^{new} - \lambda_1^{old})k_{1k} + y_2(\lambda_2^{new,clipped} - \lambda_2^{old})k_{2k} + b^{old} - b^{new}. \tag{3.39}$$

Equation 3.39[13] is used in the code implementation for updating the error of example with non-bound multiplier after optimization.

3.5 Outline of SMO algorithm

The following is the exact reproduction of the pseudo-code published by Platt on his website `http://www.research.microsoft.c` (I have modifed the pseudo-code in my code implementation in the second choice hierarchy. This modification is described in chapter 4).

Platt's pseudo-code for the SMO algorithm:

target = desired output vector
point = training point matrix

procedure takeStep($i1$, $i2$)
if $(i1 == i2)$ return 0
alph1 = Lagrange multiplier for $i1$
$y1 = target[i1]$
$E1$ = SVM output on $point[i1]$ - $y1$ (check in error cache)
$s = y1 * y2$
Compute L, H
if $(L == H)$
    return 0
$k11 = kernel(point[i1], point[i1])$
$k12 = kernel(point[i1], point[i2])$
$k22 = kernel(point[i2], point[i2])$
$eta = 2 * k12 - k11 - k22$
if $(eta < 0)$
{
    $a2 = alph2 - y2 * (E1 - E2)/eta$
    if $(a2 < L)$ $a2 = L$
    else if $(a2 > H)$ $a2 = H$
}
else
{
    $Lobj$ = objective function at $a2 = L$
    $Hobj$ = objective function at $a2 = H$
    if $(Lobj > Hobj + eps)$
        a2 = L
    else if $(Lobj < Hobj - eps)$
        $a2 = H$
    else
        $a2 = alph2$
}

if $(|a2 - alph2| < eps * (a2 + alph2 + eps))$
    return 0
$a1 = alph1 + s * (alph2 - a2)$
Update threshold to reflect change in Lagrange multipliers
Update weight vector to reflect change in a1 and a2, if linear SVM
Update error cache using new Lagrange multipliers
Store a1 in the alpha array
Store a2 in the alpha array
return 1
endprocedure

procedure examineExample(i2)
$y2 = target[i2]$
$alph2$ = Lagrange multiplier for $i2$
$E2$ = SVM output on $point[i2]$ - $y2$ (check in error cache)
$r2 = E2 * y2$
if (($r2 <$ -tol and $alph2 < C$) or ($r2 >$ tol and $alph2 > 0$))
{
    if (number of non-zero and non-C alpha > 1)
    {
        $i1$ = result of second choice heuristic
        if takeStep$(i1, i2)$
            return 1
    }
    loop over all non-zero and non-C alpha, starting at random point
    {
        $i1$ = identity of current alpha
        if takeStep$(i1, i2)$
            return 1
    }
    loop over all possible $i1$, starting at a random point
    {
        $i1$ = loop variable
        if takeStep$(i1, e2)$
            return 1
    }
}
return 0
endprocedure

main routine:
initialize alpha array to all zero
initialize threshold to zero
$numChanged = 0$;
$examineAll = 1$;
while $(numChanged > 0 | examineAll)$
    {
        $numChanged = 0$;
        if $(examineAll)$
          loop I over all training examples
            $numChanged$ += examineExample(I)

```
    else
      loop I over examples where alpha is not 0 and not C
        numChanged += examineExample(I)
    if (examineAll == 1)
      examineAll = 0
    else if (numChanged == 0)
      examineAll = 1
}
```

The following is a **summary** of the SMO algorithm.

1 Iterate over the entire training example, searching for a λ_1, which violates the KKT condition.
If λ_1 is found. Go to step 2.
If we finish iterating over the entire training example, then we iterate over the non-bound set.
If λ_1 is found, go to step 2.
We will alternate between iterating through the entire set and the non-bound set looking for a λ_1 which violates the KKT conditions until all λ's obey the KKT condition.
Then we exit.

2 Search for λ_2 from the non-bound set.
Take the λ which gives the largest value of $|E_1 - E_2|$ as λ_2
If the two examples are identical, then abandon this λ_2. Go to step 3.
Otherwise, compute L and H value for λ_2.
If $L = H$, then optimization progress cannot be made. Abandon this λ_2 value. Go to step 3.
Otherwise,Calculate the η value.
If it is negative, then calculate the new λ_2 value.
If η is not negative, then calculate the objective function at the L and H points and use the λ_2 value which gives the higher objective function as the new λ_2 value.
If $|\lambda_2^{new} - \lambda_2^{old}|$ is less than an ε value, then abandon this λ_2 value. Go to step 3.
Otherwise go to step 4.

3 Iterate over the non-bound set starting at a random point in the set until a λ_2 that can make optimization progress in step 2 is found.
If it is not found, then iterate over the entire training example, starting at a random point, until a λ_2 that can make optimization progress in step 2 is found.
If no such λ_2 is found after these two iterations, then we skip the λ_1 value found and go back to step 1 to find a new λ_1 which violates KKT condition.

4 Calculate the new λ_1 value.
Update the threshold b, error cache and store the new λ_1 and λ_2 values.
Go back to step 1.

Everything should be made as simple as possible,
but not simpler.

— Albert Einstein, 1879-1955 —

If it's a good idea ... go ahead and do it. It is much
easier to apologize than it is to get permission.

— Grace Hopper, 1906-1992 —

4

The Software Development Of The Project

4.1 Goal of the project

The goal of this project is to implement a support vector machine on a personal computer using John Platt's Sequential Minimal Optimization Algorithm. The software can be used as a tool for understanding the theory behind support vector machine and for solving binary classification problems.

4.2 General description of the development

The software is made up of two separate packages, the training or learning program named **smoLearn** and the classification program named **smoClassify**. The classification program **smoClassify** will use the result obtained from training with the training program **smoLearn**. Other than this dependency relation, the two software packages are independent.

The implementation was developed on a Toshiba laptop with a Pentium 233 MHz CPU running on OpenLinux 2.3 and was compiled with `gcc`. The programs also compile on SunOs. The GUI part of the software was developed with Python. A machine has to have Python installed properly in order to use the GUI part of the software. A separate software package, named **demo** was also developed for testing and verification of the program during the development and testing phase. This package was developed in Python (see Appendix B for the GUI window of the **demo.py** program). Figure 4.1 shows the system use cases.

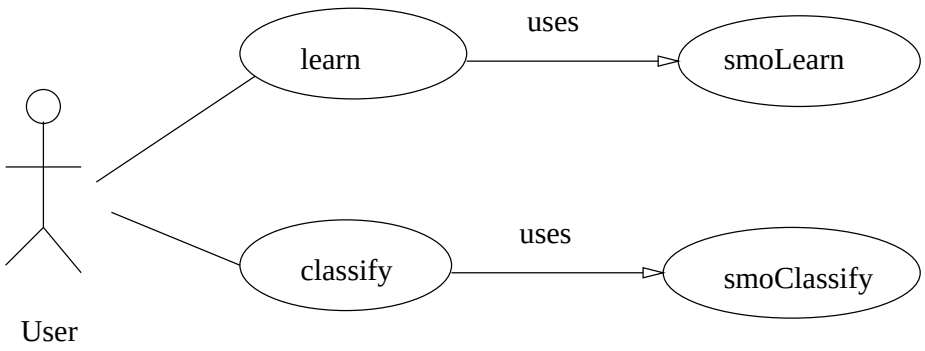


Figure 4.1: Use Cases

4.3 Specification

- Training data for the training program, in this project **smoLearn**, follows a specified file format. Similarly, the input test data for the classification program in this project **smoClassify** follows a specified file format. The file formats are described in section 4.4. If a wrong file format is used in the training, the program will detect the wrong format and abort after informing the user the wrong file format error.
- Each line of the description of a training example cannot exceed 10000 characters.
- The model file used in the software program **smoClassify** has to be a model file written by the training software **smoLearn** during a training session. It is the responsibility of the user to make sure that he/she supplies the correct model file. A model file which results from training in other software will not work with **smoClassify**.
- The result of training will be written to the user specified model file at the end of a training. The file format of this model file follows a format described in section 4.4.
- The results of classification will be written to the user specified prediction file at the end of classification. The file format of this prediction file follows a format described in section 4.4.
- Both **smoLearn** and **smoClassify** can be run in textual mode or in GUI mode. Running in GUI mode requires the installation of Python.
- Depending on the user's choice of kernel type, the GUI will prompt the user for the required input parameters and type check them.
- The C parameter which determines the trade-off between the errors and the separating margin cannot be zero.
- If the software is run in GUI mode, then the instruction messages, error messages and the program execution messages will be displayed in the text window of the GUI. These three types of messages are displayed in three different colors to facilitate reading of the information in the text window.
- The training software will display the time of the training, the threshold b, the norm of the weight vector if the kernel is linear, the number of non-bound support vectors found, the number of bound support vectors, the number of successful iterations and the number of total iterations.
- The classification software will display the time of the classification at the end of the program execution.

4.4 The Format used in the input and output files

The software will use '1' and '-1' as the class label. The file format is compatible with that used for svm^{light}, the SVM implementation developed by Thorsten Joachim. Files readable by svm^{light} will be readable by this software and vice-versa. The file format was deliberately chosen to be compatible with that of svm^{light} so that any training and classification data can be run on both software and their results compared.

4.4.1 Training file format

All comments begin with a # and will be ignored. Once training data begin in the file, there cannot be any more comments. The data file format is:

```
Data File ::= line{line}
line ::= class {feature:value}
class ::= 1 | -1
feature ::= integer
value ::= real
```

You can omit a feature, which has zero value. The feature/value pairs **MUST** be ordered by increasing feature number.

4.4.2 Test data file format

The test data file has the same format as the training file. If one does not know the class label of the test data, one can just put in either 1 or -1. The class labels **MUST** appear in the file.

4.4.3 Model file format

The result of training is written in a model file in the following format:
line 1:
If the kernel type is linear, then '0' is written followed by a comment.
If the kernel type is polynomial, then '1' is written followed by the degree of the polynomial and comment.
For example for a polynomial kernel of degree 2, line 1 will be
1 2 # polynomial of degree 2

line 2:
The number of features of the training set is followed by a comment.
If a linear kernel is used in training, then
line 3:
The weight vector of the training result is followed by a comment.
line 4:
The threshold *b* is followed by a comment.
line 5:
The *C* parameter is followed by a comment.
line 6:
The number of support vectors is followed by a comment.
line 7 to the end of file:
On each line, the value <class*lambda>[1] of a support vector is followed by the feature/value pairs of that support vector.

If a non-linear kernel is used in training, then
line 3:
The threshold *b* is followed by a comment.
line 4:
The *C* parameter is followed by a comment.
line 5:
The number of support vectors is followed by a comment.
line 6 to end of file:
On each line, the value <class*lambda> of a support vector is follwed by the feature/value pairs of that support vector.

4.4.4 Prediction file format

Each line consists of <result of svm> If a data is classified as class 1, then <result of SVM> is positive.
If a data is classified as class -1, then <result of SVM> is negative.

4.5 The software structure

The project is made up of two separate executable programs, **smoLearn** for training and **smoClassify** for classifying unknown data and a GUI program **smo.py**.

Figure 4.2, Figure 4.3 and Figure 4.4 give an overview of the project, the **smoLearn** package and the **smoClassify** package respectively. The programs were developed in C programming language because of C's execution speed for numerical calculations. Since the target users of the software are mostly researchers in machine learning who

[1] <class*lambda> is the multiplication product of class and lambda. This is positive for class 1, and negative for class -1.

Figure 4.2: Project Overview

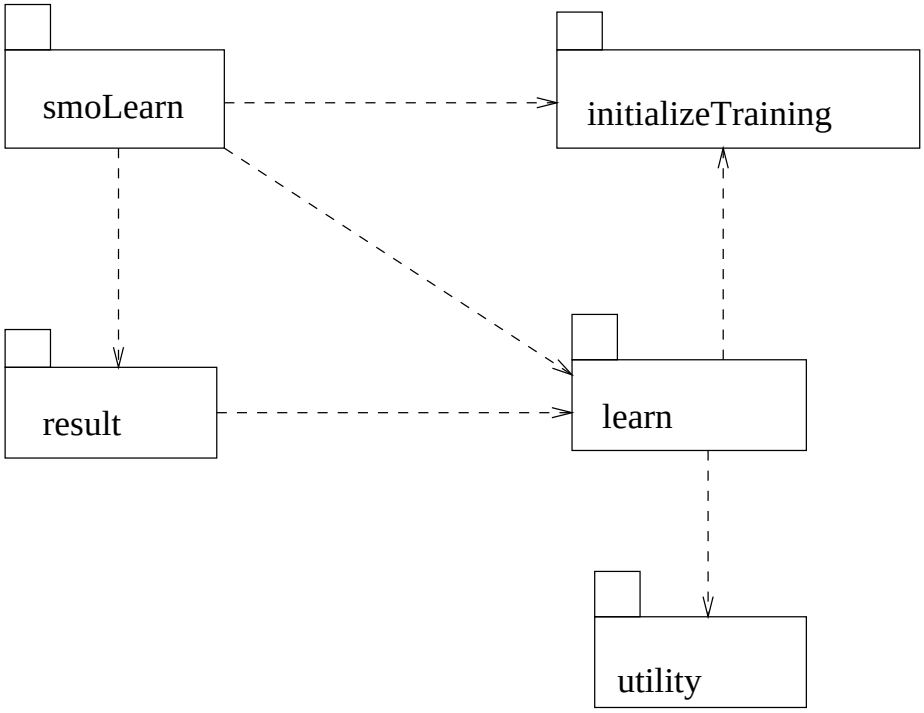


Figure 4.3: smoLearn's modules

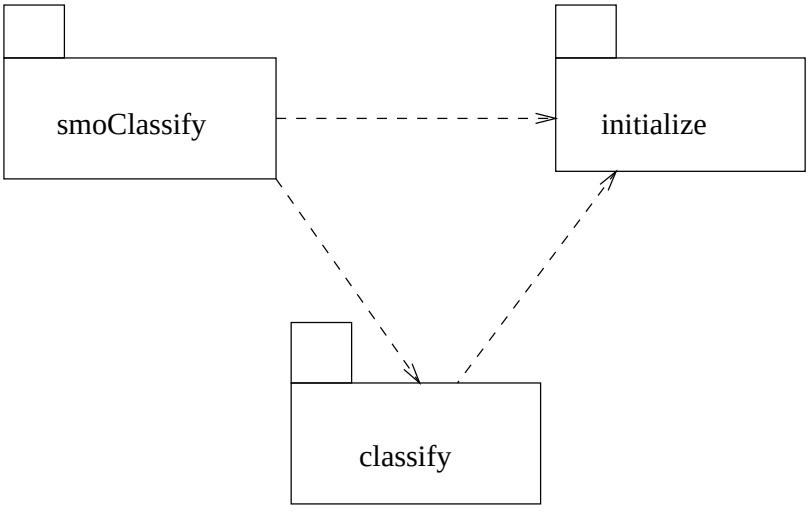


Figure 4.4: smoClassify's modules

are computer literate and are comfortable with textual commands, the software was designed to run in textual mode. However, a user-friendly GUI was also developed in Python and Tkinter.

The whole design of the software is based on simplicity and usability. Since the primary objective of the project is to provide a tool for studying support vector machine and to implement the Sequential Minimal Optimization Algorithm correctly, simplicity was given a higher priority than speed. Simple data structures and supporting algorithms are used to facilitate testing and maintenance.

4.5.1 The training software: smoLearn

The software can be invoked in textual mode by typing the following:

```
smoLearn [options] <training file name> <model file name>
```

The available optons are:

- -t type:Integer—Type of kernel function:
  type = 0: linear kernel
  type = 1: Polynomial kernel
  type = 2: RBF kernel

- -c C:Float $>$ 0—The trade-off between training errors and the margin.
  The default is 0.

- -d degree:Integer—The degree of the polynomial kernel.

- -v variance:Float—The variance of the RBF kernel

- -b bin:Integer(0 or 1)
  Default is 0.
  1 to specify the features of the training example are binary values.
  Only use -b 1 if you are certain the features of your training examples are binary value 0 and 1.

- -h
  This is the help option which displays the available options.

The followings are four examples illustrating how to invoke **smoLearn** using the different options [2].

Example 1:
To use linear kernel, a C parameter of 10.5 to train with <my training file> and <my model file> Type: **smoLearn -t 0 -c 10.5 <my training file> <my model file>**

Example 2:
To use a polynomial kernel of degree 2 and a C parameter of 10
Type: **smoLearn -t 1 -c 10 -d 2 <my training file> <my model file>**

Example 3:
To use a RBF kernel of variance of 10 and a C parameter of 3.4
Type: **smoLearn -t 2 -c 3.4 -v 10 <my training file> <my model file>**

[2] The options can be specified in any order

Figure 4.5: Menu of smo GUI

Example 4:
To use the binary feature option
Type: **smoLearn -t 0 -c 4.5 -b 1 <my training file> <my model file>**

To run the program in GUI mode, the user will type **smo.py** (run the python program as a script) or **python smo.py** to start the selection window. See Figure 4.5. When the user clicks on the **learn** button, the Learning window pops up. See Figure 4.6.

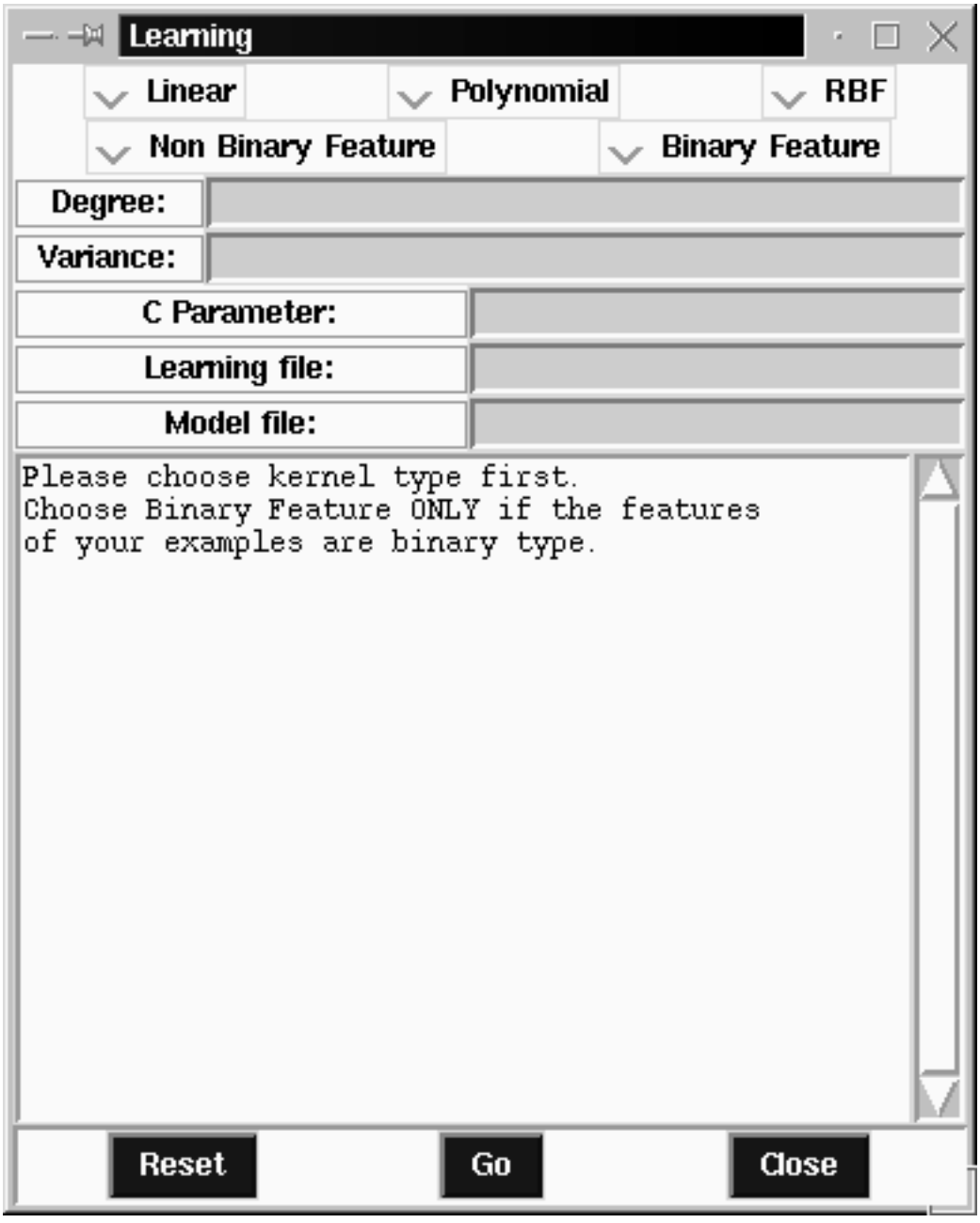


Figure 4.6: GUI of smoLearn

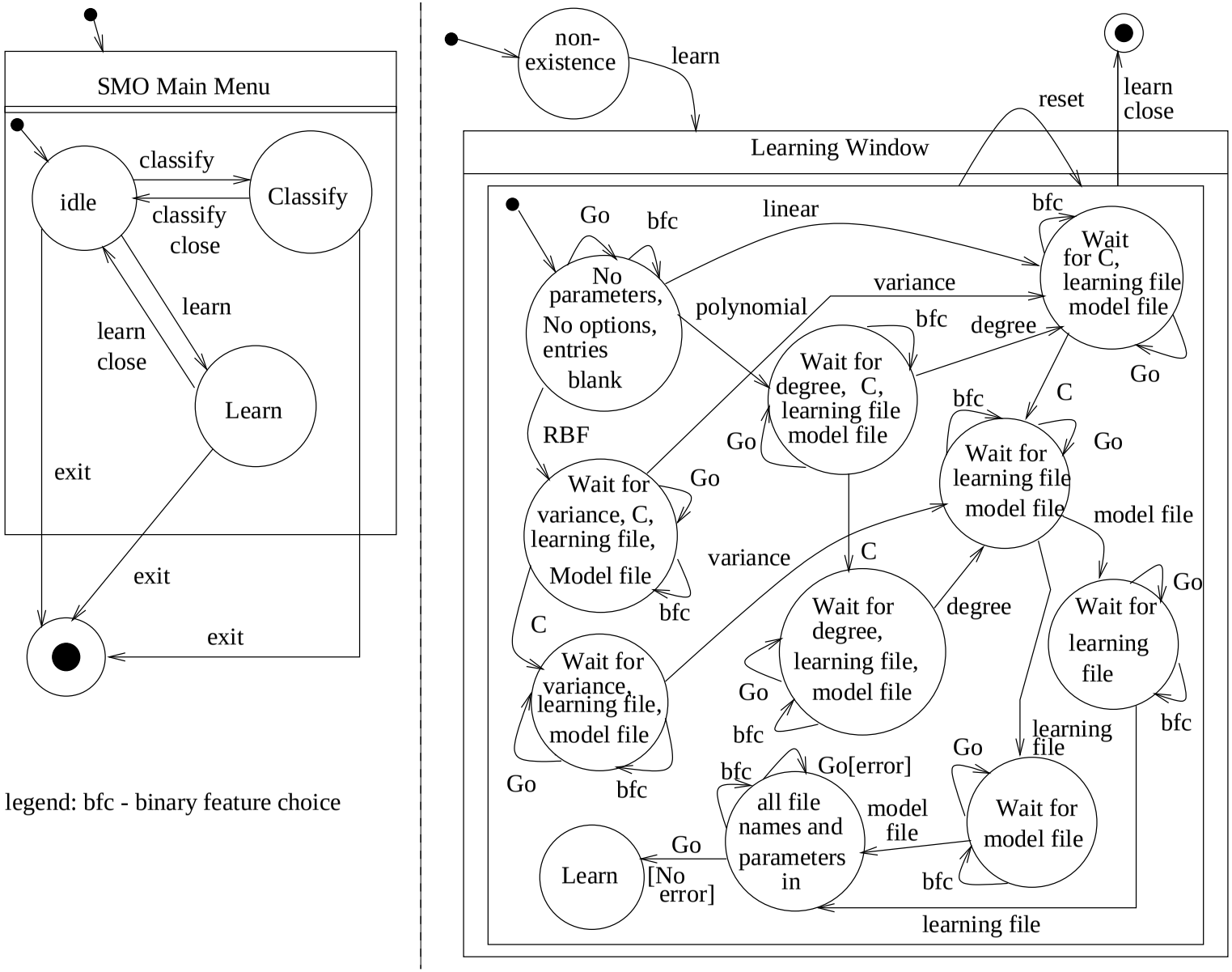


Figure 4.7: State diagram of the GUI of the smo main menu and the smoLearn window

A state diagram describing the state transitions of the main **SMO** menu and the learning window is shown in Figure 4.7. This window prompts a user to select the kernel type and based on the kernel choice, the cursor will jump to the required text entry and a message is displayed in black in the text box telling the user what other entries need to be filled in. For example in Figure 4.6, if the user selects polynomial kernel, the cursor then flashes in the degree blank field. After the user types in the degree, the cursor jumps to the C parameter blank field. After the user types in the C parameter, the cursor jumps to the 'Learning file' entry. After the user types in the learning file, the cursor will jump to the 'Model file' entry field. The GUI only type checks 'Degree', 'Variance' and 'C parameter' and makes sure that the user has input the learning file and model file names. It then calls the C program **smoLearn**, which will check if the the learning and model files entered by the user exist and can be open for reading and writing respectively. Messages to users are color coded and are displayed in the text area of the window. The information messages to users are in black, error messages are in red and program run messages are in blue. If the files do not exist or there is an error in opening the files, an error message is displayed in the text box.

Five modules make up the **smoLearn** software package. See Figure 4.3

- Main module
- initializeTraining module
- learn module
- utility module
- result module

Main module:

The module **smoLearn.c** makes up the main module. As can be seen from the learning sequence diagram Figure 4.8, **smoLearn.c** orchestrates the whole learning process.

smoLearn.c will do its type checking for correct types of parameters based on the kernel chosen by the user. This is done so that users can run the program in either GUI mode or textual command line mode. The module also checks for existence of the training file supplied by user and opens the model file with name supplied by the uesr for writing the training result to. After all the type checking, **smoLearn.c** records the type of kernel chosen by the user, the C parameter, the degree if polynomial kernel is selected, the square of the sigma (variance) if an RBF kernel is used, and the names of the training and model files supplied by the user. It opens the training file for reading and the model file for writing. If there are problems in opening those files, then the program will inform the user the errors and abort. Otherwise **smoLearn.c** will ask the **initializeTraining module** to do all the preparation tasks before the training process starts. These tasks include reading the training file and storing those training data and initializing all the data structures for the training process. If the initialization process fails, then **smoLearn.c** will abort the program. After initializing, **smoLearn.c** calls the **learn module** to start learning and starts a timer. At the end of learning, **smoLearn.c** calculates the time it took the learning algorithm to run and calls the **result module** to write the result of training in the model file. **smoLearn.c** then writes out the following statistics.
The number of successful kernel iterations.
The total number of iterations.
The threshold value, *b*.
The norm of the weight vector if a linear kernel is used.
The number of non-bound support vectors and the number of bound support vectors.

initializeTraining module

The tasks of the **initializeTraining module** consist of allocating memory for the following data structures and reading the training file.

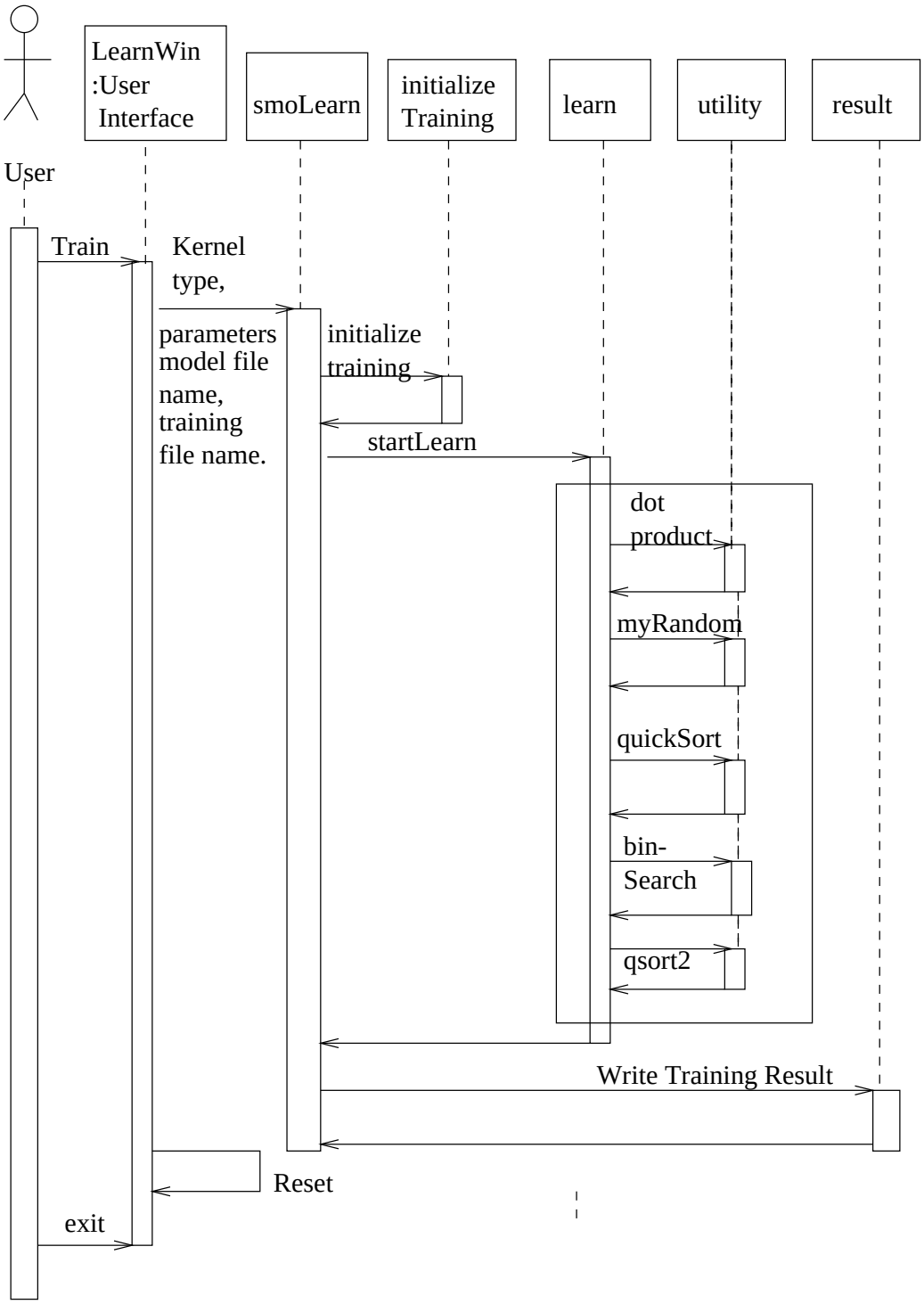


Figure 4.8: Learning sequence diagram

Data structures initialized by **initializeTraining module**:

- A structure called feature consists of an integer element for storing the id of the feature and a double element for storing the value of that feature is defined. Only features with non-zero feature value are stored. Therefore each training example has a number of these structures allocated for it according to the number of non-zero valued features it has.
- A two-dimensional arrays of pointers to the structure feature, **example ****, is allocated for storing all the information of feature id/value pairs of the examples. Hence, example[*i*][*j*] will point to the *j*th feature structure which has non-zero feature value of the *i*th example.
- An array of integers, **target**, is used to store the class labels of the training example. Class labels are '1' or '-1'. Therefore **target[*i*]** is either 1 or -1 for the *i*th example. These labels are read from the training file supplied by the user. For training file format, see section 4.4.
- An array of integers, **nonZeroFeature**, is used to store the number of non-zero valued features an example has.
- An array of doubles, **error**, is used to store the error of each example. This is the difference between the output of the support vector machine on a particular example and its class label. For the *i*th example, error[*i*] stores the output of support vector machine - target[i].
- An array of doubles, **weight**, is used to store the weight vector of the support vector machine. This is necessary only for a linear kernel.
- An array of doubles, **lambda**, is used to store the Lagrange multipliers of the training examples during training.
- An array of integers, **nonBound**, is used to record whether an example has a non-bound Lagrange multiplier during training, i.e. the value of the multiplier is in the interval (0, C) where C is the parameter which determines the trade-off between training error and margin.
- An array of integers, **unBoundIndex**, is used to index the examples which have non-bound Lagrange multipliers.
- An array of integers, **errorCache**, is used to index examples with non-bound Lagrange multipliers in the order of their increasing errors during training.
- An array of integers, **nonZeroLambda**, is used to index the examples whose Lagrange multipliers are non-zero during training.

Another task performed by **initializeTraining module** is to read the training file and store the information of the training examples.

There are 3 functions in this module, **initializeData()**, **readFile()** and **initializeTraining()**

- **int initializeData(int size)**
  This function initializes the data structure **example ****, **target**, **nonZeroFeature** and **error** to the size passed in the function argument. It will return 1 if initialization is successful, otherwise it will return 0.
- **int readFile(FILE *in)**
  This function assumes that the file pointed to by file pointer 'in' has been opened successfully. The function makes two passes of the training file. In the first pass, it just counts the number of lines in the file to have an over estimate of the number of training examples. Using this estimate, the arrays **example**, **target**, **nonZeroFeature** and **error** are allocated. In the second pass, it reads each non-comment line and stores the class label either 1 or -1 for each example in the **target** array and the feature id/value pairs in the **example** array. Only non-zero valued features are stored. While reading the lines, the function also keeps track of the maximum feature id it has encountered so far. At the end of the reading of the file, this maximum feature id is taken to be the number of features the training examples have. The user does not have to supply the number of features of the examples. The function returns 1 if reading is successful, otherwise it returns 0.

- **int initializeTraining(void)**
  This function initializes the data structures **weight**, **lambda**, **nonBound**, **unBoundIndex**, **errorCache** and **nonZeroLambda**. It returns 1 if the operation is successful, otherwise it returns 0.

Learn module

This module runs the training algorithm Sequential Minimal Optimization algorithm. It follows the pseudo codes described in section 5 of chapter 3 (except for the modification in the second choice heuristic in the **examineExample** function) and the equations derived in section 3. There are five functions in the module.

- **void startLearn(void)** This function first iterates over all the training examples and calls function **examineExample()** to check if the example violates KKT condition at each iteration. Then the examples with non-bound Lagrange multipliers are iterated over. The function alternates between iterating over all examples and all non-bound examples until all the Lagrange multipliers of the examples do not violate the KKT conditions. At this point, the function exits.
- **examineExample(int e1)** This function checks for violation of KKT conditions of example e1. If it does not violate the KKT condition then it returns 0 to the function **startLearn()**, otherwise, it uses heuristics as outlined in section 2 of chapter 3 to find another example for joint optimization. Platt uses a hierarchy of choices in the second choice heuristic for choosing the second example for optimization. In the second hierarchy of the heuristic, he iterates over the non-bound examples starting from a randomly chosen position. If this does not produce a candidate example for joint optimization, then he iterates over all examples looking for another candidate example. Since structurally, the separating plane orients itself over the optimization process, it is more likely that a bound example will become no-bound than for an example with zero Lagrange multiplier becoming a support vector. I added an hierarchy right after the second hierarchy of the second heuristic and changed the last hierarchy. If iteration over non-bound examples does not produce an optimization then the next iteration is carried out over the bound examples in selecting the second example for joint optimization. The iteration starts at a random position over the bound examples. If this still does not work out, then the iteration will be done over the examples with zero Lagrange multipliers. This function returns 1 if optimization is successful, otherwise it returns 0.
- **int takeStep(int e1, int e2)**
  Two arguments e1 and e2 are passed to the function. They are the examples chosen for joint optimization. Having chosen a second example for joint optimization, **examineExample()** calls **takeStep()** to carry out the optimization. The optimization step follows closely with what is described in section 2 of chapter 3. If the second derivative eta, (equation 3.14) is negative, then we calculate the new lambda for the second example and clip it at the ends of the line $L2$ and $H2$. Otherwise, we choose either $L2$ or $H2$ as the new lambda value depending on which one gives a higher objective function value. If joint optimization is successful, **takeStep()** returns 1 to the function **examineExample()**, otherwise it returns 0.

  At the end the function updates the condition of the support vector machine and all the data structures, which are necessary not only for the optimization process, but also for the **examineExample()** function. Each optimization step occuring in the function **takeStep()** will result in a change in errors for the non-bound examples and change in the threshold value. The first hierarchy of the second choice heuristic used by **examineExample()** requires choosing the second example with the largest positive error if the error of the first example is negative and one with the largest negative error if the error of the first example is positive. The second hierarchy of the second choice heuristic requires iterating over the examples with non-bound lambdas. The third hierarchy requires iteration over the bound examples. An array named **errorCache** holds the index to the non-bound examples arranged in order of increasing error. An array named **unBoundIndex** holds the index to the non-bound examples. An array named **nonZeroLambda** holds the index to the examples whose lambdas are not

zero. These arrays are updated at the end of each successful optimization. If after an optimization, the lambda of the *i*th example is non-bound whereas before it is not non-bound, then the index *i* is added to the end of the **errorCache**, and the **unBoundIndex** arrays. If that example has zero lambda before and has non-zero lambda after optimization, then it is added to the **nonZeroLambda** array. Then **takeStep()** will call the **quicksort()** function to sort the **unBoundIndex** array and the **nonZeroLambda** array in ascending order. In this case, we always have a sorted index array to examples with non-zero lambdas and non-bound lambdas. Sorting those indexes let us quickly locate an index by using a binary search. This is necessary when we have to remove them from the array because the example has its lambda changed from non-bound to bound or in the **nonZeroLambda** array case, when its lambda changes from non-zero to zero. To remove the index from those arrays, we use a pointer named unBoundPtr for the **unBoundIndex** array and lambdaPtr for the **nonZeroLambda** array. These pointers will always point to the last element of the respective arrays. After we locate the position of the example index and we want to remove the index from either the **unBoundIndex** array or the **nonZeroLambda** array, we mark that position with the number equal to the total number of example + 1. That index will be at the end of the array when we quicksort the array. We just decrement the pointer to remove the index from the list of examples of non-bound lambdas or the list of examples with non-zero lambda after quicksorting.

The same method applies to the **errorCache** array. The array holds indexes of examples with non-bound lambdas sorted in increasing order of errors. We use the **qsort2()** in the utility module to sort the indexes in ascending order of error after all the errors of non-bound examples have been updated at the end of optimization. A pointer, errorPtr, always points to the last element of the array. When an example whose lambda was not non-bound and becomes non-bound after optimization, we add that index to the end of the array and calls qsort2() to update the array. If example *i* has its lambda changed from non-bound to bound, we will put into error[*i*] an error which is one greater than the largest error in the non-bound examples. Calling **qsort2()** will then put *i* at the end of the **errorCache** array. We can remove the index *i* just by decrementing the errorPtr.

- **double dotProduct(FeaturePtr *x, int sizeX, FeaturePtr *y, int sizeY)**
  The **dotProduct()** function calculates the dot product of two examples and returns the result to the calling function. For a sparse binary matrix, the calculation is sped up by the method suggested by Platt [13]. Instead of doing multiplication, we just increment by 1 whenever the two vectors have the same non-zero valued feature.
- **double calculateError(int n)**
  This function is called by **takeStep()** at the end of a successful optimization to update the errors of all the examples with non-bound lambdas. It calculates the error of a non-bound example. The updating just follows the equation 3.39 in section 1 of chapter 3

utility module

This module contains 6 functions which provide support for the **learn** module to carry out its task. They are functions which are not specific utility functions and can be used for many applications.

- **double power(double x, int n)**
  This function just calculates and returns the value of a double x, raised to the power of an integer, n. It returns 1 when n is zero. It assumes that n is zero or a positive integer.
- **int binSearch(int x, int *v, int n)**
  This function searches for x in an integer array v of size n. It returns -1 if x is not in the array, otherwise it returns the array position where x is found.
- **void swap(int *v, int i, int j)**
  This function swaps the ith element with the jth element of an integer array.

- **void quicksort(int *v, int left, int right)**
  This function uses the quicksort algorithm to sort the elements bounded by the left and right indexes of the integer array v in ascending order.
- **qsort2(int *v, int left, int right, double *d)**
  This function sorts the elements bounded by the left and right indexes of the integer array v in an order such that d[v[i]] $<$ d[v[$i+1$]] in array d for the ith and $(i+1)$th elements of array v.
- **int myrandom(int n)**
  This function returns a random integer in the interval [0, n-1]. The seed is initially set at 0 and is automatically increased by one at each call.

result module

There is only one function in this module.
void writeModel(FILE *out) takes the passed argument, an open file pointer, for writing and writes the results of the training in a fixed format. It assumes that the file has been open successfully for writing. See section 4.4 regarding the model file format.

4.5.2 The classification software: smoClassify

smoClassify is a stand alone software package which is invoked by the following command when run in textual mode
smoClassify <**model file name**> <**test data file name**> <**prediction file name**>
Running the program in GUI mode, the user will click on the **classify** button on the selection window (see Figure 4.5) to invoke **smoClassify**. The classify window then pops up and prompts the user to input the model file name, the test data file name and the prediction file name (see Figure 4.9). Messages to user are displayed in the text area of the classify window and they are color coded. Information messages to the user are displayed in black, error messages are displayed in red and the program display of **smoClassify** are displayed in blue. The GUI only checks that the user has supplied all three file names. When this is done, it calls the C program **smoClassify**. **smoClassify** checks for existence of model file and test data file and opens these two files for reading as well as the prediction file for writing.

Figure 4.9: GUI of smoClassify

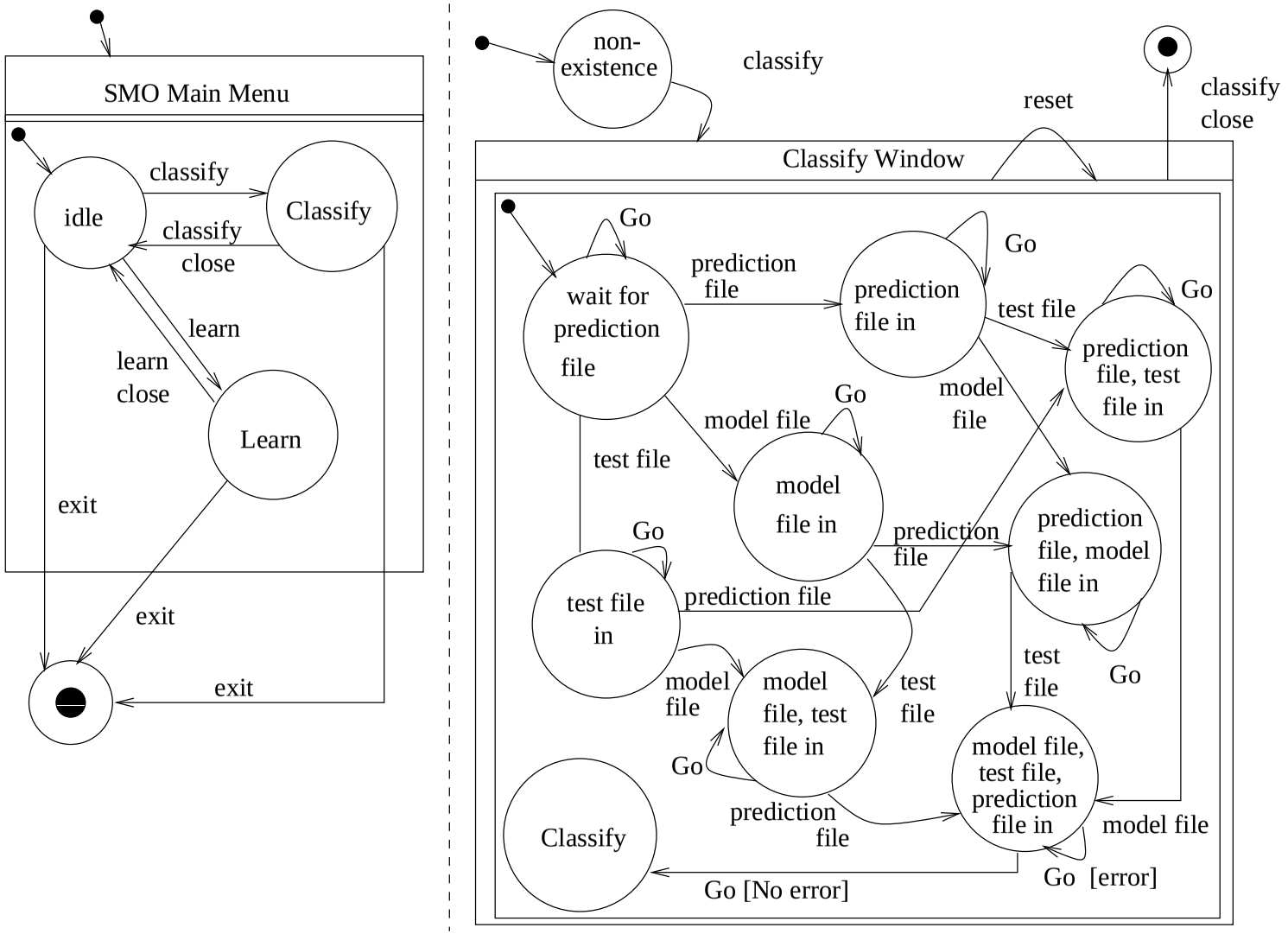


Figure 4:10: State diagram of SMO Main menu and the Classify window.

Figure 4.10 gives the state diagram of the GUI of the **SMO** Main menu and the classify window.

The software program consists of three modules. See the **smoClassify** package diagram, Figure 4.4.

- **Main module**
- **initialize module**
- **classify module**

Main module

The main module consists of **smoClassify.c** which directs the classification process. See Figure 4.11.

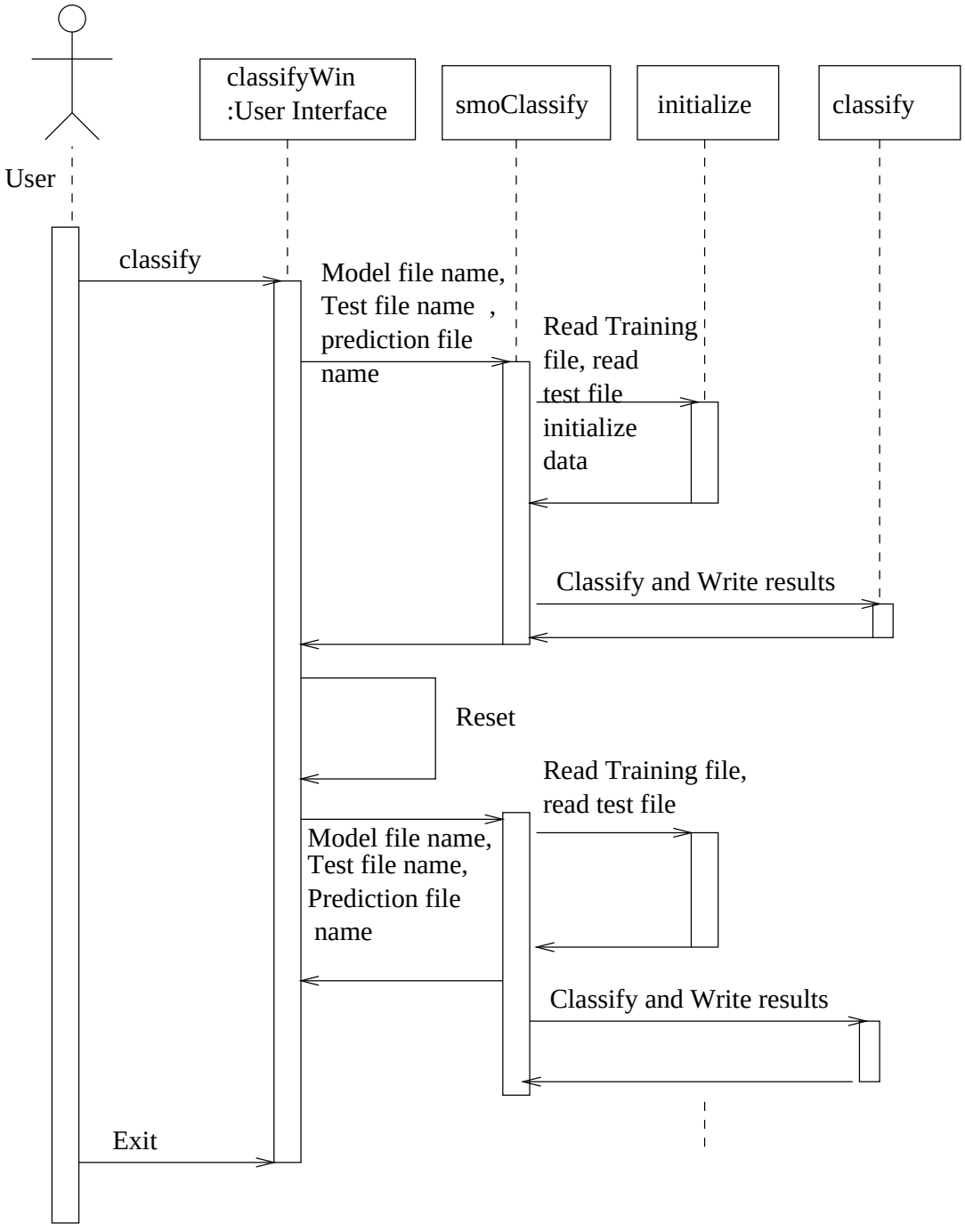


Figure 4.11: Classifying sequence diagram

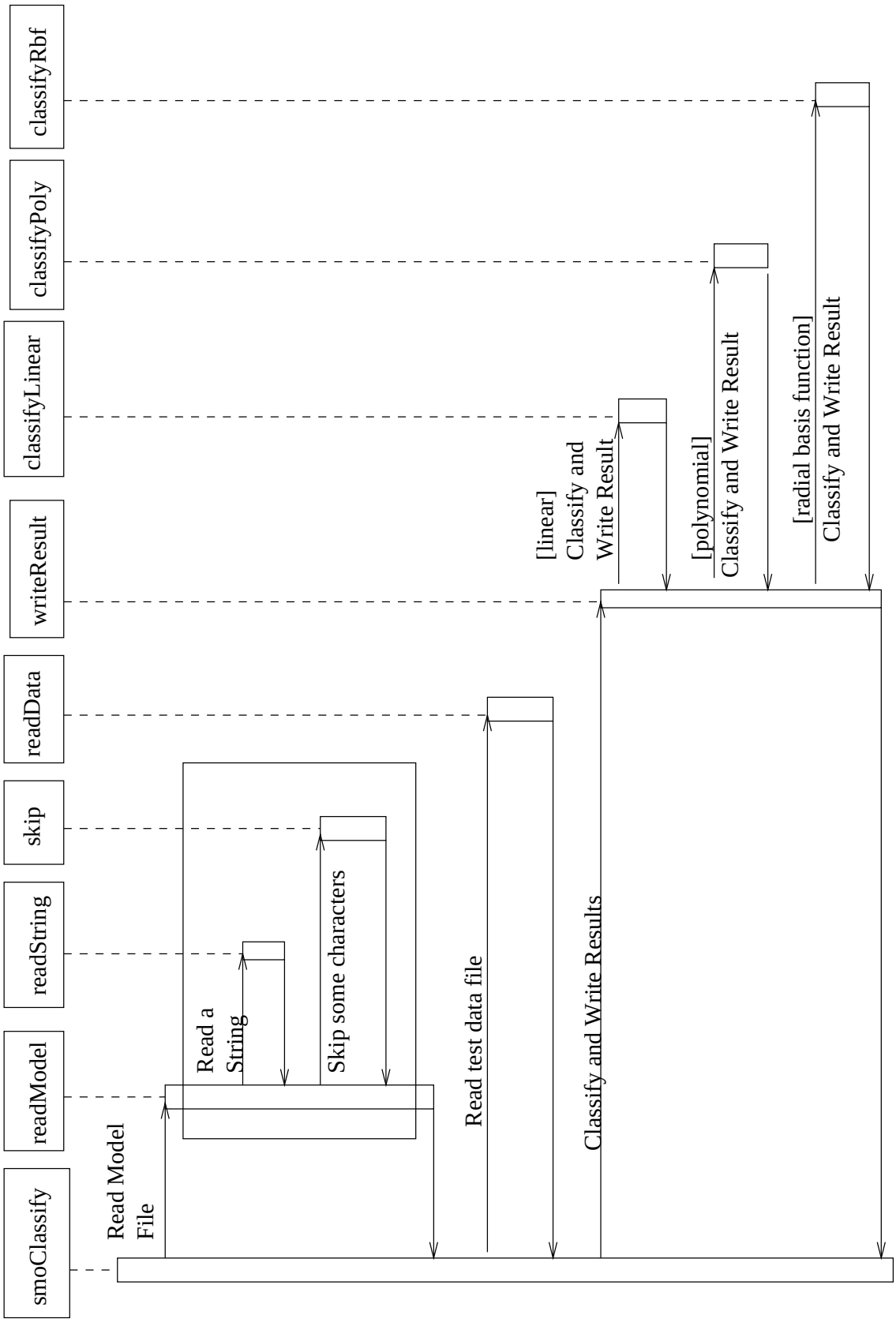


Figure 4.12: Details of classifying sequence

Figure 4.12 shows a more detailed classify sequence. **smoClassify.c** opens the given model file, test data file for reading and the prediction file for writing. Then it calls the **initialize module** to prepare data structures for the classification task. If there is a problem in preparing the data structures and reading the model and data files, then the program aborts, otherwise, **smoClassify.c** calls the **classify module** to classify the test data and writes the result to the prediction file. If there is an error in the classification process, the program aborts.

initialize module

The tasks of **initialize module** are to initialize all the data structures for storing information which will be read from the model file. The two public interfaces are **int readModel(FILE *in)** and **int readData(FILE *in)**. The followings are the data structures initialized by this module.

- **example**, a two-dimensional array of pointers to the structure feature. The structure feature is defined as:

  ```
  struct feature
  {
        int id;
        double value;
  }
  ```

  This array will store the feature id/value pairs of each test data read from the test data file.
- **sv**, an array of pointers to the structure features (same as **example**) for storing the id/value pairs of the support vectors read from the model file.
- **target**, an array of integers for storing the class label of the test data read from the test data file. In this software, this information is not used at all. The user can supply the class label if s/he knows the class label of the data ahead of time and wants to see how a particular trained model performs. If the user is only interested in the classification of some unclassified data, then s/he can put down any class labels in the data file. Please refer to section 4.4 of chapter 4 regarding the data file format.
- **nonZeroFeature**, an array of integerss for storing the number of non-zero valued features each test data has.
- **lambda**, an array of doubles for storing the Lagrange multipliers of each test data.
- **svNonZeroFeature**, an array of integers for storing the number of non-zero valued features a support vector has.
- **weight**, an array of doubles for storing the weight vector of a trained support vector machine. This is only necessary for linear kernel.
- **output**, an array of doubles for storing the output of a trained support vector machine on each test data.

This module is made up of 6 functions.

- **int readString(char *store, char delimiter, FILE *in)**
  This function assumes the file pointed to by file pointer 'in' has been opened successfully for reading. It reads all the characters in a file up to the delimiter and stores the characters read in the given array **store**. It returns 1 if reading is successful, otherwise it returns 0.
- **int initializeModel(int size)**
  This function initializes the arrays **lambda**, **svNonZeroFeature** and **sv** to the given size. It returns 1 if initialization is successful, otherwise it returns 0.
- **int readModel(FILE *in)**
  This function assumes that the file pointed to by file pointer 'in' has been opened successfully for reading. This function reads the model file to find out the followings.

the number of features of the examples used in the training
the kernel type
the weight vector if the training was done using a linear kernel
the threshold *b*
the *C* parameter
the number of support vectors found in the training
the feature id/value pairs
the product (class label*Lagrange multiplier) of the support
vectors listed in the model file.

The function returns 1 if reading was successful, otherwise it returns 0.

- **int readData(FILE *in)**
  This function assumes that the file pointed to by file pointer 'in' has been opened successfully for reading. It behaves like the function **readFile()** in the **initializeTraining** module of the program **smoLearn**. It just reads the test data file to store the feature id/value pairs of each test data. The function returns 1 if reading is successful, otherwise it returns 0.
- **void skip(char end, FILE *in)**
  This function assumes that the file pointed to by file pointer 'in' has been successfully opened for reading. It skips all the characters read in the file until the end character is encountered.

classify module

This module is responsible for calculating the output of the trained support vector machine on each test data and for writing out the classification result to a given prediction file. The public interface of the module is the function **writeResult(FILE *out)**. The module is made up of the following functions:

- **double dotProduct(FeaturePtr *x, int sizeX, FeaturePtr *y, int sizeY)**
  Two vectors of the data are passed as argument together with the number of non-zero valued features they have to this function. The function calculates the dot product of the two vectors and returns the dot product.
- **double wtDotProduct(double *w, int sizeX, FeaturePtr *y, int sizeY)**
  If the kernel used in training is the linear kernel, the weight vector was calculated and written to the model file at the end of training. The output of the support vector machine can be more quickly calculated by finding the dot product of the weight vector with the test data vector. This function is passed the weight vector, the vector of data together with the number of nonzero valued features of the weight vector and the data vector. It calculates and returns their dot product.
- **double power(double x, int n)**
  This function returns the value of x raised to the power of n. When n is zero, 1 is returned. It is assumed that n is zero or a positive integer.
- **int writeResult(FILE *out)**
  The function assumes that the file pointed to by file pointer 'out' has been opened successfully for writing. The function calls the function **classifyLinear()**, or **classifyPoly()**, or **classifyRbf()** depending on whether the kernel used in training was linear, polynomial or radial basis function respectively. It returns 1 if classification and writing is successful, otherwise it returns 0.
- **int classifyLinear(FILE *out)**
  This function assumes that the file pointed to by file pointer 'out' has been opened successfully for writing. It also times the classification and writes the time to stdout at the end of the classification. The function calculates the output of the support vector machine on each test data and writes the output of the SVM on the file pointed to by 'out'. If the output is positive, the class label of the data is 1, else it is -1. The output of the support vector machine for a linear kernel is calculated by means of the following equation.

$$\text{SVM } output = \mathbf{w}.\mathbf{x} - b,$$

where **w** is the weight vector of the SVM and *b* is the threshold.

- **int classifyPoly(FILE *out)**
  This function assumes that the file pointed to by file pointer 'out' has been opened successfully for writing. It also times the classification and writes the time to stdout at the end of the classification. The function calculates the output of the support vector machine on each test data and writes the output of the SVM on the file pointed to by file pointer 'out'. If the output is positive, the class label of the data is 1, else it is -1. The output of the support vector machine on a test data **x** for a polynomial kernel is calculated by means of the following equation.

$$\text{SVM } output = \sum_{i=1}^{n} \lambda_i y_i (1 + \mathbf{x_i}.\mathbf{x})^d - b,$$

  where λ_i is the Lagrange multiplier, y_i is the class label of a support vector $\mathbf{x_i}$, n is the number of support vectors and b is the threshold.
- **int classifyRbf(FILE *out)**
  This function assumes that the file pointed to by file pointer 'out' has been opened successfully for writing. It also times the classification and writes the time to stdout at the end of the classification. The function calculates the output of the support vector machine on each test data and writes the output of the SVM to the file pointed to by file pointer 'out'. If the output is positive, the class label of the data is 1, else it is -1. The output of the support vector machine on a test data **x** for a radial basis function kernel is calculated by means of the following equation.

$$\text{SVM } output = \sum_{i=1}^{n} \lambda_i y_i \exp^{\frac{-\|\mathbf{x_i}-\mathbf{x}\|^2}{2\sigma^2}} - b,$$

  where λ_i is the Lagrange multiplier, y_i is the class label of a support vector $\mathbf{x_i}$, n is the number of support vectors and b is the threshold.

If you do something once, people will call it an accident. If you do it twice, they call it a coincidence. But do it a third time and you've just proven a natural law.

— Grace Hopper, 1906-1992 —

5 Testing

Two types of tests were performed on **smoLearn** and **smoClassify**. The first type of tests was performed using artificial data sets while the second type was performed using two benchmark test sets.

5.1 Artficial data test

Four 2-dimensional test sets are generated for testing the correctness of **smoLearn** and **smoClassify**. There are 8 test files in the artificial data set. They are described in the sequel:

- testFile1_1 - linearly separable. Used for training on a linear SVM.
- testFile1_2 - Used for linear kernel classification testing.
- testFile2_1 - not linearly separable. Used for training on a linear SVM.
- testFile2_2 - Used for linear kernel classfication testing.
- testFile3_1 - nonLinearly separable. Used for training on a polynomial SVM.
- testFile3_2 - Used for polynomial kernel classification testing.
- testFile4_1 - nonlinearly separable. Used for training on a rbf SVM.
- testFile4_2 - Used for radial basis function classification testing.

Four tests were performed on **smoLearn** using the four training data sets. At the end of training, reclassification tests were done on the training data itself followed by generalization tests using the corresponding test data. An example of the test procedure is as follows:

1. Train a linear SVM using testFile1_1 and write the model file created.
2. Do the reclassification on testFile1_1 and note the reclassification results
3. Do a classification using testFile1_2 and note the classification results.

These tests were repeated using the software svm^{light} which can be downloaded from the website http://svm.first.gmd.de The purpose of the tests is to verify the correctness of the implementation of SVM using the three different types of kernels.

5.1.1 testFile1_1 and testFile1_2

testFile1_1 consists of 12 data which can be linearly separated. The purpose of the test is to verify the linear SVM training model created by **smoLearn** and the classification correctness of **smoClassify**

Training Software	Threshold b	Non-Bound SVs	Bound SVs	Reclassification Result testFile1_1	Classification with smoClassify testFile1_2
smoLearn	5.000000	2	0	12 correct 12 total	12 correct 12 total
svm^{light}	5.000000	2	0	12 correct 12 total	12 correct 12 total

Table 5.1: smoLearn for a linear SVM on testFile1_1, $C = 4.5$

Training Software	Threshold b	Non-Bound SVs	Bound SVs	Reclassification Result testFile2_1	Classification with smoClassify testFile2_2
smoLearn	2.330163	3	13	7 incorrect 26 correct	10 correct 10 total
svm^{light}	2.3280552	4	12	7 incorrect 26 correct	10 correct 10 total

Table 5.2: smoLearn for a linear SVM on testFile2_1, $C = 4.5$

TestFile1_2 consists of 12 data placed in the instance space of testFile1_1. Two points are placed within the margin. One of them is placed right on the separating plane. The rest of the data are evenly distributed in the instance space outside the margins.

5.1.2 testFile2_1 and testFile2_2

testFile2_1 consists of 33 data with two classes of data within very close distance from each other. Two points labelled as class -1 are placed within the region of the points labelled as class 1. They are errors.
testFile2_2 consists of 10 data which are correctly labelled as class 1 and class -1. The purpose of this file is to test the generalization performance of **smoClassify** when the SVM was trained with errors in the training space.

5.1.3 testFile3_1 and testFile3_2

testFile3_1 consists of 40 data points which can be nonlinearly separated. This file is used to test the polynomial SVM implementation of **smoLearn**.
testFile3_2 consists of 14 data with some data points within the margins between the two classes. The purpose of this file is to test the correctness of the polynomial classification implementation of **smoClassify**.

5.1.4 testFile4_1 and testFile4_2

testFile4_1 consists of 75 data points which can be nonlinearly separated. Data of class -1 are sandwiched between data labelled as class 1. There are two nonlinear optimal separating planes. The purpose of the test is to test the RBF SVM implementation of **smoLearn**
testFile4_2 consists of 30 test data. They are uniformly distributed in the instant space of testFile4_1. The purpose of this file is to test the correctness of the RBF classification implementation of **smoClassify**.

5.1.5 Test Results

The test results are shown in Table 5.1 through 5.5. Each table shows the test results of **smoLearn** and **smoClassify** together with those obtained with *svm*^{light}

Training Software	Threshold b	Non-Bound SVs	Bound SVs	Reclassification Result testFile2_1	Classification with smoClassify testFile2_2
smoLearn	3.835000	2	13	33 correct 33 total	10 correct 10 total
svm^{light}	3.8350001	2	13	33 correct 33 total	10 correct 10 total

Table 5.3: smoLearn for a linear SVM on testFile2_1, $C = 10.5$

Training Software	Threshold b	Non-Bound SVs	Bound SVs	Reclassification Result testFile3_1	Classification with smoClassify testFile3_2
smoLearn	-14.331886	5	9	1 incorrect 39 correct	14 total All correct
svm^{light}	-14.325301	5	9	1 incorrect 39 correct	14 total All correct

Table 5.4: smoLearn for a polynomial SVM on testFile3_1, degree = 2, $C = 10.5$

Training Software	Threshold b	Non-Bound SVs	Bound SVs	Reclassification Result testFile4_1	Classification with smoClassify testFile4_2
smoLearn	-0.170436	17	12	2 incorrect 73 correct	30 total 30 correct
svm^{light}	-0.16964309	17	12	2 incorrect 73 correct	30 total 30 correct

Table 5.5: smoLearn for a linear SVM on testFile4_1, $C = 4.5$, $\sigma^2 = 1$

5.1.6 Discussion of the artificial data test

Table 5.1 shows that **smoLearn**, **smoClassify** produce the same result as *svm*^{light} in training and classification.

Two tests were performed using testFile2_1: one with a C parameter of 4.5 and another with a C parameter of 10.5. **smoLearn** and **smoClassify** produce results which agree with those of *svm*^{light}. A larger C parameter is used in this test to see if the SVM will be stringent with outliers, those data which are very close together, but belong to different classes. With the larger C parameters, the margin narrows and the errors of those outliers are eliminated. The reclassification result indicates no misclassification. Two errors were placed in the region of the class 1 data. In both tests, the trained SVM is able to generalize well. Worth mentioning here is that when applying testFile2_1 to *svm*^{light} with a C parameter of 4.5, *svm*^{light} does not behave consistently. In some tests, it kept oscillating and did not stop and the program ran very slow. The data in testFile2_1 is distributed in a way such that there are two rows of class 1 labelled data running almost parallel to each other with one row located very close to another parallel row of class -1 labelled data. When the C parameter is not very large, a SVM tries to optimize with a larger margin. However, this means the row of class 1 data close to the row of the class -1 data will be misclassified. *svm*^{light} oscillates between those two rows of class 1 data, unable to decide which one to take.

The training result of **smoLearn** in learning using a polynomial kernel deviates from *svm*^{light}'s by 0.046%. The reclassification and classification results of **smoClassify** agree with that of *svm*^{light}.

The RBF kernel function used in **smoLearn** is

$$\exp^{-\frac{||x-y||^2}{2\sigma^2}}$$

and the RBF kernel function used in *svm*^{light} is

$$\exp^{-\frac{||x-y||^2}{\sigma^2}}$$

In order to compare my results with *svm*^{light}'s, the test with testFile4_1 was conducted with RBF kernel function in my implementation changed to match that of *svm*^{light}. The result deviates from *svm*^{light} by 0.47%. The reclassification and classification result of **smoClassify** agrees with *svm*^{light}.

5.2 Testing with benchmark data

The **smoLearn** program was also tested on two sets of benchmarks: the UCI Adult benchmark set and a web classification task. These benchmark data can be downloaded from John Platt's webpage `http://research.microsoft.com/~jplatt`. Each test set in these benchmarks is in .dst file format. A description of the .dst format can be found in the README file at Platt's website. The Python script "dst2mySmo.py" converts these data sets to the format which can be read by **smoLearn** and *svm*^{light} written by Joachims. The machine used in the tests was "vial", a Pentium 450MHz computer running Linux in the McGill computer lab R.105n. The tests were also conducted on my laptop computer, a Pentium 233 MHz CPU machine using Linux. The timings were about double those obtained using "vial".

5.2.1 The UCI Adult benchmark data set test results

The UCI Adult benchmark has 14 attributes which make up a census form of a household. Eight of the 14 attributes are categorical and six are continuous. The six continuous attributes were discretized into quintiles, which then gave a total of 123 binary attributes. A SVM was trained on these data and was then given a census form for predicting if the household earns more than 50,000 dollars. There are 9 data sets in the Adult benchmark set. Two sets of tests were conducted using the first 6 data sets of the adult benchmark. The first set of tests were conducted by training a linear SVM with $C = 0.05$. The second set of tests were conducted by training a Gaussian SVM with $C = 1$, and a Gaussian variance of 10. The test results of **smoLearn** and Platt's SMO [13] on the adult benchmark are shown in Tables 5.6 through 5.9. Table 5.10 compares the results of **smoLearn** with Platt's published data. It lists the deviation of my results from Platt's published results.

Training Set Size	smoLearn Time CPU sec	Threshold b	Non-Bound SVs	Bound SVs	smoLearn Iterations
1605	22	0.884539	41	634	3423
2265	51	1.125423	50	928	4691
3185	108	1.177335	58	1211	6729
4781	286	1.247689	62	1791	8704
6414	526	1.278382	67	2368	11315
11221	1922	1.324703	73	4085	19030

Table 5.6: smoLearn for a linear SVM on the Adult data set, $C = 0.05$

Training Set Size	smoLearn Time CPU sec	Threshold b	Non-Bound SVs	Bound SVs	smoLearn Iterations
1605	18	0.429848	107	585	3198
2265	42	0.295591	166	846	5259
3185	88	0.244956	187	1111	7277
4781	248	0.260752	236	1651	9976
6414	450	0.171952	310	2182	13529
11221	1545	0.089810	489	3720	27114

Table 5.7: smoLearn for a Gaussian SVM on the Adult data set, $C = 1$, $\sigma^2 = 10$

Training Set Size	Platt's SMO Time CPU sec	Threshold b	Non-Bound SVs	Bound SVs	smoLearn Iterations
1605	0.4	0.884999	42	633	3230
2265	0.9	0.12781	47	930	4635
3185	1.8	1.17302	57	1210	6950
4781	3.6	1.24946	63	1791	9847
6414	5.5	1.26737	61	2370	10669
11221	17.0	1.32441	79	4039	17128

Table 5.8: Platt's SMO for a linear SVM on the Adult data set, $C = 0.05$

Training Set Size	Platt's SMO Time CPU sec	Threshold b	Non-Bound SVs	Bound SVs	smoLearn Iterations
1605	15.8	0.428453	106	585	3349
2265	32.1	0.28563	165	845	5149
3185	66.2	0.243414	181	1115	6773
4781	146.6	0.2599	238	1650	1082
6414	258.8	0.159936	298	2181	14832
11221	781.4	0.0901603	460	3746	25082

Table 5.9: Platt's SMO for a Gaussian SVM on the Adult data set, $C = 1$, $\sigma^2 = 10$

Test data name	Linear Kernel		RBF Kernel	
	Difference	Percentage	Difference	Percentage
adult-1a	-0.00046	0.05	+0.001395	0.32
adult-2a	-0.002387	0.21	+0.009961	3.49
adult-3a	+0.004315	0.37	+0.001542	0.63
adult-4a	-0.001771	0.14	+0.000852	0.32
adult-5a	+0.011012	0.86	+0.012016	7.5
adult-6a	+0.000293	0.22	-0.0003503	0.39

Table 5.10: Deviation of smoLearn's results from Platt's published result in Adult data set

Discussion of the tests on the adult data set

The tests were conducted on a Pentium 450MHz CPU machine running Linux. No dot product cache was employed in these tests although a version of the project codes using dot product cache was written and tested successfully with artifical data sets created using my demo.py program and an adult-1a and adult-2a test sets. The decision to forego dot product cache in testing with large benchmark data is based on the following reasons. Dot product cache for large data set of over 3000 requires at least 144 MBytes of memory. Not many PCs are equipped with this large amount of memory. With the dot product cache incorporated into the codes, the time of the test run on adult-1a using a laptop equipped with 32 MBytes memory is far worse than that without dot product cache. It took hours to run on adult-3a because of the constant memory paging. Moreover, the Unix system function clock() which was used in obtaining the CPU time gives very large number in CPU time which apparently includes the disk access time during page swapping. The same tests with the version of the codes (before code fine tuning and adding an extra hierarchy to the second heuristic) written to incorporate dot product cache were run on a Celeron 300 MHz machine with 64 MBytes. The timings were 5 sec on the adult-1a and 18 sec on adult-2a. However, the test on adult-3a jumped to 639 sec because of the page swapping problem. The rest of the data sets then took hours to run. Dot product cache does improve the speed by a great deal provided the machine has a lot of memory. Doing away with dot product cache on an ordinary PC makes more sense. Also, Platt's tests were run without any dot product cache.

Platt's tests were conducted on an unloaded 266 MHz Pentium II processor running Windows NT4. His codes were written in Microsoft's Visual C++ 5.0. **smoLearn** and **smoClassify** were written in C and compiled on a gcc compiler under OpenLinux 2.3. As the goal of the project is to make sure that the SMO algorithm is implemented correctly, speed is not a focus in this report. There are many issues which complicate the performance comparison. Apart from platforms and processors, different optimization methods, fine tuning, data structures and some variation in the algorithm can affect the performance a great deal. At the very beginning of the testing phase, tests were run on a Pentium 233 MHz laptop. The CPU time which covered the execution of the entire SMO algorithm, including kernel evaluation time (excluding file I/O time) was 96 sec. on adult-1a, 247 sec on adult-2a and 462 sec on adult-3a. After fine tuning the codes, changing the dot product calculations to take advantage of sparse matrices by doing addition instead of multiplication, and adding another hierarchy to the second heuristic of Platt's algorithm, the CPU time on the laptop was cut down to 46 sec on adult-1a, 98 sec on adult-2a and 199 sec on adult-3a. Fine tuning the codes improves performance drastically. However, the choice of data structures can affect the performance as well. As mentioned in chapter 4, the software development of the project, very simple data structures - arrays were chosen to store all necessary data and quicksort and binary search were used to sort the errors of unbound examples because simple data structures make verifying and testing the programs easier. A run of **smoLearn** using the utility gprof showed that over 87% of the time is spent in the dot product calculations. The quicksort and binary searches in the codes account for only about 1 to 2% of the CPU time. Almost 12% of the time is spent in the calculateError() function of the learn module. This function in turn calls the dotProduct() function. Without knowing how Platt implemented his codes and the data structures he used, it is difficult to compare the performance, other than the big O performance.

The performance of the SMO algorithm is in the order between $O(n)$ and $O(n^2)$. Table 5.6 and Table 5.7 show

Web Data Set	Class Label 1	Class Label -1	fraction(%)OF No. OF Class -1 In Total
web-1a	201	6	2.90
web-2a	276	10	3.50
web-3a	394	14	3.43
web-4a	587	19	3.14
web-5a	805	26	3.13
web-6a	1389	56	3.88

Table 5.11: Distribution of examples with all feature values equal zero labelled with both class 1 and -1

that the big O performance of the implementation is as one would expect from the algorithm.

Table 5.7 shows a very interesting result. The performance of the Gaussian SVM on the 6 web tests are only on the average 1.5 times slower than Platt's implementation, but the Linear SVM runs on the average 76 times slower than Platt's implementation. Comparing Table 5.6 and Table 5.7 we see that the Gaussian SVM runs faster than the linear SVM on **smoLearn** whereas the Gaussian SVM on Platt's implementation runs slower than its linear counterpart. An integer array **nonZeroFeature[]** is used to store the number of non-zero feature each training example has. For training data with binary feature values, the dot product of an example with itself is equivalent to the number of non-zero features that example has. In order to save memory, only the id of the non-zero valued feature needs to be stored. In the Gaussian kernel evaluation, the value for the self dot product of an example was read from the **nonZeroFeature[]** array. As pointed out above, the dot product calculations dominate the time of the whole algorithm. Any way to speed up the dot product calculations will speed up the algorithm tremendously. This also indicates that the internal data structures used can affect the performance.

The number of iterations depends on the details on the algorithm implementation as well as on the epsilon chosen. The epsilon used in this implementation is the machine epsilon. The number of iterations do not seem to affect the run time a lot.

The threshold values, *b*, obtained in the linear SVM differ less than 1% from Platt's published values. The threshold values, *b*, obtained in the Gaussian SVM differ from Platt's values by an average of 2.1%. The number of bound and non-bound support vectors found depend on the orientation of the separating plane which is reflected in the threshold value *b*. Both table 5.6 and table 5.7 show that the number of bound and non-bound support vectors are close to Platt's values.

5.2.2 The Web benchmark data set test results

This benchmark set contains training data for a web page categorization and can be downloaded from John Platt's webpage `http://research.microsoft.com/~jplatt`.

The data are in .dst data format and they were converted by the Python script "dst2mySmo.py" to the file format which can be read by **smoLearn** and svm^{light}. 300 keywords are used as binary attributes of the web pages to be trained. This benchmark set differs from the adult benchmark in having contradictory training examples which have zeros for all features and were labelled both class '1' and class '-1'. Table 5.11 shows the distribution of these contradictory cases in each of 6 test sets of the Web benchmark. The fraction(%) column of table 5.11 represents the fraction(%) of the examples with all features zero which are labelled class -1. Tables 5.12, 5.13, 5.14, 5.15 and 5.16 show the **smoLearn** results on the web test sets, Platt's SMO results and the deviations of **smoLearn**'s results from those of Platt.

Discussion of the tests on the Web data set

There are contradictory examples in the web data sets. In this implementation, the contradictory examples are read

Training Set Size	smoLearn Time CPU sec	Threshold b	Non-Bound SVs	Bound SVs	smoLearn Iterations
2477	19	1.108767	122	44	35301
3470	29	1.138071	148	70	46775
4912	63	1.094352	173	105	68645
7366	155	1.087816	181	170	125051
9888	324	1.078966	204	249	197254
17188	1393	1.029931	228	476	352755

Table 5.12: smoLearn for a linear SVM on the Web data set, $C = 1$

Training Set Size	smoLearn Time CPU sec	Threshold b	Non-Bound SVs	Bound SVs	smoLearn Iterations
2477	25	0.190476	301	43	13401
3470	54	0.034370	379	68	20181
4912	137	0.017607	466	90	36895
7366	218	0.027240	546	127	40560
9888	431	-0.045497	715	169	50591
17188	1684	-0.190532	969	335	104640

Table 5.13: smoLearn for a Gaussian SVM on the Web data set, $C = 5$, $\sigma^2 = 10$

Training Set Size	Platt's SMO Time CPU sec	Threshold b	Non-Bound SVs	Bound SVs	smoLearn Iterations
2477	2.2	1.08553	123	47	25296
3470	4.9	1.10861	147	72	46830
4912	8.1	1.06354	169	107	66890
7366	12.7	1.07142	194	166	88948
9888	24.7	1.08431	214	245	141538
17188	65.4	1.02703	252	480	268907

Table 5.14: Platt's SMO for a linear SVM on the Web data set, $C = 1$

Training Set Size	Platt's SMO Time CPU sec	Threshold b	Non-Bound SVs	Bound SVs	smoLearn Iterations
2477	26.3	0.177057	439	43	10838
3470	44.1	0.0116676	544	66	13975
4912	83.6	-0.0161608	616	90	18978
7366	156.7	-0.0329898	914	125	27492
9888	248.1	-0.0722572	1118	172	29751
17188	581.0	-0.19304	1780	316	42026

Table 5.15: Platt's SMO for a Gaussian SVM on the Web data set, $C = 5$, $\sigma^2 = 10$

Test data name	Linear Kernel		Rbf Kernel	
	Difference	Percentage	Difference	Percentage
web-1a	+0.023237	2.14	+0.013419	7.58
web-2a	+0.029461	2.66	+0.0227024	194.58
web-3a	+0.030812	2.9	+0.0337678	209.5
web-4a	+0.016396	1.52	+0.0602298	182.57
web-5a	-0.005344	0.49	+0.0267602	37.03
web-6a	+0.002901	0.28	+0.002509	1.30

Table 5.16: Deviation of smoLearn's results from Platt's published result in Web benchmark

in together with the rest of the examples and training was done using those contradictory examples. The examples labelled as class 1 in the training set when they have zeros for all the features account for an average of 3.4% of all the examples with the same characteristics (see Table 5.11). The rest of the examples with the same characteristics are labelled as class '-1'. We can consider those 3.4% as errors. Normally, this would mean that there is deficiency in the feature extracting method and the trainer has to revise his/her feature extraction method in order to eliminate the contradictory cases. He/she might add additonal features for those contradictory examples. The machine should be very general and leave the feature extraction responsibility to the trainer. Other implementations might handle the contradictory cases differently. Without knowing how Platt handled the contradictory cases, it is hard to compare **smoLearn's** results with his. Investigation was made by adding additional features to eliminate the contradiction. The results are very much different from Platt's data. This is not a correct approach. A machine should not have any a priori knowledge of the data. Experiment was conducted by incorporating the majority of those samples which have all feature values zero and are labelled as class '-1' and ignoring the small percentage which are labelled as class '1'. The result shows that the trained SVM is able to generalize well in reclassification. The closest results to Platt's which are shown in tables 5.12 and 5.13 were obtained by training the SVM on the contradictory cases. In light of the above, the following discussion will focus on analyzing the results rather than comparing them with Platt's.

Since there is always roughly the same small percentage of those contradictory cases in all 6 test sets, we can assume they are noises. Table 5.12 shows that all the examples with zero values for all the features will be classified as class '-1' by the trained SVM because the thresholds are positive. Those small percentages of contradictory examples labeled as class '1' in the training are errors and they are bound support vectors in the training. Support Vector Machine can generalize well when trained with noisy data.

Table 5.13 shows a different result from that of the linear SVM. The threshold b for test sets web-5a and web-6a are negative. If we use those two trained SVMs to classify examples with all feature values zero, they will be classified as class '1' even though in our training sample, 97% of examples with this characteristics are labelled class '-1' in the training set. If we consider those approximately 3% of contradictory examples in the training sample as noise, then this noise obscures the larger percentage of examples in the training having the same characteristic but are labelled class '-1'. The trained SVM cannot generalize well in the reclassification.

Table 5.15 shows that in Platt's SMO implementation, the threshold b becomes negative in the Gaussian SVM starting from test set web-3a. How an implementation of the SVM handles the contradictory cases will affect the error rate of classification of the resulting machine.

Perhaps the most immediate question would be why it is that the linear SVM behaves differently from the Gaussian SVM. The kernel function is the function of distance that is used to determine the weight of each training example. In a linear kernel, the distance is Euclidean distance. A Gaussian SVM is equivalent to a Radial Basis Function classifier [17]. The distance function of a Radial Basis function classifier in my implementaton is

$$\exp^{-\frac{d^2(\mathbf{x_i},\mathbf{x})}{2\sigma_i^2}} \tag{5.1}$$

where $\mathbf{x_i}$ is an instance from the training set and $d(\mathbf{x_i}, \mathbf{x})$ is the distance of $\mathbf{x_i}$ from $\mathbf{x}$.

The distance function is a Gaussian function centered at $\mathbf{x_i}$ with some variance σ_i^2. As the distance between $\mathbf{x_i}$ and $\mathbf{x}$ increases, the kernel function value decreases. There are many ways to implement a RBF classifier. One can find the set of kernel functions spaced uniformly throughout the instance space or find the clusters of instances and then put a kernel function centered at each cluster [10]. The Gaussian SVM in this implementation is equivalent to a radial basis function classifier because it uses equation 5.1. The SVM finds the support vectors which are used as centers for the kernel functions. Each kernel function has the same variance. This implementation of SVM includes those contradictory cases in the training and these cases will always be errors and be bound support vectors. In a Gaussian SVM, these errors become the centers of the kernel functions. If the distribution of instances in the training sample clusters closely to the contradictory instances, then those support vectors will exert more weights and affect the classification error rate. In the case of web-5a and web-6a, even though a big percentage of examples with all their features zeros are labelled class '-1' in the training sample, they will be classify as class 1 in the reclassifcation (classification with the training sample).

I think and think for months and years. Ninety-nine times, the conclusion is false. The hundredth time I am right.

— Albert Einstein, 1879-1955 —

6 Conclusion

The test results using the artificial data and benchmarks show that the implementation of a Support Vector Machine through Sequential Minimal Optimization algorithm has been carried out correctly. There are many issues arising during the implementation, issues such as performance, optimization, handling of contradictory cases in training a benchmark. Perhaps the most important result of this project was the understanding gained through the implementation.

Different learning algorithms have different ways to handle examples with all their features zero as well as the contradictory cases. Since the underlying structure of SVM involves solving a kernel matrix problem, the decision was made to incorporate the examples which have all their feature values zero and let the kernel matrix be semidefinite. There are two issues here. The first issue is to allow the kernel matrix to be semidefinite. Support vector machine trained with a semidefinite kernel is able to generalize well. The second issue is to train the SVM with the contradictory cases included. The result shows that special care has to be taken to deal with such cases. They represent the boundary between two classes. Table 8 shows that depending on how the implementation handles the contradictory cases, large classification errors could result.

A dot product cache does improve the performance of the SMO algorithm. However one has to weigh the benefit against the large amount of memory required. svm^{light} uses cache in its implementation and training svm^{light} on the Gaussian kernel using data from web-3a took hours on my laptop with only 32 MB. Training a Gaussian SVM on the Web benchmarks without a dot product cache on my laptop with **smoLearn** took from 60 sec for web-1a to 919 sec for web-5a. Since most personal computers do not have a lot of memory, improving the performance of the algorithm through code optimization, modification of the heuristic used in choosing the second example for joint optimization, optimization of the dot product calculation and error updating in the implementation are more meaningful.

smoLearn has proved to be very robust. A 2-dimensional artificial data set which has a very close margin and can be linearly separated with errors (see section 5.1, test with testFile2_1) was designed and tested with both svm^{light} and **smoLearn** using a C parameter of 4.5. **smoLearn** ran through the training in less than 1 second with 26 correct and 7 incorrect in reclassifcation and 100% correct on classification with the test data. svm^{light} did not run consistently. On some occasions it kept oscillating and the algorithm ran very slowly and did not even stop.

Since SMO is a training algorithm, most of the tests were done on the training part of the software **smoLearn** and the benchmark tests verify the correctness of the program. The classification part of the software, **smoClassify**, was done on artificial data only. More investigation into speeding up the classification can be done in the future.

Currently a lot of researchers use *svm*^{light} for their research works. You might ask why one should go through the trouble of implementing his/her own SVM when one can download a free package from the web site of *svm*^{light}. My experience of this project convinces me that the benefit of implementing one's own SVM far outweighs the time and effort invested in the implementation.

SVM is still a relatively new subject. Books on SVM just started to appear in 1999. Still, a lot of work, both theoretical and practical needs to be done in SVM. SVM is very mathematically involved. There is no better way to gain a better understanding of all the mathematics and theory behind SVM than to learn through the implementation and experimentation. Moreover, there are cases when *svm*^{light} fails to function properly while my implementation proceeds without problems and gives good classification results. Another important advantage of implementing one's own SVM is the ability to customize codes to specific applications. SVM implemented using the SMO algorithm has proven to be very efficient in applications which use sparse matrices [4]. If the software one downloads does not have good documentation, then it is very hard to customize the codes to one's need. There is also the advantage of being able to maintain, improve one's implementation and experiment with ideas as one gains more knowledge in SVM. The most attractive part of the SMO algorithm is its ability to do away with the need for Quadratic Programming and still give good performance.

Bibliography

[1] C. Burges, B. Schölkopf and V. Vapnik. Extracting support data for a given task. *Proceedings First International Conference on Knowledge Discovery and Data Mining.* AAAI Press, Menlo Park, 1995.

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

[3] H. Drucker, C. Burges, L. Kaufman, A. Smola and V. Vapnik. Support Vector Regression Machines. *Advances in Neural Information Processing Systems*, 9:155-161, 1997.

[4] S. Dumais. Using SVMs for text categorization. *IEEE Intelligent Systems.* July/August, pp.21-23, 1998.

[5] J. Eiker, M. Kupferschmid. *Introduction to Operations Research.* John Wiley & Sons, New York, 1988.

[6] I. Guyon. SVM Application List. http://www.clopinet.com/isabelle/Projects/SVM/applist.html.

[7] T. Joachims. Text categorization with Support Vector Machines: learning with many relevant features. *Proc. 10th European Conferencr Machine Learning ECML*, Springer-Verlag, New York, 1998.

[8] P. Langley. *Elements of Machine Learning.* Morgan Kaufmann, San Francisco, 1988.

[9] D. Matterra and S. Haykin. Support Vector Machines for dynamic reconstruction of a chaotic system. *Advances in Kernel Methods Support Vector Machine* pp211-241, MIT Press, Cambridge, 1999.

[10] T. Mitchell. *Machine learning.* McGraw-Hill, New York, 1997.

[11] D. Opitz. An empirical evaluation of Bagging and Boosting. *Fourteenth National Conference on Artificial Intelligence (AAAI)*, Providence, Rhode Island, 1997. http://www.cs.umt.edu/CS/FAC/OPITZ/papers/aaai97.html

[12] E. Osuna, R. Freund and F. Girosi. Support vector machines: training and applications. *MIT AI Memo 1602*, 1997.

[13] J. Platt. Fast training of SVMs using Sequential Minimal Optimization. *Advances in Kernel Methods Support Vector Machine*, pp185-208, MIT Press, Cambridge, 1999.

[14] M. Pontil and A. Verri. Properties of Support Vector Machines. *MIT AI Memo 1612*, 1998.

[15] D. Roobaert. Improving the generalisation of Linear Support Vector Machines: an application to 3D object recognition with cluttered background. *Proc. Workshop on Support Vector Machines at the 16th International Joint Conference on Artificial Intelligence*, pp29-33, July 31-August 6, Stockholm, Sweden, 1999.

[16] M. Rychetsky, S. Ortmann and M. Glesner. Support Vector approaches for engine knock detection. *International Joint Conference on Neural Networks(IJCNN 99)*, Washington, USA, July, 1999.

[17] B. Schölkopf, K. Sung, C. Burges, F. Girosi, P. Niyogi, T. Poggio and V. Vapnik. Comparing Support Vector Machines with Gaussian kernels to Radial Basis Function classifiers. *IEEE Transactions on Signal Processing*, vol. 45, no.11, Novemember 1997.

[18] B. Schölkopf, C. Burges and A. Smola. *Advances in Kernel Methods Support Vector Machine*. MIT Press, Cambridge, 1999.

[19] P. Vannerem, K. Müller, B. Schölkopf, S. Smola and S. Söldner-Rembold. Classifying LEP data with Support Vector algorithm. *Proceedings of AIHENP'99*, 1999.

[20] V. Vapnik. *The Nature of Statistical Learning Theory*. Springer-Verlag, New York, 1995.

[21] A. Zien, G. Rätsch, S. Mika, B. Schölkopf, C. Lemmen, A. Smola A, T. Lengauer and K. Müller. Engineering Support Vector Machine kernels that recognize Translation Initiation Sites. *German Conference on Bioinformatics*, 1999. http://www.bioinfo.de/isb/gcb99/talks/zien

A

Mathematics Review

The general equation of a plane in d dimensions is

$$\mathbf{w} \cdot \mathbf{x} = k$$

where $\mathbf{w}$ is the normal to the hyperplane, and $\mathbf{x}$ is a $d \times 1$ vector, and k is a scalar constant.
$\frac{|k|}{\|\mathbf{w}\|}$ is the minimum distance from the origin to the plane.

The following are a summary from the book "Introduction to operations research" by Eiker J, and Kuperferschmid M. (1988)

Matrices

$\mathbf{x}$ is a d-dimensional vector.
$f(\mathbf{x})$ is a scalar-valued function of $\mathbf{x}$.
The derivative or gradient of f with respect to $\mathbf{x}$ is

$$\nabla f(\mathbf{x}) = grad\, f(\mathbf{x}) = \frac{\partial f(\mathbf{x})}{\partial \mathbf{x}} = \begin{pmatrix} \frac{\partial f(\mathbf{x})}{\partial x_1} \\ \frac{\partial f(\mathbf{x})}{\partial x_2} \\ \cdot \\ \cdot \\ \cdot \\ \frac{\partial f(\mathbf{x})}{\partial x_d} \end{pmatrix}.$$

The Hessian matrix $H(\mathbf{x})$ is defined as follows:

$$H(\mathbf{x}) = \begin{pmatrix} \frac{\partial^2 f(\mathbf{x})}{\partial x_1^2} & \frac{\partial^2 f(\mathbf{x})}{\partial x_1 \partial x_2} & \cdot & \frac{\partial^2 f(\mathbf{x})}{\partial x_1 \partial x_n} \\ \frac{\partial^2 f(\mathbf{x})}{\partial x_2 \partial x_1} & \frac{\partial^2 f(\mathbf{x})}{\partial x_2^2} & \cdot & \frac{\partial^2 f(\mathbf{x})}{\partial x_2 \partial x_n} \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \frac{\partial^2 f(\mathbf{x})}{\partial x_n \partial x_1} & \frac{\partial^2 f(\mathbf{x})}{\partial x_n \partial x_2} & \cdot & \frac{\partial^2 f(\mathbf{x})}{\partial x_n^2} \end{pmatrix}.$$

A matrix M is any $n \times n$ matrix and z is a vector of length n:
M is positive definite $\iff z^T Mz$ is $> 0, \forall z \neq 0$
M is positive semidefinite $\iff z^T Mz$ is $\geq 0, \forall z \neq 0$
M is negative definite $\iff z^T Mz$ is $< 0, \forall z \neq 0$
M is negative semidefinite $\iff z^T Mz$ is $\leq 0, \forall z \neq 0$

$\mathbf{x}$ is a stationary point of $f(\mathbf{x})$ if $\frac{\partial f(\mathbf{x})}{\partial \mathbf{x}} = 0$
If $\mathbf{x}$ is a stationary point of $f(\mathbf{x})$, then:
$H(\mathbf{x})$ is positive definite $\longrightarrow \mathbf{x}$ is a strict minimizing point
$H(\mathbf{x})$ is positive semidefinite $\forall x$ in some neighborhood of $x \longrightarrow \mathbf{x}$ is a strict minimizing point
$\mathbf{x}$ is a minimizing point $\longrightarrow H(\mathbf{x})$ is positive semidefinite

$H(\mathbf{x})$ is negative definite $\longrightarrow \mathbf{x}$ is a strict maximizing point
$H(\mathbf{x})$ is negative semidefinite $\forall \mathbf{x}$ in some neighborhood of $\mathbf{x} \longrightarrow \mathbf{x}$ is a strict maximizing point
$\mathbf{x}$ is a maximizing point $\longrightarrow H(\mathbf{x})$ is negativesemidefinite

The Lagrange Multiplier Theorem

Given a nonlinear programming problem
minimize $f(\mathbf{x})$ $\mathbf{x} \in R^n$
subject to

$$g_i(\mathbf{x}) = 0\,, \qquad i = 1, \ldots, m.$$

If $\mathbf{x}$ is the local minimizing point for the nonlinear programming problem, and $n > m$
and g_i have continuous first partial derivatives with respect to the x_i,
and the $\nabla g_i(\mathbf{x})$ are linearly independent vectors,
then there is a vector $\lambda = [\lambda_1, \lambda_2, \ldots, \lambda_m]^T$ such that

$$\nabla f(\mathbf{x}) + \sum_{i=1}^{m} \lambda_i \nabla g_i(\mathbf{x}) = 0.$$

The numbers λ_i are called Lagrange multipliers and they can be positive, negative or zero.

Solving equality constrained nonlinear program by the Lagrange Multiplier Theorem

1. Form the Lagrangian function

$$L(\mathbf{x}, \lambda) = f(\mathbf{x}) + \sum_{i=1}^{m} \lambda_i g_i(\mathbf{x})$$

2. Convert the constrained optimization into an unconstrained problem of finding the extremum of $L(\mathbf{x}, \lambda)$ by taking the derivatives

$$\frac{\partial L(\mathbf{x}, \lambda)}{\partial \lambda_i} = g_i(\mathbf{x}) = 0, \qquad \text{i=1,...,m,}$$

$$\frac{\partial L(\mathbf{x}, \lambda)}{\partial x_j} = 0, \qquad \text{j = 1, ..., n.}$$

Inequality-constrained non-linear problem

We want to ignore the inactive constraints in the Lagrangian equation and convert the problem to an equality problem.

$$L(\mathbf{x},\lambda) = f(\mathbf{x}) + \sum_{i=1}^{m} \lambda_i g_i(\mathbf{x}).$$

We can exclude inactive constraints by requiring the Lagrange multipliers corresponding to that slack inequality be zero.
We want λ_i to be zero when $g_i(\mathbf{x}) \neq 0$ and λ_i to be non-zero when $g_i(\mathbf{x})$ is zero where $\mathbf{x}$ is the optimal point. The following orthogonality condition can ensure that the Lagrangian function will ignore inactive constraints and incorportate active ones as if they were equalitiy ones. The orthogonality condition is:

$$\lambda_i g_i(\mathbf{x}) = 0.$$

λ is orthogonal to $g(\mathbf{x})$.
Enforcing the orthogonality condition ensures that the Lagrangian function L ignores inactive constraints and incorporate active ones as if they were equalities. Now the Lagrangian function for an inequality-constrained problem is indistinguishable from that for an equivalent equality-constrained problem involving only active constraints. However now the Lagrange multipliers must be > 0 at a minimizing point if a constraint qualification holds for an inequality-constrained problem. All these conditions for a system of nonlinear equations and inequalities constraints are called the Karush-Kuhn-Tucker conditions.

The Karush-Kuhn-Tucker (KKT) Conditions

Given the canonical form of a nonlinear programming problem
minimize $f(\mathbf{x})$, $\mathbf{x} \in R^n$
subject to

$$g_i(\mathbf{x}) \leq 0 \qquad \text{i = 1,...,m.}$$

If the Lagrangian function is given by

$$L(\mathbf{x},\lambda) = f(\mathbf{x}) + \sum_{i=1}^{m} \lambda_i g_i(\mathbf{x})$$

then the KKT conditions for the problem are:

$\frac{\partial L}{\partial x_j} = 0$	$j = 1,\ldots,n$	**gradient condition**
$\lambda_i g_i(\mathbf{x}) = 0$	$i = 1,\ldots,m$	**orthogonality condition**
$g_i(\mathbf{x}) \leq 0$	$i = 1,\ldots,m$	**feasibility condition**
$\lambda_i \geq 0$	$i = 1,\ldots,m$	**nonnegativity**

The KKT conditions are the conditions that are used to solve problems with inequality constraints.

How to tell if a point $(\mathbf{x},\lambda)$ which satisfies the KKT condition is a maximum, minimum or neither ?
Convexity can be used to decide if a KKT point is a minimizing point or not.

Convex functions

A function f is convex iff for all choices of points $\mathbf{x_1}$ and $\mathbf{x_2}$

$$f(\alpha\mathbf{x_1} + (1-\alpha)\mathbf{x_2}) \leq \alpha f(\mathbf{x_1}) + (1-\alpha) f(\mathbf{x_2})$$

for all α satisfying $0 \leq \alpha \leq 1$

A function f is *strictly convex* if each of the inequalities in the preceding definition is a strict inequality. A function is concave if $-f$ is a convex function.

We can use definiteness of the Hessian Matrix to test for convexity of a function.
$H(\mathbf{x})$ is positive semidefinite $\forall$ values of $\mathbf{x} \iff f(\mathbf{x})$ is a convex function.

$H(\mathbf{x})$ is positive definite $\forall$ values of $\mathbf{x} \rightarrow f(\mathbf{x})$ is a strictly convex function.

If a function is convex, then the stationary point must be a minimizing point.
If we add the requirement that the objective and constraint functions be convex functions, then the solution of the KKT condition gives the global minimizing point. KKT Theorem sums up the necessary and sufficient conditions for a local minimizing point to be a global minima.

KKT Theorem

Given the canonical form nonlinear programming problem
minimize $f(\mathbf{x})$, $\mathbf{x} \in R^n$
subject to

$$g_i(\mathbf{x}) \leq 0 \qquad \text{i = 1,...,m.}$$

The necessary conditions for a local minimizing point to satisfy KKT condition are:
if

- f, g_i are differentiable, i = 1,...,m.
- $\mathbf{x}$ is a local minimizing point, and
- a constraint qualification holds

then
there is a vector λ such that $(\mathbf{x}, \lambda)$ satisfies the KKT conditions;

The sufficient conditions for a point satisfying KKT condition to be a global minimizing point are:
if

- $(\mathbf{x}, \lambda)$ satisfies the KKT conditions
- f, g_i are convex functions, i = 1,...,m

then
$\mathbf{x}$ is a global minimizing point.

The KKT Method for solving inequality-constrained non-linear programming problems

1) Form the Lagrangian function

$$L(\mathbf{x}, \lambda) = f(\mathbf{x}) + \sum_{i=1}^{m} \lambda_i g_i(\mathbf{x})$$

2) Find all solutions $(\mathbf{x}, \lambda)$ to the following systems of nonlinear algebraic equations and inequalities

$\frac{\partial L}{\partial x_j} = 0$	$j = 1, \ldots, n$	**gradient condition**
$g_i(\mathbf{x}) \leq 0$	$i = 1, \ldots, m$	**feasibility**
$\lambda_i g_i(\mathbf{x}) = 0$	$i = 1, \ldots, m$	**orthogonality**
$\lambda_i \geq 0$	$i = 1, \ldots, m$	**nonnegativity**

3) If f and $g_i(\mathbf{x})$ are all convex, then the points $\mathbf{x}$ are global minimizing points.
Otherwise examine each $(\mathbf{x}, \lambda)$ to see if $\mathbf{x}$ is a minimizing point.

B

demo.py

demo.py is an interactive python program for demonstrating SVM and for testing the programs **smoLearn** and **smoClassify**. The following is a brief description of the use of the program. Figure B.1 shows the demonstration window of the program.

Training:

1. Select the **pos** or **neg** radio button.
   Clicking on the canvas with the left mouse button will create a training point with the label of +1 if **pos** is selected, -1 if **neg** is selected. Positive points are represented by filled blue circles. Negative points are represented by open circles.

2. Click on the **CREATE TRAINING FILE** button to open a dialog window for inputting the name of the training file.

3. Click on the **TRAIN** button to open a learning window similar to the one in Figure 4.6.

4. Select the kernel type and input the C parameter and other parameters according to the type of kernel selected. Input the model file name.

5. Click on the **OK** button to start training. Training results are displayed on the canvas at the end of the training. Margin support vectors are circled with black circles. Bound support vectors are circled with red circles. If the kernel type is linear, the optimal separating line, the line joining the positive margin support vectors and the line joining the negative margin support vectors are displayed as well. For the polynomial kernel, the user has to click on the **PLOT** button to have the corresponding lines displayed.

Additional options:

- A user can also click on the **LOAD EXISTING TRAINING FILE** button to load an existing training file, which was created by this **demo.py** program before.
- Clicking on the **CREATE POSTSCRIPT** button will create a postscript file of what is shown on the canvas.

Classification:

1. Select the **classify positive** or **classify negative** radio button. Clicking with the left mouse button will create a point labelled with +1 if **classify positive** is selected and a point labelled with -1 if **classify negative** is selected. The positive points are represented by filled yellow circles and the negative ones by filled green circles.

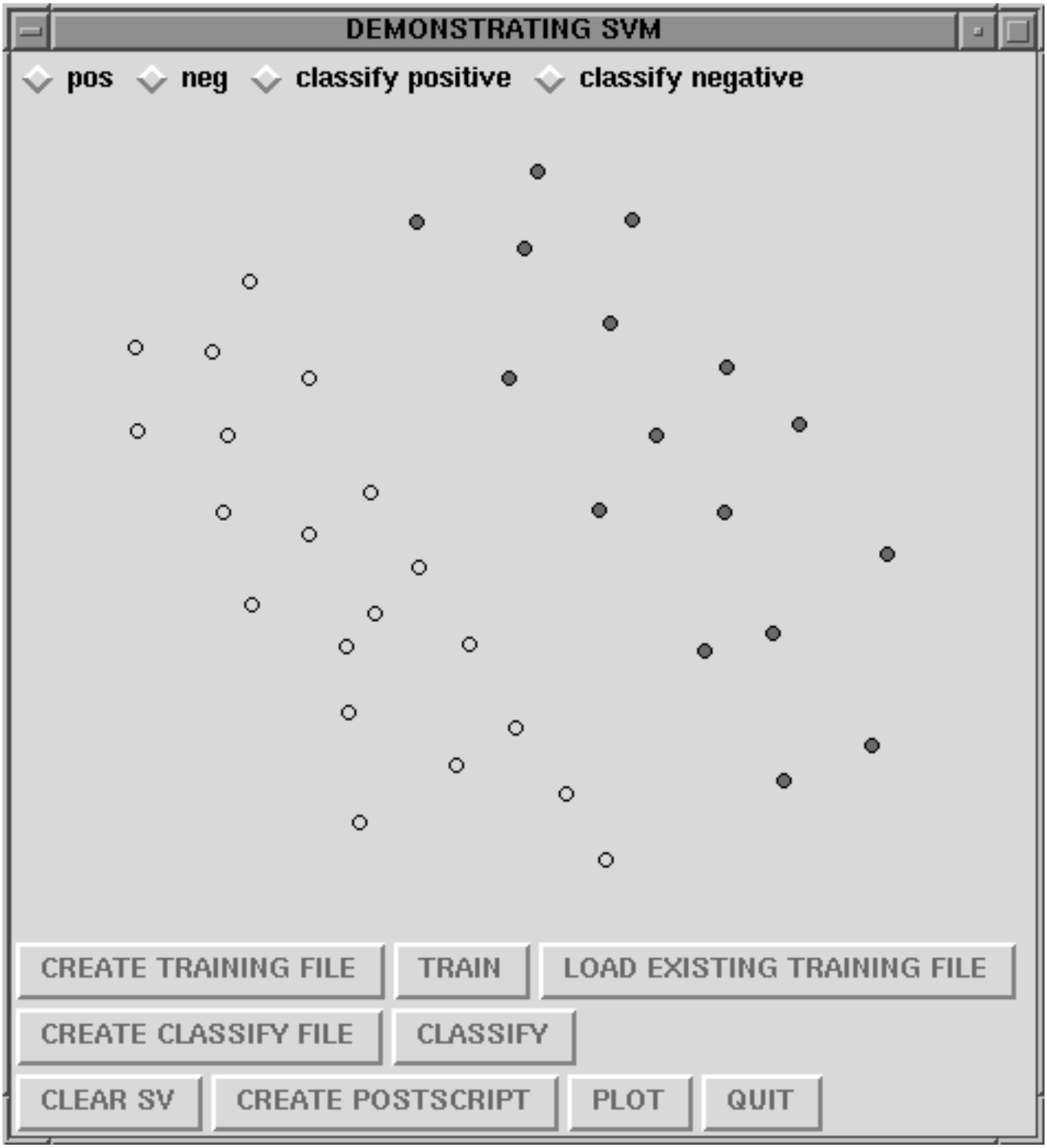


Figure B.1: Demonstration window of the program demo.py

2. Clicking on the **CREATE CLASSIFY FILE** will open a dialog window for inputting the test file name.

3. Clicking on the **CLASSIFY** button will open a dialog window for inputting the name of the prediction file name. Clicking the **OK** button in the dialog window will start the classification.

   At the end of the classification, the points which are classified as positive will be marked by yellow circles and those classified negative by green circles.
   Note: The model file is the model file which has been created after a training with the training points which are presently on the canvas.

Clicking on the **QUIT** button will close the demonstration window.

C
SMO codes

```
/********************************  smoLearn.c  ***************************

Purpose: To build an svm based on the given training data and the
         selected kernel function.

Notes: This program is composed of the following modules:
       smoLearn.c - main program module
       initializeTraining.c - read training data and initialize data structures
       learn.c - build an svm based on the given training data using
                 the SMO algo.
       utility.c - provide functions for the learning process.
       result.c - writing the training file in a model file

Programmer: Ginny Mak
Location: Montreal, Canada.
Last Updated: April 6, 2000.

**/

#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <ctype.h>
#include <float.h>
#include <getopt.h>
#include "initializeTraining.h"
#include "learn.h"
#include "result.h"


#define DEFAULT 4.5
#define EPS DBL_EPSILON
#define OPTSTRING "t:c:b:d:v:h"

char usage[] =
"\n\
General options:\n\
[-h]   help (this description of command line args)\n\
[-b]   Specify nature of feature value. 0 for nonbinary. 1 for binary \n\
```

```
      Default is 0.\n\
\n\
Required options:\n\
[-t]   type of kernel,0 for linear, 1 for polynomial, 2 for RBF\n\
[-c]   c parameter\n\
[-d]   degree of polynomial is needed if option [-t] is 1\n\
[-v]   variance of RBF function is need if option [-t] is 2\n\
\n";


void useMsg( char *progname ) {
  fprintf( stderr, "\nUse: %s options\n", progname );
  fprintf(stderr, "%s", usage );
}


int main( int argc, char **argv )
{
  char *progname = argv[0];
  FILE *in , *out;
  char opt;
  int i, j;
  double startTime;

  C = 0;
  kernelType = -1;
  degree = 0;
  sigmaSqr = 0;
  binaryFeature = 0;

  /*********************************

      Check command line options.

   *********************************/
  optarg = NULL;
  while (((opt = getopt(argc, argv, OPTSTRING)) != -1)) {
    switch(opt) {
    case 't':
      if ( strcmp(optarg, "0") == 0 )
        kernelType = 0;
      else if ( strcmp(optarg, "1") == 0 )
        kernelType = 1;
      else if ( strcmp(optarg, "2") == 0 )
        kernelType = 2;
      else {
        fprintf( stderr, "kernel type is either 0,1 or 2\n");
        exit(1);
      }
      break;
    case 'c':
      if( sscanf(optarg, "%f", &C) == 0 ) {
        fprintf(stderr,"Expect a positive number for C.\n");
        exit(1);
      }
      else
        C = atof(optarg);
      if( C <= 0 ) {
        fprintf( stderr, "C has to be > 0\n");
        exit(1);
      }
      break;
    case 'd':
```

```
      if( sscanf(optarg, "%d", &degree) == 0 ) {
        fprintf( stderr, "Expect degree to be a positive integer.\n");
        exit(1);
      }
      else
        degree = atoi(optarg);
      if ( degree <= 0 ) {
        fprintf( stderr, "degree has to be a positive integer.\n");
        exit(1);
      }
      break;
    case 'v':
      if( sscanf(optarg, "%f", &sigmaSqr) == 0 ) {
        fprintf(stderr,"Expect a positive number for variance.\n");
        exit(1);
      }
      else
        sigmaSqr = atof(optarg);
      if( sigmaSqr <= 0 ) {
        fprintf( stderr, "variance has to be > 0\n");
        exit(1);
      }
      rbfConstant = 1/(2*sigmaSqr);
      break;
    case 'b':
      if( sscanf(optarg, "%d", &binaryFeature) == 0 ) {
        fprintf( stderr, "binaryFeature option is either 0 or 1.\n");
        exit(1);
      }
      else
        binaryFeature = atoi(optarg);
      if ( binaryFeature != 0 && binaryFeature != 1 ) {
        fprintf( stderr, "binaryFeature option is either 0 or 1.\n");
        exit(1);
      }
      break;
    case 'h':
      useMsg( progname );
      exit(1);
      break;
    default:
      useMsg( progname );
      exit(1);
      break;
    }
}

/*****************************************

   Check all necessary parameters are in

 *****************************************/
if( kernelType == -1 ) {
  fprintf( stderr, "Kernel type has not been specified.\n");
  exit(2);
}
else if( kernelType == 1 && degree == 0 ) {
  fprintf( stderr, "Degree has not been specified.\n");
  exit(2);
}
else if( kernelType == 2 && sigmaSqr == 0 ) {
  fprintf( stderr, "Variance has not been specified.\n");
  exit(2);
```

```
}
else if( C == 0 )
  C = DEFAULT;

/*****************************************                                    170

    Check training file and model file

 *****************************************/
if (( in = fopen( argv[argc-2], "r") ) == NULL ) {
  fprintf( stderr, "Can't open %s\n", argv[argc-2] );
  exit(2);
}
if (( out = fopen( argv[argc-1], "w") ) == NULL ) {
  fprintf( stderr, "Can't open %s\n", argv[argc-1] );                        180
  exit(2);
}

printf("smo_learn is preparing to learn...\n");
if( ! readFile( in ) ) {
  fprintf( stderr, "Error in initializing. Program exits.\n" );
  exit (1);
}
else
  fclose( in );                                                              190

if( !initializeTraining()) {
  fprintf( stderr, "Error in initializing data structure. Program exits.\n");
  exit(1);
}

printf("Start training ...\n");
startTime = clock()/CLOCKS_PER_SEC;
startLearn();
printf("Training is completed\n");                                          200

/*******************************

   Print training statistics.

 *******************************/
printf("CPU time is %f secs\n", clock()/CLOCKS_PER_SEC - startTime);
printf("Writing training results ...\n");
writeModel( out );
fclose( out );                                                              210
printf("Finish writing training results.\n");
printf("no of iteration is %f\n", iteration);
printf("threshold b is %f\n", getb());
if ( kernelType == 0 )
  printf("norm of weight vector is %f\n", calculateNorm());
printf("no. of unBound multipliers is %d\n", unBoundSv );
printf("no. of bounded multipliers is %d\n", boundSv );

/*****************

                                                                            220
   Free memory

 *****************/
free( target );
free( lambda );
free( nonZeroFeature );
free( error );
free( nonBound );
```

```
  free( weight );
  free( unBoundIndex );
  free( nonZeroLambda );
  for( i = 0; i <= numExample; i++ ) {
    free( example[i] );
  }
  free( example );
  free( errorCache );

  return 0;
}
```

```
/*********************  initializeTraining.h  *******************************

 Purpose: Header for data structure initializing module.
 Notes: None.

 Programmer: Ginny Mak
 Location: Montreal, Canada.
 Last Updated: April 6, 2000.
**/

/*********************  Public defines and data structure ******************/

typedef struct feature{
  int id;
  float value;
} Feature;

typedef struct feature * FeaturePtr;

FeaturePtr ** example;

int *target;
double *lambda;
int *nonZeroFeature;
double *error;
int *nonBound;
double *weight;
int numExample;
int maxFeature;
int numNonBound;
int *unBoundIndex;
int dataSetSize;
int *errorCache;
int *nonZeroLambda;



/*************************  Public Functions  ******************************/

extern int readFile( FILE *in );
extern int initializeTraining( void );
```

```
/*****************************  initializeTraining.c  **********************

 Purpose: Initialize all data structures for training by reading information
          from the training data file.

 Notes: It is assumed that each line of input training file has a maximum of
        10000 characters.
```

```
 Programmer: Ginny Mak
 Location: Montreal, Canada.
 Last Updated: April 6, 2000.

**/

#include <stdio.h>
#include <stdlib.h>
#include <limits.h>
#include <math.h>
#include <float.h>
#include "initializeTraining.h"


/******************** Private Defines and Data Structures  *****************/

#define DATATEMP1 10000
#define DATATEMP2 1000
#define EPS DBL_EPSILON
#define MAXLINE LONG_MAX

static int exampleIndex;
static int featureIndex;

/***************************** Declared Functions  ************************/

static double dotProduct( FeaturePtr *x, int sizeX, FeaturePtr *y, int sizeY );
static int initializeDate( int size );


/*****************************  Private Functions  ***********************/

/*FN**************************************************************************

int initializeData ( int size )
Returns: int – Fail(0) on error, Succeed(1) otherwise.
Purpose: To initialize data structures for training.
Notes: None.
**/

static int initializeData( int size )
{
  example = (FeaturePtr **) malloc( size * sizeof( FeaturePtr * ) );
  if ( example == NULL ) {
    fprintf( stderr, "Memory allocation failed.\n");
    return 0;
  }
  target = (int *) malloc( size * sizeof( int ) );
  if ( target == NULL ) {
    fprintf( stderr, "Memory allocation failed.\n");

    return 0;
  }
  nonZeroFeature = (int *) malloc( size * sizeof( int ) );
  if ( nonZeroFeature == NULL ) {
    fprintf( stderr, "Memory allocation failed.\n");
    return 0;
  }
  error = (double *) malloc( size * sizeof( double ) );
  if( error == NULL ) {
    fprintf( stderr, "Memory allocation failed.\n");
    return 0;
```

```
  }
  return 1;
}


/****************************  Public Functions  ************************/

/*FN***************************************************************************

int readFile( FILE *in )                                                    80
Returns: int – Fail(0) on error, Succeed(1) otherwise.
Purpose: This function reads the training file to extract the class, the
         id/value pairs of features of each example.
         Data Structures are initialized to store these informations
         for the training process.
 Notes: It is assumed that the size of each line of the feature
        description of data is not more than 10000 characters long and
        the training file has been opened successfully for reading.
**/
                                                                            90
int readFile( FILE *in )
{
  char temp[DATATEMP1];
  char temp2[DATATEMP2];
  char label[DATATEMP1];
  int numFeature;
  int i,j, specialIndex;
  int lineCount;
  int idValue;
  double featureValue;                                                     100
  char c;
  extern int maxFeature;
  extern int numExample;
  int numZeroFeature = 0;

  lineCount = 0;
  dataSetSize = 0;
  /*************************************

     Estimate the number of examples.                                      110

   *************************************/
  while (fgets(temp, MAXLINE, in ) != NULL )
    dataSetSize++;

  dataSetSize++;
  rewind ( in );

  if( !initializeData( dataSetSize ) ) {
    fprintf( stderr, "Error in initializing data\n" );                     120
    return 0;
  }

  numExample = 0; exampleIndex = 0;
  maxFeature = 0;
  printf("Reading training file . . .\n");
  while( ( c = getc(in) ) != EOF ) {
  /*************************************

      Ignore comment and blank lines                                       130

   *************************************/
    while( c == '#' || c == '\n' ) {
```

```
    if( c == '#' ) {
      while( ( c = getc(in) ) != '\n' )
        ;
    }
    if( c == '\n' )
      c = getc( in );
  }
  if( c == EOF )
    break;

  /**********************************

      Read one line of descriptioon

   **********************************/
  else {
    exampleIndex++;
    i = 0; numFeature = 0;
    temp[i] = c; i++;
    while( ( c = getc(in) ) != '\n' ) {
      temp[i] = c; i++;
      if( c == ':' )
        numFeature++;
    }
    temp[i] = '\0';
    lineCount++;
    /************************************************

        Each line should start with a class label.

     ************************************************/
    j = 0;
    while ( temp[j] != ' ' ) {
      label[j] = temp[j]; j++;
    }
    label[j] = '\0';
    if( atoi(label) != 1 && atoi(label) != -1 ) {
      fprintf( stderr, "Expect a class label in line %d\n", lineCount);
      return 0;
    }

    numExample++;
    nonZeroFeature[exampleIndex] = numFeature;

    if( numFeature != 0 ) {
      example[exampleIndex] = (FeaturePtr *)malloc( numFeature * sizeof( FeaturePtr ) );
      if ( example[exampleIndex] == NULL ) {
        fprintf( stderr, "Memory allocation failed\n");
        return 0;
      }

      for (j = 0; j < numFeature; j++) {
        example[exampleIndex][j] = (FeaturePtr) malloc (sizeof(struct feature));
        if ( example[exampleIndex][j] == NULL ) {
          fprintf( stderr, "Memory allocation failed\n");
          return 0;
        }
      }
    }
    else {
      /*****************************************************

          We have example with all the features zero. We just
```

```
        record their number of nonzero feature as zero.

     ******************************************************/
    example[exampleIndex] = (FeaturePtr *)malloc( sizeof( FeaturePtr ) );
    if ( example[exampleIndex] == NULL ) {
      fprintf( stderr, "Memory allocation failed\n");
      return 0;
    }
    example[exampleIndex][0] = (FeaturePtr)malloc( sizeof(struct feature) );
    if ( example[exampleIndex][0] == NULL ) {
      fprintf( stderr, "Memory allocation failed\n");
      return 0;
    }
    nonZeroFeature[exampleIndex] = 0;
}

/*******************************************

    Extract the class label of the example.

 *******************************************/
i = 0;
while( temp[i] != ' ') {
  temp2[i] = temp[i]; i++;
}
temp2[i] = '\0';
target[exampleIndex] = atoi( temp2 );
error[exampleIndex] = 0 - target[exampleIndex];
i++;

if( numFeature != 0 ) {
  /************************************************

      Extract and store the feature id/value pairs.

    ************************************************/
  featureIndex = 0;
  while( temp[i] != '\0' ) {
    j = 0;
    while( temp[i] != ':' ) {
      temp2[j] = temp[i];
      i++; j++;
    }
    temp2[j] = '\0';
    if( sscanf( temp2, "%d", &idValue) == 0 ) {
      fprintf( stderr, "Expect an integer for id in line %d\n", lineCount);
      return 0;
    }
    example[exampleIndex][featureIndex]->id = atoi( temp2 );
    j = 0; i++;
    while( temp[i] != ' ' && temp[i] != '\0' ) {
      temp2[j] = temp[i];
      i++; j++;
    }
    temp2[j] = '\0';
    if( sscanf( temp2, "%f", &featureValue) == 0 ) {
      fprintf( stderr, "Expect a real number for feature value in line %d\n", lineCount );
      return 0;
    }
    if( fabs(atof( temp2 )) > EPS ) {
      example[exampleIndex][featureIndex]->value = atof( temp2 );
      featureIndex++;
    }
```

```
        } /* end while not end of line */

        /*********************************************************

            Update the number of nonzero features of the example
            and the largest feature id found so far for all the
            examples read.

         *********************************************************/
        nonZeroFeature[exampleIndex] = featureIndex;
        if( example[exampleIndex][featureIndex - 1]-> id > maxFeature )
          maxFeature = example[exampleIndex][featureIndex - 1]->id;
      }
    } /* end else */
  } /* end while not EOF */
  printf("Finish reading training file.\n");
  return 1;
}


/*FN*************************************************************************

int initializeTraining( void )
Returns: int - Fail(0), on error Succeed(1) otherwise.
Purpose: To initialize dpCache, weight vector, threshold b, lambda, and nonBound arrays.
Notes: None.
**/

int initializeTraining( void )
{
  int i, j;

  printf("Initializing data structures . . .\n");
  weight = (double *) malloc( (maxFeature+1) * sizeof(double) );
  if( weight == NULL ) {
    fprintf( stderr, "Memory allocation failed.\n");
    return 0;
  }
  for ( i = 1; i <= maxFeature; i++ )
    weight[i] = 0;

  lambda = (double *) malloc( (numExample+1) * sizeof(double) );
  if( lambda == NULL ) {
    fprintf( stderr, "Memory allocation failed.\n");
    return 0;
  }
  for( i = 1; i <= numExample; i++ )
    lambda[i] = 0;

  nonBound = (int *) malloc( (numExample+1) * sizeof(int) );
  if( nonBound == NULL ){
    fprintf( stderr, "Memory allocation failed.\n");
    return 0;
  }
  for ( i = 1; i <= numExample; i++ )
    nonBound[i] = 0;
  numNonBound = 0;

  unBoundIndex = (int *) malloc( numExample * sizeof( int ) );
  if( unBoundIndex == NULL ) {
    fprintf( stderr, "Memory allocation failed.\n");
    return 0;
  }
```

```
  errorCache = (int *) malloc( numExample * sizeof(int) );
  if( errorCache == NULL ) {
    fprintf( stderr, "Memory allocation failed.\n");
    return 0;
  }

  nonZeroLambda = (int *) malloc( numExample * sizeof(int) );
  if( nonZeroLambda == NULL ) {
    fprintf( stderr, "Memory allocation failed.\n");
    return 0;
  }

  printf("Finish initializing data structures.\n");
  return 1;
}
```

```
/************************  learn.h  *********************************

Purpose: The header file for module learn.c
Notes: None.

Programmer: Ginny Mak
Location: Montreal, Canada.
Last Updated: April 6, 2000.

**/

double C ;
double b;
int degree ;
int kernelType ;
double sigmaSqr;
double rbfConstant;
double iteration;
double totalIteration;
int binaryFeature;


/************************  Public Functions  ****************************/

extern void startLearn( void );
```

```
/************************  learn.c  *********************************

Purpose: To learn from the given training examples.
Notes: None.

Programmer: Ginny Mak
Location: Montreal, Canada.
Last Updated: April 6, 2000.

**/

#include <stdio.h>
#include <stdlib.h>
#include <float.h>
#include <math.h>
```

```
#include <time.h>
#include <limits.h>
#include <ctype.h>
#include "initializeTraining.h"
#include "utility.h"
#include "learn.h"

/****************** Private defines and data structures ********************/

#define MAX(X, Y) ((X) > (Y) ?(X) : (Y))
#define MIN(X, Y) ((X) < (Y) ?(X) : (Y))
#define TOL 0.001
#define EPS DBL_EPSILON
#define MAXNUM DBL_MAX

static double lambda1;
static double lambda2;
static double E1;
static int unBoundPtr = -1 ;
static int errorPtr = -1;
static int unBounde1 = 0;
static int unBounde2 = 0;
static int numNonZeroLambda = 0;
static int lambdaPtr = -1;


/************************ Declared Functions ******************************/

static double dotProduct( FeaturePtr *x, int sizeX, FeaturePtr *y, int sizeY );
static double calculateError( int n );
static int takeStep( int e1, int e2 );
static int examineExample( int e1 );


/************************* Private Functions ******************************/
/*FN**************************************************************************

double dotProduct( FeaturePtr *x, int sizeX, FeaturePtr *y, int sizeY )
Returns: the dot product of two given vectors
Purpose: Calculate the dot product of two given data vectors.
Note: This function does not change the passed argument.
**/

static double dotProduct( FeaturePtr *x, int sizeX, FeaturePtr *y, int sizeY )
{
  int num1, num2, a1, a2;
  int p1 = 0; int p2 = 0;
  double dot = 0;

  if( sizeX == 0 || sizeY == 0 )
    return 0;
  if( binaryFeature == 0 ) {
    num1 = sizeX; num2 = sizeY;
    while( p1 < num1 && p2 < num2 ) {
      a1 = x[p1]->id;
      a2 = y[p2]->id;
      if( a1==a2 ) {
        dot += (x[p1]->value)*(y[p2]->value);
        p1++; p2++;
      }
      else if ( a1 > a2 )
        p2++;
      else
```

```
        p1++;
      }
    }
    else {
      num1 = sizeX; num2 = sizeY;
      while( p1 < num1 && p2 < num2 ) {
        a1 = x[p1]->id;
        a2 = y[p2]->id;
        if( a1==a2 ) {
          dot += 1;
          p1++; p2++;
        }
        else if ( a1 > a2 )
          p2++;
        else
          p1++;
      }
    }
    return dot;
}

/*FN************************************************************************

double calculateError( int n )
Returns: double - the error of an example n
Purpose: To calculate the error of example n.
Notes: read access to target[ ], lambda[ ], dpCache[ ][ ], kernelType, b,
       rbfConstant
**/

static double calculateError( int n )
{
    double svmOut, interValue;
    int i, index;

    svmOut = 0;
    if( kernelType == 0 ) {
      for( i = 0; i < numNonZeroLambda; i++ ) {
        index = nonZeroLambda[i];
        svmOut += lambda[index]*target[index]*dotProduct(example[index],
                  nonZeroFeature[index], example[n], nonZeroFeature[n]);
      }
    }
    else if( kernelType == 1 ) {
      for( i = 0; i < numNonZeroLambda; i++ ) {
        index = nonZeroLambda[i];
        svmOut += lambda[index]*target[index]*power(1+dotProduct(example[index],
                  nonZeroFeature[index], example[n], nonZeroFeature[n]), degree);
      }
    }
    else if( kernelType == 2 ) {
      for( i = 0; i < numNonZeroLambda; i++ ) {
        index = nonZeroLambda[i];
        /*****************************************************

          Calculate the abs(example[index] - example[n])^2

         *****************************************************/
        if( binaryFeature == 0 ) {
          interValue = dotProduct(example[n], nonZeroFeature[n], example[n],
                       nonZeroFeature[n]) - 2*dotProduct(example[index],
                       nonZeroFeature[index], example[n], nonZeroFeature[n])
```

```
                    + dotProduct(example[index], nonZeroFeature[index],
                    example[index], nonZeroFeature[index]);
      }
      else {
        interValue = nonZeroFeature[n] − 2∗dotProduct(example[index],
                    nonZeroFeature[index], example[n], nonZeroFeature[n])
                    + nonZeroFeature[index];
      }                                                                          150
      svmOut += lambda[index]∗target[index]∗exp(−interValue∗rbfConstant);
    }
  }
  return (svmOut − b − target[n]);
}


/∗FN************************************************************************
                                                                             160
int takeStep (int e1, int e2 )
Returns: int – 0 if optimization is not successful
               1 if joint optimization of e1 and e2 is successful.
Purpose: Joint optimize the langrange multipliers of e1 and e2.
Notes: None.
∗∗/

int takeStep( int e1, int e2 )
{
  double k11, k12, k22, eta;                                                 170
  double a1, a2, f1, f2, L1, L2, H1, H2, Lobj, Hobj;
  int y1, y2, s, i, j, index, findPos, temp;
  double interValue1, interValue2;
  double  E2, b1, b2, f, interValue, oldb;

  if( e1 == e2 )
    return 0;

  lambda1 = lambda[e1]; lambda2 = lambda[e2];
  y1 = target[e1]; y2 = target[e2];                                          180
  s = y1∗y2;

  if( nonBound[e2] )
    E2 = error[e2];
  else
    E2 = calculateError( e2 );

  if( y1 != y2 ) {
    L2 = MAX( 0, lambda2 − lambda1 );
    H2 = MIN( C, C + lambda2 − lambda1 );                                    190
  }
  else {
    L2 = MAX( 0, lambda1 + lambda2 − C );
    H2 = MIN( C, lambda1 + lambda2 );
  }
  if( fabs( L2 − H2 ) < EPS ) /∗ L2 equals H2 ∗/
    return 0;
  if( kernelType == 0 ) {  /∗ linear ∗/
    if( binaryFeature == 0 ) {
      k11 = dotProduct(example[e1], nonZeroFeature[e1], example[e1],         200
            nonZeroFeature[e1]);
      k22 = dotProduct(example[e2], nonZeroFeature[e2], example[e2],
            nonZeroFeature[e2]);
    }
    else {
```

```
    k11 = nonZeroFeature[e1];
    k22 = nonZeroFeature[e2];
  }
  k12 = dotProduct(example[e1], nonZeroFeature[e1], example[e2],
        nonZeroFeature[e2]);
}
else if( kernelType == 1 ) { /* polynomial */
  if( binaryFeature == 0 ) {
    k11 = power( 1 + dotProduct(example[e1], nonZeroFeature[e1],
          example[e1], nonZeroFeature[e1]), degree );
    k22 = power( 1 + dotProduct(example[e2], nonZeroFeature[e2],
          example[e2], nonZeroFeature[e2]), degree );
  }
  else {
    k11 = power( 1 + nonZeroFeature[e1], degree );
    k22 = power( 1 + nonZeroFeature[e2], degree );
  }
  k12 = power( 1 + dotProduct(example[e1], nonZeroFeature[e1],
        example[e2], nonZeroFeature[e2]), degree );
}
else if( kernelType == 2 ) {  /* rbf */
  k11 = 1;
  k22 = 1;
  if( binaryFeature == 0 ) {
    interValue = dotProduct(example[e1], nonZeroFeature[e1], example[e1],
                 nonZeroFeature[e1]) - 2*dotProduct(example[e1],
                 nonZeroFeature[e1], example[e2], nonZeroFeature[e2])
                 + dotProduct(example[e2], nonZeroFeature[e2], example[e2],
                 nonZeroFeature[e2]);
  }
  else {
    interValue = nonZeroFeature[e1] - 2*dotProduct(example[e1],
                 nonZeroFeature[e1], example[e2], nonZeroFeature[e2])
                 + nonZeroFeature[e2];
  }
  k12 = exp( -interValue*rbfConstant );
}
eta = 2*k12 - k11 - k22;
if( eta < 0 ) {
  a2 = lambda2 - y2*(E1 - E2)/eta;
  /*********************************

     Constrain a2 to within box

   *********************************/
  if( a2 < L2 )
    a2 = L2;
  else if( a2 > H2 )
    a2 = H2;
}
else {
  /***********************************

     Handle the degenerative case.

   ***********************************/
  L1 = lambda1 + s*(lambda2 - L2);
  H1 = lambda1 + s*(lambda2 - H2);
  f1 = y1*(E1+b) - lambda1*k11 - s*lambda2*k12;
  f2 = y2*(E2+b) - lambda2*k22 - s*lambda1*k12;
  Lobj = -0.5*L1*L1*k11 - 0.5*L2*L2*k22 - s*L1*L2*k12 - L1*f1 - L2*f2;
  Hobj = -0.5*H1*H1*k11 - 0.5*H2*H2*k22 - s*H1*H2*k12 - H1*f1 - H2*f2;
  if( Lobj > Hobj + EPS )
```

```
    a2 = L2;
  else if( Lobj < Hobj – EPS )
    a2 = H2;
  else
    a2 = lambda2;
}
if( fabs(a2 – lambda2) < EPS*(a2+lambda2 + EPS))
  return 0;

/*****************************

    Find the new lambda1

 *****************************/
a1 = lambda1  + s*(lambda2 – a2);
if( a1 < 0 )
  a1 = 0;
/*************************************

    Check e1, e2 for unbound lamdas

 *************************************/
if( a1 > 0 && a1 < C )
  unBounde1 = 1;
else
  unBounde1 = 0;
if( a2 > 0 && a2 < C )
  unBounde2 = 1;
else
  unBounde2 = 0;

/*******************************************

    Update the number of non-zero lambda

 *******************************************/
if( a1 > 0 ) {
  if( numNonZeroLambda == 0 ) {
    lambdaPtr++;
    nonZeroLambda[lambdaPtr] = e1;
    numNonZeroLambda++;
  }
  else if( numNonZeroLambda == 1 && nonZeroLambda[0] != e1 ) {
    lambdaPtr++;
    nonZeroLambda[lambdaPtr] = e1;
    numNonZeroLambda++;
    if(e1 < nonZeroLambda[0]) {
      temp = e1;
      nonZeroLambda[1] = nonZeroLambda[0];
      nonZeroLambda[0] = e1;
    }
  }
  else if( numNonZeroLambda > 1 ) {
    if( binSearch(e1, nonZeroLambda, numNonZeroLambda ) == –1 ) {
      lambdaPtr++;
      nonZeroLambda[lambdaPtr] =  e1;
      numNonZeroLambda++;
      quicksort( nonZeroLambda, 0, lambdaPtr);
    }
  }
}
if( a2 > 0 ) {
  if( numNonZeroLambda == 0 ) {
```

```
      lambdaPtr++;
      nonZeroLambda[lambdaPtr] = e2;
      numNonZeroLambda++;
    }
    else if( numNonZeroLambda == 1 && nonZeroLambda[0] != e2 ) {
      lambdaPtr++;
      nonZeroLambda[lambdaPtr] = e2;
      numNonZeroLambda++;
      if(e2 < nonZeroLambda[0]) {                                          340
        temp = e2;
        nonZeroLambda[1] = nonZeroLambda[0];
        nonZeroLambda[0] = e2;
      }
    }
    else if( numNonZeroLambda > 1 ) {
      if( binSearch(e2, nonZeroLambda, numNonZeroLambda ) == -1 ) {
        lambdaPtr++;
        nonZeroLambda[lambdaPtr] =  e2;
        numNonZeroLambda++;                                                350
        quicksort( nonZeroLambda, 0, lambdaPtr);
      }
    }
  }

  /***************************

      Update the threshold b.

   ***************************/                                           360
  oldb = b;
  if( kernelType == 0 ) {
    if( binaryFeature == 0 ) {
      b1 = E1 + y1*(a1-lambda1)*dotProduct(example[e1],
           nonZeroFeature[e1], example[e1], nonZeroFeature[e1])
           + y2*(a2-lambda2)*dotProduct(example[e1], nonZeroFeature[e1],
           example[e2], nonZeroFeature[e2]) + oldb;

      b2 = E2 + y1*(a1-lambda1)*dotProduct(example[e1],
           nonZeroFeature[e1], example[e2], nonZeroFeature[e2])           370
           + y2*(a2-lambda2)*dotProduct(example[e2], nonZeroFeature[e2],
           example[e2], nonZeroFeature[e2])+ oldb;
    }
    else {
      b1 = E1 + y1*(a1-lambda1)*nonZeroFeature[e1]
           + y2*(a2-lambda2)*dotProduct(example[e1], nonZeroFeature[e1],
           example[e2], nonZeroFeature[e2]) + oldb;

      b2 = E2 + y1*(a1-lambda1)*dotProduct(example[e1],
           nonZeroFeature[e1], example[e2], nonZeroFeature[e2])           380
           + y2*(a2-lambda2)*nonZeroFeature[e2]+ oldb;
    }
  }
  else if( kernelType == 1 ) {
    if( binaryFeature == 0 ) {
      b1 = E1 + y1*(a1-lambda1)*power(1+dotProduct(example[e1],
           nonZeroFeature[e1], example[e1], nonZeroFeature[e1]), degree)
           + y2*(a2-lambda2)*power(1+dotProduct(example[e1],
           nonZeroFeature[e1], example[e2], nonZeroFeature[e2]), degree)
           + oldb;                                                         390

      b2 = E2 + y1*(a1-lambda1)*power(1+dotProduct(example[e1],
           nonZeroFeature[e1], example[e2], nonZeroFeature[e2]), degree)
           + y2*(a2-lambda2)*power(1+dotProduct(example[e2],
```

```
            nonZeroFeature[e2], example[e2], nonZeroFeature[e2]), degree)
            + oldb;
    }
    else {
      b1 = E1 + y1*(a1-lambda1)*power(1+nonZeroFeature[e1], degree)
            + y2*(a2-lambda2)*power(1+dotProduct(example[e1],
            nonZeroFeature[e1], example[e2], nonZeroFeature[e2]), degree)
            + oldb;

      b2 = E2 + y1*(a1-lambda1)*power(1+dotProduct(example[e1],
            nonZeroFeature[e1], example[e2], nonZeroFeature[e2]), degree)
            + y2*(a2-lambda2)*power(1+nonZeroFeature[e2], degree) + oldb;
    }
  }
  else if( kernelType == 2 ) {
    if( binaryFeature == 0 ) {
      interValue = dotProduct(example[e1], nonZeroFeature[e1], example[e1],
                   nonZeroFeature[e1]) - 2*dotProduct(example[e1],
                   nonZeroFeature[e1], example[e2], nonZeroFeature[e2])
                   + dotProduct(example[e2], nonZeroFeature[e2], example[e2],
                   nonZeroFeature[e2]);
    }
    else {
      interValue = nonZeroFeature[e1] - 2*dotProduct(example[e1],
                   nonZeroFeature[e1], example[e2], nonZeroFeature[e2])
                   + nonZeroFeature[e2];
    }
    b1 = E1 + y1*(a1-lambda1) + y2*(a2-lambda2)*exp(-interValue*rbfConstant)
         + oldb;

    b2 = E2 + y1*(a1-lambda1)*exp(-interValue*rbfConstant)
         + y2*(a2-lambda2) + oldb;
  }

  if( fabs(b1-b2) < EPS )  /* b==b1==b2 */
    b = b1;

  else if( !unBounde1 && !unBounde2 )
    b =  (b1+b2)/2;

  else {
    if( unBounde1 )
      b = b1;
    if( unBounde2 )
      b = b2;
  }

  /*************************************************

      Update the weight vector if kernel is linear.

   *************************************************/
  if( kernelType == 0 ){  /* linear */
    for( i = 0; i < nonZeroFeature[e1]; i++ ) {
      if( binaryFeature == 0 )
        weight[example[e1][i]->id] += y1*(a1-lambda1)*(example[e1][i]->value);

      else
        weight[example[e1][i]->id] += y1*(a1-lambda1);
    }

    for( i = 0; i < nonZeroFeature[e2]; i++ ){
      if( binaryFeature == 0 )
```

```
      weight[example[e2][i]->id] += y2*(a2-lambda2)*(example[e2][i]->value);
    else
      weight[example[e2][i]->id] += y2*(a2-lambda2);
  }
}

/*****************************************************

    Update the error cache and store a1 and a2.
    Start with existing nonbound examples first.

 *****************************************************/
for( i = 0; i < numNonBound; i++ ) {
  index = unBoundIndex[i];
  if( kernelType == 0 )
    error[index] += y1*(a1-lambda1)*dotProduct(example[e1],
                    nonZeroFeature[e1], example[index],
                    nonZeroFeature[index])
                    + y2*(a2-lambda2)*dotProduct(example[e2],
                    nonZeroFeature[e2], example[index],
                    nonZeroFeature[index]) + oldb - b;
  else if( kernelType == 1 )
    error[index] += y1*(a1-lambda1)*power(1+dotProduct(example[e1],
                    nonZeroFeature[e1], example[index],
                    nonZeroFeature[index]), degree)
                    + y2*(a2-lambda2)*power(1+dotProduct(example[e2],
                    nonZeroFeature[e2], example[index],
                    nonZeroFeature[index]), degree) + oldb - b;
  else if( kernelType == 2 ) {
    if( binaryFeature == 0 ) {
      interValue1 = dotProduct(example[e1], nonZeroFeature[e1],
                    example[e1], nonZeroFeature[e1])
                    - 2*dotProduct(example[e1], nonZeroFeature[e1],
                    example[index], nonZeroFeature[index])
                    + dotProduct(example[index], nonZeroFeature[index],
                    example[index], nonZeroFeature[index]);
      interValue2 = dotProduct(example[e2], nonZeroFeature[e2],
                    example[e2], nonZeroFeature[e2])
                    - 2*dotProduct(example[e2], nonZeroFeature[e2],
                    example[index], nonZeroFeature[index])
                    + dotProduct(example[index], nonZeroFeature[index],
                    example[index], nonZeroFeature[index]);
    }
    else {
      interValue1 = nonZeroFeature[e1] - 2*dotProduct(example[e1],
                    nonZeroFeature[e1], example[index],
                    nonZeroFeature[index]) + nonZeroFeature[index];
      interValue2 = nonZeroFeature[e2] - 2*dotProduct(example[e2],
                    nonZeroFeature[e2], example[index],
                    nonZeroFeature[index]) + nonZeroFeature[index];
    }

    error[index] += y1*(a1-lambda1)*exp(-interValue1*rbfConstant)
                    + y2*(a2-lambda2)*exp(-interValue2*rbfConstant)
                    + oldb - b;
  }
}
lambda[e1] = a1;
lambda[e2] = a2;

/*************************************

    Calculate the error for e1 and e2.
```

```
 ************************************/
if( unBounde1 )
  error[e1] = 0;

if( unBounde2 )
  error[e2] = 0;

if( errorPtr > 0 )
  qsort2( errorCache, 0, errorPtr, error );                                    530
/*****************************************************************

    Update the nonbound set, the unBoundIndex set and the error
    cache index.

 *****************************************************************/
if( !nonBound[e1] && unBounde1 ) {
  /************************************************

      e1 was bound and is unbound after optimization.                          540
      Add e1 to unboundIndex[ ], increment the number
      of nonbound and update nonBound[ ].
      Update error cache indexes and sort them in
      increasing order of error.

   ************************************************/
  unBoundPtr++;
  unBoundIndex[unBoundPtr] = e1;
  nonBound[e1] = 1;
  numNonBound++;                                                               550
  quicksort( unBoundIndex, 0, unBoundPtr );
  errorPtr++;
  errorCache[errorPtr] = e1;
  if( errorPtr > 0 ) {
    qsort2( errorCache, 0, errorPtr, error );
  }
}
else if( nonBound[e1] && !unBounde1 ) {
  /*****************************************************

                                                                               560
      e1 was nonbound and is bound after optimization.
      Remove e1 from unboundIndex[ ], decrement the number
      of nonbound and update nonBound[ ].
      Update error cache indexes and sort them in
      increasing order of error.

   *****************************************************/
  findPos = binSearch( e1, unBoundIndex, numNonBound );
  /*********************************************************

                                                                               570
       Mark this previous e1 position in array unBoundIndex[ ]
       with a number greater than all possible indexes so
       that when we do sorting, this number is at the end of
       range of the nonbound indexes

   *********************************************************/
  unBoundIndex[findPos] = numExample+1;
  quicksort( unBoundIndex, 0, unBoundPtr );
  unBoundPtr--;
  numNonBound--;                                                               580
  nonBound[e1] = 0;
  if( errorCache[errorPtr] == e1 )
    errorPtr--;
```

```
    else {
      error[e1] = error[errorCache[errorPtr]] + 1;
      qsort2( errorCache, 0, errorPtr, error );
      errorPtr--;
    }
  }
  if( !nonBound[e2] && unBounde2 ) {                                    590
    /*************************************************

        e2 was bound and is unbound after optimization.
        Add e2 to unboundIndex[ ], increment the number
        of nonbound and update nonBound[ ].
        Update error cache indexes and sort them in
        increasing order of error.

     *************************************************/
    unBoundPtr++;                                                       600
    unBoundIndex[unBoundPtr] = e2;
    nonBound[e2] = 1;
    numNonBound++;
    quicksort( unBoundIndex, 0, unBoundPtr );
    errorPtr++;
    errorCache[errorPtr] = e2;
    if( errorPtr > 0 ) {
      /* sort these errorCache indexes in increasing
        order of error */
      qsort2( errorCache, 0, errorPtr, error );                         610
    }
  }
  else if( nonBound[e2] && !unBounde2 ) {
    /*******************************************************

        e2 was nonbound and is bound after optimization.
        Remove e2 from unboundIndex[ ], decrement the number
        of nonbound and update nonBound[ ].
        Update error cache indexes and sort them in
        increasing order of error.                                      620

     *******************************************************/
    findPos = binSearch( e2, unBoundIndex, numNonBound );
    /*********************************************************

        Mark this previous e2 position in array unBoundIndex[ ]
        with a number greater than all possible indexes so
        that when we do sorting, this number is at the end of
        range of the nonbound indexes
                                                                        630
     *********************************************************/
    unBoundIndex[findPos] = numExample+1;
    quicksort( unBoundIndex, 0, unBoundPtr );
    unBoundPtr--;
    numNonBound--;
    nonBound[e2] = 0;
    /* update error cache indices */
    if( errorCache[errorPtr] == e2 )
      errorPtr--;
    else {                                                              640
      error[e2] = error[errorCache[errorPtr]] + 1;
      qsort2( errorCache, 0, errorPtr, error );
      errorPtr--;
    }
  }
  /*************************************************************
```

```
    iteration is the number of successful joint optimizations.

   ************************************************************/
  iteration += 1;
  return 1;
}


/*FN*************************************************************************

int examineExample( int e1 )
Returns: int – 0 if no optimization takes place, 1 otherwise.
Purpose: to iteratively optimize the langrangian multipliers of the examples
Notes: None.
**/

int examineExample( int e1 )
{
  double r1;
  int found, e2,i, j, index, y1;

  found = 0;
  totalIteration ++;
  y1 = target[e1];
  lambda1 = lambda[e1];
  if( nonBound[e1] )
    E1 = error[e1];
  else
    E1 = calculateError( e1 );

  r1 = E1*y1;
  if( (r1 < -TOL && lambda1 < C) || (r1 > TOL && lambda1 > 0) ) {
    /*********************************

        e1 violates KKT condition.

     *********************************/
    if( numNonBound > 1 ) {
      /*****************************************************************

          This is the first hierarchy of the second choice heuristic.

       *****************************************************************/
      if ( E1 > 0 ) {
        if( error[errorCache[0]] >= EPS )
          found = 0;
        else {
          e2 = errorCache[0];
          found = 1;
        }
      }
      else if ( E1 < 0 ) {
        if ( error[errorCache[errorPtr]] <= EPS )
          found = 0;
        else {
          e2 = errorCache[errorPtr];
          found = 1;
        }
      }
      if (found)
        if( takeStep( e1, e2 ) )
          return 1;
```

```
    }
    /*****************************************************************

        This is the second hierarchy of the second choice heuristic

     *****************************************************************/
    if ( numNonBound > 1 ) {
      index = myrandom( numNonBound );
      e2 = unBoundIndex[index];
      for( i = 0; i < numNonBound; i++ ) {
        if ( takeStep( e1, e2 ) )
          return 1;
        index++;
        if( index == numNonBound )
          index = 0;
        e2 = unBoundIndex[index];
      }
    }
    /*****************************************************************

        This is the third hierarchy of the second choice heuristic.

     *****************************************************************/
    if ( numNonZeroLambda > 1 ) {
      index = myrandom( numNonZeroLambda );
      e2 = nonZeroLambda[index];
      /* only use bounded lambdas */
      for( i = 0; i < numNonZeroLambda; i++ ) {
        if( nonBound[e2] == 0 ) {
          if ( takeStep( e1, e2 ) )
            return 1;
        }
        index++;
        if( index == numNonZeroLambda )
          index = 0;
        e2 = nonZeroLambda[index];
      }
    }
    /*****************************************************************

        This is the fourth hierarchy of the second choice heuristic.

     *****************************************************************/
    e2 = myrandom( numExample ) + 1;
    for( i = 0; i < numExample; i++ ) {
      /*****************************************

          Only use examples with zero lambda.

       *****************************************/
      if( lambda[i] < EPS ) {
        if( takeStep( e1, e2 ) )
          return 1;
      }
      e2++;
      if( e2 == numExample + 1 )
        e2 = 1;
    }
  }
  return 0;
}
```

```
/************************  Public Functions  ****************************

void startLearn( void )
Returns: Nothing.
Purpose: To find the support vectors from the training examples.                    780
Notes: None.
**/

void startLearn( void )
{
  int i;
  int numChanged = 0;
  int examineAll = 1;

  b = 0; iteration = 0; totalIteration = 0;                                         790
  while( numChanged > 0 || examineAll ) {
    numChanged = 0;
    if( examineAll ) {
      for( i = 1; i <= numExample; i++ )
        numChanged += examineExample(i);
    }
    else {
      for( i = 0; i < numNonBound; i++ )
        numChanged += examineExample( unBoundIndex[i] );
    }                                                                               800
    if( examineAll == 1 )
      examineAll = 0;
    else if ( numChanged == 0 )
      examineAll = 1;
  }
}
```

```
/*****************************  utility.h  *********************************

 Purpose: Header of the utility module.
 Notes: None

 Programmer: Ginny Mak
 Location: Montreal, Canada
 Last Updated: April 6, 2000.

**/                                                                              10

/************************************  Public Functions  ********************/

extern double power ( double f, int n );
extern int binSearch(int x, int *v, int n);
extern void swap(int *v, int i, int j);
extern void quicksort(int *v, int left, int right);
extern void qsort2(int *v, int left, int right, double *d);
extern int myrandom(int n);
```

```
/*****************************  utility.c  *********************************

 Purpose: To supply some utility functions
 Notes: None.

 Programmer: Ginny Mak
```

```
 Location: Montreal, Canada.
 Last Updated: April 6, 2000.

**/

#include <stdio.h>

/********************  Private Data ************************************/

static int seed = 0;

/***************************  Public Functions  ************************/

/*FN***************************************************************************

double power( double f, int n )

Returns: double – value of f raised to the power of n

Purpose: Calculate the value of a double raised to the power of n.
         Return 1 if n is zero.

Notes: Does not work with negative n.
**/

double power( double f, int n )
{
        int i;
        double p;

        p = 1.0;
        for ( i = 1; i <= n; i++ )
                p = p * f;
        return p;
}


/*FN***************************************************************************

int binSearch( int x, int *v, int n)

Returns: int – the location in the array v where the integer x is found,
         or -1 when there is no match.

Purpose: Perform binary search for an integer x in an integer array of size n.

Notes: None.
**/

int binSearch( int x, int *v, int n)
{
  int low, high, mid;

  low = 0;
  high = n - 1;
  while ( low <= high ) {
    mid = ( low+high )/2;
    if( x < v[mid] )
      high = mid - 1;
    else if ( x > v[mid] )
      low = mid + 1;
    else
      return mid;
```

```
  }
  return -1;
}


/*FN*************************************************************************

void swap( int *v, int i, int j )

Returns: None

Purpose: To exchange v[i] with v[j]

Notes: None.
**/

void swap ( int *v, int i, int j )
{
  int temp;

  temp = v[i];
  v[i] = v[j];
  v[j] = temp;
}


/*FN*************************************************************************

void quicksort ( int *v, int left, int right )

Returns: None.

Purpose: To sort an array of integers between position left and right.

Notes: array v between left and right is sorted in ascendiing order.
**/

void quicksort ( int *v, int left, int right )
{
  int i, last;
  void swap( int *v, int i, int j );

  if ( left >= right )
    return;
  swap( v, left, (left+right)/2 );
  last = left;
  for( i = left + 1; i <= right; i++ )
    if( v[i] < v[left] )
      swap( v, ++last,  i );
  swap( v, left, last );
  quicksort( v, left, last - 1  );
  quicksort( v, last + 1, right );
}


/*FN*************************************************************************

void qsort2 (int *v, int left, int right, double *d)

Returns: None.

Purpose: Array v is an array of indices of array d.
         Sort array v in ascending order
```

```
        so that d[v[i]] > d[v[i-1]].

Notes: array d is unchanged, array v is changed.
**/

void qsort2 ( int *v, int left, int right, double *d )
{
  int i, last;                                                          140
  void swap (int v[], int i, int j);

  if ( left >= right )
    return;
  swap( v, left, (left+right)/2 );
  last = left;
  for ( i = left + 1; i <= right; i++ )
    if( d[v[i]] < d[v[left]] )
      swap( v, ++last, i );
  swap( v, left, last );                                                150
  qsort2 ( v, left, last - 1, d );
  qsort2 ( v, last+1, right, d );
}


/*FN*************************************************************************

int myrandom( int n )

Returns: A random integer between 0 and n-1                             160

Purpose: To generate a random number in the range of 0 and n-1

Notes: None.
**/

int myrandom( int n )
{
  unsigned long int next;
                                                                        170
  next = seed * 1103515245 + 12345;
  seed++;
  next = (unsigned int) (next/65536)%( n );
  return next;
}
```

```
/************************** result.h ************************

Purpose: Header for result module.

Notes: None.

Programmer: Ginny Mak
Location: Montreal, Canada.
Last Updated: April 6, 2000.
                                                                        10
**/

int boundSv;
int unBoundSv;
```

```
/********************** Public Functions ******************/

extern void writeModel( FILE *out );
extern double calculateNorm( void );
extern double getb( void );
```

```
/************************ result.c ********************************

Purpose: Write result of training in model file.

Notes:

Programmer: Ginny mak
Location: Montreal, Canada.
Last Updated: April 6, 2000.

**/

#include <stdio.h>
#include <math.h>
#include <float.h>
#include "initializeTraining.h"
#include "learn.h"
#include "result.h"


/******************** Public Functions ***************************

void writeModel( FILE *out )

Returns: None

Purpose: To write the training results to a model file.

Notes: Model file is assumed to have been opened for writing.
       The training results are written according to predetermined
       format.
       Read access to kernelType, sigmaSqr,degree, maxFeature,
       weight[ ], b, C, lambda[ ], target[ ], example[ ][ ]
**/


void writeModel( FILE *out )
{
  int i, j, numSv;

  numSv = 0; unBoundSv = 0; boundSv = 0;
  /*********************************************************************

     Find out the number of support vectors, both bound and unbound.

   *********************************************************************/
  for ( i = 1; i <= numExample; i++ ) {
    if( lambda[i] > 0) {
      numSv++;
      if( lambda[i] < C )
        unBoundSv++;
      else
        boundSv++;
    }
  }
```

```
  /***************************

     Write to the model file.

   ***************************/
  if( kernelType == 0 )
    fprintf( out, "%d # Linear\n", 0 );
  else if( kernelType == 1 )
    fprintf( out, "%d %d # Polynomial with degree %d\n", 1, degree, degree );
  else if( kernelType == 2 )
    fprintf( out, "%d %f # rbf with variance %f\n", 2, sigmaSqr,
             sigmaSqr );

  fprintf( out, "%d # Number of features\n", maxFeature );

  if( kernelType == 0 ) {
    for( i = 1; i <= maxFeature; i++ )
      fprintf( out, "%f ", weight[i] );
    fprintf( out, "# weight vector\n" );
  }

  fprintf( out, "%18.17f # Threshold b\n", b );

  fprintf( out, "%18.17f # C parameter\n", C );

  fprintf( out, "%d # Number of support vectors\n", numSv );

  /****************************************

     Write the value of (class label*lambda)

   ****************************************/
  if( numSv != 0 ) {
    for( i = 1; i <= numExample; i++ ) {
      if( lambda[i] > 0  ) {
        if( target[i] == 1 )
          fprintf( out, "%.17f ", lambda[i] );
        else if( target[i] == -1 )
          fprintf( out, "%.17f ", -lambda[i] );

        for( j = 0; j < nonZeroFeature[i]; j++ )
          fprintf( out, "%d:%.17f ", example[i][j]->id,
                   example[i][j]->value );
        fprintf( out, "\n" );
      }
    }
  }
}


/*FN************************************************************************
double calculateNorm( void )

Returns: double, the norm of the weight vector.

Purpose: To calculate the norm of the weight vector.

Notes: None.
**/

double calculateNorm( void )
```

```
{
  int i;
  double n;

  n = 0;
  for( i = 1; i <= maxFeature; i++ )
    n += weight[i]*weight[i];
  return sqrt(n);
}

/*FN*************************************************************************

double getb( void )

Returns: double , b

Purpose: Return b value

Notes: None.
**/

double getb( void )
{
        return b;
}
```

```
/*************************  smoClassify.c  ***********************

 Purpose: Program main function.

 Notes: This program is composed of the following modules:

        smoClassify.c - main program module.
        initialize.c  - Initialize all data structures required to do
                        classification by reading the information from
                        the model file and the test data file.
        classify.c    - Do the classification calculations and
                        write the result to the prediction file.

 Programmer: Ginny Mak
 Location: Montreal, Canada.
 Last Updated: April 6, 2000.

**/

#include <stdio.h>
#include "initialize.h"
#include "classify.h"


int main(int argc, char *argv[])
{
  FILE *modelIn, *dataIn, *out;
  double startTime;

  /*******************************

     Check on command line usage

   *******************************/
  if( argc != 4 ) {
```

```
    fprintf( stderr, "Usage: Command line\n" );
    exit(1);
  }
  else {
    if(( modelIn = fopen( argv[1], "r" ) ) == NULL ) {
      fprintf( stderr, "Can't open %s\n", argv[1] );
      exit(2);
    }

    if(( dataIn = fopen( argv[2], "r" ) ) == NULL ) {
      fprintf( stderr, "Can't open %s\n", argv[2] );
      exit(2);
    }

    if(( out = fopen( argv[3], "w" ) ) == NULL ) {
      fprintf( stderr, "Can't open %s\n", argv[3] );
      exit(2);
    }
  }

  /*********************

      Read model file

   *********************/
  if( ! readModel( modelIn ) ) {
    fprintf( stderr, "Error in reading model file %s\n", argv[1] );
    exit (3);
  }
  else
    fclose( modelIn );
  printf("Finish reading model file\n");

  /************************

      Read test data file

   ************************/
  if( !readData( dataIn )) {
    printf("Error reading data file\n");
    exit(4);
  }
  fclose( dataIn );
  printf("Finish reading test data file\n");

  /************************

      Start classfying.

   ************************/
  if ( writeResult( out ) )
    printf("Classification is completed\n");
  else
    fprintf( stderr, "Classification process failed\n");

  fclose( out );
  return 0;
}
```

```
/******************************  initialize.h  ****************************
```

Purpose: Header for program data structures initialization module

```
 Notes: None.

 Programmmer: Ginny Mak
 Location: Montreal, Canada.
 Last Updated: April 6, 2000.
**/

/*****************  Public defines and Data structures  ****************/

typedef struct feature{
  int id;
  double value;
}  Feature;

typedef struct feature* FeaturePtr;

FeaturePtr **example;
FeaturePtr **sv;
double *lambda;
int *svNonZeroFeature;
int *nonZeroFeature;
int *target;
double *weight;
double *output;
double rbfConstant;
int degree;
double b;
int numSv;
int numExample;
int kernelType;
int maxFeature;
int *zeroFeatureExample;

/******************************** Public Functions  ************************/

extern int readModel( FILE *in );
extern int readData( FILE *in );
```

```
/***************************  initialize.c  ******************************

 Purpose: Initialize all data structures for classification by
          reading information from the model file and the data file.

 Notes: None.

 Programmer: Ginny Mak
 Location: Montreal, Canada.
 Last Updated: April 6, 2000.
**/

#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#include "initialize.h"
#include "classify.h"

/******************************** Private Defines and Data Structures *******/

#define MODELTEMP 100
#define DATATEMP1 10000
#define DATATEMP2 1000
```

```
static double C;
static double sigmaSqr;
static maxFeatureRead;

/****************************** Declared Functions ***********************/

static int readString( char *store, char delimiter, FILE *in );
static void skip( char end, FILE *in );
static int initializeModel( int size );
static int initializeData( int dataSize );


/****************************** Private Functions ***********************/

/*FN**************************************************************************

int readString( char *store, char delimiter, FILE *in )

Returns: int – Fail(0) on error, Succeed(1) otherwise.
         a string stored in char *store

Purpose: To read characters from a file until the delimiter is encountered.
         All the characters read except the delimiter are stored in the
         passed character array store.

Notes: The passed file pointer's position changes when the function returns.
       New line and EOF is not a delimiter. If they are read before the
       delimiter is encountered then the function returns 0.
**/

static int readString( char *store, char delimiter, FILE * in )
{
  char c;
  int i;

  i = 0;
  c = getc( in );
  while (c != delimiter && c != '\n' && c != EOF ) {
    store[i] = c;
    i++;
    c = getc( in );
  }
  if ( c == EOF || c == '\n' )
    return 0;
  else
    store[i] = '\0';
  return 1;
}

/*FN**************************************************************************

void skip( char end, FILE *in )

Returns: None

Purpose: To skip all the characters read up to the end character.

Note: The passed file pointer's position changes when the function returns.
**/

void skip(char end , FILE * in)
```

```
{
  char c;

  while((c = getc( in )) != end)
    ;
}


/*FN*************************************************************************

static int initializeModel( int size )

Returns: int – Faild(0) on error, Succeed(1) otherwise.

Purpose: To initialize data for readModel function.

Notes: None.
**/

static int initializeModel( int size )
{
  int i;

  lambda = (double *)malloc( (size + 1)* sizeof(double) );
  if ( lambda == NULL) {
    fprintf( stderr, "Memory allocation failed\n");
    return 0;
  }
  svNonZeroFeature = (int *) malloc( (size + 1) * sizeof( int ) );
  if (svNonZeroFeature == NULL ) {
    fprintf( stderr, "Memory allocation failed\n");
    return 0;
  }
  sv = (FeaturePtr **)malloc( (size + 1) * sizeof(FeaturePtr *));
  if ( sv == NULL ) {
    fprintf( stderr, "Memory allocation failed\n");
    return 0;
  }
  for(i = 1; i <= numSv; i++) {
    sv[i] = (FeaturePtr *)malloc((maxFeature) * sizeof(FeaturePtr));
    if ( sv[i] == NULL ) {
      fprintf( stderr, "Memory allocation failed\n");
      return 0;
    }
  }
}

/*FN*************************************************************************

static int initializeData( int dataSize )

Returns: Int – Fail(0) on error, Succeed(1) otherwise

Purpose: To initialize data for classification.

Note: None
**/

static int initializeData( int dataSetSize )
{
  example = (FeaturePtr **) malloc( dataSetSize * sizeof(FeaturePtr *) );
  if ( example == NULL ) {
    fprintf( stderr, "Memory allocation failed\n");
```

```
    return 0;
  }
  output = (double *) malloc( dataSetSize * sizeof(double) );
  if ( output == NULL ) {
      fprintf( stderr, "Memory allocation failed\n");
      return 0;
  }
  target = (int *) malloc( dataSetSize * sizeof(int) );
  if ( target == NULL ) {
      fprintf( stderr, "Memory allocation failed\n");
    return 0;
  }
  zeroFeatureExample = (int *)malloc( dataSetSize/2 * sizeof(int) );
  if( zeroFeatureExample == NULL ) {
    fprintf( stderr, "Memory allocation failed.\n");
    return 0;
  }
  nonZeroFeature = (int *) malloc( dataSetSize * sizeof(int) );
  if ( nonZeroFeature == NULL ) {
    fprintf( stderr, "Memory allocation failed\n");
    return 0;
  }
}



/****************************** Public Functions ************************/


/*FN*************************************************************************

int readModel( FILE *in )

Returns: int – Fail(0) on error, Succeed(1) otherwise.

Purpose: Read and store informaton read from given model file.

Notes: Assume the model file has been successfully open for reading.
       The model file is a model file resulted from training using
       program smoLearn.
**/

int readModel( FILE *in )
{
  int numNonZeroFeature;
  int i, j, n;
  char c;
  char temp[MODELTEMP];


  printf("Reading model file...\n");
  c = getc( in );
  /************************************************

     If c is 0, then the kernel type is linear.
     If c is 1, then the kernel type is polynomial,
     read the degree of polynomial
     If c is 2, then the kernel type is rbf,
     read the variance.

   ************************************************/
  if( c == '0' )
    kernelType = 0;
```

```
else if( c == '1' ) {
  kernelType = 1;
  skip(' ', in);
  readString(temp, ' ', in);
  degree = atoi( temp );
}
else if( c == '2' ) {
  kernelType = 2;
  skip(' ', in);
  readString(temp, ' ', in);
  sigmaSqr = atof( temp );
  rbfConstant = 1/(2*sigmaSqr);
}
skip('\n', in);

/*************************************************

   Read number of features of training examples

 *************************************************/
readString(temp, ' ', in);
maxFeatureRead = atoi( temp );
skip('\n', in);

/*************************************************

   Read the weight if the kernel type is linear

 *************************************************/
if ( kernelType == 0 ) {
  weight = (double *) malloc( (maxFeatureRead+1) * sizeof( double ) );
  if ( weight == NULL ) {
    fprintf( stderr, "Memory allocation failed.\n" );
    return 0;
  }
  i = 0;
  for( i = 0; i <= maxFeatureRead; i++ )
    weight[i] = 0;
  for( i = 1; i <= maxFeatureRead; i++ ) {
    readString( temp, ' ', in );
    weight[i] = atof( temp );
  }
  skip('\n', in);
}

/************************

   Read the threshold b

 ************************/
readString( temp, ' ', in );
b = atof( temp );
skip('\n', in );

/* read C parameter */
readString( temp, ' ', in );
C = atof( temp );
skip('\n', in );

/*****************************************

   Read the number of support vectors
```

```
   ***************************************/
  readString( temp, ' ', in );
  numSv = atoi(temp);
  skip( '\n', in );
  if( ! initializeModel( numSv ) )                                    280
    return 0;

  /*****************************************************************

     Read the product of lambda of a support vector with class label

   *****************************************************************/
  for( i = 1; i <= numSv; i++ ) {
    readString( temp, ' ', in );
    lambda[i] = atof( temp );                                         290
    j = 0;
    /*************************************************************

       Read the id/value pairs of the features of a support vector

     *************************************************************/
    while( readString( temp, ':', in ) ) {
      sv[i][j] = (FeaturePtr) malloc(sizeof(struct feature));
      if( sv[i][j] == NULL ) {
        fprintf( stderr, "Memory allocation failed\n");               300
        return 0;
      }
      sv[i][j]->id = atoi( temp );
      if(!readString( temp, ' ', in))
        readString( temp, ' ', in );
      sv[i][j]->value = atof(temp);
      j++;
    }
    svNonZeroFeature[i] = j;
  }                                                                   310

  return 1;
}


/*FN*************************************************************************

int readData( FILE *in )

Returns: int – Fail(0) on error, Succeed(1) otherwise.                320

Purpose: This function reads the data file to extract  and store the
         id/value pairs of features of each test data.

Notes: The size of each line of the feature description of data cannot
       be more than 10000 characters long.
       The passed file pointer's position is at EOF if the function
       returns successfully.
**/
                                                                      330
int readData( FILE *in )
{
  char temp[DATATEMP1];
  char temp2[DATATEMP2];
  int numFeature, maxFeature;
  int exampleIndex, featureIndex;
  int i,j,dataSetSize;
  char c;
```

```
int zeroFeatureNumber = 0;
                                                                    340
dataSetSize = 0;
/*********************************

    Estimate the number of data.

 *********************************/
while( (c = getc( in )) != EOF ) {
  if( c == '\n' )
    dataSetSize++;
}                                                                   350
dataSetSize++;
rewind ( in );

if( !initializeData( dataSetSize ) )
  return 0;

/************************************************************

    NonZeroFeature[0] stores the number of non-zero features
    of the weight vector.                                           360

 ************************************************************/
nonZeroFeature[0] = maxFeature;

numExample = 0; exampleIndex = 0;
printf("Reading test data file . . .\n");
while( ( c = getc(in) ) != EOF ) {
  while( c == '#' || c == '\n' ) {
    /************************************
                                                                    370
       Ignore comment and blank line

     ************************************/
    if( c == '#' ) {
      while( ( c = getc(in) ) != '\n' )
        ;
    }
    if( c == '\n' )
      c = getc( in );
  }                                                                 380
  if( c == EOF )
    break;

  /*****************************

     Read one line of input

   *****************************/
  else {
    exampleIndex++;                                                 390
    i = 0; numFeature = 0;
    temp[i] = c; i++;
    while( ( c = getc(in) ) != '\n' ) {
      temp[i] = c; i++;
      if( c == ':' )
        numFeature++;
    }
    temp[i] = '\0';
    numExample++;
    nonZeroFeature[exampleIndex] = numFeature;                      400
```

```
if ( numFeature != 0 ) {
  /*************************************************************

     Allocate memory for id/value pairs of the test data

   *************************************************************/
  example[exampleIndex] = (FeaturePtr *)malloc( numFeature *
                          sizeof( FeaturePtr ) );
  if ( example[exampleIndex] == NULL ) {                                    410
    fprintf( stderr, "Memory allocation failed\n");
    return 0;
  }
  for (j = 0; j < numFeature; j++) {
    example[exampleIndex][j] = (FeaturePtr)
                               malloc (sizeof(struct feature));
    if ( example[exampleIndex][j] == NULL ) {
      fprintf( stderr, "Memory allocation failed\n");
      return 0;
    }                                                                       420
  }
}
else {
   example[exampleIndex] = (FeaturePtr *)
                           malloc( sizeof( FeaturePtr ) );
   if ( example[exampleIndex] == NULL ) {
     fprintf( stderr, "Memory allocation failed\n");
     return 0;
   }
   example[exampleIndex][0] = (FeaturePtr)                                  430
                              malloc( sizeof(struct feature) );
   if ( example[exampleIndex][0] == NULL ) {
     fprintf( stderr, "Memory allocation failed\n");
     return 0;
   }
   example[exampleIndex][0]->id = 1;
   example[exampleIndex][0]->value = 0;
   nonZeroFeature[exampleIndex] = 0;
   zeroFeatureExample[zeroFeatureNumber] = exampleIndex;
   zeroFeatureNumber++;                                                     440
}

/*********************************

   Extract class of example

 *********************************/
i = 0;
while( temp[i] != ' ' ) {
  temp2[i] = temp[i]; i++;                                                  450
}
temp2[i] = '\0';
target[exampleIndex] = atoi(temp2);
i++;

if ( numFeature != 0 ) {
  /*************************************************************

     Extract id/value pairs of the features of the test data
                                                                            460
   *************************************************************/
  featureIndex = 0;
  while( temp[i] != '\0' ) {
    j = 0;
```

```
        while( temp[i] != ':' ) {
          temp2[j] = temp[i]; i++; j++;
        }
        temp2[j] = '\0';
        example[exampleIndex][featureIndex]->id = atoi(temp2);
        j = 0; i++;
        while( temp[i] != ' ' && temp[i] != '\0' ) {
          temp2[j] = temp[i];
          i++; j++;
        }
        temp2[j] = '\0';

        if( atof(temp2) != 0 ) {
          example[exampleIndex][featureIndex]->value = atof(temp2);
          featureIndex++;

        }
      } /* end while */

      nonZeroFeature[exampleIndex] = featureIndex;
      if( example[exampleIndex][featureIndex - 1]->id > maxFeature )
        maxFeature = example[exampleIndex][featureIndex - 1]->id;
    }
  } /* end else */
} /* end of while not EOF */

/*********************************************************************

  If the maxFeature read from model file is less than the number of
  features the test data has, then extend the weight vector.

 *********************************************************************/
if ( maxFeatureRead < maxFeature ) {
  weight = realloc( weight, maxFeature+1 );
  for ( i = maxFeatureRead+1; i <= maxFeature; i++ )
    weight[i] = 0;
}

return 1;
}
```

```
/*********************      classify.h      ******************************

Purpose: Header for program classification module.

Programmer: Ginny Mak
Location: Montreal, Canada.
Last Updated: April 6, 2000.

**/

/*********************  Public functions  ************************/

extern int writeResult( FILE *out );

extern int classifyLinear( FILE *out );

extern int classifyPoly ( FILE * out );

extern int classifyRbf ( FILE * out );
```

```
/************************** classify.c ************************

Purpose: Classify a given set of test.
         The classification result is written to the prediction file.

Notes: None.
Programmer: Ginny Mak
Location: Montreal, Canada.
Last Updated: April 6, 2000
**/

#include <stdio.h>
#include <time.h>
#include <math.h>
#include "classify.h"
#include "initialize.h"

/************************ Declared Functions ***********************/

static double dotProduct( FeaturePtr *x, int sizeX, FeaturePtr *y, int sizeY );
static double power( double x, int n );
static double wtDotProduct( double *w, int sizeX, FeaturePtr *y, int sizeY );
static void freeMemory( void );


/************************** Private Functions **********************

/*FN**************************************************************************

static double dotProduct( FeaturePtr *x, int sizeX, FeaturePtr *y, int sizeY )

Returns: the dot product of two given vectors

Purpose: Calculate the dot product of two given data vectors.

Note: This function does not change the passed arguments.
**/

static double dotProduct( FeaturePtr *x, int sizeX, FeaturePtr *y, int sizeY )
{
  int num1, num2, a1, a2;
  int p1 = 0; int p2 = 0;
  double dot = 0;

  if( sizeX == 0 || sizeY == 0 )
    return 0;
  num1 = sizeX; num2 = sizeY;
  while( p1 < num1 && p2 < num2 ) {
    a1 = x[p1]->id;
    a2 = y[p2]->id;
    if( a1==a2 ) {
      dot += (x[p1]->value) * (y[p2]->value);
      p1++; p2++;
    }
    else if ( a1 > a2 )
      p2++;
    else
      p1++;
  }
  return dot;
}
```

```
/*FN*************************************************************************

double wtDotProduct( double *w, int sizeX, FeaturePtr *y, int sizeY )

Returns: double – the dot product between the weight vector  and an example.

Purpose: To speed up classifying by just using the weight vector
         when using linear kernel.

Notes: This function only applies to linear kernel.
**/

static double wtDotProduct( double *w, int sizeX, FeaturePtr *y, int sizeY )
{
  int num1, num2, a2;
  int p1 = 1; int p2 = 0;
  double dot = 0;

  if( sizeX == 0 || sizeY == 0 )
    return 0;
  num1 = sizeX; num2 = sizeY;
  while( p1 <= num1 && p2 < num2 ) {
    a2 = y[p2]->id;
    if( p1==a2 ) {
      dot += (w[p1]) * (y[p2]->value);
      p1++; p2++;
    }
    else if ( p1 > a2 )
      p2++;
    else
      p1++;
  }
  return dot;
}


/*FN*************************************************************************

static double power( double x, int n )

Returns: double – the value of x raised to the power of n

Purpose: To calculate x raised to the power of n.

Notes: x can be positive or negative. n >= 0
**/

static double power( double x, int n )
{
  int i;
  double p;

  p = 1.0;
  for ( i = 1; i <= n; i++ )
    p = p*x;
  return p;
}

/*FN*************************************************************************

static void freeMemory( void )
```

Returns: Nothing.

Purpose: Free memory allocated in the program.

Notes: None.
**/

```
static void freeMemory( void )
{
  int i;

  free( lambda );
  free( svNonZeroFeature );
  free( nonZeroFeature );
  free( target );
  free( weight );
  free( output );
}
```

/************************** *Public functions* **************************/

/*FN***

*int writeResult(FILE *out)*

Returns: int – Fail(0) on errors, Succeed(1) otherwise

Purpose: To call classification functions to classify the test data according to the kernel type and then write the results to the prediction file.

Note: Results are written to the prediction file.
**/

```
int writeResult( FILE *out )
{
  int result;

  if ( kernelType == 0 )
    result = classifyLinear( out );

  else if ( kernelType == 1 )
    result = classifyPoly( out ) ;

  else if ( kernelType == 2 )
    result = classifyRbf( out );

  if (result)
    return 1;
  return 0;
}
```

/*FN***

*int classifyLinear(FILE *out)*

Returns: int – Fail(0) on errors, Succeed(1) otherwise

Purpose: Classify the test data using a linear kernel and write the classification result to the prediction file.

```
Notes: None.
**/

int classifyLinear( FILE *out )
{
  int i;
  double startTime;

  printf("Start classifying . . .\n");
  startTime = clock()/CLOCKS_PER_SEC;
  for( i = 1; i <= numExample; i++ )
    output[i] = wtDotProduct( weight, maxFeature, example[i],
                              nonZeroFeature[i] );

  printf( "Classifying time is %f seconds\n",
          clock()/CLOCKS_PER_SEC - startTime );
  printf( "Finish classifying.\n");
  for ( i = 1; i <= numExample; i++ )
    fprintf( out, "%18.17f\n", output[i] - b );
  return 1;
}

/*FN*************************************************************************

int classifyPoly(  FILE *out )

Returns: int - Fail(0) on errors, Succeed(1) otherwise

Purpose: Classify the test data using a polynomail kernel and write
         the classification results to the prediction file.

Notes: None.
**/


int classifyPoly( FILE *out )
{
  int i, j;
  double startTime;

  printf("Start classifying . . .\n");
  startTime = clock()/CLOCKS_PER_SEC;
  for( i = 1; i <= numExample; i++ ) {
    output[i] = 0;
    for( j = 1; j <= numSv; j++ )
      output[i] += lambda[j] * power(1 + dotProduct(sv[j],svNonZeroFeature[j],
                   example[i],nonZeroFeature[i]), degree );
  }
  printf( "Classifying time is %f seconds\n",
          clock()/CLOCKS_PER_SEC - startTime );
  printf( "Finish classifying.\n");
  for ( i = 1; i <= numExample; i++ )
    fprintf( out, "%18.17f\n", output[i] - b );
  return 1;
}


/*FN*************************************************************************
int classifyRbf(  FILE *out )
```

Returns: int – Fail(0) on errors, Succeed(1) otherwise

Purpose: Classify the test data using rbf kernel and write the classification results to the prediction file.

Notes: None.
**/

```
int classifyRbf( FILE *out )
{
  int i, j;
  double devSqr;
  double startTime;

  printf("Start classifying . . .\n");
  startTime = clock()/CLOCKS_PER_SEC;
  for( i = 1; i <= numExample; i++ ) {
    output[i] = 0;
    for( j = 1; j <= numSv; j++ ) {
      devSqr = dotProduct(sv[j],svNonZeroFeature[j],sv[j],svNonZeroFeature[j])
             - 2*dotProduct(sv[j],svNonZeroFeature[j], example[i],
                            nonZeroFeature[i] )
             + dotProduct( example[i],nonZeroFeature[i], example[i],
                           nonZeroFeature[i] );
      output[i] += lambda[j] * exp( - devSqr*rbfConstant );
    }
  }
  printf( "Classifying time is %f seconds\n",
          clock()/CLOCKS_PER_SEC - startTime );
  printf( "Finish classifying.\n");
  for( i = 1; i <= numExample; i++ )
    fprintf( out, "%18.17f\n", output[i] - b );
  return 1;
}
```