Theses   Theses and Dissertations

Summer 2004

SLIMSVM : a simple implementation of support vector machine for analysis of microarray data

Avik Karmaker

New Jersey Institute of Technology

ABSTRACT

SLIMSVM: A SIMPLE IMPLEMENTATION OF SUPPORT VECTOR MACHINE FOR ANALYSIS OF MICROARRAY DATA

by
Avik Karmaker

Support Vector Machine (SVM) is a supervised machine learning technique being widely used in multiple areas of biological analysis including microarray data analysis. *SlimSVM* has been developed with the intention of replacing *OSU SVM* as the classification component of *GenoIterSVM* in order to make it independent of other SVM packages. *GenoIterSVM*, developed by Dr. Marc Ma, is a SVM implementation with an iterative refinement algorithm for improved accuracy of classification of genotype microarray data. *SlimSVM* is an object-oriented, modular, and easy-to-use implementation written in C++. It supports dot (linear) and polynomial (non-linear) kernels. The program has been tested with artificial non-biological and microarray data. Testing with microarray data was performed to observe how *SlimSVM* handles medium-sized data files (containing thousands of data points) since it would ultimately be used to analyze them. The results were compared to those of *LIBSVM*, a leading SVM software, and the comparison demonstrates that implementation of *SlimSVM* was carried out accurately.

SLIMSVM: A SIMPLE IMPLEMENTATION OF SUPPORT VECTOR MACHINE FOR ANALYSIS OF MICROARRAY DATA

by
Avik Karmaker

A Thesis
Submitted to the Faculty of
New Jersey Institute of Technology
In Partial Fulfillment of Requirements for the Degree of
Master of Science in Computational Biology

Department of Computer Science

August 2004

Blank Page

APPROVAL PAGE

SLIMSVM: A SIMPLE IMPLEMENTATION OF SUPPORT VECTOR MACHINE FOR ANALYSIS OF MICROARRAY DATA

Avik Karmaker

Dr. Marc Q. Ma, Thesis Advisor   Date
Assistant Professor of Computer Science, NJIT

Dr. Usman W. Roshan, Committee Member   ′ Date
Assistant Professor of Computer Science, NJIT

Dr. Frank Shih, Committee Member   Date
Professor of Computer Science, NJIT

BIOGRAPHICAL SKETCH

Author: Avik Karmaker

Degree: Master of Science

Date: August 2004

Education:

- Master of Science in Computational Biology
  New Jersey Institute of Technology, Newark, NJ, 2004

- Bachelor of Science in Biology
  Binghamton University, Binghamton, NY, 2001

Major: Computational Biology

"Brightest flame burns the quickest."
In memory of my mother.

ACKNOWLEDGEMENT

TABLE OF CONTENTS

TABLE OF CONTENTS
(Continued)

LIST OF TABLES

LIST OF SYMBOLS

$\mathbb{R}$	Real numbers
$y \in Y$	Output and output space
$\mathbf{x} \in X$	Input and input space
$\mathbb{F}$	Feature space
$\mathbf{\Phi} : \mathbb{R}^N \to \mathbb{F}$	Mapping to feature space
$\mathbf{w}$	Weight vector
b	Bias
$\boldsymbol{\alpha}$	Dual variables or Lagrange multiplier
L	Primal Lagrangian
W	Dual Lagrangian
l	Training set size
h	VC dimension
ξ	Slack variables

CHAPTER 1

INTRODUCTION

In the past several years, a new technology known as microarray has received a great amount of attention from biologists and biomedical researchers. Microarray expression experiments allow the recording of expression levels of thousands of genes, giving scientists a better picture of simultaneous interactions among these genes. These experiments can be categorized into two general groups. The first group of experiments consists of monitoring each gene several times under various conditions (Chu, 1998; Derisi, 1997). This type of experiments has allowed for the identification of genes that are functionally related to common expression patterns (Brown, 2000; Wen, 1998). The second group of experiments evaluates each gene in a single environment but in different types of tissues (Derisi, 1996; Golub, 1999; Zhu, 1998). This type of experiments could be used to identify genes whose expression levels help to diagnose diseases (Golub, 1999).

Support Vector Machine (SVM) is a machine learning technique based on the Structural Risk Minimization Principle (Vapnik, 1995). SVM is used for various purposes, some of which include image detection, 3-D object recognition, hand writing detection, text categorization, speech recognition, and temperature prediction. It has also been shown to perform well in various areas of biological research such as protein structure prediction, alternative splice site prediction, breast cancer prognosis, and microarray analysis.

Recently, Dr. Marc Ma (NJIT), in collaboration with Dr. Honghua Li (Cancer Institute of New Jersey), developed *GenoIterSVM* for analysis of microarray data (Ma et al., 2004). *GenoIterSVM* integrates SVM classification algorithm with an iterative refinement algorithm for improved accuracy of genotype classification using microarray data. *OSU SVM Classifier Matlab Toolbox (Version 3.00)* is used as the SVM classification component of *GenoIterSVM*. *OSU SVM* is based on *LIBSVM Version 2.33*. Analysis of microarray data with *GenoIterSVM* showed that dot kernel (linear classification) is the most effective kernel for genotype classification. In order to make *GenoIterSVM* a stand-alone distribution independent of other SVM packages, *SlimSVM*, a C++ implementation based on the algorithms of *LIBSVM* version 2.5, was developed with the intention of replacing *OSU SVM* as the SVM component. Eventually, *GenoIterSVM* would be developed in MATLAB for genotype microarray data analysis, and implementation of *SlimSVM* is the first step towards creating a working SVM implementation in MATLAB. There are many implementations of SVM available in different programming languages that allow the user to perform data classification and regression analysis of data using different options. A brief description of some of the implementations is provided below.

GIST is a web-based support vector machine. It can be found at http://svm.sdsc.edu/cgi-bin/nph-SVMsubmit.cgi. Since it is run on a web server, its use is limited to small training and testing jobs. The interactive site also limits the number of concurrent jobs. Compared to most other SVM implementations, *GIST* uses different formats for input files and output files. Input files for both training and testing are tab-delimited text files where the first row consists of a feature name or id and the first

column consists of data point names. The rest of the file contains a matrix of numbers where each value corresponds to a feature of a data point. Unlike other SVMs, the class labels for the training data points are put in a tab-delimited text file and submitted separately to the server for training. For training purposes, *GIST* provides several options for normalizing data. The user can also select the kernels for training. The default is dot kernel and the other two kernels supported are polynomial and Radial Basis Function (RBF). The website allows the user to enter values for the parameters of the kernels depending on the kernel selected. The training output file consists of all the data points from the training input file. The support vectors are indicated by nonzero weight values. For each of the data points, a value for the discriminant, which indicates the distance of the data point from the hyperplane, is provided. The classification output file contains the predicted class for each of the data points and the discriminant value. Unlike other SVM implementations, *GIST* does not provide any documentation on the algorithms used and it can only perform binary classification.

SVMsequel is a SVM implementation in Haskell. The program is based on the kernel minover algorithm. Unlike other SVM implementations, it provides a shell for executing a set of predefined commands for training and classifications instead of executables, as well as more kernel options. Besides the typical kernels such as linear, polynomial, RBF and sigmoid kernels, it includes information diffusion kernels, diffusion kernels on graphs, string kernels and tree kernels. Furthermore, it allows the user to combine kernels. It can perform both binary classification and multi class classification. *SVMsequel* supports heterogeneous data; it can handle data consisting of both strings and real values. The different data types are separated and grouped together for training and

classification. The data sets are prepared by using commands. The data sets consisting of strings are in dense format and those of real values are in sparse format. It also allows the user to normalize the data. For training, the user can enter the values for parameters or can perform cross-validation to estimate the parameters. Once training is completed the model can be saved in a file for classification. After completion of training, the results displayed on screen include the values of parameters, type of kernel used, and number of support vectors. When classification is performed, the result of the classification is saved in a file specified by the user or it can be displayed on screen. Even though *SVMsequel* offers more kernel options and supports a heterogeneous data format for input file, the program is heavily command based and the user would need to rely immensely on the user manual to use the program.

SVM^{Light} is an implementation of SVM in C. It has been developed by Thorsten Joachims. The optimization technique used in SVM^{Light} is used by many other implementations of SVM. The algorithm performs working set selection based on steepest feasible descent. It also contains shrinking heuristics and use of folding in the linear case. The kernel evaluations are cached, making the program more memory efficient. In addition to solving classification problems, SVM^{Light} solves regression and ranking problems. The latest version of SVM^{Light} also includes an algorithm for approximately training large transductive SVMs. The optimization algorithm and the caching technique enable SVM^{Light} to handle several hundred-thousand data points. It includes dot, polynomial, RBF, and sigmoid kernels. In addition, the user can enter his own kernels. SVM^{Light} supports sparse data format for the input files for both training and classification. Since SVM^{Light} is command based program, training requires the user to

enter the training parameters, the name of the input file containing training data, and the file name for writing the training model through stdin. For classification, the user has to enter the names of the file containing the data to be classified, the model file, and the file where the result of classification is to be written.

mySVM is based on the optimization algorithm of SVM^{Light}. While it is implemented in C, there is also a java version available. In addition to pattern recognition, it supports regression estimation and distribution estimation. It consists of two modules-one for training called *mySVM* and another for classification called *predict*. What makes *mySVM* unique is that it supports multiple formats for the input file and also allows the user to use stdin to input the data if the user chooses not to use an input file. The input file comprises SVM parameters, kernel parameter, and data sets. For the input file, SVM parameters and at least one set of data for training must be provided. If no kernel parameter is provided, dot kernel is used as the default. In addition to dot kernel, *mySVM* provides polynomial kernel, radial kernel, neural kernel, and anova kernel. It also lets the user enter his own kernels. If more than one set of data is provided in the input file, the first set is used for training and the subsequent sets are used either for testing or predicting, depending on whether Y value is provided for these data sets. *mySVM* performs cross validation on the training examples as well. The data points of the input file can be either in sparse format or in dense format. *mySVM* supports multiple formats such as xy, yx, and xyx for the dense format. The training output is shown on the screen. The statistics printed include total number of support vectors, bounded support vectors, and the minimal and maximal values of the alphas. The optimization result of training is

saved in a file in the format of the input file. When testing is performed, the predicted class labels are saved in a file.

LIBSVM is a library of support vector machines in C developed by Chih-Chung Chang and Chih-Jen Lin. It is based on SVM^{Light} and the sequential minimal optimization algorithm. In addition to performing support vector classification, it performs support vector regression and one-class distribution estimation. In terms of classification it supports C-SVM and nu-SVM and for regression it implements epsilon-SVR and nu-SVR. It supports dot, polynomial, RBF, and sigmoid kernels but does not allow the user to enter his own kernels. *LIBSVM* supports both binary and multi-class classification. Like SVM^{Light}, *LIBSVM* also implements shrinking heuristics. The input for both training and classification are in the same format as those for SVM^{Light}. It supports sparse data format for the input files and also implements a caching technique for efficient memory handling. It offers cross-validation for model selection and performs probability estimates for the classification and regression functions. The user can use weighted cost for unbalanced classes, as well as many other tools for performing additional functions using SVM.

Since further development of *GenoIterSVM* would be carried out by Dr. Ma's group, the design and implementation of *SlimSVM* emphasized ease of use and minimalism while maintaining accuracy and efficiency of training and classification. It supports dot kernel for linear classification and polynomial kernel for non-linear classification. The polynomial kernel was implemented since it is the simplest kernel that can perform non-linear classification. *SlimSVM* is menu based and prompts the user for input as needed. It can perform binary and multi-class classification and uses sparse data

format for the input file. The implementation was tested with artificial non-biological data and microarray data used in (Ma et al., 2004). Testing with microarray data was carried out to observe how *SlimSVM* handles medium sized data files (thousands of data points) since ultimately *SlimSVM* would be used to analyze microarray data.

The purpose of this thesis is to provide a comprehensive description of the mathematical basis of SVM and the development process of *SlimSVM*. The thesis consists of seven chapters. In the second chapter, a theoretical overview of SVM is presented. In the third chapter, a description of the dual form of SVM is given. The implementation techniques and algorithms used are described in the fourth chapter. The fifth chapter consists of the description of the software files and the development process. In the sixth chapter, the description and results of testing are provided. The thesis concludes with a summary of the results of developing *SlimSVM* and possible future work with it.

CHAPTER 2

SUPPORT VECTOR MACHINES

2.1 Theory

Support Vector Machine is a supervised learning method. Given the task of classification of data, the objective is to find a rule which will assign the data into several classes based on characteristics of the data. In the simplest case, the data can be separated into two different classes. The task can be formalized as an estimation of the function $f: \mathbb{R}^N \rightarrow \{-1, 1\}$ using input-output training data

$$(\mathbf{x}_1, y_1), ..., (\mathbf{x}_l, y_l) \in \mathbb{R}^N \times Y, \quad Y = \{-1, +1\} \tag{2.1}$$

such that f will correctly classify unseen examples $(\mathbf{x}, y)$. An unseen example will be assigned to class $+1$ if $f(\mathbf{x}) \geq 0$ and to class -1 otherwise.

The unseen or test examples are assumed to have the same probability distribution $P(\mathbf{x}, y)$ as the training data. The best function f that can be obtained is the one that minimizes expected error, or risk R, which is the possible average error from f on the test examples drawn randomly from the sample distribution $P(\mathbf{x}, y)$:

$$R = \int \frac{1}{2} |f(\mathbf{x}) - y| dP(\mathbf{x}, y) \tag{2.2}$$

However, since the underlying probability distribution of the test examples is not known, a function close to the optimal one has to be estimated based on the training examples and function class F the solution f is chosen from. This estimation is done by approximating the minimum of the risk (Boser et. al, 1992) by the training error or empirical risk:

$$R_{emp}[f] = \left(\frac{1}{l}\right)\sum_{i=1}^{l}\frac{1}{2}\left|f(\mathbf{x}_i) - y\right|$$
(2.3)

It is possible to have the empirical risk converge towards the expected risk by placing conditions on the learning machines as the number of sample data increases. However, with small sample sizes for training the problem of over fitting could arise as illustrated in the figure below:
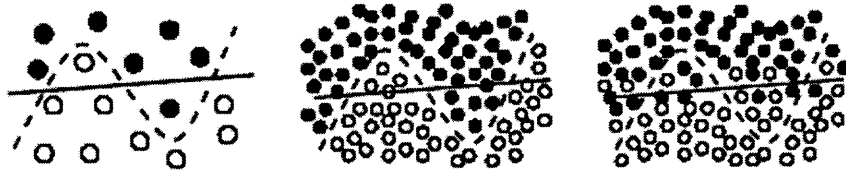
Figure 2.1 Example of over fitting dilemma. When the sample size is small, either the dashed line or the solid line could be the decision function. The solid line is less complex and less accurate since it misclassifies two data points (left). When the sample size is large, the solid line under fits the data if the dashed line is correct (middle). When the sample size is large, the dashed line over fits the data if the solid line is correct (right) (Muller, 2001).

The problem of over fitting can be overcome by reducing the complexity of the decision function f (Vapnik, 1995). In most training methods, the classifier tends to minimize misclassification error by increasing the complexity of the decision function. Complex decision functions result in small training errors, which imply small misclassification errors for the unseen data. By making the decision function complex, the classifier tends to produce a decision surface biased by the training sample and it becomes less flexible for the unseen data. The figure below illustrates this problem:
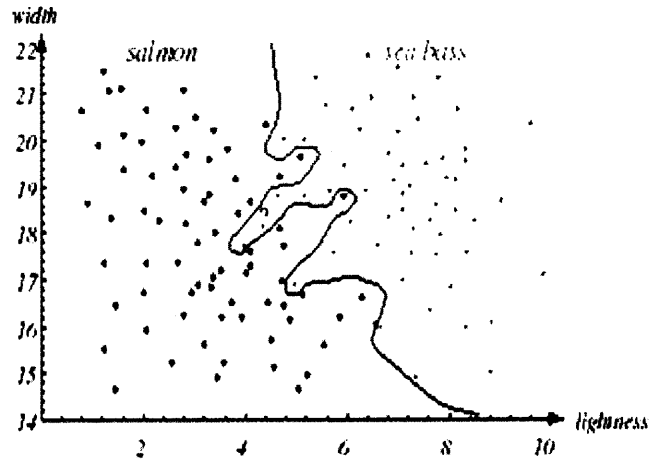
Figure 2.2 Training sample biased decision surface. Overly complex models for the fish will lead to decision boundaries that are complicated. While such a decision may lead to perfect classification for the training samples, it would lead to poor performance on unseen data (Duda et al.1973).

In the figure above, the decision boundary is not linear. According to the decision boundary the unseen data denoted by "?", should belong to the class "sea bass." However, by visual inspection, it is more likely to belong to the "salmon" class. The complexity of a decision function can be specifically controlled by VC theory and the SRM principle (Vapnik, 1998).

2.1.1 VC Theory

VC (Vapnik and Chervonenkis) theory shows that it is imperative to restrict the class of functions that f is chosen from to one that has a capacity suitable for the amount of available training data (Advances in Kernel Methods, 1). According to the VC theory, the concept of complexity mentioned earlier is captured by the VC dimension of the function class F that the estimate f is chosen from. The VC dimension is defined as the largest number h of data points from the training sample that can be shattered or separated for all

possible ways using functions of the class; this provides bounds on the training error. The minimization of these bounds leads to the SRM principle (Vapnik, 1995).

2.1.2 SRM Principle

Structural Risk Minimization (SRM) chooses function class F and the function f such that an upper bound on the training error is minimized. This can be computed using the following theorem:

Let h denote VC dimension of the function class F and let R_{emp} be defined by (2.3). For all $\delta > 0$ and $f \in F$ the inequality bounding the risk

$$R[f] \leq R_{emp}[f] + \sqrt{\frac{h\left(\ln\frac{2l}{h} + 1\right) - \ln\left(\delta/4\right)}{l}} \tag{2.4}$$

holds with probability of at least 1- δ for h< l.

The goal of SRM is to minimize the generalization error given by $R[f]$ by getting a small training error $R_{emp}[f]$ while keeping the function class as small as possible. Two extreme conditions arise for (2.4):

  i)   a very small class may give a vanishing square root term but a large training error might remain

  ii)   a very large function class may give a vanishing empirical error but a large square root term.

The best class is usually the one in between the two extremes which would yield a function that explains the data well and involves a small risk in getting that function. This can be illustrated by the diagram below:
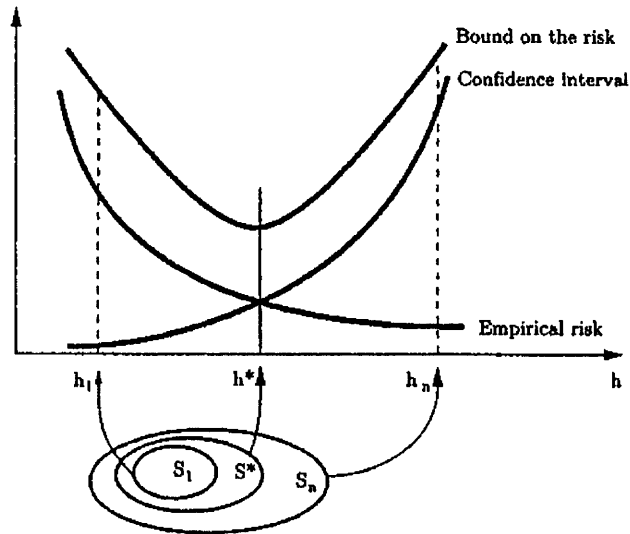
Figure 2.3 Schematic illustration of Equation 2.4(Vapnik, 1996).

In the above diagram, the vertical axis represents overall error and the horizontal axis represents complexity of the decision function. The overall error decreases as the complexity of the decision function increases. However, when the complexity of the function goes beyond a certain point the overall error begins to increase. Since the goal is to find the best tradeoff between empirical error and complexity, $h*$ should be chosen as the complexity of the decision function since it is in between the two extremes.

Support Vector Machines, SVM, are able to achieve the goal of minimizing the upper bound of $R[f]$ by minimizing a bound on the VC dimension h and $R_{emp}[f]$ simultaneously during training. The design of SVMs is based on a set of hyperplanes.

2.2 Hyperplane Classifiers

To design learning algorithms with a class of functions whose capacity can be computed, Vapnik and Chervonenkis considered functions of the form:

$$f(\mathbf{x}) = (\mathbf{w} \cdot \mathbf{x}) + b \qquad \mathbf{w} \in \mathbf{R}^N, b \in \mathbf{R} \tag{2.5}$$

corresponding to the decision functions:

$$f(\mathbf{x}) = \mathrm{sgn}(\mathbf{w} \cdot \mathbf{x}) + b \qquad\qquad (2.6)$$

According to Statistical Learning Theory (Vapnik, 1998), for the class of hyperplanes, the VC dimension h itself can be bounded in terms of another quantity called *margin*. The margin is the minimal distance from the hyperplane to the closest points of the two classes. It can be measured by the length of the weight vector $\mathbf{w}$ in Equations 2.5 and 2.6. The weight vector, $\mathbf{w}$, is perpendicular to the hyperplane as shown in the diagram below:
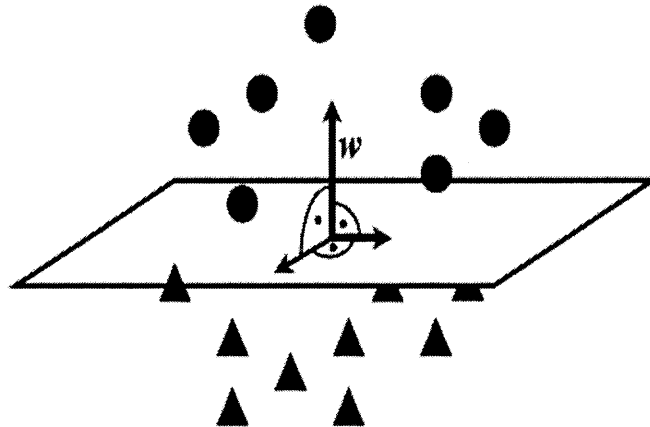
Figure 2.4 Weight vector diagram. The circles and the triangles belong to two different classes. The normal to the hyperplane is the $\mathbf{w}$ vector (Source: Mukherjee).

In order to get a better understanding of how a hyperplane and margins can be used to separate two classes, let's look at a simple problem of separating circles and rectangles. In the following diagram (Figure. 2.5), the hyperplane is represented by a solid line. It is located halfway between the two dotted lines. Since the problem can be separated into two classes, there is a weight vector $\mathbf{w}$ and a threshold b such that $y_i \cdot ((\mathbf{w} \cdot \mathbf{x}_i) + b) > 0$ where $i = 1,\ldots,l$. The threshold b is the minimum distance from the origin to

the hyperplane. The values of **w** and *b* can be rescaled to obtain points closest to the hyperplane that satisfy the following:

$$|(\mathbf{w} \cdot \mathbf{x}_i) + b| = 1 \tag{2.7}$$

to obtain a canonical form (w, b) of the hyperplane satisfying:

$$y_i \cdot ((\mathbf{w} \cdot \mathbf{x}_i) + b) \geq 1 \tag{2.8}$$

When considering samples from two different classes, in this case rectangles(x_1) and circles(x_2):

$$(\mathbf{w} \cdot \mathbf{x}_1) + b = 1 \tag{2.9}$$

$$(\mathbf{w} \cdot \mathbf{x}_2) + b = -1 \tag{2.10}$$

The margin is given by the distance of these two points from the hyperplane, measured perpendicularly, and it equals $\dfrac{2}{\|\mathbf{w}\|}$. The points on the margins are called the support vectors.
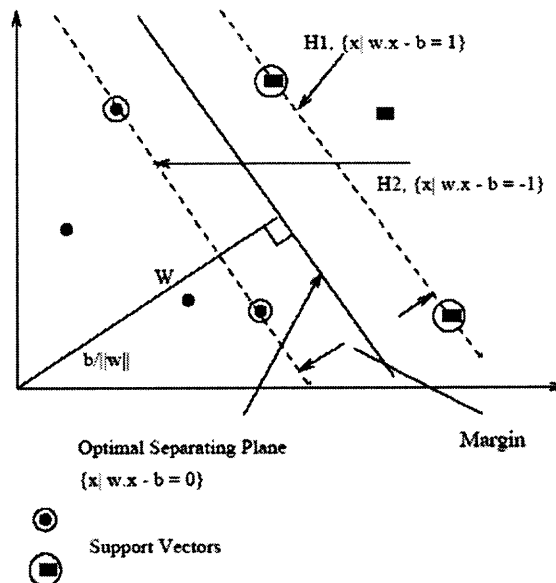
Figure 2.5 Hyperplane and margins.

The goal is to maximize the distance between the hyperplane and the closest points on the margins with the constraint that there are two margins on opposite sides of the hyperplane. The objective can be achieved by the following optimization problem:

$$\max_{\mathbf{w}} \min_{x_i} \frac{y_i(\mathbf{w}\cdot\mathbf{x}_i)}{\|\mathbf{w}\|} \text{ subject to } y_i(\mathbf{w}\cdot\mathbf{x}_i)\geq 1, i = 1, ..., l \tag{2.11}$$

An equivalent but simpler form of 2.11 is the following (Vapnik, 1998):

$$\min \tau(\mathbf{w}) = \frac{1}{2}\|\mathbf{w}\|^2 \text{ subject to } y_i(\mathbf{w}\cdot\mathbf{x}_i)\geq 1, i = 1, ..., l \tag{2.12}$$

Support Vector Machine finds the hyperplane that maximizes the margins. The following diagrams show the advantage of maximizing the margins.
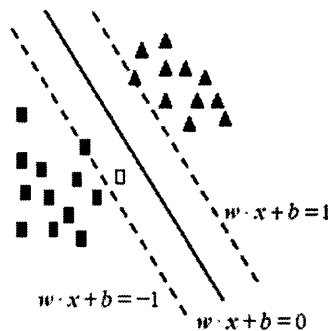
Figure 2.6 Advantage of maximized hyperplane. The solid line represents the maximized margin $\mathbf{w}\cdot\mathbf{x} + b = 0$. The points on the dashed lines are the support vectors and the lines are represented by $\mathbf{w}\cdot\mathbf{x} + b = \pm 1$. using the maximized hyperplane, a new point represented by the blank rectangle is classified correctly(Mukherjee).
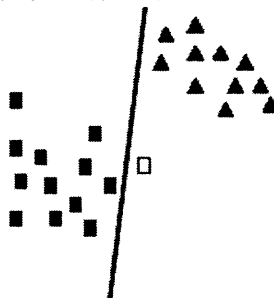
Figure 2.7 Disadvantage of non-maximized hyperplane. The solid line represents the hyperplane. The data points are same as the above diagram but since the hyperplane is not maximized, the blank rectangle is misclassified (Mukherjee).

2.3 Linearly Nonseparable Data

In practice, the input data may not be linearly separable, and as a result a linear separating hyperplane may not be found. This might happen if there is a high noise level, which causes the classes to overlap. If a linearly separable hyperplane doesn't exist, a linear decision function can still be found using a set of variables called slack variables (Vapnik, 1995), ξ,

$$\xi_i \geq 0, \ I = 1, \ldots, l \tag{2.13}$$

along with relaxed constraints:

$$y_i \cdot ((\mathbf{w} \cdot \mathbf{x}_i) + b) \geq 1 - \xi_i, \ i = 1, \ldots, l \tag{2.14}$$

With the inclusion of ξ, 2.12 becomes

$$\tau(\mathbf{w}, \xi) = \frac{1}{2} \|\mathbf{w}\|^2 + C \sum_{i=1}^{l} \xi_i \tag{2.15}$$

subject to the constraints 2.13 and 2.14. The Equation 2.15 trades off the two goals of finding a hyperplane with minimum $\|\mathbf{w}\|$ and finding a hyperplane that separates the data well by minimizing ξ_i. The parameter C in 2.15 regulates this tradeoff.
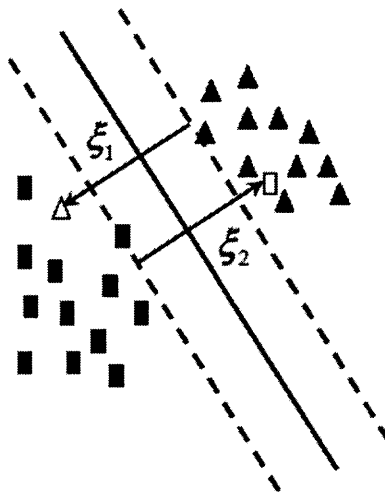
Figure 2.8 Nonlinearly separable data with smaller C (Mukherjee).

If the value for C is large, the margin is small but classification is more accurate. On the other hand, if the value for C is small, the margin is large and there could possibly be classification errors. This is demonstrated by Figures 2.8 and.2.9. In Figure 2.8, with smaller C, the margin is large and as a result the blank rectangle and the blank triangle are misclassified. In Figure 2.9, with larger C the margin is small, the blank rectangle and the blank triangle are correctly classified. The dotted line represents the solution of the classifier. For both figures, the slack variables, ξ, represent the distance of the misclassified points from the dashed lines for the corresponding classes.
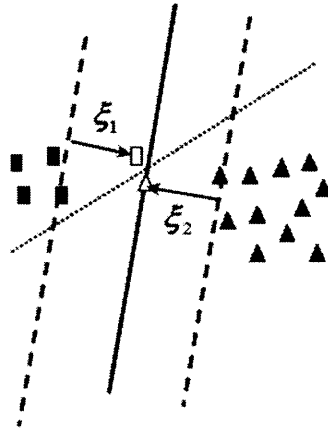
Figure 2.9 Nonlinearly separable data with larger C (Mukherjee).

2.4 Kernel and Feature Spaces

In most cases of classification, the separating plane is non linear. The theory of SVM can be extended to handle such cases. This is done by mapping the input data x_1, ... , $x_l \in \mathbb{R}^N$ into a potentially much higher dimensional feature space $\mathbb{F}$ (Boser et. al, 1992):

$$\Phi : \mathbb{R}^N \to \mathbb{F} \qquad (2.16)$$

$$x \mapsto \Phi(x) \qquad (2.17)$$

$$\mathbf{x} = \left(\mathbf{x}_1, ..., \mathbf{x}_n\right) \tag{2.18}$$

The linear SVM algorithm is applied to the higher dimensional feature space. The sample data can be described as:

$$\left(\Phi\left(\mathbf{x}_1\right), y_1\right), ..., \left(\Phi\left(\mathbf{x}_l\right), y_l\right) \in \mathbb{F} \times Y \tag{2.19}$$

The optimal separating linear hyperplane in the feature space corresponds to a nonlinear separating hyperplane in the original input space. This is demonstrated by the following diagram.
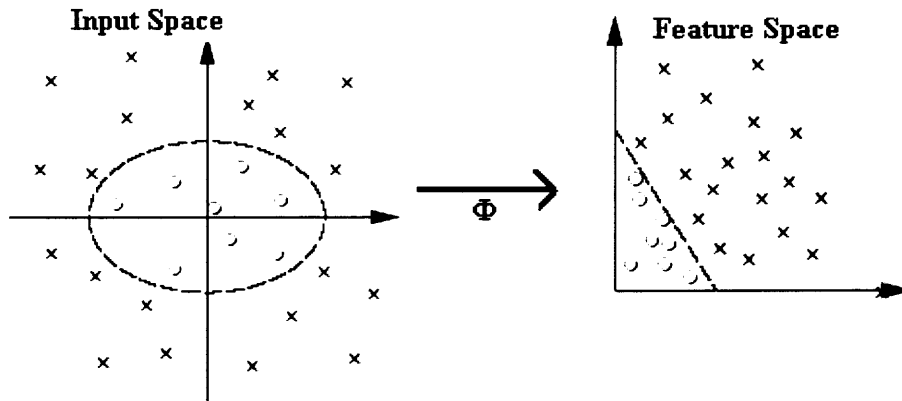
Figure 2.10 Mapping of training data to a feature space. In the input space the hyperplane is nonlinear(ellipsoidal), but when the data is mapped to a feature space the hyperplane is linear(Scholkopf, 2001).

With the introduction of mapping to feature space, Equation 2.5 can be rewritten as

$$f\left(\mathbf{x}\right) = \mathbf{w} \cdot \Phi\left(\mathbf{x}\right) + b \tag{2.20}$$

The normal to the hyperplane can also be rewritten as a linear combination of the training points in the feature space:

$$\mathbf{w} = \sum_{i=1}^{l} c_i \Phi\left(\mathbf{x}_i\right) \tag{2.21}$$

By substituting Equation 2.21 into Equation 2.20, Equation 2.20 can be rewritten as

$$f(\mathbf{x}) = \sum_{i=1}^{l} c_i \Phi(\mathbf{x}_i) \cdot \Phi(\mathbf{x}) + b \qquad (2.22)$$

The equation for decision function (Equation 2.6) can be rewritten as

$$f(\mathbf{x}) = \text{sgn}\left(\mathbf{w} \cdot \Phi(\mathbf{x})\right) + b = \text{sgn} \sum_{i=1}^{l} c_i \Phi(\mathbf{x}_i) \cdot \Phi(\mathbf{x}) + b \qquad (2.23)$$

A function K, called kernel function, can be used to simplify Equation 2.23. The kernel function can be defined as the dot product of two points in the feature space. The equation for the kernel function is written as (Advances in Kernel Methods, 4)

$$K\left(\mathbf{x}_i \cdot \mathbf{x}_j\right) \equiv \Phi(\mathbf{x}_i) \cdot \Phi(\mathbf{x}_j) \qquad (2.24)$$

By substituting Equation 2.24 into Equation 2.23, the decision function obtained is

$$f(\mathbf{x}) = \text{sgn} \sum_{i=1}^{l} c_i K\left(\mathbf{x}_i \cdot \mathbf{x}_j\right) + b \qquad (2.25)$$

The support vectors will have nonzero coefficients c_i since they are the closest points to the maximum margin hyperplane in the feature space. All other points will have a coefficient value of zero.

The following is an example of how mapping into a feature space and kernel function is connected (Mukherjee). Suppose in a microarray experiment the expression levels of two genes for breast cancer (BRCA1 and BRCA2) are measured. For each sample, the expression vector is $\mathbf{x} = (x_{BRCA1}, x_{BRCA2})$ and the following mapping into the feature space is used:

$$\Phi : \mathbf{x} \rightarrow \{x_{BRCA1}^2, x_{BRCA2}^2, \sqrt{2}x_{BRCA1}x_{BRCA2}, x_{BRCA1}, x_{BRCA2}, 1\}$$

If there are two samples $\mathbf{x}$ and $\mathbf{z}$, then the kernel function would be:

$$K(\mathbf{x}, \mathbf{z}) \equiv \Phi(\mathbf{x}) \cdot \Phi(\mathbf{z}) = (\mathbf{x} \cdot \mathbf{z} + 1)^2$$

$$= x_{BRCA1}^2 z_{BRCA1}^2 + x_{BRCA2}^2 z_{BRCA2}^2 + 2x_{BRCA1}x_{BRCA2}z_{BRCA1}z_{BRCA2} +$$
$$x_{BRCA1}z_{BRCA1} + x_{BRCA2}z_{BRCA2} + 1$$

This kernel function is called a second order polynomial kernel. This kernel uses information about expression levels of individual genes and also expression levels of pairs of genes which can be interpreted as a model that takes into account co-regulation information. The polynomial kernel is a commonly used kernel function and its general form can be written as

$$K(x,y) = ((x \cdot y) + \theta)^d \tag{2.26}$$

where d is the degree of the polynomial function.

CHAPTER 3

DUAL FORM OF SVM

As discussed in the previous chapter, the goal of SVM is to maximize the distance between the hyperplane and the closest points on the margins with the constraint that there are two margins on opposite sides of the hyperplane. The objective can be achieved by the following quadratic optimization problem

$$\min \tau(\mathbf{w}) = \frac{1}{2}\|\mathbf{w}\|^2 \text{ subject to } y_i(\mathbf{w}\cdot\mathbf{x}_i) \geq 1, i = 1, ..., l \tag{3.1}$$

This constrained optimization problem can be solved by introducing Lagrange multipliers $\alpha_i \geq 0$, $i = 1, ..., l$. The primal Lagrangian is

$$L(\mathbf{w}, b, \boldsymbol{\alpha}) = \frac{1}{2}\|\mathbf{w}\|^2 - \sum_{i=1}^{l} \alpha_i \left(y_i \cdot \left((\mathbf{x}_i \cdot \mathbf{w}) + b \right) - 1 \right) \tag{3.2}$$

The task is to minimize Equation 3.1 with respect to primal variables $\mathbf{w}$, b, and to maximize it with respect to dual variable α_i. The corresponding dual functions are found by differentiating with respect to $\mathbf{w}$ and b

$$\frac{\partial L(\mathbf{w}, b, \boldsymbol{\alpha})}{\partial \mathbf{w}} = \mathbf{w} - \sum_{i=1}^{l} y_i \alpha_i \mathbf{x}_i = 0 \tag{3.3}$$

$$\frac{\partial L(\mathbf{w}, b, \boldsymbol{\alpha})}{\partial b} = \sum_{i=1}^{l} y_i \alpha_i = 0 \tag{3.4}$$

Substituting the following relations obtained from Equations 3.3 and 3.4

$$\sum_{i=1}^{l} y_i \alpha_i = 0 \tag{3.5}$$

and

$$\mathbf{w} = \sum_{i=1}^{l} y_i \alpha_i \mathbf{x}_i \tag{3.6}$$

into Equation 3.2 eliminates the primal variables and the Wolfe dual of the optimization problem is found

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

$$= \frac{1}{2}\sum_{i,j=1}^{l}\alpha_i\alpha_j y_i y_j\left(\mathbf{x}_i\cdot\mathbf{x}_j\right) - \sum_{i,j=1}^{l}\alpha_i\alpha_j y_i y_j\left(\mathbf{x}_i\cdot\mathbf{x}_j\right) + \sum_{i=1}^{l}\alpha_i$$

$$= \sum_{i=1}^{l}\alpha_i - \frac{1}{2}\sum_{i,j=1}^{l}\alpha_i\alpha_j y_i y_j\left(\mathbf{x}_i\cdot\mathbf{x}_j\right) \tag{3.7}$$

In terms of the feature space, Equation 3.2 can be represented as

$$L(\mathbf{w},b,\alpha) = \frac{1}{2}\|\mathbf{w}\|^2 - \sum_{i=1}^{l}\alpha_i\left(y_i\cdot\left(\left(\Phi(\mathbf{x}_i)\cdot\mathbf{w}\right)+b\right)-1\right) \tag{3.8}$$

and Equation 3.6 as

$$\mathbf{w} = \sum_{i=0}^{l}\alpha_i y_i\Phi(\mathbf{x}_i) \tag{3.9}$$

By replacing $(\Phi(\mathbf{x}_i)\cdot\Phi(\mathbf{x}_j))$ with the kernel function $K(\mathbf{x}_i\cdot\mathbf{x}_j)$, the dual optimization problem (Equation 3.7) now becomes

$$\max_{\alpha}\sum_{i=1}^{l}\alpha_i - \frac{1}{2}\sum_{i,j=1}^{l}\alpha_i\alpha_j y_i y_j K\left(\mathbf{x}_i\cdot\mathbf{x}_j\right) \tag{3.10}$$

subject to

$$\alpha_i \geq 0,\, i = 1,\ldots,l \tag{3.11}$$

and

$$\sum_{i=1}^{l}y_i\alpha_i = 0 \tag{3.5}$$

By solving the quadratic dual optimization problem, the coefficients α_i are found, which are needed to express the $\mathbf{w}$ that solves Equation 3.1. The hyperplane decision function can be written as

$$f(\mathbf{x}) = \text{sgn}\left(\sum_{i=1}^{l} \alpha_i y_i \left(\Phi(\mathbf{x}_i) \cdot \Phi(\mathbf{x}_j)\right) + b\right) \tag{3.12}$$

$$= \text{sgn}\left(\sum_{i=1}^{l} \alpha_i y_i K(\mathbf{x}_i \cdot \mathbf{x}_j) + b\right) \tag{3.13}$$

The solution vector **w** thus has an expansion in terms of a subset of the training patterns, namely those patterns whose α_i is not zero, and these are the support vectors. By the Karush-Kuhn-Tucker(KKT) complimentarity conditions

$$\alpha_i \cdot \left[y_i \left((\mathbf{x}_i \cdot \mathbf{w}) + b\right) - 1 \right] = 0 \text{ where } i = 1, ..., l \tag{3.14}$$

the support vectors lie on the margin. All remaining data points in the training set are irrelevant because their constraint (Equation 3.1) does not play a role in the optimization and they do not appear in the expansion (Equation 3.9). This properly captures the intuition of SVM since the hyperplane is completely determined by the data points closest to it, but not by any of the other data points. The threshold value, b, from Equations 3.12 and 3.13 is computed using Equation 3.14.

As mentioned in the previous chapter, when dealing with linearly non separable data, Equation 3.1 is written as (Joachims, 1998)

$$\tau(\mathbf{w}, \xi) = \frac{1}{2} \|\mathbf{w}\|^2 + C \sum_{i=1}^{l} \xi_i \tag{3.15}$$

When the equation is rewritten after the inclusion of kernels and in terms of Lagrange multipliers, it leads to the same problem as Equation 3.10 subject to slightly different constraints:

$$0 \le \alpha_i \le C, i = 1, ..., l \tag{3.16}$$

and

$$\sum_{i=1}^{l} y_i \alpha_i = 0 \qquad\qquad (3.5)$$

The difference is found in the upper bound C on the Lagrange multipliers α_i. This limits the influence of individual data points which could be outliers. As above, the hyperplane takes the form of Equations 3.12 and 3.13. The threshold value can be computed by taking advantage of the fact that for all support vectors $\mathbf{x}_i$ with $\alpha_i \leq C$, the slack variable ξ_i is zero. This follows the KKT conditions.

CHAPTER 4

IMPLEMENTATION OF SVM

4.1 General Formula

As discussed in the previous chapter, the training of support vector machine leads to the following quadratic programming problem:

$$\text{minimize } W(\boldsymbol{\alpha}) = -\sum_{i=1}^{l} \boldsymbol{\alpha}_i + \frac{1}{2}\sum_{i,j=1}^{l} \boldsymbol{\alpha}_i \boldsymbol{\alpha}_j y_i y_j K(\mathbf{x}_i \cdot \mathbf{x}_j) \tag{4.1}$$

subject to

$$\sum_{i=1}^{l} \boldsymbol{\alpha}_i y_i = 0 \tag{4.2}$$

and

$$0 \le \boldsymbol{\alpha}_i \le C, \, i = 1, \, ..., \, l \tag{4.3}$$

In the above equations, $\boldsymbol{\alpha}$ is a vector of l variables. Each component of $\boldsymbol{\alpha}_i$ corresponds to a training data point($\mathbf{x}_i$, y_i). The solution of Equation 4.1 is a vector $\boldsymbol{\alpha}^*$ for which Equation 4.1 is minimized and constraints 4.2 and 4.3 are fulfilled. Equation 4.1 can be rewritten as

$$\text{minimize } W(\boldsymbol{\alpha}) = -\boldsymbol{\alpha}^T 1 + \frac{1}{2}\boldsymbol{\alpha}^T \mathbf{Q} \boldsymbol{\alpha} \tag{4.4}$$

subject to

$$\boldsymbol{\alpha}^T y = 0 \tag{4.5}$$

and

$$0 \le \boldsymbol{\alpha}_i \le C \tag{4.6}$$

where Q is a l by l positive semidefinite matrix defined as $Q_{ij} = y_i y_j K(\mathbf{x}_i, \mathbf{x}_j)$.

4.2 Optimization Techniques Used

The number of the training data points, l, determines the size of the optimization problem (Equation 4.4). As the size of l increases, the size of the Q matrix also increases since it is defined as l^2. If the number of data points for training for a given classification task is 10,000, then it may be impossible to keep the Q matrix in memory and the training would be computationally time intensive. In order to tackle this problem and make the implementation more efficient, several different algorithms are implemented in the program. The descriptions of the algorithms and how they improve the efficiency are provided below.

4.2.1 Decomposition Method

In order to deal with training SVMs with regards to problems with many training examples, this program uses a decomposition method proposed by Osuna (Osuna et. al, 1997). The purpose of the method is to decompose the quadratic problem into a series of smaller tasks. In each iteration, the variables α_is of Equation 4.4 are divided into two parts: an active set and an inactive set. The active set, B, consists of free variables. Free variables are those which can be updated in the current iteration. The active set is also referred to as the working set. The inactive set, N, consists of fixed variables which are temporarily fixed at a particular time. The working set has a constant size q which could be equal to l or smaller than l.

Algorithm

1. Given a working set B of size q, find α^1 as the initial solution. Set k =1.

2. If α^k is an optimal solution of Equation 4.4 then stop. Otherwise find a working set B of size q and inactive set N of size l-q. Define α_B^k and α_N^k to be subvectors of α^k corresponding to B and N, respectively.

3. Solve the following sub-problem

$$\text{minimize } W(\alpha) = -\alpha_B^T\left(1-Q_{BN}\alpha_N\right)+\frac{1}{2}\alpha_B^T Q_{BB}\alpha_B +\frac{1}{2}\alpha_N^T Q_{NN}\alpha_N - \alpha_N^T 1 \tag{4.7}$$

subject to

$$\alpha_B^T y_B + \alpha_N^T y_N \tag{4.8}$$

and

$$0 \le \alpha_m \le C, m=1,\ldots,q \tag{4.9}$$

where

$$\alpha = \begin{bmatrix} \alpha_B \\ \alpha_N \end{bmatrix} \qquad y = \begin{bmatrix} y_B \\ y_N \end{bmatrix} \qquad Q = \begin{bmatrix} Q_{BB} & Q_{BN} \\ Q_{NB} & Q_{NN} \end{bmatrix} \tag{4.10}$$

4. Set α_B^{k+1} to be the optimal solution of 4.7 and $\alpha_N^{k+1} \equiv \alpha_N^k$.

5. Set $k \leftarrow k + 1$ and go to step 2.

The variables in set N are fixed and because of that the terms $\frac{1}{2}\alpha_N^T Q_{NN}\alpha_N - \alpha_N^T 1$ are constant. They can be omitted from Equation 4.7 without having any effect on the solution. Therefore, the problem can be written as

$$\text{minimize } W(\alpha) = -\alpha_B^T\left(1-Q_{BN}\alpha_N\right)+\frac{1}{2}\alpha_B^T Q_{BB}\alpha_B \tag{4.11}$$

The active set B is updated in each iteration. Equation 4.11 is a positive semidefinite quadratic programming problem and obtaining a fast solution to the problem depends heavily on selecting good working sets (Joachims, 1998).

4.2.2 Verification of Optimal Solution

In the algorithm for the decomposition method, the second step verifies if α is an optimal solution of Equation 4.4. If α is not the optimal solution then the subproblem (Equation 4.7) is solved. In order to determine if the current α is the optimal solution, KKT conditions are checked. The Lagrange multiplier, α^T, in the equality constraint (Equation 4.5) can be denoted with λ^e and the Lagrange multipliers for the lower and upper bounds in Equation 4.6 can be denoted with λ^l and λ^u respectively. The current α is the optimal solution for Equation 4.4 if there exists λ^e, λ^l and λ^u such that

$$g(\alpha) + \left(\lambda^e y - \lambda^l + \lambda^u \right) = 0 \tag{4.12}$$

$$\lambda_m^l \left(-\alpha_m \right) = 0 \tag{4.13}$$

$$\lambda_m^u \left(\alpha_i - C \right) = 0 \tag{4.14}$$

$$\lambda^l \geq 0 \tag{4.15}$$

$$\lambda^u \geq 0 \tag{4.16}$$

where $m = 1, \ldots, l$.

The $g(\alpha)$ is a vector of partial derivatives at α and it is defined as

$$g(\alpha) = Q\alpha - 1 \tag{4.17}$$

If the optimality conditions do not hold as described above, the decomposition method works on the subproblem (Equation 4.11).

4.2.3 Selection of the Working Set

When the working set is selected, it is imperative to select a set of variables so that the current iteration will make progress towards finding the optimal solution by minimizing $W(\alpha)$ (Joachims, 1998). It is an important issue of the decomposition method. The primal and dual forms of the KKT condition (Equation 4.12) are the same. Equation 4.12 can be written as

$$
\begin{aligned}
\left(Q\alpha - 1 + by\right)_m &\geq 0 && \text{if } \alpha_m = 0 \\
\left(Q\alpha - 1 + by\right)_m &= 0 && \text{if } 0 < \alpha_m < C \\
\left(Q\alpha - 1 + by\right)_m &\leq 0 && \text{if } \alpha_m = C
\end{aligned}
\tag{4.18}
$$

where b is the same as the decision function. It also coincides with Equation 4.5 which is the Lagrange multiplier of the linear constraint. Using the assumption that $C > 0$ and $y_m = \{-1, 1\}$, Equation 4.18 further implies that

$$
\begin{aligned}
y_m &= 1, \alpha_m < C \Rightarrow \left(Q\alpha - 1\right)_m + b \geq 0 \Rightarrow b \geq -\left(Q\alpha - 1\right)_m = -\nabla f\left(\alpha\right)_m, \\
y_m &= 1, \alpha_m > 0 \Rightarrow \left(Q\alpha - 1\right)_m + b \leq 0 \Rightarrow b \leq -\left(Q\alpha - 1\right)_m = -\nabla f\left(\alpha\right)_m, \\
y_m &= -1, \alpha_m > 0 \Rightarrow \left(Q\alpha - 1\right)_m - b \leq 0 \Rightarrow b \geq \left(Q\alpha - 1\right)_m = \nabla f\left(\alpha\right)_m, \\
y_m &= -1, \alpha_m < C \Rightarrow \left(Q\alpha - 1\right)_m - b \geq 0 \Rightarrow b \leq \left(Q\alpha - 1\right)_m = \nabla f\left(\alpha\right)_m
\end{aligned}
\tag{4.19}
$$

where $f\left(\alpha\right) \equiv -\alpha^T 1 + \dfrac{1}{2}\alpha^T Q\alpha$ and $\nabla f\left(\alpha\right)$ is the gradient of $f\left(\alpha\right)$ at α.

Two elements i and j that violate the KKT conditions the most are used to select the working set $B = \{i, j\}$ where i and j are defined in the following way

$$
i = \operatorname{argmax}\left(\left\{-\nabla f\left(\alpha\right)_m \mid y_m = 1, \alpha_m < C\right\}, \left\{\nabla f\left(\alpha\right)_m \mid y_m = -1, \alpha_m > 0\right\}\right)
\tag{4.20}
$$

$$
j = \operatorname{argmin}\left(\left\{\nabla f\left(\alpha\right)_m \mid y_m = -1, \alpha_m < C\right\}, \left\{-\nabla f\left(\alpha\right)_m \mid y_m = 1, \alpha_m > 0\right\}\right)
\tag{4.21}
$$

In the decomposition method, the algorithm checks to see if α is the current solution. If it is, the following problem is solved:

$$\text{maximize } V(\mathbf{d}) = -\nabla f(\boldsymbol{\alpha})^T \mathbf{d} \tag{4.22}$$

subject to

$$y^T \mathbf{d} = 0 \tag{4.23}$$

$$d_m \geq 0 \text{ if } \boldsymbol{\alpha}_m = 0 \tag{4.24}$$

$$d_m \leq 0 \text{ if } \boldsymbol{\alpha}_m = C \tag{4.25}$$

$$-1 \leq \mathbf{d} \leq 1 \tag{4.26}$$

$$\left| \{ d_m : d_m \neq 0 \} \right| = q \tag{4.27}$$

In the above equations, $\mathbf{d}$ represents the steepest feasible descent. The idea is to find $\mathbf{d}$ with q non-zero elements. The Equation 4.22 tries to find $\mathbf{d}$. Equations 4.23, 4.24 and 4.25 ensure that $\mathbf{d}$ is projected along the equality constraint (Equation 4.5) and follows the active bound constraints. Equation 4.26 normalizes the descent vector to make the optimization problem well-posed. Equation 4.27 states that the direction of descent will only involve q variables. The components of $\boldsymbol{\alpha}$ with nonzero $\mathbf{d}_i$ are included in the working set B. This ensures that the working set selected contains the steepest feasible direction of descent. It is evident from the above information that if $q = 2$ then the solution to Equation 4.22 is

$$i = \operatorname{argmin} \left\{ \nabla f(\boldsymbol{\alpha})_m d_m \mid y_m d_m = 1; d_m \geq 0, \text{if } \alpha_m = 0; d_m \leq 0, \text{if } \alpha_m = C \right\} \tag{4.28}$$

$$j = \operatorname{argmin} \left\{ \nabla f(\boldsymbol{\alpha})_m d_m \mid y_m d_m = -1; d_m \geq 0, \text{if } \alpha_m = 0; d_m \leq 0, \text{if } \alpha_m = C \right\} \tag{4.29}$$

Equations 4.28 and 4.29 are the same as Equations 4.20 and 4.21. To find the optimal solution one more condition is checked. When the following condition holds true

$$g_i \leq -g_j \tag{4.30}$$

then the current α is the optimal solution and g_i and g_j are defined as

$$g_i \equiv \begin{cases} -\nabla f(\alpha)_i & \text{if } y_i = 1, \alpha_i < C, \\ \nabla f(\alpha)_i & \text{if } y_i = -1, \alpha_i > 0, \end{cases} \qquad (4.31)$$

$$g_j \equiv \begin{cases} -\nabla f(\alpha)_j & \text{if } y_j = -1, \alpha_j < C, \\ \nabla f(\alpha)_j & \text{if } y_j = 1, \alpha_j > 0 \end{cases} \qquad (4.32)$$

The stopping criterion is written and implemented as

$$g_i \le g_j + \varepsilon \qquad (4.33)$$

where ε is a small positive number.

4.2.4 Shrinking

For many classification problems the number of support vectors is significantly smaller than the number of training data points. If it were known beforehand which of the training data points could be used as SVs, training on those data points would result in the same solution. It would also make the decomposition method more efficient and faster, since the size of data points is reduced. In addition, with regards to problems with noisy training data points, there are often many SVs with α_i at the upper bound C called "bounded support vectors" (BSVs). The non SVs and BSVs can be treated in the same way- if the BSVs were known beforehand, the corresponding α_i could then be fixed at C leading to a new optimization problem with fewer training data points (Joachims, 1998).

During the decomposition method, it often becomes clear that certain training data points are likely to be BSVs or non SVs. These data points have α_i equal to C for several iterations (Chang, 2004). If these data points are eliminated, the size of the problem becomes l' and the decomposition method could be used to work on a smaller problem.

The solution of the smaller problem could be used to construct the solution of Equation 4.7. The subproblem that the decomposition method works on is defined below.

Let R denote the indices corresponding to the SVs, S denote indices of BSVs and T the indices of non SVs. The subproblem is similar to Equation 4.7:

$$\text{minimize } W(\alpha_R) = -\alpha_R^T\left(1-(\mathbf{Q}_{RS}1)\cdot C\right) + \frac{1}{2}\alpha_R^T\mathbf{Q}_{RR}\alpha_R + \frac{1}{2}C1^T\mathbf{Q}_{SS}C1 - |Y|C \tag{4.34}$$

subject to

$$\alpha_R^T y_R + C1^T y_S = 0 \tag{4.35}$$

and

$$0 \le \alpha_R \le C1 \tag{4.36}$$

where

$$\alpha = \begin{bmatrix} \alpha_R \\ \alpha_S \\ \alpha_T \end{bmatrix} \qquad y = \begin{bmatrix} y_R \\ y_S \\ y_T \end{bmatrix} \qquad Q = \begin{bmatrix} Q_{RR} & Q_{RS} & Q_{RT} \\ Q_{SR} & Q_{SS} & Q_{ST} \\ Q_{TR} & Q_{TS} & Q_{TT} \end{bmatrix} \tag{4.37}$$

In the above Equation (4.34), $\frac{1}{2}C1^T\mathbf{Q}_{SS}C1 - |Y|C$ is constant and it can be dropped without changing the solution. If the solution of the equation does not correspond with Equation 4.4, the whole problem has to be optimized again starting from a point α where α_B is an optimal solution of Equation 4.12 and α_N are BSVs and non SVs. The algorithm for shrinking of training data points is provided below (Chang, 2004).

Algorithm

1. After every min (l, 1000) iterations, some variables are shrinked. During the iteration process

$$\min\left(\left\{\nabla f\left(\alpha^k\right)_m \mid y_m = -1, \alpha_m < C\right\}, \left\{-\nabla f\left(\alpha^k\right)_m \mid y_m = 1, \alpha_m > 0\right\}\right) = -g_j$$
$$< g_i = \max\left(\left\{-\nabla f\left(\alpha^k\right)_m \mid y_m = 1, \alpha_m < C\right\}, \left\{\nabla f\left(\alpha^k\right)_m \mid y_m = -1, \alpha_m > 0\right\}\right)$$

(4.38)

is not satisfied yet.

There are two sets of values that are inactivated in order to perform the shrinking by reducing the set R of activated variables. For those

$$g_m \equiv \begin{cases} -\nabla f(\alpha)_m & \text{if } y_m = 1, \alpha_m < C, \\ \nabla f(\alpha)_m & \text{if } y_m = -1, \alpha_m > 0, \end{cases}$$

(4.39)

if

$$g_m \leq -g_j$$

(4.40)

and α_m resides at a bound then α_m can be deactivated since the value may not change anymore.

Similarly, for those

$$g_m \equiv \begin{cases} -\nabla f(\alpha)_m & \text{if } y_m = -1, \alpha_m < C, \\ \nabla f(\alpha)_m & \text{if } y_m = 1, \alpha_m > 0, \end{cases}$$

(4.41)

if

$$-g_m \geq g_i$$

(4.42)

and α_m resides at a bound then α_m is deactivated.

2. The decomposition method achieves the following tolerance condition

$$g_i \leq -g_j + 10\varepsilon$$

(4.43)

When this condition is fulfilled, the whole gradient is reconstructed. Based on the reconstructed gradient, Equations 4.40 and 4.42 are used to deactivate the variables for the decomposition method.

This is done because the procedures in Step #1 might be too aggressive and it is possible that the decomposition method works on an incorrectly shrinked problem (Equation 4.34).

4.2.5 Reconstruction of the Gradient

Since the gradient is reconstructed in Step #2 of the shrinking algorithm, the cost of performing reconstruction is reduced by keeping the following during the iterations

$$\overline{G}_i = C\sum_{\alpha_j} Q_{ij} , i = 1,\ldots,l \tag{4.44}$$

The gradient is constructed as

$$\nabla f\left(\boldsymbol{\alpha}\right)_i = \sum_{j=1}^{l} Q_{ij}\alpha_j = \overline{G}_i + \sum_{0<\alpha_j<C} Q_{ij}\alpha_j \tag{4.45}$$

4.2.6 Calculation of b and ρ

The value of ρ is calculated after the optimal solution $\boldsymbol{\alpha}$ of Equation 4.4 is found. It is used in the decision function. In order to calculate ρ the following formula is used

$$\rho = \frac{s_1 + s_2}{2} \tag{4.46}$$

where s_1 and s_2 correspond to $y_i = 1$ and $y_i = -1$, respectively. When considering the $y_i = 1$ class, if there are α_i which fulfill $0 < \alpha_i < C$, then $s_1 = \nabla f\left(\boldsymbol{\alpha}\right)_i$. In order to avoid numerical errors s_1 is calculated in the following way:

$$s_1 = \frac{\sum_{0<\alpha_i<C,y_i=1} \nabla f\left(\boldsymbol{\alpha}\right)_i}{\sum_{0<\alpha_i<C,y_i=1} 1} \tag{4.47}$$

If, on the other hand, there is no α_i which fulfills $0 < \alpha_i < C$, s_1 is taken as the midpoint of the range since s_1 must satisfy the following:

$$\max_{\alpha_i=C,y_1=1} \nabla f(\alpha)_i \le s_1 \le \min_{\alpha_i=0,y_1=1} \nabla f(\alpha)_i \tag{4.48}$$

The s_2 value for $y_i = -1$ is calculated the same way.

Calculation of *b*

The values of s_1 and s_2 are used to calculate the *b* term also. The following formula is used

$$-b = \frac{s_1 - s_2}{2} \tag{4.49}$$

4.2.7 Caching

In this implementation of SVM, the technique of caching is used to reduce computational time. The caching technique used is a simple least-recent-use strategy. Since the size of Q matrix is l^2 and it is fully dense, it becomes impossible to keep in memory if the training sample size is large. This problem is solved by calculating and storing elements of Q_{RR} of Equation 4.34 as needed. This reduces the computational cost of later iterations as well.

4.2.8 Implementation of Solution to Decomposition Method Sub-problem

The sub-problem described in section 4.2.1 can be considered to be a simple problem consisting of only two variables

$$\min_{\alpha_i,\alpha_j} \frac{1}{2} \begin{bmatrix} \alpha_i & \alpha_j \end{bmatrix} \begin{bmatrix} Q_{ii} & Q_{ij} \\ Q_{ji} & Q_{jj} \end{bmatrix} \begin{bmatrix} \alpha_i \\ \alpha_j \end{bmatrix} + \left(Q_{i,N}\alpha_N - 1\right)\alpha_i + \left(Q_{j,N}\alpha_N - 1\right)\alpha_j \tag{4.50}$$

subject to

$$y_i\alpha_i + y_j\alpha_j = \Delta' \equiv -\mathbf{y}_N^T\boldsymbol{\alpha}_N^k \tag{4.51}$$

and

$$0 \le \alpha_i, \alpha_j \le C \tag{4.52}$$

If $\alpha_i = y_i\left(-\mathbf{y}_N^T \mathbf{\alpha}_N^k - y_j \alpha_j\right)$ is substituted into the objective function of Equation 4.11, an

unconstrained minimization on α_j is solved (Platt, 1998) and the solution obtained is

$$\alpha_j = \begin{cases} \alpha_j + \frac{-G_i - G_j}{Q_{ii} + Q_{jj} + 2Q_{ij}} & \text{if } y_i \ne y_j, \\ \alpha_j + \frac{G_i - G_j}{Q_{ii} + Q_{jj} - 2Q_{ij}} & \text{if } y_i = y_j, \end{cases} \tag{4.53}$$

where

$$G_i \equiv \nabla f(\alpha)_i \quad \text{and} \quad G_j \equiv \nabla f(\alpha)_j \tag{4.54}$$

If the value of α_j^{new} is outside the feasible region of Equation 4.11 then it is

clipped into the feasible region and it is assigned as the new α_j. For example, if $y_i = y_j$ and

$C \le \alpha_i + \alpha_j \le 2C$ then α_j^{new} must satisfy the following

$$L \equiv \alpha_i + \alpha_j - C \le \alpha_j^{new} \le C \equiv H \tag{4.55}$$

The largest value of α_i^{new} and α_j^{new} can be C. Therefore, if

$$\alpha_j + \frac{G_i - G_j}{Q_{ii} + Q_{jj} - 2Q_{ij}} \le L \tag{4.56}$$

then

$$\alpha_j^{new} \equiv L \tag{4.57}$$

and

$$\alpha_j^{new} = \alpha_i + \alpha_j - \alpha_j^{new} = C \tag{4.58}$$

In *SlimSVM*, Equation 4.53 is rewritten and implemented as (Chang, 2004)

$$\alpha_j = \begin{cases} \alpha_j + \frac{-G_i - G_j}{Q_{ii} + Q_{jj} + 2Q_{ij}, +0} & \text{if } y_i \ne y_j, \\ \alpha_j + \frac{G_i - G_j}{Q_{ii} + Q_{jj} - 2Q_{ij}, +0} & \text{if } y_i = y_j, \end{cases} \tag{4.59}$$

4.2.9 Unbalanced Data

In order to handle cases where the classification data consists of substantially different number of data in each of the classes, Osuna has proposed to use different penalty parameters in the formula for SVM. Using different penalty parameters, Equation 3.15 becomes

$$\tau(\mathbf{w}, \xi) = \frac{1}{2} \|\mathbf{w}\|^2 + C_+ \sum_{y_i=1} \xi_i + C_- \sum_{y_i=-1} \xi_i \tag{4.60}$$

and its dual becomes

$$\min_{\alpha} W(\alpha) = -\alpha^T 1 + \frac{1}{2} \alpha^T Q \alpha \tag{4.61}$$

subject to

$$\alpha^T \mathbf{y} = 0 \tag{4.62}$$

and

$$\begin{aligned} 0 \le \alpha_i \le C_+, \quad & \text{if } y_i = 1 \\ 0 \le \alpha_i \le C_-, \quad & \text{if } y_i = -1 \end{aligned} \tag{4.63}$$

For unbalanced data, the implementation to obtain the solution is the same, except for Equation 4.50 where only the constraints change. The modified constraints are as follows

$$\begin{aligned} 0 \le \alpha_i \le C_i, \quad & \text{if } y_i = 1 \\ 0 \le \alpha_i \le C_j, \quad & \text{if } y_i = -1 \end{aligned} \tag{4.64}$$

C_i and C_j can be treated as C_+ and C_- depending on y_i and y_j.

4.2.10 Multi-class Classification

In order to perform multi-class classification, the one-against-one approach (Knerr et al., 1990) is used. To train the data, $k(k-1)/2$ classifiers are constructed, where k is the number

of classes. Each classifier trains data from different classes. For training data from the ith and jth classes, the following binary classification problem is solved

$$\tau\left(\mathbf{w}^{ij}, \xi^{ij}\right) = \frac{1}{2}\left\|\mathbf{w}^{ij}\right\|^2 + C\left(\sum_t \left(\xi^{ij}\right)_t\right) \tag{4.65}$$

subject to

$$\left(\left(\left(\mathbf{w}^{ij}\right)^T \Phi\left(\mathbf{x}_t\right)\right) + b^{ij}\right) \geq 1 - \xi_t^{ij}, \quad \text{if } \mathbf{x}_t \text{ is in the } i\text{th class}$$

$$\left(\left(\left(\mathbf{w}^{ij}\right)^T \Phi\left(\mathbf{x}_t\right)\right) + b^{ij}\right) \leq -1 + \xi_t^{ij}, \quad \text{if } \mathbf{x}_t \text{ is in the } j\text{th class} \tag{4.66}$$

and

$$\xi_t^{ij} \geq 0 \tag{4.67}$$

where $t = 1, \ldots l$.

CHAPTER 5

DEVELOPMENT OF *SLIMSVM*

5.1 General Description of the Software

SlimSVM is a menu driven software consisting of ten files. The files are compiled with gcc on *SunOS 5.9*. A *makefile* containing all the commands needed for compilation is also provided. Both the training and classification is done by *SlimSVM.cpp*. A file containing the training data is specified by the user and *SlimSVM.cpp* writes the model in another file provided by the user. The model file is used for classification and the results are written in a file specified by the user. The other files included in the software contain functions that are used for both training and classification. The descriptions of the functions are provided in section 5.5.

5.2 Software Structure
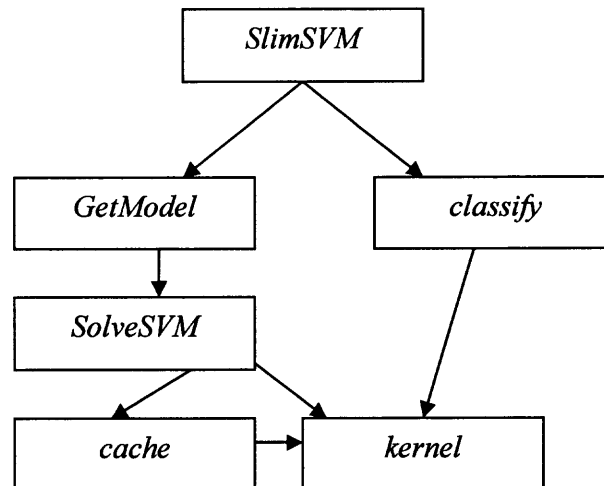
Figure 5.1 Software Structure.

5.3 Software Specifications

- Training data for the training part of the program follow a specified file format. Similarly, the input test data for the classification part of the program follow a specified file format. A description of the training file and classification input file format is provided in section 5.4.

- For both the training and classification files, the feature values of the data points must be normalized by scaling to the range [-1. +1] or [0, 1] in order to prevent feature values in greater numeric ranges from dominating those in smaller numeric ranges.

- The model file used for the classification part of the program has to be a model file written by the training module of the program. It is the responsibility of the user to make sure that the correct model file is provided. A model file that results from training in other software will not work with *SlimSVM*.

- The result of the training will be written in a user specified file at the end of training. The file format description is provided in section 5.4.

- The results of classification will be written in a user specified file at the end of classification. The file format description is provided in section 5.4.

- The degree parameter of polynomial kernel for training must be greater than 0; otherwise, an error message is shown.

- The stopping criterion, epsilon, must be greater than 0; otherwise, an error message is shown.

- The C parameter, which determines the tradeoff between the errors and the separating margin, must be greater than 0; otherwise, an error message is shown.

- For the weight parameter, both class labels and their weight must be entered; otherwise, the default value of 1 is used.

- The training part of the software will display on screen the number of iterations performed, the value for the objective and rho, and the number of support vectors and bound support vectors obtained by training.

- The classification module of the software will display on screen the total number of data points to be classified, total number of correct classifications, accuracy of classification, and the mean squared error.

5.4 File Format

5.4.1 Training Input File

The data in the input file is in sparse format as follows:

Class Label Feature ID: Feature Value Feature ID: Feature Value

The class label must be provided for the training file and the class labels must be correct.

Note that some other implementations of SVM use dense format where only the class label and feature values are provided for the data points. A sample from the training file is shown in Figure A1.

5.4.2 Training Model File

For the dot kernel, the result of the training obtained from *SlimSVM* is written in the following format:

Line1: Type of Kernel Used for training

Line 2: Total Number of Classes

Line 3: Total Number of Support Vectors

Line 4: Value of rho

Line 5: The labels of each class separated by a space

Line 6: Number of support vectors in each class corresponding to the class labels separated by a space

Line 7 to EOF: the support vectors are listed in the same file format as the input file and the test file. The support vectors are grouped and listed together according to their class labels.

A sample from the training model file using dot kernel is shown in Figure A2. For the polynomial kernel, the result of the training obtained from *SlimSVM* is written in the following format:

Line1: Type of Kernel Used for training

Line 2: Polynomial Degree

Line 3: Value for gamma

Line 4: Value for coefficient

Line 5: Total number of classes in the model

Line 6: Total number of support vectors in the model

Line 7: Value for rho

Line 8: The labels of each class separated by a space

Line 9: Number of support vectors in each class corresponding to the class labels separated by a space

Line 10 to EOF: the support vectors are listed in the same file format as the input file and the test file. The support vectors are grouped and listed together according to their class labels.

A sample from the training model file using polynomial kernel is shown in Figure A3.

5.4.3 Classification Test File

The format of the test file is same as that for the input file for training. Class labels must be provided but they do not have to be correct, since the classification option of *SlimSVM* program will actually determine the labels based on the model obtained from the training of *SlimSVM*.

5.4.4 Classification Result File

The classification file contains the labels predicted by classification part of *SlimSVM*. For example, with regards to binary classification, if the data belongs to the first class it is labeled 1 and the data in the other class is labeled 2. A sample from the classification file is shown in Figure A4.

5.5 Software File Description

As mentioned above, *SlimSVM* consists of ten files containing functions for both the training and classification. In this section, a description of the C++ source files is provided along with descriptions of the functions and the algorithm used in each file. Note that only the description of cpp files is provided and not the header files, since the header files contain variable and function declaration.

5.5.1 Description of *SlimSVM.cpp*

This file builds a SVM model based on the training file and user input of parameters. The user can select one of two types of training options: training with default value of parameters or training with custom values. The user can select the parameters to change. If a wrong value is entered for the parameters, an error message is displayed. The user can also select to view a brief description and the default values of the parameters. The program stores the model in a file provided by the user. It also classifies data based on the model file obtained from training and stores the result of classification in a file provided by the user.

5.5.1.1 Description of the Functions of *SlimSVM.cpp*

- **void defaultTrain()** – This function asks the user to enter the training data file name and the model file name to write the output of training. It also contains the default values for each of the parameters used for training. After it reads in the training data file it stores the data points in the struct training_data data_points. The struct is defined in *SlimSVM.h*. Then the function build_model stores the training model in the struct training_model model. The function store_model writes the model to the model file specified by the user. The memory used by the model is freed by the free_model function.

- **void customTrain()** – This function allows the user to view a brief description and the default values for the parameters to be used for training. It also allows the user to enter custom values for the parameters to be used for training. If the user chooses to view the parameter description then the defaultValues() function is called. If the user chooses to change the parameters then the customValues() function is called.

- **void defaultValues()** – This function displays a brief description and the default values of the parameters for training. After displaying them, the customTrain() function is called so the user could select and enter values for parameters.

- **void customValues()** – This function asks the user to enter the training data file name and the model file name to write the output of training. It displays the list of parameters that can be changed. The function allows the user to select the parameters to be modified and reads in the values entered by the user. If incorrect values are entered then an error message is displayed on screen. After the values

for the parameters are read, the function build_model stores the training model in the struct training_model model. The function store_model writes the model to the model file. The memory used by the model is freed by the free_model function.

- **void read_problem(const char *)** – This function reads in the data in the training data file specified by the user. If the file is not found or if it cannot be opened, an error message is displyed. After opening the file the function scans through it to calculate the total number of data points and total number of features for each of the data points. Then it allocates memory for arrays to hold the labels of each data point along with the features. The function scans through the file again to read in the labels and the features of each data point.

- **void classify()** – This function does the classification of data based on the training model obtained from the training module of *SlimSVM*. It asks the user to enter the names of the file containing data to be classified, the model file, and the file where classification results are to be written. If any of these files cannot be opened an error message is displayed. The function opens the input file for classification and gets the label of each data point and the features one data point at a time, then calls the get_class(const struct training_model *, const struct feature *) function to get the label after classification. The newly classified label is stored in the result_class variable and written to the output file. This function also outputs classification accuracy statistics to the screen. It calls the free_model function to free the memory used by the training model.

5.5.1.2 Flow Chart of *SlimSVM.cpp*
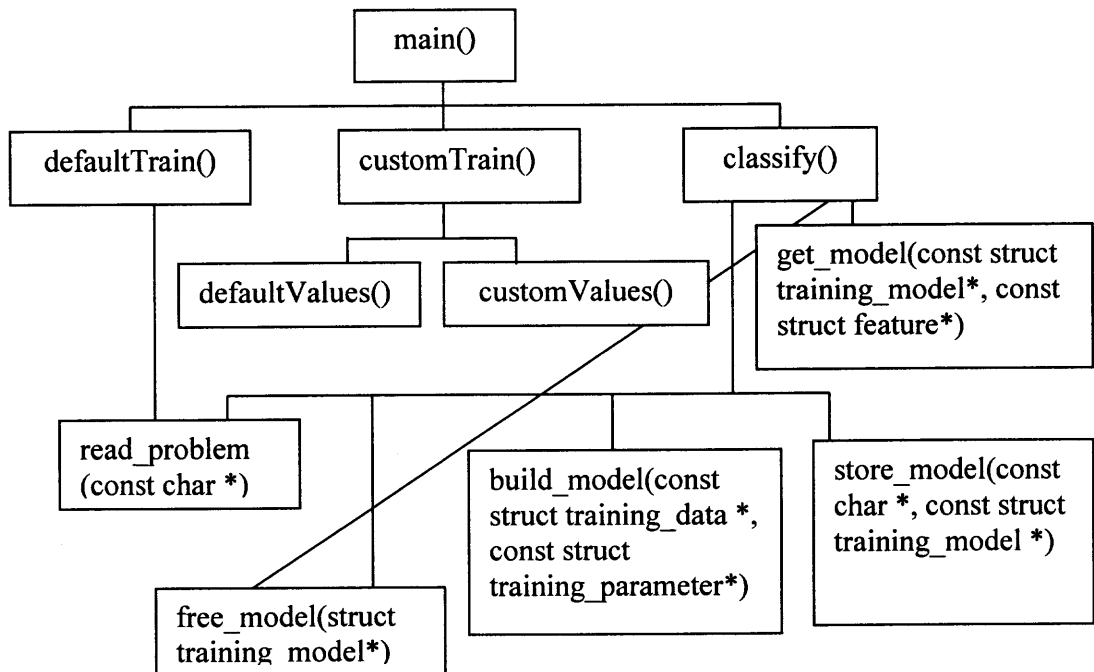
Figure 5.2 Function Structure of *SlimSVM.cpp*.

5.5.2 Description of *GetModel.cpp*

This file contains functions that are used for building a training model. The functions extract the data from the struct data_points and group them by the labeled classes provided in the training file. After training, the model is stored in a file specified by the user.

5.5.2.1 Description of the functions of *GetModel.cpp*

- **training_model *build_model(const training_data * data_points, const training_parameter *param)** – This function prepares the data read from the training file, calls the function train_SlimSVM for training, and stores the model from training to be written to the model file. First, it allocates memory and stores

the parameter values for training. In preparation for training, the function first counts the number of classes in the training file and the number of data points in each class. It then allocates an array to store the data points. The data points are grouped by classes and stored in the array. Next, the function calculates the weighted cost of each class as entered by the user. It allocates two arrays of a size equal to the number of classes. In the weight_label array, the class labels are stored. In the array weight_array, the product of the weight for each class and the cost parameter is stored. After calculating the weighted cost of each of the classes, the function calls the train_SlimSVM function, which takes the data structure that holds the data points read from the training file, the parameters from training, and the weighted costs for the classes as its function parameters. The result of the train_SlimSVM function is used to change values in the array bool_array which is initialized to false. If the result of train_SlimSVM is not zero (α value is not zero) for a data point, the bool_array value is changed to true for the data point and it is considered to be a support vector. Then the function stores the labels of classes and the value for rho from the training in struct model. The number of indices with a true value in bool_array is counted and stored as the number of support vectors of the training model. The true value of bool_array is also used to store the data points and their features in struct model. The support vectors are grouped and stored by the same class in model.

- **decision_function train_SlimSVM(const training_data *data_points, const training_parameter *param, double Ca, double Cb)** – This function allocates an array called alpha and declares struct dec_soln of type SlimSVM_solution

which is defined in *SolveSVM.h*. The function calls train_model function with the function parameters data_points, param entered by the user, dec_soln and the weighted costs Ca and Cb. It outputs the value for rho and objective to the screen. It also counts the number of support vectors and bound support vectors and outputs to the screen. The function then stores the value for the decision function obtained by train_model function and returns the decision function to the calling function.

- **static void train_model(const training_data *prob, const training_parameter* param, double *alpha, SolveSVM::SlimSVM_solution* si, double Ca, double Cb)** – This function allocates memory for holding the class labels of the data points and an array called neg_ones. The alpha array declared in train_SlimSVM function is initialized to zero. The neg_ones array is initialized with -1 and the array for class labels is initialized with the class labels. It declares an object of the SolveSVM class called SlimSVM and invokes the solve function defined in *SolveSVM.cpp*.

- **int store_model(const char *model_file_name, const training_model *model)** – This function stores the model from training in a file specified by the user. It prints the kernel type, the number of classes, total number of support vectors, the value for rho, the number of support vectors in each class, and the support vectors along with their labels and the features. If the kernel type used for training is polynomial, then it prints the degree, gamma, and coefficient of the polynomial kernel.

- **void free_model(training_model * model)** – This function frees the memory for all the data structures used by the model.

- **Q_matrix(const training_data& prob, const training_parameter& param, const schar *y_):Kernel(prob.l, prob.x, param)** – This function serves as the constructor for the Q_matix class. The function initializes the kernel constructor. It also allocates the cache size set by the cache value in the training parameter.

- **float *get_Q_column(int i, int len) const** – This function calculates the kernel values for a column in the Q matrix and returns the values to the calling function.

- **void swap_index(int i, int j) const** – This function swaps the values in the cache, kernel and the y array between the indices specified in the function parameter.

- **~Q_matrix()** – This is the destructor for the Q_matrix class. It deletes memory taken up by the cache and the y array.

5.5.2.2 Flow Chart of *GetModel.cpp*

```
struct training_model
*build_model(const struct
training_data *, const struct
training_parameter *)
```

```
decision_function
train_SlimSVM(const
svm_problem *, const
training_parameter *, double,
double)
```

```
static void train_model(const
svm_problem *, const
training_parameter*, double
*,SolveSVM::SlimSVM_soluti
on*, double, double)
```

```
void SolveSVM::Solve(int ,
const Kernel&, const double *,
const schar *, double *, double,
double, double,
SlimSVM_solution*)
```

Figure 5.3 Function Structure of *GetModel.cpp*.

5.5.3 Description of *SolveSVM.cpp*

This file contains functions that are used to calculate the values of the alpha array used to get the training model. The Solve function does the calculations and it is called by the train_model function in *GetModel.cpp*.

5.5.3.1 Description of the Functions of *SolveSVM.cpp*

- **void SolveSVM:: Solve(int l, const Kernel& Q, const double *b_, const schar *y_, double *alpha_, double Ca, double Cb, double eps, SlimSVM_solution* si)** – As mentioned above, this function is called by train_model. First, the Solve function copies all the parameters of the function and initializes the boolean shrink_status to false. It declares a character array called alpha_status and calls the update_alpha_status() function to initialize the array. It also allocates another array called active_set_array. The arrays for gradient calculation are then declared and initialized. To begin calculation for the alpha array, the function calls select_working_set function to get the indices i and j of the working set. If the select_working_set function does not satisfy the stopping criterion then it returns i and j. On the other hand, if it does satisfy the stopping criterion, the gradients are reconstructed using the rebuild_gradient function. Then the select_working_set function is called again to verify whether the stopping criterion has been satisfied. If it has been satisfied, then Solve() continues with the rest of the function to calculate alpha at i and j. If the stopping criterion has not been satisfied, then the shrink_problem function is called to shrink the active set.

Using the indices, the get_Q_column function is called to get the values in the columns of Q matrix, and they are stored in the arrays Q_i and Q_j. Then the weighted costs at the indices are stored using the get_C function. The value stored in indices i and j of the alpha array is copied into old_alpha_value1 and old_alpha_value2 for later calculations. Depending on whether or not the class labels are same at indices i and j, the value for alpha is calculated and stored at

those indices using different constraints. After the value for alpha array is calculated, old_alpha_value for both i and j are subtracted from alpha[i] and alpha[j] and the result is stored in delta_alpha_i and delta_alpha_j. Then using delta_alpha_i and delta_alpha_j, the gradient is updated. Next, the function is_upper_bound() is called with i and j to get boolean values, and the alpha_status array is updated with update_alpha_status() at i and j. Using the boolean values, the gradient_0 is updated.

After optimizing the alpha value, the calculate_rho() function is called to calculate the value for rho and store it in the solution_info struct called si. The objective value calculation is done using the gradient and the alpha value, with the result also stored in si. Finally, the entire solution to the optimization problem is stored in the alpha_[] from the alpha array. The values for weighted costs of the classes are also stored in si. The total number of iterations performed during the optimization is displayed on screen. The arrays declared and used by this function are deleted.

- **int SolveSVM::select_working_set(int &max_i, int &max_j)** – This function verifies whether the stopping criterion was reached and it also returns the indices for the working set for the optimization problem. If the stopping criterion is satisfied, the function returns 1. Otherwise, it returns the indices i and j to the calling function through the function parameters and returns 0 to the calling function. Depending on the label of the training data, different constraints are used to find the maximum gradient values to determine whether the stopping

criterion has been satisfied. The function then returns the indices of the maximum gradient values to be used for working set.

- **void SolveSVM::shrink_problem()** – This function shrinks the active set for the optimization of the alpha values. First, it checks to see if the stopping criterion has been satisfied by calling the select_working_set() function. If it has been, the rest of the code in the function is not executed. The select_working_set function returns the indices i and j for the working set. Using the i and j values, the values in the gradient array and the y array containing the class label are multiplied and stored in variables grad_i and grad_j. For indices starting from 0 to the size of active set, if the alpha_status array contains a 0 or 1, different conditions are checked depending on the class label contained in the y array. After all the conditions are met, the value in active_set_size is decremented. Then the swap_index function is called with the index that satisfied all the conditions and the value of active_set_size. If the shrink_status variable is true or if -(grad_j + grad_i) > epsilon*10 then the function is returned. The shrink_status variable is declared and initialized to false in *SolveSVM.h*. Finally, for indices decrementing from *l* to active_set_size, if the alpha_status array contains 0 or 1, different conditions are checked depending on the class label contained in the y array. After all the conditions are met, the index that satisfied all the conditions checked and the value in the active_set_size array are passed to swap_index function. The index and the active_set_size are incremented.

- **void SolveSVM::swap_index(int i, int j)** – This function swaps the value in different arrays between the indices specified by the two integers. The indices that are to be swapped are passed to the function as integers by the calling function.

- **void SolveSVM::rebuild_gradient()** – This function rebuilds the gradient arrays. If active_set_size is equal to the number of data points in the training file then it exits and returns to the calling function. Otherwise, both gradients are reconstructed. To reconstruct the gradient array for the indices from active_set_size to the number of training data points, the values in the b array, which contains -1, is added to the value in the gradient_0 array and the result is stored. Depending on whether the alpha_status array contains the value 2 at those indices, alpha_value is then multiplied by the values in the Q_i array at those indices and the result is stored in the gradient array.

- **double SolveSVM::get_C(int i)** – This function returns the weighted cost of the class by checking the array containing the labels of the classes. The position of the array checked is equal to the integer passed to the function.

- **void SolveSVM::update_alpha_status(int i)** – This function calls the get_C function and the result of get_C is compared with the contents of the alpha array. The index compared is the integer passed to the function by the calling function. Depending on whether the content of the alpha array is greater than, less than, or equal to the weighted cost of classes, the alpha_status array is stored with 0, 1 or 2.

- **bool SolveSVM::is_lower_bound(int i)** – This function takes an integer as a function parameter and returns true to the calling function if the alpha_status array at the integer index contains 0.

- **bool SolveSVM::is_upper_bound(int i)** – This function takes an integer as a function parameter and returns true to the calling function if the alpha_status array at the integer index contains 1.

- **bool SolveSVM::is_free(int i)** – This function takes an integer as a function parameter and returns true to the calling function if the alpha_status array at the integer index contains 2.

- **double SolveSVM::calculate_rho()** – This function calculates the value for rho and returns the value to the calling function. The value for rho is used to calculate the decision function. In order to calculate rho, the values in y array are first multiplied by the values in the gradient for indices from 0 to size of the active set. The result of the product is then stored in variable yG_prod. Using is_lower_bound and is_upper_bound functions, the value of the upper_bound and lower_bound variables are obtained in each iteration. The upper_bound variable stores the minimum of yG_prod and the value already stored in it. The lower_bound variable stores the maximum of yG_prod and the value already stored in it. Also, the number of indices in the alpha_status array containing the value 2 is counted and stored in num_free and the value of yG_prod is added to sum_free. If the value stored in num_free variable is greater than 0, rho is calculated by dividing sum_free by num_free. On the other hand, if the value in

num_free is less than 0, rho is calculated by dividing the sum of upper_bound and lower_bound by 2.

5.5.3.2 Flow Chart of *SolveSVM.cpp*
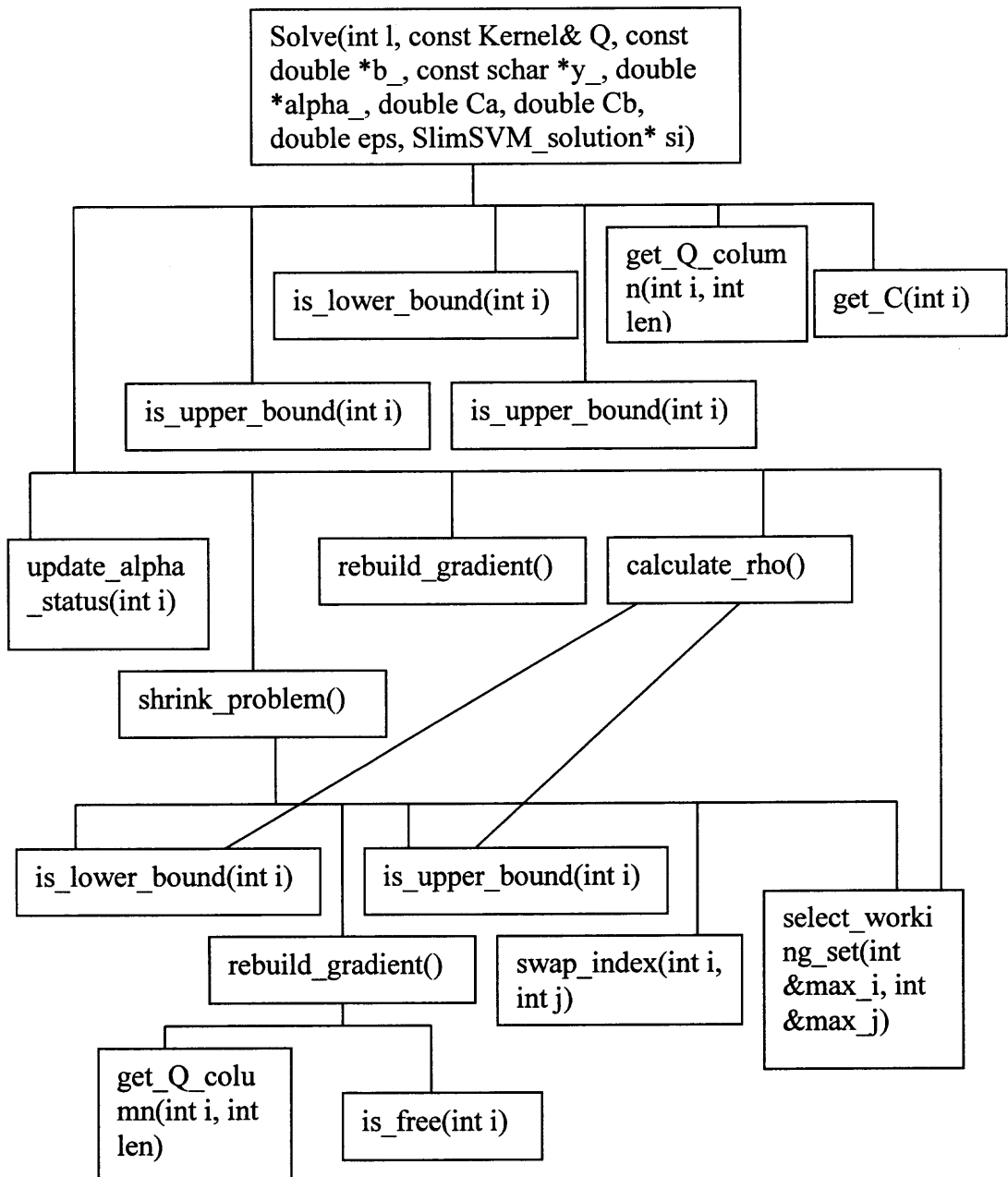
Figure 5.4 Function Structure of *SolveSVM.cpp*.

5.5.4 Description of *classify.cpp*

The file contains functions that are used to determine the classes of the input data for classification. The file also contains a function that reads in the model obtained by training module of *SlimSVM*. The model read is used to classify the data.

5.5.4.1 Description of the Functions of *classify.cpp*

- **double getclass(const training_model *model, const feature *x)** – This function is called by the classify() function in *SlimSVM.cpp*. It takes the training model as one of the parameters, takes the features of a data point for classification as the other parameter, and then reads in the number of classes in the model. The function also allocates arrays called class_count and dec_class. The class_count array is initialized to 0. The dec_class array is passed as a parameter to the get_values function along with the model and x. The value in the dec_class array is used to increment the count in the class_count array for each of the indices. A variable called max_count is initialized to 0. For all the indices in the class_count array, the index with the highest value is stored in max_count and is returned as the label.

- **void get_values(const training_model *model, const feature *x, double* dec_class)** - This function is used to determine the class of the data points based on the model file. The result of calculation for determining the classes are stored in the array called dec_class. It allocates an array called kernel_val where the result of dot or polynomial kernel calculation is stored. Then a value called sum, which is the sum of product of class label and kernel calculation, is calculated.

The value for rho is subtracted from sum and the result is stored in dec_class array. This value is used by the get_class function to predict the class as described above.

- **Training_model *read_model(const char *model_file_name)** – This function is called by the classify() function in *SlimSVM.cpp*. It takes the model file name as its function parameter. Before the function reads in the model file, it allocates memory for the model and parameter of the model. It scans through the model file and reads in the kernel type, value for degree, value for gamma, value for coefficient, total number of classes, total number of support vectors, value for rho, the labels of classes, and the number of support vectors in each class. The values for degree, gamma, and coefficient are read only if the kernel type is polynomial. If the kernel type is neither dot kernel nor polynomial then an error message is displayed. After reading in the model information, the function scans through rest of the file and reads in the class labels of the support vectors and the features of each of the support vectors in the model file. When the support vectors are read and stored the function returns the model to the calling function. Since this function is called by the classify function in *SlimSVM.cpp*, it is included in Fig. 5.2 for function structure.

5.5.4.2 Flow Chart of *classify.cpp*

```
┌─────────────────────────┐
│ double get_class(const  │
│ training_model *, const │
│ feature *)              │
└─────────────────────────┘
            │
┌─────────────────────────┐
│ void get_values(const   │
│ training_model *, const │
│ feature *, double *)    │
└─────────────────────────┘
            │
┌─────────────────────────┐
│ Kernel:: k_function(x,  │
│ model->SV[i], model-    │
│ >model_param)           │
└─────────────────────────┘
```
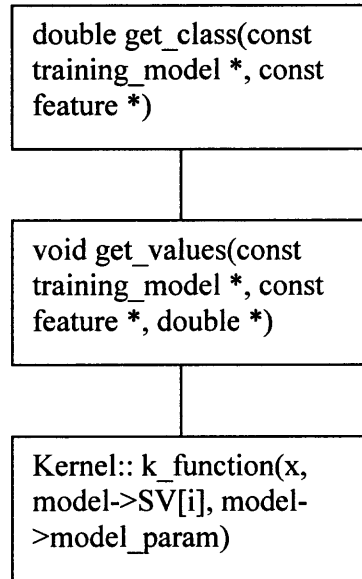
Figure 5.5 Function Structure of *classify.cpp*.

5.5.5 Description of *kernel.cpp*

This file contains functions that calculate linear kernel and polynomial kernel values and return the result to the calling functions. The header file for *kernel.cpp* contains the function headers and the variables used by the functions. In addition, it contains a template function called make_duplicate that copies data values from one array to another array.
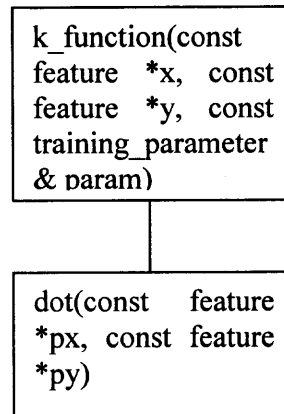
5.5.5.1 Description of the Functions of *kernel.cpp*

- **Kernel::Kernel(int l, feature * const * x_, const training_parameter& param):kernel_type(param.kernel_opt), degree(param.poly_degree), gamma(param.poly_gamma), coef(param.coef)** – This is the constructor for the kernel class. Its function parameters include the number of the data points in the

training file, the data point features, and training parameters. It initializes kernel type, degree, gamma, and coefficient using the values from the training parameter. Based on the kernel type, the constructor calls the functions for dot or polynomial kernels, which are defined as private in the header file.

- **Kernel::~Kernel()** – This is the destructor. It frees the memory for the arrays used by the constructor.

- **double Kernel::dot(const feature *px, const feature *py)** – This function calculates values for the dot kernel and returns the values to the calling function. The function first checks if the features for the dot product calculation are valid. The end of data point feature is indicated by -1 and it is not used for calculation. If the end is not reached it calculates the dot product of the values stored in each of the features and adds the result to variable sum.

- **double Kernel::k_function(const feature *x, const feature *y, const training_parameter& param)** – Depending on the kernel type specified by the training parameter, this function calculates the value for the linear kernel and polynomial kernel. For linear kernel calculation, only the dot function is called. For polynomial kernel calculation, dot function along with degree, gamma, and coefficient values from the training parameters are used. The result is returned to the calling function.

5.5.5.2 Flow Chart of *kernel.cpp*

```
┌─────────────────────┐
│ k_function(const    │
│ feature *x, const   │
│ feature *y, const   │
│ training_parameter  │
│ & param)            │
└──────────┬──────────┘
           │
┌──────────┴──────────┐
│ dot(const   feature │
│ *px, const  feature │
│ *py)                │
└─────────────────────┘
```

Figure 5.6 Function Structure of *kernel.cpp*.

5.5.6 Description of *cache.cpp*

This file contains functions for the cache. As mentioned in the previous chapter, caching is used in order to handle large data sets more efficiently.

5.5.6.1 Description of the Functions of *cache.cpp*

- **Cache::Cache(int l_, int size_):l(l_),size(size_)** – This function is the constructor. It initializes integers size and l using the values passed as function parameters, and then allocates an array called head of size l to hold each node of a circular linked list. The function also initializes the size of cache to hold all the elements in the linked list based on the training parameter value for cache size.

- **Cache::~Cache()** – This function is the destructor. It frees the memory used by the constructor.

- **void Cache::lru_delete(head_t *h)** – This function deletes the data node specified by the calling function from the current location.

- **void Cache::lru_insert(head_t *h)** – This function inserts a node of data into the circular list when a function calls it. The new node is inserted at the last position of the list.

- **int Cache::get_data(const int index, float **data, int len)** – This function allocates more memory as needed and returns the address of the head of the linked list to the calling function.

- **void Cache::swap_index(int i, int j)** – This function swaps the values stored in the indices as specified by the function parameters. In order to swap values it checks different conditions; if they are satisfied, the values are swapped.

5.5.6.2 Flow Chart of *cache.cpp*
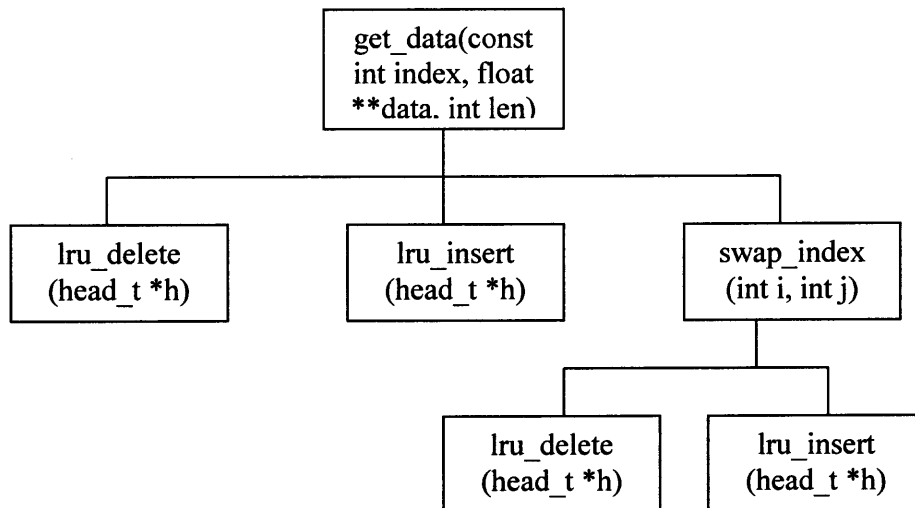
Figure 5.7 Function Structure of *cache.cpp*.

CHAPTER 6

TESTING

Two sets of testing were done on *SlimSVM*. The first set of testing was performed using artificial non-biological data sets, while the second set was performed using actual biological microarray data used in (Ma et al., 2004) provided by Dr. Honghua Li at Cancer Institute of New Jersey.

6.1 Artificial Data Set

For the testing of *SlimSVM* with artificial data sets, three sets of data files were created. The data in the files were two dimensional and generated randomly. Each set of data consists of two files-one file for training and the other for testing on the model obtained from training. A brief description of the files is provided below:

- set1_1.txt- Contains two classes of data points. Used for training on linear kernel for binary classification.

- set1_2.txt- Contains two classes of data points. Used for testing on linear kernel for binary classification.

- set2_1.txt- Contains three classes of data points. Used for training on linear kernel for multi-class classification.

- set2_2.txt- Contains three classes of data points. Used for testing on linear kernel for multi-class classification.

- set3_1.txt- Contains two classes of data points. Used for training on polynomial kernel for binary classification.

- set3_2.txt- Contains two classes of data points. Used for testing on polynomial kernel for binary classification.

For each of the data sets, the testing procedure used is as follows:

1. Train with input file for training and save the training model.

2. Perform a classification using the test file.

3. Use MATLAB to generate results of classification.

6.1.1 Classification with set1_1.txt and set1_2.txt

File set1_1.txt consists of 184 data points which can be linearly separated. The purpose of the test is to verify the linear SVM training model created by *SlimSVM*. The training with set1_1.txt file was done using default parameter values and a summary of the training result is provided in Table 6.1.

Table 6.1 Summary of training result using set1_1.txt

Kernel_Type:	Linear
#of Iterations:	47
Number_of_Classes:	2
Obj:	-59.608703
Rho:	-1.842552
Total Number of SVs:	88
Total SVs in Class #1:	44
Total SVs in Class #2:	44
Total Number of BSVs:	86

The model file generated from training was used to classify the data points in set1_2.txt. Figure 6.1 below shows the data points before classification and Figure 6.2 shows the data points after classification. The result of the classification is summarized in Table 6.2.

Table 6.2 Summary of testing result using set1_2.txt

Total # of Data To Classify:	343
# of Correct Classification:	281
Accuracy:	81.9242% (281/343)
Mean squared error:	0.180758
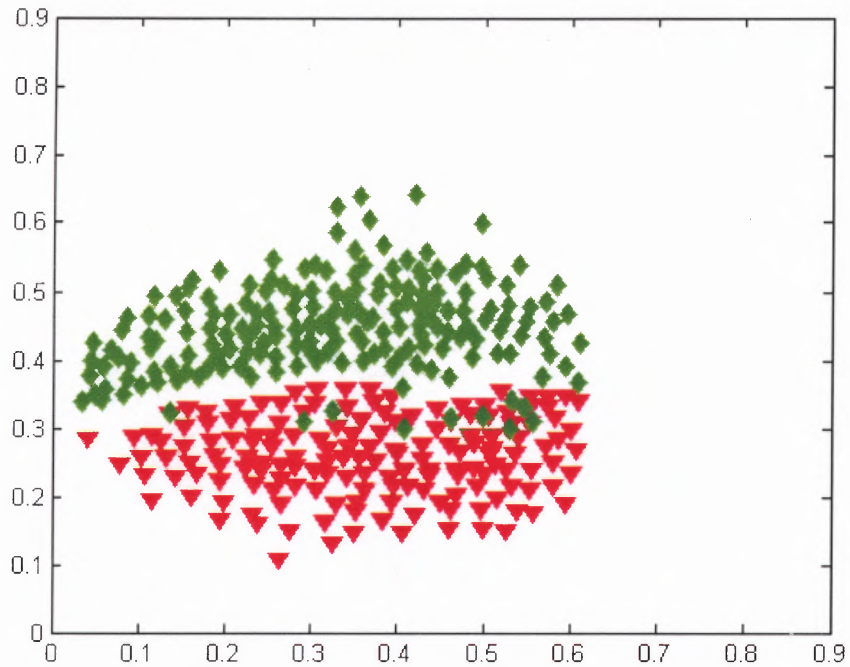
Figure 6.1 Data points from set1_2.txt file before classification. The data points from the first class are displayed as upside down red triangles and the second class is displayed as green diamonds.
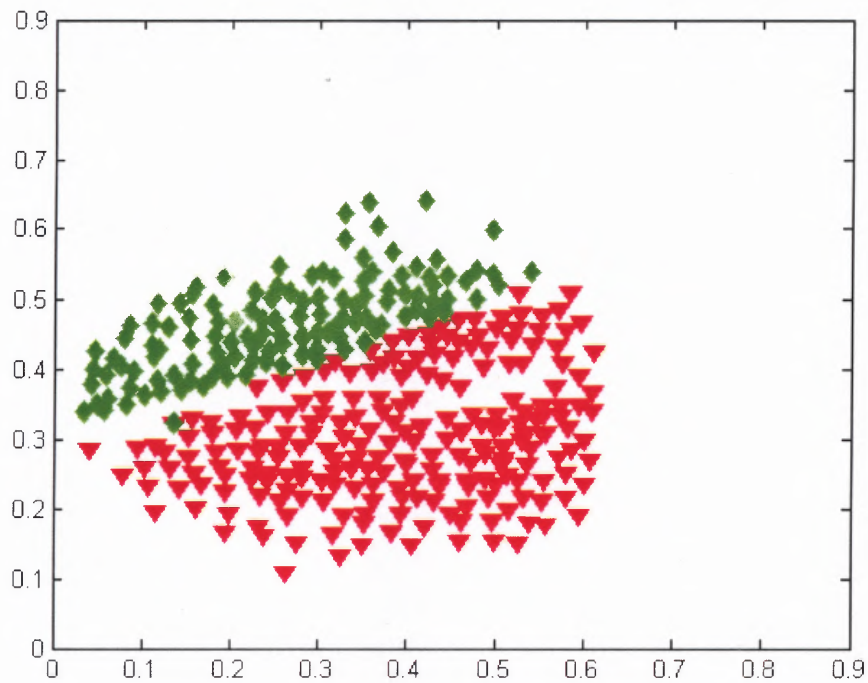
Figure 6.2 Data points from set1_2.txt file after classification.

6.1.2 Classification with set2_1.txt and set2_2.txt

File set2_1.txt consists of 127 data points which can be linearly separated. As mentioned above, unlike test set 1, the files in this set contain data from three classes. The purpose of the test is to verify the linear SVM training model for multi-class classification. The training with set2_1.txt file was also done using default parameter values and a summary of the training result is provided in Table 6.3.

Table 6.3 Summary of training result using set2_1.txt.

| Kernel_Type:		Linear
Number_of_Classes:		3
Total_Number_of_SVs:		108
Class #1	#of Iterations:	29
	Obj:	-36.242969
	Rho:	1.925537
	Total SVs in Class #1:	27
Class #2	#of Iterations:	18
	Obj:	-19.379895
	Rho:	0.040593
	Total SVs in Class #1:	46
Class #3	#of Iterations:	36
	Obj:	-52.625579
	Rho:	-3.071748
	Total SVs in Class #1:	35

The model file generated from training was used to classify the data points in set2_2.txt. Figure 6.3 below shows the data points before classification and Figure 6.4 shows the data points after classification. The result of the classification is summarized in Table 6.4.

Table 6.4 Summary of testing result using set2_2.txt.

Total # of Data To Classify:	90
# of Correct Classification:	85
Accuracy:	94.4444% (85/90)
Mean squared error:	0.0555556
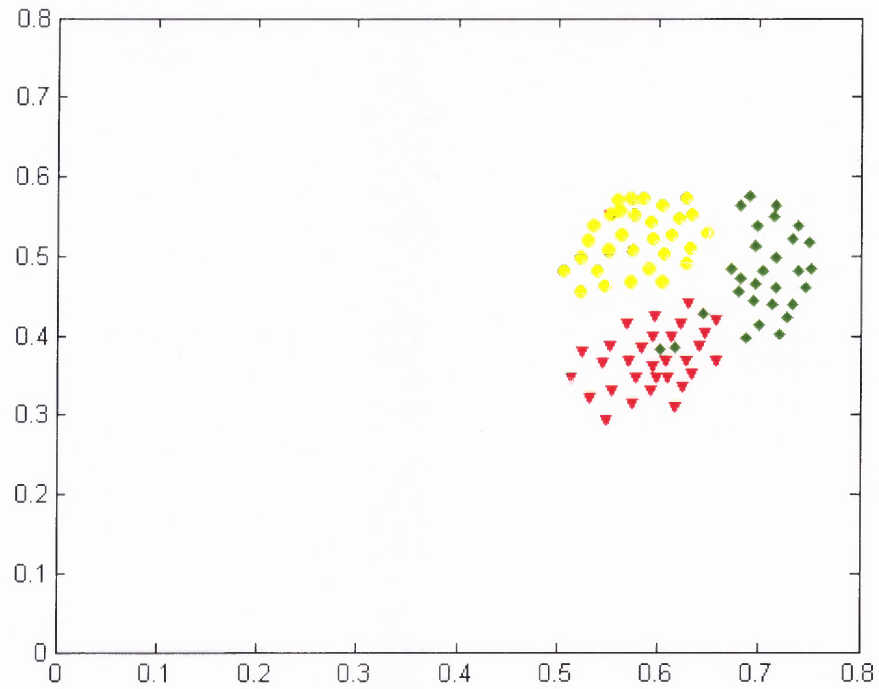
Figure 6.3 Data points from set2_2.txt file before classification. The data points from the first class are displayed as upside down red triangles, the second class is displayed as green diamonds and the third class is represented by yellow circles.
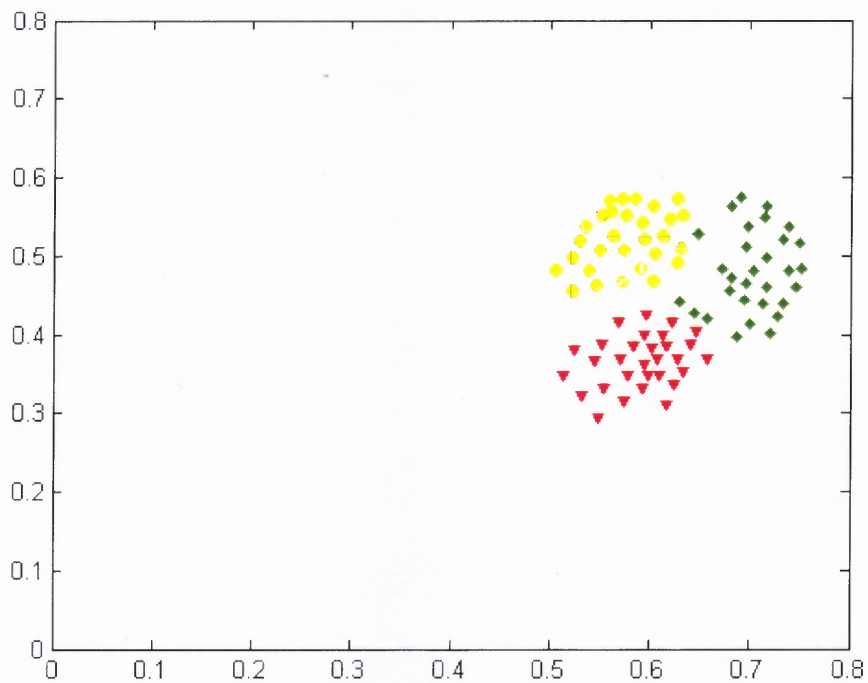
Figure 6.4 Data points from set2_2.txt file after classification.

6.1.3 Classification with set3_1.txt and set3_2.txt

File set3_1.txt consists of 164 data points which can be nonlinearly separated. The purpose of the test is to verify the polynomial kernel of SVM training model. The training of *SlimSVM* with set3_1.txt file was done using polynomial kernel with degree 2 and a cost value of 10 entered as custom parameter values. A summary of the training result is provided in Table 6.5.

Table 6.5 Summary of training result using set3_1.txt.

Kernel_Type:	Polynomial
Degree:	2
Gamma:	0.5
Coefficient:	0
#of Iterations:	112
Number_of_Classes:	2
Obj:	-448.023908
Rho:	3.741204
Total Number of SVs:	65
Total SVs in Class #1:	32
Total SVs in Class #2:	33
Total Number of BSVs:	62

The model file generated from training was used to classify the data points in set3_2.txt. The data points in this file are also nonlinearly separable. Figure 6.5 below shows the data points before classification and Figure 6.6 shows the data points after classification. The result of the classification is summarized in Table 6.6.

Table 6.6 Summary of testing result using set3_2.txt.

Total # of Data To Classify:	236
# of Correct Classification:	189
Accuracy:	80.0847% (189/236)
Mean squared error:	0.199153
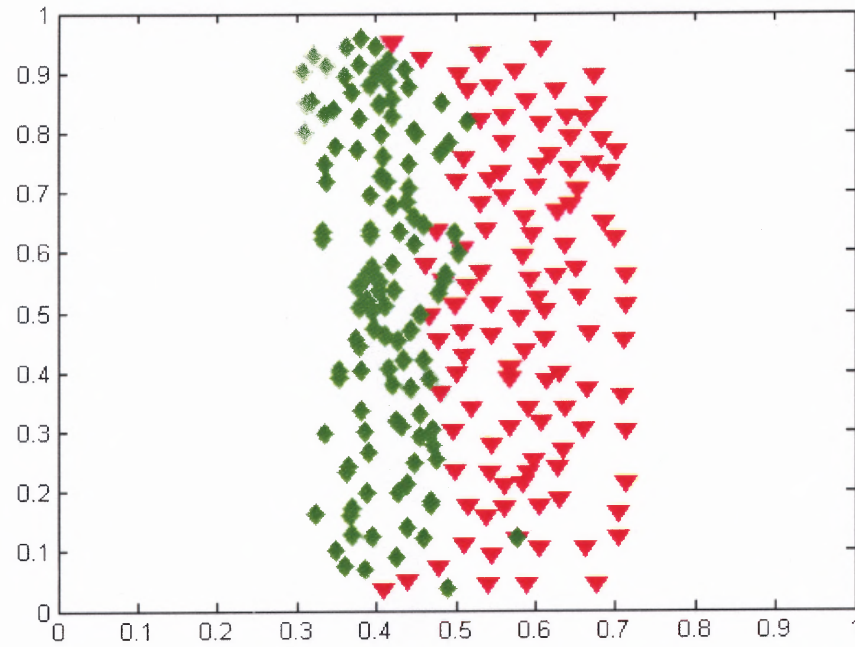
Figure 6.5 Data points from set3_2.txt file before classification. The data points from the first class are displayed as upside down red triangles, the second class is displayed as green diamonds.
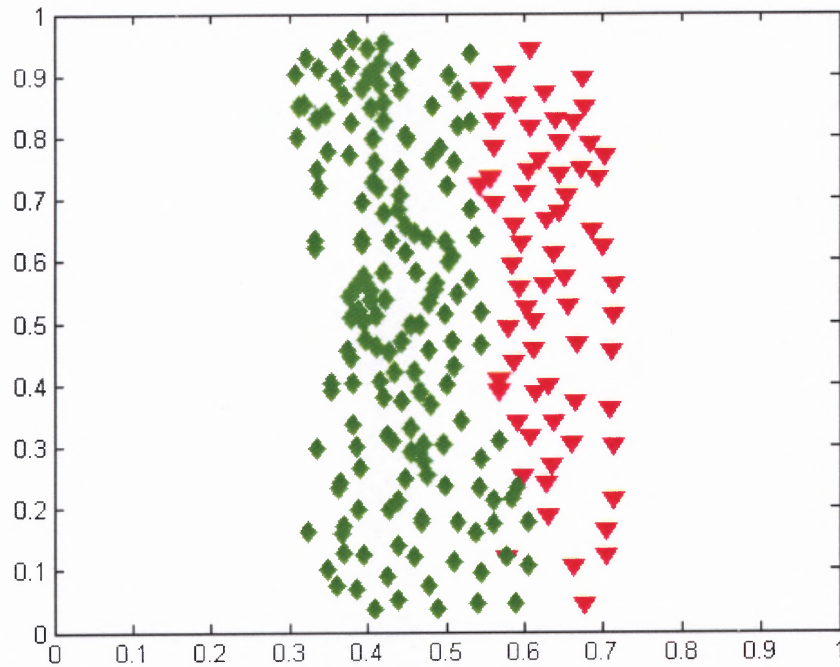
Figure 6.6 Data points from set3_2.txt file after classification.

6.2 Microarray Data

A second set of testing was done using microarray data. Since the ultimate reason for developing *SlimSVM* is analyzing microarrays, data from microarray experiments was used for testing.

6.2.1 Preparation of Data

From the microarray result file, data values from red and green intensity columns were extracted and grouped into three classes using a MATLAB script for both the training file and the testing file. In order to make the data files compatible with *SlimSVM*, Microsoft Excel was used to organize the data points in sparse format. Also, for each of the data point features the log was taken. The log value of each of the features was multiplied by 0.10 in order to normalize the data. All calculations were performed in Excel.

6.2.2 Classification of Data

The training file contains 2,293 two-dimensional data points. The first feature of each data point is from the red intensity column and the second feature is from the green intensity column. The training was done with default parameters for both the dot kernel and polynomial kernel. The result of training is summarized below in Table 6.7 and Table 6.8.

Table 6.7 Summary of microarray data training results using dot kernel.

Kernel_Type:		Linear
Number_of_Classes:		3
Total_Number_of_SVs:		445
Class #1	#of Iterations:	58
	Obj:	-41.342053
	Rho:	-0.029807
	Total SVs in Class #1:	126
Class #2	#of Iterations:	140
	Obj:	-151.198313
	Rho:	0.337679
	Total SVs in Class #1:	101
Class #3	#of Iterations:	105
	Obj:	-121.323754
	Rho:	0.190440
	Total SVs in Class #1:	218

Table 6.8 Summary of microarray data training results using polynomial kernel.

Kernel_Type:		Polynomial
Degree:		2
Gamma:		0.5
Coefficient:		0
Number_of_Classes:		3
Total_Number_of_SVs:		1346
Class #1	#of Iterations:	249
	Obj:	-296.614848
	Rho:	-0.083188
	Total SVs in Class #1:	490
Class #2	#of Iterations:	371
	Obj:	-503.575280
	Rho:	-0.668591
	Total SVs in Class #1:	458
Class #3	#of Iterations:	330
	Obj:	-439.077555
	Rho:	-0.613765
	Total SVs in Class #1:	398

The model file generated by training for both linear and non-linear training was used for classification of the unknown data points. The results are summarized in the tables below and figures are provided also for visualization of the results.

Table 6.9 Summary of classification results using dot kernel training model.

Total # of Data To Classify:	4419
# of Correct Classification:	4340
Accuracy:	98.2123% (4340/4419)
Mean squared error:	0.0334917

Table 6.10 Summary of classification results using polynomial kernel training model.

Total # of Data To Classify:	4419
# of Correct Classification:	4164
Accuracy:	94.2295% (4164/4419)
Mean squared error:	0.0902919
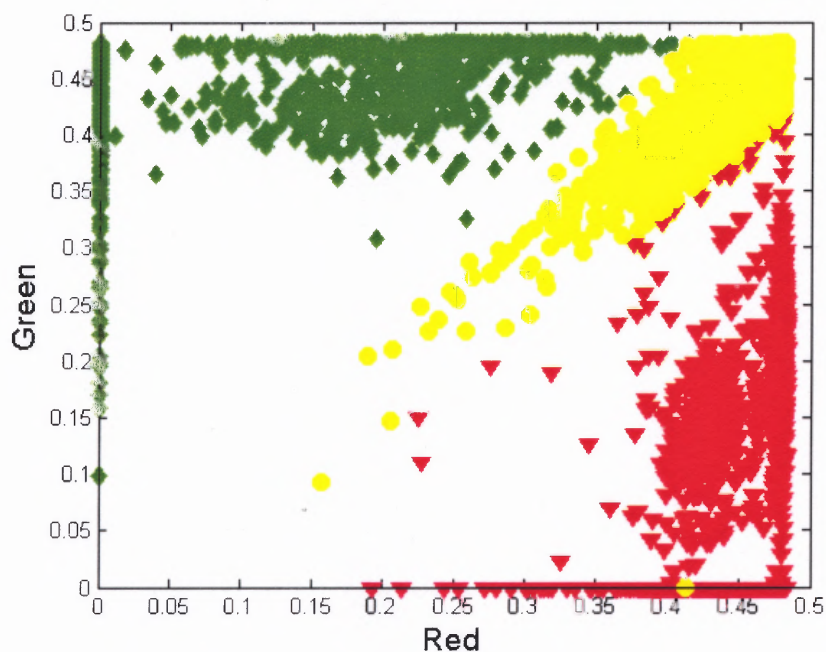
Figure 6.7 Microarray data points before classification. The data points from the first class are displayed as upside down red triangles, the second class is displayed as green diamonds and the third class is represented by yellow circles.
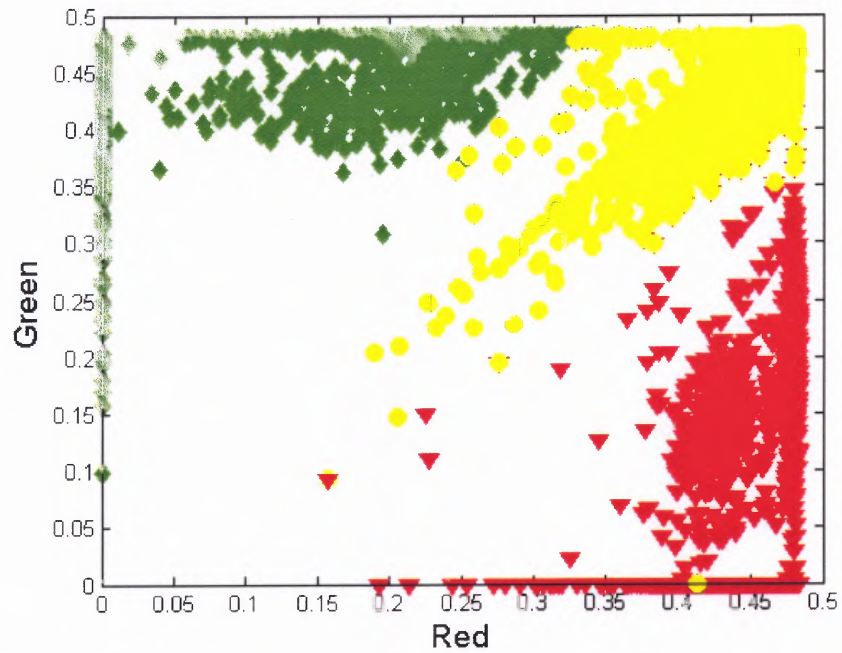
Figure 6.8 Microarray data points after classification using dot kernel.
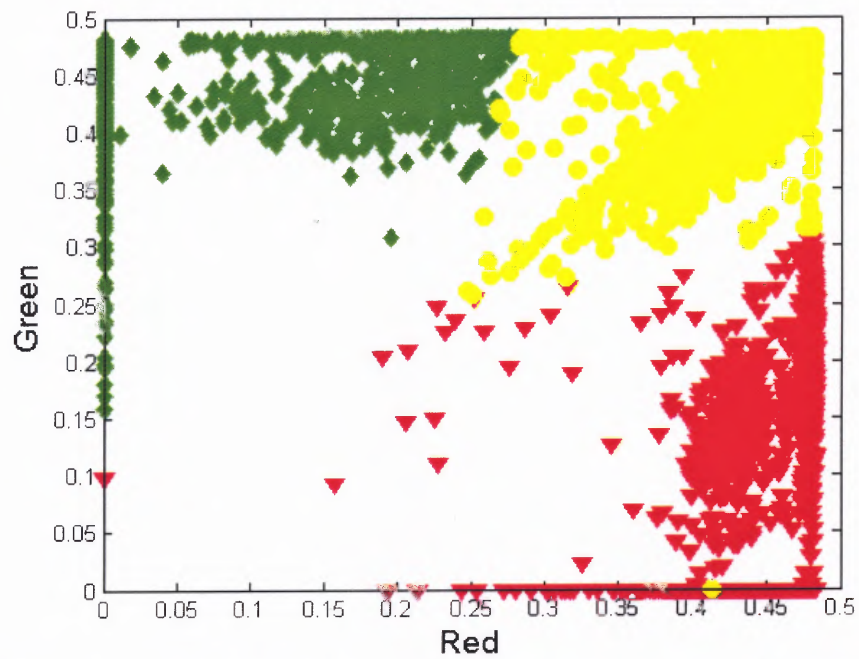
Figure 6.9 Microarray data points after classification using polynomial kernel.

6.3 Discussion of Classification Results

In order to verify the results obtained by *SlimSVM*, training and classification were performed using *LIBSVM* with the same data sets for both the artificial data and the microarray data. The results were identical, which indicates that the results from *SlimSVM* are correct. It should be noted that *SlimSVM* can handle multidimensional data points for both training and classification. It has been tested with multidimensional data that has been provided with *LIBSVM*, and the results were equivalent. In addition, the testing was not done with data sets containing millions of data points because the microarray data is usually two-dimensional and on the order of thousands or tens of thousands.

CHAPTER 7

CONCLUSION

Comparison of the training and classification results using the artificial and microarray data with those of *LIBSVM* shows that the implementation of *SlimSVM* has been carried out correctly. Training and testing with both artificial and microarray data took approximately the same amount of time. The number of data points in the artificial data files ranged from 90 to 343. The microarray data file for training consisted of 2,293 data points, and the classification file consisted of 4,419 data points. This shows that *SlimSVM* can perform training and classification at the same level of efficiency for small(hundreds) to medium(thousands) sets of data. Since microarray data files usually contain thousands or tens of thousands of data points, testing with larger data sets were not performed.

Most available implementations of SVM support dot and polynomial kernels along with additional kernels and functionalities not used by *GenoIterSVM*. Since *SlimSVM* was implemented to be used as the SVM classifier component of *GenoIterSVM* instead of *OSU SVM*, it supports only dot and polynomial kernel for classification as used by *GenoIterSVM* for genotype microarray analysis. The implementation of *SlimSVM* was designed to be simple and easy-to-use so that it can be easily used and modified. The program files are fully documented and commented for easy understanding of the algorithms. The documented code along with this thesis can be used for modification, maintenance and extension of *SlimSVM* by Dr. Ma's

group when developing *GenoIterSVM* as stand-alone program, since it is going to serve as the classifier of *GenoIterSVM*.

APPENDIX

SAMPLES FROM INPUT AND OUTPUT FILES

The appendix contains samples from the training file, model files and the classification

result file.

```
1 1:0.436000 2:0.376000
1 1:0.460000 2:0.380000
1 1:0.482000 2:0.394000
1 1:0.506000 2:0.414000
1 1:0.484000 2:0.422000
1 1:0.444000 2:0.408000
1 1:0.416000 2:0.386000
1 1:0.422000 2:0.420000
1 1:0.458000 2:0.442000
1 1:0.492000 2:0.450000
1 1:0.470000 2:0.470000
1 1:0.424000 2:0.454000
1 1:0.398000 2:0.430000
1 1:0.370000 2:0.384000
1 1:0.396000 2:0.354000
1 1:0.450000 2:0.348000
```

Figure A1 Data sample from training file.

```
Kernel_Type: Linear
Number_of_Classes: 2
Total_Number_of_SVs: 52
Rho: -0.300799
Labels: 1 2
Number_of_SVs: 26 26
SVs:
1 1:0.436 2:0.376
1 1:0.46 2:0.38
1 1:0.482 2:0.394
1 1:0.484 2:0.422
1 1:0.444 2:0.408
1 1:0.416 2:0.386
```

Figure A2 Sample from training model file using dot kernel.

77

```
Kernel_Type: Polynomial
Degree: 2
Gamma: 0.02
Coefficient: 0
Number_of_Classes: 2
Total_Number_of_SVs: 60
Rho: -0.0001262
Labels: 1 2
Number_of_SVs: 30 30
SVs:
1 1:0.436 2:0.376
1 1:0.46 2:0.38
1 1:0.482 2:0.394
1 1:0.506 2:0.414
```

Figure A3 Sample from training model file using polynomial kernel.

```
Result of classification based on set1_1_model.txt
1
1
1
2
1
2
2
2
1
1
1
1
1
```

Figure A4 Sample from classification result file.

REFERENCES

1. Boser, B. E., Guyon, I. M. and Vapnik, V. N. (1992), A Training Algorithm for Optimal Margin Classifier. *In Proceedings of the 5th Annual ACM Workshop on Computational Learning Theory*, 144-152.

2. Brown, M., Grundy, W., Lin, D., Cristianini, N., Sugnet, C., Furey, T., M. Ares, J. and Haussler, D. (2000). Knowledge-based analysis of Microarray Gene Expression Data by Using Support Vector Machines. *Proc. Natl. Acad. Sci. USA,* 97, 262-267.

3. Chang, C. C. and Lin, C. J. LIBSVM: A Library for Support Vector Machines, 2001. http://www.csie.ntu.edu.tw/~cjlin/libsvm/

4. Chu, S., DeRisi, J., Eisen, M., Mulholland, J., Botstein, D., Brown, P. and Herskowitz, I. (1998). The Transcriptional Program of Sporulation in Budding Yeast. *Science*, 282, 699-705.

5. Cristianini, N., & Shawe-Taylor, J. (2000). *An Introduction to Support Vector Machines and Other Kernel-Based Learning Methods.* Cambridge, UK: Cambridge University Press.

6. Daume, H. *SVMsequel Documentation.* Retrieved August 7, 2004, from SVMsequel Web site: http://www.isi.edu/~hdaume/SVMsequel/

7. DeRisi, J., Iyer, V. and Brown, P. (1997). Exploring the Metabolic and Genetic Control of Gene Expression on Genomic Scale. *Science,* 278, 680-686.

8. DeRisi, J., Penland, L., Brown, P., Bittner, M., Meltzer, P., Ray, M., Chen, Y., Su, Y. and Trent, J. (1996). Use of CDNA Microarray to Analyse Gene Expression Patterns in Human Cancer. *Nat. Genet.*, 4, 457-460.

9. Ding, C., Dubchak, I. (2001). Multi-class Protein Folding Recognition Using Support Vector Machine and Neural Networks. *Bioinformatics*, 17, 349-358.

10. Duda, R. and Hart, P. (1973). *Pattern Classification and Scene Analysis.* New York: Wiley.

11. *GIST:Support Vector Machine 1.0.* http://svm.sdsc.edu/cgi-bin/nph-SVMsubmit.cgi

12. Golub, T., Slonim, D., Tamayo, P., Huard, C., Gaasenbeek, M., Mesirov, J., Coller, H., Loh, M., Downing, J., Caligiuri, M., Bloomfield, C. and Lander, E. (1999). Molecular Classification of Cancer: Class Discover and Class Prediction by Gene Expression Monitoring. *Science,* 286, 531-537.

13. Hsu, C-W., Chang, C. and Lin, C. *A Practical Guide to Support Vector Classification.* Retrieved March 23, 2004, from LIBSVM: A Library for Support Vector Machines Web site: http://www.csie.ntu.edu.tw/~cjlin/papers/guide/guide.pdf

14. Joachims, T. Making Large-scale SVM Learning Practical. (1998). In B. Scholkopf, C. Burgess and A. Smola(Eds.), *Advances in Kernel Methods – Support Vector Learning,* Massachusetts: MIT Press.

15. Joachims, T. *SVM^{Light} Support Vector Machine.* http://svmlight.joachims.org/.

16. Knerr, S., Personnaz, L., and Dreyfus, G. (1990). Single-Layer Learning Revisited: A Stepwise Procedure for Building and Training a Neural Network. In J. Fogelman (Ed.), *Neurocomputing:Algorithms, Architectures and Applications.* New York: Springer-Verlag.

17. Ma, M., Zhang, K., Hu, G., Wang, H., Luo, M., Wang, J. and Li, H. (2004). SNP Genotype-calling Using SVMs with Iterative Refinement. In preparation.

18. Mukherjee, S. Classifying Microarray Data Using Support Vector Machines. *Understanding and Using Microarray Analysis Techniques: A Practical Guide (in press).*

19. Muller, K., Mika, S., Ratsch, G., Tsuda, K., and Scholkopf, B. (2001). An Introduction to Kernel-Based Learning Algorithm. *IEEE Transactions of Neural Networks,* 12(2), 181-202.

20. Osuna, E., Freund, R. and Girosi, F. (1997). Training Support Vector Machines: An Application to Face Detection. *In proceedings of CVPR,* 97, 130-136.

21. Platt, J. Fast Training of Support Vector Machines Using Sequential Minimal Optimization. (1998) In B. Scholkopf, C. Burgess and A. Smola(Eds.), *Advances in Kernel Methods – Support Vector Learning,* Massachusetts: MIT Press

22. Rüping, S. mySVM - A Support Vector Machine. http://www-ai.cs.uni-dortmund.de/SOFTWARE/MYSVM/index.html

23. Saunders, C., Stitson, M., Weston, J. Bottou, L., Scholkopf, B. and Smola, A.(1998). Support Vector Machine Reference Manual. *Technical Report CSD-TR-98-03.*

24. Scholkopf, B. and Smola, A. (2001).*Learning with Kernels.* Massachusetts: MIT Press.

25. Vapnik, V. (1995). *The Nature of Statistical Learning Theory.* New York: Springer-Verlag.

26. Vapnik, V. Structure of Statistical Learning Theory. (1996). In A. Gammerman (Ed.), *Computational Learning and Probabilistic Reasoning* (pp. 33-41).New York: Wiley.

27. Vapnik, V. (1998). *Statistical Learning Theory*. New York: Wiley.

28. Wen, X, Fuhrman, S., Michaels, G., Carr, D., Smith, S., Barker, J. and Somogyi, R. (1998). Large-scale temporal gene expression mapping of central nervous system development. *Proc. Natl. Acad. Sci. USA,* 95, 334-339.

29. Zhu, H., Cong, J., Mamtora, G., Gingeras, T. and Schenk, T. (1998) Cellular gene expression altered by human cytomegalovirus: global monitoring with oligonucleotide arrays. *Proc. Natl. Acad. Sci. USA,* 95, 14470-14475.