8-7-2008

Distributed Support Vector Machine Learning

Kenneth C. Armond Jr.
University of New Orleans

Distributed Support Vector Machine Learning

A Thesis

Submitted to the Graduate Faculty of the
University of New Orleans
in partial fulfillment of the
requirements for the degree of

Master of Science
in
Computer Science
Bioinformatics

by

Kenneth C. Armond Jr.

B.S. Texas A&M University, 2003

August, 2008

Acknowledgment

I would like to thank my advisor, Dr. Stephen Winters-Hilt. He has helped me tremendously to understand the importance of academic research. He gave me the enthusiasm needed to begin research in the field of Bioinformatics, which I previously did not know anything about. I aspire to keep this enthusiasm in future endeavors, as he has after years of research. His intelligence and accomplishments make him an excellent mentor yet he still knows how to balance work and fun.

I would like to thank Sam Merat who was my student mentor and we worked together on the early SVM Java code that eventually led to further work on the different chunking methods described here. His knowledge was integral to helping me understand the SVM algorithm.

I would like to thank Carl Baribault who was always happy and willing to answer the many questions I had. His collaboration on the Support Vector Reduction method was very helpful. I have gained much insight on a variety of subjects from talking with him.

I would also like to think the other members of the Bioinformatics group for their help and support.

Finally, I would like to express my deep gratitude to my wife, Stephanie, and my son, Linus. Their encouragement and unconditional support have made getting this degree much easier to attain than previously thought.

Table of Contents

List of Figures

List of Tables

List of Illustrations

Abstract

Support Vector Machines (SVMs) are used for a growing number of applications. A fundamental constraint on SVM learning is the management of the training set. This is because the order of computations goes as the square of the size of the training set. Typically, training sets of 1000 (500 positives and 500 negatives, for example) can be managed on a PC without hard-drive thrashing. Training sets of 10,000 however, simply cannot be managed with PC-based resources. For this reason most SVM implementations must contend with some kind of chunking process to train parts of the data at a time (10 chunks of 1000, for example, to learn the 10,000). Sequential and multi-threaded chunking methods provide a way to run the SVM on large datasets while retaining accuracy. The multi-threaded distributed SVM described in this thesis is implemented using Java RMI, and has been developed to run on a network of multi-core/multi-processor computers.

Keywords

Distributed

Parallel

SVM

Support Vector Machine

Machine Learning

SMO

Sequential Minimization Optimization

Chapter 1. Introduction

SVMs are becoming more popular since researchers and engineers find it to be a better, more robust replacement for older learning methods such as neural networks. Another attractive aspect of SVMs is that they do not require much manual parameter manipulation which makes their use much easier. Neural networks are a good example of an outdated learning method that requires a decent amount of expertise in order to get the training parameters properly set for each different dataset. Many areas of study such as text categorization, face recognition, handwriting recognition, pedestrian detection, and DNA hairpin classification are increasingly using SVMs as their main classification method.

The main problem with current day SVMs is that they cannot process large datasets in a timely manner. This problem is compounded further when multiple SVM training rounds are needed as with SVM clustering methods being developed by the Winters-Hilt Group (but not discussed further here). Chunking the training set into smaller datasets provides a solution to this problem. Sequential chunking runs the SVM on the first chunk and then sends the support feature vectors (SVs) and sometimes some non-SVs to be added into the training data for the next chunk. This continues until all chunks have been run. What is not as commonly discussed are multi-threaded chunk processing methods. In part, this may be because of the many subtleties that have been encountered in these efforts and this is a major focus of this thesis. Multi-threaded chunking breaks the training dataset into smaller chunks to be trained separately. When all chunks have been trained, SVs and sometimes some non-SVs are brought together and re-chunked. This occurs until the dataset is small enough to be handled by the basic SVM. In order to take advantage of multi-system networks, a multi-threaded distributed chunking method has also been developed to spread the load and significantly decrease training time.

During research of this topic, observations have shown that not all support vectors are needed in order to define an accurate hyperplane. To capitalize on this fact, a Support Vector Reduction method has been developed to drop the weakest SVs. This method has not only further advanced the stand-alone SVM but has significantly increased the effectiveness of the chunking methods.

A plethora of results have been compiled to support the effectiveness of the discussed methods which have been placed throughout the discussion. The Appendix contains the implementation of the SVM methods written in Java.

Chapter 2. Support Vector Machines

2.1 Introduction

Support Vector Machines [10] (SVMs) are discriminators that use structural risk minimization to find a decision hyperplane with a maximum margin between separate groupings of feature vectors. SVMs are often used to classify binary and multi-class datasets. The chunking algorithms discussed below concentrate on binary classification. The feature vectors have been extracted from different blockade level frequencies, the emission probabilities, and transition probabilities to arrive at probability vectors usually composed of 150 components [1].

When SVMs were created in 1995, a quadratic programming algorithm was used [5]. This was slow and only small datasets could be run with them. In 1998, Platt created Sequential Minimal Optimization (SMO) which is an algorithm that uses minimal sets of Lagrange multipliers (here two) to bypass having to use a quadratic algorithm [2]. The SMO SVM iterates through the dataset comparing and updating the Lagrange multipliers (alphas) two at a time. This simplification into smaller steps provides a significant increase in speed when compared to the older quadratic algorithms. This new approach to SVMs has opened its use to a wide variety of applications.

2.2 Applications of Support Vector Machines

Text categorization is used to classify documents into different predefined categories. Some examples of documents are news articles, websites, and newsgroup postings. The feature vectors are composed of the word stems of frequently used words within the document. Word stems refer to the core of the word which ignores word additions such as “ing”, “ed”, and plural endings. Also, common grammar words like “the”, “and”, and “is” are not counted as feature vectors since they are not much use for distinguishing between document types. Inverse

document frequency (IDF), which is calculated from the frequency that words occur across all documents in the training set, is used to scale the feature vectors for better performance. Since feature spaces tend to be relatively large, information gain is often used to rank the feature vectors to find the ones with the highest mutual information. SVMs have replaced methods such as Naïve Bayes Classifiers, the Rocchio Algorithm, k-Nearest Neighbors, and Decision Tree Classifiers for text categorization [11].

Face recognition can be used to determine if a human face exists in an image. This is useful for human-computer interfaces, surveillance systems, and other automated processes that require face detection. SVMs are excellent for this use since they can find details such as facial expressions, mustaches, etc. even in varying light conditions. Feature vectors are created from face/non-face pixels taken over several scales to perform the binary classification. Preprocessing of the image is performed to get rid of data points that can later contribute to noise. First, the image is scanned for pixels that are too close to call between face/non-face statuses. Next, an illumination gradient correction is done to normalize the lighting thus accounting for bright lights, glare, and shadows. Lastly, histogram equalization is done to distinguish between large contrasts of brightness. Once the SVM is trained, it can be used against other images to classify the location of faces. Some of the previous methods of face recognition were done using Neural Networks, labeled graphs, and clustering and distribution-based modeling [12].

Handwritten digit recognition is another area of study that benefits from SVMs. This can be used for a variety of applications. Attributed from the study by LeCun et al. [13], this could be used to scan in address information for the U.S. Postal Service. The training data is generated by taking each handwritten digit and putting it into a 20x20 pixel box. Each pixel contributes to a weighted sum that is used to get the output units. The pixels that are all or majority black are

considered part of the written number. The majority white pixels are the unwritten region of the box. The pixel data make up the polarized feature vectors which are using for training/testing. Many other methods have been used to classify this type of data including Neural Networks, Nearest Neighbor Classifiers, and variations of the LeNet Classifier [13].

SVMs are currently used to classify channel current data produced from nanopore detectors. Strands of DNA hairpins are pulled into the channel via an applied potential. The electrodes send pA electrical currents across the channel and the current fluctuations are collected by a live data stream into a computer. This data is then sent through a Hidden Markov Model (HMM) process to remove noise. The probabilities established by the HMM make up the 150 component feature vectors. The polarity comes from the two different molecules that are being classified such as 9GC and 9CG. SVMs are useful for this area of study since the data is often difficult to separate by previous methods such as Neural Networks. Several different kernel spaces must be used in training in order to find the best one for the given type of dataset.

2.3 Derivation of Binary SVM [1]

Feature vectors are denoted by x_{ik}, where index i labels the feature vectors ($1 \leq i \leq M$) and index k labels the N feature vector components ($1 \leq i \leq N$). For the binary SVM, labeling of training data is done using label variable $y_i = \pm 1$ (with sign according to whether the training instance was from the positive or negative class). For hyperplane separability, elements of the training set must satisfy the following conditions: $w_\beta x_{i\beta} - b \geq +1$ for i such that $y_i = +1$, and $w_\beta x_{i\beta} - b \leq -1$ for $y_i = -1$, for some values of the coefficients $w_1,..., w_N$, and b (using the convention of implied sum on repeated Greek indices). This can be written more concisely as: $y_i(w_\beta x_{i\beta} - b) - 1 \geq 0$. Data points that satisfy the equality in the above are known as "support vectors" (or "active constraints").

Once training is complete, discrimination is based solely on position relative to the discriminating hyperplane: $w_\beta x_{i\beta} - b = 0$. The boundary hyperplanes on the two classes of data are separated by a distance 2/w, known as the "margin," where $w^2 = w_\beta w_\beta$. By increasing the margin between the separated data as much as possible the optimal separating hyperplane is obtained. In the usual SVM formulation, the goal to maximize w^{-1} is restated as the goal to minimize w^2. The Lagrangian variational formulation then selects an optimum defined at a saddle point of

$$L(\mathrm{w},\mathrm{b};\alpha) = \frac{\mathrm{w}_\beta \mathrm{w}_\beta}{2} - \alpha_\gamma y_\gamma (\mathrm{w}_\beta \mathrm{w}_{\gamma\beta} - \mathrm{b}) - \alpha_0$$

$$\text{where } \alpha_0 = \sum_\gamma \alpha_\gamma \ , \ \alpha_\gamma \geq 0 \quad (1 \leq \gamma \leq M)$$

The saddle point is obtained by minimizing with respect to $\{w_1,...,w_N,b\}$ and maximizing with respect to $\{\alpha_1, ..., \alpha_M\}$. If $y_i(w_\beta x_{i\beta} - b) - 1 \geq 0$, then maximization on α_i is achieved for $\alpha_i = 0$. If $y_i(w_\beta x_{i\beta} - b) - 1 = 0$, then there is no constraint on α_i. If $y_i(w_\beta x_{i\beta} - b) - 1 < 0$, there is a constraint violation, and $\alpha_i \rightarrow \infty$. If absolute separability is possible, the last case will eventually be eliminated for all α_i, otherwise it is natural to limit the size of α_i by some constant upper bound, i.e., $\max(\alpha_i) = C$, for all i. This is equivalent to another set of inequality constraints with $\alpha_i \leq C$. Introducing sets of Lagrange multipliers, ξ_γ and $\mu_\gamma (1 \leq \gamma \leq M)$, to achieve this, the Lagrangian becomes:

$$L(\mathrm{w},\mathrm{b};\alpha,\xi,\mu) = \frac{\mathrm{w}_\beta \mathrm{w}_\beta}{2} - \alpha_\gamma [y_\gamma (\mathrm{w}_\beta \mathrm{x}_{\gamma\beta} - \mathrm{b}) + \xi_\gamma] + \alpha_0 + \xi_0 C - \mu_\gamma \xi_\gamma$$

$$\text{where } \xi_0 = \sum_\gamma \xi_\gamma \ , \ \alpha_0 = \sum_\gamma \alpha_\gamma \text{ and } \alpha_\gamma \geq 0 \text{ and } \xi_\xi \geq 0 \quad (1 \leq \gamma \leq M)$$

At the variational minimum on the $\{w_1,...,w_N,b\}$ variables, $w_\beta = \alpha_\gamma y_\gamma x_{\gamma\beta}$, and the Lagrangian

simplifies to:

$$L(\alpha) = \alpha_0 - \frac{\alpha_\delta y_\delta x_{\delta\beta} \alpha_\gamma y_\gamma x_{\gamma\beta}}{2}$$

$$\text{with } 0 \le \alpha_\gamma \le C \ \ (1 \le \gamma \le M) \text{ and } \alpha_\gamma y_\gamma = 0 ,$$

where only the variations that maximize in terms of the α_γ remain (known as the Wolfe Transformation). In this form the computational task can be greatly simplified.

By introducing an expression for the discriminating hyperplane, $f_i = w_\beta x_{i\beta} - b = \alpha_\gamma y_\gamma x_{\gamma\beta} x_{i\beta} - b$, the variational solution for $L(\alpha)$ reduces to the following set of relations (known as the Karush-Kuhn-Tucker, or KKT, relations):

(i) $\alpha_i = 0$, $y_i f_i \ge 1$

(ii) $0 < \alpha_i < C$, $y_i f_i = 1$

(iii) $\alpha_i = C$, $y_i f_i \le 1$

When the KKT relations are satisfied for all of the α_γ (with $\alpha_\gamma y_\gamma = 0$ maintained) the solution is achieved. The constraint $\alpha_\gamma y_\gamma = 0$ is satisfied for the initial choice of multipliers by setting the α's associated with the positive training instances to 1/N(+) and the α's associated with the negatives to 1/N(-), where N(+) is the number of positives and N(-) is the number of negatives. Once the Wolfe transformation is performed it is apparent that the training data (support vectors in particular, KKT class (ii) above) enter into the Lagrangian solely via the inner product $x_{i\beta} x_{j\beta}$. Likewise, the discriminator f_i, and KKT relations, are also dependent on the data solely via the $x_{i\beta} x_{j\beta}$ inner product.

Generalization of the SVM formulation to data-dependent inner products other than $x_{i\beta} x_{j\beta}$ are possible and are usually formulated in terms of the family of symmetric positive definite functions (reproducing kernels) satisfying Mercer's conditions [10].

The SVM discriminators are trained by solving their KKT relations using the SMO procedure of [8]. The method described here follows the description of [8] and begins by selecting a pair of Lagrange multipliers, $\{\alpha_1,\alpha_2\}$, where at least one of the multipliers has a violation of its associated KKT relations. For simplicity it is assumed in what follows that the multipliers selected are those associated with the first and second feature vectors: $\{x_1,x_2\}$. The SMO procedure then "freezes" variations in all but the two selected Lagrange multipliers, permitting much of the computation to be circumvented by use of analytical reductions:

$$L(\alpha_1,\alpha_2;\alpha_{\beta'\geq 3}) = \alpha_1 + \alpha_2 - \frac{(\alpha_1^2 K_{11} + \alpha_2^2 K_{22} + 2\alpha_1\alpha_2 y_1 y_2 K_{12})}{2} - \alpha_1 y_1 v_1 - \alpha_2 y_2 v_2 + \alpha_{\beta'} U_{\beta'} - \frac{\alpha_{\beta'}\alpha_{y'} y_{\beta'} K_{\beta' y'}}{2}$$

with $\beta',\gamma' \geq 3$, and where $K_{ij} \equiv K(x_i, x_j)$, and $v_i \equiv \alpha_{\beta'} y_{\beta'} K_{i\beta'}$ with $\beta' \geq 3$. Due to the constraint $\alpha_\beta y_\beta = 0$, we have the relation: $\alpha_1 + s\alpha_2 = -\gamma$, where $\gamma \equiv y_1 \alpha_{\beta'} y_{\beta'}$ with $\beta' \geq 3$ and $s \equiv y_1 y_2$. Substituting the constraint to eliminate references to α_1, and performing the variation on α_2: $\partial L\ (\alpha_2\ ;\ \alpha_{\beta'} \geq 3)/\partial\alpha_2 = (1 - s) + \eta\alpha_2 + s\gamma(K_{11} - K_{22}) + s y_1 v_1 - y_2 v_2$, where $\eta \equiv (2K_{12} - K_{11} - K_{22})$. Since v_i can be rewritten as $v_i = w_\beta x_{i\beta} - \alpha_1 y_1 K_{i1} - \alpha_2 y_2 K_{i2}$, the variational maximum $\partial L\ (\alpha_2\ ;\ \alpha_{\beta'} \geq 3)/\partial\alpha 2 = 0$ leads to the following update rule:

$$\alpha_2^{new} = \alpha_2^{old} - \frac{y_2((w_\beta x_{1\beta} - y_1) - (w_\beta x_{2\beta} - y_2))}{\eta}$$

Once α_2^{new} is obtained, the constraint $\alpha_2^{new} \leq C$ must be re-verified in conjunction with the $\alpha_\beta y_\beta = 0$ constraint. If the $L\ (\alpha_2;\alpha_{\beta'} \geq 3)$ maximization leads to a α_2 new that grows too large, the new α_2 must be "clipped" to the maximum value satisfying the constraints. For example, if $y_1 \neq y_2$, then increases in α_2 are matched by increases in α_1. So, depending on whether α_2 or α_1 is nearer its maximum of C, we have $\max(\alpha_2) = \text{argmin}\{\alpha_2 + (C - \alpha_2)\ ;\ \alpha_2 + (C - \alpha_1)\}$. Similar arguments provide the following boundary conditions:

(i) if $s = -1$, $\max(\alpha_2) = \text{argmin}\{\alpha_2\ ;\ C + \alpha_2 - \alpha_1\}$, and $\min(\alpha_2) = \text{argmax}\{0\ ;\ \alpha_2 - \alpha_1\}$, and (ii) if $s =$

+1, $\max(\alpha_2) = \text{argmin}\{C ; \alpha_2 + \alpha_1\}$, and $\min(\alpha_2) = \text{argmax}\{0 ; \alpha_2 + \alpha_1 - C\}$.

In terms of the new $\alpha_2^{new, clipped}$, clipped as indicated above if necessary, the new α_1 becomes:

$\alpha_1^{new} = \alpha_1^{old} + s(\alpha_2^{old} \alpha_2^{new,clipped})$ where $s \equiv y_1 y_2$ as before. After the new α_1 and α_2 values are obtained there still remains the task of obtaining the new b value. If the new α_1 is not "clipped" then the update must satisfy the non-boundary KKT relation: $y_1 f(x_1) = 1$, i.e., $f^{new}(x_1) - y_1 = 0$. By relating f^{new} to f^{old} the following update on b is obtained:

$$b_1^{new} = b - (f^{new}(x_1) - y_1) - y_1(\alpha_1^{new} - \alpha_1^{old})K_{11} - y_2(\alpha_2^{new,clipped} - \alpha_2^{old})K_{12}$$

If α_1 is clipped but α_2 is not, the above argument holds for the α_2 multiplier and the new b is:

$$b_2^{new} = b - (f^{new}(x_2) - y_2) - y_2(\alpha_2^{new} - \alpha_2^{old})K_{22} - y_1(\alpha_1^{new,clipped} - \alpha_1^{old})K_{12}$$

If both α_1 and α_2 values are clipped then any of the b values between b_1^{new} and b_2^{new} is acceptable, and following the SMO convention, the new b is chosen to be:

$$b^{new} = \frac{b_1^{new} + b_2^{new}}{2}$$

Now that there is a more detailed understanding of the inner workings of the SVM, the tuning parameters are discussed in the following section.

2.4 SMO Parameters

The SVM takes a set of parameters that are used to finely tune the classifier for different types of datasets and for better performance. The C parameter is a constant value that serves as the upper boundary on the Lagrange multipliers (alphas). The C value constrains the alpha from potentially going to an infinite value. It trades off a wide margin with a few possible margin failures [2].

The tolerance parameter is a margin of error put on the one (right hand side) when checking the KKT relations. For instance, instead of checking only against one, we are checking

the range .999 – 1.001 (for tolerance .001). The choice of tolerance influences the SVM convergence.

The epsilon parameter is another margin of error used when checking the H and L objective functions [2]. These functions are used on the alphas when η is negative which is uncommon. The H and L functions [2] clip the new alpha to keep it within the bounds of 0 and C. The value of η comes from a simple calculation using the values from the kernel matrix: $\eta = 2K(x_1,x_2) - K(x_2,x_2) - K(x_1,x_1)$ [1].

The kernel matrix is produced from the specified kernel and the sigma constant parameter. The matrix values are multiplied with the alphas in order to change the mapping of the data points for easier classification. The choice of kernel determines the spatial mapping of the data. Further details on a set of novel kernels are given in the next section.

2.5 'Stabilization' Kernels

The kernel is probably the most important parameter to focus on when tuning the SVM. There are a wide variety of kernels to choose from so one must find the best kernel that works for the given type of dataset. The kernel is used to map the feature vectors into a multi-dimensional space. The idea is to find the kernel that best maps the feature vectors in a way that allows the hyperplane to find the optimal separation, thus the best decision.

For DNA hairpin feature vector datasets, our observations have shown the best kernels to be the Gaussian, Absdiff, and Sentropic kernels. The Gaussian and Absdiff kernels are regularized distances in the form of an exponential distance measure ($d^2(x,y)$). The Gaussian kernel ($d^2(x,y) = \Sigma_k(x_k - y_k)^2$) is common since it tends to produce good results when used with a wide variety of datasets. The Absdiff ($d^2(x,y) = \Sigma_k(|x_k - y_k|)^{1/2}$) and Sentropic ($D(x,y) = D(x||y) + D(y||x)$) Kernels [1] tend to work better with more cohesive datasets since they seem to provide a

larger kernel space. The Sentropic kernel is based on a regularized information divergence (D(x,y)) instead of a geometric distance. This can help produce to a more precise hyperplane. Out of the three kernels mentioned above, Absdiff and Sentropic produce similar results when considering the mean of the Sensitivity (SN) and Specificity (SP), which is used to measure accuracy, and the size of the final chunk for the chunking methods. The choice of kernel makes a difference in the size of the chunks, mainly the number of support vectors, which impacts the run time of the algorithm. The accuracy of Absdiff (0.854) and Sentropic (0.855) are nearly identical when using multi-threaded chunking (Tables 2.1 and 2.2). Gaussian is close behind with accuracy 0.833 (Table 2.3). For the sequential chunking method, the same case with accuracy applies since Absdiff (0.898) and Sentropic (0.891) produce similar results (Tables 2.4 and 2.5). Once again, Gaussian is close behind with accuracy 0.864 (Table 2.6). In these data runs, 30% of the support vector set was passed to the next set of chunks for the multi-threaded chunking method and 100% of the support vector set was passed for the sequential chunking method. These chunking parameters were chosen since they produced the best accuracy for the given chunking method. This is analyzed in more depth later in section 4.

When focusing on the size of the final chunk, Absdiff (1472) and Sentropic (1481) are once again similar for sequential chunking (Tables 2.7 and 2.8). Gaussian takes the lead with a final chunk size of 1264 (Table 2.9). For the multi-threaded chunking method, Absdiff (791) and Sentropic (787) once again fall behind Gaussian (690) for the final chunk size comparisons (Tables 2.10, 2.11, and 2.12). For these data runs, 80% of the support vector set and 60% of the polarization set were passed to the next set of chunks for the multi-threaded chunking method and 100% of the support vector set and 50% of the polarization set were passed for the sequential chunking method. These chunking parameters were chosen since they produce a similar

accuracy when compared to the parameters discussed above (30% for multi-threaded and 100% for sequential) while passing a larger amount of feature vectors in order to properly test the chunk size performance of each kernel.

After testing the three kernels, Absdiff was chosen as the best kernel for the DNA hairpin datasets used here since it has high accuracy and also takes the least amount of iterations to converge which contributes to it being the fastest for training these datasets. For multi-threaded chunking, the mean of iterations is 16.7 and the elapsed time is 1393.5 milliseconds (Table 2.1). For sequential chunking, the mean of iterations is 51.6 and the elapsed time is 10586.1 milliseconds (Table 2.4). After considering all of the above results, Absdiff was the chosen kernel in all following data runs in sections 3 and 4.

Table 2.1
Multi-threaded chunking using different DNA hairpin datasets
SVM Parameters: Absdiff kernel with sigma=.5, C = 10,
Epsilon = .001, Tolerance = .001
Pass 30% of support vectors
This table shows the different multi-threaded chunking data runs performed on assortments of DNA hairpin pairs. The last line of the table presents the mean of the data runs to gauge the accuracy when using the Absdiff kernel.

Distributed Chunked SMO			**Chunk Size 200 of 800 total feature vectors**			
Data	**Iterations**	**# of SVs**	**SN**	**SP**	**(SN+SP)/2**	**Elapsed Time (ms)**
8GC9AT	8	222	0.97	0.83	0.9	1947
8GC9CG	8	226	0.91	0.89	0.9	1471
8GC9GC	65	205	0.93	0.96	0.945	1412
8GC9TA	28	209	0.84	0.93	0.885	1489
9AT9CG	8	238	0.77	0.65	0.71	1308
9AT9GC	10	228	0.74	0.71	0.725	1342
9AT9TA	10	232	0.9	0.91	0.905	1265
9CG9GC	8	238	0.66	0.85	0.755	1236
9CG9TA	10	222	0.92	0.91	0.915	1232
9GC9TA	12	224	0.92	0.88	0.9	1233
Mean	**16.7**	**224.4**	**0.856**	**0.852**	**0.854**	**1393.5**

Table 2.2
Multi-threaded chunking using different DNA hairpin datasets
SVM Parameters: Sentropic kernel with sigma=.5, C = 10,
Epsilon = .001, Tolerance = .001
Pass 30% of support vectors
This table shows the different multi-threaded chunking data runs performed on assortments of DNA hairpin pairs. The last line of the table presents the mean of the data runs to gauge the accuracy when using the Sentropic kernel.

Distributed Chunked SMO			**Chunk Size 200 of 800 total feature vectors**			
Data	**Iterations**	**# of SVs**	**SN**	**SP**	**(SN+SP)/2**	**Elapsed Time (ms)**
8GC9AT	14	221	0.97	0.89	0.93	2667
8GC9CG	30	202	0.91	0.9	0.905	1993
8GC9GC	27	208	0.91	0.93	0.92	2003
8GC9TA	38	208	0.95	0.88	0.915	2017
9AT9CG	8	232	0.79	0.72	0.755	2531
9AT9GC	21	237	0.71	0.8	0.755	2121
9AT9TA	8	234	0.85	0.87	0.86	2318
9CG9GC	9	237	0.74	0.69	0.715	2132
9CG9TA	8	230	0.84	0.94	0.89	2003
9GC9TA	10	224	0.94	0.87	0.905	1945
Mean	**17.3**	**223.3**	**0.86**	**0.849**	**0.855**	**2173**

Table 2.3
Multi-threaded chunking using different DNA hairpin datasets
SVM Parameters: Gaussian kernel with sigma=.05, C = 10,
Epsilon = .001, Tolerance = .001
Pass 30% of support vectors
This table shows the different multi-threaded chunking data runs performed on assortments of DNA hairpin pairs. The last line of the table presents the mean of the data runs to gauge the accuracy when using the Gaussian kernel.

Distributed Chunked SMO			**Chunk Size 200 of 800 total feature vectors**			
Data	**Iterations**	**# of SVs**	**SN**	**SP**	**(SN+SP)/2**	**Elapsed Time (ms)**
8GC9AT	24	174	0.93	0.83	0.88	2629
8GC9CG	13	170	0.87	0.85	0.86	1732
8GC9GC	59	167	0.95	0.76	0.855	1970
8GC9TA	66	158	0.96	0.8	0.88	1713
9AT9CG	35	192	0.88	0.72	0.8	1490
9AT9GC	34	191	0.79	0.81	0.8	1750
9AT9TA	36	181	0.79	0.87	0.83	1456
9CG9GC	75	192	0.78	0.63	0.705	1912
9CG9TA	38	178	0.82	0.89	0.855	1533
9GC9TA	37	166	0.84	0.89	0.865	1922
Mean	**41.7**	**176.9**	**0.861**	**0.81**	**0.833**	**1810.7**

Table 2.4
Sequential chunking using different DNA hairpin datasets
SVM Parameters: Absdiff kernel with sigma=.5, C = 10,
Epsilon = .001, Tolerance = .001
Pass 100% of support vectors
This table shows the different sequential chunking data runs performed on assortments of DNA hairpin pairs. The last line of the table presents the mean of the data runs to gauge the accuracy when using the Absdiff kernel.

Sequential Chunked SMO			Chunk Size 200 of 800 total feature vectors			
Data	Iterations	# of SVs	SN	SP	(SN+SP)/2	Elapsed Time (ms)
8GC9AT	100	554	0.96	0.95	0.955	12610
8GC9CG	114	557	0.92	0.92	0.92	16901
8GC9GC	58	524	0.94	0.97	0.955	8914
8GC9TA	68	542	0.97	0.95	0.96	10000
9AT9CG	37	727	0.83	0.8	0.815	10936
9AT9GC	23	727	0.83	0.83	0.83	9757
9AT9TA	9	661	0.93	0.93	0.93	7563
9CG9GC	15	751	0.78	0.77	0.775	9218
9CG9TA	41	597	0.92	0.89	0.905	10267
9GC9TA	51	567	0.95	0.92	0.935	9695
Mean	**51.6**	**620.7**	**0.903**	**0.893**	**0.898**	**10586.1**

Table 2.5
Sequential chunking using different DNA hairpin datasets
SVM Parameters: Sentropic kernel with sigma=.5, C = 10,
Epsilon = .001, Tolerance = .001
Pass 100% of support vectors
This table shows the different sequential chunking data runs performed on assortments of DNA hairpin pairs. The last line of the table presents the mean of the data runs to gauge the accuracy when using the Sentropic kernel.

Sequential Chunked SMO			Chunk Size 200 of 800 total feature vectors			
Data	Iterations	# of SVs	SN	SP	(SN+SP)/2	Elapsed Time (ms)
8GC9AT	52	570	0.96	0.95	0.955	14479
8GC9CG	78	545	0.93	0.93	0.93	16844
8GC9GC	22	525	0.92	0.95	0.935	10065
8GC9TA	130	550	0.97	0.95	0.96	18304
9AT9CG	32	722	0.83	0.82	0.825	15273
9AT9GC	44	734	0.81	0.82	0.815	15452
9AT9TA	39	693	0.9	0.9	0.9	14419
9CG9GC	33	747	0.75	0.79	0.77	14855
9CG9TA	73	616	0.89	0.91	0.9	16978
9GC9TA	54	597	0.91	0.93	0.92	14159
Mean	**55.7**	**629.9**	**0.89**	**0.895**	**0.891**	**15082.8**

Table 2.6
Sequential chunking using different DNA hairpin datasets
SVM Parameters: Gaussian kernel with sigma=.05, C = 10,
Epsilon = .001, Tolerance = .001
Pass 100% of support vectors
This table shows the different sequential chunking data runs performed on assortments of DNA hairpin pairs. The last line of the table presents the mean of the data runs to gauge the accuracy when using the Gaussian kernel.

Sequential Chunked SMO			**Chunk Size 200 of 800 total feature vectors**			
Data	**Iterations**	**# of SVs**	**SN**	**SP**	**(SN+SP)/2**	**Elapsed Time (ms)**
8GC9AT	97	434	0.93	0.88	0.905	14978
8GC9CG	83	395	0.88	0.85	0.865	15653
8GC9GC	134	396	0.94	0.88	0.91	17783
8GC9TA	114	396	0.96	0.82	0.89	15988
9AT9CG	62	503	0.79	0.82	0.805	20789
9AT9GC	89	488	0.81	0.85	0.83	20833
9AT9TA	111	477	0.91	0.89	0.9	18057
9CG9GC	60	523	0.91	0.54	0.725	20233
9CG9TA	78	436	0.88	0.9	0.89	17910
9GC9TA	89	409	0.9	0.94	0.92	12422
Mean	**91.7**	**445.7**	**0.891**	**0.84**	**0.864**	**17464.6**

Table 2.7
Dataset = 9GC9CG_9AT9TA (1600 feature vectors)
SVM Parameters: Absdiff kernel with sigma=.5, C = 10,
Epsilon = .001, Tolerance = .001
Pass 100% of support vectors and 50% of polarization set
Final Chunk Performance: {SN, SP} = {.87, .84}
This table shows the sequential chunking method focusing on the chunk sizes during the data run. The Absdiff kernel chunk performance is represented here. The breakdown of each feature vector set is displayed to show how the percentage parameters are used to pass portions of each set to the next chunk.

	Chunk 1	**Chunk 2**	**Chunk 3**	**Chunk 4**
Total Chunk Size	400	787	1143	1472
Support Vectors	373	700	1002	1320
Polarization Set	27	86	140	152
Penalty Set	0	0	0	0
Violator Set	0	1	1	0
Support Vectors Passed	373	700	1002	
Polarization Set Passed	14	43	70	
Total Passed Set	387	743	1072	

Table 2.8
Dataset = 9GC9CG_9AT9TA (1600 feature vectors)
SVM Parameters: Sentropic kernel with sigma=.5, C = 10,
Epsilon = .001, Tolerance = .001
Pass 100% of support vectors and 50% of polarization set
Final Chunk Performance: {SN, SP} = {.875, .82}
This table shows the sequential chunking method focusing on the chunk sizes during the data run. The Sentropic kernel chunk performance is represented here. The breakdown of each feature vector set is displayed to show how the percentage parameters are used to pass portions of each set to the next chunk.

	Chunk 1	Chunk 2	Chunk 3	Chunk 4
Total Chunk Size	400	792	1150	1481
Support Vectors	383	707	1011	1320
Polarization Set	17	85	139	160
Penalty Set	0	0	0	0
Violator Set	0	0	0	1
Support Vectors Passed	383	707	1011	
Polarization Set Passed	9	43	70	
Total Passed Set	392	750	1081	

Table 2.9
Dataset = 9GC9CG_9AT9TA (1600 feature vectors)
SVM Parameters: Gaussian kernel with sigma=.05, C = 10,
Epsilon = .001, Tolerance = .001
Pass 100% of support vectors and 50% of polarization set
Final Chunk Performance: {SN, SP} = {.715, .85}
This table shows the sequential chunking method focusing on the chunk sizes during the data run. The Gaussian kernel chunk performance is represented here. The breakdown of each feature vector set is displayed to show how the percentage parameters are used to pass portions of each set to the next chunk.

	Chunk 1	Chunk 2	Chunk 3	Chunk 4
Total Chunk Size	400	754	1036	1264
Support Vectors	309	521	697	881
Polarization Set	90	229	334	372
Penalty Set	1	4	4	11
Violator Set	0	0	1	0
Support Vectors Passed	309	521	697	
Polarization Set Passed	45	115	167	
Total Passed Set	354	636	864	

Table 2.10
Dataset = 9GC9CG_9AT9TA (1600 feature vectors)
SVM Parameters: Absdiff kernel with sigma=.5, C = 10,
Epsilon = .001, Tolerance = .001
Pass 80% of support vectors and 60% of polarization set
Final Chunk Performance: {SN, SP} = {.855, .795}
This table shows the multi-threaded chunking method focusing on the chunk sizes during the data run. The Absdiff kernel chunk performance is represented here. The breakdown of each feature vector set is displayed to show how the percentage parameters are used to pass portions of each set to the next set of chunks.

	C1	C2	C3	C4	C5	C6	C7	C8	C9	C10
Total Chunk Size	400	400	400	400	423	423	425	504	504	791
Support Vectors	373	377	378	388	402	402	403	466	460	699
Polarization Set	27	23	22	12	21	21	22	38	43	92
Penalty Set	0	0	0	0	0	0	0	0	0	0
Violator Set	0	0	0	0	0	0	0	0	1	0
Support Vectors Passed	1218	-	-	-	968	-	-	742	-	-
Polarization Set Passed	53	-	-	-	40	-	-	49	-	-
Total Passed Set	1271	-	-	-	1008	-	-	791	-	-

Table 2.11
Dataset = 9GC9CG_9AT9TA (1600 feature vectors)
SVM Parameters: Sentropic kernel with sigma=.5, C = 10,
Epsilon = .001, Tolerance = .001
Pass 80% of support vectors and 60% of polarization set
Final Chunk Performance: {SN, SP} = {.845, .755}
This table shows the multi-threaded chunking method focusing on the chunk sizes during the data run. The Sentropic kernel chunk performance is represented here. The breakdown of each feature vector set is displayed to show how the percentage parameters are used to pass portions of each set to the next set of chunks.

	C1	C2	C3	C4	C5	C6	C7	C8	C9	C10
Total Chunk Size	400	400	400	400	423	423	425	503	504	787
Support Vectors	383	380	383	379	409	396	442	465	446	666
Polarization Set	17	17	17	21	21	14	27	38	56	121
Penalty Set	0	0	0	0	0	0	0	0	0	0
Violator Set	0	1	0	0	0	0	1	0	2	0
Support Vectors Passed	1224	-	-	-	966	-	-	730	-	-
Polarization Set Passed	47	-	-	-	41	-	-	57	-	-
Total Passed Set	1271	-	-	-	1007	-	-	787	-	-

Table 2.12
Dataset = 9GC9CG_9AT9TA (1600 feature vectors)
SVM Parameters: Gaussian kernel with sigma=.05, C = 10,
Epsilon = .001, Tolerance = .001
Pass 80% of support vectors and 60% of polarization set
Final Chunk Performance: {SN, SP} = {.85, .83}
This table shows the multi-threaded chunking method focusing on the chunk sizes during the data run. The Gaussian kernel chunk performance is represented here. The breakdown of each feature vector set is displayed to show how the percentage parameters are used to pass portions of each set to the next set of chunks.

	C1	C2	C3	C4	C5	C6	C7	C8	C9	C10
Total Chunk Size	400	400	400	400	401	401	403	458	458	690
Support Vectors	291	309	316	305	318	320	313	341	354	495
Polarization Set	108	90	83	93	83	81	88	116	103	194
Penalty Set	1	1	1	0	0	0	2	1	0	1
Violator Set	0	0	0	2	0	0	0	0	1	0
Support Vectors Passed	980	-	-	-	764	-	-	558	-	-
Polarization Set Passed	225	-	-	-	152	-	-	132	-	-
Total Passed Set	1205	-	-	-	913	-	-	690	-	-

Chapter 3. Support Vector Reduction

3.1 Methods

Support Vector Reduction (SVR) is a process that is run right after the SVM learning step is complete. Instead of going on to testing data against the training results to get accuracy, we further reduce the support vector set. One way to do this is to coerce some alphas to zero which means they would now fall into the polarization set and further away from the hyperplane. Converting the smaller alphas to zeros makes the most sense since a larger alpha indicates that the data point is stronger towards its grouping (polarized sign). This is done using a user-defined alpha cut off value. All alpha values that are under the cut off are pushed to zero. It is not entirely trivial since certain mathematical constraints must be met. The constraint that must be met for this method is the linear equality constraint [2]:

$$\sum_{i=1}^{N} y_i \alpha_i = 0$$

Therefore, the alpha values not meeting the cutoff cannot just be forced to zero unless the value is retained somewhere else in the set. This is done by first sorting the alpha values of the support vectors. Then for each alpha that does not meet the cut off value, the small left over value is added to the largest alpha of the same polarity. Since the list is sorted it can loop through and evenly distribute the left over values through the larger alphas starting with the largest. The reduction process can cut the support vector count down without significantly affecting the accuracy. Other observations have shown that the easier the dataset to classify, the larger the reduction.

3.2 SVR Results

Figure 3.1 shows the results of the SVR method on the non-chunking SMO SVM. For this dataset, 0.19 seems like the best cut off value to use for future data runs since it retains the accuracy while reducing the support vectors. As shown in Figure 3.1, the total run time decreases as support vectors are reduced. This is due to a decrease in testing time since there are not as many support vectors to test against. For the 9GC9CG_9AT9TA dataset, 140 support vectors (10.5% of total) were dropped without affecting the accuracy.

Data runs using sequential (Figure 3.3) and multi-threaded (Figure 3.2) chunking methods with SVR show similar results. The chunking results tend to be a bit choppier since the SVM algorithm makes some approximations thus the hyperplane will not be exactly the same for every data run and this behavior is amplified in the chunking methods. Nonetheless, the trend lines show that using the SVR method definitely cuts down on support vectors and decreases testing time. For sequential chunking (Figure 3.3), an alpha cut-off value of 0.25 caused 87 support vectors (7.2%) to be dropped without affecting accuracy. For multi-threaded chunking (Figure 3.2), an alpha cut-off value of 0.22 dropped 26 support vectors (6.2%) while retaining accuracy.

Figure 3.1
SMO (non-chunking) Support Vector Reduction
Dataset: 9GC9CG_9AT9TA (1600 feature vectors)
SVM Parameters: Absdiff kernel with sigma=.5, C = 10,
Epsilon = .001, Tolerance = .001
This graph shows the rate of support vectors reduced as the alpha cutoff value is increased. The alpha cutoff value 0.19 is chosen as the best since it is the last value before accuracy begins to degrade. This chosen value reduces 140 support vectors.

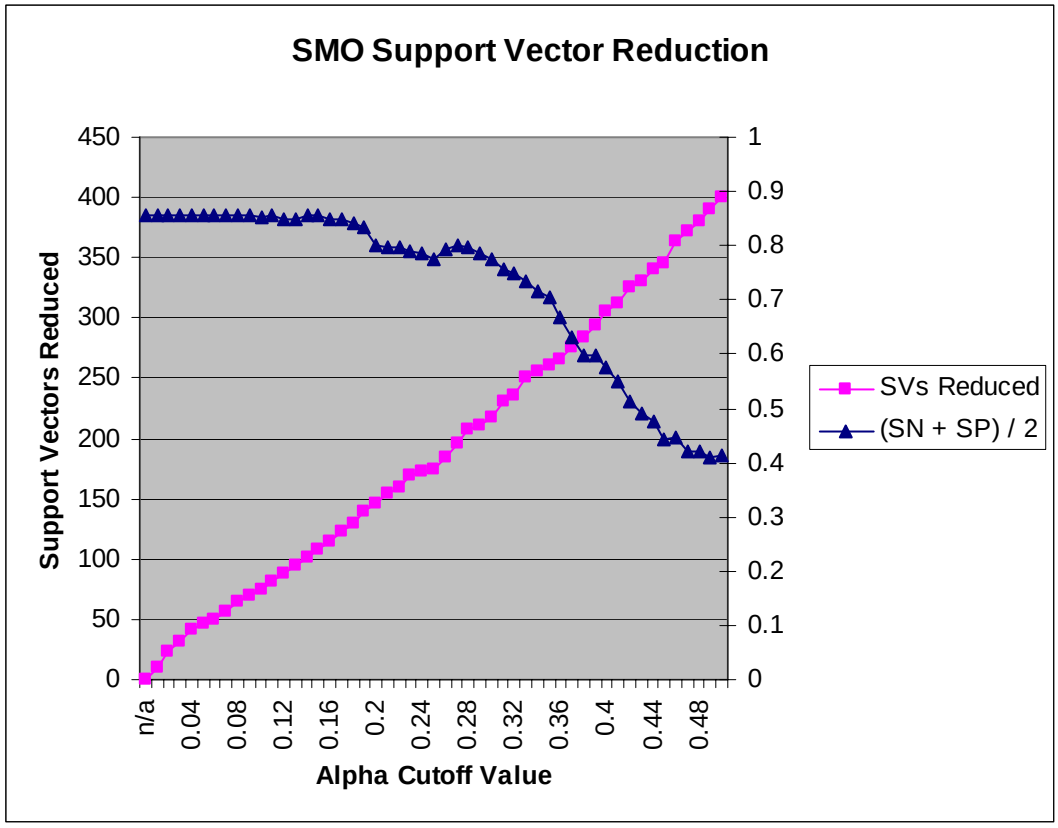


Figure 3.2
Multi-threaded Chunking Support Vector Reduction
Dataset: 9GC9CG_9AT9TA (1600 feature vectors), Starting chunk size=400
SVM Parameters: Absdiff kernel with sigma=.5, C = 10,
Epsilon = .001, Tolerance = .001
Passing 30% of Support Vectors
This graph shows the rate of support vectors reduced as the alpha cutoff value is increased. The alpha cutoff value 0.22 is chosen as the best since it is the last value before accuracy begins to degrade. This chosen value reduces 26 support vectors.

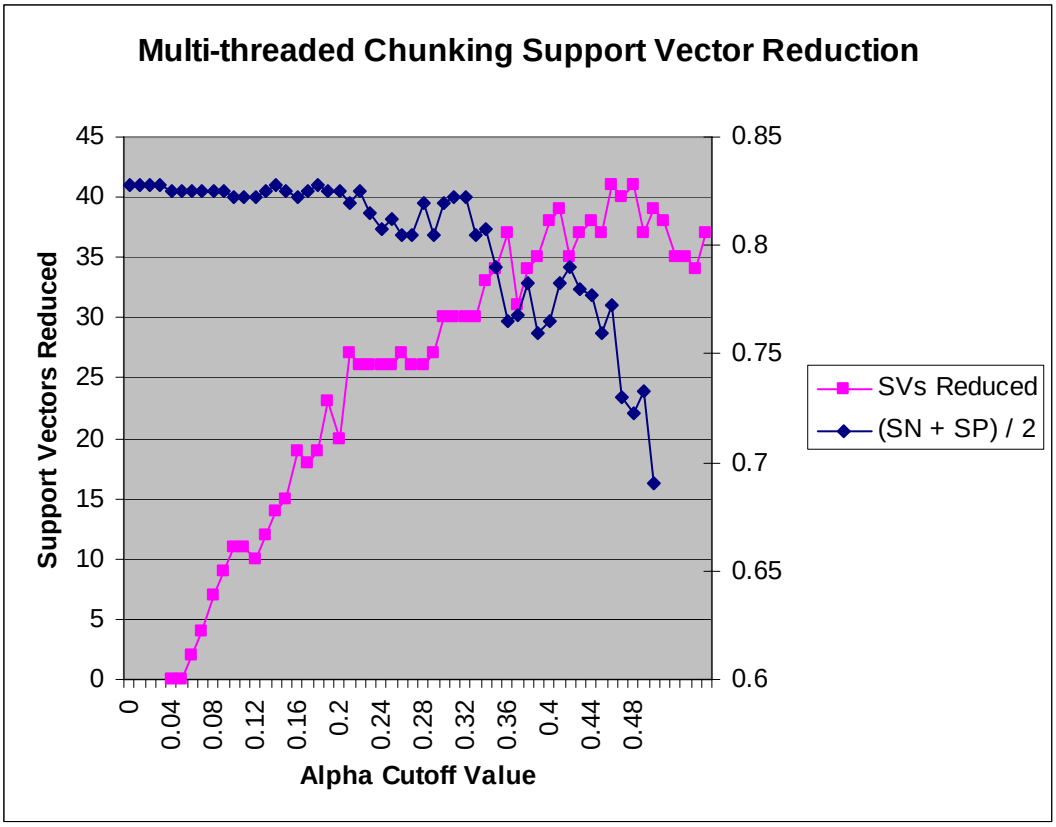
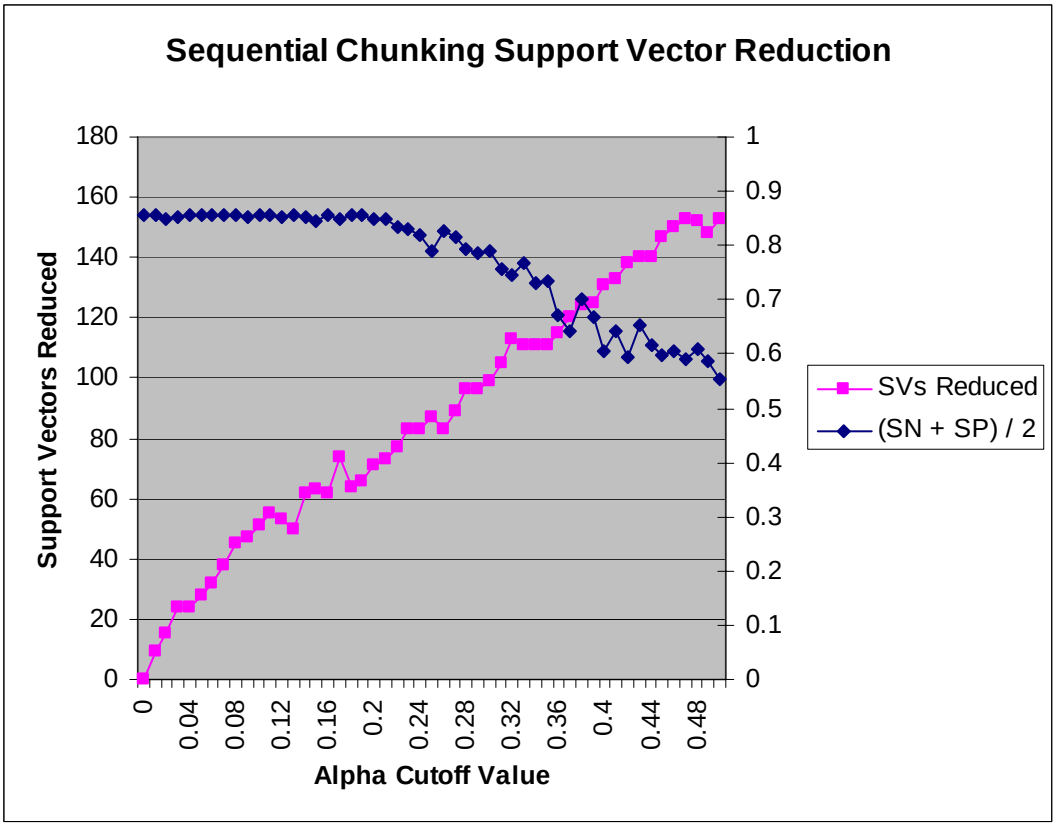


Figure 3.3
Sequential Chunking Support Vector Reduction
Dataset: 9GC9CG_9AT9TA (1600 feature vectors), Starting chunk size=400
SVM Parameters: Absdiff kernel with sigma=.5, C = 10,
Epsilon = .001, Tolerance = .001
Passing 100% of Support Vectors
This graph shows the rate of support vectors reduced as the alpha cutoff value is increased. The alpha cutoff value 0.25 is chosen as the best since it is the last value before accuracy begins to degrade. This chosen value reduces 87 support vectors.



Chapter 4. SVM Chunking

SVM chunking provides an alternative method to running a typical SVM (SMO) on a dataset by instead breaking up the training data and running the SVM on smaller chunks of data. In the chunking process, feature vectors associated with strong data points are retained from chunk to chunk, while weak data points are discarded. Chunking becomes a necessity when classifying large datasets. In this context, a large dataset refers to one that has over 5,000 features vectors where each vector has 150 components. Since the order of computations increases by the square of the size of the training set, most PCs would not have enough memory to support a kernel matrix of 10,000 or more training instances.

Initially, the training data is shuffled then broken into chunks. The number and size of the chunks depends on the size of the dataset to be trained. In the Java implementation of this algorithm, the user specifies the size of each chunk and the chunks are broken up accordingly. If the chunks don't divide evenly, which is the case most of the time, the few remaining feature vectors are added to the last chunk. When training on the chunk is complete, the resulting trained feature vectors each fit into a separate set. If the SVM classifies well, the largest set consists of the support feature vectors. The KKT violators make up another set. KKT violators refer to feature vectors that violate one of the KKT relations. The violator set is usually zero at the end of the training process, unless some minimal number of violators is allowed upon learning completion. The polarization set consists of the feature vectors that have been classified as a positive or negative one. These are feature vectors that pass the KKT relations and have an alpha coefficient equal to zero. The penalty set consists of the feature vectors which pass the KKT relations and have alpha coefficients equal to C (the max value). These sets give the user a choice of which kind of feature vectors they want to pass to the next chunk(s). To keep the SVM

converging to a better solution, on the next chunk run, several support feature vectors and sometimes some of the polarization set are passed to the next chunk(s). The percentages of each feature vector set depend on which kernel is used and the dataset.

There are different methods of extracting the feature vectors from the different sets. The specified percentages of feature vectors are pseudo-randomly chosen from each of the sets except for one. The support feature vectors extraction method differs since it extracts the feature vectors that have the best scores. Each score represents the distance of the feature vector from the hyperplane. It makes sense to choose feature vectors whose scores are closer to the hyperplane in order to pass a tighter hyperplane on to the next chunk(s). Passing a more precise hyperplane should speed up the next SVM run.

4.1 Previous SVM chunking methods

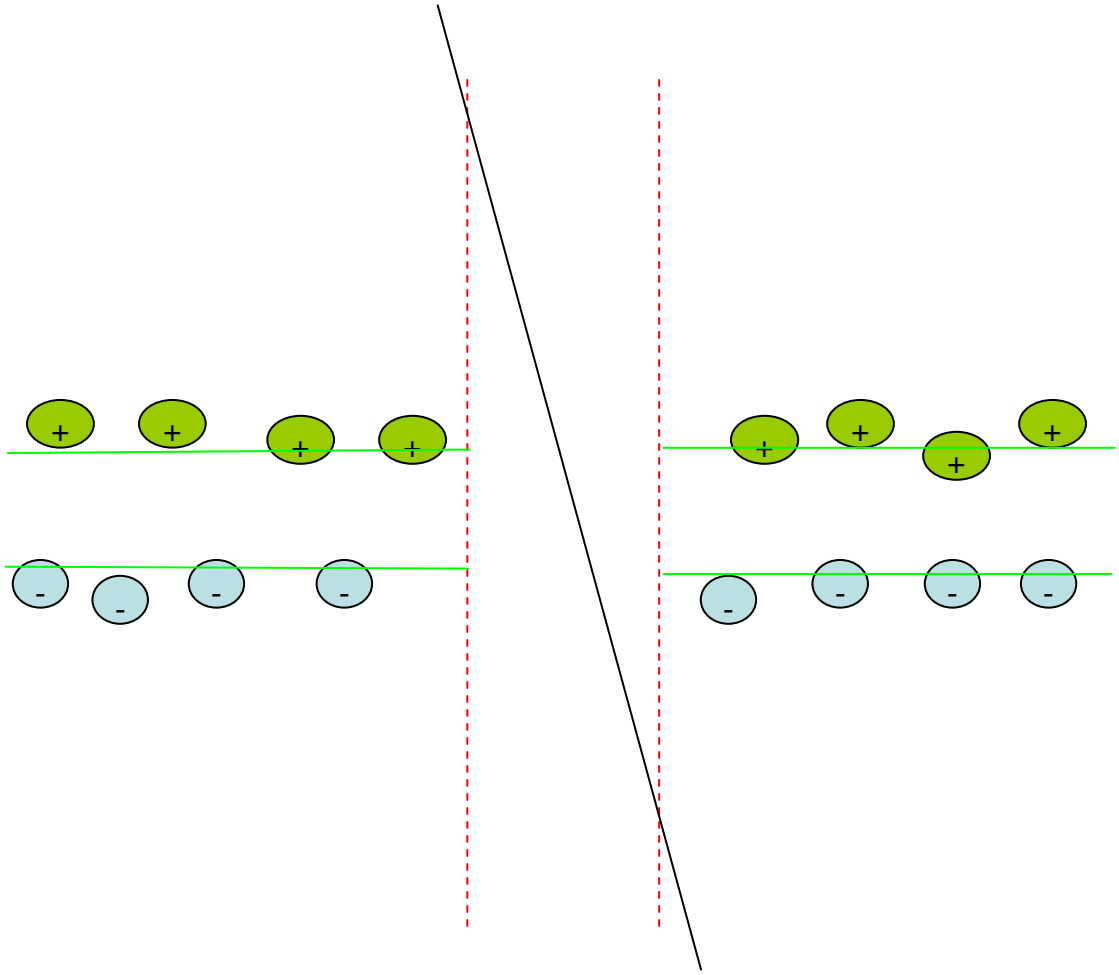
Zanghirati and Zanni [6] developed the variable projection method (VPM) for training SVMs in parallel. This method is based off of Joachim's SVM light decomposition techniques [7] which delve further into the inner workings of the SMO algorithm [2]. First, the feature vector indices are divided into two categories, the free and fixed sets based upon their alphas (Langrange Multipliers). The free set represents the KKT violators which need to be further optimized while the fixed set is the alphas that already fulfill the KKT equations. An alpha variable from each set is used to solve each quadratic sub problem in order to optimize the free set alphas until convergence. Though this sounds similar to Platt's SMO [2], Joachim performs some additional tricks to cut down on the number of iterations needed to converge [7]. VPM provides a parallel solution to computing the kernel matrix which is the most memory intensive part of the SVM. The kernel calculations are spread among several processing elements and the rows of the matrix are spread and usually duplicated across the memory of those processing

elements. Since the rows are duplicated, they must be synchronized after each local computation. VPM is implemented using standard C and MPI communication routines.

Hans Peter Graf et al. [9] developed the Cascade SVM to parallelize SVMs. This method begins by breaking the large dataset into chunks. The SVM is run on each separate chunk in the first layer. When the SVMs have all converged, new chunks are created from the resulting support vectors from the pairs of first layer chunks which make up the second layer of chunks. This occurs until a final chunk is reached. The final set of support vectors is then fed back into each first layer chunk. If further optimization is possible and needed, the entire process is rerun until the global optimum is met. If the global optimum is not needed due to decent initial training results, the process can be halted after one run of the network of chunks. Allowing the Cascading SVM to continue running will eventually produce the global optimum. This method seems intuitive but after testing, we have found that passing 100% of support vectors down to the next set of chunks without also passing some non support vectors or using the SVR method does not work properly with the DNA hairpin data used here. The data run never finishes, in fact, since it cannot further reduce the support vectors to converge to the final chunk. This weakness of the method, not apparent at first sight or mentioned in [9], may be understood if the SVs from different chunks are sufficiently different and training on SVs from individual chunks are themselves separated at the chunk groups (Illustration 4.1).

As shown in the illustration, the actual margins for each chunk (between the two green lines) are not recognized since another significant margin exists. The SVM continually works on trying to fit the hyperplane between the intercalating chunk margin (between the red dashed lines) and never finishes running.

Illustration 4.1
100% SV Passing for multi-threaded chunking without SVR



The chunking methods presented here have some similarities to the Cascade SVM. As discussed above, the large dataset is broken into smaller chunks and the SVM is run on each separate chunk. Instead of bringing the results of paired chunks together, all chunk results are brought together and re-chunked as occurred in the first layer. This process occurs until the final

chunk is calculated which gives the trained result. Since SVM parameters vary for different datasets, the user has the option to tune the percentage of support vectors and non support vectors to pass to the next set of chunks. Additionally, passed support vectors are chosen wisely to produce a tighter hyperplane to better distinguish the polarization sets. Another aspect not covered in the Cascade method is the SVR method. This method runs as part of the core SVM learning task on each chunk. It uses a user-defined alpha cutoff value for further tuning and can significantly reduce the number of support vectors passed to the next set of chunks. These additional steps reduce the size of the chunks thus making the algorithm run faster without loss of accuracy. Details of these steps are discussed in more depth in chapter 3.

4.2 Sequential Chunking (Linear Topology)

Sequential chunking is one form of chunking which is not multi-threaded. This method runs the SVM on the first chunk, and then sends the support feature vectors (SVs) and sometimes non-SVs to be added onto the training data for the next chunk. This continues until the final chunk has been run. When using sequential chunking, feature vector passing can be difficult since passing too many features on to the next chunk can result in large datasets in the later chunks in the process. Support feature vectors are the most valuable to pass to the next chunk since they define the hyperplane.

4.3 Sequential Chunking Results

For the DNA hairpin datasets used here, results have shown that the ideal chunking parameter for sequential chunking is 100% of the support vector set. This produced the best accuracy (0.855) within stable conditions (Figure 4.1). Illustration 4.2 displays a sample run and the size of each chunk as the algorithm progresses through the chunks. Table 2.7 shows the feature vector set composition of each chunk corresponding to the data run for Illustration 4.2.

Illustration 4.2 Linear Topology Chunk Progression

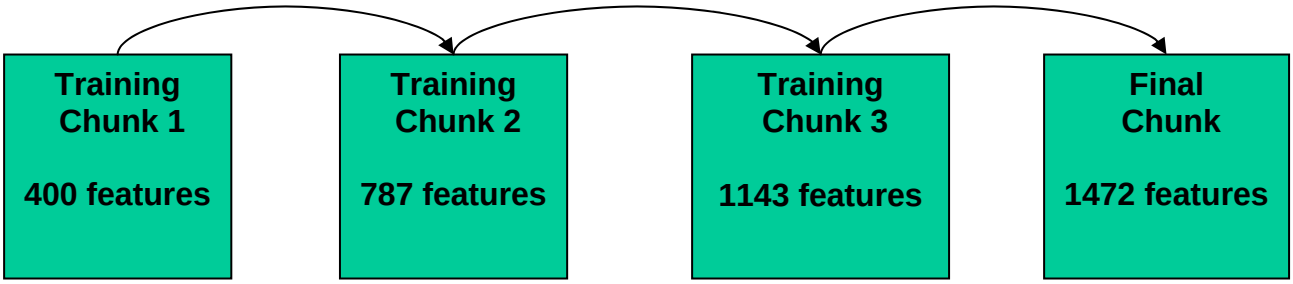
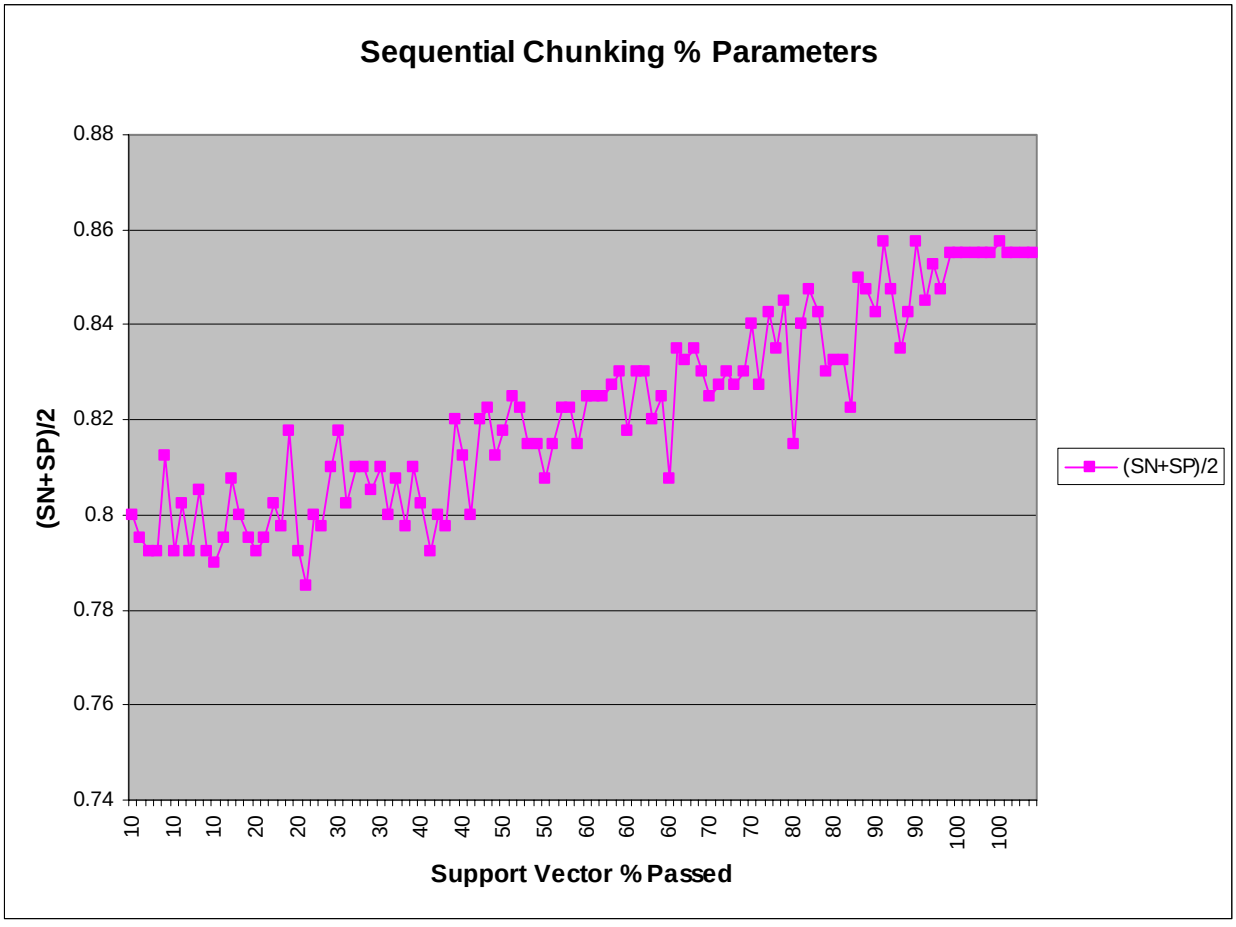


Figure 4.1
Dataset = 9GC9CG_9AT9TA (1600 feature vectors)
SVM Parameters: Absdiff kernel with sigma=.5, C = 10, Epsilon = .001, Tolerance = .001
This shows the trend for sequential chunking when using different support vector and polarization set percentage parameters. Every variation of multiples of ten up to 100 was used for each of the two sets. For example, when the SV % parameter was 10, the polarization set % parameter would vary from 0 to 100 in multiples of ten. For most of the data run, especially the more stable part at 100 % SVs, the variation of the polarization set did not seem to have much effect on the outcome.



4.4 Multi-threaded Chunking (Binary Tree Topology)

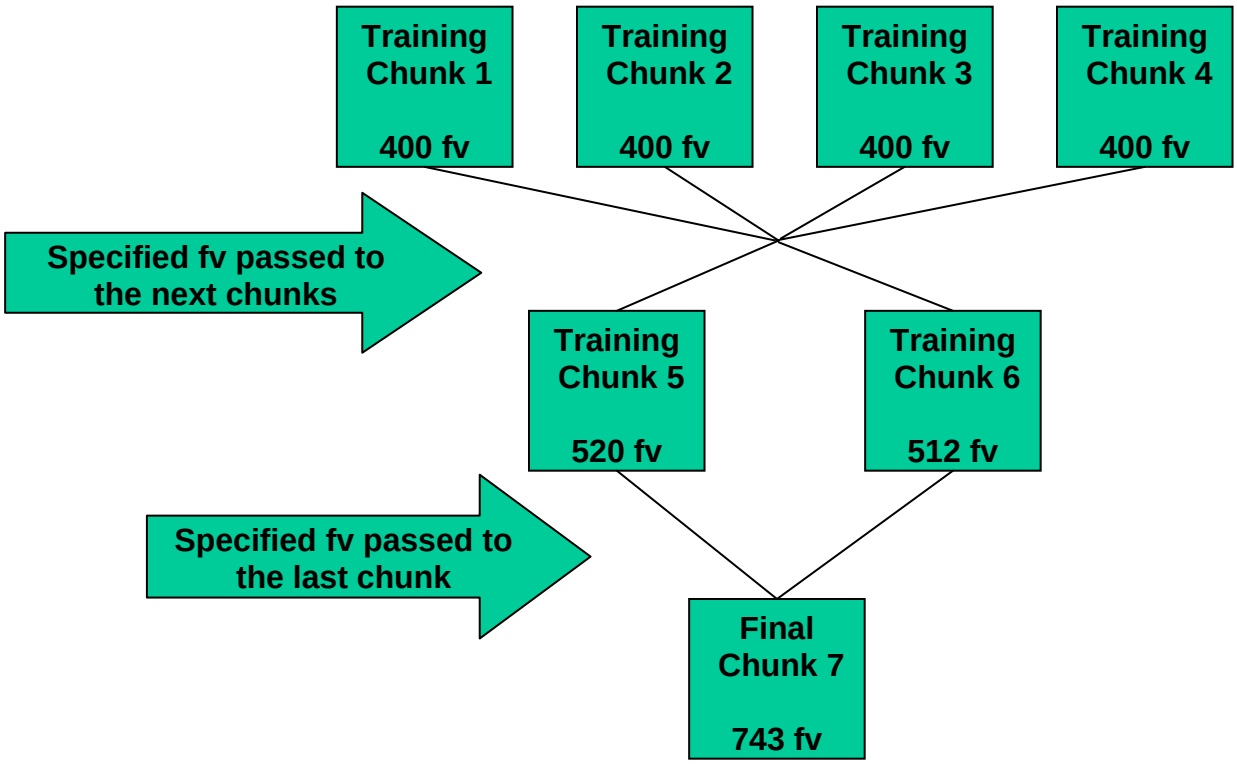
The multi-threaded chunking method simultaneously runs the chunks using multiple threads. Once all of the threaded chunks are finished training, the chunk results are collected into an array. The same user defined percentages of feature vector sets are used here except this time those percentages of feature vectors are extracted from each chunk. All of the chosen feature vectors to be passed are stored together then re-chunked if the current data set is large enough to be chunked again. Re-chunking occurs when the data set is greater than or equal to twice the specified chunk size. If this is not the case, the final chunk is run alone to get the final result. The main use of the multi-threaded chunking method is with a single computer with multiple processors/cores.

Observations have shown that sending 100% of the support vectors to the next chunk generally causes the chunking to run continuously without ever ending with a result. At least, on these challenging datasets, this is an expected behavior since passing all support vector data to the next chunk level would just be re-chunking the same support vectors that were already done the first time. Some data must be dropped to converge to the final chunk whether it is through the chunking parameters or the SVR method. Dropping too much data can affect the final accuracy while retaining too much data increases training time. Therefore the chunking percentage parameters must be properly tuned in order to get the best results for each dataset.

4.5 Multi-threaded Chunking Results

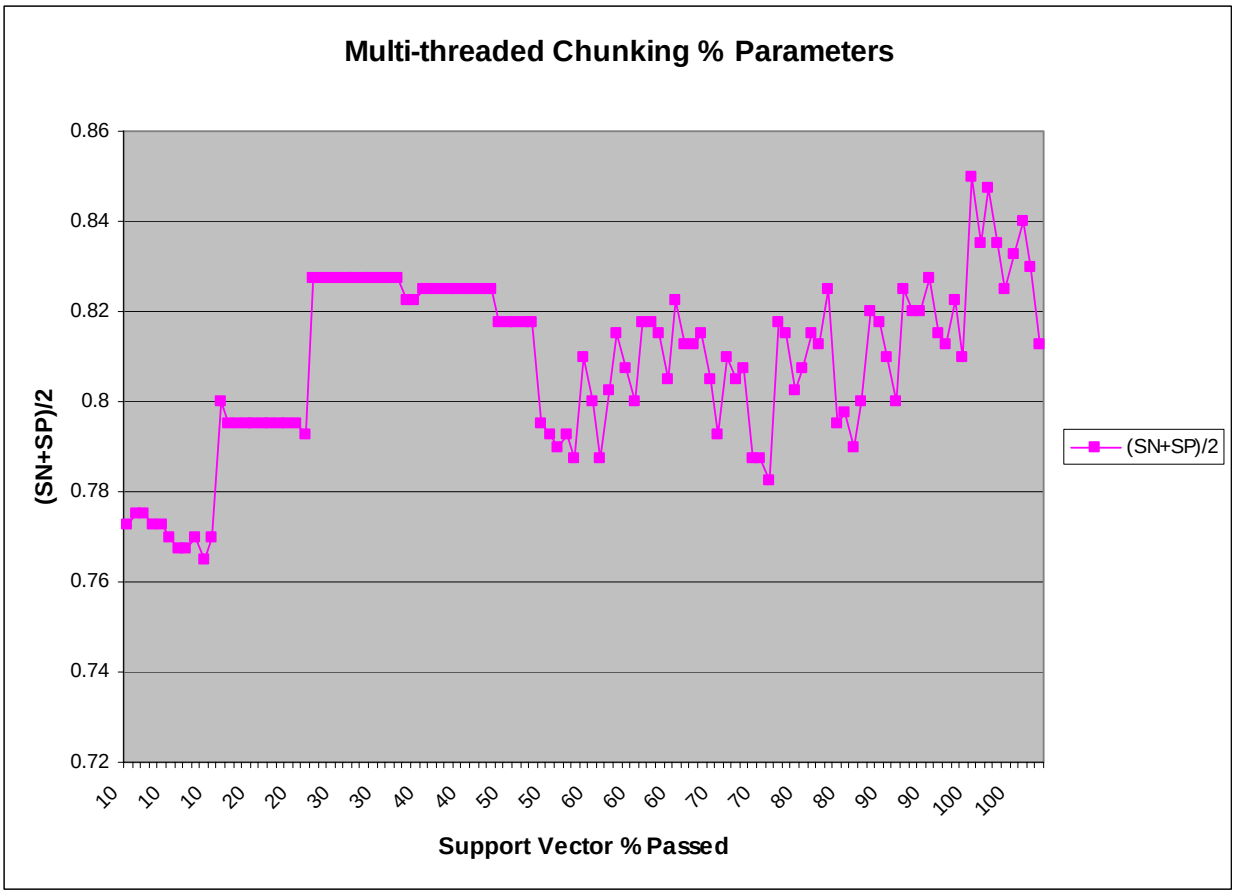
The best accuracy result (0.83) within stable conditions for multi-threaded chunking were obtained using 30% of the support vectors set as shown in Figure 4.3. Illustration 4.3 displays a sample multi-threaded chunking run and the size of each chunk as the algorithm progresses through the sets of chunks.

Illustration 4.3 Binary Tree Topology Chunk Progression
The numbers in each chunk represent the number of feature vectors (fv).



speed up of $\frac{N}{\ln(N)}$

Figure 4.2
Dataset = 9GC9CG_9AT9TA (1600 feature vectors)
SVM Parameters: Absdiff kernel with sigma=.5, C = 10, Epsilon = .001, Tolerance = .001
This shows the trend for multi-threaded chunking when using different support vector and polarization set percentage parameters. Every variation of multiples of ten up to 100 was used for each of the two sets. For example, when the SV % parameter was 10, the polarization set % parameter would vary from 0 to 100 in multiples of ten. For most of the data run, especially the more stable part at 30 % SVs, the variation of the polarization set did not have much effect on the outcome.



4.6 Multi-threaded Distributed Chunking

Multi-threaded distributed chunking is a multi-server/multi-CPU (core) approach to the previous multi-threaded chunking method. Java RMI is used to handle the remote calls between the client and servers. The client program runs multi-threaded remote calls to a user specified set of servers (round robin). Each server and the client machine have an SVM Server listening.

When the client program runs, a chunk is passed to each available processor/core in the network until all or as many as possible are training simultaneously. As the chunks finish, the results are passed back to the client. Each "chunk level" may take multiple batches depending on the chunk size and amount of processors/cores available. The final chunk is largest so the client program should be processed on the machine with the most computing power. This not only speeds up the final chunk but allowing larger chunks should produce better final results. The main benefit of this method is a significant decrease in run time for large datasets. As shown below in Table 4.1, multi-threaded distributed chunking performs well when it comes to run time. Network overhead causes it to be slightly slower than the non-remote multi-threaded chunking method. With extremely large datasets (i.e. 60,000 feature vectors), the Remote method would be faster.

Table 4.1
Dataset = 9GC9CG_9AT9TA (1600 feature vectors)
SVM Parameters: Absdiff kernel with sigma=.5, C = 10, Epsilon = .001, Tolerance = .001
For chunking methods: Pass 90% of support vectors,
Starting chunk size = 400, maxChunks = 2
For SV Reduction methods: Alpha cut off value = .15
This table shows the overall performance of the different SVM methods. The distributed chunking had three identical networked machines (see Appendix for details).

SVM Method	Sensitivity	Specificity	(SN + SP) / 2	Total Time (ms)
SMO (non-chunked)	0.87	0.84	0.86	47708
Sequential Chunking	0.84	0.86	0.85	27515
Multi-threaded Chunking	0.88	0.78	0.83	7855
SMO (non-chunked) with SV Reduction	0.91	0.81	0.86	43662
Sequential Chunking with SV Reduction	0.90	0.82	0.86	18479
Multi-threaded Chunking with SV Reduction	0.85	0.83	0.84	5232
Multi-threaded Distributed Chunking with SV Reduction	0.85	0.83	0.84	5973

Chapter 5. Conclusion

Support Vector Machines are extremely useful for classifying data and therefore dominating over other methods in a variety of fields and applications. Since the main weakness of SVMs is the long training time when running large datasets, it is only natural that one would develop multi-threaded distributed methods especially since many typical computers today have multiple cores/processors, each with a continuously growing capacity for RAM.

An overall comparison of the SVM methods explained here can be found in Table 4.1 (above). Sequential chunking has the benefit of holding onto accuracy when compared to running the straight SVM (SMO) but the run times can be higher since the method does not run in parallel. Though this performance hit is significantly countered when using the SVR method. Another benefit of sequential chunking is that it helps cut down on high memory usage for the kernel matrix build which can make the difference since the straight SVM may bog down from hard drive thrashing if the machine does not have enough memory. The only negative aspect is that chunk size can still eventually grow too much for extremely large datasets so the chunking percentage parameters must be adjusted accordingly.

Multi-threaded chunking has a significant run time performance increase which is further improved when employing the SVR method. The multi-threaded aspect allows training of extremely large datasets which may not be possible using sequential chunking. Additionally, using the multi-threaded distributed method allows users to add several more machines to make the algorithm train even faster. This aspect makes the size of the dataset no longer a concern for SVM training, which opens up the practical use of SVM methods to more applications than originally envisaged (e.g. SVM-based clustering).

References

[1] Winters-Hilt, S., Yelunder, A., McChesney, C., and Landry, M., "Support Vector Machine Implementations for Classification & Clustering", BMC Bioinformatics, (2006).

[2] Platt, J.C., "Sequential Minimal Optimization: A Fast Algorithm for Training Support Vector Machines", Microsoft Research, Technical Report MSR-TR-98-14, (1998).

[3] Osuna, E., Freund, R., and Girosi, F., "An Improved Training Algorithm for Support Vector Machines", Proc. IEEE NNSP '97, (1997).

[4] Vapnik, V., Estimation of Dependencies Based on Empirical Data, Springer-Verlag, (1982).

[5] Cortes, C., Vapnik, V., "Support Vector Networks", Machine Learning, 20:273-297, (1995).

[6] Zanghirati, G., Zanni L., "A parallel solver for large quadratic programs in training support vector machines", Parallel Computing, Vol. 29, pp.535-551, 2003.

[7] Joachims, T., Making large-scale SVM learning practical, in: B. Sch€lkopf, C.J.C. Burges, A. Smola (Eds.), Advances in Kernel Methods—Support Vector Learning, MIT Press, Cambridge, MA, 1998.

[8] Platt, J.C., Fast training of support vector machines using sequential minimal optimization, in: B. Sch€lkopf, C. Burges, A. Smola (Eds.), Advances in Kernel Methods—Support Vector Learning, MIT Press, Cambridge, MA, 1998.

[9] Graf, H.P., Cosatto, E., Bottou, L., Durdanovic, I., Vapnik, V., Parallel Support Vector Machines: The Cascade SVM, in proceedings NIPS, 2004

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

[11] Joachims, T., "Text Categorization with Support Vector Machines", LS VIII Technical Report, No. 23, University of Dortmund, (1997).

[12] Osuna, E., Freund, R., Girosi, F., "Training Support Vector Machines: An Application to Face Detection," *Proc. Computer Vision and Pattern Recognition '97*, 130-136, (1997).

[13] LeCun, Y., Jackel, L. D., Bottou, L., Cortes, C., Denker, J. S., Drucker, H., Guyon, I., Muller, U. A., Sackinger, E., Simard, P. and Vapnik, V., "Learning Algorithms for Classification: A Comparison on Handwritten Digit Recognition," *Neural Networks: The Statistical Mechanics Perspective*, Oh, J. H., Kwon, C. and Cho, S. (Ed.), World Scientific, 261-276, (1995).

Appendix

Note 1: All data sets used in this paper were recorded in the Winters-Hilt Children's Hospital laboratory using a nanophore detector. This raw data was then fed through a Hidden Markov Model which produced the feature vectors.

Note 2: All data runs (except for Table 4.1) were done on a (PC) machine with Debian Linux containing dual AMD Athlon MP 2400 (2 ghz.) processors with 2 gigabytes of RAM.

Note 3: The Table 4.1 data runs were done using three Sun machines with Ubuntu Linux each containing a quad processor, dual core AMD Opteron 280 (1 ghz.) processor with 8 gigabytes of RAM.

A.1 Java code implementation of sequential chunking method

SequentialChunksSVM.java

```
package edu.uno.cs.bioinformatics.svm.chunking;

import cern.colt.list.IntArrayList;
import edu.uno.cs.bioinformatics.data.ModelParameters;
import edu.uno.cs.bioinformatics.data.TrainingData;
import edu.uno.cs.bioinformatics.svm.SVMModel;
/**
 * The SequentialChunkTrial breaks up the training data and runs the SVM on
the chunks.
 * The chunks are trained in sequence in order to pass support features to
the next chunk.
 */
public class SequentialChunksSVM extends AbstractChunksSVM {

  static final long serialVersionUID = 25123L;

  private ChunkParameters chunkPar;
  private TrainingData trainingData;
  private ModelParameters param;
  private SVMModel model;
  private ChunkResult chunkRes;
  private SVMchunkLearner svmEngine;
  private SVMchunkLearner newEngine;

    /**
     * The feature vector indices that are
     */
  private IntArrayList penaltySet = new IntArrayList();

    /**
     * The feature vector indices that are
```

```
     */
  private IntArrayList polarizationSet = new IntArrayList();

    /**
     * The feature vector indices that are KKT violators.
     */
  private IntArrayList kktViolators = new IntArrayList();

  /**
   * The support feature vector indices.
   */
  private IntArrayList svList = new IntArrayList();

  /**
   * Save the first chunk to use for testing in order to verify the integrity
of the chunking process.
   */
  protected TrainingData firstChunk;

  /**
   * Constructs a new sequential chunk trial with the given SVMLearner,
TrainingData, and Parameters.
   * @param maxSize    the maximum size of each chunk
   * @param trainingData  the feature vectors and labels to train on
   * @param param      the user defined parameters object
   */
  public SequentialChunksSVM(SVMchunkLearner svmEngine, ChunkParameters
chunkPar, TrainingData trainingData, ModelParameters param) {
    this.svmEngine = svmEngine;
    this.chunkPar = chunkPar;
    this.trainingData = trainingData;
    this.param = param;
  }

  /**
   * Default Constructor
   */
  public SequentialChunksSVM() {}

  /**
   * The runSeqChunkedSVM method runs the SVM chunks sequentially.
   * @return  the results model
   */
  public SVMModel runSeqChunkedSVM() {

    int featsCount = trainingData.getLabels().size();
    int chunkCount = (int)Math.ceil((featsCount*1.)/(chunkPar.chunkSize*1.));
    TrainingData[] data = makeDataChunks(chunkCount);
    this.firstChunk = data[0];
    String resCapture = "Chunk Number\tChunk Size";
    int chunkTally = 0;

    //traverse through the chunks
    for (int i=0; i<chunkCount; i++) {

      // create fresh engine since svmEngine will retain its previous
      // information (svlist, etc.)
```

```
this.newEngine = svmEngine.like();
chunkTally++;
  System.out.println("Chunk: " + chunkTally + "\t\tChunkSize=" +
      data[i].getLabels().size());
  resCapture += "\n" + chunkTally + "\t" + data[i].getLabels().size();

// run the SVM
this.chunkRes = newEngine.learnSVM(data[i], param);
this.polarizationSet = chunkRes.getPolarizationSet();
this.penaltySet = chunkRes.getPenaltySet();
this.kktViolators = chunkRes.getkktViolators();
this.svList = chunkRes.getSvList();
this.model = chunkRes.getModel();

if(i < chunkCount-1) {
  /*
   * Pass some features on to the next chunk.
   * Create temp array to store training features:
   */
  TrainingData tempData[] = new TrainingData[7];

  // Load the entire next chunk
  tempData[0] = data[i+1];

  /*
   * Use the chunk parameters to set how many of each feature type will
   * be passed on to the next chunk
   */
  if(penaltySet.size() > 0 && chunkPar.penaltySet > 0) {
    int[] indices = extractRandFeatures(penaltySet,
      chunkPar.penaltySet);
    tempData[3] = new
      TrainingData(data[i].getFeats().viewSelection(indices, null),
      data[i].getLabels().viewSelection(indices));
  }
  if(polarizationSet.size() > 0 && chunkPar.polarizationSet > 0) {
    int[] indices = extractRandFeatures(polarizationSet,
      chunkPar.polarizationSet);
    tempData[4] = new
      TrainingData(data[i].getFeats().viewSelection(indices, null),
      data[i].getLabels().viewSelection(indices));
  }
  if(svList.size() > 0 && chunkPar.suppVectorCount > 0) {
    int[] indices = extractBestScoredFeatures(svList.size(),
      chunkPar.suppVectorCount, chunkRes.getScores());
    tempData[5] = new
      TrainingData(data[i].getFeats().viewSelection(indices, null),
      data[i].getLabels().viewSelection(indices));
  }
  if(kktViolators.size() > 0 && chunkPar.kktViolators > 0) {
    int[] indices = extractRandFeatures(kktViolators,
      chunkPar.kktViolators);
    tempData[6] = new
      TrainingData(data[i].getFeats().viewSelection(indices, null),
      data[i].getLabels().viewSelection(indices));
  }
```

```
        System.out.println("\nPassing features to next chunk -->  -->  -->");
        System.out.print("  penaltySet: "+(int)Math.ceil(penaltySet.size() *
            (chunkPar.penaltySet/100.0)));
        System.out.print("  polarizationSet:
            "+(int)Math.ceil(polarizationSet.size() *
            (chunkPar.polarizationSet/100.0)));
        System.out.print("  kktViolators: " +
            (int)Math.ceil(kktViolators.size() *
            (chunkPar.kktViolators/100.0)));
        System.out.print("  Support Vectors: "+(int)Math.ceil(svList.size() *
            (chunkPar.suppVectorCount/100.0)));

        int total = (int)Math.ceil(penaltySet.size() *
            (chunkPar.penaltySet/100.0)) +
            (int)Math.ceil(polarizationSet.size() *
            (chunkPar.polarizationSet/100.0)) +
            (int)Math.ceil(kktViolators.size() *
            (chunkPar.kktViolators/100.0)) + (int)Math.ceil(svList.size() *
            (chunkPar.suppVectorCount/100.0));
        System.out.print("  Total passed: " + total);
        System.out.println("\n-->  -->  -->  -->  -->  -->\n");

        // join the data into a single training set to pass on
        TrainingData nextTrainData = TrainingData.joinTrainingData(tempData);
        data[i+1] = nextTrainData;
      }
    }
    System.out.println("\n@@@@@@@ Specified Chunk Parameters (Percentages):\n
      Penalty Set: " + chunkPar.penaltySet +
      "   Polarization Set: " + chunkPar.polarizationSet);
    System.out.println("  KKT Violators: " + chunkPar.kktViolators + "
      Support Vectors: " + chunkPar.suppVectorCount);
    System.out.print("   Chunk Size: " + chunkPar.chunkSize);
    System.out.println("\n\n" + resCapture);

    this.svmEngine = newEngine;
    return model;
  }

  /**
   * The makeDataChunks method breaks the data into chunks using a shuffled
set of indices making the order pseudo-random.
   * @param chunkCount   the amount of chunks needed
   * @return  an array of TrainingData objects
   */
  private TrainingData[] makeDataChunks(int chunkCount){

    int featsCount = trainingData.getLabels().size();

    // Create a collection to shuffle and load it with indices
    IntArrayList mixedIdcs = new IntArrayList(featsCount);
    for (int i=0; i<featsCount; i++) {
      mixedIdcs.add(i);
    }
    mixedIdcs.shuffle();

    /*
```

```
   * Divide into chunks using the shuffled index list.
   */
  int chunkSize = (int)Math.floor(featsCount/chunkCount);
  int sizeRemainder = featsCount%chunkSize;
  int chunksInd [][] = new int[chunkCount][];
  int k = 0;
  for (int i=0 ; i<chunkCount ; i++) {
    int size = (i != chunkCount-1) ? chunkSize : (chunkSize+sizeRemainder);
    chunksInd[i] = new int[size];

    for (int j=0 ; j<size ; j++) {
      chunksInd[i][j] = mixedIdcs.get(k);
      k++;
    }
  }

  /*
   * build array of trainingData objects
   */
  TrainingData data[] = new TrainingData[chunkCount];
  for (int i=0 ; i<chunkCount ; i++) {
    data[i] = new TrainingData(
        trainingData.getFeats().viewSelection(chunksInd[i], null).copy(),
        trainingData.getLabels().viewSelection(chunksInd[i]).copy());
  }

  return data;
}

/**
 * @return  the user defined parameters
 */
public ModelParameters getParam() {
  return param;
}

/**
 * @param param    the parameters object to set
 */
public void setParam(ModelParameters param) {
  this.param = param;
}

/**
 * @return  the training data
 */
public TrainingData getTrainingData() {
  return trainingData;
}

/**
 * @param trainingData  the training data to set
 */
public void setTrainingData(TrainingData trainingData) {
  this.trainingData = trainingData;
}
```

```
  public ChunkParameters getChunkPar() {
    return chunkPar;
  }

  public void setChunkPar(ChunkParameters chunkPar) {
    this.chunkPar = chunkPar;
  }

  public TrainingData getFirstChunk() {
    return firstChunk;
  }

  public void setFirstChunk(TrainingData firstChunk) {
    this.firstChunk = firstChunk;
  }

  public SVMchunkLearner getSvmEngine() {
    return svmEngine;
  }
}
```

A.2 Java code implementation of multi-threaded chunking method

DistributedChunksSVM.java

```
package edu.uno.cs.bioinformatics.svm.chunking;

import java.util.concurrent.BrokenBarrierException;
import java.util.concurrent.CyclicBarrier;

import cern.colt.list.IntArrayList;
import edu.uno.cs.bioinformatics.data.ModelParameters;
import edu.uno.cs.bioinformatics.data.TrainingData;
import edu.uno.cs.bioinformatics.svm.SVMModel;
import edu.uno.cs.bioinformatics.svm.chunking.SVMchunkLearner;

public class DistributedChunksSVM extends AbstractChunksSVM {

  private ChunkParameters chunkPar;
  private TrainingData trainingData;
  private ModelParameters param;
  private SVMModel model;
  private ChunkResult chunkRes;
  private SVMchunkLearner svmEngine;

    /**
     * The feature vector indices that are non-violators where alpha = C
     */
  private IntArrayList penaltySet = new IntArrayList();

    /**
     * The feature vector indices that are non-violators where alpha = 0
     */
  private IntArrayList polarizationSet = new IntArrayList();

    /**
```

```
   * The feature vector indices that are KKT violators.
   */
private IntArrayList kktViolators = new IntArrayList();

/**
 * The support feature vector indices.
 */
private IntArrayList svList = new IntArrayList();

/**
 * Constructs a new distributed chunk trial with the given SVMLearner,
 * TrainingData, and Parameters.
 * @param svmEngine the SVM learner to use
 * @param chunkPar  the user defined chunking parameters
 * @param trainingData  the features vector and labels to train on
 * @param param      the user defined SVM parameters
 */
public DistributedChunksSVM(SVMchunkLearner svmEngine, ChunkParameters
    chunkPar, TrainingData trainingData, ModelParameters param) {
  this.svmEngine = svmEngine;
  this.chunkPar = chunkPar;
  this.trainingData = trainingData;
  this.param = param;
}

/**
 * Default Constructor
 */
public DistributedChunksSVM() {}

/**
 * The runDistributedChunkedSVM method distributes and runs the SVM chunks
 * using multi-threading.
 * @return  the results model
 */
public SVMModel runDistributedChunkedSVM() {

  int featsCount = trainingData.getLabels().size();
  int chunkCount =
    (int)Math.ceil((featsCount*1.)/(chunkPar.getChunkSize()*1.));
  int maxChunks = chunkPar.getMaxChunks();
  int chunkLevel = 0;
  int chunkTally = 0;
  TrainingData nextDataSet = trainingData;
  String resCapture = "Chunk Level\tChunk Number\tChunk Size";

  while (chunkCount > 1) {

    TrainingData[] data = makeDataChunks(chunkCount, nextDataSet);
    // re-initialize a smaller trial occurs
    maxChunks = chunkPar.getMaxChunks();
    int cnt = 0;

    System.out.println("\n~~~~~~Chunk Level: " + chunkLevel);
    chunkLevel++;
    /*
     * tempData is used to store the different categories of feature
```

```
 * vectors to create the next set of chunks.
 * Create temp array to store training features
 */
TrainingData tempData[] = new TrainingData[6 * chunkCount];
int j = 0;
int c = 0;

// traverse through groupings of chunks at a time to prevent memory
// overflow
while (chunkCount > cnt) {

  SVMchunkLearnerThread[] svms = new SVMchunkLearnerThread[maxChunks];
  /*
   * This next part calls the SVM threads and gets back the chunk
   * results
   */
  System.out.println("\nRunning the SVM Threads on Remote Servers...");

  if ((chunkCount-cnt) < maxChunks) {
    maxChunks = chunkCount-cnt;
  }

  CyclicBarrier barrier = new CyclicBarrier(maxChunks+1, new
      BarrierAction());

  for (int i=0; i<maxChunks; i++) {
    //run a thread
    try {
      svms[i] = new SVMchunkLearnerThread(svmEngine.like(),
            data[c],(ModelParameters)param.clone(), barrier);
    }
    catch (NullPointerException e) {
      e.printStackTrace();
    }
    catch (CloneNotSupportedException e) {
      e.printStackTrace();
    }
    new Thread(svms[i]).start();

    // increment current chunk count
    cnt++;

    // increment running tally of total chunks
    chunkTally++;
    System.out.println("Chunk: " + chunkTally + "\t\tChunkSize=" +
      data[c].getLabels().size());
    resCapture += "\n" + chunkLevel + "\t" + chunkTally + "\t" +
      data[c].getLabels().size();
    c++;
  }

  //wait for all of the threads to come back
  try {
    barrier.await();
  }
  catch (BrokenBarrierException e) {
    e.printStackTrace();
```

```
  }
  catch (InterruptedException e) {
    e.printStackTrace();
  }

  System.out.println("\nReceived the chunk results from threads!");

  //traverse through the chunks
  for (int i=0, e=cnt-maxChunks; i < maxChunks; i++,e++) {
    this.chunkRes = svms[i].getChunk();
    this.polarizationSet = chunkRes.getPolarizationSet();
    this.penaltySet = chunkRes.getPenaltySet();
    this.kktViolators = chunkRes.getkktViolators();
    this.svList = chunkRes.getSvList();
    this.model = chunkRes.getModel();

    /*
     * Use the chunk parameters to set how many of each feature type
     * will be passed on to the next chunk
     */
    if(polarizationSet.size() > 0 && chunkPar.polarizationSet > 0) {
      int[] indices = extractRandFeatures(polarizationSet,
            chunkPar.polarizationSet);
      tempData[j] = new
            TrainingData(data[e].getFeats().viewSelection(indices,
            null), data[e].getLabels().viewSelection(indices));
      j++;
    }
    if(penaltySet.size() > 0 && chunkPar.penaltySet > 0) {
      int[] indices = extractRandFeatures(penaltySet,
            chunkPar.penaltySet);
      tempData[j] = new
            TrainingData(data[e].getFeats().viewSelection(indices,
            null), data[e].getLabels().viewSelection(indices));
      j++;
    }
    if(kktViolators.size() > 0 && chunkPar.kktViolators > 0) {
      int[] indices = extractRandFeatures(kktViolators,
            chunkPar.kktViolators);
      tempData[j] = new
            TrainingData(data[e].getFeats().viewSelection(indices,
            null), data[e].getLabels().viewSelection(indices));
      j++;
    }
    if(svList.size() > 0 && chunkPar.suppVectorCount > 0) {
      int[] indices = extractBestScoredFeatures(svList.size(),
      chunkPar.suppVectorCount, chunkRes.getScores().copy());
      tempData[j] = new
            TrainingData(data[e].getFeats().viewSelection(indices,
            null), data[e].getLabels().viewSelection(indices));
      j++;
    }
  }
} //end of while loop that traverse groupings of chunks

System.out.println("\nPassing features to make next set of chunks (or
      the final chunk) -->  -->  -->");
```

```
    System.out.print("  penaltySet: " + (chunkCount *
          (int)Math.ceil(penaltySet.size() * (chunkPar.penaltySet/100.0))));
    System.out.print("  polarizationSet: " + (chunkCount *
          (int)Math.ceil(polarizationSet.size() *
          (chunkPar.polarizationSet/100.0))));
    System.out.print("  kktViolators: " + (chunkCount *
          (int)Math.ceil(kktViolators.size() *
          (chunkPar.kktViolators/100.0))));
    System.out.print("  Support Vectors: " + (chunkCount *
          (int)Math.ceil(svList.size() *
          (chunkPar.suppVectorCount/100.0))));

    int total = chunkCount * ((int)Math.ceil(penaltySet.size() *
          (chunkPar.penaltySet/100.0))
          + (int)Math.ceil(polarizationSet.size() *
          (chunkPar.polarizationSet/100.0))
          + (int)Math.ceil(kktViolators.size() *
          (chunkPar.kktViolators/100.0))
          + (int)Math.ceil(svList.size() *
          (chunkPar.suppVectorCount/100.0)));
    System.out.print("  Total passed: " + total);
    System.out.println("\n-->  -->  -->  -->  -->  -->\n");

    // join the data into a single training set to pass on
    nextDataSet = TrainingData.joinTrainingData(tempData);

    // reset the chunkCount variable according to the chunkSize
    chunkCount =
          (int)Math.floor((nextDataSet.getLabels().size()*1.)/
          (chunkPar.getChunkSize()*1.));

    if (chunkCount <= 1) {
      // run the final chunk
      System.out.println("\nRunning the final chunk...");
      System.out.println("Final Chunk Size:" +
          nextDataSet.getLabels().size());
      chunkLevel++;
      chunkTally++;
      resCapture += "\n" + chunkLevel + "\t" + chunkTally + "\t" +
          nextDataSet.getLabels().size();

      this.chunkRes = svmEngine.learnSVM(nextDataSet, param);
      this.model = chunkRes.getModel();
    }
  }

  System.out.println("\n@@@@@@@ Specified Chunk Parameters:\n  Penalty Set:
    " + chunkPar.penaltySet + "   Polarization Set: " +
    chunkPar.polarizationSet);
  System.out.println("   KKT Violators: " + chunkPar.kktViolators + "
    Support Vectors: " + chunkPar.suppVectorCount);
  System.out.println("  Chunk Size: " + chunkPar.chunkSize + "\n");
  System.out.println(resCapture);

  return model;
}
```

```
/**
 * The makeDataChunks method breaks the data into chunks using a shuffled
 * set of indices making the order pseudo-random.
 * @param chunkCount   the amount of chunks needed
 * @return  an array of TrainingData objects
 */
private TrainingData[] makeDataChunks(int chunkCount,
    TrainingData trnData){

  int featsCount = trnData.getLabels().size();

  // Create a collection to shuffle and load it with indices
  IntArrayList mixedIdcs = new IntArrayList(featsCount);
  for (int i=0; i<featsCount; i++) {
    mixedIdcs.add(i);
  }
  mixedIdcs.shuffle();

  /*
   * Divide into chunks using the shuffled index list.
   */
  int chunkSize = (int)Math.floor(featsCount/chunkCount);
  int sizeRemainder = featsCount%chunkSize;
  int chunksInd [][] = new int[chunkCount][];
  int k = 0;
  for (int i=0 ; i<chunkCount ; i++) {
    int size = (i != chunkCount-1) ? chunkSize : (chunkSize+sizeRemainder);
    chunksInd[i] = new int[size];

    for (int j=0 ; j<size ; j++) {
      chunksInd[i][j] = mixedIdcs.get(k);
      k++;
    }
  }

  /*
   * build array of trainingData objects
   */
  TrainingData data[] = new TrainingData[chunkCount];
  for (int i=0 ; i<chunkCount ; i++) {
    data[i] = new TrainingData(
        trnData.getFeats().viewSelection(chunksInd[i], null).copy(),
        trnData.getLabels().viewSelection(chunksInd[i]).copy());
  }

  return data;
}

/**
 * @return  the user defined parameters
 */
public ModelParameters getParam() {
  return param;
}

/**
 * @param param    the parameters object to set
```

```
   */
  public void setParam(ModelParameters param) {
    this.param = param;
  }

  /**
   * @return  the training data
   */
  public TrainingData getTrainingData() {
    return trainingData;
  }

  /**
   * @param trainingData  the training data to set
   */
  public void setTrainingData(TrainingData trainingData) {
    this.trainingData = trainingData;
  }

  public ChunkParameters getChunkPar() {
    return chunkPar;
  }

  public void setChunkPar(ChunkParameters chunkPar) {
    this.chunkPar = chunkPar;
  }

  /** The action that would coordinate each Worker thread's results. */
    private static class BarrierAction implements Runnable {
        public void run() {
        }
    }
}
```

A.3 Java code implementation of multi-threaded distributed chunking method

RemoteSVMchunksLearnerThread.java

```
package edu.uno.cs.bioinformatics.rmisvm.chunking;

import java.rmi.RMISecurityManager;
import java.util.concurrent.BrokenBarrierException;
import java.util.concurrent.CyclicBarrier;
import java.rmi.Naming;
import edu.uno.cs.bioinformatics.data.TrainingData;
import edu.uno.cs.bioinformatics.data.ModelParameters;
import edu.uno.cs.bioinformatics.svm.chunking.ChunkResult;
import edu.uno.cs.bioinformatics.svm.chunking.SVMchunkLearner;

public class RemoteSVMchunkLearnerThread implements Runnable {

  private TrainingData trainSet;
  private ModelParameters parms;
  private ChunkResult chunk;
  private SVMchunkLearner learner;
  private String server;
```

```
  private final CyclicBarrier barrier;

  // Constructor
  public RemoteSVMchunkLearnerThread(String server, SVMchunkLearner learner,
      TrainingData trainSet, ModelParameters parms, CyclicBarrier barrier) {

    this.learner = learner;
    this.trainSet = trainSet;
    this.parms = parms;
    this.barrier = barrier;
    this.server = server;
  }

  public void run() {
    if (System.getSecurityManager() == null) {
          System.setSecurityManager(new RMISecurityManager());
    }
    try {
      RemoteSVMchunkLearner rmi = (RemoteSVMchunkLearner)
          Naming.lookup(server);
      chunk = rmi.runSVM(learner, trainSet, parms);

      try {
        barrier.await();
      }
      catch (InterruptedException ex) {
        System.err.println("InterruptedException in
            RemoteSVMchunkLearnerThread!");
      }
      catch (BrokenBarrierException ex) {
        System.err.println("BrokenBarrierException in
            RemoteSVMchunkLearnerThread!");
      }
    }
    catch (Exception e) {
      System.err.println("RemoteSVMchunkLearnerThread exception: " +
    e.getMessage());
        e.printStackTrace();
    }
  }

  public ChunkResult getChunk() {
    return chunk;
  }
}
```

RemoteSVMchunksLearnerImpl.java

```
package edu.uno.cs.bioinformatics.rmisvm.chunking;

import java.rmi.server.UnicastRemoteObject;
import java.rmi.RemoteException;
import java.rmi.RMISecurityManager;
import java.rmi.registry.*;
import edu.uno.cs.bioinformatics.data.ModelParameters;
import edu.uno.cs.bioinformatics.data.TrainingData;
import edu.uno.cs.bioinformatics.svm.chunking.ChunkResult;
```

```
import edu.uno.cs.bioinformatics.svm.chunking.SVMchunkLearner;

public class RemoteSVMchunkLearnerImpl extends UnicastRemoteObject implements
      RemoteSVMchunkLearner {

  static final long serialVersionUID = 235L;

  public RemoteSVMchunkLearnerImpl() throws RemoteException {
    super();
  }

  public ChunkResult runSVM(SVMchunkLearner svmEngine, TrainingData
      featsTrain, ModelParameters parms) {

    System.out.println("Training......(may take a long time)");
    ChunkResult chunk = svmEngine.learnSVM(featsTrain, parms);

    return chunk;
  }

  public static void main(String[] args) {

    if (System.getSecurityManager() == null) {
      System.setSecurityManager(new RMISecurityManager());
    }
    try {
      RemoteSVMchunkLearner engine = new RemoteSVMchunkLearnerImpl();
      Registry registry = LocateRegistry.getRegistry();

      // this is the HTTP way with local registry (new registry per server)
      registry.rebind("RemoteSVMchunkLearner", engine);
      System.out.println("Remote SVMchunkLearner ready!");

    }
    catch (Exception e) {
      System.err.println("RemoteSVMchunkLearner exception: ");
        e.printStackTrace();
    }
  }
}
```

A.4 Java code implementation of SMO SVM (non-chunked)

Note: The following code was developed with Sam Merat and others from the Winters-Hilt group.

SMOLearner.java

```
package edu.uno.cs.bioinformatics.svm;

import java.util.Random;
import cern.colt.list.IntArrayList;
import cern.colt.matrix.DoubleMatrix1D;
import cern.colt.matrix.DoubleMatrix2D;
import cern.colt.matrix.impl.DenseDoubleMatrix1D;
import edu.uno.cs.bioinformatics.data.TrainingData;
import edu.uno.cs.bioinformatics.svm.SMOParameters;
import edu.uno.cs.bioinformatics.data.ModelParameters;
```

```
import edu.uno.cs.bioinformatics.svm.SVMModel;

/**
 *
 * @author Ken Armond and Sam Merat
 *
 */

/**
 * The SMOLearner class contains the implementation of the Platt SVM.
 */
public class SMOLearner implements SVMLearner
{
  static final long serialVersionUID = 23L;

  /**
   * The model which will be loaded with the final results after training.
   */
  private SVMModel model;

  /**
   * The kernel matrix
   */
  private DoubleMatrix2D kernelMat;

  /**
   * The feature vectors and labels for training.
   */
  private TrainingData trainingData;

  /**
   * The seed for the pseudo-random number generator.
   */
  private long seed;

  /**
   * The error cache matrix
   */
  private DoubleMatrix1D errorCache;

  /**
   * The Langrangian multiplier storage matrix
   */
  private DoubleMatrix1D alphas;

  /**
   * The user defined parameters from the properties file
   */
  private SMOParameters param;

  /**
   * The threshold value (also known as the B value)
   */
  private double threshold;

  private int iter;
```

```
private SVReductionForceZero postProcedure;

/**
 * Default constructor
 */
public SMOLearner() {
  this.postProcedure = new SVReductionForceZero();
}

private synchronized double outputNonlinear(int i) {

  double alphaJ = 0.0;
    double sum = 0.0;
    DoubleMatrix1D labels = trainingData.getLabels();

    for (int j=0; j < alphas.size(); j++) {
        if ((alphaJ = alphas.get(j)) > 0.0)
          sum += alphaJ * labels.get(j) * kernelMat.get(i, j);
    }
    return sum - threshold;
}

private int examineExample(int i2)
{
  double r2 = 0.0;
  double E2 = 0.0;
  double alpha_2 = alphas.get(i2);
  double y2 = trainingData.getLabels().get(i2);

  double cVal = param.getCVal();
  double tolerance = param.getTolerance();
  int featsCount = trainingData.getFeats().rows();

  Random rand = new Random(seed);

  if (alpha_2 > 0 && alpha_2 < cVal)
    E2 = errorCache.get(i2);
  else
    E2 = outputNonlinear(i2) - y2;

  r2 = E2 * y2;

  /*
   * if alpha2 violates the KKT condition within a tolerance
   * then look for an alpha1 and optimize both alphas (take_step(i1,i2))
   */
  if ((r2 < -tolerance && alpha_2 < cVal)
     || (r2 > tolerance && alpha_2 > 0))
  {
    {
      /*
       * Once a alpha2 is chosen, SMO chooses alpha1 to maximize
       * the size of the step taken during joint optimization
       * (take_step(i1,i2))
       */
       int i1 = -1;
       double tmax = 0;
```

```
   for (int k = 0; k < featsCount; k++)
   {
       double alpha_k = alphas.get(k);

       if (0 < alpha_k && alpha_k < cVal)
       {
           double temp;
           double E1 = errorCache.get(k);

          /*
           *SMO approximates the step size by absolute value of (E1-E2)
           */
           temp = Math.abs(E1 - E2);
           if (temp > tmax) {
               tmax = temp;
               i1 = k;
           }
       }
   }
   if (i1 > -1 ) {
       if (takeStep(i1, i2) == 1)
         return 1;
   }
}

/*
 * At this point no positive progress was made (last paragraph
 * in Platt's paper section 2.4).
 * first check the non bound alphas from a random place
 */
{
  int k = 0;
   int i1 = -1;
   int k0 = Math.abs(rand.nextInt() * featsCount);

   for (k = k0; k < featsCount + k0; k++)
   {
       i1 = k % featsCount;
       double alpha_k = alphas.get(i1);
       if (0 < alpha_k && alpha_k < cVal)
       {
           if (takeStep(i1, i2) == 1)
               return 1;
       }
   }
}

/*
 * if still no progress then iterate through all feature vectors
 * starting from a random place
 */
{
  int k = 0, i1=-1;
  int k0 = Math.abs(rand.nextInt() * featsCount);

  for (k = k0; k < featsCount + k0; k++)
```

```
      {
          i1 = k % featsCount;
          if (takeStep(i1, i2) == 1)
              return 1;
      }
    }
  }

  return 0;
}

/**
 * The takeStep method takes two feature vectors and recalculates the
 * Lagrangian multipliers (alphas) associated with them.
 * It also updates the error cache array and threshold.
 *
 * @param i1     first feature vector index.
 * @param i2     second feature vector index.
 * @return int     returns 1 if the alphas get updated successfully.
 */

public int takeStep(int i1, int i2)
{
  double cVal = param.getCVal();
  double epsilon = param.getEpsilon();
  int featsCount = trainingData.getFeats().rows();

  double alpha_old_1, alpha_old_2;  // old_values of alpha_1, alpha_2
  double alpha_new_1, alpha_new_2;    // new values of alpha_1, alpha_2
  double y1, y2, s, E1, E2, L, H, k11, k22, k12, eta, Lobj, Hobj;

  if (i1 == i2) {
    return 0;
  }

  alpha_old_1 = alphas.get(i1);
  y1 = trainingData.getLabels().get(i1);

  if (alpha_old_1 > 0 && alpha_old_1 < cVal)
    E1 = errorCache.get(i1);
  else
    E1 = outputNonlinear(i1) - y1;

  alpha_old_2 = alphas.get(i2);
  y2 = trainingData.getLabels().get(i2);

  if (alpha_old_2 > 0 && alpha_old_2 < cVal)
    E2 = errorCache.get(i2);
  else
    E2 = outputNonlinear(i2) - y2;

  s = y1 * y2;

  if (y1 == y2)
  {
    double gamma = alpha_old_1 + alpha_old_2;
    if (gamma > cVal)
```

```
  {
    L = gamma-cVal;
    H = cVal;
  }
  else
  {
    L = 0;
    H = gamma;
  }
}
else
{
  double gamma = alpha_old_2 - alpha_old_1;
  if (gamma > 0)
  {
    L = gamma;
    H = cVal;
  }
  else
  {
    L = 0;
    H = cVal + gamma;
  }
}

if (L == H) {
  return 0;
}

k11 = kernelMat.get(i1, i1);
k12 = kernelMat.get(i1, i2);
k22 = kernelMat.get(i2, i2);
eta = k11 + k22 - 2*k12;

if (eta > 0)
{
  alpha_new_2 = alpha_old_2 + y2 * (E1 - E2) / eta;
  if (alpha_new_2 < L)
    alpha_new_2 = L;
  else if (alpha_new_2 > H)
    alpha_new_2 = H;
}
else
{
  double f1 = y1 * (E1 + threshold) - alpha_old_1 * k11 - s * alpha_old_2
              * k12;
  double f2 = y2 * (E2 + threshold) - alpha_old_2 * k22 - s * alpha_old_1
              * k12;
  double l1 = alpha_old_1 + s * (alpha_old_2-L);
  double h1 = alpha_old_1 + s * (alpha_old_2-H);

  Lobj = l1*f1 + L*f2 + 1/2 * ( (l1*l1)*k11 + (L*L)*k22 + 2*s*L*l1*k12 );
  Hobj = h1*f1 + H*f2 + 1/2 * ( (h1*h1)*k11 + (H*H)*k22 + 2*s*H*h1*k12 );

  if (Lobj < Hobj-epsilon)
    alpha_new_2 = L;
  else if (Lobj > Hobj+epsilon)
```

```
    alpha_new_2 = H;
  else
    alpha_new_2 = alpha_old_2;
}

if (Math.abs(alpha_new_2 - alpha_old_2) < epsilon * (alpha_new_2 +
    alpha_old_2 + epsilon) ) {
  return 0;
}

alpha_new_1 = alpha_old_1 + s * (alpha_old_2 - alpha_new_2);

if (alpha_new_1 < 0)
{
  alpha_new_2 += s * alpha_new_1;
  alpha_new_1 = 0;
}
else if (alpha_new_1 > cVal)
{
  double t = alpha_new_1 - cVal;
  alpha_new_2 += s * t;
  alpha_new_1 = cVal;
}

/* updating the threshold */
double b1, b2, bnew, delta_b;
b1 = threshold + E1 + y1 * (alpha_new_1 - alpha_old_1) * k11 + y2 *
     (alpha_new_2 - alpha_old_2) * k12;
b2 = threshold + E2 + y1 * (alpha_new_1 - alpha_old_1) * k12 + y2 *
     (alpha_new_2 - alpha_old_2) * k22;

if (alpha_new_1 > 0 && alpha_new_1 < cVal)
  bnew = b1;
else if (alpha_new_2 > 0 && alpha_new_2 < cVal)
  bnew = b2;
else
  bnew = (b1 + b2) / 2;

delta_b = bnew - threshold;
threshold = bnew;

/*
 * updating the error cache
 */
double t1 = y1 * (alpha_new_1-alpha_old_1);
double t2 = y2 * (alpha_new_2-alpha_old_2);

for (int i = 0; i < featsCount; i++)
{
  double alpha_i = alphas.get(i);

  if (0 < alpha_i && alpha_i < cVal)
  {
    double k1i = kernelMat.get(i1, i);
    double k2i = kernelMat.get(i2, i);;
    double error_old_i = errorCache.get(i);
    double error_new_i = error_old_i + t1*k1i + t2*k2i - delta_b;
```

```
      errorCache.set(i, error_new_i);
    }
  }
  errorCache.set(i1, 0.0);
  errorCache.set(i2, 0.0);

  /*
   * updating the alphas
   */
  alphas.set(i1, alpha_new_1);
  alphas.set(i2, alpha_new_2);

  return 1;
}

/**
 *The learnSVM method runs the SVM and returns a model of the trained data.
 *The support vectors are collected into a matrix and saved to the model.
 *The Bound KKT violators, Unbound KKT violators, and the threshold are
 *also saved to the model.
 *@param data     the feature vectors to be trained.
 *@param mparam     the user defined parameters set in the properties file.
 *@return model    the results of the SVM training (Support Vectors, KKT
 *Violators, etc.)
 */
public SVMModel learnSVM(TrainingData data, ModelParameters mparam)
  throws IllegalArgumentException {

  /*
   * Checking the validity of the data and param
   */
  if (data == null || mparam == null || !(mparam instanceof SMOParameters))
    throw new IllegalArgumentException("The training data or the parameters
                                        are undefined.");
  else if (mparam.getKernelf().getKernelMat() == null)
    mparam.getKernelf().buildDenseKernelMatrix(data.getFeats());
  {
    this.param = (SMOParameters) mparam;
    this.trainingData = data;
    this.kernelMat = param.getKernelf().getKernelMat();
    this.seed = param.getSeed();
  }

  /*
   * The number of features to be used for training
   */
  int featsCount = data.getFeats().rows();

  /*
   * initialize temporary alphas and errorCache vectors
   */
  this.errorCache = new DenseDoubleMatrix1D(featsCount);
  this.alphas = new DenseDoubleMatrix1D(featsCount);

  /*
   * Various local vars
   */
```

```
double cVal = param.getCVal();
int maxIter = param.getMaxIter();
/*
 * The SMO outer loop
 */
int numChanged   = 0;
  int examineAll   = 1;

  iter = 0;

  while ((numChanged > 0 || examineAll == 1) &&
      iter < ((maxIter == 0)?iter+1:maxIter))
  {
    iter++;
    numChanged = 0;
    if (examineAll == 1)
    {
      for (int i = 0 ; i < featsCount; i++)
      {
        numChanged += examineExample(i);
      }
    }
    else
    {
      for (int i = 0 ; i < featsCount; i++)
      {
        double alpha = alphas.getQuick(i);
        if (alpha != cVal || alpha != 0.0)
        {
          numChanged += examineExample(i);
        }
      }
    }

    if  (examineAll == 1)
      examineAll = 0;
    else if (numChanged == 0)
      examineAll = 1;
  }

  // perform alpha cutoff rule to force small alphas to zero
  postProcedure.execute(this, data);

  System.out.print("SMO DONE");

  /*
   * make the model object
   */
  {
    model = new SVMModel(param);
    model.setIter(iter);
    model.setThreshold(threshold);

    /*
     * allocate the Support Vectors and Polarization Vectors
     */
    IntArrayList nonZeroAlphaIndList = new
```

```
                IntArrayList((int)featsCount/2);
        IntArrayList polarizationIndList = new
                IntArrayList((int)featsCount/2);
        int upto = 0;
        for (int i = 0; i < featsCount; i++)
        {
          double alpha = alphas.get(i);
          if (alpha != 0.0 && alpha != cVal)
            nonZeroAlphaIndList.add(i);

          // count the vectors in the polarization set
          if (alpha == 0) {
            polarizationIndList.add(i);
          }
          ++upto;
        }

        nonZeroAlphaIndList.trimToSize();
        model.setSuppFeats(data.getFeats().
              viewSelection(nonZeroAlphaIndList.elements(),null).copy());

        /*
         * allocate the alpha corresponding to support vectors
         */
        DoubleMatrix1D tmpSvAlphas =
        trainingData.getLabels().like(nonZeroAlphaIndList.size());
        for (int i=0 ; i<nonZeroAlphaIndList.size() ; i++)
        {
          tmpSvAlphas.setQuick(i,
          alphas.get(nonZeroAlphaIndList.getQuick(i)));
        }
        model.setSvAlphas(tmpSvAlphas);

        System.out.println(" ; #SV = "+tmpSvAlphas.size() + " ; b =
        "+model.getThreshold()+ " ;  Iterations:" + iter);
        /*
         * allocate original labels
         */
        model.setSvLabels(trainingData.getLabels().
              viewSelection(nonZeroAlphaIndList.elements()).copy());
        model.setSvIndices(nonZeroAlphaIndList);
        model.setPolarizationIndices(polarizationIndList);
      }

  return model;
}

/**
 * The like() method returns a new copy of a SVMLearner object.
 */
public SVMLearner like() {
  return new SMOLearner();
}

public DoubleMatrix1D getAlphas() {
  return alphas;
}
```

```
  public SMOParameters getParam() {
    return param;
  }

  public SVReductionForceZero getPostProcedure() {
    return postProcedure;
  }

  public void setPostProcedure(SVReductionForceZero postProcedure) {
    this.postProcedure = postProcedure;
  }
}
```

Kenneth C. Armond Jr.
39598 Walnut Dr.
Pearl River, LA 70452
July 1, 2008

Dr. Stephen Winters-Hilt
Department of Computer Science
University of New Orleans
New Orleans, LA 70148

Dear Dr. Winters-Hilt:
I am completing a masters thesis at the University of New Orleans entitled "Distributed Support Vector Machine Learning". I would like your permission to reprint in my thesis excerpts from the following: *"Support Vector Machine Implementations for Classification & Clustering", BMC Bioinformatics, (2006).*
The excerpts to be reproduced are: SVM derivation
The requested permission extends to any future revisions and editions of my thesis. These rights will in no way restrict republication of the material in any other form by you or by others authorized by you. Your signing of this letter will also confirm that you own the copyright to the above-described material. If these arrangements meet with your approval, please sign this letter where indicated below. Thank you very much.
Sincerely,
Kenneth C. Armond Jr.

PERMISSION GRANTED FOR THE USE REQUESTED ABOVE:

Stephen Winters-Hilt
Date: 6/25/2008

Vita

Kenneth Armond Jr. was born in New Orleans, Louisiana. In 2003, he graduated from Texas A&M University in College Station, Texas with a Bachelor's Degree of Science in Computer Engineering (Electrical Engineering track). During and after this time, he had various internships in the Information Technology field followed by a three year stint of full time work as a Software Engineer. He will return to work in industry to use his master's degree but he plans to eventually return to academia to work on a doctorate degree in Computer Science.