

Parallel Tuning of Support Vector Machine Learning Parameters for Large and Unbalanced Data Sets

Tatjana Eitrich¹ and Bruno Lang²

¹ John von Neumann Institute for Computing,
Central Institute for Applied Mathematics,
Research Centre Juelich, Germany
`t.eitrich@fz-juelich.de`

² Applied Computer Science and Scientific Computing Group,
Department of Mathematics,
University of Wuppertal, Germany
`Bruno.Lang@math.uni-wuppertal.de`

Abstract. We consider the problem of selecting and tuning learning parameters of support vector machines, especially for the classification of large and unbalanced data sets. We show why and how simple models with few parameters should be refined and propose an automated approach for tuning the increased number of parameters in the extended model. Based on a sensitive quality measure we analyze correlations between the number of parameters, the learning cost and the performance of the trained SVM in classifying independent test data. In addition we study the influence of the quality measure on the classification performance and compare the behavior of serial and asynchronous parallel parameter tuning on an IBM p690 cluster.

1 Introduction

Support vector machines (SVMs) are one of the well accepted machine learning methods [1]. Numerous experiments have confirmed that the linear learning approach in combination with problem adapted implicit feature mappings leads to highly reliable nonlinear classification functions. Much work has been done to make SVM algorithms run very fast [2].

In recent years, however, a significant number of nontrivial problems has surfaced in the context of SVMs. The size of the classification problems increases rapidly, while at the same time better results are desired. The quality issue is particularly important if the data sets are unbalanced, which means that either the number of positive and negative data differ significantly, or the cost of a false positive classification differs significantly from the cost of a false negative, or both. Often, the differing costs of misclassifications have been neglected, and the success of a particular approach has been measured only by totaling the number of incorrectly classified test points.

Support vector machine classification involves a learning phase, in which the training data are used to adjust the classification parameters. This procedure, which can be formulated as a quadratic programming problem, is controlled by a—typically very small—set of learning parameters. Usually these have to be set by the user. Quoting [3], “*There is a lot of papers published about the SVM algorithms and kernel methods, but very few of them address the parameters tuning to get the high quality results usually presented [...] these results are difficult to reproduce because of the influence of the parameter settings.*” In addition, it is often not clear which quality measure had been used.

In this paper we address the classification of *large unbalanced* data sets with SVMs, taking the differing costs of misclassifications into account. Large data sets feature two properties that are important for the training. On the one hand, they allow considering models with a higher number of learning parameters, so that nonlinearity can be captured more precisely than with models involving only a few parameters. (For smaller data sets, the number of parameters is limited by overfitting effects.) Finding appropriate values for many parameters, however, can no longer be done by hand or simple grid search, but must be automated. To this end we embed the learning in a numerical optimizer, which repeatedly trains an SVM with different settings of the learning parameters and strives to find parameters that optimize a suitable quality measure; see Section 3 for details. The overall procedure for adjusting the *learning parameters and classification parameters* is summarized in Figure 1. Note that evaluating the quality measure involves validating the SVM on data different from the training data.

The negative effect of large data sets is the high computational complexity. To reduce the overall learning time, each SVM training is done with a highly efficient quadratic program solver, and a parallelized optimizer is used for tuning the learning parameters; see Sections 2 and 4. In Section 5 numerical experiments with a large, hard classification problem will show that our automated approach is able to yield good results in a reliable manner. This is important for users from other fields because our method requires no human interaction and no familiarity with the underlying SVM theory to tune the SVM to a particular classification problem.

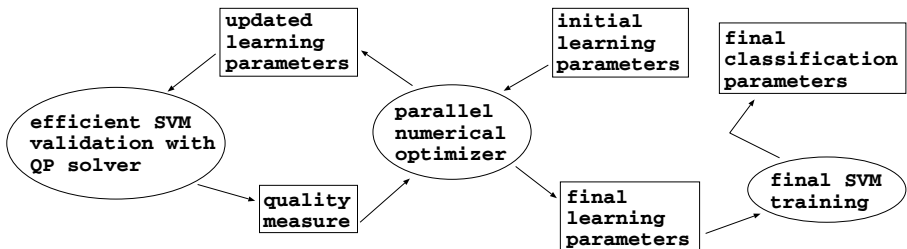


Fig. 1. Main components of the SVM system

2 Support Vector Learning

The task of support vector learning is to determine functions that can be used to classify data points. In this paper we consider only binary problems and leave out multi-class learning and regression. In the binary case support vector learning is the process of using so-called reference data of given input–output pairs $\{(\mathbf{x}_i, y_i) \in \mathbb{R}^n \times \{-1, 1\}, i = 1, \dots, l\}$ to find an optimal separating hyperplane $\mathbf{w}^T \mathbf{x} + b = 0$. Using assumptions of statistical learning theory the desired classifier is then defined as $h(\mathbf{x}) = \text{sgn}(f(\mathbf{x}))$ with the linear decision function $f(\mathbf{x}) = \mathbf{w}^T \mathbf{x} + b$; see [4,5] for details.

If the data are not linearly separable then a *kernel* $K : \mathbb{R}^n \times \mathbb{R}^n \rightarrow \mathbb{R}$ is used to learn a nonlinear decision function

$$f_{\text{nonlin}}^*(\mathbf{x}) = \sum_{i:0 < \alpha_i} y_i \alpha_i^* K(\mathbf{x}_i, \mathbf{x}) + b^*.$$

Here the classification parameters α_i^* and b^* are given by the unique global solution of a suitable (dual) quadratic optimization problem [5]

$$\min_{\alpha \in \mathbb{R}^l} g(\alpha) := \frac{1}{2} \alpha^T H \alpha - \sum_{i=1}^l \alpha_i \quad (1)$$

with $H \in \mathbb{R}^{l \times l}$, $H_{ij} = y_i K(\mathbf{x}_i, \mathbf{x}_j) y_j$ ($1 \leq i, j \leq l$), constrained to

$$\alpha^T \mathbf{y} = 0, \quad \mathbf{0} \leq \alpha \leq \mathbf{C}. \quad (2)$$

The kernel function K must be provided by the user.

Note that the the Hessian H is usually dense, and therefore the complexity of evaluating the objective function g in (1) scales quadratically with the number l of training pairs, leading to very time-consuming computations. A well-known method for the solution of such problems is the decomposition algorithm [6] that repeatedly selects a subset of the free variables and optimizes (1) over these variables. Its main advantage is the flexibility concerning the size of the subproblems. Decomposition provides a framework for handling large SVM training tasks but it does not define how to solve the reduced quadratic programming problems. To obtain good overall times it is necessary to have efficient QP solvers for the subproblems. We use our own implementation of the projection method described in [7]. This method is suitable for large data sets. It defines problems with diagonal matrices and solves them iteratively with a fast inner solver [8]. Thus a single optimization step of the decomposition method becomes very fast.

3 Learning Parameters and Quality Management

The constraints (2) involve learning parameters C_i , $i = 1, \dots, l$, which have to be chosen before SVM training. Often a single value $C_i \equiv C$ is used for simplicity. In [9] the authors gave evidence that for unbalanced data sets at least two values

should be used: $C_i = C^+$ if the i th training point is positive ($y_i = +1$), and $C_i = C^-$ otherwise ($y_i = -1$). In addition to correcting different sizes of the two classes, the (C^+, C^-) model can also capture different costs of false positive and false negative classifications. Since the data set treated in Section 5 is even more unbalanced than the example in [9], with a very small number of positive points, we again used the (C^+, C^-) model.

In addition to this weighting approach we consider generalizing the kernel function. One of the most commonly used functions is the Gaussian kernel,

$$K^G(\mathbf{x}, \mathbf{z}) = \exp\left(-\frac{\sum_{k=1}^n (x_k - z_k)^2}{2\sigma^2}\right) \quad (\mathbf{x}, \mathbf{z} \in \mathbb{R}^n), \quad (3)$$

where the standard deviation $\sigma > 0$ is chosen identically for all features of the data. The reason is again that hand tuning of the learning parameters requires their number to be very small.

If the learning parameters can be adjusted automatically then their number can be increased, and in the extreme case we may assign a different standard deviation to each feature [9]:

$$K^G(\mathbf{x}, \mathbf{z}) = \exp\left(-\sum_{k=1}^n \frac{(x_k - z_k)^2}{2\sigma_k^2}\right). \quad (4)$$

As a reasonable compromise between (3) and (4), one might divide the features into different groups (such as “binary” and “continuous”) and assign one σ value to each group.

Comparing SVMs trained with these extended models to SVMs trained with the usual uniform approach confirms that the added complexity indeed leads to better classification results. Interestingly, allowing different σ values for the features can also yield additional information. Based on the optimized value σ_k one can estimate the relevance of the corresponding feature k , and thus one gets an implicit *feature selection* mechanism for free. To our knowledge, however, the option of tuning different σ values in the context of support vector learning has not been considered elsewhere.

We also work on other generalized kernels and on other parameters that are relevant for the training phase. For example, the decomposition method and the QP solver use several internal parameters, which may be tuned to enhance the performance [6,7]. Both issues cannot be discussed here due to space limitations.

The tuning of the learning parameters can be implemented by optimizing a certain quality measure, which is obtained in validation steps. Optimizing a nontrivial parameter model is almost impossible if a discrete quality measure is used, e.g., the number of validation errors. Following the ideas in [9] we use the continuous effectiveness measure

$$E_\beta = 1 - \frac{(\beta^2 + 1)\text{pr} \cdot \text{se}}{\beta^2 \cdot \text{pr} + \text{se}} \in [0, 1], \quad (5)$$

which we have to *minimize*. The sensitivity *se* (which percentage of the positive data have been recognized ?) and the precision *pr* (which percentage of the points

that have been classified “positive” are indeed positive ?) are computed with a special smooth error measure. The quantity β can be used to enforce or diminish the influence of sensitivity. In Section 5 we will present results achieved with (5) for different values of β and discuss the problem of defining the quality measure.

4 Automatic Parallel Parameter Optimization

Tuning a nontrivial number of parameters can be very time consuming, and therefore it is reasonable to use parallel computing resources. There are three ways to insert parallelism during the SVM model selection stages: parallelizing the training of a single SVM, training several SVMs in parallel, and using a parallel algorithm for parameter optimization. Concerning the first option, promising parallelization techniques for decomposition methods exist [10], whereas parallel SMO [2] methods are currently investigated, but seem not to be reliable yet. The second option has also been addressed with mostly straight-forward approaches, e.g., parallel mixture of SVMs [11], parallel training of binary SVMs for multi-class problems [12], and parallel cross validation models [13]; see [3] for a short overview.

Concerning parallel parameter optimization, ongoing work is on parallel grid search techniques [14]. Grid search uses a predefined set of values for each parameter and determines which combination of these values yields the best results. Thus parallel grid search is an easy and perfectly scalable method that needs no communication at all. Unfortunately this approach scales *exponentially* in the number of parameters and therefore is applicable only for very simple models.

Since we are interested in tuning complex models with larger numbers of parameters, we rely on an efficient numerical optimizer instead. We decided to use the *APPSPACK* [15] software for this task because it does not require derivatives of the objective function and because an MPI-based parallel version is available. Parallelism is achieved by assigning evaluations of the objective function E_β to different processors, the so-called workers. Note that each evaluation of E_β requires a complete cross-validation, which means i) to train SVMs on different training points for a given set of learning parameters, ii) to validate the trained SVMs on different test points and iii) to compute the quality measure. Based on these values, the optimizer selects new promising search directions in the parameter space and checks for convergence. Good load balancing is achieved by using an asynchronous scheme. Currently we exploit only the parallelism provided by *APPSPACK*; the training of single SVMs and the validation routine have not yet been parallelized.

Mapping of SVM Learning onto the *APPSPACK* Environment. The *APPSPACK* software package is freely available. A *configure* script automatically locates the commands and system files that are required for compiling and installing the package, and automatically generates appropriate *makefiles*. Due to some IBM-specific settings these *makefiles* could not be used directly for building the libraries and executables for our machine, but a few additional steps had to be done. More details on the JUMP supercomputer will be given

in Chapter 5. Once we had successfully configured *APPSPACK* we built the libraries and executables by using the *makefiles*. All in all the installation of the software is easy and the *APPSPACK* developer team gives instructions if requested.

The second step consisted of integrating SVM learning into the *APPSPACK* framework. This required only minor changes of the SVM code because the executable just has to read a file containing values for all parameters, to evaluate the objective function E_β , and to provide an output file which should contain either a single numeric entry that is the function value or an error string. *APPSPACK* is able to generate the input files and to read the output without additional instructions. Please note that the optimizer examines the function values exclusively, whereas the underlying simulation is not of any interest. Thus its usage is easy to realize for support vector learning and any other supervised machine learning algorithm. The users' final task is to provide an *apps*-file containing the relevant solver information like

- the number of parameters,
- lower and upper bounds for them (infinite bounds are allowed), and the
- executable name.

Optionally one can set

- the initial parameter vector for a hot start,
- the maximum number of evaluations,

and many more. Some examples are provided, too. For the parallel version the number of workers ω is deduced from the submission of the MPI job via $\omega = \text{proc} - 1$, where *proc* is the number of processors. A single CPU, the master, is used to assign work, i.e., trial points, to the workers. *APPSPACK* is robust due to the toleration of error strings. Even if a single function evaluation fails, the optimization won't stop. For our quality measure (5) such a situation may occur in the case $\text{pr} = \text{se} = 0$, when E_β is not defined.

In the following section we will also compare results of the serial and parallel version to show drawbacks and advantages of both methods. To our knowledge this is the first presentation of work on parallel numerical optimization of non-standard SVM parameters.

5 Results and Discussion

The numerical experiments were performed with the so-called thyroid data set available from [16]. There are 7200 instances with 15 binary and 6 continuous attributes. The task is to determine whether a patient is hypothyroid. Therefore one class, representing 93% of the data, has the characteristic “*not hypothyroid*”. The remaining instances are considered to belong to a single class “*hypothyroid*”, even if a closer inspection would allow to classify them further as either “*hyperfunction*” or “*subnormal functioning*.” This merging is usually proposed, and sometimes the dataset is even distributed in this form with the task of finding

hypothyroid persons. Note that the merging of classes is somewhat critical as we do not know the level of similarity between them. However we try to design a sensitive binary classifier that is able to find as many hypothyroid points as possible. In addition to grossly unequal class sizes, the data set is unbalanced with high cost for false negative results. In [16] the data is already partitioned into a training set of 3772 points and a test set of 3428 points. Since the percentage of positive and negative instances in the proposed training set is compatible with the overall distribution we didn't change this partitioning.

The thyroid data set was used in [17] for performance analysis of multilayer neural networks. The best net reached a classification performance of 95%. It was also stated that due to the imbalance of the data a learning method must perform better than 93%. This is true only for scenarios where errors have always the same weight and are not considered separately. Unfortunately, [17] does not give data concerning the distribution of the errors. In [18] the performance of SVMs for the same data set is given. Standard SVMs achieved between 93% and 95% accuracy on the test set. There the SVM results were compared with results on fuzzy SVM learning. The latter approach led to classification rates between 95% and 97%. Again, the distribution of the errors was not specified.

Our numerical experiments were performed on the Juelich Multi Processor (JUMP) at Research Centre Juelich [19]. JUMP is a distributed shared memory parallel computer consisting of 41 frames (nodes). Each node contains 32 IBM Power4+ processors running at 1.7 GHz and 128 GB shared main memory. All in all the 1312 processors have an aggregate peak performance of 8.9 TFlop/s. Since we used a single node of JUMP for our tests, the *APPSPACK* manager process could assign jobs to 31 workers.

Throughout the tests, some control parameters were kept fixed. For the decomposition method in the SVM training we chose a working set size of 100, and the stopping criterion was defined according to [6] with $\epsilon = 0.001$. The quality measure E_β was computed via a simple twofold cross validation. We did not specify a starting point or a maximum number of evaluations for *APPSPACK*. In contrast to some published results, we kept training data and test data strictly separated. The former were used only for validating and training the SVMs, and the latter were used only for assessing the quality of the final optimized SVM.

The Influence of the Quality Measure. One of the most challenging problems for unbalanced data sets is to find a reasonable trade-off between high sensitivity and high precision. In our quality measure E_β the relative weight of these two important goals is controlled by the parameter β . Since increasing β gives more weight to sensitivity, we expect a reduction of the false negative points, at the cost of a potential growth of false positive results. Indeed the data obtained with a single- σ model confirm this expectation; see Table 1.

Since in our example the cost for false negative classifications is significantly higher than for false positive, we are primarily interested in sensitivity, so a higher value of β should be used. Small values for β led to good overall results with increasing numbers of false negative points. Note that the 98% test per-

Table 1. Test results for different quality measures

| β | 0.5 | 0.75 | 1.0 | 1.5 | 2.5 |
|----------------------------|------------|--------|-------|-------|--------------|
| trial points | 125 | 79 | 75 | 78 | 62 |
| function evaluations | 101 | 62 | 58 | 60 | 46 |
| E_β | 0.092 | 0.102 | 0.108 | 0.099 | 0.072 |
| training errors | 48 | 51 | 88 | 120 | 170 |
| σ | 91.15 | 52.05 | 28.84 | 25.75 | 64.42 |
| C^+ | 100000 | 100000 | 10280 | 39670 | 100000 |
| C^- | 21790 | 19560 | 1000 | 1000 | 1000 |
| ratio C^+/C^- | 4.6 | 5.1 | 10.3 | 39.7 | 100 |
| false negative test points | 7 | 5 | 4 | 3 | 1 |
| false positive test points | 63 | 68 | 99 | 134 | 196 |
| test sensitivity | 97% | 98% | 98% | 99% | 100% |
| overall test errors | 70 | 73 | 103 | 137 | 197 |
| test performance | 98% | 98% | 97% | 96% | 94% |

formance compares favorably with the results obtained with neural networks or fuzzy SVMs.

Even if sensitivity is important, at some point the attempt to reduce the false negatives further leads to so many additional false positives that the overall cost increases again. This reflects the fact that for large and very unbalanced data sets it is dangerous to optimize only sensitivity because this can lead to weak classifiers. In certain situations, however, it can be important to be able to design very sensitive classifiers, e.g., when false positive points can be located by experiments after classification. Possibly the overall performance might also be improved further with another quality measure or with very small values β , but these issues have not yet been investigated.

The Optimal Ratio of C^+ and C^- . While unequal evaluation of slack variables during training seems to be accepted universally in the field of support vector learning [20], detailed descriptions of results or of the effects of this model generalization are not available. Thus tuning of these parameters is not trivial, if done by hand. Since the ratio of positive and negative points is about 7% the natural weighting choice [9] would be $C^+/C^- \approx 14$. However the data in Table 1 indicate that this ratio is not adequate for minimizing either the total number of errors or the sensitivity. The C^+/C^- ratios given in the table were delivered automatically by the numerical optimizer. One can see that the ratio increases for larger values of β , which is exactly what one would expect.

Computational Cost. Simple tuning methods like grid search are very popular due to the predictable number of training stages. In Table 1 we show the number of steps for automatic parameter tuning. Not all trial points generated by *APPSPACK* led to a new function evaluation (cross validation) because sometimes points were regularly pruned or function values in the cache could be reused. For optimizing 3 parameters we had to do between 46 and 101 cross validations, which is at least one order of magnitude less than any reasonable grid search would need.

Generalized vs. Standard Kernel. In Section 3 we showed how the standard Gaussian kernel can be extended by using different standard deviations for (groups of) the features. The thyroid data have 21 features, of which 15 are binary and 6 are continuous. Therefore we used two kernel parameters σ_{bin} and σ_{cont} . In Table 2 we compare the results to those for the standard kernel with a single parameter σ . For both runs we used $\beta = 1.5$. The number of function evaluations for optimizing the generalized model is more than twice as large as for the 3-parameter model. On the other hand, the cost-sensitive quality measure E_β could be reduced. This improvement could be seen in the final test, too. *Both* the false negative and false positive classifications could be lowered.

Table 2. Comparison of the standard and the generalized kernel

| model | standard | generalized |
|----------------------------|----------|-------------|
| function evaluations | 60 | 140 |
| E_β | 0.108 | 0.098 |
| σ | 25.75 | — |
| σ_{bin} | — | 72.16 |
| σ_{cont} | — | 31.38 |
| C^+ | 39670 | 13380 |
| C^- | 1000 | 1000 |
| ratio C^+/C^- | 39.7 | 13.4 |
| false negative test points | 3 | 2 |
| false positive test points | 134 | 110 |
| overall errors | 137 | 112 |

From $\sigma_{\text{cont}} < \sigma_{\text{bin}}$ we conclude that the significance of the binary features is high in comparison to the continuous values. It is interesting to see that $\sigma = 25.75$ is not between the two new σ values.

Serial vs. Parallel Optimization. The training time for a single SVM can vary significantly depending on the values of the learning parameters. For example, it is known that larger values of C lead to longer training times. Asynchronous parallel pattern search (APPS) is a parallel optimization approach that is well suited to such situations since it does not synchronize the system at the end of every single iteration. The cost for the good load balancing is some additional function evaluations in the parallel mode.

Results in [21] indicate a small number of additional function evaluations for multi-processor APPS. By contrast, our results in Table 3 for $\beta = 0.75$ show that the number of function evaluations in parallel mode can be significantly larger than in serial mode so that the efficiency is reduced. Usage of 8 processors led to 80 function evaluations, which is 60% more than with the serial version. However, usage of 8, 16 or 32 CPUs decreases overall running time of SVM parameter tuning.

During the tests we observed an increasing number of workers without a job, i.e., more and more processors did no longer receive trial points for function

Table 3. Overhead for parallel optimization

| mode | serial | 7 workers | 15 workers | 31 workers |
|----------------------------|--------|-----------|------------|------------|
| function evaluations | 49 | 80 | 62 | 62 |
| E_β | 0.102 | 0.101 | 0.102 | 0.102 |
| training errors | 50 | 50 | 51 | 51 |
| σ | 62.88 | 72.16 | 52.05 | 52.05 |
| C^+ | 69060 | 100000 | 100000 | 100000 |
| C^- | 13380 | 19560 | 19560 | 19560 |
| ratio C^+/C^- | 5.2 | 5.1 | 5.1 | 5.1 |
| false negative test points | 4 | 4 | 5 | 5 |
| false positive test points | 74 | 74 | 68 | 68 |

evaluations. This is due to caching effects and the decreasing number of new trial points during the final steps. Thus the asynchronous scheme cannot sustain a large degree of parallelism when the system is near convergence.

The optimization results in terms of accuracy, however, depend only slightly on the number of processors. The number of training errors is nearly the same for all tests. Misclassifications in the test set differ only a little bit and the values of our quality measure are nearly equal for all tests. Note that the parallel mode yields larger values for C^+ and C^- , whereas their ratio remains almost constant. This is a very interesting detail and gives evidence to the assumption that the ratio C^+/C^- should always be considered, too. The most significant differences between serial and parallel optimization with *APPSPACK* can be seen in the σ values. They differ in both directions up to 20%.

Please note that SVM training for a fixed set of parameter values can be formulated as a global optimization problem with a single optimum, but the task of parameter tuning might lead to a large number of local minima. Since we are interested in robust methods for SVM parameter tuning it might be interesting to analyze in future tests *APPSPACK*'s ability to avoid local minima.

6 Conclusions and Future Directions

We have introduced an automated parameter optimization scheme for support vector learning. Our scheme is to a wide degree portable and can be adapted very easily. The *APPSPACK* software is freely available and runs on different platforms in serial and parallel mode. While we have used our own implementation of support vector learning, using publicly available SVM software is also possible; [14] might be a good choice. We have shown results for different quality measures, different models with varying numbers of parameters, and serial and parallel computing mode.

In the future we plan integrating different kernels into a single SVM model and a hierarchical parallelization combining parallel SVM training with parallel parameter optimization to speed up the model selection even more.

Acknowledgements

We would like to thank Tamara Kolda for continuous help with *APPSPACK*-related questions. We are grateful to Wolfgang Frings, Inge Gutheil, Ruth Zimmermann and the ZAM team at Juelich for technical support, several remarks and careful reading. We also would like to thank the unknown referees for their valuable comments.

References

1. Vapnik, V.N.: Statistical learning theory. Wiley & Sons, New York (1998)
2. Platt, J.: Fast training of support vector machines using sequential minimal optimization. In Schölkopf, B., Burges, C.J.C., Smola, A.J., eds.: *Advances in Kernel Methods — Support Vector Learning*, Cambridge, MA, MIT Press (1999) 185–208
3. Poulet, F.: Multi-way distributed SVM algorithms. In: *Proc. of ECML/PKDD 2003 Int. Workshop on Parallel and Distributed Algorithms for Data Mining*. (2003)
4. Cristianini, N., Shawe-Taylor, J.: *An Introduction to Support Vector Machines and Other Kernel-Based Learning Methods*. Cambridge University Press, Cambridge, UK (2000)
5. Schölkopf, B., Smola, A.J.: *Learning With Kernels*. MIT Press, Cambridge, MA (2002)
6. Hsu, C.W., Lin, C.J.: A simple decomposition method for support vector machines. *Machine Learning* **46** (2002) 291–314
7. Serafini, T., Zanghirati, G., Zanni, L.: Gradient projection methods for quadratic programs and applications in training support vector machines. *Optimization Methods and Software* **20** (2005) 353–378
8. Pardalos, P.M., Kuvorov, N.: An algorithm for a singly constrained class of quadratic programs subject to upper and lower bounds. *Mathematical Programming* **46** (1990) 321–328
9. Eitrich, T., Lang, B.: Efficient optimization of support vector machine learning parameters for unbalanced datasets. Preprint BUW-SC 2005/2, University of Wuppertal (2005)
10. Zanghirati, G., Zanni, L.: A parallel solver for large quadratic programs in training support vector machines. *Parallel Computing* **29** (2003) 535–551
11. Collobert, R., Bengio, S., Bengio, Y.: A parallel mixture of SVMs for very large scale problems. *Neural Computation* **14** (2002) 1105–1114
12. Selikoff, S.: The SVM-tree algorithm (2003) http://scott.selikoff.net/papers/CS678-_Final_Report.pdf.
13. Celis, S., Musicant, D.R.: Weka-parallel: machine learning in parallel. Computer Science Technical Report 2002b, Carleton College (2002)
14. Chang, C.C., Lin, C.J.: LIBSVM: a library for support vector machines. (2001) Software available at <http://www.csie.ntu.edu.tw/~cjlin/libsvm>.
15. Gray, G.A., Kolda, T.G.: APPSPACK 4.0: asynchronous parallel pattern search for derivative-free optimization. Sandia Report SAND2004-6391, Sandia National Laboratories, Livermore, CA (2004)
16. Hettich, S., Blake, C.L., Merz, C.J.: UCI Repository of machine learning databases. (1998) <http://www.ics.uci.edu/~mllearn/MLRepository.html>.
17. Schiffmann, W., Joost, M., Werner, R.: Synthesis and performance analysis of multilayer neural network architectures. Technical Report 16/1992, University of Koblenz (1992)

18. Inoue, T., Abe, S.: Fuzzy support vector machines for pattern classification. In: Proc. Intl. Joint Conf. Neural Networks (IJCNN'01). (2001) 1449–1454
19. Detert, U.: Introduction to the JUMP architecture. (2004) <http://jumpdoc.fz-juelich.de>.
20. Markowetz, F.: Support vector machines in bioinformatics. Master's thesis, University of Heidelberg (2001)
21. Hough, P.D., Kolda, T.G., Torczon, V.J.: Asynchronous parallel pattern search for nonlinear optimization. *SIAM Journal on Scientific Computing* **23** (2001) 134–156