Efficient Optimization of Multi-class Support Vector Machines with MSVMpack

Emmanuel Didiot, Fabien Lauer

Efficient Optimization of Multi-class Support Vector Machines with MSVMpack

Emmanuel Didiot and Fabien Lauer

LORIA, Université de Lorraine, CNRS, Inria,
Nancy, France

Abstract. In the field of machine learning, multi-class support vector machines (M-SVMs) are state-of-the-art classifiers with training algorithms that amount to convex quadratic programs. However, solving these quadratic programs in practice is a complex task that typically cannot be assigned to a general purpose solver. The paper describes the main features of an efficient solver for M-SVMs, as implemented in the MSVMpack software. The latest additions to this software are also highlighted and a few numerical experiments are presented to assess its efficiency.

Keywords: Quadratic programming, classification, support vector machines, parallel computing

1 Introduction

The support vector machine (SVM) [2, 14] is a state-of-the-art tool for binary classification. While there is mostly one main algorithm for binary SVMs, their multi-class counterparts have followed various development paths. As a result, four main types of multi-class support vector machines (M-SVMs) can be found in the literature [15, 4, 11, 8] while others are still developed.

From the optimization point of view, training an M-SVM amounts to solving a convex quadratic program with particular difficulties that make the task unsuitable for a general purpose solver. The paper describes how to deal with such difficulties and how to obtain an efficient implementation, in particular regarding parallel computing issues.

The described implementation was first released in 2011 as an open-source software, MSVMpack (available at `http://www.loria/~lauer/MSVMpack/`) [10], which remains, to the best of our knowledge, the only parallel software for M-SVMs. We here present the inner details of this implementation and its latest improvements.

Paper Organization. M-SVMs are introduced in Section 2 with their training algorithm. Then, Section 3 details their efficient optimization, while the latest additions to MSVMpack are exposed in Section 4. Finally, numerical experiments are reported in Section 5 and Section 6 highlights possible directions for further improvements.

2 Multi-class Support Vector Machines

We consider Q-category classification problems, where labels $Y \in \{1, \ldots, Q\}$ are assigned to feature vectors $X \in \mathbb{R}^d$ via a relationship assumed to be of probabilistic nature, i.e., X and Y and random variables with an unknown joint probability distribution. Given a training set, $\{(x_i, y_i)\}_{i=1}^m$, considered as a realization of m independent copies of the pair (X, Y), the aim is to learn a classifier $f : \mathbb{R}^d \to \{1, \ldots, Q\}$ minimizing the risk defined as the expectation of the 0–1 loss returning 1 for misclassifications (when $f(X) \neq Y$) and 0 for correct classifications (when $f(X) = Y$).

Given a feature vector $x \in \mathbb{R}^d$, the output of an M-SVM classifier is computed as

$$f(x) = \arg \max_{k \in \{1,\ldots,Q\}} h_k(x) + b_k,$$

where the $b_k \in \mathbb{R}$ are bias terms and all component functions h_k belong to some reproducing kernel Hilbert space with the positive-definite function $K : \mathbb{R}^d \times \mathbb{R}^d \to \mathbb{R}$ as reproducing kernel [1]. As an example, the kernel function is typically chosen as the Gaussian RBF kernel function $K(x, x') = \exp\left(-\|x - x'\|^2 / 2\sigma^2\right)$.

The training algorithm for an M-SVM depends on its type, considered here as one of the four main types available in the literature, but can always be formulated as a convex quadratic program. More precisely, the dual form of the quadratic program is usually considered:

$$\max_{\alpha \in \mathcal{C}} \; J_d(\alpha) = -\frac{1}{2}\alpha^T \boldsymbol{H} \alpha + c^T \alpha, \tag{1}$$

where $\alpha \in \mathbb{R}^{Qm}$ is the vector of dual variables, $\boldsymbol{H}$ and c depend on the type of M-SVM and the training data, and $\mathcal{C}$ is a convex set accounting for all linear constraints that also depend on the M-SVM type and the training parameters. Given the solution α^* to this quadratic program, the functions h_k and the bias terms b_k become available.

The four main types of M-SVMs are the ones of Weston and Watkins [15] (hereafter denoted WW), Crammer and Singer [4] (CS), Lee, Lin and Wahba [11] (LLW) and the M-SVM^2 of Guermeur and Monfrini [8] (MSVM2). We refer to these works for a detailed discussion of their features and the derivation of the corresponding dual programs (see also [7] for a generic M-SVM model unifying the four formulations and [9] for a comparison of a subset of them).

3 Efficient Optimization

The main algorithm used in MSVMpack for solving (1) is the Frank-Wolfe algorithm [6]. In this algorithm, each step is obtained via the solution of a linear program (LP), as depicted in the following procedure minimizing $-J_d(\alpha)$.

1. Initialize $t = 0, \alpha(0) = 0$.
2. Compute the gradient vector $g(t) = -\nabla J_d(\alpha(t))$.

3. Compute a feasible direction of descent as

$$u = \arg\max_{v \in \mathcal{C}} \; g(t)^T v. \qquad (LP)$$

4. Compute the step length $\lambda = \min \left\{ 1, \; \dfrac{g(t)^T (u - \alpha(t))}{(u - \alpha(t))^T \boldsymbol{H} (u - \alpha(t))} \right\}$.
5. Update $\alpha(t+1) = \alpha(t) + \lambda(u - \alpha(t))$.
6. $t \leftarrow t + 1$.
7. Compute $U(t)$ and repeat from Step 2 until $J_d(\alpha(t))/U(t) > 1 - \epsilon$.

Checking the Kuhn-Tucker (KKT) optimality conditions during training may be too much time consuming for large data sets (large m). Thus, we measure the quality of $\alpha(t)$ thanks to the computation of an upper bound $U(t)$ on the optimum $J_d(\alpha^*)$. Given an $U(t)$ that converges towards this optimum, the stopping criterion is satisfied when the ratio $J_d(\alpha)/U(t)$ reaches a value close to 1. In MSVMpack, such a bound is obtained by solving the primal problem with the primal variables h_k fixed to values computed using the current $\alpha(t)$ instead of α^* in the formulas giving the optimal h_k's. This partial optimization requires little computation for all M-SVMs except the M-SVM^2, for which another simple quadratic program has to be solved.

3.1 Decomposition Method

Despite the quadratic programming form of (1), solving large-scale instances of such problems requires some care in practice. In particular, a major issue concerns the memory requirements: the Qm-by-Qm matrix $\boldsymbol{H}$ is dense as it is computed from the typically dense m-by-m kernel matrix $\boldsymbol{K}$, and thus it simply cannot be stored in the memory of most computers. This basic limitation prevents any subsequent call to a general purpose solver in many cases.

Dedicated optimization algorithms have been proposed to train SVMs (see, e.g., [3]) and they all use decomposition techniques, i.e., block-coordinate descent or chunking, such as sequential minimal optimization (SMO) [12, 13]. The basic idea is to optimize at each iteration only with respect to a subset of variables, here corresponding to a subset $\{(x_i, y_i) : i \in S\}$ of the data. With s the size of the subset $S = \{S_1, \ldots, S_s\}$ kept small, this makes training possible by requiring at each step only access to an s-by-m submatrix $\boldsymbol{K}_S$ of $\boldsymbol{K}$ with entries given by

$$(\boldsymbol{K}_S)_{ij} = K(x_{S_i}, x_j), \quad i = 1, \ldots, s, \; j = 1, \ldots, m. \qquad (2)$$

From $\boldsymbol{K}_S$, all the information required to perform a training step can be computed with little effort.

Working Set Selection. Most implementations of SVM learning algorithms include a dedicated working set selection procedure that determines the subset S and the variables that will be optimized in the next training step. Such procedures are of particular importance in order to reduce the numbers of iterations

and of kernel evaluations required by the algorithm. In addition, these methods typically lead to convergence without having to optimize over all variables thanks to the sparsity of SVM solutions (given an initialization of α with zeros). In such cases, only a small subset of the kernel matrix need to be computed, thus also limiting the amount of memory required for kernel cache (see Sect. 3.2).

A working set selection strategy was proposed in [4] in order to choose the chunk of data considered in the next training step of the CS type of M-SVM. This strategy is based on the KKT conditions of optimality applied to the dual problem. The data points with maximal violation of the KKT conditions are selected. This violation can be measured for each data point (x_i, y_i) by

$$\psi_i = \max_{k \in \{j \,:\, \alpha_{ij}(t) > 0\}} g_{ik}(t) - \min_{k \in \{1,\dots,Q\}} g_{ik}(t), \tag{3}$$

where $g_{ik}(t)$ is the partial derivative of $J_d(\alpha(t))$ with respect to α_{ik}, the $(iQ+k)$th dual variable which is associated to the ith data pair (x_i, y_i) and the kth category.

Unfortunately, there is no simple measure of the violation of the KKT conditions for the other types of M-SVMs. Thus, for these, we use a random selection procedure for the determination of S. [1]

In the following, the subscript $_S$ is used to denote subvectors (or submatrices) with all entries (or rows) associated to the working set, e.g., $\alpha_S(t)$ contains all dual variables α_{ik}, with $i \in S$ and $1 \leq k \leq Q$.

3.2 Kernel Cache

Kernel function evaluations are the most demanding computations in the training of an M-SVM. Indeed, every evaluation of $K(x_i, x_j)$ typically involves at least $O(d)$ flops and computing the s-by-m submatrix $\boldsymbol{K}_S$ of $\boldsymbol{K}$ at each training step requires $O(smd)$ flops.

Many of these computations can be avoided by storing previously computed values in memory. More precisely, the kernel cache stores a number of rows of $\boldsymbol{K}$ that depends on the available memory. When the solver needs to access a row of $\boldsymbol{K}$, either the row is directly available or it is computed and stored in the kernel cache. If the cache memory size is not sufficient to store the entire kernel matrix, rows are dropped from the cache when new ones need to be stored. While most binary SVM solvers use a so-called "last recently used" (LRU) cache to decide which kernel matrix row should be dropped, MSVMpack does not apply a particular rule. The reason is that, for binary SVMs, efficient working set selection methods often lead to many iterations based on the same rows, while the random selection of MSVMpack does not imply such a behavior.

[1] The implementation of Weston and Watkins model as proposed in [9] for the BSVM software does include a working set selection strategy. However, this strategy relies on a modification of the optimization problem in which the bias terms, b_k, are also regularized. Here, we stick to the original form of the problem as proposed in [15] and thus cannot use a similar strategy.

3.3 Parallel Computations

MSVMpack offers a parallel implementation of the training algorithms of the four M-SVMs (WW, CS, LLW, MSVM2). This parallelization takes place at the level of a single computer to make use of multiple CPUs (or cores) rather than a distributed architecture across a cluster. The approach is based on a simple design with a rather coarse granularity, in which kernel computations and gradient updates are parallelized during training.

Precomputing the Kernel Submatrix. As previously emphasized, the kernel computations, i.e., the evaluation of $K(x_i, x_j)$, are the most intensive ones compared to training steps and updates of the model (updates of α). In addition, they only depend on the training data, which is constant throughout training. Thus, it is possible to compute multiple kernel function values at the same time or while a model update is performed. Technically, the following procedure is executed on N_{CPU} CPUs (the steps truly running in parallel are shown in italic).

1. *Select a working set* $S = \{S_1, \ldots, S_s\} \subset [\![1, m]\!]$ *of* s *data indexes.*
2. *Compute the kernel submatrix* $\boldsymbol{K}_S$ *as in* (2).
3. Wait for the M-SVM model to become available (in case another CPU is currently taking a step and updating the model).
4. Take a training step and update the model, i.e., compute u_S, λ and $\alpha_S(t+1) = \alpha_S(t) + \lambda(u - \alpha_S(t))$ as in the Frank-Wolfe algorithm depicted in Sect. 3.
5. Release the model (to unblock waiting CPUs).
6. Loop from step 1 until the termination criterion (discussed in Sect. 3) is satisfied.

Note that, for the CS model type, the working set selection is not random and actually makes use of the model (see section 3.1). In this case, it cannot be easily parallelized and the scheme above is modified to select the working set for the next training step between steps 4 and 5, while step 6 loops only from step 2.

With this procedure, a CPU can be blocked waiting in step 5 for at most $(N_{CPU} - 1)$ times the time of a training step. In most cases, this quantity is rather small compared to the time required to compute the kernel submatrix $\boldsymbol{K}_S$. In practice, this results in a high working/idle time ratio for all CPUs.

Computing the Gradient Update. The next most intensive computation after the kernel function evaluations concerns the gradient vector $g(t)$ that is required at each step.

First, note that a naive implementation computing the required subset of the gradient vector $g(t)$ before each step with $g_S(t) = \boldsymbol{H}_S \alpha(t) + c_S$, where $\boldsymbol{H}_S$ contains a subset of s rows of $\boldsymbol{H}$, is ineffective as it involves all the columns of $\boldsymbol{H}_S$. In addition, at every evaluation of the termination criterion, we require the entire gradient vector, which would involve the entire matrix $\boldsymbol{H}$.

A better strategy is to update the gradient at the end of each step with

$$g(t+1) = g(t) + \boldsymbol{H}(\alpha(t+1) - \alpha(t)),$$

where

$$\forall i \notin S,\ k \in [\![1, Q]\!],\ \alpha_{ik}(t+1) - \alpha_{ik}(t) = 0,$$

resulting in only $O(mQs)$ operations of the form:

$$\forall i \in [\![1, m]\!],\ k \in [\![1, Q]\!], \quad g_{ik}(t+1) = g_{ik}(t) - \sum_{j \in S} \sum_{1 \leq l \leq Q} H_{jl}(\alpha_{jl}(t+1) - \alpha_{jl}(t)).$$

This computation can be parallelized by splitting the gradient vector $g \in \mathbb{R}^{Qm}$ in $N_g \leq N_{CPU}$ parts, each one of which is updated by a separate CPU.

MSVMpack automatically balance the number N_K of CPUs precomputing kernel submatrices and the number N_g of those used for the gradient update along the training process. Indeed, at the beginning, the main work load is on the kernel matrix ($N_K = N_{CPU}$), but as the kernel cache grows, less and less computations are required to obtain a kernel submatrix. Thus, CPUs can be redistributed to the still demanding task of updating the gradient (N_g increases and $N_K = N_{CPU} - N_g + 1$). If the entire kernel matrix fits in memory, all CPUs eventually work for the gradient update.

Cross Validation. K-fold cross validation is a standard method to evaluate a classifier performance. After dividing the training set into N_{folds} subsets, the method trains a model on $(N_{folds} - 1)$ subsets, test it on the remaining subset and iterates this for all test subsets. Thus, the cross validation procedure requires training of N_{folds} models and can also be parallelized with two possible scenarios. In case the number of CPUs is larger than the number of models to be trained ($N_{CPU} > N_{folds}$), each one of the N_{folds} models is trained in parallel with N_{CPU}/N_{folds} CPUs applying the scheme described above to compute the kernel function values and gradient updates. Otherwise, the first N_{CPU} models are trained in parallel with a single CPU each and the next one starts training as soon as the first one is done, and so on. When all models are trained, the N_{folds} test subsets are classified sequentially, but nonetheless efficiently thanks to the parallelization of the classification described next.

The parallelized cross validation also benefits from the kernel cache: many kernel computations can be saved as the N_{folds} models are trained in parallel from overlapping subsets of data. More precisely, MSVMpack implements the following scheme. A *master* kernel cache stores complete rows of the global kernel matrix computed from the entire data set. For each one of the N_{folds} models, a *local* kernel cache stores rows of a kernel submatrix computed with respect to the corresponding training subset. When a model requests a row that is not currently in the *local* kernel cache, the request is forwarded to the *master* cache which returns the complete row with respect to the entire data set. Then, the elements of this row are mapped to a row of the *local* cache. This mapping discards the columns corresponding to test data for the considered model and takes care of the data permutation used to generate the training subsets. In this scheme, every kernel computation performed to train a particular model benefits to other models as well. In the case where the entire kernel matrix fits in memory, all values are computed only once to train the N_{folds} models.

Making Predictions on Large Test Sets. Finally, the classification of large test sets can also benefit from parallelization via a straightforward approach. The test set is cut into N_{CPU} subsets and the classifier output is computed on each subset by a different CPU.

3.4 Vectorized Kernel Functions

In many cases, data sets processed by machine learning algorithms contain real numbers with no more than 7 significant digits (which is often the default when writing real numbers to a text file) or even integer values. On the other hand, the typical machine learning software loads and processes these values as double precision floating-point data, which is a waste of both memory and computing time. MSVMpack can handle data in different formats and use a dedicated kernel function for each format. This may be used to increase the speed of kernel computations on single precision floats or integers.[2]

In particular, proper data alignment in memory allows MSVMpack to use vectorized implementations of kernel functions. These implementations make use of the Single Instruction Multiple Data (SIMD) instruction sets available on modern processors to process multiple components of large data vectors simultaneously. With this technique, the smallest the data format is (such as with single-precision floats or short integers), the faster the kernel function is.

4 New Features in Latest MSVMpack

Since its first version briefly described in [10], MSVMpack was extended with a few features. Notably, it is now available for Windows and offers a Matlab interface. It also better handles unbalanced data sets by allowing the use of a different value of the regularization parameter for each category.

The parallelized cross validation described in Section 3.3 is an important new feature (available since MSVMpack 1.4) that improves the computational efficiency in many practical cases. Also, the software defaults are now to use as much memory and as many processors as available (early releases required the user to set these values) and the dynamical balancing between CPUs assigned to the kernel matrix and those assigned to the gradient updates was also improved.

5 Numerical Experiments

An experimental comparison of MSVMpack with other implementations of M-SVMs is provided here for illustrative purposes. The aim is not to conclude on what type of M-SVM model is better, but rather to give an idea of the efficiency of the different implementations. The following implementations are considered:

[2] Note that, in a kernel evaluation $v = K(x, z)$, only the format of x and z changes and the resulting value v is always in double precision floating-point format. All other computations in the training algorithms also remain double-precision.

- J. Weston's own implementation of his M-SVM model (WW) in Matlab included in the Spider[3],
- the BSVM[4] implementation in C++ of a modified version of the same M-SVM model (obtained with the `-s 1` option of the BSVM software),
- K. Crammer's own implementation in C of his M-SVM model (CS) named MCSVM[5],
- and Lee's implementation in R of hers (LLW) named SMSVM[6].

Note that both the Spider and SMSVM are mostly based on non-compiled code. In addition, these two implementations require to store the kernel matrix in memory, which makes them inapplicable to large data sets. BSVM constitutes an efficient alternative for the WW model type. However, to the best of our knowledge, SMSVM is the only implementation (beside MSVMpack) of the LLW M-SVM model described in [11].

The characteristics of the data sets used in the experiments are given in Table 1. The ImageSeg data set is taken from the UCI repository[7]. The USPS_500 data set is a subset of the USPS data provided with the MCSVM software. The Bio data set is provided with MSVMpack. The CB513_01 data set corresponds to the first split of the 5-fold cross validation procedure on the entire CB513 data set [5]. MSVMpack uses the original MNIST data[8] with integer values in the range $[0, 255]$, while other implementations use scaled data downloaded from the LIBSVM homepage[9]. BSVM scaling tool is applied to the Letter data set downloaded from the UCI repository[10] to obtain a normalized data set in floating-point data format.

All methods use the same kernel functions and hyperparameters as summarized in Table 1. In particular, for MCSVM, we used $\beta = 1/C$. When unspecified, the kernel hyperparameters are set to MSVMpack defaults. Also, SMSVM requires $\lambda = \log_2(1/C)$ with larger values of C to reach a similar training error, so $C = 100000$ is used. Other low level parameters (such as the size of the working set) are kept to each software defaults. These experiments are performed on a computer with 2 Xeon processors (with 4 cores each) at 2.53 GHz and 32 GB of memory running Linux (64-bit).

The results are shown in Tables 2–3. Though no definitive conclusion can be drawn from this small set of experiments, the following is often observed in practice:

[3] `http://people.kyb.tuebingen.mpg.de/spider/`
[4] `http://www.csie.ntu.edu.tw/~cjlin/bsvm/`
[5] `http://www.cis.upenn.edu/~crammer/code-index.html`
[6] `http://www.stat.osu.edu/~yklee/software.html`
[7] `http://archive.ics.uci.edu/ml/datasets/Image+Segmentation`
[8] MNIST data sets are available at `http://yann.lecun.com/exdb/mnist/`. The data are originally stored as bytes, but the current implementation of the RBF kernel function in MSVMpack is faster with short integers than with bytes, so short integers are used in this experiment.
[9] `http://www.csie.ntu.edu.tw/~cjlin/libsvmtools/datasets/`
[10] `http://archive.ics.uci.edu/ml/datasets/Letter+Recognition`

- *Test error rates of different implementations of the same M-SVM model are comparable.* However, slight differences can be observed due to different choices for the stopping criterion or the default tolerance on the accuracy of the optimization.
- *Training is slower with* MSVMpack *than with other implementations for small data sets (with 15000 data or fewer).* This can be explained by better working set selection and shrinking procedures found in other implementations. For small data sets, these techniques allow other implementations to limit the computational burden related to the kernel matrix. Training with these implementations often converges with only few kernel function evaluations.
- *Training is faster with* MSVMpack *than with other implementations for large data sets (with 60000 data or more).* For large data sets, the true benefits of parallel computing, large kernel cache and vectorized kernel functions as implemented in MSVMpack become apparent. In particular, kernel function evaluations are not the limiting factor for MSVMpack in these experiments with this hardware configuration.
- *Embedded cross validation is fast.* Table 3 shows that a 5-fold cross validation can be faster with the parallel implementation that saves many kernel computations by sharing the kernel function values across all folds.
- MSVMpack *training is less efficient on data sets with a large number of categories (such as* Letter *with* $Q = 26$*).* However, for data sets with up to 10 categories (such as MNIST), MSVMpack compares favorably with other implementations.
- *Predicting the labels of a data set in test is faster with* MSVMpack *than with other implementations (always true in these experiments).* This is mostly due to the parallel implementation of the prediction. For particular data sets, data format-specific and vectorized kernel functions also speed up the testing phase.
- MSVMpack *leads to models with more support vectors than other implementations.* This might be explained by the fact that no tolerance on the value of a parameter α_{ik} is used to determine if it is zero or not (and if the corresponding example x_i is counted as a support vector).

The ability of MSVMpack to compensate for the lack of cache memory by extra computations in parallel is illustrated by Table 4 for the WW model type (which uses a random working set selection and thus typically needs to compute the entire kernel matrix). In particular, for the Bio data set, changing the size of the cache memory has little influence on the training time. Note that this is also due to the nature of the data which can be processed very efficiently by the linear kernel function for bit data format. For the CB513_01 data set, very large cache sizes allow the training time to be divided by 2 or 3. However, for smaller cache sizes (below 8 GB) the actual cache size has little influence on the training time.

Table 1. Characteristics of the data sets.

Data set	**#classes**	**#training examples**	**#test examples**	**data dim.**	**data format**	**Training parameters**
ImageSeg	7	210	2100	19	double	$C = 10$, RBF kernel, $\sigma = 1$ data normalized
USPS_500	10	500	500	256	float	$C = 10$, RBF kernel, $\sigma = 1$
Bio	4	12471	12471	68	bit	$C = 0.2$, linear kernel
CB513_01	3	65210	18909	260	short	$C = 0.4$, RBF kernel
CB513	3	84119	5-fold CV	260	short	$C = 0.4$, RBF kernel
MNIST	10	60000	10000	784	short	$C = 1$, RBF kernel, $\sigma = 1000$
					double	data normalized, $\sigma = 4.08$
Letter	26	16000	4000	16	float	$C = 1$, RBF kernel, $\sigma = 1$ data normalized

6 Conclusions

The paper discussed the main features of an efficient solver for training M-SVMs as implemented in MSVMpack.

Future work will focus on implementing methods to ease the tuning of hyperparameters, such as the computation of regularization paths. Extending the software to deal with cost-sensitive learning will also be considered. Another possible direction of research concerns the parallelization in a distributed environment in order to benefit from computing clusters in addition to multi-cores architectures.

References

1. Berlinet, A., Thomas-Agnan, C.: Reproducing Kernel Hilbert Spaces in Probability and Statistics. Kluwer Academic Publishers, Boston (2004)
2. Boser, B., Guyon, I., Vapnik, V.: A training algorithm for optimal margin classifiers. In: Proc. of the 5th Annual Workshop on Computational Learning Theory. pp. 144–152 (1992)
3. Bottou, L., Chapelle, O., DeCoste, D., Weston, J. (eds.): Large-Scale Kernel Machines. The MIT Press, Cambridge, MA (2007)
4. Crammer, K., Singer, Y.: On the algorithmic implementation of multiclass kernel-based vector machines. Journal of Machine Learning Research 2, 265–292 (2001)
5. Cuff, J.A., J., B.G.: Evaluation and improvement of multiple sequence methods for protein secondary structure prediction. Proteins 34(4), 508–519 (1999)
6. Frank, M., Wolfe, P.: An algorithm for quadratic programming. Naval Research Logistics Quarterly 3(1–2), 95–110 (1956)
7. Guermeur, Y.: A generic model of multi-class support vector machine. International Journal of Intelligent Information and Database Systems 6(6), 555–577 (2012)
8. Guermeur, Y., Monfrini, E.: A quadratic loss multi-class SVM for which a radius-margin bound applies. Informatica 22(1), 73–96 (2011)

Table 2. Relative performance of different M-SVM implementations.

Data set	**M-SVM model**	**Software**	**#SV**	**Training error (%)**	**Test error (%)**	**Training time**	**Testing time**
ImageSeg	WW	Spider	113	5.24	11.05	11s	3.3s
		BSVM	98	2.38	9.81	0.1s	0.1s
		MSVMpack	192	0	10.33	1.2s	0.1s
	CS	MCSVM	97	2.86	8.86	0.1s	0.1s
		MSVMpack	141	0	9.24	3.3s	0.1s
	LLW	SMSVM	210	0.48	7.67	15s	0.1s
		MSVMpack	182	0	10.57	23s	0.1s
	$MSVM^2$	MSVMpack	199	0	10.48	4.5s	0.1s
USPS_500	WW	Spider	303	0.40	10.20	4m 19s	0.2s
		BSVM	170	0	10.00	0.2s	0.1s
		MSVMpack	385	0	10.40	2.5s	0.1s
	CS	MCSVM	284	0	9.80	0.5s	0.3s
		MSVMpack	292	0	9.80	30s	0.1s
	LLW	SMSVM	500	0	12.00	5m 58s	0.1s
		MSVMpack	494	0	11.40	1m 22s	0.1s
	$MSVM^2$	MSVMpack	500	0	12.00	22s	0.1s
Bio	WW	Spider	out	of	memory		
		BSVM	2200	6.23	6.23	11s	4.6s
		MSVMpack	4301	6.23	6.23	4m 00s	0.5s
	CS	MCSVM	1727	6.23	6.23	9s	4.2s
		MSVMpack	3647	6.23	6.23	8s	0.5s
	LLW	SMSVM	out	of	memory		
		MSVMpack	12467	6.23	6.23	2m 05s	1.1s
	$MSVM^2$	MSVMpack	12471	6.23	6.23	11m 44s	0.9s
CB513_01	WW	Spider	out	of	memory		
		BSVM	39183	19.46	25.70	1h 41m 52s	9m 02s
		MSVMpack	42277	16.94	25.55	10m 31s	26s
	CS	MCSVM	41401	17.07	25.45	4h 12m 35s	27m 11s
		MSVMpack	40198	16.93	25.41	4m 04s	32s
	LLW	SMSVM	out	of	memory		
		MSVMpack	54313	22.14	27.65	11m 46s	32s
	$MSVM^2$	MSVMpack	62027	14.31	25.32	1h 08m 54s	47s
MNIST	WW	Spider	out	of	memory		
		BSVM	13572	0.078	1.46	4h 03m 29s	2m 39s
		MSVMpack	14771	0.015	1.41	2h 50m 29s	15s
	CS	MCSVM	13805	0.038	2.76	1h 54m 26s	4m 09s
		MSVMpack	14408	0.032	1.44	49m 04s	15s
	LLW	SMSVM	out	of	memory		
		MSVMpack	47906	0.282	1.57	4h 36m 45s	45s
	$MSVM^2$	MSVMpack	53773	0.027	1.51	10h 55m 10s	55s
Letter	WW	Spider	out	of	memory		
		BSVM	7460	4.30	5.85	6m 20s	5s
		MSVMpack	7725	3.14	4.75	24m 38s	3s
	CS	MCSVM	6310	2.03	3.90	2m 38s	4s
		MSVMpack	7566	4.72	6.92	6m 54s	3s
	LLW	SMSVM	out	of	memory		
		MSVMpack	16000	16.90	18.80	3h 56m 08s	4s
	$MSVM^2$	MSVMpack	16000	5.08	7.28	*48h 00m 00s	3s

*: manually stopped before reaching the default optimization accuracy.

Table 3. Results of a 5-fold cross validation on the CB513 data set. The times for training and testing on a single fold are also recalled to emphasize the benefit of the parallel cross validation.

M-SVM	**Cross validation**		**Single fold**	
model	**error**	**time**	**training time**	**test time**
WW	23.63 %	21m 07s	10m 31s	26s
CS	23.55 %	12m 05s	4m 04s	32s
LLW	25.52 %	30m 29s	11m 46s	32s
$MSVM^2$	23.36 %	2h 55m 05s	1h 08m 54s	47s

Table 4. Effect of the cache memory size on the training time of MSVMpack for the WW model type.

Bio

Cache memory size in MB	10	60	120	300	600	1200
and in % of the kernel matrix	$< 1\%$	5%	10%	25%	50%	100%
Training time	5m 55s	6m 00s	6m 13s	4m 29s	3m 56s	4m 00s

CB513_01

Cache memory size in MB	1750	3500	8200	16500	24500	30000
and in % of the kernel matrix	5%	10%	25%	50%	75%	92%
Training time	31m 49s	30m 40s	26m 06s	23m 00s	15m 52s	10m 31s

9. Hsu, C.W., Lin, C.J.: A comparison of methods for multi-class support vector machines. IEEE Transactions on Neural Networks 13(2), 415–425 (2002)
10. Lauer, F., Guermeur, Y.: MSVMpack: a multi-class support vector machine package. Journal of Machine Learning Research 12, 2269–2272 (2011), `http://www.loria.fr/~lauer/MSVMpack`
11. Lee, Y., Lin, Y., Wahba, G.: Multicategory support vector machines: Theory and application to the classification of microarray data and satellite radiance data. Journal of the American Statistical Association 99(465), 67–81 (2004)
12. Platt, J.: Fast training of support vector machines using sequential minimal optimization. In: Schölkopf, B., Burges, C., Smola, A. (eds.) Advances in Kernel Methods: Support Vector Learning, pp. 185–208. MIT Press (1999)
13. Shevade, S., Keerthi, S., Bhattacharyya, C., Murthy, K.: Improvements to the SMO algorithm for SVM regression. IEEE Transactions on Neural Networks 11(5), 1188–1193 (2000)
14. Vapnik, V.N.: The Nature of Statistical Learning Theory. Springer-Verlag New York, Inc. (1995)
15. Weston, J., Watkins, C.: Multi-class support vector machines. Tech. Rep. CSD-TR-98-04, Royal Holloway, University of London (1998)