MSVMpack: a Multi-Class Support Vector Machine package

—

`http://www.loria.fr/~lauer/MSVMpack`

—

Documentation of Version 1.5

F. Lauer

July 3, 2014

Contents

1 Install and setup

MSVMpack can be installed on either Linux, Mac OS X or Windows. Another alternative is to use MSVMpack through the platform-independent web interface (see section 2.6).

1.1 Linux and Mac OS X

To install MSVMpack, simply follow the instructions below.

1. Download and extract the archive:
   `unzip MSVMpack1.5.zip`
   (this should create a directory named `MSVMpack1.5`)
2. Go to this directory and build the programs:
   `cd MSVMpack1.5`
   `make`
   (section 3.1 gives a description of additional options for this step)
3. For a system-wide install of the software type:
   `make install`
   (this step requires root privileges)

This should build the MSVMpack library and generate three command-line tools:

- `trainmsvm`: used to train an M-SVM,
- `predmsvm`: used to make predictions with a trained M-SVM,
- `msvmserver`: the web server.

See the `README` file for a full description of the contents of the `MSVMpack1.5` directory.

Note for Mac OS X users. The developer tools *with the optional command-line tools* must be installed on the system in order to build MSVMpack. These tools can be found on the installation disk of Mac OS X or on Apple website.

1.2 Windows

Precompiled executables are provided in the `MSVMpack1.5\Windows\bin` directory. To use these and run MSVMpack, follow the instructions below.

1. Download and extract the archive in a directory of your choice, say `DIR`
2. Open an MS-DOS window: Start → Execute... → `cmd`
3. Go to the right directory:
   `cd DIR`
4. Run the setup program:
   `MSVMpack1.5\Windows\setPath.bat`
5. For a 32 bits environment, additionnally run :
   `MSVMpack1.5\Windows\Utils\32bits\setup32bits.bat`

6. Close the MS-DOS window and open a new one for the setup to take effect.

Then, you can use `trainmsvm.exe` and `predmsvm.exe` as explained in the following sections. In order to benefit from special optimization options or include your own custom kernel functions, you need to recompile MSVMpack. How to build MSVMpack on Windows is detailed in Section 3.2.

2 User's guide

This section describes the main features of the 3 interfaces provided with the library: the web interface, the command-line tools and the matlab interface. The complete list of command-line parameters can be obtained by calling the tools without arguments, *e.g.*, `trainmsvm`. See section 4 for more information on technical features that may help to deal with large data sets.

2.1 Quick start with the web interface

To start the MSVMpack Server, simply issue the command

```
msvmserver start
```

The web interface can now be accessed locally with a browser by entering the following address (see section 2.5 for other options):

```
http://127.0.0.1:8080/
```

At the first connection, you will be asked to setup the server from the admin page to which you can login with the `admin` user and default password (`admin`). You can safely click the "Save settings and start using MSVMpack server" on this page for now (these settings can be changed later) and go to the HOME page from where you can train and test the M-SVM models.

In the "Train a model" section, choose a data file from the list, *e.g.*, `iris.train` (the number of instances and features should then appear to the right of the list). Choose one of the four M-SVM algorithms, a kernel function, *e.g.*, the Gaussian RBF, and click the "Start training" button at the bottom. The complete command line issued to produce the result is printed and you can click on "WATCH" to see the output of the algorithm.

When training is completed you can see the resulting model, *e.g.*, `msvm.model` (right-click on the `msvm.model` file and *save link as...* to download), and go back to the home page.

To test the new model, scroll down to the "TEST" section of the HOME page and choose a data file, *e.g.*, `iris.test`, before clicking the "Test" button. The recognition rate with some additional statistics will be printed on the next page and you can look at the output of the classifier for each data point in the "output file".

2.2 Quick start with the command-line tools

To train an M-SVM with default parameters, simply use the following command[1] (assuming `doc/` is the current directory in which `myTrainingData` can be found):

```
trainmsvm myTrainingData
```

This command trains an M-SVM[2] model as described in [5] (see the next section and Table 1 for alternative M-SVM models). At the end of training, the resulting M-SVM is stored by default in the file `msvm.model`. Note that the program assumes that the labels are given in the last column of the file `myTrainingData` and that they take their values in the set $[\![1, Q]\!]$, where Q is the number of classes (see Sect. 3.9 for a full description of the file format).

The following example shows how to set the parameters otherwise than to the default values:

```
trainmsvm myTrainingData myMSVM.model -m WW -c 3.0 -k 2 -p 2.5
```

When a `.model` file is specified on the command line, it is used to store the resulting M-SVM model. In this example, the model of Weston and Watkins (WW) [11] is considered, the soft-margin parameter[2] C is set to 3 and the kernel type is set to 2 (Gaussian RBF) with a kernel

[1] On some systems, you may be required to specifiy the path with for instance `./trainmsvm` .

[2] Note that MSVMpack implements the original formulation of each M-SVM, in which the complexity of the machine is tuned through a soft-margin parameter C instead of a regularization constant λ as described in [7].

parameter value of 2.5 (corresponding here to the standard deviation). Use `trainmsvm` without argument to see the full list of parameters and their default values.

Once training is done, you can test the classifier on another data set by using `predmsvm` as:

```
predmsvm myTestData myMSVM.model pred.outputs
```

where `myTestData` is the file containing the test data and `myMSVM.model` is the file of the trained M-SVM (assumed to be `msvm.model` if omitted). On output, the file `pred.outputs` will contain, for each test point, the computed output values for the Q classes and the predicted label. If the data file `myTestData` additionnally contains the labels of the test examples in the last column, the test error and the confusion matrix are automatically computed.
Note: the training and test files must be specified as the first argument to `trainmsvm` and `predmsvm`, respectively.

2.3 `trainmsvm` usage

The default behavior of `trainmsvm` is to keep training until a predefined level of accuracy is reached in terms of the ratio

$$R = \frac{\text{value of the dual objective function}}{\text{upper bound on the optimum}}, \tag{1}$$

thus defining the stopping criterion as $R \geqslant 1 - \varepsilon$ (see [7] for details on the computation of the upper bound). The value of the accuracy level $1 - \varepsilon$ can be set through the option `-a`, *e.g.*, for an accuracy level of 95%:

```
trainmsvm myData myMSVM.model -a 0.95
```

In practice, the ratio is only computed every 1000 iterations on the basis of the best upper bound, *i.e.*, the smallest value of the upper bound obtained till that point. Each one of these values corresponds to the estimate of the primal objective function computed at the current iteration. In addition to these periodic evaluations, the user has the possibility to force the evaluation on demand with the key shortcut `ctrl-C`. This extra evaluation also prints some additional information and asks the user if he wants to stop training or continue.

The following describes some additional features of the `trainmsvm` tool.

Infinite training. Setting the accuracy level to zero with `-a 0` disables the evaluation of the model during training and the computation of the ratio (1), which leads to slightly faster training. However, in this case, training must be explicitly stopped by the user through the key shortcut `ctrl-C`.

M-SVM model type. The four M-SVMs found in the literature are implemented in MSVMpack. The `-m` flag allows one to choose the type of M-SVM model according to Table 1.

Table 1: Model types of the four M-SVMs.

Option flag	M-SVM model type name & reference
`-m WW`	Weston and Watkins [11]
`-m CS`	Crammer and Singer [2]
`-m LLW`	Lee, Lin, and Wahba [8]
`-m MSVM2` (default)	Guermeur and Monfrini [5]

Kernel function. The `-k` flag is used to choose the kernel function according to Table 2, while `-p` *value* is used to set the value of the kernel parameter.

In addition to the standard kernel functions, MSVMpack allows one to easily add up to three custom kernels as detailed in section 3.3. These kernels can be used by invoking `trainmsvm` with `-k 5` for custom kernel 1, `-k 6` for custom kernel 2 or `-k 7` for custom kernel 3.

Table 2: Kernel functions.

Option flag	Kernel function	Default parameter
`-k 1` (default)	Linear $k(x,z) = \langle x, z\rangle$	
`-k 2`	Gaussian RBF $k(x,z) = \exp(\frac{-\|x-z\|_2^2}{2\sigma^2})$	$\sigma = \sqrt{5 \times dim(x)}$
`-k 3`	Homogeneous polynomial $k(x,z) = (\langle x, z\rangle)^d$	$d = 2$
`-k 4`	Non-homogeneous polynomial $k(x,z) = (1 + \langle x, z\rangle)^d$	$d = 2$
`-k 5, -k 6, -k 7`	Custom kernels	0

Multiple kernel parameters. For kernel functions that take multiple kernel parameters, their values are set by `-P` *#parameters value1 value2 value3...* (with a capital `P`). Alternatively, `-P filename` or `-P` *#parameters* `filename` can be used to read the parameter values from the file `filename` containing either the number of parameters followed by their values or only the parameter values, respectively. Note that `filename` cannot start with a number (*i.e.*, `01mykernel_par` is wrong but `mykernel_par01` or `mykernel_par.01` are correct). See section 3.9 for examples.

Class-dependent values of C for unbalanced data sets. For unbalanced data sets, using a different value of the hyperparameter C for each class can be helpful. The default behavior of the `-c` option is to assign the same value of C to every class. However, with the option `-C` (with a capital C), the user can set the values of C_k, for all $k \in [\![1, Q]\!]$, as for instance in

```
trainmsvm myTrainingData myMSVM.model -C 1 2.2 0.3
```

where $C_1 = 1$, $C_2 = 2.2$ and $C_3 = 0.3$.

Cross validation. MSVMpack implements k-fold cross validation by first computing a random permutation of the data instances and then dividing the data set in k subsets of equal size (note that one of the subsets can be slightly larger in the case where the number of data is not a multiple of k). This random permutation is made to ease the obtention of subsets that contain examples of all categories in the typical case where the data set is sorted with respect to the class labels. To compute the `k`-fold estimate of the error rate, add the option `-cv k` to `trainmsvm` command line, e.g.,

```
trainmsvm myTrainingData -m CS -c 7.0 -cv 5
```

for a 5-fold cross validation, which results in the following output:

```
================== Cross validation summary ========================
 Fold    Training error rate     Test error rate
  1      6.62 %                  7.00 %
  2      5.88 %                  8.50 %
  3      6.25 %                  3.50 %
  4      7.00 %                  6.50 %
  5      6.12 %                  7.50 %

====== 5-fold cross validation error rate estimate = 6.60 % =======
```

During the procedure the models trained on data subsets and their outputs are stored in the files `cv.fold-XX.model` and `cv.fold-XX.outputs`, where `XX` stands for the index of the corresponding data subset.

Optimization method. The optimization method used to train an M-SVM can be chosen with the `-o` flag. The default optimization method (`-o 0`) in MSVMpack is the Frank–Wolfe method [4]. Rosen's gradient projection method [10] is also available through `-o 1` for the M-SVM^2 model type.

Data normalization. Data normalization in MSVMpack is embedded in the model, *i.e.*, the data files need not be changed and the data can retain their original (and usually more meaningful) scale. To normalize a data set before training, use the `-n` flag on the command line. Note that, even if this flag is missing, MSVMpack may recommend normalization in cases where large differences are detected between the scales of different features. The `-u` flag can be used to bypass this test and force MSVMpack to use unnormalized data.

Data format. MSVMpack can handle data in different formats and use a different kernel function for each format. This may be used for instance to increase the speed of kernel computations on single precision floats or integers. See section 4.4 for the details.

Automatic testing. Computing the test error of an M-SVM at the end of training can be done by simply appending a test data file to the command line, *i.e.*,

```
trainmsvm myTrainingData myMSVM.model myData.test
```

In this case, the corresponding filename (here, `myData.test`) must include the extension `.test`. If, in addition, another filename with the `.outputs` extension is specified, then the outputs of the M-SVM computed on the test set are stored in this file.

Initialization and saving options. By default, training starts with all $\alpha_{ik} = 0$ (except for the M-SVM of [2], in which case $\alpha_{iy_i} = C$), but an initialization file (with the `.init` extension) conforming to the matrix file format (see Sect. 3.9) for the α_{ik} can also be specified on the command line. In addition, the values of the α_{ik} at the end of training can also be extracted from the model and saved in a dedicated file (in matrix format) by specifying a filename with the `.alpha` extension on the command line. In this case, the command line could look like:

```
trainmsvm myTrainingData myMSVM.model myAlpha.init myAlpha.alpha \
          myTestData.test myPrediction.outputs -c 10.0 -k 2
```

The order in which the optional files are specified is free (but the extensions are not). However, the training data must always be the first argument.

How `.com` files can help. Another way to train an M-SVM with full control over the parameters is to use a command file with a `.com` extension (this extension is mandatory, but can be further suffixed as in `first.com.example`). A command file assumes the following particular format:

```
3                          --> Number of classes Q
2                          --> Nature of kernel function
10.0 10.0 10.0             --> Values of C (one for each class)
4                          --> Chunk (or working set) size
Data/iris.app              --> Training data file
Alpha/iris.init            --> File of initial values of alpha
Alpha/iris.alpha           --> File for saved alpha
```

Assuming `myComFile.com` is such a file, the corresponding M-SVM is trained by running:

```
trainmsvm myComFile.com myMSVM.model
```

A few example `.com` files are provided in the `Comfiles/` directory.
Note: all parameter values from a `.com` file can be overridden by a command-line option, *e.g.*,

```
trainmsvm myComFile.com myMSVM.model -c 10.0
```

will always use $C = 10$, irrespective of the contents of `myComFile.com`.

Retraining. To retrain an existing model, *i.e.*, resume training from where it was stopped, use the `-r` option. This can also be used to increase the accuracy level of a model in terms of the ratio (1). For instance, if `myMSVM.model` was trained with `-a 0.95`,

```
trainmsvm -r -a 0.99 myMSVM.model
```

further trains the model until it reaches an accuracy level of 99% (no need to respecify the training data or the parameter values). Note however that the upper bound in (1) is not saved in the model, so that training is restarted without a good upper bound on the optimum. Thus, lower values of the ratio R can be observed at the beginning of resumed training without implying a loss of training time. In such cases, the accuracy level is simply underestimated until a good upper bound is found.

Model sparse format. To save the model in sparse format (see Sect. 3.9), append the `-s` option to the command line. To convert an existing model to the sparse format, use

```
trainmsvm -S myMSVM.model
```

Note that sparse models cannot be retrained, since some of the training data are lost.

Log files. Information on the training process is logged to the `.log` file specified in the command line (if any). This piece of information can then be plotted through a call to `plotlog` (or `plotlog_msvm2` for an M-SVM^2 model), *e.g.*:

```
trainmsvm myData -m WW myTraining.log
plotlog myTraining.log myPlot.ps
```

In this example, optimization information is logged to `myTraining.log`, while an M-SVM model of WW type is trained with default parameters on `myData`. The recorded information (see Sect. 3.9 for details) is then plotted on screen and the plot is saved to `myPlot.ps` (if the `.ps` file is omitted, `myTraining.log.ps` is used). Note that `plotlog` requires `gnuplot` for plotting.

Model saved periodically. In case the program stops unexpectedly (e.g., due to a power failure) the model is periodically saved in the temporary file `msvm.model.tmp` (where `msvm.model` is the chosen filename for the model). If training stops normally, the trained model is saved in, e.g., `msvm.model` and the temporary file is removed. The period is set to 20 minutes (via the constant `MONITOR_PERIOD` in `libtrainMSVM.h`).

2.4 `predmsvm` usage

The additional features for `predmsvm` are the following.

- If the optional `.outputs` file is omitted, the outputs are displayed on screen instead of being written to a file.
- The `-t N` option can be used to set the number of processors used by `predmsvm` to `N` (the default behavior is to use all available processors).
- The `-i` option can be used to read a model file and print some information:

```
predmsvm -i myMSVM.model

msvm.model       : M-SVM^2 (MSVM2)
 version         : MSVMpack 1.2
 training set    : myTrainingData
 training error  : 3.20 %
 data format     : double precision floating-point (64-bit)
        Q        : 3
        kernel   : Gaussian RBF (sigma = 2.500000)
```

```
C_1     : 1.000
C_2     : 2.200
C_3     : 0.300
nSV     : 120
```

2.5 `msvmserver` usage

The MSVMpack Server can be started with the command line

```
msvmserver start
```

optionally followed by a port number (the default port is 8080, since port 80 requires root privileges). The following options are used to control the server launched from the current directory.

- `msvmserver stop` is used to stop a running server.
- `msvmserver restart` is used to restart a running server.
- `msvmserver status` tells if the server is running.

Warning: if started, the MSVMpack Server provides unauthenticated HTTP access to the computer, including the possibility for users to upload *any* file in the `Data/` subdirectory and to process this file with MSVMpack, which was not written with security in mind. This means that running the server on an open network is probably a bad idea.

Note that security issues remain the responsibility of the person running this software.

Setup of the server. Several options can be set via the ADMIN page (while the server is running) which can be accessed from the HOME page with the `admin` login (default password is `admin`).

The data and model file paths are relative to the `MSVMpack1.5/webpages/` directory. Their defaults place all data files in `MSVMpack1.5/webpages/Data/` and all the models trained via the web interface in `MSVMpack1.5/webpages/Models`.

You can also set the default amount of memory for kernel cache, the maximal number of processors (cores) that MSVMpack can use or change the `admin` password from here.

2.6 Web interface

The web interface offers a more intuitive way to use MSVMpack and set the training parameters without having to type any command line in a terminal. It is accessible from any platform (including Windows) with a classical (javascript capable) browser. The only requirement is to have a network access to a Linux computer with a running instance of the MSVMpack server (see section 2.5 above for server-side instructions). The HOME page is divided in three sections: upload of data files, training and test.

Data files. In order to train or test an M-SVM model on some data, you must first upload the corresponding data file to the server. This action can be performed from the HOME page, while data files can be deleted or downloaded from the ADMIN page.

Training. The HOME page allows one to choose an M-SVM algorithm and set all the parameters for its training before clicking the "Start training" button.

Model files and test. All models created with the web interface are saved in a dedicated directory (defined in the ADMIN page). All models found in this directory (and these only) can be used to make predictions from the web interface. In particular, the bottom of the HOME page allows one to choose a trained model and a test set to predict labels on this data.

2.7 Using MSVMpack from Matlab

MSVMpack can be used from Matlab since version 1.3. However, this interface is only implemented as a set of wrapper functions which call the command-line tools `trainmsvm` and `predmsvm`. Therefore, it cannot be used as a platform-independent alternative to MSVMpack.

To start using MSVMpack from Matlab, you must add the `matlab` subdirectory to Matlab's PATH:

```
addpath('..path_of_installation../MSVMpack1.5/matlab/');
```

Then, you can try the test program `example.m` which will run a simple example and show the basic usage of the commands. The Matlab interface offers the following five commands (for which we only show the basic usage here, the `help` function might provide more details).

- `trainmsvm`: training function returning a trained model via

  ```
  model = trainmsvm(X, Y, options)
  ```

  where `X` is an $N \times d$-matrix of N examples with d features each, `Y` is a column vector of N labels and `options` is a string containing the optional arguments to pass to the `trainmsvm` command line (see section 2.3 for details).

- `predmsvm`: prediction function returning a vector of predicted labels and a matrix of real-valued outputs via

  ```
  [labels, outputs] = predmsvm(model, X, Y)
  ```

  where `model` is a result of `trainmsvm`.

- `kfold`: computes the K-fold cross-validation error estimate via

  ```
  [cv_error, labels] = kfold(K, X, Y, options)
  ```

  where `X`, `Y` and `options` are defined as for `trainmsvm`. On output, `labels` contains the labels predicted for the whole training set, *i.e.*, for each data subset without information about this subset during training. Note that this is a sequential Matlab implementation and that using `trainmsvm` with the option `'-cv K'` might be faster.

- `loadmsvm`: model loading function returning a model from a model name via

  ```
  model = loadmsvm( 'modelname' )
  ```

  where the model name is either given explicitly, deduced from the training set name or set to the default `mymsvm` at training time (see `help trainmsvm`). This function can also be used to load a model trained by MSVMpack tools outside Matlab.

- `loaddata`: data loading function returning a data set via

  ```
  dataset = loaddata( 'datafile' )
  ```

  where `datafile` refers to a file in MSVMpack data format (see section 3.9).

All these functions implicitly work with data and model files created on the spot to interact with MSVMpack command-line tools. These files are created in the current directory and they also ensure that no data nor models are lost between Matlab sessions.

Models and data sets are stored as Matlab structures with self-explanatory fields. For instance a model is represented as:

```
model =

                name: 'mymsvm'
             version: 1.4000
                type: 'WW'
            longtype: 'Weston and Watkins (WW)'
                   Q: 3
         kernel_type: 1
      kernel_longtype: 'Linear'
        nb_kernel_par: 0
           kernel_par: []
    training_set_name: 'mymsvm.train'
       training_error: 0.2167
              nb_data: 300
            dim_input: 2
                    C: [3x1 double]
        normalization: []
                    b: [3x1 double]
                alpha: [300x3 double]
                    X: [300x2 double]
                    Y: [300x1 double]
             SVindex: [166x1 double]
```

and the value of α_{ik} can be accessed as `model.alpha(i,k)`.

Custom kernels. The Matlab interface does not provide the possibility to add custom kernels implemented as Matlab code. The kernel must be implemented in C as described in section 3.3 and used by passing the corresponding `'-k'` option to `trainmsvm`.

3 Developer's guide

This section describes some additional options for building MSVMpack and how to call the library from another C program.

3.1 `make` rules and options

The following rules and options are defined in the `Makefile`:

- `make` : default "build all" rule which uses the best SSE flag available on the computer.
- `make SSE=-msse2` : do not use SSE instruction sets above SSE2 (for increased compatibility across multiple computers).
- `make clean` : remove all object and executable files.
- `make debug` : compile with `-g -O0` flags for debugging purposes.
- `make profile` : allow one to analyze performance with `gprof`.

Note that, on Mac OS X, `make` must be replaced by `make -f Makefile.osx`. However, this is not required for the default rule, *i.e.*, `make` can be used for a simple installation of the software.

3.2 Building MSVMpack on Windows

To build MSVMpack on Windows, follow the instructions below.

1. Install Mingw64 by using the installer that is available in the `MSVMpack1.5\Windows\Utils` directory.
   (Please verify that you select 4.8.1 version, x64 architecture, posix threads and seh exception)
2. Add `"mingw64\bin"` path to the windows environment variable PATH.
   For example `PATH=...;C:\Program Files\mingw-builds\x64-4.8.1-posix-seh-rev5\mingw64\bin`.
   How to set the path and environment variables in Windows :
   (a) From the Desktop, right-click `My Computer` and click `Properties`.
   (b) Click `Advanced System Settings` link in the left column.
   (c) In the `System Properties` window click the `Environment Variables` button.
3. Open an MS-DOS window : Start → Execute... → `cmd`
4. Go to the right directory:
   `cd DIR\Windows` (`DIR` is the directory where the MSVMpack1.5 source files are located)
5. Use `makefile64.bat` to compile `trainmsvm.exe` and `predmsvm.exe`
   (use `makefile32.bat` in a 32 bits Windows environment)

Be careful, the new compiled programs will be placed in the `bin` folder and the original programs `trainmsvm.exe` and `predmsvm.exe` will be overwritten. To recover the original programs, you can use `setup32bits.bat` or `setup64bits.bat` in the `Utils\32bits` and `Utils\64bits` folders. It copies the original binaries located in `bin32` or `bin64` in the `bin` folder.

3.3 Custom kernels

Custom kernels can be easily added to MSVMpack by editing the file
`MSVMpack/src/custom_kernels.c`
Simply follow the instructions in this file to implement your own kernel functions. Then rebuild by running

```
make
```

to include them in MSVMpack. Note that this modifies your local copy of MSVMpack and that MSVM models do not include explicit kernel functions. This means that models trained with your local copy of MSVMpack can only be used by MSVMpack copies that include the same custom kernel functions.

3.4 Using the library

To use the MSVMpack library, you need to include the three main header files located in `MSVMpack/include/` as:

```
#include "libMSVM.h"        // Generic structure and function declarations
#include "libevalMSVM.h"    // Evaluation functions (also used during training)
#include "libtrainMSVM.h"   // Training functions (not required for predictions only)
```

and link with the `libmsvm.a` static library located in `MSVMpack/lib/`, *e.g.*,

```
DIR= path_to_MSVMpack_directory
LIB=-lmsvm -lm -ldl -lpthread
gcc myProgram.c -I$(DIR)/include -I$(DIR)/lp_solve_5.5 -L$(DIR)/lib $(LIB)
```

3.5 A simple example

The following example code (also available in `doc/example.c` with the corresponding `Makefile`) shows how to train an M-SVM model and use it to classify a test set.

```
#include "libMSVM.h"        // Generic structure and function declarations
#include "libtrainMSVM.h"   // Training functions (not required for predictions only)
#include "libevalMSVM.h"    // Evaluation functions (also used during training)

int main(void) {
    struct Model *model;                    // declare a model
    struct Data *training_set, *test_set;   // declare the data sets

    long status;                // for MSVM_train return code
    double accuracy = 0.98;     // desired accuracy level of 98%
    long chunk_size = 4;        // size of the chunk used for training
    int cache_memory = 0;       // Amount of cache memory (0 = max)
    int nprocs = 0;             // Number of available CPUs (0 = all)
    long *labels;

    model = MSVM_make_model(MSVM2); // Create an empty model with default parameters
    if(model == NULL) {
        printf("Error in model creation\n");
        exit(1);
    }

    // Change some parameters
    model->nature_kernel = RBF;
    printf("Changed the kernel type in model to Gaussian RBF.\n");

    MSVM_model_set_C(1.0, 3, model);
    printf("Initialized the hyperparameters C to %1.2lf for all %ld classes.\n",model->C[1],model->Q);
    model->C[2] = 2.0;
    model->C[3] = 3.0;
    printf("Changed hyperparameters C_2 to %1.2lf and C_3 to %1.2lf.\n",model->C[2],model->C[3]);

    printf("Loading the data... \n");
    // The data format can be either DOUBLE, FLOAT, INT, SHORT, BYTE, or BIT
    training_set = MSVM_make_dataset("myTrainingData",DOUBLE); // load the training data
    test_set = MSVM_make_dataset("myTestData",DOUBLE);         // load the test data
```

```
    printf("Calling MSVM_train()...\n");
    /* Train the model on the training_set
        with default initialization (first NULL)
        no periodic saving of alpha (second NULL)
        no log file (third NULL)
    */
    status = MSVM_train(model, training_set, chunk_size, accuracy,
                        cache_memory, nprocs, NULL, NULL, NULL);

    if(status >= 0) {
        printf("Training done without error, will now classify the test set.\n");

        /* Allocate the array for predicted labels
            (size should be (size of test set + 1) )
        */
        labels = (long *)malloc(sizeof(long) * (test_set->nb_data + 1));

        /* Classify the test set,
         store the predicted labels in 'labels'
         and print outputs on screen (because filename=NULL)
        */
        MSVM_classify_set(labels, test_set->X, test_set->y, test_set->nb_data,
                          NULL, model, nprocs);
    }
    else
        printf("Error in training.\n");

    // Free the memory
    free(labels);
    MSVM_delete_model(model);
    MSVM_delete_dataset(training_set);
    MSVM_delete_dataset(test_set);

    return 0;
}
```

For other example uses of the library, see `src/trainMSVM.c`.

3.6 Structures

The library provides the following structure (defined in `libMSVM.h`) to store an M-SVM model with all its parameters:

```
struct Model {
    float version;                              // MSVMpack model version
    enum MSVM_type {WW=0,CS=1,
                    LLW=2,MSVM2=3} type;        // Type of the M-SVM
    enum Algorithm {FrankWolfe=0,
                    Rosen=1} algorithm;         // optimization method
    long Q;                                     // number of classes
    char *training_set_name;                    // name of training set
    long nb_data;                               // number of SVs
    long dim_input;                             // dimension of a description x
    enum Datatype datatype;                     // type of data
    enum Kernel_type nature_kernel;             // type of kernel
    double *kernel_par;                         // kernel function parameters
    double *C;                                  // soft-margin parameters
    double training_error;                      // training error rate
    double ratio;                   // optimization accuracy
```

```
    long iter;                      // optimization iterations
    int crossvalidation;            // #fold in cross validation (0 otherwise)

    double **alpha;                 // (nb_data x Q) matrix of alpha
    double *partial_average_alpha;
    double sum_all_alpha;
    double *b_SVM;                  // vector b
    void **X;                       // support vectors
    long *y;                        // labels of the SVs
    double **normalization;         // means and std used to normalize data
    double **W;                     // weights of the linear model
                                    // (for linear kernel only)
    // Format-specific data storage
    double **X_double;
    float **X_float;
    int **X_int;
    short int **X_short;
    unsigned char **X_byte;

// Thread synchronization
pthread_mutex_t mutex;
};
```

where the `Kernel_type` enumeration is defined in `kernel.h` as

```
enum Kernel_type {
    // kernel functions for DOUBLE datatype
    LINEAR=1,    // Linear
    RBF=2,       // Gaussian RBF
    POLY_H=3,    // Homogeneous polynomial
    POLY=4,      // Non-homogeneous polynomial
    CUSTOM1=5,   // Custom kernels
    CUSTOM2=6,
    CUSTOM3=7,
    // kernels for FLOAT datatype
    LINEAR_FLOAT=11,
    RBF_FLOAT=12,
    POLY_H_FLOAT=13,
    POLY_FLOAT=14,
    CUSTOM1_FLOAT=15,
    CUSTOM2_FLOAT=16,
    CUSTOM3_FLOAT=17
    // kernels for INT datatype
    LINEAR_INT=21,
    ...
};
```

Another generic structure is used for data sets:

```
struct Data {
    char *name;    // name of the data set
    long nb_data;  // number of data
    long dim;      // dimension of the data
    void **X;      // matrix with the x_i as rows
    long *y;       // vector of labels y_i
    long Q;        // number of classes in data set
    double **X_double;
    float **X_float;
```

```
    int **X_int;
    short int **X_short;
    unsigned char **X_byte;
    enum Datatype datatype;
};
```

where `Datatype` is defined as `enum Datatype {DATATYPE_DOUBLE=1, DATATYPE_FLOAT=2, DATATYPE_INT=3, DATATYPE_SHORT=4, DATATYPE_BYTE=5, DATATYPE_BIT=6}`.

3.7 Function reference

All functions provided by MSVMpack have the prefix `MSVM_`. In particular, the following general purpose functions are defined in `libMSVM.h`:

```
/* Model handling functions */
struct Model *MSVM_make_model(enum MSVM_type type);
    Creates an empty model of a given type.

struct Model *MSVM_load_model(char *model_file);
    Loads a model from a file.

void MSVM_delete_model(struct Model *model);
    Deletes a model and frees the memory except for the data
    in model->X and model->y.

void MSVM_delete_model_with_data(struct Model *model);
    Deletes a model and frees all the memory including the SVs.

int MSVM_save_model(const struct Model *model, char *model_file);
    Saves a model to a .model file.

int MSVM_save_model_sparse(const struct Model *model, char *model_file);
    Saves a model to a .model file using the sparse format.

long MSVM_init_model(struct Model *model, char *com_file);
    Initializes the parameters of a model from a .com file.

void MSVM_model_set_C(double C, long Q, struct Model *model);
    Sets the values of C_k to C for k=1,..., Q.
    (this also sets the value of Q in the model)

/* Data set handling functions */
struct Data *MSVM_make_dataset(char *data_file, enum Datatype datatype);
    Loads a dataset in datatype format from a file
    or creates an empty Data structure if data_file is NULL.

void MSVM_delete_dataset(struct Data *dataset);
    Deletes a dataset and frees the memory.

double MSVM_normalize_data(struct Data *dataset, struct Model *model);
    Normalizes the columns of X in a dataset and returns the difference
    between the largest and smallest std before normalization.
```

The following training functions are defined in `libtrainMSVM.h`:

```
long MSVM_train(struct Model *model, struct Data *training_set,
    long chunk_size, const double accuracy, int cache_memory, int nprocs,
```

```
    char *alpha0_file, char *model_tmp_file, char *log_file);

    Trains an M-SVM model on the training_set with a given chunk_size
    by using cache_memory MB of memory and nprocs CPUs
    until the desired accuracy is reached.

double **MSVM_train_cv(struct Model *model, struct Data *training_set,
    int K, long chunk_size, const double accuracy, int cache_memory,
    int nprocs, char *log_file);

    Perform K-fold Cross-validation and returns
    a 2-by-(K+1) array of error_rates:

        error_rates[0] = [Overall training error rate,
                          training error_rate on fold 1,
                          ...,
                          training error_rate on fold K]

        error_rates[1] = [Cross-validation estimate of the error,
                          test error_rate on fold 1,
                          ...,
                          test error_rate on fold K]

        So the cv error estimate is in error_rates[1][0].


long MSVM_init_train_comfile(struct Model *model, char *com_file,
    char *training_file, char *alpha0_file, char *alpha_file, char *log_file);

    Initializes the parameters and filenames used for training an M-SVM
    from a .com file.
```

The source file `libtrainMSVM.c` also contains the internal code for handling the kernel cache (see section 4.3).

The following evaluation functions are provided in `libevalMSVM.h`:

```
long MSVM_classify(void *x, const struct Model *model, double *real_outputs)

    Computes the label of x WITHOUT normalizing the data.
     (use MSVM_classify_set() to include normalization)
    If real_outputs is not NULL, also provide the Q real-valued outputs
    of the model in (real_outputs[1]...real_outputs[Q]).

void MSVM_classify_set(long *labels, void **X, long *y, long m, char *outputs_file,
    const struct Model *model, const int nprocs)

    Computes the predicted labels of an M-SVM for a set of m data points X
    in parallel over nprocs CPUs.
    Also computes the test error and some statistics if y is not NULL.
    Resulting output is saved into outputs_file or printed on screen
    only if outputs_file is NULL.
    If labels is NULL, the predicted labels are not stored in memory.
    Note: this function takes care of data normalization if needed
     (X should not be normalized).

double MSVM_eval(double *best_primal_upper_bound, double **gradient, double **H_alpha,
    double **H_tilde_alpha, struct Model *model, const int verbose, FILE *fp)
```

```
Evaluates the ratio between the value of the dual objective function and
the upper bound on the optimum.
```

These training and evaluation functions are wrappers that are used to call the proper function depending on the model type (WW, CS, LLW or MSVM2). The training functions for a particular model type appear in separate files named `libtrainMSVM_XX.c` and `libevalMSVM_XX.c`, where `XX` stands for the model type.

The other functions included in the library are for internal use and should be called only by the functions described above.

3.8 Matrix format

The matrix format conforms to the one of [9]. For a double-precision real matrix X, we have:

- `X` of type `double**`: pointer to the matrix X
- `X[i]` of type `double*`: pointer to the ith row of the matrix X
- `X[i][j]` of type `double`: element (i, j) of the matrix X

This means that notations like `X[(i-1)*N+j]` cannot be used to access $(X)_{ij}$.
Note: the ranges for the subscripts i and j are $[\![1, M]\!]$ and $[\![1, N]\!]$, respectively, for a matrix of size $M \times N$ (do not use $i = 0$ or $j = 0$).

3.9 File formats

Here is the list of the different file formats described below and the corresponding naming conventions.

- Data file format: `.train` or `.test` (not always mandatory)
- Model file format: `.model`
- Output file format: `.outputs`
- Log file format: `.log`
- Kernel parameter file format: no naming convention, but cannot start with a number
- Matrix file format: used by `.init` and `.alpha` files

Data file format. The library can load data sets from data files in the following format:

```
1200     --> number of data
4        --> dimension of the data
7.400000 2.800000 6.100000 1.900000 3.000000
7.900000 3.800000 6.400000 2.000000 3
...
3.400000 -2.800000 0.5600000 2.200000 2
1.300000 0.800000 5.100000 1.500000 1.000000
|<-----     vector x_i      ----->|  y_i
```

where the labels y_i can be either positive integers (in an integer or floating-point data format) or omitted (for test data). If the labels are included, the number of classes in a data set is automatically set to $\max_i y_i$ when the file is read by `MSVM_make_dataset("data_filename")`.

Model file format (`.model`). The M-SVM models are saved in the following format:

```
1.1                              --> MSVMpack model version
3                                --> type of the M-SVM
3                                --> number of classes Q
2                                --> type of kernel
1 2.500000                       --> kernel parameters (#par par1 par2...)
myTrainingData                   --> training set filename
0.032000                         --> training error rate
120                              --> number of SVs
4                                --> dimension of the SVs (= dimension of the data)
10.000000 10.000000 10.000000 --> values of C_k (soft-margin parameters)
2.0000000000 1.2000000000 2.5000000000 0.4600000000     --> std for normalization
4.0000000000 3.2000000000 2.8000000000 1.0120000000     --> means for normalization
1.1635254839 -3.5845297998 2.4210043159                 --> bias vector b
0.0000000000 6.1803244144 0.0000000000                  --> vector alpha for a SV
5.1000000000 3.5000000000 1.4000000000 0.2000000000 1   --> the corresponding SV
...
13.5696087594 31.9953451299 0.0000000000                --> vector alpha for a SV
6.3000000000 2.8000000000 5.1000000000 1.5000000000 3   --> the corresponding SV
|<-----            vector x_i               ----->| y_i
```

When using `MSVM_save_model()`, the entire training set is saved as SVs in the model. The function `MSVM_save_model_sparse()` should be used to save only the true SVs. Note that the former method allows the model to be retrained (resume training from where it was stopped), whereas the sparse format does not allow this feature.

Output file format (`.outputs`). The files of outputs assume the following format:

```
    1.510324    -1.058388    -0.451936    1
    1.511907    -0.600563    -0.911344    1
   ...
   -0.465904     0.924967    -0.459063    2
   -0.448488     0.856948    -0.408460    2
      |                           |       |
   h_1(x_i)        ...         h_Q(x_i)   predicted label for x_i
```

Log file format (`.log`). The log file format is used to record information during the training process. It assumes the following form:

```
1000 236.011772 234.107674 256.439555 246.631361
2000 240.308282 241.653037 251.815730 244.529922
3000 241.525854 241.474467 247.055776 244.238342
  |      |         |         |          |
#iter   dual       U1        U2         U3
```

where `dual` is the value of the dual objective function at iteration `#iter` and the meaning (and existence) of `U1, U2` and `U3` depends on the model type. For all the M-SVMs except the M-SVM^2, `U1` is the value of the upper bound on the optimum used to check the convergence, whereas `U2` and `U3` do not exist. On the other hand, for an M-SVM^2, `U1` is the objective function value of the *unconstrained* primal problem directly estimated from the current α (not a valid upper bound), `U2` is a cheap upper bound on the optimum obtained by projecting the estimated values of the slack variables onto the constraints and `U3` is the value of the optimized upper bound.

Kernel parameter file format. The parameters of the kernel function can be passed through a file (which is particularly useful for custom kernels using many parameters). The file can take two forms depending on the `trainmsvm` command line. With

```
trainmsvm myData.train -k 5 -P myKernel_parameters
```

the file `myKernel_parameters` must follow the format:

```
6 1.1052 2.00032 3.015 4.0 0.55 0.6
|   |<--- list of parameters --->|
|
number of parameters
```

If the number of parameters is explicitly given on the command line as in

```
trainmsvm myData.train -k 5 -P 6 myKernel_parameters
```

then the file `myKernel_parameters` becomes:

```
1.1052 2.00032 3.015 4.0 0.55 0.6
  |<--- list of parameters --->|
```

4 Optimization features

This section describes the features that allow MSVMpack to make use of the full resources of the computer in terms of CPU and memory. The basic idea is to reduce the computational burden of kernel function evaluations by acting at three different levels:

1. multiple kernel function values are computed in parallel,
2. computed kernel values are cached in memory (as much as possible),
3. kernel functions are optimized (vectorized) for faster evaluations.

Putting these three features together results in a training algorithm which is less limited by the latency of kernel function evaluations. In practice, each training step (except the first one) starts with kernel values that either have been already computed in parallel during the previous step or are stored in cache memory. Thus, the limiting factor becomes the time of the computations required by the optimization step. The most intensive task in this step is the update of the gradient vector, which is also parallelized in MSVMpack.

Note that the behavior described above can only be observed when using a computer with enough memory or CPUs (cores). However, the implemented scheme allows MSVMpack to compensate a lack of memory by using more CPUs or to remain efficient with few CPUs by using extra memory.

4.1 Parallel implementation

MSVMpack offers a parallel implementation of the training algorithms of the four M-SVMs (WW, CS, LLW, M-SVM^2). This parallelization takes place at the level of a single computer to make use of multiple CPUs (or cores) and is not designed for cluster computing across multiple computers.

Settings. The main parameter of this implementation is the number of working threads. For best performance, this number should correspond to the number of available CPUs (or cores). By default, `trainmsvm` and `predmsvm` will automatically detect the number of CPUs at run time, but a particular number `N` can be enforced with the `-t` option, *e.g.*, as in

```
trainmsvm myTrainingData myMSVM.model -t N
```

This can be useful on architectures that include hyperthreading, for which the system typically assumes a number of CPUs that is twice the number of physical CPUs (or cores).

Design. The implementation follows a simple design with a rather coarse granularity, in which kernel computations are parallelized. These computations are the most intensive ones compared to training steps and model updates. In addition, they only depend on the training data, which is constant throughout training. It is thus possible to compute multiple kernel function values at the same time or while a model update is performed. In summary, the program creates a number of working threads which implement the following scheme (in which the steps running in parallel are shown in italic).

1. *Select a random chunk of data.*
2. *Compute the kernel submatrix corresponding to this data chunk.*
3. Wait for the M-SVM model to become available (in case another thread is currently updating the model).
4. Take a training step and update the model.
5. Release the model (to unblock waiting threads).
6. Loop from step 1 until training is stopped.

Note that, for the CS model type, the data chunk selection is not random and actually makes use of the model (see section 4.2). 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.

Assuming a number of threads equal to the number of CPUs, a thread can be blocked waiting in step 5 for at most $(\#\text{CPUs}-1)\times(\text{time required by a training step})$. In most cases, this quantity is rather small compared to the time required to compute the kernel submatrix. This results in a high working/idle time ratio. However, a small additional delay can be observed when `MSVM_eval()` is called (every 1000 iterations) to evaluate the criterion (1). This is particularly true for the M-SVM^2, for which the computation of the upper bound on the optimum is more involved due to the form of the primal objective function.

The next most intensive computation is the update of the gradient vector after each training step. This computation is parallelized by splitting the gradient vector in several parts, each one of which is updated in a separate thread. For best efficiency, MSVMpack maintains the number of working threads equal to the number of CPUs. At the beginning of training, all working threads apply the procedure above to compute kernel function values in parallel and a single thread (the one that is currently in step 4) is used to compute the gradient updates[3]. As training goes, kernel values are cached (see section 4.3 below) and the work load of kernel computations decreases. Former working threads are thus stopped and more threads are used to update the gradient[4]. If the kernel matrix fits in memory, then all CPU resources are eventually assigned to these gradient updates.

The classification function `MSVM_classify_set()` (used by `predmsvm`) is also parallelized to increase the speed of predictions on large test sets. The approach is rather straightforward: the test set is cut into subsets of size `SET_SIZE` (constant defined in `libevalMSVM.h`) and the model output is computed on each subset in a separate thread.

The `K`-fold cross validation procedure (called with `-cv K`) requires training of `K` 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, each one of the `K` models is trained in parallel with #CPUs/`K` working threads applying the scheme above to compute the kernel function values and gradient updates. Otherwise, the first #CPUs models are trained in parallel with a single working thread and the next one starts training as soon as the first one is done, and so on. When all models are trained, the `K` test subsets are classified sequentially, but nonetheless efficiently thanks to the parallelization of the classification function `MSVM_classify_set()` described above.

[3] A different setting is used for the CS model type as the dedicated working set selection method leads to a better use of the kernel cache. Thus, much less kernel evaluations are required and more threads can be used for gradient updates.

[4] See the function `switch_thread()` in `libtrainMSVM.c` for the parameters of these thread switches.

Thread implementation. We used POSIX threads (Pthreads) to implement parallelism on shared memory multiprocessor architectures. POSIX threads constitute the IEEE/The Open Group standard for thread implementation available across many platforms. However, this choice leads to the need for a specific port of the software for Windows platforms. Windows users can read about the Pthreads-win32 project at `http://sourceware.org/pthreads-win32/` for a possible alternative solution (not tested).

4.2 Working set selection

Most implementations of SVM learning algorithms include a dedicated working set selection procedure that determines the subset of 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 (not even for one training step). In such cases, only a small subset of the kernel matrix need to be computed, thus limiting the amount of memory required for kernel cache (see section 4.3).

4.2.1 M-SVM model of Crammer and Singer (CS)

MSVMpack includes the working set selection strategy proposed in [2] in order to choose the chunk of data considered in the next optimization step. This strategy is based on the Karush-Kuhn-Tucker (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[5]

$$\psi_i = \max_{k \in \{j \,:\, \alpha_{ij} > 0\}} G_{ik} - \min_{k \in \{1,\ldots,Q\}} G_{ik}, \tag{2}$$

where G_{ik} is the gradient with respect to α_{ik} of the dual objective function being minimized.

In practice, the conditions $\alpha_{ik} > 0$ involved in the computation of the maximum in ψ_i has to be tested up to a given precision with a threshold. However, note that, contrary to other implementations that typically use $\max_i \psi_i < \epsilon$ as a stopping criterion, MSVMpack stops the algorithm on the basis of the ratio (1) and uses ψ_i only for working set selection. Thus, the value chosen for the threshold does not influence the quality of the solution obtained at the end of training (but can however influence the speed of convergence to that solution).

4.2.2 Other M-SVM models

Unfortunately, there is no simple measure of the violation of the KKT conditions for the other M-SVM models. Thus, for these models, MSVMpack uses a random selection procedure for the next data points to consider in the chunk.

Note that the implementation of Weston and Watkins model as proposed in [6] 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 term $\sum_{k=1}^{Q} b_k^2$ is also minimized. MSVMpack solves the original form of the problem as proposed in [11] and thus cannot use a similar strategy.

4.3 Kernel cache

MSVMpack can use extra memory to cache the kernel matrix (or a subset of it). Note that, in comparison with other SVM implementations, working set selection in MSVMpack is based on a random selection of a data chunk (except for the CS type, see section 4.2) and no shrinking is performed (see for instance [1] for a definition of shrinking and more information on other implementations). As a result, the software may store the entire kernel matrix if it fits in memory, so that large cache memory sizes will help to deal with large data sets (other implementations with proper working set selection and/or shrinking typically require much fewer kernel evaluations and cache memory).

[5] The change in notation compared to [2] is due to the fact that MSVMpack solves the dual problem with respect to the variables α_{ik} instead of $\tau_{ik} = C\delta_{ik} - \alpha_{ik}$.

Settings. By default, training algorithms use the entire physical memory minus 1 GByte for kernel caching, but this can be changed from the command line with the `-x` option. For instance:

```
trainmsvm myDataFile -x 4096
```

will set the maximal amount of kernel cache memory to 4 GBytes.

Design. The kernel cache maintains a list of the kernel matrix rows that are already cached. When a worker thread needs a particular row, the following three cases can occur.

1. *The row is already cached and available:* the worker thread gets a pointer to this row in cache memory.
2. *The row is not available in cache:* the worker thread computes the row in cache memory and then uses a pointer to this row.
3. *The row is not available in cache but another thread is currently computing it:* the worker thread waits for the computation of the row to terminate and then gets a pointer to this row in cache memory.

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 computed and stored.

Cross validation. In the parallelized cross validation procedure described in Sect. 4.1, many kernel computations can be saved as the `K` models are trained in parallel from overlapping subsets of the data set. 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 `K` models, a *local* kernel cache stores rows of a kernel submatrix computed with respect the corresponding training subset. When a working thread 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 model and takes care of the data permutation used to generate the training subsets at the beginning of the cross validation. In this scheme, every kernel computation performed by a working thread 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 `K` models.

4.4 Data format-specific and 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 different kernel function for each format. This may be used for instance to increase the speed of kernel computations on single precision floats or integers. Note that, in a kernel computation $v = k(x, z)$, 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.

Settings. The data format can be chosen from the command line by using the appropriate option according to Table 3. E.g., for single precision floating-point data format:

```
trainmsvm mySinglePrecisionData -m WW -k 2 -f
```

In this example, the data are loaded in memory as single-precision floats and the kernel function (the Gaussian RBF in this example) may be computed faster (see below).

Table 3: Data format options for `trainmsvm`.

Option flag	Data format
none (default)	double precision floating-point (64-bit)
`-f`	single precision floating-point (32-bit)
`-I`	32-bit integer
`-i`	16-bit short integer
`-B`	byte (8-bit unsigned integer)
`-b`	bit (all non-zero values are considered as 1)

Vectorization. Proper data alignment in memory allows MSVMpack to use vectorized implementations of kernel functions when compiled on SSE4.2, SSE4.1, SSE3 or SSE2 capable processors. These implementations make use of the SIMD (Single Instruction Multiple Data) instruction sets available on such processors to process multiple components of large data vectors simultaneously. Note that this is mostly effective for small data formats such as single-precision floats or short integers.

5 Some 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:

- J. Weston's own implementation of his M-SVM model (WW) in Matlab included in the Spider[6],
- the BSVM[7] 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[8],
- and Lee's implementation in R of hers (LLW) named SMSVM[9].

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 [8].

The characteristics of the data sets used in the experiments are given in Table 4. The ImageSeg data set is taken from the UCI repository[10]. 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 [3]. MSVMpack uses the original MNIST data[11] with integer values in the range $[0, 255]$, while other implementations use scaled data downloaded from the LIBSVM homepage[12]. BSVM scaling tool is applied to the Letter data set downloaded from the UCI repository[13] to obtain a normalized data set in floating-point data format.

[6] `http://people.kyb.tuebingen.mpg.de/spider/`
[7] `http://www.csie.ntu.edu.tw/~cjlin/bsvm/`
[8] `http://www.cis.upenn.edu/~crammer/code-index.html`
[9] `http://www.stat.osu.edu/~yklee/software.html`
[10] `http://archive.ics.uci.edu/ml/datasets/Image+Segmentation`
[11] 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.
[12] `http://www.csie.ntu.edu.tw/~cjlin/libsvmtools/datasets/`
[13] `http://archive.ics.uci.edu/ml/datasets/Letter+Recognition`

All methods use the same kernel functions and hyperparameters as summarized in Table 4. 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 5–6. Though no definitive conclusion can be drawn from this small set of experiments, the following is often observed in practice:

- *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 as included since* MSVMpack 1.4 *is fast.* Table 6 shows that a 5-fold cross validation can be very fast 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* $Q = 26$*).* 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 classification function `MSVM_classify_set()`, which allows multiple predictions to be computed simultaneously. For particular data sets, data format-specific and vectorized kernel functions also speed up the testing phase.
- MSVMpack *leads to models with more SVs 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 7 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 (see the `dot_bit()` function in `kernel.c`). 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.

6 License and how to cite

MSVMpack is released under the GPL, of which a copy can be found in `MSVMpack/gpl.txt`. This package includes `lp_solve`[14], which is used to solve the linear programming problems, and

[14] `lp_solve` is freely available under the terms of the GNU lesser general public license at `http://lpsolve.sourceforge.net/`.

Table 4: Characteristics of the data sets.

Data set	#classes	#training examples	#test examples	data		Training parameters
				dim.	format	
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

`mongoose`[15] for the web server.

When referring to the software, please include the following citation:

```
@article{MSVMpack,
  author = {F. Lauer and Y. Guermeur},
  title = {{MSVMpack}: a Multi-Class Support Vector Machine Package},
  journal = {Journal of Machine Learning Research},
  volume = {12},
  pages = {2269-2272},
  year = {2011},
  note = {\url{http://www.loria.fr/~lauer/MSVMpack}}
}
```

which should look like [7].

References

[1] L. Bottou, O. Chapelle, D. DeCoste, and J. Weston, editors. *Large-Scale Kernel Machines*. The MIT Press, Cambridge, MA, 2007.

[2] K. Crammer and Y. Singer. On the algorithmic implementation of multiclass kernel-based vector machines. *Journal of Machine Learning Research*, 2:265–292, 2001.

[3] J. A. Cuff and Barton G. J. Evaluation and improvement of multiple sequence methods for protein secondary structure prediction. *Proteins*, 34(4):508–519, 1999.

[4] M. Frank and P. Wolfe. An algorithm for quadratic programming. *Naval Research Logistics Quarterly*, 3(1–2):95–110, 1956.

[5] Y. Guermeur and E. Monfrini. A quadratic loss multi-class SVM for which a radius-margin bound applies. *Informatica*, 22(1):73–96, 2011.

[6] C.-W. Hsu and C.-J. Lin. A comparison of methods for multi-class support vector machines. *IEEE Transactions on Neural Networks*, 13(2):415–425, 2002.

[7] F. Lauer and Y. Guermeur. MSVMpack: a multi-class support vector machine package. *Journal of Machine Learning Research*, 12:2269–2272, 2011. `http://www.loria.fr/~lauer/MSVMpack`.

[15] `mongoose` is freely available under the terms of the GPL at `http://code.google.com/p/mongoose/`.

Table 5: 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	(S) 48h 00m 00s	3s

(S): manually stopped before reaching the default optimization accuracy.

Table 6: Results of a 5-fold cross validation on the **CB513 data set**. The given times are sums over the 5 training and testing steps of the cross validation. MSVMpack 1.4 includes a parallelized cross validation procedure. The star superscript indicates that the software is run on a computer with 12 CPUs and 64 GB of memory.

M-SVM model	**Software**	**Cross validation error**	**Training time**	**Testing time**	**Total time**
WW	BSVM	23.96 %	9h 48m 40s	46m 29s	10h 34m 50s
	MSVMpack 1.0	23.72 %	1h 05m 11s	1m 51s	1h 07m 02s
	MSVMpack 1.4*	23.63 %	N/A	N/A	21m 07s
CS	MCSVM	23.55 %	22h 52m 10s	2h 08m 33s	25h 00m 43s
	MSVMpack 1.0	23.63 %	1h 00m 36s	2m 06s	1h 02m 42s
	MSVMpack 1.4*	23.55 %	N/A	N/A	12m 05s
LLW	MSVMpack 1.0	25.65 %	1h 14m 21s	2m 33s	1h 16m 54s
	MSVMpack 1.4*	25.52 %	N/A	N/A	30m 29s
$MSVM^2$	MSVMpack 1.0	23.47 %	6h 44m 50s	2m 49s	6h 47m 39s
	MSVMpack 1.4*	23.36 %	N/A	N/A	2h 55m 05s

Table 7: Effect of the cache memory size on the training time of MSVMpack for the WW model type.

Bio

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

CB513_01

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

[8] Y. Lee, Y. Lin, and G. Wahba. 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.

[9] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery. *Numerical recipes in C: the art of scientific computing*. Cambridge University Press, Cambridge, MA, 2nd edition, 1992.

[10] J. B. Rosen. The gradient projection method for nonlinear programming. Part I. Linear constraints. *Journal of the Society for Industrial and Applied Mathematics*, 8(1):181–217, 1960.

[11] J. Weston and C. Watkins. Multi-class support vector machines. Technical Report CSD-TR-98-04, Royal Holloway, University of London, 1998.