mySVM - Manual

Stefan Rüping
Computer Science Department, AI Unit
University of Dortmund
`rueping@ls8.cs.uni-dortmund.de`

October 30, 2000

Contents

1 Introduction

mySVM is an implementation of the support vector machine (see [Vapnik, 1998] for details) based on the algorithm presented in [Joachims, 1999]. It supports pattern recognition and regression estimation SVMs, ν-SVMs ([Schölkopf et al., 2000a]) and SV distribution support estimation ([Schölkopf et al., 2000b]). mySVM works with linear or quadratic and even asymmetric loss functions. An advantage of mySVM are the multiple input formats for the examples files, which in many cases reduce the need for converting your data into the SVM format.

NOTE: mySVM is still under development, so some things might not work as expected (and this manual certainly is one of those things). Please send me an email if you find any bugs.

2 The Optimisation Algorithm

2.1 SVMs

To calculate the SVM for a given set of examples (x_i, y_i) and a capacity constant C one has to solve the following optimisation problem:

Minimize

$$\Phi(w,\xi,\xi^*) = \frac{1}{2}(w^T w) + C\left(\sum_{i=1}^{l}\xi^* + \sum_{i=1}^{l}\xi\right)$$

with respect to

$$\begin{aligned}
(w^T x_i) + b &\geq y_i - \varepsilon - \xi_i^* \quad , i = 1,\ldots,n && (1)\\
(w^T x_i) + b &\leq y_i + \varepsilon + \xi_i \quad , i = 1,\ldots,n && (2)\\
\xi_i^* &\geq 0 \quad , i = 1,\ldots,n && (3)\\
\xi_i &\geq 0 \quad , i = 1,\ldots,n && (4)
\end{aligned}$$

Equivalently one can solve the dual formulation of the optimization problem: Maximize

$$\begin{aligned}
W(\alpha,\alpha^*) &= -\frac{1}{2}\sum_{i,j=1}^{l}(\alpha_i^* - \alpha_i)(\alpha_j^* - \alpha_j)(x_i \cdot x_j)\\
&\quad + \sum_{i=1}^{l} y_i(\alpha_i^* - \alpha_i) - \sum_{i=1}^{l}\varepsilon(\alpha_i^* + \alpha_i) && (5)
\end{aligned}$$

with respect to

$$\begin{aligned}
0 \leq \alpha_i, \alpha_i^* \leq C \qquad &, i = 1,\ldots,n && (6)\\
\sum_{i=1}^{l}\alpha_i^* = \sum_{i=1}^{l}\alpha_i & && (7)
\end{aligned}$$

2.2 Decomposition algorithms

Instead of solving the dual optimization problem directly, it proves to be better to iteratively decompose the problem into a small working set S and minimize the target function on the working set only, keeping the other variables fixed (see [Osuna et al., 1997]).

The idea of [Joachims, 1999], which is implemented in mySVM, is to compute the working set on the basis of feasible directions, i.e. a direction with only q non-zero elements that will minimize the target function the most. The working set then consists of the q non-zero elements

2.3 A Decomposition algorithm for regression

The drawback of the regression formulation of the SVM compared to the pattern recognition formulation is that the regression formulation contains two variables α and α^* for each example while the pattern recognition formulation contains only one variable α for each example. The reason for this is that in the pattern recognition case one can replace the constraints

$$\begin{aligned}
(w^T x_i) + b &\geq 1 - \xi_i^* \mathrm{iff} y_i = 1 && (8)\\
(w^T x_i) + b &\leq -1 + \xi_i \mathrm{iff} y_i = -1 && (9)\\
\xi_i^* &\geq 0 \quad , i = 1, \ldots, n && (10)\\
\xi_i &\geq 0 \quad , i = 1, \ldots, n && (11)
\end{aligned}$$

in the primal formulation by

$$\begin{aligned}
y_i((w^T x_i) + b) - 1 + \xi_i &\geq 0 && (12)\\
\xi_i &\geq 0 \quad , i = 1, \ldots, n && (13)
\end{aligned}$$

However, in the regression case it is clear that only on of the slack variables ξ_i and ξ_i^* can be positive and therefore in the dual formulation only one of the vartiables α_i and α_i^* is non-zero. So it suffices to solve the SVM optimization problem for n variables instead of all 2*n variables.

The problem with this approach is that one does not know if α_i or α_i^* will be zero in the optimal solution of the SVM problem. The solution is to solve this problem iteratively during the optimization:

1. Compute the q feasible directions D.
2. For each feasible direction i: If one of α_i, α_i^* is non-zero, put this variable into the working set. Else put one the variables into the working set, whose KKT condition is not fulfilled.
3. Replace the constraints $\alpha_i^{(*)} \geq 0$ on the subproblem by $\alpha_i^{(*)} \geq -\varepsilon$ for a sufficiently small $\varepsilon > 0$.
4. Solve the new subproblem on the working set.

5. For each new α_i from the working set: If $\alpha_i < 0$ set $\alpha_i^* = -\alpha_i$ and $\alpha_i = 0$. Replace each α_i^* from the working set correspondingly.

After the optimization converged, all $\alpha_i^(*)$ with $|\alpha_i^(*)| < \varepsilon$ are set to zero.

It is easy to see that the KKT conditions for the q-dimensional subproblem are the same as the corresponding equations from the whole optimization problem, so if the α_i from the optimal solution is non-zero, but α_i^* is non-zero during the optimization process, then at some point α_i^* will be pushed against its lower bound and thus α_i will become positive in the next iteration. By this the optimization will finally converge to the optimal solution.

3 Installation and Usage

The installation of mySVM is very simple: Put the mySVM files into a directory of your choice (let's assume `mySVM` for this tutorial) and type `make` to compile mySVM. This will compile the programs `mysvm` and `predict` and put them into a subdirectory of `mySVM/bin/` that corresponds to your machine and OS type. `mysvm` is used to train a SVM and to test the result. `predict` is used to predict more examples with an already trained SVM.

3.1 Using `mysvm`

As already said, `mysvm` is used to train a SVM on a given example set and to test the resulting SVM on some other example sets. You need to supply the following information:

1. The parameters of the SVM and the training process
2. A kernel definition
3. A training set
4. example sets to predict or test the SVM on (optional)

You can pass this input to `mysvm` in one or more files. If no input files are given, the input is read from STDIN. The input formats are explained in section 4.

4 Input Formats

4.1 How input is read by mySVM

There are three types of input: SVM parameters, kernel parameter and example sets. Each type of input has to be given in an own section of the input files, beginning with the keywords `@parameters`, `@kernel` or `@examples`, respectively. There can be multiple input section per file, but one input section cannot be split-ted into multiple files. In all input section lines beginning with # are being ignored.

You need to supply the SVM parameters and at least one example set as the training set. If no kernel definition is given, a dot kernel is assumed. If you enter more than one example set, the first set is taken as the training set and the following example sets are either used as test sets (if y-values are given) or the classification of the examples in these sets are predicted (if no y-values are given).

If no input files are given or the filename "-" is given, the data is read from STDIN.

4.2 mySVM parameters

The SVM parameters can be divided into three types: Parameters, that define the SVM itself, parameters, that influence the optimization process of the training and parameters, that define different ways of training.

4.2.1 SVM definition

The definition of the SVM itself:

`C` <*float*>: The capacity parameter of the SVM (must be positive). NOTE: The given C is divided by the number of examples for training!

`pattern`: Use pattern recognition SVM.

`regression`: Use regression recognition SVM (default).

`nu` <*float*>: Use the nu-SVM algorithm instead of the normal SVM algorithm with the given value for nu (see [Schölkopf et al., 2000a] for details on nu-SVMs).

`distribution`: Do SV estimation of the support of the examples (see [Schölkopf et al., 2000b]). Nu must be set.

The definition of the loss function of the SVM:

`L+` <*float*>: factor that positive deviation (prediction too high) is penalized by (must be non-negative).

`L-` <*float*>: factor that negative deviation (prediction too small) is penalized by (must be non-negative).

`epsilon+` <*float*>: positive insensitivity. Constant that the prediction can be higher than the functional value without being penalized (must be non-negative).

`epsilon-` <*float*>: negative insensitivity. Constant that the prediction can be smaller than the functional value without being penalized (must be non-negative).

`epsilon` *<float>*: insensitivity. Constant that the prediction can deviate from the functional value without being penalized. Sets both `epsilon+` and `epsilon-` (must be non-negative).

`quadraticLoss+`: Use quadratic loss for positive deviation.

`quadraticLoss-`: Use quadratic loss for negative deviation.

`quadraticLoss`: Use quadratic loss for both positive and negative deviation.

4.2.2 Optimizer parameters

`working_set_size` *<integer>*: Number of examples in working set (must be greater than 1).

`max_iterations` *<integer>*: Stop after this many iterations (default 1000000)..

`shrink_const` *<integer>*: Shrink a variable if it is optimal this many iterations (default 100).

`is_zero` *<float>*: Numeric precision (default: 1e-10).

`descend` *<float>*: The minimum descend the target function has to make in each iteration step (default: 1e-10).

`convergence_epsilon` *<float>*: Precision that the KKT constraints must be fulfilled with (default 1e-3).

`kernel_cache` *<integer>*: Size of the kernel cache im MB. Set as large as will fit into the main memory (default 40).

4.2.3 Global parameters

`verbosity` *[1..5]*: Amount of messages the SVM gives. Ranges from 1 (almost none messages) over 3 (default) to 5 (flood, for debugging purposes only).

`format` and `delimiter`: You can set a default for the examples file format. See the description in section 4.4.

4.2.4 Training algorithms

`cross_validation` *<integer>*: Do cross validation on the training examples. The examples are divided into the given number of chunks and each chunk is predicted by the classificator learned on the remaining chunks.

`cv_inorder`: Do cross validation in the order that the examples are given. Otherwise the examples will be randomly permuted before cross validation.

`cv_window` *<integer>*: Do cross validation by moving a window of the given number of chunks through the examples and predicting the following chunk of examples. Useful for time series prediction.

`search_C` *[amg]*: Find the optimal C in the range of `cmin` to `cmax`. If the parameter `a` is given, `cdelta` is added to C in each iteration. With `m`, C is multiplied with `cdelta` in each iteration. The parameter `g` searches C with using the method of the golden ration up to a relative error of `cdelta` (Warning: it is assumed that the function $C \mapsto loss$ is convex, which generally is correct only up to a certain degree. Use it if you have no idea where the optimal C will lie).

`cmin` *<float>*: Parameter used with `search_C`.

`cmax` *<float>*: Parameter used with `search_C`.

`cdelta` *<float>*: Parameter used with `search_C`.

4.3 Kernel definition

The kernel definition has to begin with the line `type` *<kernel-type>*, where *<kernel-type>* is one of `dot, polynomial, radial, neural, anova, user, user2, prod_aggregation, sum_aggregation`.

4.3.1 dot kernel

The dot kernel is defined by

$$k(x, y) = x * y$$

i.e. it is the inner product of x and y.

4.3.2 polynomial kernel

The dot kernel is defined by

$$k(x, y) = (x * y + 1)^d$$

with the parameter `degree` *<integer>*.

4.3.3 radial kernel

The radial kernel is defined by

$$k(x, y) = \exp(-\gamma ||x - y||^2)$$

with the parameter `gamma` *<float>*.

4.3.4 neural kernel

The neural kernel is defined by

$$k(x, y) = \tanh(ax * y + b)$$

with the parameters `a` *<float>* and `b` *<float>*. Note that not all choices of a and b lead to a valid kernel function.

4.3.5 anova kernel

The anova kernel is defined by

$$k(x,y) = (\sum_i \exp(-\gamma(x_i - y_i)))^d$$

with the parameters `gamma` <*float*> and `degree` <*integer*>.

4.3.6 user kernel

You can enter your own kernel definition in the file `kernel.cpp` in the method `kernel_user_c::calculate_K`. The following parameters are provided:

`param_i_1` **...** `param_i_5` <*integer*>

`param_f_1` **...** `param_f_5` <*float*>

4.3.7 user kernel 2

You can enter your own kernel definition in the file `kernel.cpp` in the method `kernel_user2_c::calculate_K`. As parameters two arrays of size `number_param` (defined in `kernel_user2_c::kernel_user2_c()`) are provided: `param_i` for integer values and `param_f` for float values.

4.3.8 aggregation kernels

It is known that if $k(x,y)$ is a kernel, then also

$$k_{\text{agg}}((x_1,x_2,x_3),(y_1,y_2,y_3)) = k(x_2,y_2)$$

is a kernel. Also the sum and the product of two kernels is a kernel. Therefore the following functions are kernels too:

$$k((x_1,\ldots,x_n),(y_1,\ldots,y_n)) = \sum_{i=1}^{p} k_i(\pi_i(x),\pi_i(y))$$

and

$$k((x_1,\ldots,x_n),(y_1,\ldots,y_n)) = \prod_{i=1}^{p} k_i(\pi_i(x),\pi_i(y))$$

where the π_i are projections. Aggregation kernels are defined as follows: First the number of kernels to be aggregated has to be given: `number_parts` <*integer*>.

Then for every kernel part the range of the attributes to take hast to be given: `range` <*integer*> <*integer*>.

Following that all part kernels are defined separately as usual, each beginning with the line `@kernel`.

4.4 Example definition

Examples can be given in two different formats: sparse and dense. In the sparse format, only non-zero attributes are given whereas in the dense format all attributes have to be given.

The following parameters exist in both cases:

`dimension` *<integer>*: Number of attributes of the examples. If not given, then in the sparse case the the dimension is set to the maximal index occuring. In the dense case the number of attributes in the first line containing an example is used.

`number` *<integer>*: Total number of examples. This parameter is optional, but reading the examples is faster if it is given. If a wrong number of examples is read, a warning is issued.

`b` *<float>*: The additional constant of the hyperplane.

`format` *<string>*: Format of the examples. *<string>* can be `sparse` for the sparse format or one of `x, xy, yx, xya, xay, yxa, yax, axy, xyx` for the dense format. When using the dense format, the string gives the order of the x-, y- and alpha-values of the examples. You can set a default format in the parameters definition, otherwise the default format is `yx`.

`delimiter` *char*: Character by which the attributes of an example are separated (default: space). You can also set a default delimiter in the parameters definition.

In both cases there must be one example per line. In the dense case the line consists of all x-attributes, y-value and alpha-value (if given) in the defined order, separated by white-space. In the sparse case, the lines consists of entries of the format *<integer>*:*<float>*, where the integer gives the attribute number (starting at 1). The alpha-value is given by `a`:*<float>* and the y-value is given by `y`:*<float>* or simply *<float>*.

example: The following three example files are equivalent:

```
@examples
format xy
dim 2
1 1 2
-1 1 4
2 0 0
```

```
@examples
format sparse
1:1 2:1 y:2
2:1 y:4 1:-1
0 1:2
```

```
2 1 1
4 -1 1
0 2 0
```

4.5 Output of mySVM

Here is a sample output of mySVM. Let's walk through it step by step.

```
> mysvm TRAIN.dat TEST.dat PARAM.dat
Reading TRAIN.dat
   read 53 examples, dimension = 24.
Reading TEST.dat
   read 5 examples, dimension = 24.
Reading PARAM.dat
```

The SVM has successfully read in a training set of 53 examples, a test set of 5 examples and the parameters.

```
Training started with C = (1,1).
.........
*** Convergence
Done training: 8 iterations.
```

The training has converged, now some statistics are given.

```
Target function: -4.92116964
----------------------------------------
Average loss  : 7.77034764 (loo-estim: 495.949566)
Max. loss pos : 33.5115981
Max. loss neg : 29.7526961
Support Vectors : 27
Bounded SVs     : 21
min SV: -0.0188679245
max SV: 0.0188679245
|w| = 0.648979262
max |x| = 158.12337
VCdim <= 10530.6156
```

The values given are: The value of the function to minimize, the average loss and a predictor for the leave-one-out loss, the maximal and minimal loss of a single example, the number of support vectors and the number of support vectors at the upper bound, the minimal and maximal value of the alphas, the 2-norm of the hyperplane vector, the maximum size of the examples and an estimator of the VCdim computed from the last two values.

What you cannot see is that this is a regression SVM. Had it been a pattern recognition SVM, also the training results and leave-one-out estimators for recall and precision would have been printed.

If there are less than 30 attributes, the full hyperplane vector is printed:

```
w[0] = 0.174969116

⋮

w[23] = 0.00362512157
b = 90.6365681

Saving trained SVM to TRAIN.dat.svm
```

The optimization result is saved in the file <*filename*>`.svm`. It is in the example file format with the alpha-values given, so you can use the trained SVM again by taking this file and the original parameters.

```
Starting tests
Testing examples from file TEST.dat
```

The examples from the other example files are used for testing the SVM (the result is given in the same format as above) or, if no y-values are given, are predicted and saved in the file <*filename*>`.svm`.

Note: Special SVM algorithms such as ν-SVM and SV distribution support estimation may give additional information!

References

[Joachims, 1999] Joachims, T. (1999). Making large-scale SVM learning practical. In Schölkopf, B., Burges, C., and Smola, A., editors, *Advances in Kernel Methods - Support Vector Learning*, chapter 11. MIT Press, Cambridge, MA.

[Osuna et al., 1997] Osuna, E., Freund, R., and Girosi, F. (1997). An improved training algorithm for support vector machines. In Principe, J., Giles, L., Morgan, N., and Wilson, E., editors, *Neural Networks for Signal Processing VII — Proceedings of the 1997 IEEE Workshop*, pages 276 – 285, New York. IEEE.

[Schölkopf et al., 2000a] Schölkopf, B., Smola, A. J., Williamson, R. C., and Bartlett, P. L. (2000a). New support vector algorithms. *Neural Computation*, 12:1207–1245.

[Schölkopf et al., 2000b] Schölkopf, B., Williamson, R. C., Smola, A. J., and Shawe-Taylor, J. (2000b). Sv estimation of a distribution's support. In Solla, S., Leen, T., and Müller, K.-R., editors, *Neural Information Processing Systems 12*. MIT Press. forthcoming.

[Vapnik, 1998] Vapnik, V. (1998). *Statistical Learning Theory*. Wiley, Chichester, GB.