Toolbox SimpleSVM Documentation

Gaëlle Loosli

Abstract

Here is a quick guide for SimpleSVM toolbox for Matlab. It contains a quick start tour as well as some details about special features. We also give an overview of the algorithm.

1 How to use it

1.1 Installation and test

After downloading the archive, you just need to unzip it where it is the most convenient for you. Then you add your repertory in your matlab path (`'addpath ('/home/.../simpleSVM/')'`). You're done!
To check how well it works, you can launch `'graphicalInterface'` if you are running Matlab 7, or `'graphicalInterface_v6'` if you are running a previous version (I didn't check for Matlab 5 but it should work). You can also launch `'demo_online'`. This little program will execute a serie of examples that uses the different features of the toolbox.

1.2 Quick start tour

Here we give a code sample that uses the toolbox the simplest way. It computes the binary SVM solution for a checkers problem.

```
global svModel
[x,y,xt,yt]=dataset('Checkers',200,50,0.5); % generates data
donnees = data(x,y,xt,yt);                  % stores data
noyau = kernel('rbf',.9);                   % stores kernel
parametres = param(500,50,'binary','chol'); % stores parameters
trainSVM(donnees, noyau, parametres);       % train the SVM
prediction = testSVM;                       % gives the results
                                            % on the test set
```

1.3 Description of structures and main functions

This toolbox uses five structures to store the data, parameters and results. The main one is a global variable called `svModel` that should be declared in any program using the toolbox. It was made as a global variable for memory purpose. Matlab makes a copy of every input parameter called for a function which is a problem for large datasets. The global variable prevents it, but you need to be careful about it.
This structure contains four structures:

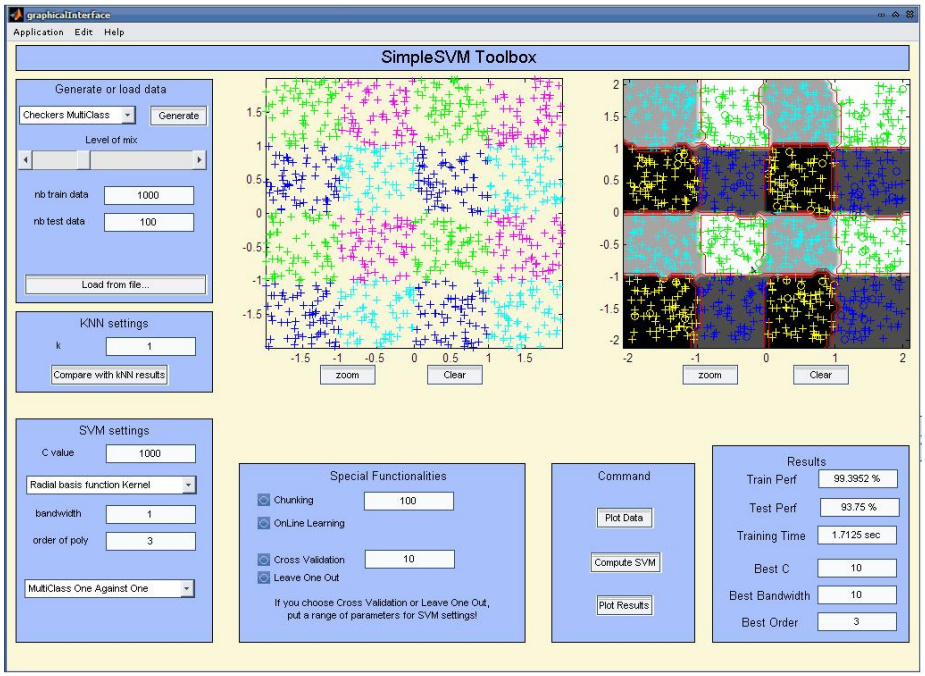

Figure 1: Graphical Interface

- the dataset,
- the kernel,
- the parameters,
- the (output) model.

Data. It contains the train set, test set and their labels. This structure can be created thanks to the function `data` as follows:
`mydata = data(trainMatrix, trainLabels, testMatrix, testLabels);`
`mydata` is now as shown:

```
mydata =

      trainvec: [2x96 double]
      trainlab: [96x1 double]
       testvec: [2x48 double]
       testlab: [48x1 double]
    trainvec_t: []
    trainlab_t: []
     testvec_t: []
     testlab_t: []
```

The four empty fields are used to store original data when they are manipulated (in multiclass algorithm for instance). Note that the orientation of the matrix and vectors is important (each example is a column and the labels are a column vector).

Kernel. It contains the type of kernel you want to use and its parameters. This structure is created via the function `kernel`:
`mykernel = kernel('rbf', 0.9);`
`mykernel` is now as shown:

```
mykernel =

     name: 'rbf'
    sigma: 0.9000
```

The other type is 'poly' and take as a parameters the order and the coefficient:
`mykernel = kernel('poly', [5,0.5]);`
`mykernel` is now as shown:

```
mykernel =

      name: 'poly'
    degree: 5
     scale: 0.5000
```

Parameters. It contains all the parameters that are needed to run the SVM algorithm. Some are optional or have default values that are almost always good.
`parametres = param(C,step,multiclass,method,gap,type,transfo,fid,span,verbose);`
Here is an example of parameters you can set.

```
myparam =

                C: 500
             step: 50
        multiclass: ''
            method: 'chol'
               gap: 1.0000e-005
              type: 'none'
   transformations: []
               fid: 1
              span: []
           verbose: 1
```

- The `C` value is the smoothing value. It is the only one that it is essential to set. Its default value is 100 if you create your structure with no argument,
- the `step` is the size of the subsets the algorithm consider when checking the accuracy of the current solution. Values between 10 and 200 are good enough (see the algorithm for more details),

- the `multiclass` is a string that can contains '1vs1' or '1vsall' or '1class' if tour data have more or less than 2 classes,
- the `method` refers to the method the linear system is solved ('chol' for cholesky decomposition or 'qr' for QR update decomposition),
- `gap` is the KKT gap and gives the accuracy required for the solution (remember that Matlab accuracy is 10^{-16} during computation. Do not give 0 for the gap, the algorithm could not converge because of the precision of the machine!),
- the `type` is a string that use used for specific variation of the algorithm such as 'invariant',
- the `transformations` is used for the invariances,
- `fid` is 1 by default and sets the default print to the screen. You can set it the the fid of a text file,
- `span` is not used in this version yet,
- if `verbose` is set to 0 nothing will be printed out.

Model. This structure is created by the algorithm and contains the results as well as enough information to start another run from the last solution (really useful for cross validation for instance, if you slightly change the parameters, the solution will be close to the previous one. Here again more details are given with the algorithm).

```
svModel.model =

     iclass: 1
      alpha: {[117x1 double]}
    indices: {[117x1 double]}
         mu: {[-1.0831]}
         G1: {[500x500 double]}
         G2: {[500x500 double]}
        nbC: 53
```

- `iclass` basically gives the number of classifiers that are stored in the model. For instance it's one for binary classification or 1class SVM and the number of classes for the '1 against all' multiclass,
- `alpha` contains the lagrangian coefficient of each support vector.
- `indices` gives the indices of the support vectors,
- `mu` is the bias of the solution (often design as b in SVM formulations),
- `G1, G2` contains the cached kernel matrix decomposition,
- `nbC` is the number of bounded vectors among the support vectors.

dataset This function generates data for specific problems.

1.4 Run it!

Let's now give the main steps to use this toolbox.

1. Declare your global variable (`global svModel;`),
2. load or generate your data - make sure they have the good format,
3. create your data structure (at least 2 arguments) `mydata`,
4. create your param structure (at least 1 argument, be serious!) `myparam`,
5. create your kernel structure (2 arguments) `mykernel`,
6. train it! (`trainSVM(mydata, mykernel, myparam)`),
7. test it! (`prediction <- testSVM`).

You will find many examples of use in the example directory.

1.5 Special features

Multiclass 1vs1 Use it when you have more than 2 classes by setting `param.multiclass = '1vs1'`. It will train a SVM for every couple of classes and decide the output by a vote of these binary classifiers,

Multiclass 1vsall use it when you have more than 2 classes by setting `param.multiclass = '1vsall'`. It will train a SVM per class, considering all the other classes as one. The decision is given by the maximum of the classifiers,

One-Class use it to do some default detection or novelty detection, all the point you are training on are considered as the same class. Set `param.multiclass = '1class'`,

Large scale dataset treatment (Chunking) try it if you think the solver is really too slow for your huge dataset. You will use `chunkingSVM(mydata, mykernel, myparam, sizeChunk)` instead of `trainSVM`,

Cross Validation use it to define the best parameters among a list of parameters (for the kernel!). `[bestP, bestC, best0, maxPerf, performance] = crossvalidation(nbfolders, mydata, myparam, mykernel, bandwidths, slacks, orders)` where the last three arguments are vectors containing the different parameters to test,

Leave One Out use it to define the best parameters among a list of parameters (for the kernel!) when you have few training data. `[bestP, bestC, best0, maxPerf, performance] = leaveOneOut(mydata, myparam, mykernel, bandwidths, slacks, orders)` where the last three arguments are vectors containing the different parameters to test,

Graphical interface try it!

Invariances Work in Progress... documentation will be available soon

DC SVM with non positive kernels, Work in Progress... documentation will be available soon

2 Details about the SimpleSVM algorithms

The binary discrimination SVM problem with the sample $(\mathbf{x}_i, y_i), i = 1, m$ and labels $y_i \in \{-1, 1\}$ is the solution of the following optimization problem under constraints:

$$\left\{\begin{array}{lll} \min\limits_{f,b,\boldsymbol{\xi}} & \frac{1}{2}\|f\|^2_{\mathcal{H}} + C\sum\limits_{i=1}^{m}\xi_i & \\ \text{avec} & y_i(f(\mathbf{x}_i)+b) > 1-\xi_i & i=1,m \\ \text{et} & \xi_i \geq 0, & i=1,m \end{array}\right. \tag{1}$$

where C is a scalar that adjusts the smoothness of the decision function, b is a scalar called bias and ξ_i are slack variables.

The solution of this problem is also the saddle point of the lagrangian :

$$\mathcal{L}(f,b,\boldsymbol{\xi},\boldsymbol{\alpha},\boldsymbol{\beta}) = \frac{1}{2}\|f\|^2_{\mathcal{H}} + C\sum_{i=1}^{m}\xi_i - \sum_{i=1}^{m}\alpha_i\big(y_i(f(\mathbf{x}_i+b))-1+\xi_i\big) - \sum_{i=1}^{m}\beta_i\xi_i \tag{2}$$

from which we retrieve a part of the Kuhn-Tucker conditions :

$$\left\{\begin{array}{rcl} \nabla_f\mathcal{L}(f,b,\boldsymbol{\xi},\boldsymbol{\alpha},\boldsymbol{\beta}) &=& 0 \\ \dfrac{\partial\mathcal{L}(f,b,\boldsymbol{\xi},\boldsymbol{\alpha},\boldsymbol{\beta})}{\partial b} &=& 0 \\ \dfrac{\partial\mathcal{L}(f,b,\boldsymbol{\xi},\boldsymbol{\alpha},\boldsymbol{\beta})}{\partial\boldsymbol{\xi}} &=& 0 \end{array}\right. \Leftrightarrow \left\{\begin{array}{rcll} f(\mathbf{x}) - \sum\limits_{i=1}^{m}\alpha_i y_i k(\mathbf{x},\mathbf{x}_i) &=& 0 & \\ \sum\limits_{i=1}^{m}\alpha_i y_i &=& 0 & \\ C-\alpha_i-\beta_i &=& 0 & i=1,m \end{array}\right.$$

It follows that $f(\mathbf{x}) - \sum_{i=1}^{m}\alpha_i y_i k(\mathbf{x},\mathbf{x}_i)$. Thanks to this relation we can eliminate f in the lagrangian to end up with the following dual formulation :

$$\left\{\begin{array}{lll} \max\limits_{\boldsymbol{\alpha}\in\mathbf{R}^m} & -\frac{1}{2}\boldsymbol{\alpha}^\top G\boldsymbol{\alpha} + \mathbf{e}^\top\boldsymbol{\alpha} & \\ \text{with} & \boldsymbol{\alpha}^\top\mathbf{y} = 0 & \\ \text{and} & 0 \leq \alpha_i \leq C & i=1,m \end{array}\right. \tag{3}$$

where G is the influence matrix of general term $G_{ij} = y_i y_j k(\mathbf{x_i},\mathbf{x_j})$ and $\mathbf{e} = [1,\ldots,1]^\top$. The SVM solution is then given by solving a quadratic optimization problem of dimension m under box constraints.

2.1 Relations between primal and dual variables

In the preceding problem each unknown α_i can be interpreted as the influence of the example $(\mathbf{x}_i, y_i)$ in the solution. Taking into account that only the points lying on the frontiers are important for the discrimination task and that there is *a priori* a few number of them, a large number of α coefficients will be equal to 0. Than it is pertinent to separate the m unknown into three groups of points : $[1,m] = I_s \cup I_0 \cup I_c$, defined according to the associated Lagrange multipliers values α :

[I_s] the group of **supports** points is the one of the candidates support vectors. They are the ones for which $0 < \alpha_i < C$. Those points are lying inside the box constraints. This group is also called working set since it contains the vectors we have to work out to know their associated α values,

[I_c] the group of **bounded** points is the one for which $\alpha_i = C$. These points are lying on the box boundaries. If these points are to close or completely mixed with the opposite class, we bound their contribution to the solution. Doing so we regularize the solution. In the final solution, all those points will have their α value fixed to C : they will be constraint and will influence the solution,

[I_0] the group of **inactive** points is the one for which $\alpha_i = 0$. These points are also lying on the box boundaries. In this case the vectors are far from the frontier between the classes. In the final solution, all those points will have their α value fixed to 0 : they will be constraint but will not influence the solution.

These three groups lead to another formulation of the optimization problem :

$$\begin{cases} & \max\limits_{\boldsymbol{\alpha}\in\mathbf{R}^m} & -\frac{1}{2}\boldsymbol{\alpha}^\top G\boldsymbol{\alpha} + \mathbf{e}^\top\boldsymbol{\alpha} & \\ \text{with} & & \boldsymbol{\alpha}^\top\mathbf{y} = 0 & \\ \text{and} & & 0 \leq \alpha_i \leq C & i = I_s \\ \text{and} & & \alpha_i = 0 & i \in I_0 \\ \text{and} & & \alpha_i = C & i \in I_c \end{cases} \tag{4}$$

The relation between the dual parameters obtained in equation 4 and the primal parameters (equation 1) is important. We can show it by writing the lagrangian of the dual problem (equation 4) as a minimization problem :

$$\mathcal{L}(\boldsymbol{\alpha}, \lambda, \boldsymbol{\mu}, \boldsymbol{\nu}) = \tfrac{1}{2}\boldsymbol{\alpha}^\top G\boldsymbol{\alpha} - \mathbf{e}^\top\boldsymbol{\alpha} - \lambda\boldsymbol{\alpha}^\top\mathbf{y} - \boldsymbol{\nu}^\top\boldsymbol{\alpha} + \boldsymbol{\mu}^\top(\boldsymbol{\alpha} - C\mathbf{e}) \tag{5}$$

where the Lagrange multipliers $\boldsymbol{\alpha}, \boldsymbol{\mu}$ and $\boldsymbol{\nu}$ have to be positive. This lagrangian leads us back to the primal problem and can be compared to the one in equation 2 when replacing f by $\boldsymbol{\alpha}$:

$$\mathcal{L}(\boldsymbol{\alpha}, b, \boldsymbol{\xi}, \boldsymbol{\beta}) = \tfrac{1}{2}\boldsymbol{\alpha}^\top G\boldsymbol{\alpha} - \mathbf{e}^\top\boldsymbol{\alpha} - b\boldsymbol{\alpha}^\top\mathbf{y} + \boldsymbol{\xi}^\top(\boldsymbol{\alpha} - \boldsymbol{\beta} + C\mathbf{e}) \tag{6}$$

We consider three different cases (I_s, I_0 et I_c) to find the equivalence between the parameters.

[$0 < \alpha < C$]: in this case the Kuhn-Tucker condition that makes the lagrangian's gradient with respect to $\boldsymbol{\alpha}$ vanishing is writen : $G\boldsymbol{\alpha} + \lambda\mathbf{y} - \mathbf{e} = 0$. We also have the initial constraint saturated, that is to say $y_i(f(\mathbf{x}_i) + b) = 1$. We can re-write it as $G\boldsymbol{\alpha} + b\mathbf{y} - \mathbf{e} = 0$. The Lagrange multiplier λ associated to the equality constraint is thus equal to the bias $\lambda = b$.

[$\alpha = 0$]: in this case $\boldsymbol{\xi} = \boldsymbol{\mu} = 0$ and $\boldsymbol{\nu} = G\boldsymbol{\alpha} + b\mathbf{y} - \mathbf{e}$. If the Lagrange multiplier is positive, the model's prediction minus one has to be positive too.

[$\alpha = C$]: in this case $\boldsymbol{\xi} \neq 0, \boldsymbol{\nu} = 0$ and $\boldsymbol{\mu} = -G\boldsymbol{\alpha} - b\mathbf{y} + \mathbf{e} = \boldsymbol{\xi}$. If the Lagrange multiplier is positive, the slack variable is positive too. This imposes to the model's prediction minus one to be negative in that case.

These relations are summarized in table 1. Checking these optimality conditions is equivalent to computing $G\boldsymbol{\alpha} + b\mathbf{y} - \mathbf{e}$ and then making sure that on the one hand for each non support vector ($\alpha = 0$) this quantity is positive and on the other hand for each bounded point ($\alpha = C$) this quantity is negative.
To solve this problem in a efficient way it is clever to take into account the points situation regarding the constraints that are satisfied or not.

Set	Initial constraints	Primal constraints	Dual constraints
I_0	$y_i(f(\mathbf{x}_i)+b) > 1$	$\alpha = 0, \quad \xi = 0$	$\nu > 0, \quad \mu = 0$
I_c	$y_i(f(\mathbf{x}_i)+b) = 1-\xi_i$	$\alpha = C, \quad \xi > 0$	$\nu = 0, \quad \mu > 0$
I_s	$y_i(f(\mathbf{x}_i)+b) = 1$	$0 < \alpha < C, \quad \xi = 0$	$\nu = 0, \quad \mu = 0$

Table 1: Constraints for the three type of variables.

2.2 Outlines of the method

The goal of any SVM algorithm is double : The training set has to be split into the three groups. Once this is done, the problem has to be solved. It turns out that this second phase is relatively simpler than the first one. Let us assume that we know the point's repartition (I_s, I_0 et I_c are given) : then the inequality constraints are useless (they are implicitly contained in the definition of the three groups). The only α_i that remain unknown are for $i \in I_s$. Indeed by definition $\alpha_i = 0$ for $i \in I_0$ and $\alpha_i = C$ for $i \in I_c$. The remaining α_i are found resolving the following optimization problem :

$$\begin{cases} \max\limits_{\boldsymbol{\alpha}\in\mathbf{R}^{|I_s|}} & -\frac{1}{2}\boldsymbol{\alpha}^\top G_s \boldsymbol{\alpha} + \mathbf{e}_s^\top \boldsymbol{\alpha} \\ \text{with} & \boldsymbol{\alpha}^\top \mathbf{y}_s + C\mathbf{e}_c^\top \mathbf{y}_C = 0 \end{cases} \tag{7}$$

with $\mathbf{e}_s = \mathbf{e}(I_s) + 2CG(I_s, I_c)\mathbf{e}(I_c)$, $G_s = G(I_s, I_s)$, $\mathbf{y}_s = \mathbf{y}(I_s)$, $\mathbf{y}_c = \mathbf{y}(I_c)$ and $\mathbf{e}_c$ is a vector of ones. Note here that the dimension of the problem is the cardinal of the set I_s (that can be smaller than m, the initial dimension of the problem). The Khun-Tucker conditions gives the system to solve that will provide the coefficients $\boldsymbol{\alpha}$ that are still unknown :

$$\begin{pmatrix} G_s & \mathbf{y}_s \\ \mathbf{y}_s^\top & 0 \end{pmatrix} \begin{pmatrix} \boldsymbol{\alpha} \\ \lambda \end{pmatrix} = \begin{pmatrix} \mathbf{e}_s \\ -C\mathbf{e}_c^\top \mathbf{y}_c \end{pmatrix} \tag{8}$$

If the solution given by this system contained a component violating the constraints (a negative or above C component), then it would mean that the initial points repartition I_s, I_0 and I_c was wrong. This component has then to be removed from I_s and be put in I_0 or I_c.
In the case all the components $\boldsymbol{\alpha}$ fit the constraints, it does not necessarily mean that we have found the optimum solution. We still have to check that the positivity constraints on the associated Lagrange multipliers (*cf.* previous paragraph) are respected. To do so we check that for each point belonging to I_0, $G\boldsymbol{\alpha} + b\mathbf{y} - \mathbf{e} > 0$, and that for each point belonging to I_c, $G\boldsymbol{\alpha} + b\mathbf{y} - \mathbf{e} < 0$. If it is not, the point that violates the constraints is removed from its group I_0 or I_c and put in I_s.
We have now given the algorithm principle. It is an iterative algorithm which adds or removes points to I_s one by one at each step. We will see that the cost strictly decreases at each step, which guaranties that the method converges. Moreover, the matrices G_s only differs by one row and one column from the previous step. Thus we can compute the new solution from the previous one. Doing so we reduce the complexity of each step from $\mathcal{O}(n^3)$ to $\mathcal{O}(n^2)$.
The algorithm (1) summarizes the steps of *simpleSVM* algorithm.
The algorithm stops when $\widetilde{\boldsymbol{\alpha}}^*$ is admissible and all the points from I_0 and I_c satisfy their constraints. Then no admissible descent direction exists anymore.

Algorithm 1 : *SimpleSVM*

1. $(I_s, I_0, I_c) \leftarrow$ initialise
while minimumReached=FALSE
    2. $(\alpha, \lambda) \leftarrow$ solve the system without constraints(I_s)
    if $\exists \alpha_i \leq 0$ or $\exists \alpha_i \geq C$
        3.1 project α inside the admissible set
        3.2 transfers the associated point from I_s to I_0 or I_c
    else
        4. look for the best candidate x_{cand} in I_c and I_0
        if x_{cand} is found
            5. transfer x_{cand} to I_s
        else
            6. minimumReached $\leftarrow$ TRUE
        end if
    end if
end while

Since the solution depends on the repartition between I_s, I_0 and I_c and since there exists only a finite number of points and thus of combinations, knowing that the cost strictly decreases at each steps, the algorithm cannot loop and reaches the global solution in a finite time.

3 Known bugs

We are waiting for you to report bugs and make comments! One remark: do not hesitate to use the `clear global svModel` or `clear all` if you feel there's something really wrong going on, global variables are somehow... dirty!