# SKMsmo documentation

Guillaume Obozinski

August 21, 2006

This is the documentation for the Matlab code that implements the algorithm of Francis Bach and Gert Lanckriet as described in [1],[2]. The name *SKMsmo* stands for **Support Kernel Machine solved by Sequential Minimal optimization**

### 0.1 In a nutshell

The two main functions are `learn_SKM.m` and `learn_SKM2.m` which are both implementing the SKMsmo algorithm with two different types of input format: the first one assumes that the kernels are stored in files to be read, and the second takes the kernel set as a Matlab matrix. If the code is only given one kernel, the code is automatically switching to the (very similar) SMO algorithm for SVM. You can test the code with the script `code_test.m` which will generate some 50 Mb of kernels and use them to learn two SKMs.

## 1 Main functions

The format for the functions are:

```
[SKM]=learn_SKM(output_dir,filename_prefix,C0,Y,train,test,k1,k2,...,km)
[SKM]=learn_SKM2(C0,Y,train,test,K_stack)
```

**output dir** name of the directory where the output is to be stored. If empty, the results are not written but just returned in the SKM structure.

**filename prefix** is the prefix of the files output by the code in `<output_dir>`

**C0** The regularisation value(s):

- if `C0` is scalar, the same regularization is used for all data points
- if `C0` is a pair of values, the first regularization is applied to the positives (against FN) and the second is applied to the negatives (against FP)
- `C0` can also be a full vector of the same size as the training set or the full set.

**Y** is assumed to be a column vector of the same length as the number of entries in the kernel matrices, containing the $+1, -1$ labels (only training are required, test are used for error measure on test)

**train** is the list of the indices of the training points in Y and the kernel matrices

**test** is the list of the indices of the testing points in Y and in the kernels matrices

**Kstack** is the matrix obtained by stacking on top of one another all the kernels to be used.

**k1,...,km** the names of the files containing the kernels with the following format assumed: each file is tab separated with first row and columns containing headers that are ignored by the code.

## 2 The parameters

Most of the parameters of the algorithm are set in the script `init_parameters.m` which is called before the core code is executed. It should make it convenient for the user to edit that file and modify the settings. *The code is of course be usable without changing these parameters at all.* However, the parameters `acc.tau`, `acc.tau1` and `acc.tau2` are parameters that the user might want to adjust to achieve a better trade-off between speeding-up the algorithm or getting a more precise solution.

Here is a description of the all the parameters found in `init_parameters.m` in order of appearance (see that .m file):

- `d` is a $m \times 1$ vector containing the $d_j$ as described in the paper. They are usually set to 1 assuming the kernels have been normalized by their trace. (To match the SDP formulation of SKM, if $c$ is the constraint on the sum of the traces $\mathrm{tr}(K_j)$ then one should use $d_j = \frac{\mathrm{tr}(K_j)}{c}$. Setting all the $d_j$ to 1 corresponds to an assumption that the different kernels are on the same scale, for instance that they all have traces that are equal or close. A kernel which has a larger scale is otherwise likely to dominate in the combination.

- **The Dekker-Brent parameters** are used by the Dekker-Brent line search algorithm[1] which is a subroutine that performs each coordinate descent step. In general the following parameters shouldn't be changed: `dkb.machine_precision` is an upper bound on the machine precision used to avoid numerical errors, `dkb.maxiter` is the maximal number of bisections/interpolations, `dkb.bisect_acc` is the tolerance used to assess whether a zero has been found (This value should be quite small)

- **The regularization parameters** correspond to the Moreau-Yoshida regularizations that are successively solved to approach the non-differentiable problem. `reg.value` is the initial value of the regularization and matches the parameter $a_j$ in the paper. This value is then decreased iteratively by a multiplicative factor which is `reg.dec_rate`. `reg.min_reg` is the minimal value of the regularization accepted by the algorithm (and beyond which the computations become numerically ill-conditioned). The algorithm stops and returns the best solution found so far if it reaches this value. `reg.iter_max` is the maximal number of decreases of the regularization values allowed.

- **The precision parameters**

  - `acc.tau` is the precision parameter to assess the convergence of each regularization step. The algorithm decides it has found a solution to a regularized problem (`reg optimal`) if the variable `tau_here` is less than `acc.tau`
  - `acc.tau1` and `acc.tau2` are respectively the precision parameters for the slacks $\epsilon_1$ and $\epsilon_2$ in the paper (`eps1` and `eps2` in the code). If both these precisions are met then the non-differentiable problem is considered solved and the code terminates (`OPTIMAL!`)
  - `acc.clip_etas` is the threshold on the values of the unnormalized kernel weights `etas` to set them to zero
  - `acc.clip_alphas` is the parameter to clip the value of $\alpha_i$ to 0 or $C_i$ (setting this parameter to a too small value leads to bad assignments of the datapoints to the sets $I_M$, $I_0^+$, $I_0^-$,$I_C^+$ and $I_C^-$)
  - `acc.Dalphas_res` is the relative size of the smallest step in $\alpha$ acceptable measured by $\frac{\|\alpha^{(t+1)}-\alpha^{(t)}\|}{\|\alpha^{(t)}\|+\epsilon_0}$ If for more than `acc.max_nb_alpha_unchanged` iterations the step are too small according to this precision, the program gives up at the current regularization level
  - `acc.Detas_res` is the desired resolution for the weights: if the weight are not changing at thar resolution for more than `acc.max_nb_etas_unchanged` iterations then the weights are frozen and the algorithm switches to the SMO algorithm for SVM.

[1] The Dekker-Brent method combines the advantages of bisection and interpolation with a second order polynomial to find the zeros of a function, in our case the zero of a directional derivative

- **The counters**
  - `counters.maxiter_per_reg` Sets the maximal number of iterations per regularized problem (message: `over maximal number of iterations`)
  - `counters.global` is the global counter;
- when `option.store_history` is set to 1 the code does some additional computation and bookkeeping to return in the final SKM object the history of the different variables as they evolve during the computation.

# 3 The output

The output is a structure `SKM` which has the following fields:

- `etas`: $\eta$
- `etas_tilde`: $\tilde{\eta}$ the kernel weights
- `alphas`: $\alpha$
- `bias`: $b$
- `eps1`: $\epsilon_1$
- `eps2`: $\epsilon_2$
- `tau`: $\epsilon$ aka `tau`
- `nb_iter`: the value of `counters.global`
- `Y` the labels of the training points
- `duality_gap` the duality gap at the end of the computation ie `obj_ub-obj_lb`
- `discrim`: $w \cdot x + b = \sum_j \eta_j K_j D(y)\alpha + b$
- `outflags` flags that characterize the termination of the algorithm.
- `discrim_test` The value of the SKM on testing points.
- `err_test` The 0-1 loss on the testing set if the true labels of the testing point where given in `Y`.

Other fields available if `option.store_history=1`

- `Gas`: the history of the regularized problems objective values
- `obj`: the history of the objective value
- `obj_ub`: the history of $\frac{1}{2}\max_j \alpha D(y)K_j D(y)\alpha - \mathbf{1}^\top \alpha$
- `obj_lb`: the history of $-\frac{1}{2}\max_j \alpha D(y)K_j D(y)\alpha - C^\top \xi$
- `step_loss`: the history of the 0-1 loss
- `cost`: the history of the hinge loss
- `taus`: the history of $\epsilon$
- `eps1s`: the history of $\epsilon_1$
- `eps2s`: the history of $\epsilon_2$

# References


[1] Francis R. Bach, Gert R. G. Lanckriet, Michael I. Jordan (2004). Multiple Kernel Learning, Conic Duality, and the SMO Algorithm. *Proceedings of the Twenty-first International Conference on Machine Learning*

[2] Bach, F.R., Lanckriet, G.R.G., Jordan, M.I. (2004). Fast Kernel Learning using Sequential Minimal Optimization . Technical Report CSD-04-1307, Division of Computer Science, University of California, Berkeley.