Data Mining 2
TP 2.5 - Multiple Kernel Learning

Thomas Robert

Load data

```
load Data5mixture
n=length(yapp);
```

Initial kernels

On commence par calculer 3 kernels gaussiens différents grâce à la fonction `trainKernel` que l'on écrit.

```
function [K, Kt, G, ypred, nerr] = trainKernel(Xapp, yapp, Xt, yt, kernel, kerneloption, C)

n = length(yapp);
nt = length(yt);

K=svmkernel(Xapp,kernel,kerneloption) ;
G = (yapp*yapp') .*K;
lambda = 1e-12;
e = ones(n,1) ;
[alpha,b,pos] = monqp(G,e,yapp,0,C,lambda,0) ;
Kt = svmkernel(Xt,kernel,kerneloption,Xapp(pos,:)) ;
ypred = Kt*(yapp(pos) .*alpha) + b;
nerr = 100*length(find(yt.*ypred <0) ) /(nt) ;
```

On calcule les kernels.

```
% kernel 1 : gaussian, b=.1, C=100
kernel = 'gaussian';
kerneloption1 = 0.1;
[K1, Kt1, G1, ~, nerr] = trainKernel(Xapp, yapp, Xt, yt, kernel, kerneloption1, 100);
disp(['Erreur kernel 1 : ' num2str(nerr) '%']);

% kernel 2 : gaussian, b=.5, C=100
kerneloption2 = 0.5;
[K2, Kt2, G2, ~, nerr] = trainKernel(Xapp, yapp, Xt, yt, kernel, kerneloption2, 100);
disp(['Erreur kernel 2 : ' num2str(nerr) '%']);

% kernel 3 : gaussian, b=5, C=100
kerneloption3 = 5;
[K3, Kt3, G3, ~, nerr] = trainKernel(Xapp, yapp, Xt, yt, kernel, kerneloption3, 100);
disp(['Erreur kernel 3 : ' num2str(nerr) '%']);
```

```
Erreur kernel 1 : 27.829%
Erreur kernel 2 : 28.4146%
Erreur kernel 3 : 23.8032%
```

Mixin kernel

On calcule un premier mélange de kernels

```
% Build a kernel as the weighted mean of the 3 previously computed kernels
% with weights mu1 = 10 and mu2 = mu3 = 1.
mu = [10 ; 1 ; 1];
mu = mu/sum(mu) ;
```

```
G = (yapp*yapp') .*(mu(1) *K1+mu(2) *K2+mu(3) *K3);

% Train a new SVM on this kernel kernel using monqp
e = ones(n, 1);
C = 100;
lambda = 1e-12;
[alpha ,b,pos] = monqp(G, e, yapp, 0, C, lambda, 0);

% Evaluate the SVM on the test set
Kt1 = svmkernel(Xt,kernel ,kerneloption1 ,Xapp(pos ,:) ) ;
Kt2 = svmkernel(Xt,kernel ,kerneloption2 ,Xapp(pos ,:) ) ;
Kt3 = svmkernel(Xt,kernel ,kerneloption3 ,Xapp(pos ,:) ) ;
Kt = mu(1) *Kt1+mu(2) *Kt2+mu(3) *Kt3;
ypred = Kt*(yapp(pos) .*alpha) + b;

% error rate
nerr = 100*length(find(yt.*ypred <0) ) /(nt) ;
disp(['Erreur mélange initial : ' num2str(nerr) '%']);
```

```
Erreur mélange initial : 18.6649%
```

Gradient iteration

```
% initialize g
g =[alpha'*G1(pos,pos) *alpha;alpha'*G2(pos,pos) *alpha;alpha'*G3(pos, pos) *alpha];
g = g/norm(g) ;
d = g-g(1) ;
d(1) = -sum(d) ;

% iterate
for i=1:50
    g = [alpha'*G1(pos,pos) *alpha ; alpha'*G2(pos,pos) *alpha ; alpha'*G3(pos,pos) *alpha ];
    g = g/norm(g) ;
    d = g-g(1) ;
    d(1) = -sum(d) ;
    step = 0.002; % fix step gradient is bad!
    mu = max(0 ,mu - step*d) ;
    mu = mu/sum(mu) ;
    G = (yapp*yapp') .*(mu(1) *K1+mu(2) *K2+mu(3) *K3) ;
    [alpha ,b,pos] = monqp(G,e,yapp ,0 ,C,lambda ,0) ;
end

% Error rate
Kt1 = svmkernel(Xt,kernel ,kerneloption1 ,Xapp(pos ,:) ) ;
Kt2 = svmkernel(Xt,kernel ,kerneloption2 ,Xapp(pos ,:) ) ;
Kt3 = svmkernel(Xt,kernel ,kerneloption3 ,Xapp(pos ,:) ) ;
Kt = mu(1) *Kt1+mu(2) *Kt2+mu(3) *Kt3;
ypred = Kt*(yapp(pos) .*alpha) + b;
nerr = 100*length(find(yt.*ypred <0) ) /(nt) ;
disp(['Erreur mélange final : ' num2str(nerr) '%']);
```

```
Erreur mélange final : 16.1909%
```

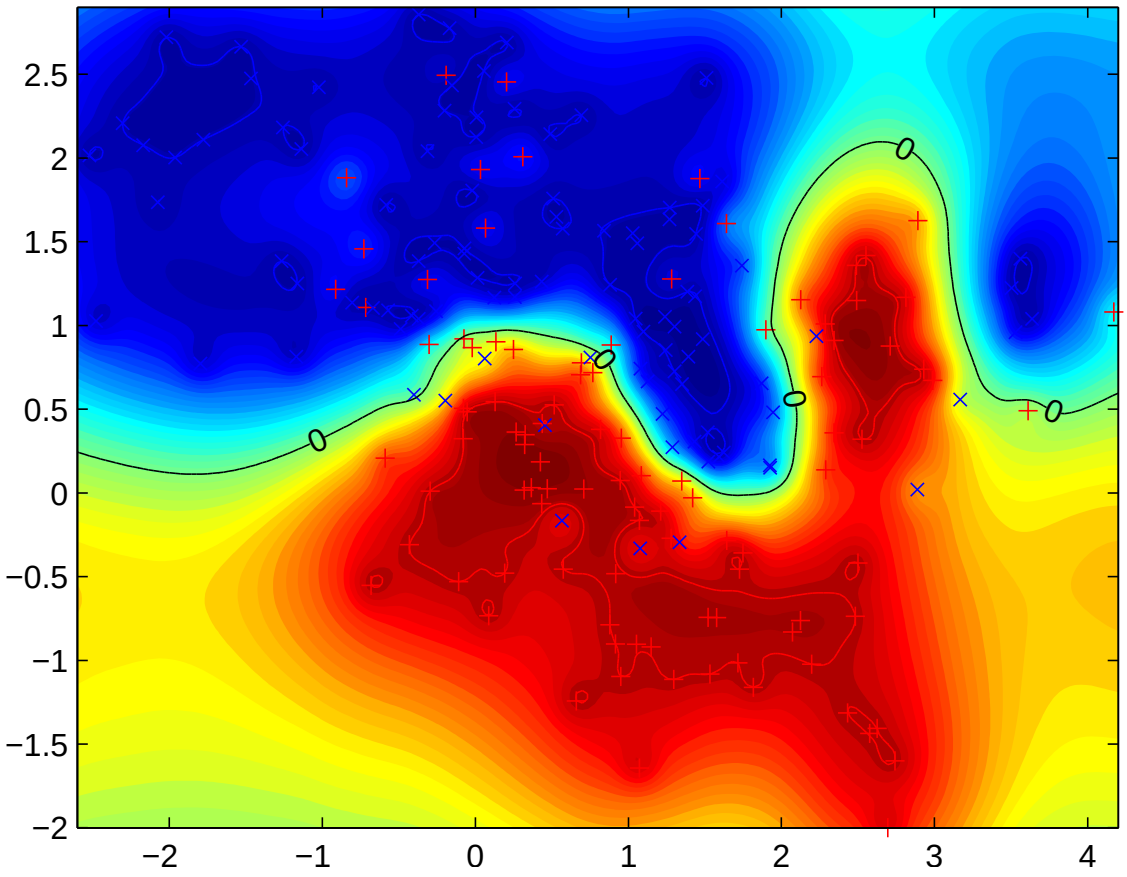
SimpleSKM

```
% Sets parameters
verbose=1;
options.algo= 'svmclass'; % Choice of algorithm in mklsvm can be either
% svmclass  or svmreg
%----------------------------------------------------
% choosing the stopping criterion
%----------------------------------------------------
options.stopvariation=0; % use variation of weights for stopping
options.stopKKT=0; % set to 1 if you use KKTcondition for
options.stopdualitygap=1; % set to 1 for using duality gap for
%----------------------------------------------------
% choosing the stopping criterion value
```

```
%-------------------------------------------------------
options.seuildiffsigma=1e-5; % stopping criterion for weight
options.seuildiffconstraint=0.001; % stopping criterion for KKT
options.seuildualitygap=0.001; % stopping criterion for duality gap
%-------------------------------------------------------
% Setting some numerical parameters
%-------------------------------------------------------
options.goldensearch_deltmax=1e-3; % initial precision of golden
options.numericalprecision=1e-8; % numerical precision weights below
options.lambdareg = 1e-8; % ridge added to kernel matrix
%-------------------------------------------------------
% some algorithms paramaters
%-------------------------------------------------------
options.firstbasevariable= 'first'; % tie breaking method for the base
% variable in the reduced gradient
options.nbitermax=500; % maximal number of iteration
options.seuil=0; % forcing to zero weights lower
options.seuilitermax=10; % value , for iterations lower than
options.miniter=0; % minimal number of iterations
options.verbosesvm=0; % verbosity of inner svm algorithm
options.efficientkernel=0; % use efficient storage of kernels


% Build K for MKL

kernelt={'gaussian' 'gaussian'};
kerneloptionvect={ [kerneloption1 kerneloption2 kerneloption3] [kerneloption1 kerneloption2
    kerneloption3]};
variablevec={'all' 'single'};
classcode=[1 -1];
[nbdata ,dim]=size(Xapp) ;
[kernel ,kerneloptionvec ,variableveccell]=CreateKernelListWithVariable( ...
    variablevec ,dim,kernelt ,kerneloptionvect) ;
[Weight ,InfoKernel]=UnitTraceNormalization(Xapp,kernel ,kerneloptionvec, ...
    variableveccell) ;
K=mklkernel(Xapp,InfoKernel ,Weight ,options) ;

% Train
[beta,w,b,posw,story(i) ,obj(i) ] = mklsvm(K,yapp,C,options ,verbose) ;

% Test
Kt=mklkernel(Xt,InfoKernel ,Weight ,options ,Xapp(posw ,:) ,beta) ;
ypred=Kt*w+b;
nerr = 100*length(find(yt.*ypred <0) ) /(nt) ;
disp(['Erreur mélange avec SimpleMLK : ' num2str(nerr) '%']);

% Plot
[xtest1 xtest2] = meshgrid([ -1:.01:1.2]*3.5 ,[ -1:0.01:1]*3) ;
[nn mm] = size(xtest1) ;
Xtest = [reshape(xtest1 ,nn*mm,1) reshape(xtest2 ,nn*mm,1) ];
Kt=mklkernel(Xtest ,InfoKernel ,Weight ,options ,Xapp(posw ,:) ,beta);
ypred=Kt*w+b;
ypred = reshape(ypred ,nn,mm);
figure(1)
contourf(xtest1 ,xtest2 ,ypred ,50) ;shading flat;hold on;
plot(Xapp(yapp==1 ,1) ,Xapp(yapp==1 ,2) , '+r') ;
plot(Xapp(yapp== -1 ,1) ,Xapp(yapp== -1 ,2) , 'xb') ;
[cc, hh]=contour(xtest1 ,xtest2 ,ypred ,[ -100000 1] , '--r') ;
[cc, hh]=contour(xtest1 ,xtest2 ,ypred ,[ -100000 -1 ] , '--b') ;
[cc, hh]=contour(xtest1 ,xtest2 ,ypred ,[ -100000 0] , 'k') ;
axis([min(Xt(: ,1) ) max(Xt(: ,1) ) min(Xt(: ,2) ) max(Xt(: ,2) ) ]) ;
clabel(cc,hh) ;
```

```
-------------------------------------------------
Iter | Obj.    | DiffBetas | DualGap  | KKT C.   |
-------------------------------------------------
1    | 9126.8656 | 0.3691   | 0.0929 | 0.3020
2    | 9082.5737 | 0.0655   | 0.0041 | 0.0842
3    | 9080.7206 | 0.0305   | 0.0103 | 0.0298
4    | 9080.3480 | 0.0084   | 0.0017 | 0.0108
5    | 9080.2867 | 0.0033   | 0.0008 | 0.0042
Duality gap criteria reached
```

```
Erreur mélange avec SimpleMLK : 9.7643%
```



Résultat de SimpleMKL

Conclusion

On voit que le mélange de kernel est une méthode particulièrement efficace. Elle permet ici, en mélangeant 3 kernels plutôt mauvais (autour de 23 et 27% d'erreur) d'obtenir un résultat final de 9,7%, très bon compte tenu du jeu de données assez bruité.

Comment choisir un kernel ? Comment en construire un ?

Le choix du kernel dépend de la fonction de répartition des données, le but du kernel étant de permettre de passer de n'importe quelle fonction de répartition à un espace dans lequel les points sont linéairement séparable.

Il faut donc comprendre ou faire une pré-analyse des données pour réussir à avoir une idée de quel type de kernel est le plus approprié. Le choix peut parfois être simple quand les données ont peu de variables et que l'on arrive à comprendre facilement le sens de ces variables, mais il peut également être très dur si le nombre de variables augmente et que leur signification est difficile à interpréter.

Si on arrive a comprendre correctement les données et leur réparation, il est possible de construire un kernel selon ses besoins spécifiques. Il suffit pour cela d'écrire la fonction permettant de passer de l'espace de départ des variables vers un nouvel espace dans lequel les classes seront linéairement séparables.

Cela nécessite cependant que les bibliothèques de calcul utilisées supportent le fait de pouvoir utiliser son propre kernel et ne limite pas le choix à une liste de kernels prédéfinis.