Data Mining 2
TP 2.1 SVM & Kernel

Thomas Robert

L'objectif du TP est de tester la méthode du SVM avec kernel qui permet de résoudre un problème non linéaire avec un SVM linéaire en changeant les données d'espace.

Durant le TP, le même code (fourni) a été testé et modifié afin de tester différents paramètres de la méthode. Les résultats de ces modifications seront présentés, le code est présenté en annexe.

1 Temps de calcul

La première version du code avait pour but de comparer les temps de calcul.

On voit bien que les méthodes de calcul optimisées sont plus rapide que les méthodes de calcul généralistes / manuelles.

Par exemple, le calcul du kernel manuel (déjà légèrement optimisé) prend 0,18s alors que la méthode `svnkernel` obtient le même résultat en 0,018s, c'est à dire 10 fois plus rapidement.

Par ailleurs, la résolution du problème de minimisation par `cvx` prend 20.5s contre 0.075s pour `monqp` soit presque 300 fois plus vite. On voit ici que `cvx` est une toolbox très pratique pour la souplesse qu'elle offre dans le formulation du problème de minimisation, mais qu'elle est (et c'est logique) beaucoup moins rapide qu'une méthode de résolution optimisée.

Pour avoir des temps de calcul raisonnable quand on augmente le nombre de données, il faut donc retravailler les problèmes optimisation pour revenir à une forme standard permettant d'utiliser des solveurs optimisés.

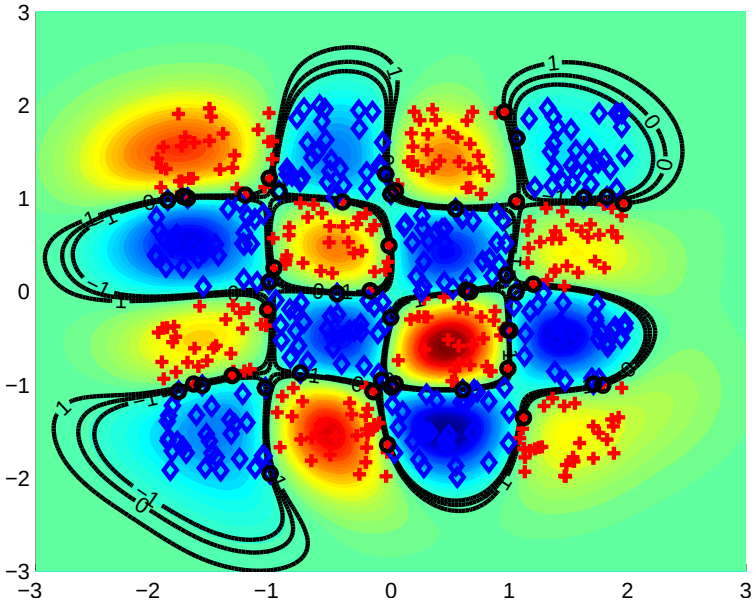


Figure 1 – Résultat du calcul avec 500 données

2 Test avec 10 000 données

On teste cette fois le même code (en ne gardant que les méthodes optimisées) avec 10000 données. On voit que le SVM arrive à bien résoudre le problème de classification et que les frontières tracent bien le quadrillage prévu.

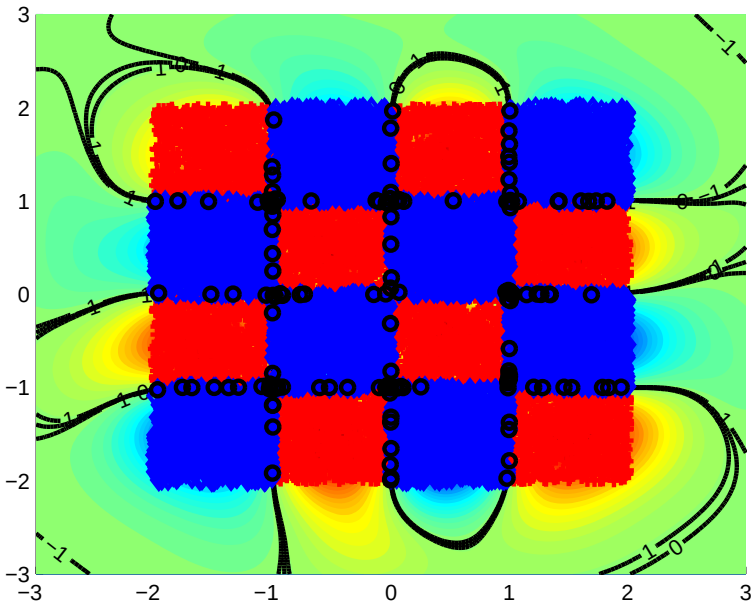


Figure 2 – Résultat du calcul avec 10 000 données

3 Modification de la largeur de bande

Cette fois ci, on modifie la largeur de bande (paramètre du noyau) en la divisant par 10.

On voit que la *bandwidth* règle bien la dispersion des données donnant des classes plus petites (et plus vraiment représentatives de la distribution des données). On est ici trop proche des points et donc mauvais en généralisation.

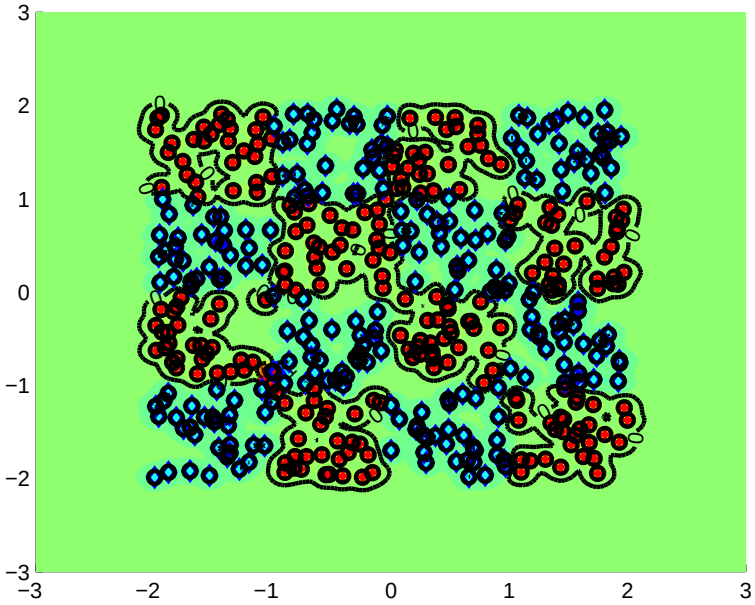
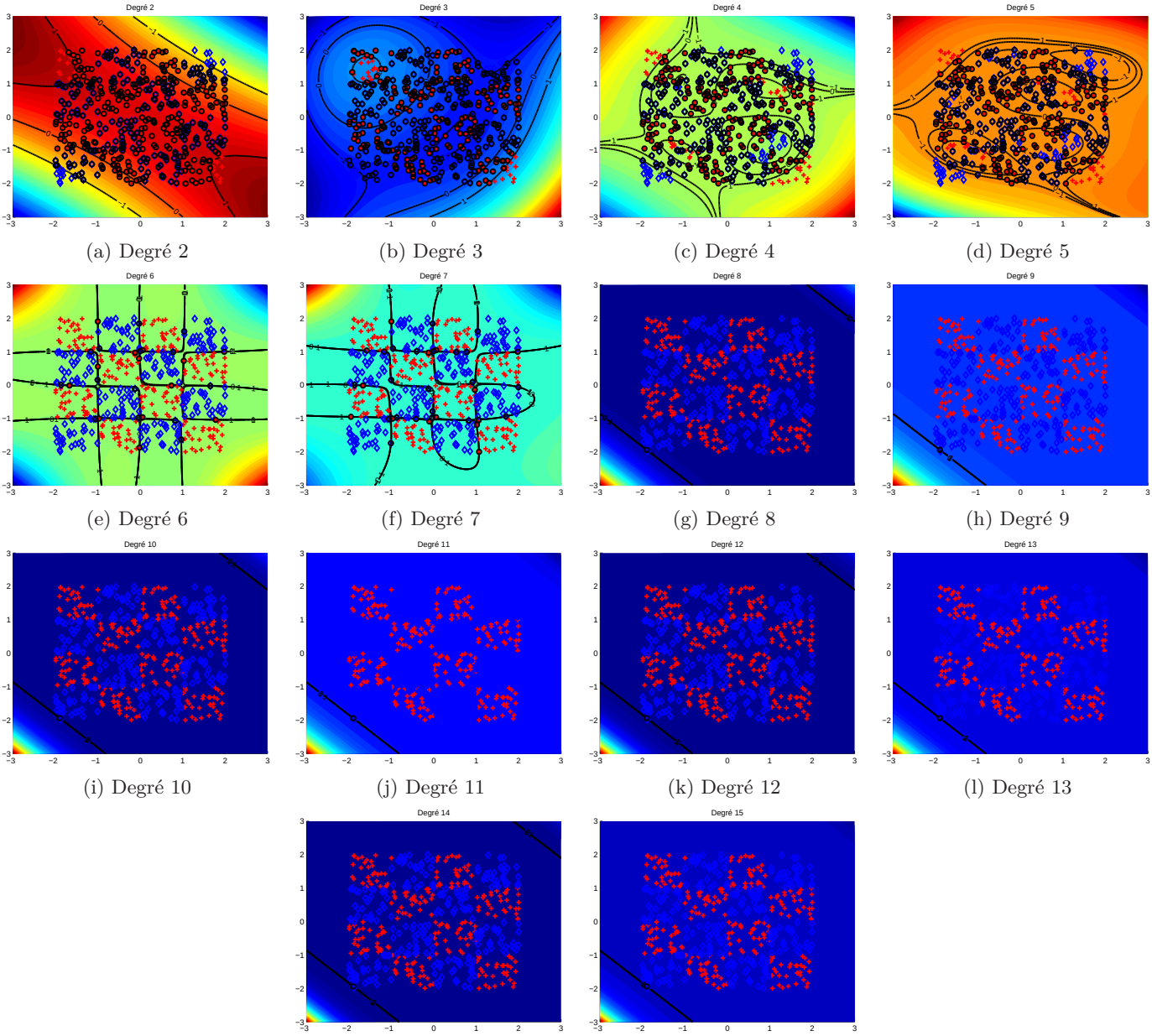


Figure 3 – Résultat avec bandwidth divisée par 10

4 Utilisation d'un kernel polynomial

On essaye maintenant un kernel polynomial au lieu de kernel gaussien utilisé précédemment.

Les degrés de 2 à 5 n'ont pas permis d'obtenir de résultats satisfaisants. Par contre, les degrés 6 et 7 ont permis d'obtenir des résultats satisfaisants. Les degrés supérieurs à 8 ont ensuite "divergés" et n'ont pas permis d'avoir de bons résultats.



(a) Degré 2 (b) Degré 3 (c) Degré 4 (d) Degré 5

(e) Degré 6 (f) Degré 7 (g) Degré 8 (h) Degré 9

(i) Degré 10 (j) Degré 11 (k) Degré 12 (l) Degré 13

(m) Degré 14 (n) Degré 15

Figure 4 – Résultats avec kernel polynomial

5 SVM L2

Enfin, après un peu de maths pour trouver le dual du SVM L2, on a testé ce problème.

La différence de résultat avec le SVM L1 est faible et à l'?il nu, on voit peu de différences. On constate tout de même quelques variations pour un même jeu de données.

Le SVM L2 ayant moins de contraintes, il est plus rapide à résoudre par CVX que le SVM L1 : 5s contre 20s. Pour `monqp` par contre, aucun différence puisque cet algorithme ne "voit" de toute façon pas la disparition de la contrainte.

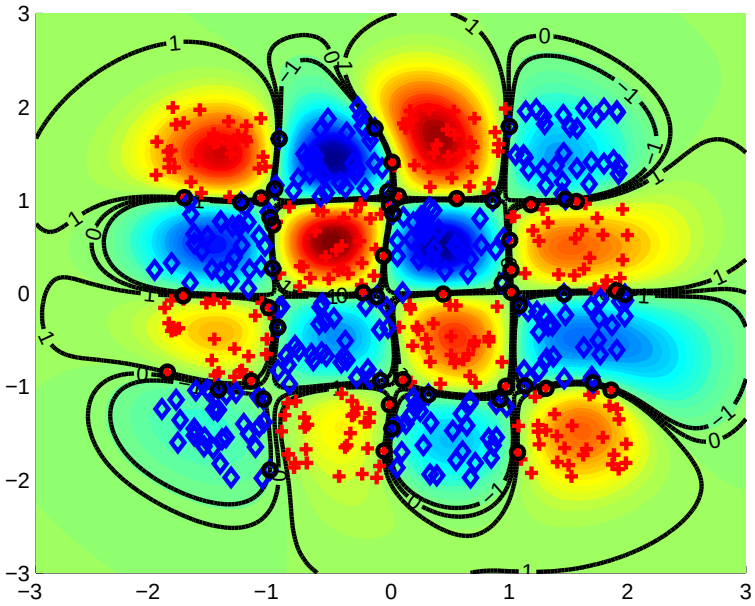


Figure 5 – Résultat de SVM L2

6 Annexes (code Matlab)

6.1 Temps de calcul

```
%% Question 1
%
% Ce code génère un jeu de données en forme de grille d'échec et l'affiche.

n = 500;
sigma=1.4;
[Xapp,yapp,Xtest ,ytest]=dataset_KM('checkers',n,n^2 ,sigma) ;
[n,p] = size(Xapp) ;
figure(1) ;
clf;
set(gcf,'Color',[1 ,1 ,1])
hold on
h1=plot(Xapp(yapp==1 ,1) ,Xapp(yapp==1 ,2) , '+r') ;
set(h1,'LineWidth',2) ;
h2=plot(Xapp(yapp== -1 ,1) ,Xapp(yapp== -1 ,2) ,'db') ;
set(h2,'LineWidth',2) ;

%% Question 2
%
% On calcule le kernel gaussien manuellement puis avec la fonction
% |svmkernel|. On notera le paramètre |kerneloption| qui est l'écart type
% dans la formule du kernel gaussien.

% Compute a gaussian kernel and the matrix on your data with kerneloption =
% .5.
tic
D = (Xapp * Xapp'); % produit scalaire
N = diag(D); % normes
D = -2*D + N*ones(1, n) + ones(n,1) *N'; % Dij = ||t-s||^2 = -2 xi'*xj + ||xi||^2 + ||xj||^2
kerneloption = .5;
s = 2 * kerneloption^2;
K = exp(-D/s);
G = (yapp*yapp') .* K;
toc

% Compute the same gaussian kernel using the svmkernel function of the
% SVMKM toolbox
kernel = 'gaussian';
kerneloption = .5;
tic
K=svmkernel(Xapp, kernel, kerneloption);
G = (yapp*yapp') .* K;
toc

%% 2. c
%
% On résoud le problème de SVM dual avec cvx

e = ones(n,1);
C = 10000;
cvx_begin
    variable a(n)
    dual variables de dp dC
    minimize( 1/2*a'*G*a - e'*a )
    subject to
        de : yapp'*a == 0;
        dp : a >= 0;
        dC : a <= C;
cvx_end

%% 2.d
%
% On résoud cette fois le problème avec le solveur de problème quadratique
% |monqp|

tic
lambda = eps^.5;
[alpha ,b,pos] = monqp(G,e,yapp ,0 ,C,lambda ,0) ;
toc

```

```
%% Question 3
%
% Affichage du résultat avec un |meshgrid|

[xtest1 xtest2] = meshgrid([ -1:.01:1]*3 ,[ -1:0.01:1]*3) ;

nn = length(xtest1);
Xgrid = [reshape(xtest1, nn*nn,1) reshape(xtest2 ,nn*nn,1) ];
Kgrid = svmkernel(Xgrid ,kernel ,kerneloption ,Xapp(pos ,:) ) ;
ypred = Kgrid*(yapp(pos) .*alpha) + b;
ypred = reshape(ypred,nn,nn);
contourf(xtest1 ,xtest2 ,ypred ,50) ; shading flat;
hold on;
[cc,hh]=contour(xtest1 ,xtest2 ,ypred ,[ -1 0 1] , 'k') ;
clabel(cc,hh) ;
set(hh, 'LineWidth', 2) ;
h1=plot(Xapp(yapp==1 ,1), Xapp(yapp==1 ,2) , '+r' , 'LineWidth' ,2) ;
h2=plot(Xapp(yapp== -1 ,1) ,Xapp(yapp== -1 ,2) , 'db' , 'LineWidth' ,2) ;
xsup = Xapp(pos ,:) ;
h3=plot(xsup(: ,1) ,xsup(: ,2) , 'ok', 'LineWidth',2) ;
axis([ -3 3 -3 3]) ;


%% Conclusion sur les temps de calcul
%
% On voit bien que les méthodes de calcul optimisées sont plus rapide que
% les méthodes de calcul généralistes / manuelles.
%
% Par exemple, le calcul du kernel manuel (déjà légèrement optimisé) prend
% 0,18s alors que la méthode |svnkernel| obtient le même résultaten 0,018s,
% c'est à dire 10 fois plus rapidement.
%
% Par ailleurs, la résolution du problème de minimisation par |cvx| prend
% 20.5s contre 0.075s pour |monqp|. On voit ici que |cvx| est une toolbox
% très pratique pour la souplesse qu'elle offre dans le formulation du
% problème de minimisation, mais qu'elle (et c'est logique) beaucoup moins
% rapide qu'une méthode de résolution optimisée.
%
% Pour avoir des temps de calcul raisonnable quand on augmente le nombre de
% données, il faut donc retravailler les problèmes optimisation pour
% revenir à une forme standard permettant d'utiliser des solveurs
% optimisés.
```

6.2 Test avec 10 000 données

```
%% Question 1
%
% Ce code génère un jeu de données en forme de grille d'échec et l'affiche.

n = 10000;
sigma=1.4;
[Xapp,yapp,Xtest ,ytest]=dataset_KM('checkers',n,n^2 ,sigma) ;
[n,p] = size(Xapp) ;
figure(1) ;
clf;
set(gcf,'Color',[1 ,1 ,1])
hold on
h1=plot(Xapp(yapp==1 ,1) ,Xapp(yapp==1 ,2) , '+r') ;
set(h1,'LineWidth',2) ;
h2=plot(Xapp(yapp== -1 ,1) ,Xapp(yapp== -1 ,2) ,'db') ;
set(h2,'LineWidth',2) ;

%% Question 2
%
% On calcule le kernel gaussien manuellement puis avec la fonction
% |svmkernel|. On notera le paramètre |kerneloption| qui est l'écart type
% dans la formule du kernel gaussien.

% Compute the same gaussian kernel using the svmkernel function of the
% SVMKM toolbox
kernel = 'gaussian';
kerneloption = .5;
K=svmkernel(Xapp, kernel, kerneloption);
G = (yapp*yapp') .* K;

```

```
%% 2.d
%
% On résoud cette fois le problème avec le solveur de problème quadratique
% |monqp|

e = ones(n,1);
C = 10000;
lambda = eps^.5;
[alpha ,b,pos] = monqp(G,e,yapp ,0 ,C,lambda ,0) ;

%% Question 3
%
% Affichage du résultat avec un |meshgrid|

[xtest1 xtest2] = meshgrid([ -1:.01:1]*3 ,[ -1:0.01:1]*3) ;

nn = length(xtest1);
Xgrid = [reshape(xtest1, nn*nn,1) reshape(xtest2 ,nn*nn,1) ];
Kgrid = svmkernel(Xgrid ,kernel ,kerneloption ,Xapp(pos ,:) ) ;
ypred = Kgrid*(yapp(pos) .*alpha) + b;
ypred = reshape(ypred,nn,nn);
contourf(xtest1 ,xtest2 ,ypred ,50) ; shading flat;
hold on;
[cc,hh]=contour(xtest1 ,xtest2 ,ypred ,[ -1 0 1] , 'k') ;
clabel(cc,hh) ;
set(hh, 'LineWidth', 2) ;
h1=plot(Xapp(yapp==1 ,1), Xapp(yapp==1 ,2) , '+r' , 'LineWidth' ,2) ;
h2=plot(Xapp(yapp== -1 ,1) ,Xapp(yapp== -1 ,2) , 'db' , 'LineWidth' ,2) ;
xsup = Xapp(pos ,:) ;
h3=plot(xsup(: ,1) ,xsup(: ,2) , 'ok', 'LineWidth',2) ;
axis([ -3 3 -3 3]) ;

%%
% On utilise les méthodes les plus rapides uniquement compte tenu du nombre
% de données conséquent.
```

6.3 Modification de la largeur de bande

```
%% Question 1
%
% Ce code génère un jeu de données en forme de grille d'échec et l'affiche.

n = 500;
sigma=1.4;
[Xapp,yapp,Xtest ,ytest]=dataset_KM('checkers',n,n^2 ,sigma) ;
[n,p] = size(Xapp) ;
figure(1) ;
clf;
set(gcf,'Color',[1 ,1 ,1])
hold on
h1=plot(Xapp(yapp==1 ,1) ,Xapp(yapp==1 ,2) , '+r') ;
set(h1,'LineWidth',2) ;
h2=plot(Xapp(yapp== -1 ,1) ,Xapp(yapp== -1 ,2) ,'db') ;
set(h2,'LineWidth',2) ;

%% Question 2
%
% On calcule le kernel gaussien manuellement puis avec la fonction
% |svmkernel|. On notera le paramètre |kerneloption| qui est l'écart type
% dans la formule du kernel gaussien.

% Compute the same gaussian kernel using the svmkernel function of the
% SVMKM toolbox
kernel = 'gaussian';
kerneloption = .05;
K=svmkernel(Xapp, kernel, kerneloption);
G = (yapp*yapp') .* K;

%% 2.d
%
% On résoud cette fois le problème avec le solveur de problème quadratique
% |monqp|

e = ones(n,1);
C = 10000;
```

```
lambda = eps^.5;
[alpha ,b,pos] = monqp(G,e,yapp ,0 ,C,lambda ,0) ;

%% Question 3
%
% Affichage du résultat avec un |meshgrid|

[xtest1 xtest2] = meshgrid([ -1:.01:1]*3 ,[ -1:0.01:1]*3) ;

nn = length(xtest1);
Xgrid = [reshape(xtest1, nn*nn,1) reshape(xtest2 ,nn*nn,1) ];
Kgrid = svmkernel(Xgrid ,kernel ,kerneloption ,Xapp(pos ,:) ) ;
ypred = Kgrid*(yapp(pos) .*alpha) + b;
ypred = reshape(ypred,nn,nn);
contourf(xtest1 ,xtest2 ,ypred ,50) ; shading flat;
hold on;
[cc,hh]=contour(xtest1 ,xtest2 ,ypred ,[ -1 0 1] , 'k') ;
clabel(cc,hh) ;
set(hh, 'LineWidth', 2) ;
h1=plot(Xapp(yapp==1 ,1), Xapp(yapp==1 ,2) , '+r' , 'LineWidth' ,2) ;
h2=plot(Xapp(yapp== -1 ,1) ,Xapp(yapp== -1 ,2) , 'db' , 'LineWidth' ,2) ;
xsup = Xapp(pos ,:) ;
h3=plot(xsup(: ,1) ,xsup(: ,2) , 'ok', 'LineWidth',2) ;
axis([ -3 3 -3 3]) ;

%%
%
% On voit que la /bandwidth/ règle bien la dispersion des données donnant
% des classes plus petites (et plus vraiment représentatives de la
% ditsribution des données).
```

6.4 Utilisation d'un kernel polynomial

```
%% Question 1
%
% Ce code génère un jeu de données en forme de grille d'échec et l'affiche.

n = 500;
sigma=1.4;
[Xapp,yapp,Xtest ,ytest]=dataset_KM('checkers',n,n^2 ,sigma) ;
[n,p] = size(Xapp) ;


for kerneloption = 2:15

    % Affichage
    figure ;
    clf;
    set(gcf,'Color',[1 ,1 ,1])
    hold on
    h1=plot(Xapp(yapp==1 ,1) ,Xapp(yapp==1 ,2) , '+r') ;
    set(h1,'LineWidth',2) ;
    h2=plot(Xapp(yapp== -1 ,1) ,Xapp(yapp== -1 ,2) ,'db') ;
    set(h2,'LineWidth',2) ;
    title(['Degré ' int2str(kerneloption)]);

    %% Question 2
    %
    % On calcule le kernel gaussien manuellement puis avec la fonction
    % |svmkernel|. On notera le paramètre |kerneloption| qui est l'écart type
    % dans la formule du kernel gaussien.

    % Compute a gaussian kernel and the matrix on your data with kerneloption =
    % .5.
    % D = (Xapp * Xapp'); % produit scalaire
    % N = diag(D); % normes
    % D = -2*D + N*ones(1, n) + ones(n,1) *N'; % Dij = ||t-s||^2 = -2 xi'*xj + ||xi||^2 + ||xj||^2
    % kerneloption = .5;
    % s = 2 * kerneloption^2;
    % K = exp(-D/s);
    % G = (yapp*yapp') .* K;

    % Compute the same gaussian kernel using the svmkernel function of the
    % SVMKM toolbox
    kernel = 'poly';
```

```
    K=svmkernel(Xapp, kernel, kerneloption);
    G = (yapp*yapp') .* K;

    %% 2.d
    %
    % On résoud cette fois le problème avec le solveur de problème quadratique
    % |monqp|

    lambda = eps^.5;
    [alpha ,b,pos] = monqp(G,e,yapp ,0 ,C,lambda ,0) ;

    %% Question 3
    %
    % Affichage du résultat avec un |meshgrid|

    [xtest1 xtest2] = meshgrid([ -1:.01:1]*3 ,[ -1:0.01:1]*3) ;

    nn = length(xtest1);
    Xgrid = [reshape(xtest1, nn*nn,1) reshape(xtest2 ,nn*nn,1) ];
    Kgrid = svmkernel(Xgrid ,kernel ,kerneloption ,Xapp(pos ,:) ) ;
    ypred = Kgrid*(yapp(pos) .*alpha) + b;
    ypred = reshape(ypred,nn,nn);
    contourf(xtest1 ,xtest2 ,ypred ,50) ; shading flat;
    hold on;
    [cc,hh]=contour(xtest1 ,xtest2 ,ypred ,[ -1 0 1] , 'k') ;
    clabel(cc,hh) ;
    set(hh, 'LineWidth', 2) ;
    h1=plot(Xapp(yapp==1 ,1), Xapp(yapp==1 ,2) , '+r' , 'LineWidth' ,2) ;
    h2=plot(Xapp(yapp== -1 ,1) ,Xapp(yapp== -1 ,2) , 'db' , 'LineWidth' ,2) ;
    xsup = Xapp(pos ,:) ;
    h3=plot(xsup(: ,1) ,xsup(: ,2) , 'ok', 'LineWidth',2) ;
    axis([ -3 3 -3 3]) ;

end

%%
%
% Les degrés de 2 à 5 n'ont pas permis d'obtenir de résultats satisfaisants.
% Par contre, les degrés 6 et 7 ont permis d'obtenir des résultats
% satisfaisants. Les degrés supérieurs à 8 ont ensuite "divergés" et n'ont
% pas permis d'avoir de bons résultats.
```

6.5 SVM L2

```
%% Question 1
%
% Ce code génère un jeu de données en forme de grille d'échec et l'affiche.

n = 500;
sigma=1.4;
[Xapp,yapp,Xtest ,ytest]=dataset_KM('checkers',n,n^2 ,sigma) ;
[n,p] = size(Xapp) ;
figure(1) ;
clf;
set(gcf,'Color',[1 ,1 ,1])
hold on
h1=plot(Xapp(yapp==1 ,1) ,Xapp(yapp==1 ,2) , '+r') ;
set(h1,'LineWidth',2) ;
h2=plot(Xapp(yapp== -1 ,1) ,Xapp(yapp== -1 ,2) ,'db') ;
set(h2,'LineWidth',2) ;

%% Question 2

% Compute the same gaussian kernel using the svmkernel function of the
% SVMKM toolbox
kernel = 'gaussian';
kerneloption = .5;
K=svmkernel(Xapp, kernel, kerneloption);
G = (yapp*yapp') .* K;

%% 2. c
%
% On résoud le problème de SVM dual avec cvx

e = ones(n,1);
```

```
C = 10000;
cvx_begin
    variable a(n)
    dual variables de dp dC
    minimize( 1/2*a'*(G+1/C)*a - e'*a )
    subject to
        de : yapp'*a == 0;
        dp : a >= 0;
cvx_end

%% 2.d
%
% On résoud cette fois le problème avec le solveur de problème quadratique
% |monqp|

e = ones(n,1);
C = 10000;
lambda = eps^.5;
[alpha ,b,pos] = monqp(G,e,yapp ,0 ,C,lambda ,0) ;

%% Question 3
%
% Affichage du résultat avec un |meshgrid|

[xtest1 xtest2] = meshgrid([ -1:.01:1]*3 ,[ -1:0.01:1]*3) ;

nn = length(xtest1);
Xgrid = [reshape(xtest1, nn*nn,1) reshape(xtest2 ,nn*nn,1) ];
Kgrid = svmkernel(Xgrid ,kernel ,kerneloption ,Xapp(pos ,:) ) ;
ypred = Kgrid*(yapp(pos) .*alpha) + b;
ypred = reshape(ypred,nn,nn);
contourf(xtest1 ,xtest2 ,ypred ,50) ; shading flat;
hold on;
[cc,hh]=contour(xtest1 ,xtest2 ,ypred ,[ -1 0 1] , 'k') ;
clabel(cc,hh) ;
set(hh, 'LineWidth', 2) ;
h1=plot(Xapp(yapp==1 ,1), Xapp(yapp==1 ,2) , '+r' , 'LineWidth' ,2) ;
h2=plot(Xapp(yapp== -1 ,1) ,Xapp(yapp== -1 ,2) , 'db' , 'LineWidth' ,2) ;
xsup = Xapp(pos ,:) ;
h3=plot(xsup(: ,1) ,xsup(: ,2) , 'ok', 'LineWidth',2) ;
axis([ -3 3 -3 3]) ;

%% Conclusion
%
% En faisant du L2 SVM, on gagne passe de 20s à 5s de temps de calcul par
% CVX car on est moins contraint.
```