Data Mining 2
TP Ensembles de classifieurs

Thomas Robert

1 Le *bootstrap / bagging*

1.1 Principe

La première partie du TP consiste à appliquer la méthode de *bootstrap / bagging*. De manière générale, cette méthode consiste à tirer B jeux d'apprentissage par tirages aléatoires avec remise sur le jeu de données initial. Le résultat final consiste à « mélanger » les B résultats unitaires obtenus pour chaque jeu d'apprentissage afin d'obtenir un résultat plus stable et plus performant en généralisation.

Le « mélange » dépend de ce qu'on évalue. Par *bootstrap* on évalue une valeur (par exemple l'écart-type), le « mélange » consiste donc dans ce cas à faire la moyenne des écarts-type. Par *bagging*, on évalue un classifieur cette fois, le « mélange » pourra par exemple être un vote majoritaire (mieux vaut dans ce cas choisir B astucieusement pour éviter des égalités...).

Notons également que le *bootstrap* permet également de calculer une sorte d'intervalle de confiance sur le résultat que l'on obtient. Il suffit de calculer l'écart-type de l'ensemble des B valeurs trouvées. Pour le *bagging*, un principe similaire permet de calculer l'erreur en test (données non tirées dans l'apprentissage) pour chaque tirage, et l'écart-type de cette erreur.

1.2 Application du *bootstrap* pour le calcul de *skewness*

On commence par estimer la *skewness* d'un jeu de données par *bootstrap*. (Voir annexe 4.1.2). On choisi $B = 50$ tirages comme recommandé tirés par la fonction `tireBootstrap` (voir annexe 4.1.1).

On obtient le résultat suivant : $\gamma = 0.52 \pm 0.072$

1.3 Application du *bagging* pour calcul d'un mélange de kppv et arbre de décision

On applique cette fois la méthode du bagging pour déterminer un mélange de classifieur. On code donc une fonction `[tauxErreurOOBMoy, ecartErreurOOB, tauxErreurTestBagging, tauxErreurTestSansBagging] = baggingEval(X, Y, Xtest, Ytest, B, classifieur, estimateur)` (voir annexe 4.2.1)
dont les paramètres sont :

- Entrée :
  - `X, Y, Xtest, Ytest` : jeu d'apprentissage et de test
  - `B` : nombre d'itération pour le *bagging*
  - `classifieur` : *function handler* prtools vers un classifieur (exemple : `@knnc` ou `@treec`)
  - `estimateur` : *function handler* prtools vers un estimateur de classifieur (exemple : `@knn_map` ou `@tree_map`)
- Sortie :
  - `tauxErreurOOBMoy` : taux d'erreur *out-of-bag* moyen
  - `ecartErreurOOB` : écart type du taux d'erreur *out-of-bag* moyen
  - `tauxErreurTestBagging` : taux d'erreur du mélange de classifieur produit par bagging sur l'ensemble de test
  - `tauxErreurTestSansBagging` : taux d'erreur en test du classifieur calculé seul sur l'ensemble d'apprentissage

Cette méthode à été testée sur divers jeux de données avec divers paramètres (voir annexe 4.2.2), pour les classifieurs *k-plus proches voisins* et *arbre de décision*. On constate qu'en moyenne, le bagging permet d'obtenir un résultat légèrement voire bien meilleur que le classifieur seul.

Le résultat le plus impressionnant est sur le jeu de données *pima* avec la méthode *knn*, où le taux d'erreur en bagging est de 0,3% contre 21,6% sans bagging.

Jeu	Taux erreur OOB	Taux erreur test bagging	Taux erreur test sans bagging
Clown - knn	8.2 ± 6.8 %	$9,3$ %	$10,7$ %
Clown - tree	$8,2 \pm 6,4$ %	$8,6$ %	$9,3$ %
Pima - knn	$32,7 \pm 2,2$ %	$0,37$ %	$21,6$ %
Pima - tree	$34,4 \pm 3,0$ %	0 %	0 %
Satimage - knn	$10,5 \pm 0,64$ %	$9,7$ %	$9,8$ %
Satimage - tree	$18,0 \pm 0,9$ %	$10,5$ %	$17,0$ %

2 Le *Boosting*

2.1 Principe

La seconde méthode de mélange de classifieurs testée est la méthode de *boosting*, qui consiste à pondérer un ensemble de classifieurs et à pondérer les données au fur et à mesure du calcul des classifieurs afin d'avoir en moyenne la bonne estimation.

2.2 Application

On veut tester la *boosting* avec un ensemble de souches binaire.

Afin de tester la méthode d'*AdaBoost*, on code une fonction (voir annexe refboostingEval)
`[YappClassCumule, tauxErreurApp, YtestClassCumule, tauxErreurTest] = AdaBoostEval(Xapp, Yapp, Xtest, Ytest, T)`
dont les paramètres sont les suivants :

- Entrée :
  - `Xapp, Yapp, Xtest, Ytest` : jeu d'apprentissage et de test
  - `T` : nombre d'itérations
- Sortie :
  - `YappClassCumule` : Résultat de l'évaluation sur l'apprentissage au fur et à mesure des itérations
  - `tauxErreurApp` : Taux d'erreur sur le test au fur et à mesure des itérations
  - `YtestClassCumule` : Résultat de l'évaluation sur l'apprentissage au fur et à mesure des itérations
  - `tauxErreurTest` : Taux d'erreur sur le test au fur et à mesure des itérations

On lance cette fonction sur plusieurs jeux de données (voir figure 1). Le fait d'avoir des résultats au fur et à mesure des itérations permet de voir à partir de quand on commence à faire du sur-apprentissage sur le jeu de données.

On constate que sur le jeu de données clown, le boosting est efficace et donne son meilleur résultat en test pour $T = 31$. Par contre, sur le jeu *pima*, le boosting fonctionne très mal et se stabilise à un taux d'erreur plutôt élevé.

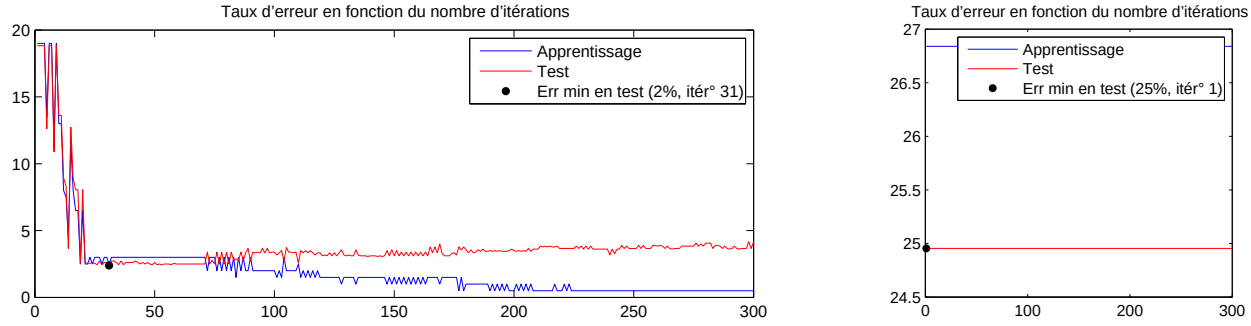


Figure 1 – Résultats du *boosting* sur les jeux *clown* (gauche) et *pima* (droite)

3 Les *Random Forests*

3.1 Principe

Le but des forêts aléatoires est de stabiliser les arbres de décision. Pour cela, on ajoute de l'aléatoire aux données d'apprentissage et on construit donc plusieurs arbres différents afin de construire une « forêt » d'arbres de décision plus robuste en généralisation qu'un arbre de décision seul, qui est lui instable.

Il y de nombreuses façon d'ajouter de l'aléatoire lors de la construction d'une forêt aléatoire. Dans notre cas, pour chaque arbre nous allons choisir un sous-ensemble des composantes initiales et faire un tirage *bootstrap* des données qui serviront à faire l'apprentissage.

3.2 Mise en oeuvre

Le codage a été fait en plusieurs étape : codage d'une fonction de calcul et d'évaluation d'arbre de décision (voir annexe 4.4), codage d'une fonction de calcul et d'évaluation de forêt aléatoire, et enfin test sur 2 jeux de test (voir annexe 4.5).

Il est important de noter que la construction d'une forêt aléatoire se fait à partir de 6 paramètres :

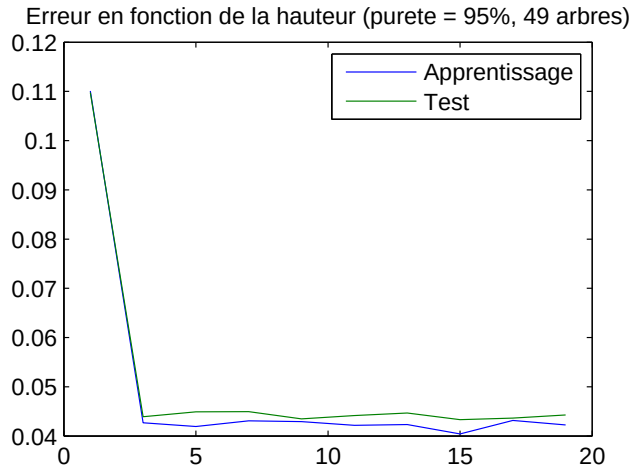
- `X, Y` : Données d'apprentissage
- `hauteurMax` : La hauteur maximale d'un arbre
- `pureteSeuil` : La pureté minimale à atteindre avant de s'arrêter (sauf si `hauteurMax` dépassée)
- `nbArbres` : Le nombre d'arbres dans la forêt
- `nbFeaturesParArbre` : Le nombre de composantes utilisées par arbre

Ces divers paramètres ont donc dû être testés. Pour cela, j'ai choisi de faire varier ces paramètres un par un et de tracer l'évolution de l'erreur en fonction du paramètre testé. (Pour stabiliser la mesure, on fait 20 fois la génération de la forêt avec les même paramètres et on moyenne les 20 mesures d'erreur.)

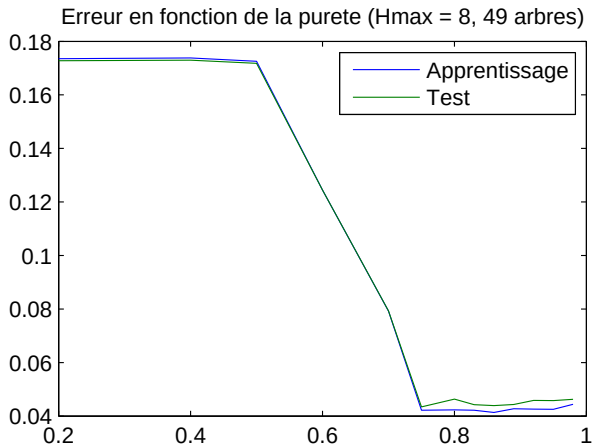
3.3 Résultats

3.3.1 Jeu de données *clown*

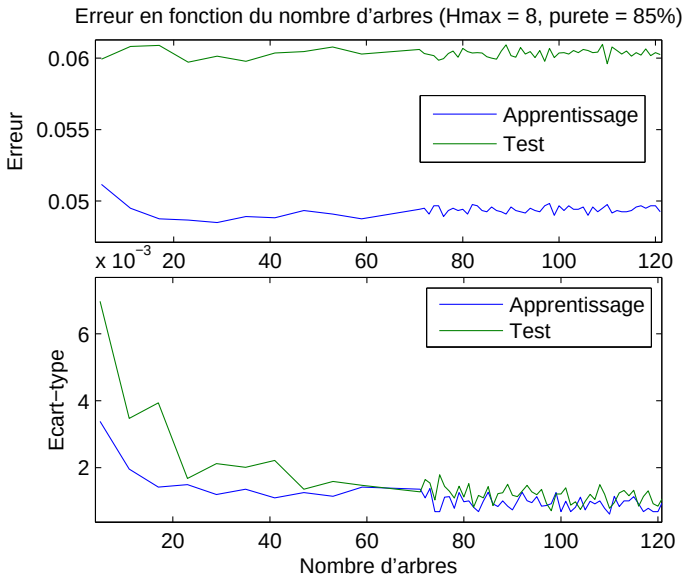
On teste la méthode avec divers paramètres et on obtient les résultats suivants :



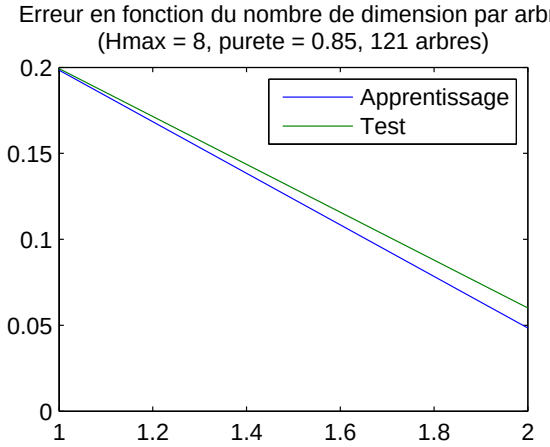
Hauteur Une hauteur supérieure à 3 semble suffisante pour avoir de bons résultats. On choisira 8 pour la suite.



Pureté On voit apparaitre un palier de $0,2$ à $0,5$ puis une chute relativement linéaire et un nouveau plateau de $0,75$ à 1. On choisira donc pour la suite une pureté minimale de 0,85.



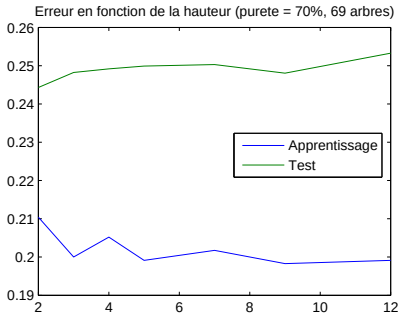
Nombre d'arbres Le nombre d'arbre a peu d'impact sur les résultats moyens. Par contre, on remarque que l'écart-type indique que l'erreur varie beaucoup d'une forêt à l'autre s'il n'y a pas assez d'arbres dans la forêt pour stabiliser l'aléatoire introduit. A partir de 50 arbres, le résultat commence à être relativement stable.



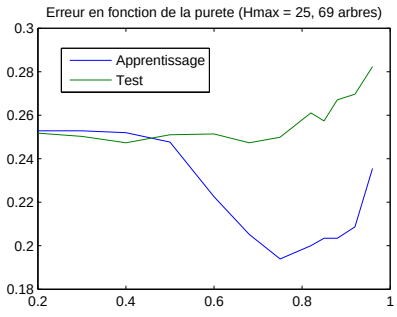
Nombre de dimensions Nous n'avons ici que 2 dimensions, il semble donc étrange de n'en garder qu'une seule par arbre. Cela à quand même été essayé avec 121 arbres dans la forêt (moins de dimensions à la fois nécessite logiquement plus d'arbres). Le résultat indique qu'il est clairement mieux de choisir de garder les 2 dimensions.

Conclusion Finalement, on choisi dont une hauteur maximale de **8 niveaux**, une pureté minimale de **85%**, **49 arbres** et **2 dimensions** par arbre pour ce jeu de données. On obtient une erreur de **4,9% en apprentissage** et de **5,9% en test**. La proximité entre ces deux valeurs montre une bonne robustesse à la généralisation.

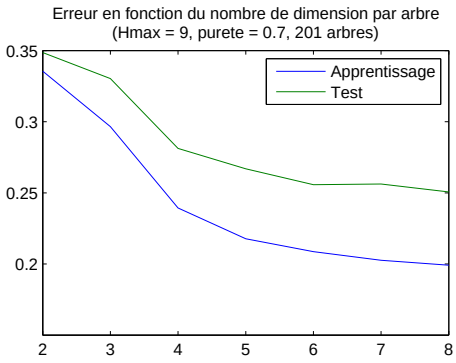
3.3.2 Jeu de données *pima*



Hauteur L'erreur évolue peu en fonction du nombre de niveaux. On choisi 9 maximum dans l'arbre pour la suite.



Pureté L'erreur en test dépend étrangement peu de la pureté. On remarque du sur-apprentissage pour une pureté supérieur à 75%. On choisira 70% de pureté pour la suite.



Nombre de dimensions Il y a 8 dimensions, il pourrait donc être intéressant de ne pas toutes les sélectionner. On voit cependant que les résultats restent meilleurs avec les 8 dimensions. Cependant, l'erreur varie relativement peu entre 5 et 8 dimensions.

Conclusion Finalement, on choisi dont une hauteur maximale de **9 niveaux**, une pureté minimale de **65%**, **69 arbres** et **6 dimensions** par arbre pour ce jeu de données. On obtient une erreur de **20% en apprentissage** et de **25% en test**.

4 Annexes (code Matlab)

Contents

4.1 Bootstrap

4.1.1 Tirage de bootstrap

```
% Tire des indices de bootstrap
%
% n : nombre d'éléments dans l'ensemble
% m : nombre d'éléments à tirer
function [bag, obag] = tireBootstrap(n, m)

    bag = randi(n, m, 1);
    obag = setdiff(1:n, bag);

end
```

4.1.2 Bootstrap sur Skewness

```
% Chargement CSV
X = csvread('skewnormal.csv');
n = length(X);
B = 50;

% On calcule B skewness
skewVals = zeros(B,1);
for i = 1:B
    skewVals(i) = skewness(X(tireBootstrap(n,n)));
end
skew = mean(skewVals)
errSkew = std(skewVals)
```

4.2 Bagging

4.2.1 Evaluation du bagging

```
function [tauxErreurOOBMoy, ecartErreurOOB, tauxErreurTestBagging, tauxErreurTestSansBagging] = ...
        baggingEval(X, Y, Xtest, Ytest, B, classifieur, estimateur)

nX = length(X);
nXtest = length(Xtest);

```

```
% BAGGING
tauxErreurs = zeros(B,1);
classifieurs = cell(1,B);
for i = 1:B
    % Tirage des jeux
    [bag, obag] = tireBootstrap(nX, nX);
    Xapp = X(bag, :);
    Yapp = Y(bag);
    Xval = X(obag, :);
    Yval = Y(obag);

    % Création jeux
    dataApp = prdataset(Xapp, Yapp);
    dataVal = prdataset(Xval, Yval);

    % Classification
    classifieurs{i} = classifieur(dataApp);
    probas = estimateur(dataVal, classifieurs{i});
    [~, classesVal] = max(probas.data, [], 2);
    tauxErreurs(i) = sum(dataVal.nlab ~= classesVal)/length(classesVal)*100;
end
tauxErreurOOBMoy = mean(tauxErreurs);
ecartErreurOOB = std(tauxErreurs);

% ENSEMBLE DE CLASSIFIEURS
dataTest = prdataset(Xtest, Ytest);
target = dataTest.nlab;

classes = zeros(nXtest, B);
for i = 1:B
    probas = estimateur(dataTest, classifieurs{i});
    [~, classesTest] = max(probas.data, [], 2);
    classes(:,i) = classesTest;
end

Ytarget = dataTest.nlab;
Yclassif = mode(classes,2);

tauxErreurTestBagging = sum(Ytarget ~= Yclassif)/nXtest*100;

% SANS BAGGING

% Création du jeu
dataApp = prdataset(X, Y);
dataTest = prdataset(Xtest, Ytest);

% Classification
classifieurSeul = classifieur(dataApp);
probas = estimateur(dataTest, classifieurSeul);
[~, classesSeul] = max(probas.data, [], 2);
Ytarget = dataTest.nlab;
tauxErreurTestSansBagging = sum(Ytarget ~= classesSeul)/length(classesSeul)*100;
```

4.2.2 Lancement d'évaluations de bagging

```
%% Clown data

clear all

nX=200;
X = zeros(nX,2);

X(1:nX/2   ,:) = randn(nX/2,2) + repmat([0  6], nX/2, 1);
X(nX/2+1:nX,1) = (rand(nX/2,1) - 0.5) * 6;
X(nX/2+1:nX,2) = X(nX/2+1:nX,1).^2 + 0.7*randn(nX/2,1);

Y = [ones(nX/2,1) ; zeros(nX/2,1)];

% découpage en apprentissage et test
[X, Y, Xtest, Ytest] = splitdata(X, Y, 0.3);
nX=length(X);
nXtest=length(Xtest);
plot(X(Y==1,1), X(Y==1,2), '.r'); hold on;
plot(X(Y==0,1), X(Y==0,2), '.b');
plot(Xtest(Ytest==1,1), Xtest(Ytest==1,2), 'or'); hold on;
```

```
plot(Xtest(Ytest==0,1), Xtest(Ytest==0,2), 'ob');
title('Jeu de test');

% Bagging

disp('K plus proches voisins');

[tauxErreurOOBMoy, ecartErreurOOB, tauxErreurTestBagging, tauxErreurTestSansBagging] = ...
    baggingEval(X, Y, Xtest, Ytest, 49, @knnc, @knn_map)

disp('Arbre de decision');

[tauxErreurOOBMoy, ecartErreurOOB, tauxErreurTestBagging, tauxErreurTestSansBagging] = ...
    baggingEval(X, Y, Xtest, Ytest, 49, @treec, @tree_map)

%% Jeu de données pima

data = csvread('pima-indians-diabetes.data');

X = data(:,1:end-1);
Y = data(:,end);

% découpage en apprentissage et test
[Xapp, Yapp, Xtest, Ytest] = splitdata(X, Y, 0.3);

%%
disp('K plus proches voisins');

[tauxErreurOOBMoy, ecartErreurOOB, tauxErreurTestBagging, tauxErreurTestSansBagging] = ...
    baggingEval(X, Y, Xtest, Ytest, 49, @knnc, @knn_map)
%%
disp('Arbre de decision');

[tauxErreurOOBMoy, ecartErreurOOB, tauxErreurTestBagging, tauxErreurTestSansBagging] = ...
    baggingEval(X, Y, Xtest, Ytest, 49, @treec, @tree_map)


%%

dataApp = tblread('sat.trn');
dataTest = tblread('sat.tst');

X = dataApp(:,1:end-1);
Y = dataApp(:,end);
Xtest = dataTest(:,1:end-1);
Ytest = dataTest(:,end);

%%
disp('K plus proches voisins');

[tauxErreurOOBMoy, ecartErreurOOB, tauxErreurTestBagging, tauxErreurTestSansBagging] = ...
    baggingEval(X, Y, Xtest, Ytest, 49, @knnc, @knn_map)
%%
disp('Arbre de decision');

[tauxErreurOOBMoy, ecartErreurOOB, tauxErreurTestBagging, tauxErreurTestSansBagging] = ...
    baggingEval(X, Y, Xtest, Ytest, 49, @treec, @tree_map)


%%
```

4.3 Boosting

4.3.1 Evaluation du boosting

```
function [YappClassCumule, tauxErreurApp, YtestClassCumule, tauxErreurTest] = ...
                                AdaBoostEval(Xapp, Yapp, Xtest, Ytest, T)

% tailles
nXapp = size(Xapp,1);
nXtest = size(Xtest,1);

% initialisations
D = zeros(nXapp, T+1); % poids
```

```
h = cell(T,1); % classifieurs
eps = zeros(T,1); % erreurs pondérées
alpha = zeros(T,1); % pondérations de classifieurs
YappClass = zeros(nXapp, T); % classifications
YtestClass = zeros(nXtest, T); % classifications

% initialisation des poids pour la première itération
D(:,1) = 1/nXapp;

% pour chaque itération
for t=1:T

    % apprentissage du classifieur
    h{t} = souchebinairetrain(Xapp, Yapp, D(:,t));

    % evaluation avec le classifieur
    YappClass(:,t) = souchebinaireval(h{t}, Xapp);

    % calcul de l'erreur pondérée
    eps(t) = sum(D(YappClass(:,t) ~= Yapp, t));

    % calcul du coef de pondération du classifieur
    alpha(t) = 1/2 * log( (1 - eps(t)) / eps(t) );

    % calcul des pondérations pour l'étape suivante
    D(:, t+1) = D(:, t) .* exp( -alpha(t) * YappClass(:,t) .* Yapp );
    D(:, t+1) = D(:, t+1) / sum(D(:, t+1));

    % evaluation avec le classifieur sur l'ensemble de test
    YtestClass(:,t) = souchebinaireval(h{t}, Xtest);
end

% évalution des performances au fur et à mesure des itérations
% en apprentissage
YappClassCumule = sign(cumsum( (YappClass .* repmat(alpha', nXapp, 1)) , 2));
tauxErreurApp = sum(YappClassCumule ~= repmat(Yapp, 1, T))/nXapp * 100;
% en test
YtestClassCumule = sign(cumsum( (YtestClass .* repmat(alpha', nXtest, 1)) , 2));
tauxErreurTest = sum(YtestClassCumule ~= repmat(Ytest, 1, T))/nXtest * 100;
[tauxErreurTestMin, tauxErreurTestMinInd] = min(tauxErreurTest);

% affichage des résultats
figure();
plot(tauxErreurApp, '-b'); hold on;
plot(tauxErreurTest, '-r');
plot(tauxErreurTestMinInd, tauxErreurTestMin, '.k', 'MarkerSize', 15);

title('Taux d''erreur en fonction du nombre d''itérations');
legend('Apprentissage', 'Test', ['Err min en test (' int2str(round(tauxErreurTestMin))...
    '%, itér° ' int2str(round(tauxErreurTestMinInd)) ')'] );
```

4.3.2 Lancement d'évaluations de boosting

```
%% Jeu clown

clear all

nX=2000;
X = zeros(nX,2);

X(1:nX/2   ,:) = randn(nX/2,2) + repmat([0 6], nX/2, 1);
X(nX/2+1:nX,1) = (rand(nX/2,1) - 0.5) * 6;
X(nX/2+1:nX,2) = X(nX/2+1:nX,1).^2 + 0.7*randn(nX/2,1);

Y = [-ones(nX/2,1) ; ones(nX/2,1)];

% découpage en apprentissage et test
[Xapp, Yapp, Xtest, Ytest] = splitdata(X, Y, 0.1);
plot(X(Y==1,1), X(Y==1,2), '.r'); hold on;
plot(X(Y==-1,1), X(Y==-1,2), '.b');
plot(Xtest(Ytest==1,1), Xtest(Ytest==1,2), 'or'); hold on;
plot(Xtest(Ytest==-1,1), Xtest(Ytest==-1,2), 'ob');
title('Jeu de test');

AdaBoostEval(Xapp, Yapp, Xtest, Ytest, 300);
```

```
23
24 %% Jeu pima
25
26 clear all
27
28 data = csvread('pima-indians-diabetes.data');
29
30 X = data(:,1:end-1);
31 Y = data(:,end);
32 Y(Y==0) = -1;
33
34 % découpage en apprentissage et test
35 [Xapp, Yapp, Xtest, Ytest] = splitdata(X, Y, 0.3);
36
37 AdaBoostEval(Xapp, Yapp, Xtest, Ytest, 300);
```

4.4 Arbres de décision

4.4.1 Construction d'un arbre de décision

```
1 function tree = decisionTreeTrain(X, Y, hauteurRestante, pureteSeuil)
2
3 % taille
4 [n, p] = size(X);
5
6 % création d'un noeud
7 tree = struct();
8
9 % décision
10 tree.decision = souchebinairetrain(X, Y, ones(n,1));
11
12 % calcul des résultats dans les fils
13 Yhat = souchebinaireval(tree.decision, X);
14
15 % si des exemples dans le fils gauche ("-1") sont mal classés
16 Xgauche = X(Yhat == -1, :);
17 Ygauche = Y(Yhat == -1);
18 pureteG = abs(sum(Ygauche == 1) - sum(Ygauche == -1))/length(Ygauche);
19
20 if (pureteG < pureteSeuil && hauteurRestante > 0)
21     tree.filsGauche = decisionTreeTrain(Xgauche, Ygauche, hauteurRestante-1, pureteSeuil);
22 end
23
24 % si des exemples dans le fils droit ("1") sont mal classés
25 Xdroit = X(Yhat == 1, :);
26 Ydroit = Y(Yhat == 1);
27 pureteD = abs(sum(Ydroit == 1) - sum(Ydroit == -1))/length(Ydroit);
28
29 if (pureteD < pureteSeuil && hauteurRestante > 0)
30     tree.filsDroit = decisionTreeTrain(Xdroit, Ydroit, hauteurRestante-1, pureteSeuil);
31 end
```

4.4.2 Évaluation d'un arbre de décision

```
1 function Yhat = decisionTreeVal(tree, X)
2
3 Yhat = souchebinaireval(tree.decision, X);
4
5 % si fils gauche existant, recalcul
6 if (isfield(tree, 'filsGauche'))
7     Yhat(Yhat == -1) = decisionTreeVal(tree.filsGauche, X(Yhat == -1, :));
8 end
9
10 % si fils droit existant, recalcul
11 if (isfield(tree, 'filsDroit'))
12     Yhat(Yhat == 1) = decisionTreeVal(tree.filsDroit, X(Yhat == 1, :));
13 end
```

4.5 Forêts aléatoires

4.5.1 Construction d'une forêt aléatoire

```
function forest = randomForestTrain(X, Y, ...
                    hauteurMax, pureteSeuil, ...
                    nbArbres, nbFeaturesParArbre)

% taille donnees
[n,p] = size(X);

forest = cell(nbArbres, 1);

for i = 1:nbArbres

    % features de l'arbre i
    forest{i}.features = randperm(p, nbFeaturesParArbre);

    % tirage du bagging
    [bag, obag] = tireBootstrap(n, n);

    % arbre i
    forest{i}.tree = decisionTreeTrain(...
                        X(bag, forest{i}.features), Y(bag), ...
                        hauteurMax, pureteSeuil);

end
```

4.5.2 Évaluation d'une forêt aléatoire

```
function Yhat = randomForestVal(forest, X)

[n,p] = size(X);
L = length(forest);

Yhats = zeros(n,L);

for i = 1:L
    Yhats(:,i) = decisionTreeVal(forest{i}.tree, X(:,forest{i}.features));
end

Yhat = mode(Yhats,2);
```

4.5.3 Test d'une forêt aléatoire

```
function [tauxErreurTrain, tauxErreurTest, stdTrain, stdTest] = randomForestEval(X, Y, Xtest, Ytest,
    hmax, purete, nbArbres, nbDim, nbIte)

if (nargin < 9)
    nbIte = 20;
end

tauxErreurTrainMat = zeros(1,nbIte);
tauxErreurTestMat = zeros(1,nbIte);

for i=1:nbIte
    forest = randomForestTrain(X, Y, hmax, purete, nbArbres, nbDim);

    Yhat = randomForestVal(forest, X);
    tauxErreurTrainMat(i) = sum(Yhat ~= Y) / length(Y);

    Ytesthat = randomForestVal(forest, Xtest);
    tauxErreurTestMat(i) = sum(Ytesthat ~= Ytest) / length(Ytest);
end

tauxErreurTrain = mean(tauxErreurTrainMat);
tauxErreurTest = mean(tauxErreurTestMat);
stdTrain = std(tauxErreurTrainMat);
stdTest = std(tauxErreurTestMat);
```

4.5.4 Test des forêts aléatoires

```
%% Clown data

clear all

nX=2000;
```

```
X = zeros(nX,2);

X(1:nX/2   ,:) = randn(nX/2,2) + repmat([0  6], nX/2, 1);
X(nX/2+1:nX,1) = (rand(nX/2,1) - 0.5) * 6;
X(nX/2+1:nX,2) = X(nX/2+1:nX,1).^2 + 0.7*randn(nX/2,1);

Y = [ones(nX/2,1) ; -ones(nX/2,1)];

% découpage en apprentissage et test
[X, Y, Xtest, Ytest] = splitdata(X, Y, 0.3);
nX=length(X);
nXtest=length(Xtest);
plot(X(Y==1,1), X(Y==1,2), '.r'); hold on;
plot(X(Y==-1,1), X(Y==-1,2), '.b');
plot(Xtest(Ytest==1,1), Xtest(Ytest==1,2), 'or'); hold on;
plot(Xtest(Ytest==-1,1), Xtest(Ytest==-1,2), 'ob');
title('Jeu de test');

%%

hauteurVals = 1:2:20;
erreurs = zeros(length(hauteurVals), 2) ;

for i = 1:length(hauteurVals)

    hauteur = hauteurVals(i);
    [a,b] = randomForestEval(X, Y, Xtest, Ytest, hauteur, 0.95, 49, 2);
    erreurs(i,:) = [a b];

end

figure
plot(hauteurVals, erreurs);
legend('Apprentissage', 'Test');
title('Erreur en fonction de la hauteur (purete = 95%, 49 arbres)');

% choix = 8

%%

pureteVals = [0.2 0.4 0.5 0.6 0.7 0.75 0.8:0.03:0.98];
erreurs = zeros(length(pureteVals), 2) ;

for i = 1:length(pureteVals)

    purete = pureteVals(i);
    [a,b] = randomForestEval(X, Y, Xtest, Ytest, 8, purete, 49, 2);
    erreurs(i,:) = [a b];

end

figure
plot(pureteVals, erreurs);
legend('Apprentissage', 'Test');
title('Erreur en fonction de la purete (Hmax = 8, 49 arbres)');

% choix = 0.85

%%

nbArbresVals = [5:6:60 71:121];
erreurs = zeros(length(nbArbresVals), 4) ;

for i = 1:length(nbArbresVals)

    nbArbres = nbArbresVals(i);
    [a,b,c,d] = randomForestEval(X, Y, Xtest, Ytest, 8, 0.85, nbArbres, 2);
    erreurs(i,:) = [a b c d];

end

%%

figure
subplot(2,1,1);
```

```
plot(nbArbresVals, erreurs(:,1:2));
legend('Apprentissage', 'Test');
title('Erreur en fonction du nombre d''arbres (Hmax = 8, purete = 85%)');
ylabel('Erreur');

subplot(2,1,2);
plot(nbArbresVals, erreurs(:,3:4));
ylabel('Ecart-type');
xlabel('Nombre d''arbres');
legend('Apprentissage', 'Test');


% choix 35

%%

erreurs = zeros(2, 2) ;

for i = 1:2

    nbArbres = nbArbresVals(i);
    [a,b] = randomForestEval(X, Y, Xtest, Ytest, 8, 0.85, 121, i);
    erreurs(i,:) = [a b];

end

figure
plot(1:2, erreurs);
legend('Apprentissage', 'Test');
title('Erreur en fonction du nombre de dimension par arbre (Hmax = 8, purete = 0.85, 121 arbres)');

% 2 dimensions

%%

[erreurTrain,erreurTest] = randomForestEval(X, Y, Xtest, Ytest, 8, 0.85, 35, 2)

%% Jeu de données pima

data = csvread('pima-indians-diabetes.data');

X = data(:,1:end-1);
Y = data(:,end);
Y(Y==0) = -1;

% découpage en apprentissage et test
[X, Y, Xtest, Ytest] = splitdata(X, Y, 0.3);
p = size(X,2);

%%

hauteurVals = [2 3 4 5  7 9 12];
erreurs = zeros(length(hauteurVals), 2) ;

for i = 1:length(hauteurVals)

    hauteur = hauteurVals(i);
    [a,b] = randomForestEval(X, Y, Xtest, Ytest, hauteur, 0.7, 69, p, 10);
    erreurs(i,:) = [a b];

end

figure
plot(hauteurVals, erreurs);
legend('Apprentissage', 'Test');
title('Erreur en fonction de la hauteur (purete = 70%, 69 arbres)');

% choix = 8

%%

pureteVals = [0.2 0.3 0.4 0.5 0.6 0.68 0.75 0.82 0.85 0.88 0.92 0.96];
erreurs = zeros(length(pureteVals), 2) ;

for i = 1:length(pureteVals)
```

```
156
157     purete = pureteVals(i);
158     [a,b] = randomForestEval(X, Y, Xtest, Ytest, 25, purete, 69, p, 5);
159     erreurs(i,:) = [a b];
160
161 end
162
163 figure
164 plot(pureteVals, erreurs);
165 legend('Apprentissage', 'Test');
166 title('Erreur en fonction de la purete (Hmax = 25, 69 arbres)');
167
168 %%
169
170 nbDimVals = [2 3 4 5 6 7 8];
171 erreurs = zeros(nbDimVals, 2) ;
172
173 for i = 1:length(nbDimVals)
174
175     nbDim = nbDimVals(i);
176     [a,b] = randomForestEval(X, Y, Xtest, Ytest, 9, 0.7, 201, nbDim, 10);
177     erreurs(i,:) = [a b];
178
179 end
180
181 figure
182 plot(nbDimVals, erreurs);
183 legend('Apprentissage', 'Test');
184 title('Erreur en fonction du nombre de dimension par arbre (Hmax = 9, purete = 0.7, 201 arbres)');
185
186 %%
187 [erreurTrain,erreurTest] = randomForestEval(X, Y, Xtest, Ytest, 9, 0.70, 69, 7)
```