ÉCOLE DES HAUTES ÉTUDES COMMERCIALES
AFFILIÉE À L'UNIVERSITÉ DE MONTRÉAL

Comparaison de méthodes ensemblistes

par

Mélanie LABARRE

Sciences de la gestion

Mémoire présenté en vue de l'obtention
du grade de maître ès sciences
(M. Sc)

Juin 2003

HEC MONTRÉAL

Retrait d’une ou des pages pouvant contenir des renseignements personnels

Sommaire

La compagnie Sélection du Reader's Digest offre des produits à ses clients par offres postales. L'entreprise a développé des modèles de prédiction afin de cibler le mieux possible les acheteurs potentiels. Ce mémoire tente de voir si des méthodes ensemblistes composées d'arbres de classification peuvent donner de meilleurs résultats que la méthode de l'entreprise. Quatre méthodes ensemblistes sont testées, le bagging, le boosting, la randomisation et une implantation des forêts aléatoires. Leurs résultats sont comparés avec ceux de la régression logistique et ceux de la méthode utilisée actuellement par l'entreprise. Lorsque validées à l'aide des données ayant servi à construire les modèles, les méthodes ensemblistes s'avèrent plus performantes que la régression logistique et que la méthode de l'entreprise, la meilleure des méthodes ensemblistes étant le boosting. Par contre, lorsque des expériences sont faites pour tester la stabilité des modèles à travers le temps, les méthodes ensemblistes, bien qu'étant toujours supérieures à la régression logistique de base, donnent des résultats comparables à ceux de la méthode de l'entreprise.

Table des matières

Index des tableaux et figures

Revue de la littérature

Introduction

Au cours des dernières années, la possibilité d'emmagasiner et de traiter de plus en plus d'information à moindres coûts a incité les compagnies à se munir de bases de données de plus en plus importantes, ce qui a favorisé l'essor du data mining. Le terme data mining fait référence à diverses techniques permettant d'extraire et d'analyser facilement l'information de bases de données, tels les réseaux de neurones, le clustering et les arbres de classification.

Ce sont les arbres de classification qui font l'objet de cette revue de littérature, plus précisément les techniques permettant d'améliorer leur précision et leur efficacité. La première section décrira brièvement le problème général que les arbres permettent de résoudre : la classification supervisée. La seconde section donnera un bref aperçu de ce qu'est un arbre. La troisième section présentera quelques-unes des méthodes permettant d'améliorer la précision des arbres : le bagging, le boosting, l'injection d'aléatoire et les forêts aléatoires. Finalement, différents domaines du marketing auxquels peuvent être appliqués les arbres seront présentés, de même que certaines des applications des arbres au marketing.

Classification

Un problème de classification supervisée se décrit comme suit : soit un ensemble de données, dont chaque observation est composée d'une part d'une variable cible, la variable que l'on cherche à classifier, et d'autre part de variables explicatives, qui serviront éventuellement à prévoir le comportement de la variable cible. Dans le cadre d'un problème de classification, la variable cible doit être catégorielle, chacune des valeurs qu'elle peut prendre étant appelée une classe. Si elle est continue, elle peut être recodée en classes, ou d'autres types de méthodes doivent être utilisés. Par contre, la plupart des méthodes acceptent les deux types de variables explicatives. L'ensemble de départ est appelé ensemble d'entraînement.

Arbres

Avant d'introduire diverses méthodes qui permettent d'améliorer la précision d'arbres de classification, il importe de présenter le fonctionnement général de ces arbres. Ceux qui souhaiteraient se renseigner davantage sur les arbres de classification peuvent consulter le livre de Breiman *et al.* [1984].

Terminologie

Un arbre est un modèle qui divise successivement et de façon récursive l'ensemble d'entraînement en sous-ensembles selon les valeurs prises par les variables explicatives qui, à chaque étape, discriminent le mieux la variable cible selon un certain critère. Idéalement, on souhaite que chacun des sous-ensembles finaux contienne des cas appartenant à la même classe. Il résulte d'un arbre un ensemble de règles simples qui

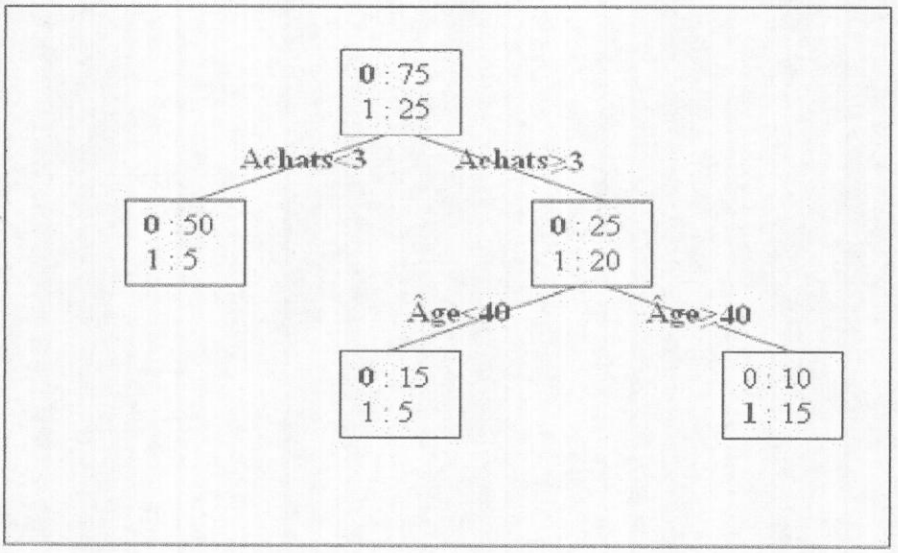
permettent de prévoir, de segmenter la population, ou d'identifier les variables explicatives discriminant le mieux la variable cible. La terminologie utilisée est la suivante : chaque arbre est formé d'un nœud racine, contenant l'ensemble des données; chaque sous-ensemble nouvellement formé devient enfant du nœud qui l'a formé, ce dernier étant un parent. Les nœuds terminaux sont appelés feuilles et sont donc des nœuds n'ayant aucun enfant.

Fonctionnement général

Un arbre est construit récursivement, en divisant chaque nœud en deux ou plusieurs enfants selon le nombre d'embranchements spécifiés. Une itération se fait de la façon suivante : pour chaque variable, les points de coupure qui permettent de mieux discriminer la variable cible selon un certain critère sont calculés. Toutes les paires "variable - point de coupure" sont comparées et celle qui discrimine le mieux la variable dépendante est choisie. Les nouveaux nœuds enfants sont alors formés, en séparant le sous-ensemble de données du nœud père selon la variable choisie. L'étape est répétée sur les nœuds nouvellement formés jusqu'à ce qu'un critère d'arrêt soit atteint. À chaque nœud est associée une classe, qui peut être celle dont la proportion est la plus élevée dans le nœud, ou qui dépend des coûts de mauvaise classification.

Voici un exemple. Soit 100 individus, auxquels a été envoyée une offre pour un livre. Le quart d'entre eux a commandé le livre. Considérons les variables "âge" et "nombre de livres achetés dans la compagnie". L'arbre de la figure 1 montre que les acheteurs type ont plus de 40 ans et ont acheté plus de trois livres. Les acheteurs sont identifiés par la classe 1, alors que les non-acheteurs sont identifiés par la classe 0.

Figure 1 : Exemple d'un arbre de classification



Cet arbre se résume en trois règles simples :

- Si le nombre de livres achetés est inférieur à trois, alors n'achètera pas;
- Si le nombre de livres achetés est supérieur ou égal à trois et que l'âge est inférieur à 40 ans, alors n'achètera pas;
- Si le nombre de livres achetés est supérieur ou égal à trois, et que l'âge est supérieur ou égal à 40, alors achètera.

Le détail de la construction de l'arbre est présenté dans la section méthodologie.

Avantages et inconvénients

L'avantage principal des arbres vient du fait qu'ils génèrent des règles simples et facilement interprétables qui permettent de mieux comprendre la structure des données. De plus, ils n'ont aucun présupposé à satisfaire, peuvent traiter facilement les valeurs manquantes et permettent une classification rapide. Finalement, il est possible d'y intégrer facilement les connaissances des experts en spécifiant par exemple des profits ou des pertes associés à chaque bonne ou mauvaise décision.

Par contre, les arbres présentent des inconvénients non négligeables. Ils sont peu performants si la variable cible est composée d'un grand nombre de classes. Mais surtout, ce sont des modèles instables, c'est-à-dire qu'une légère modification des données peut causer une grande différence dans l'arbre construit. Par contre, cet obstacle n'est pas insurmontable, et plusieurs méthodes ont été développées pour le résoudre; c'est ce dont il est question dans la prochaine section.

Méthodes ensemblistes

Définitions

Pour contrer l'instabilité des arbres et améliorer leur performance prévisionnelle, le concept des méthodes ensemblistes a été introduit. Une méthode ensembliste est, selon la définition de Dietterich [2000a], un ensemble de classificateurs individuels qui sont divers, mais précis, et dont les décisions sont combinées par moyenne ou par vote (la décision la plus populaire) pour donner une décision beaucoup plus précise. Selon Dietterich [2000b], un classificateur est précis si son taux d'erreur est plus faible qu'une classification aléatoire, et deux classificateurs sont divers s'ils font des erreurs différentes sur de nouvelles données. Notons que ces méthodes peuvent être appliquées à tout algorithmes instables, ceux pour lesquels un petit changement dans les données d'entraînement induit un grand changement dans le classificateur final.

Amélioration de la précision

Pourquoi un ensemble de classificateurs est-il plus précis qu'un classificateur seul? Selon Dietterich [2000b], il existe trois raisons. La première est statistique : puisque les classificateurs sont divers, ils font des erreurs à différents endroits, et prendre une moyenne ou un vote de leurs décisions individuelles diminue le risque de prendre la mauvaise décision. Ceci se produit davantage lorsque la proportion d'observations est trop faible par rapport au nombre de variables explicatives.

Pour comprendre les deux autres raisons, il faut souligner le fait que tout modèle peut être décrit par une fonction, mais que cette fonction n'est pas toujours facile à trouver. Dans les cas où la vraie fonction est trop difficile à trouver, des hypothèses approximant la fonction sont utilisées. Ces hypothèses sont à la base des deux autres raisons. La première d'entres elles est informationnelle : plusieurs algorithmes fonctionnent en optimisant une recherche locale, qui peut se retrouver coincée dans un optimum local, entre autres lors de la recherche du meilleur embranchement. Dans le cas où le nombre de données est important, il peut être difficile pour l'algorithme de trouver la meilleure hypothèse. Un ensemble dont les différents éléments sont construits en commençant la recherche locale à partir de plusieurs points différents fournira, au bout du compte, une meilleure approximation que tous les classificateurs pris séparément.

La dernière raison est représentationnelle : tel que mentionné plus haut, la fonction qui décrit exactement les données peut ne pas être représentée par les hypothèses fournies par les classificateurs. La somme pondérée des différentes hypothèses s'approche davantage de la vraie fonction que chacune des fonctions la formant.

Plusieurs auteurs (Breiman [1998,2000], Bauer & Kohavi [1999], Opitz & Maclin [1999]) expliquent la performance des méthodes ensemblistes par la décomposition de l'erreur. L'erreur de prédiction peut être divisée en trois parties : le taux d'erreur minimum, le biais et la variance. Le biais mesure à quel point la classification moyenne produite par l'algorithme sera éloignée de celle de la fonction cible. La variance mesure jusqu'à quel point plusieurs classificateurs construits par le même algorithme vont différer les uns des autres. Alors qu'il est impossible d'influencer le taux d'erreur minimum, et que le biais des arbres construits sans critère d'arrêt est déjà relativement faible, c'est la variance qui fait augmenter l'erreur de prédiction des arbres. S'il est possible de faire diminuer cette dernière, la précision sera alors meilleure. C'est justement ce à quoi s'appliquent les méthodes ensemblistes.

Les méthodes dont il sera question peuvent être divisées en deux principaux groupes : les méthodes qui manipulent l'ensemble d'entraînement et les méthodes qui y introduisent de l'aléatoire.

Manipulation de l'ensemble d'entraînement

La façon la plus commune de faire diminuer la variance est de perturber l'ensemble d'entraînement pour produite des ensembles alternatifs qui serviront à construire des classificateurs, et dont les résultats seront combinés afin de décider d'une classification finale. Les deux méthodes les plus populaires sont le bagging et le boosting.

Bagging

Le bagging a été introduit par Breiman [1996] et est un acronyme de *Bootstrap Aggregating*. Son fonctionnement est simple. Soit un ensemble d'entraînement formé de N observations. Un répliquât boostrap est formé en pigeant avec remise N observations de l'ensemble de départ. Ainsi, une observation peut apparaître plusieurs fois dans un même échantillon, alors qu'une autre peut ne pas être présente. Chaque répliquât contient en moyenne environ les deux tiers de l'ensemble d'origine. Un classificateur est bâti à partir de cet échantillon. La procédure est répétée plusieurs fois et un vote ou une moyenne est calculé à l'aide des classificateurs obtenus pour trouver la classe la plus populaire pour chaque observation. Le bagging ne fonctionne que sur les algorithmes instables. Il peut être vu comme une façon d'exploiter cette instabilité afin d'augmenter la précision de la classification.

Bauer & Kohavi [1999] ont fait une étude en profondeur du bagging et du boosting appliqué à l'algorithme de construction d'arbres C4.5. Ils ont trouvé que le bagging est uniformément meilleur qu'un simple classificateur pour tous les ensembles de données testés. Il n'a jamais fait augmenter l'erreur et la réduction de cette dernière est essentiellement due à la réduction de la variance, étant donné qu'il est possible, dans certains cas, de voir le biais augmenter sensiblement. Les auteurs ont constaté que les arbres formés par bagging étaient sensiblement plus gros que les arbres formés avec l'ensemble de données original, ce qu'ils ont expliqué par le fait que certaines observations se retrouvent plusieurs fois dans le même répliquât, renforçant ainsi une tendance qui aurait pu être considérée comme étant du bruit autrement. Breiman [1996], dans son introduction du bagging, observe une diminution du taux de mauvaise

classification allant de 20% jusqu'à 47%, tant sur des données générées que sur des données provenant de problèmes réels.

Bauer & Kohavi [1999] ont proposé trois améliorations au bagging, chacune augmentant la précision de la méthode précédente. La première amélioration est simplement de ne pas élaguer. La seconde consiste à ajouter des prévisions probabilistes : chaque classificateur retourne une distribution de probabilité pour les classes, ce qui donne, au bout du compte, un vote pondéré qui fait diminuer le biais et la variance. La dernière amélioration est appelée backfitting : l'arbre est construit avec le répliquât, puis l'algorithme fait passer dans l'arbre l'échantillon d'origine sans changer la structure de l'arbre. Les estimations des feuilles devraient alors être plus précises, étant basées sur plus de données. Chacune de ces méthodes a permis une légère amélioration du bagging.

Boosting

Le boosting a été introduit par Freund & Shapire en 1996, avec l'algorithme AdaBoost. Breiman a proposé en 1998 une généralisation appelée arcing, acronyme de *adaptively resample and combine*. Dans la mesure où les deux méthodes sont comparables, aucune différentiation ne sera faite entre les deux, et nous y référerons sous le nom de boosting.

Pour mieux comprendre le fonctionnement du boosting, comparons-le avec le bagging. Dans le bagging, la probabilité associée à chaque observation de l'ensemble est la même, soit 1/N. Avec le boosting, un poids différent est associé à chaque observation, au départ 1/N, et il est successivement modifié en fonction du nombre de fois où l'observation a été mal classifiée par les classificateurs construits précédemment. Plus

l'observation a été mal classifiée, plus le poids qui y est associé sera élevé, afin de forcer les classificateurs subséquents à se concentrer sur les données les plus difficiles à classifier. Ainsi, la construction du classificateur k dépend de la performance des $(k-1)^{èmes}$ classificateurs précédents. Le classificateur final est construit par un vote pondéré de tous les classificateurs, chacun étant pondéré par rapport à sa précision sur l'ensemble avec lequel il a été entraîné.

L'avantage principal du boosting est de permettre l'identification facile des valeurs qui se révèlent difficiles à classifier correctement, car elles seront celles ayant les poids les plus importants.

Il existe deux façons pour le boosting d'utiliser l'ensemble d'entraînement. La première est de sélectionner avec remise un échantillon de N observations, selon une distribution proportionnelle aux poids. La seconde est de modifier l'algorithme de construction d'arbres afin d'utiliser l'intégralité de l'ensemble d'entraînement à chaque construction, en pondérant chaque observation par son poids. Cette dernière approche à l'avantage d'utiliser chacune des observations, au moins en partie, pour construire chaque classificateur.

Bauer & Kohavi [1999] ont comparé la précision de ces deux méthodes à celle d'un classificateur seul. Encore une fois, les arbres construits sont plus grands que l'arbre construit avec l'algorithme de base. Par contre, le boosting n'est pas uniformément meilleur pour tous les ensembles testés en raison de son peu de résistance au bruit, comme nous le verrons plus loin. La diminution de l'erreur est due autant à la diminution de la variance qu'à la diminution du biais. Les auteurs ont finalement fait ressortir que le

rééchantillonnage est meilleur que l'utilisation directe de l'ensemble d'entraînement puisqu'il cause une plus grande diminution de la variance. De plus, ils ont montré que la précision de l'arcing était quelque peu meilleure que celle de AdaBoost.

Injection d'aléatoire

Il existe d'autres méthodes, proposées plus tard dans la littérature, qui impliquent une injection d'aléatoire à un niveau autre que dans tout l'ensemble d'entraînement comme c'est le cas pour le bagging. L'une d'entres elles est présentée.

La méthode dont il est question a été proposée par Dietterich [2000b] et se nomme randomisation. Plutôt que de miser sur l'instabilité des arbres, l'auteur a proposé d'introduire de l'aléatoire au niveau des décisions internes de l'arbre. Ainsi, plutôt que de choisir l'embranchement optimal selon le critère choisi, Dietterich a implanté une version d'un algorithme qui choisit aléatoirement, avec probabilités égales, parmi les 20 meilleurs embranchements possibles. La performance d'un ensemble de classificateurs ainsi construits est meilleure que la performance d'un seul arbre.

Comparaisons

Les comparaisons qui suivent ont toutes été réalisées sur des jeux de données, réels ou synthétiques, mais aucune comparaison théorique des méthodes n'a encore été effectuée. Le bagging et le boosting sont les deux méthodes qui ont été les plus comparées (Freund & Shapire [1996], Quinlan [1996], Breiman [1998], Bauer & Kohavi [1999], Opitz & Richard [1999]). Tous les auteurs s'entendent sur le fait que le boosting est plus performant que le bagging. Breiman [1998] soutient que ceci est dû au fait que le

boosting réduit davantage la variance. Par contre, la performance du bagging est constante, alors que les résultats varient beaucoup plus pour le boosting, qui peut faire diminuer radicalement l'erreur sur un ensemble, et beaucoup moins sur un suivant. Ceci est expliqué en grande partie par le peu de robustesse qu'a le boosting en présence de bruit aléatoire, ce dernier étant l'erreur présente dans la variable cible.

Ce fait a été mis en lumière par Dietterich [2000a], lors de la comparaison de la randomisation aux deux autres méthodes. Dietterich a d'abord remarqué que l'efficacité du bagging diminuait à mesure que grossissait l'ensemble d'entraînement, à moins que l'arbre formé ne grossisse proportionnellement au nombre de données, alors que l'efficacité de la randomisation n'était aucunement affectée par la taille de l'ensemble de départ. Mais la conclusion la plus intéressante de Dietterich est la suivante : lorsqu'il n'y a pas de bruit dans les données, la performance du boosting est supérieure à celle du bagging et de la randomisation. Mais lorsque 20% de bruit est ajouté aux données, l'efficacité du boosting se détériore assez pour que sa précision devienne alors inférieure à celle des deux autres méthodes. Le bagging et la randomisation sont, quant à eux, beaucoup moins affectés par le bruit et se comportent de façon similaire.

Ce prénomène s'explique par le fait que le boosting augmente itérativement le poids des observations les plus mal classifiées. Ainsi, le boosting risque de concentrer tout le poids sur le bruit, ce qui explique la diminution de la performance. La conclusion de Dietterich est que sans bruit, le boosting est de loin la meilleure méthode. Si du bruit est présent, la randomisation est préférable au bagging pour de grands ensembles.

Forêts aléatoires

Caractéristiques

Le terme forêt aléatoire a été introduit par Breiman [2001]. Ce dernier définit une forêt aléatoire comme étant un classificateur consistant en une collection de classificateurs à structure d'arbres, composés à partir d'ensembles d'entraînement indépendants et identiquement distribués, chaque arbre donnant un vote unique pour la classe la plus populaire d'une observation. Appliquées aux arbres, les méthodes introduites précédemment produisent toutes des forêts aléatoires, à l'exception du boosting. Ce dernier ne peut être considéré comme étant une forêt aléatoire étant donné que les échantillons successifs utilisés pour construire les arbres ne sont pas indépendants, puisqu'ils sont pondérés par des poids provenant des classifications antérieures. Breiman a tenté de tirer profit de toute l'expérience apprise de l'étude des méthodes introduites plus haut pour construire des forêts qui soient plus performantes encore. Il fallait, pour ce faire, que les nouvelles forêts aient certaines caractéristiques désirables : la précision doit être aussi bonne que celle du boosting, et quelquefois meilleure; elle doit être relativement robuste aux valeurs aberrantes et au bruit; elle doit être plus rapide que le bagging et le boosting; elle doit donner des estimés internes de l'erreur, de la force, de la corrélation et de l'importance des variables utiles; et finalement elle doit être simple et facilement implémentable. Breiman a expérimenté plusieurs forêts, dont les idées principales seront présentées.

Diverses forêts

La forêt la plus simple consiste à choisir aléatoirement, à chaque embranchement, un petit nombre de variables explicatives et de prendre le meilleur embranchement parmi ces variables. La forêt ainsi créée est appelée Forest-RI. Les résultats obtenus se comparent favorablement à ceux du boosting. Par contre, Forest-RI peut être beaucoup plus rapide que le bagging et le boosting. Pour l'un des ensembles de données, Forest-RI a été jusqu'à 40 fois plus rapide que le boosting, ce qui est considérable.

Breiman introduit aussi une forêt applicable aux ensembles de données caractérisés par un faible nombre de variables explicatives; ces ensembles sont désavantagés par une plus grande corrélation entre les arbres. L'approche consiste à définir plus de variables en formant des combinaisons linéaires d'un nombre aléatoire de variables explicatives. Cette procédure est appelée Forest-RC. Encore une fois, la comparaison avec le boosting est favorable, meilleure que dans le cas de Forest-RI.

Dietterich [2000a] ayant montré que la performance du boosting se détériore rapidement en présence de bruit aléatoire, Breiman a testé la performance de ses deux modèles en ajoutant 5% de bruit. Alors que le boosting se détériore de façon marquée, les résultats des deux forêts ne sont pas modifiés significativement.

Interprétation

Il est difficile de déduire un ensemble de règles à partir d'un ensemble de classificateurs, mais Breiman a développé une méthode qui, à défaut de rendre les forêts aussi facilement interprétables qu'un arbre seul, rend possible la compréhension de l'interaction des variables fournissant la précision. Sommairement, lorsqu'un

classificateur est construit, on compare le taux de mauvaise classification du modèle complet et celui du modèle sans la variable. En faisant une moyenne globale pour chacune des variables, on obtient un pourcentage de mauvaise classification pour chaque variable lorsque celle-ci n'est pas dans le modèle; ceci permet de déduire l'importance de chaque variable.

Notons, avant de terminer cette section, que les forêts aléatoires générant les plus petits taux d'erreurs sont celles pour lesquelles la corrélation entre les classificateurs est faible, et la force des classificateurs individuels est forte. Finalement, fait intéressant, Breiman a fait ses tests sur des ensembles de données utilisés pour tester plusieurs méthodes dans la littérature et, au moment des tests, il a obtenu le plus faible taux d'erreur jamais atteint pour trois des ensembles, à l'aide de Forest-RC.

Arbres et marketing

La présente section ne se veut pas une revue complète des applications des arbres au domaine du marketing, mais a pour but de donner un aperçu des multiples utilisations possibles des arbres de décision dans ce domaine, et de différents objectifs du marketing auxquels les arbres peuvent venir en aide. Il est possible de séparer le marketing en deux domaines d'application, d'abord la gestion de relation avec les clients (communément appelée CRM, *Customer Relation Management*), puis tout ce qui est issu d'Internet et du commerce électronique, le Web Mining et le marketing 1-to-1.

Gestion de relation avec les clients

Bounsaythip *et al.* [2001] décrivent en deux points l'objectif premier de la gestion de relation avec les clients : décrire les comportements des clients, et prévoir de nouveaux comportements. Les arbres peuvent être utilisés pour atteindre ces deux buts. Par leur nature, ils permettent de décrire le comportement des clients à l'aide de règles, qui elles-mêmes permettent de prévoir celui-ci. Mais ce n'est pas pas leurs seules utilités. Effectivement, la segmentation est d'une importance capitale en marketing, et, bien que des outils de data mining, tel le clustering, puissent être appliqués directement pour trouver des segments, des arbres peuvent aussi être utilisés pour ce faire. Liu *et al.* [2000] ont développé une méthode qui permet de faire du clustering à partir d'arbres de décision. C'est aussi le cas de Chou *et al.* [2000] qui présentent une autre méthode de clustering à l'aide d'arbres, de même qu'une méthode pour sélectionner des clients potentiels à partir de grands ensembles de données. Un arbre étant, du point de vue géométrique, un partitionnement de l'espace en régions rectangulaires, il peut aussi être utilisé pour identifier dans l'espace les régions denses, celles où des points se trouvent, et les régions vides. Finalement Kleingerb *et al.* [1998] présentent aussi une méthode pour des problèmes de segmentation.

Mais ces dernières ne sont pas les seules applications possibles des arbres au marketing. Bay & Passani [1999] présentent un algorithme permettant de comprendre les différences entre des ensembles contrastés. Piatetsky-Shapiro & Masand [1999] proposent une méthode pour mesurer le potentiel des campagnes de marketing direct bâties à l'aide de techniques de data mining, permettant d'estimer rapidement des

paramètres de coûts-bénéfices avant la modélisation. Bhattacharyya [2000] se penche sur les problèmes d'optimisation multi-critères, et présente une méthode qui permet d'obtenir un ensemble de modèles présentant des compromis différents respectant des objectifs multiples. Finalement, Ng *et al.* [1998] se penchent sur la rétention de clients et proposent une approche intégrant plusieurs techniques de data mining, incluant les arbres, et permettant de jauger la loyauté des clients et prévoir leur défection.

Web Mining et marketing 1-to-1

La popularité d'Internet et l'essor du commerce électronique ont mené au développement de nouveaux domaines, dont le Web mining et le marketing 1-to-1. Selon Bounsaythip [2001], le principal défi du Web mining consiste à réussir la transition d'une énorme quantité de données provenant de plusieurs serveurs et d'un nombre important de transactions vers un marketing profitable. Il s'agit aussi de découvrir de nouvelles habitudes et des différences de comportement entre consommateurs, en temps réel, pour permettre d'utiliser ces nouvelles connaissances immédiatement pour l'analyse des données. Selon l'auteure, les arbres sont surtout utilisés lors de vente-croisée.

Gallant *et al.* [2000] expliquent comment faire de la gestion de relation efficace avec les clients sur Internet. Il faut savoir maintenir un dialogue constant et cohérent avec plusieurs clients, en temps réel, et à plusieurs points de transaction, capturer l'information appropriée et significative à partir de tous les points de transaction, mesurer, suivre et gérer le portfolio des clients en terme de risque (*total lifetime value*), délivrer une information appropriée et significative à tous les points de transaction, déterminer l'impact et la valeur de différentes actions possibles avec chacun des clients à chaque

point de transaction, et finalement sélectionner l'action la plus appropriée en temps réel. Chacune de ces étapes implique un traitement de données auquel peuvent participer les arbres de classification.

Il existe bien sûr d'autres applications. Par exemple, Padmanabhan *et al.* [2001] propose une méthode d'analyse des données recueillies par un site Web pour personnaliser le site selon le visiteur, prévoir la probabilité d'achat à un site et construire des stratégies de marketing 1-to-1 efficaces.

Conclusion

Il existe plusieurs méthodes qui permettent d'améliorer la précision des arbres de décision. Parmi elles, le boosting était celle qui, jusqu'à l'arrivée des forêts aléatoires, permettait d'obtenir la plus importante baisse de taux de mauvaise classification sur des données sans bruit. Dans le cas de données où du bruit est présent, les autres méthodes sont équivalentes et peuvent être préférables au boosting. Le concept de forêt aléatoire est nouveau et prometteur, mais est encore récent et d'autres travaux sont nécessaires pour étudier ses propriétés plus en détail.

Les méthodes présentées permettent d'augmenter la précision des arbres, mais ont l'inconvénient de faire disparaître l'avantage principal des arbres, la facilité d'interprétation de la classification obtenue à l'aide de règles simples. Certains travaux récents présentent des méthodes qui permettent une certaine interprétation des résultats : cette interprétation n'est pas aussi directe que l'interprétation d'un arbre seul, mais les méthodes offrent une meilleure précision que ce dernier, et les prochaines années verront probablement l'émergence de méthodes donnant des résultats précis tout en étant interprétables.

Méthodologie

Introduction

L'une des méthodes de classification préférée des entreprises a longtemps été la régression logistique. Elle permet de classifier et d'ordonner des observations tout en donnant une description simple et instructive du lien entre les variables, ce qui n'est pas sans déplaire aux départements de marketing. La prolifération de grandes bases de données d'entreprise a favorisé la naissance et le développement de plusieurs nouvelles méthodes qui, théoriquement, devaient déclasser les méthodes classiques telle la régression logistique, mais qui, en pratique, n'ont pas toujours donné les résultats attendus. L'arbre de classification fait partie de ces méthodes : il permet de construire un modèle simple et facilement interprétable, mais son instabilité joue contre lui. Certaines entreprises ont donc préféré conserver la régression logistique comme méthode d'analyse de base, tout en se servant des arbres pour améliorer leurs modèles.

Dernièrement sont apparues de nouvelles méthodes plus sophistiquées à base d'arbres, relançant la popularité de ces derniers. L'une de ces méthodes pourrait-elle surpasser la régression logistique? C'est à cette question que la présente recherche tente de répondre. Elle vise à comparer les résultats de la régression logistique, ceux de la méthode utilisée actuellement par le Sélection du Reader's Digest, et ceux de nouvelles méthodes ensemblistes utilisant les arbres, sur la base des profits obtenus, afin de vérifier s'il existe une méthode ensembliste qui soit plus performante que la méthode utilisée actuellement par le Sélection du Reader's Digest.

Il sera d'abord question des données : leur composition sera abordée, suivie de leur préparation. Les méthodes de classification qui seront comparées seront ensuite décrites, puis il sera question de la façon dont seront construits les modèles, et finalement de la méthode qui permettra de comparer ces derniers.

Mise en situation

Le Sélection du Reader's Digest est une compagnie qui publie des magazines, des livres, des produits musicaux et vidéos. Les produits s'obtiennent par la poste. Afin de faire connaître ses produits, la compagnie envoie à des segments de sa clientèle des offres postales. Pour bien cibler les clients ayant la plus grande probabilité d'acheter les produits offerts, la compagnie sélectionne aléatoirement, pour chaque produit, un premier groupe d'individus de sa banque de données et enregistre la réponse à l'offre. Puis la même opération est refaite un certain nombre de fois et, à l'aide des réponses obtenues, il est possible de bâtir une équation prédisant l'achat du produit. C'est à la méthode qui permet de construire cette équation que ce mémoire s'intéresse.

Données

Il est important à ce stade de souligner une particularité de cette recherche, puisque le contenu de cette section et de la suivante en est fortement influencé : dans la mesure où il s'agit d'une recherche qui est effectuée en collaboration avec une compagnie privée, ses données et ses méthodes sont soumises à certaines règles de confidentialité. Il ne sera donc pas possible de présenter en détail toutes les étapes dont sera composée cette recherche, mais dans les cas où cette restriction s'imposera, l'essentiel de l'étape sera indiqué.

Composition des données

Afin d'amasser des données pour construire le modèle, la compagnie fait une étude préalable en offrant le produit à un échantillon de ses clients actuels. Un individu

est donc composé de deux types de variables : une variable cible, indiquant si le client a répondu positivement à l'offre et s'il a payé, et des variables explicatives, qui sont composées d'une part de variables démographiques, dont certaines sont tirées des données de Statistiques Canada, et d'autre part de variables décrivant l'historique du client dans la compagnie. On y retrouve entre autres les transactions déjà effectuées par le client, le nombre de produits achetés, livres, disques ou vidéos, le nombre de jours depuis le dernier achat dans chacune des catégories et depuis le premier achat. Notons que les données fournies pour cette recherche étaient déjà préparées. Le nettoyage avait été fait, et les critères sur lesquels il est basé sont confidentiels.

Structure des données

Il importe de prendre connaissance de la structure particulière des données, plus particulièrement celle de la variable cible. Elle est composée de deux classes, les acheteurs et les non-acheteurs. Cette dernière classe prédomine, constituant entre 95% et plus de 99% des observations, selon le jeu de données considéré. Le but ne sera donc pas d'obtenir des modèles où le taux de mauvaise classification est le plus bas possible : effectivement, il serait possible de classer les observations comme appartenant toutes à la classe des non-acheteurs, et ainsi obtenir des taux de mauvaise classification de 5% ou moins, soit le nombre d'acheteurs dans l'échantillon. Le rôle des modèles sera plutôt de maximiser les profits, donc de classifier autant que possible les acheteurs en acheteurs.

Pour ce qui est des variables explicatives, ce sont soit des variables binaires, soit des variables continues dont les valeurs sont réelles et positives. Certaines des variables continues peuvent prendre jusqu'à 50 000 valeurs différentes, ce qui peut influer sur la

vitesse d'exécution des algorithmes, étant donné que toutes les valeurs possibles d'une variable sont considérées comme étant candidates pour une division de l'ensemble de données.

Les ensembles de données qui seront utilisés sont présentés dans le tableau 1. Les sept premiers ensembles ont été utilisés pour générer des modèles. Les deux derniers ensembles sont appelés ensembles de validation, mais dans un sens autre que celui que l'on accorde habituellement à ce terme. Ils servent à vérifier si les résultats des modèles sont les mêmes sur des données prélevées plus tard dans le temps. Ainsi, les modèles générés avec deux des premiers ensembles sont utilisés pour classifier ces nouvelles données, et les résultats sont comparés. Les données présentées dans le tableau 1 seront utiles pour construire les modèles. Notons que les coûts de mauvaise classification ne peuvent être présentés étant donné la confidentialité des données. Nous nous contenterons de dire que l'écart entre le coût de classifier un acheteur en non-acheteur et celui de classifier un non-acheteur en un acheteur est élevé, pouvant aller jusqu'à un ratio de 50 pour 1.

Tableau 1 : Description des ensembles utilisés

Ensemble	Nombre d'observations	Nombre de variables explicatives	Pourcentage de personnes ayant payé (var.cible =1)
Ens.1	20275	191	5.05%
Ens.2	20164	74	1.69%
Ens.3	12543	166	5.19%
Ens.4	12538	49	1.80%
Ens.5	16225	223	5.76%
Ens.6	20164	37	0.93%
Ens.7	20009	103	0.61%
Valid.1	2059	191	5.05%
Valid.2	2514	74	1.79%

Méthodes de classification et implémentation

Caractéristiques recherchées

Le but premier des modèles est de classifier les individus selon qu'ils aient acheté ou non. Mais là n'est pas leur seule fonction. Ils doivent permettre d'associer à chaque individu une probabilité d'achat. La raison vient de la façon de comparer les résultats : la méthode de comparaison utilisée nécessite un ordonnancement des données selon la probabilité d'achat. Il faudra donc, si la méthode ne le permet pas directement, trouver un moyen d'assigner à chaque individu sa probabilité d'achat.

Régression logistique

La régression logistique est une méthode très utilisée lorsque vient le temps de bâtir un modèle avec une variable cible binaire. Elle permet de modéliser le comportement de la variable cible en fonction des différentes variables explicatives, et elle permet de trouver la probabilité que la variable cible soit égale à 1.

La meilleure façon de comprendre la régression logistique est de commencer le raisonnement à partir de la régression linéaire. Pour faciliter la présentation, nous ne travaillerons qu'avec une variable explicative, soit x, mais la méthode se généralise de la même façon que la régression linéaire avec plusieurs variables explicatives. Rappelons que, dans la régression linéaire, le modèle de base est le suivant : $E(y \mid x) = \beta_0 + \beta_1 x$, où $E(y \mid x)$ est la valeur moyenne de la variable cible étant donné la valeur de la variable explicative, ou, en d'autres mots, l'espérance conditionnelle de y sachant x. Or, comme

y est binaire, $E(y \mid x) = P(y = 1 \mid x)$. On pourrait donc logiquement en tirer $P(y = 1 \mid x) = \beta_0 + \beta_1 x$. Mais cette façon de voir les choses ne serait pas appropriée : effectivement, le côté gauche de l'équation est une probabilité, qui doit se situer dans l'intervalle $[0,1]$ alors que le côté droit peut quant à lui prendre ses valeurs dans l'intervalle $[-\infty, \infty]$. Il faut donc modifier l'un ou l'autre côté de l'équation afin de faire concorder les réponses possibles. Modifions le côté droit. Il faut lui appliquer une fonction qui ne prendra ses valeurs que dans l'intervalle $[0,1]$. L'une des possibilités, d'où la régression logistique tire son nom, consiste à prendre la fonction de répartition de la loi logistique, ce qui donne la formule suivante : $P(y = 1 \mid x) = \frac{e^{\beta_0 + \beta_1 x}}{1 + e^{\beta_0 + \beta_1 x}}$. Il est aussi possible de faire la transformation inverse, où pour simplifier nous renommerons $P(y = 1 \mid x)$ comme étant $\pi(x)$. Ainsi, le modèle s'écrit $\ln\left(\frac{\pi(x)}{1 - \pi(x)}\right) = \beta_0 + \beta_1 x$, le terme de gauche est nommé logit de $\pi(x)$, et la transformation est appelée transformation logit. La méthode habituellement utilisée pour estimer les coefficients du modèle est la méthode du maximum de vraisemblance. Une fois le modèle ajusté aux données, il est possible de classifier de nouvelles données selon la règle suivante : pour un x et un point de coupure p donnés, si $\hat{\pi}(x) > p$, alors $\hat{y} = 1$, sinon, $\hat{y} = 0$. Étant donné que le but de l'étude est de construire un modèle de prédiction et non pas un modèle explicatif, la revue de la régression logistique s'arrêtera ici. Il est possible de trouver de plus amples informations sur la régression logistique dans le livre de Hosmer et Lemeshow [1989].

Méthode de l'entreprise

Il n'est pas possible ici de décrire la méthode exacte utilisée par l'entreprise pour classifier ses données. Nous nous contenterons de dire que nous avons tenté d'améliorer cette méthode, et les résultats seront présentés, dans les graphiques, sous le nom Amélioré.

Arbres de classification

Voyons maintenant plus en détail les particularités des arbres de classification. Rappelons que le principe de base de l'arbre consiste à diviser successivement et de façon récursive l'ensemble d'entraînement à l'aide des variables explicatives. On souhaite que les sous-ensembles ainsi créés contiennent le plus de cas appartenant à la même classe de la variable cible. Dans la mesure où la variable cible du problème est binaire, nous présenterons la théorie pour ce cas particulier, où 0 identifie un non-acheteur alors que 1 identifie un acheteur.

Algorithme

Plusieurs algorithmes permettant de construire des arbres ont été développés, trois des plus populaires étant les suivants :

- CART (Classification And Regression Trees), introduit par Breiman *et al.* [1984],
- CHAID (Chi-Squared Automatic Interaction Detection), Kass [1980].
- C5.0 et ses prédécesseurs ID3 et C4.5, proposé par Quinlan [1993].

L'algorithme de base étant sensiblement le même d'un cas à l'autre, il sera présenté

globalement, et les différences seront indiquées au moment opportun. Notons que l'algorithme utilisé pour l'implantation des méthodes ensemblistes est CART.

Embranchement

L'embranchement permet de diviser l'ensemble d'entraînement en sous-ensembles de plus en plus petits et constitue ainsi l'opération de base de la construction des arbres. Chaque nœud est divisé en un ou plusieurs enfants, et le nombre de ces derniers dépend de l'algorithme utilisé. CART génère des embranchements binaires; un nœud sera ainsi divisé en deux nouveaux nœuds à chaque itération. CHAID et C5.0 n'ont pas de restriction quant au nombre de branches. Les embranchements binaires ont l'avantage de ne pas fragmenter les données trop rapidement, alors que les embranchements multiples permettent de générer des arbres ayant moins de niveaux de division.

Dans la mesure où tous les arbres utilisés dans le cadre de ce mémoire seront générés à l'aide de CART, la théorie sera présentée en conséquence. Il est par contre possible de généraliser facilement à un plus grand nombre de branches. Étant donné que les variables utilisées sont ordonnées, les embranchements auront la forme "variable $< c$" et "variable $\geq c$", où c est le point de coupure. Si les variables étaient catégorielles, alors la forme des embranchements serait "variable $\in S$" et "variable $\notin S$", où S est un sous-ensemble de l'ensemble des valeurs possibles de la variable.

Il faut maintenant se pencher sur le point qui est au cœur du choix de l'embranchement, le critère de séparation. Il existe quatre critères de séparation principaux, et l'utilisation de chacun d'entre eux peut potentiellement produire des arbres

différents à partir d'un même jeu de données. Les algorithmes présentés plus haut utilisent un critère en particulier, mais la plupart des implémentations de ces algorithmes permettent de choisir le critère voulu. Cette section vise à définir exactement ce que sont ces différents critères.

Le premier critère est utilisé par l'algorithme CHAID. Il s'agit, comme son nom l'indique, d'un critère basé sur le test du khi-deux, qui permet de déterminer le degré de dépendance entre deux variables. Le critère permet donc de trouver la variable explicative qui discrimine le mieux la variable cible, donc celle dont la relation avec la variable cible est la plus forte pour les données de la feuille à séparer. En utilisant comme critère le degré de signification d'un test statistique, CHAID évalue, pour chaque variable explicative, toutes les valeurs de la variable, et regroupe toutes les valeurs qui sont jugées comme étant statistiquement homogènes par rapport à la variable cible. L'embranchement qui est sélectionné est celui qui maximise l'homogénéité des nœuds résultants de la séparation.

Si la variable évaluée est continue, le test statistique utilisé sera le test de Fisher, un test d'égalité de moyennes ; si la variable est catégorielle, ce sera le test du khi-deux. Les embranchements ne sont pas nécessairement binaires.

Le second algorithme dont il est question est CART. Il utilise de façon générique un indice de diversité pour déterminer la variable explicative la plus discriminante. L'indice de diversité est un indicateur de la distribution de la variable cible dans les noeuds. S'il est élevé, il indique une distribution uniforme des classes, tandis que s'il est bas, il indique la prédominance d'une seule classe. Le but est de trouver la variable

explicative qui provoque la plus grande baisse de diversité, ou dit autrement, le plus grand gain informationnel.

Pour faciliter la formulation des indices et de la théorie qui suit, nous devons prendre un moment pour introduire quelques notations. Soit t, un nœud donné. Définissons $N_0(t)$ et $N_1(t)$ comme étant le nombre d'observations des classes 0 et 1 dans le nœud t. Définissons aussi $N(t) = N_0(t) + N_1(t)$, le nombre d'observations total dans le nœud t, et N, le nombre total d'observations de l'ensemble de données. Soit

$$p(t) = \frac{N(t)}{N} \tag{1}$$

la proportion d'observations se trouvant dans le nœud t, et soit

$$p(i \mid t) = \frac{N_i(t)}{N}, \; i = 0,1 \tag{2}$$

la proportion d'observation de la classe i dans le nœud t. Pour une division du nœud t en deux nouvelles feuilles notées t_G et t_D, notons par

$$p_G = \frac{p(t_G)}{p(t)} \text{ et } p_D = \frac{p(t_D)}{p(t)} \tag{3}$$

la proportion des observations du nœud t qui se retrouvent respectivement dans les nœuds t_G et t_D.

Nous pouvons alors définir les deux principaux indices de diversité :

- Indice de Gini : $1 - (p^2(0 \mid t) + p^2(1 \mid t))$
- Entropie : $-(p(0 \mid t) \log_2 p(0 \mid t) + p(1 \mid t) \log_2 p(1 \mid t))$

Les indices de diversité s'interprètent comme suit : plus la valeur de l'indice est faible, plus le nœud est pur en ce sens qu'il contient une forte majorité d'observations appartenant à l'une des deux classes. Lors du choix d'un embranchement, on souhaite maximiser le gain informationnel, qui se définit comme suit :

$$I(t) - (p_G I(t_G) + p_D I(t_D)),$$

où $I(t)$ est la valeur d'un indice de diversité pour le nœud t. La version de CART qui a été implantée pour générer les arbres utilise l'indice de Gini. L'algorithme C4.5 utilise quant à lui l'entropie comme indice de diversité, de même que ses successeurs C5.0 et ID3.

Les deux indices qui ont été introduits sont surtout utilisés pour la séparation des nœuds, alors que le taux de mauvaise classification, qui se définit comme la proportion d'observations mal classifiées, est surtout utilisé pour l'élagage, qui consiste à couper des feuilles de l'arbre pour en réduire la grosseur. Par contre, le choix du critère de séparation ne semble pas affecter la classification finale des observations ni l'exactitude de l'arbre. Le critère d'arrêt et l'élagage ont plus d'influence sur ces questions.

Critère d'arrêt

Il existe plusieurs critères pour décider si un nœud doit encore être séparé. Les premiers sont des critères naturels : si toutes les observations du nœud appartiennent à la même classe, ou s'il ne reste plus de variables permettant de diviser le noeud, celui-ci devient automatiquement une feuille. Il est aussi possible de décréter l'arrêt si le nombre d'observations dans le nœud est plus petit qu'un nombre fixé, ou si la proportion

d'observations dans le nœud est inférieure à un seuil fixé à l'avance. L'arrêt peut aussi s'effectuer si l'indice de diversité est inférieur à une valeur fixe. Mais quand doit-on arrêter la croissance de l'arbre? Si la croissance est arrêtée trop tôt, toute l'information disponible dans les données n'est pas exploitée, et le classificateur est biaisé. Mais si l'arbre est trop gros, le risque de sur-ajustement apparaît.

Dans le cadre de ce mémoire, nous imposerons la contrainte suivante pour éviter le risque de sur-ajustement : chaque feuille résultante doit contenir 5% au moins du nombre total d'observations de l'ensemble d'entraînement.

Assignation d'une classe

Une fois l'arbre construit, il faut assigner à chaque feuille, et si désiré à chaque nœud de l'arbre, une classe. Pour ce faire, il suffit de choisir la classe majoritaire dans le nœud, et donc de suivre les deux règles suivantes :

- Si $p(0 \mid t) < p(1 \mid t)$ alors le nœud est classifié comme étant 1.
- Si $p(0 \mid t) > p(1 \mid t)$ alors le nœud est classifié comme étant 0.
- Lorsqu'il y a égalité, le choix est arbitraire.

Valeurs manquantes

Il est possible de traiter les valeurs manquantes de plusieurs façons. La première consiste à affecter une valeur, le mode ou la moyenne des autres valeurs de la même variable, en remplacement de la valeur manquante. Mais les caractéristiques mêmes des arbres permettent de faire deux traitements plus judicieux des valeurs manquantes.

Le premier traitement consiste à considérer l'information sous-tendant le fait que certaines valeurs sont manquantes, et de considérer celles-ci comme une nouvelle caractéristique. Ainsi, si pour une variable plusieurs valeurs sont absentes, le point de coupure pourra être défini comme étant la présence ou l'absence de la valeur. L'absence de valeur est donc considérée comme une valeur en soi. C'est ce traitement des valeurs manquantes qui est implanté dans le cadre de ce mémoire.

La seconde méthode utilise la corrélation entre les variables pour définir des embranchements substituts à chaque nœud pour le cas où la valeur de la variable associée au nœud est manquante. Ainsi, lors de la construction de l'arbre, des embranchements substituts sont définis pour chacun des nœuds, et classés selon la meilleure approximation de la décision primaire. Lorsqu'une observation est classée par l'arbre, la décision primaire à un nœud est utilisée si la valeur n'est pas manquante; si la valeur est manquante, les décisions substituts sont utilisées les unes après les autres jusqu'à ce qu'il y en ait une formée d'une variable dont la valeur n'est pas manquante pour l'observation, permettant ainsi de diriger l'observation vers une autre feuille où la classification reprend son cours.

L'algorithme C5.0 utilise une méthode différente pour traiter les valeurs manquantes. Plutôt que de calculer des décisions substituts, il classe les valeurs manquantes selon une probabilité de décision à chaque nœud.

Pertes, coûts et probabilités a priori

Le dernier point dont il importe de parler est relatif aux caractéristiques du problème de classification. L'un des avantages non négligeables des arbres est qu'il est

possible d'inclure des connaissances d'experts dans la construction du modèle. Ces connaissances sont exprimées sous forme de pertes ou de profits, de coûts et de probabilités a priori.

Les matrices de pertes, de coûts et de profits sont utilisées dans le cas où il peut être nocif de faire une mauvaise classification, l'exemple classique étant de classifier quelqu'un comme n'étant pas à risque de maladies cardiaques alors qu'il l'est. Les matrices de coûts sont utiles lorsqu'il y a des coûts et des gains reliés aux décisions bien ou mal prises. Par exemple, supposons qu'envoyer une offre à un acheteur génère 50$ et coûte 1$, le prix d'un timbre. Ainsi, envoyer l'offre à un non-acheteur ne fera perdre à la compagnie que le prix du timbre, alors que ne pas envoyer l'offre à un acheteur fera perdre 49$, le profit moins le coût du timbre.

Les probabilités a priori sont généralement utilisées lorsque les proportions de l'ensemble d'entraînement ne sont pas les mêmes que celles de la population et qu'il importe de construire l'arbre en fonction des secondes et non pas des premières. Il est possible de modifier facilement l'indice de Gini pour inclure ces informations. De plus, il est possible d'utiliser les probabilités a priori pour incorporer l'information sur les coûts de mauvaise classification, tel qu'il sera fait dans la présente étude.

Soient π_0 et π_1 $(\pi_0+\pi_1=1, \pi_0 \geq 0, \pi_1 \geq 0)$, des probabilités a priori pour les classes 0 et 1. Afin d'utiliser ces probabilités dans la construction de l'arbre, tant au niveau de l'embranchement que de l'assignation d'une classe, il suffit de remplacer les quantités $p(t)$ et $p(i|t)$ dans les équations (1) à (3) par $p(t)=\frac{\pi_0 N_0(t)}{N_0}+\frac{\pi_1 N_1(t)}{N_1}$ et

$p(i \mid t) = \frac{\pi_i N_i(t)/N_i}{p(t)}$, où N_0 et N_1 $(N_0 + N_1 = N)$ sont les nombres de 0 et de 1 de la variable cible dans l'ensemble d'entraînement. Notons que (1) et (2) correspondent au choix habituel $\pi_0 = N_0/N$ et $\pi_1 = N_1/N$, qui spécifie que les probabilités a priori sont égales aux proportions de 0 et 1 dans l'ensemble d'entraînement.

Les coûts de mauvaise classification utilisés sont la perte des profits générés par une vente dans le cas où un acheteur est classifié comme un non-acheteur et le coût de l'envoi de l'offre dans le cas contraire. Comme il a été mentionné plus haut, il est possible de redéfinir les probabilités a priori afin d'inclure ces coûts. Définissons pour ce faire $C(1 \mid 0)$ et $C(0 \mid 1)$, comme étant respectivement le coût de classifier une observation comme étant 1 étant donné qu'elle est 0 en réalité et le coût de la classifier en 0 alors qu'elle est en fait 1. Pour tenir compte de ces coûts, il suffit d'utiliser de nouvelles probabilités a priori:

$$\pi'_0 = \frac{C(1 \mid 0)N_0/N}{C(1 \mid 0)N_0/N + C(0 \mid 1)N_1/N} \text{ et } \pi'_1 = \frac{C(0 \mid 1)N_1/N}{C(1 \mid 0)N_0/N + C(0 \mid 1)N_1/N}.$$

L'ajout de coûts de mauvaise classification influera aussi sur la règle d'assignation d'une classe à un noeud, qui deviendra ceci : si $C(1 \mid 0)p(0 \mid t) < C(0 \mid 1)p(1 \mid t)$, alors la feuille sera classifiée comme étant 1, et 0 sinon.

Méthodes ensemblistes

Au départ, quatre méthodes seront utilisées pour générer des ensembles d'arbres. Il a déjà été question de ces méthodes dans la revue de littérature. Il s'agit du bagging, du boosting, de la randomisation de Dietterich et des forêts aléatoires. Puis une autre

méthode, qui ne fait pas partie des méthodes présentes dans la littérature mais dont nous pensons qu'elle peut donner des résultats intéressants, sera présentée. Les méthodes ont seulement été présentées brièvement dans la revue de littérature, aussi les décrirons-nous plus en profondeur avant d'aborder quelques détails techniques, portant, entre autres, sur l'implémentation des modèles. Notons que toutes les méthodes ont été implantées en langage C++, et ont été entièrement codées par l'auteure.

Tel que nous l'avons précisé au début de la section, les modèles doivent donner comme résultat une probabilité d'achat. Or toutes les méthodes, telles que décrites dans la littérature, donnent un vote majoritaire ou une moyenne. Un ajout sera donc apporté à chaque algorithme, de façon à ce que le résultat final soit la proportion des fois où l'observation a été classifiée comme acheteur, donnant ainsi sa probabilité d'achat.

Pour faciliter la compréhension de ces méthodes, définissons d'abord $\varphi(x, L)$, qui dénote un arbre construit à partir de l'ensemble d'entraînement $L = \{(y_i, x_i), i = 1, ..., N\}$ où les y_i sont parmi les 2 classes de la variable cible et les vecteurs x_i sont les différentes valeurs prises par les variables explicatives de l'observation i. Cet arbre permet de prédire l'achat (y) étant donné les valeurs du prédicteur (x).

Bagging

L'algorithme implanté est celui présenté par Breiman [1996]. Soit une suite d'ensembles d'entraînement $\{L_1, ..., L_K\}$, chacun comptant N observations. Nous pourrions alors construire K arbres $\varphi(x, L_1), \ldots, \varphi(x, L_K)$. Il serait alors possible d'obtenir

une meilleure prédiction en utilisant la moyenne des résultats de $\{\varphi(x, L_k)\}, k = 1, ..., K$.

Par contre, il n'existe qu'un seul ensemble L pour générer les arbres. Le principe du bagging consiste à appliquer la méthode décrite précédemment avec K échantillons bootstrap $L_1^{(B)}, ..., L_K^{(B)}$. Un échantillon bootstrap $L^{(B)}$ est formé en pigeant aléatoirement mais avec remise N observations de L. Ainsi, alors que certaines observations de l'ensemble de départ ne se retrouveront pas dans $L^{(B)}$, d'autres s'y retrouveront plus d'une fois, et en moyenne, le nouvel ensemble sera formé de près des deux tiers des données appartenant à l'ensemble d'entraînement de départ.

Cette duplication de certaines observations permet de mettre l'accent sur certaines tendances qui autrement auraient été ignorées. Ainsi, alors qu'un arbre construit avec l'ensemble des données d'entraînement classifiera peu ou pas de feuilles comme étant non-acheteurs, un arbre construit avec un échantillon bootstrap pourra davantage isoler les non-acheteurs, étant donné que l'accent sera mis sur des relations qui n'étaient pas visibles auparavant. Chaque arbre pourra faire ressortir des relations différentes, et l'ensemble donnera un meilleur aperçu des relations entre la variables cibles et explicatives qu'un arbre seul.

Boosting

La version du boosting qui est implantée est une variante de celle proposée par Freund & Shapire [1996], appelée AdaboostM1. Le boosting fonctionne sensiblement de la même façon que le bagging : le but est de générer plusieurs classificateurs $\varphi(x, L_k)$ qui permettront de donner une moyenne, un vote ou une proportion de bonne classification.

La différence entre les deux algorithmes se situe au niveau des ensembles de données présentés au classificateur : ce ne sont plus des répliquâts de l'ensemble d'origine qui sont utilisés, mais l'ensemble d'origine pondéré selon les résultats des classificateurs précédents.

Plus spécifiquement, à la première itération, un arbre $\varphi_1(x, L)$ est construit à l'aide de L, puis les valeurs prédites de y sont calculées pour chaque x_i. Normalement, une fraction des observations devrait être mal classifiée. Il faut donc mettre un poids à ces observations pour indiquer au prochain arbre de tenter de mieux les classifier. L'algorithme AdaBoostM1 se présente comme suit.

1- Initialisation des poids des observations, $w_i = 1/N, i = 1,....N$.

2- Pour $k = 1,...,K$

   a. À l'aide des poids w_i, ajuster un classificateur $\varphi_k(x, L_k)$ à l'ensemble d'entraînement.

   b. Calculer l'erreur de classification, $err_k = \dfrac{\sum_{i=1}^{N} w_i b_i}{\sum_{i=1}^{N} w_i}$ où b_i est égal à 1 si l'observation i a été mal classifiée par $\varphi_k(x, L_k)$ et 0 sinon.

   c. Calculer $\alpha_k = \log\left(\dfrac{1 - err_k}{err_k}\right)$.

   d. Mettre à jour les poids, $w_i = w_i \exp(\alpha_k b_i)$.

3- Calculer le résultat : pour une valeur de x, la probabilité que $y=1$ est estimée par la moyenne pondérée $\sum_{k=1}^{K}\alpha_k\varphi_k(x,L_k)\Big/\sum_{k=1}^{K}\alpha_k$.

Ainsi, si x_i est mal classifiée par $\varphi_k(x,L_k)$, son poids est multiplié par une constante $\alpha_k \in [0,1]$; autrement son poids reste inchangé. Les exemples facilement classifiés ont un poids très petit, et l'accent est mis sur les observations difficiles à classifier. Dans la revue de littérature, nous avons souligné que les résultats obtenus à l'aide du boosting étaient meilleurs lorsque les ensembles utilisés pour construire le classificateur étaient des échantillons bootstrap et non l'ensemble d'entraînement lui-même. C'est donc une variante de l'algorithme présenté plus tôt qui est implémentée : il utilise, à l'instar du bagging, des échantillons bootstrap pour construire les arbres. Par contre, ces échantillons sont générés à l'aide des probabilités induites par les poids et non pas des probabilités uniformes et constantes comme le fait le bagging.

Il reste un point à traiter. Cette version du boosting ne peut être utilisée si les probabilités a priori sont réajustées à l'aide des coûts de mauvaise classification, du moins avec les données utilisées ici. La raison est la suivante : comme nous l'avons vu, lorsque les probabilités a priori sont réajustées, les prévisions finales seront pour la plupart l'achat, et seul un petit pourcentage des observations seront classifiées comme étant non-acheteurs. Ainsi, tous les non-acheteurs classifiés comme acheteurs entreront dans le calcul du dénominateur de l'erreur calculée en 2.b). Étant donné la grande proportion de non-acheteurs, l'erreur sera très certainement supérieure à 0.5. Le α ainsi obtenu en 2.c) sera ainsi négatif, et les observations qui auront été mal classifiées seront

pondérées par un facteur inférieur à 1 en 2.d). Les poids de ces observations seront donc diminués plutôt qu'augmentés.

De plus, comme nous l'avons vu, la principale caractéristique du boosting est de mettre l'accent sur les observations mal classifiées. Si nous présentons à l'algorithme l'ensemble avec ses probabilités initiales, les acheteurs seront dans les premières itérations classifiés comme étant non-acheteurs, puis le poids de ces observations sera augmenté et ils seront graduellement de mieux en mieux classifiés. De par la nature de l'algorithme, il n'est donc pas nécessaire de redéfinir les probabilités a priori afin de mettre l'accent sur les acheteurs.

Randomisation

Il y a peu de choses à ajouter par rapport à cette méthode. Nous avons mentionné plus haut que, lors de la création d'un arbre, le choix de la variable et du point de coupure qui mènent à la séparation d'un noeud se fait en fonction du gain informationnel. Or, il a aussi été question de l'instabilité des arbres : un petit changement dans les données peut mener à un classificateur totalement différent. Ainsi, malgré la logique du choix de la variable, prendre la meilleure ou la seconde meilleure n'a pas beaucoup d'influence sur la performance. De là vient l'idée de créer un ensemble d'arbres qui différeraient quant aux variables choisies pour séparer les feuilles, donc quant aux décisions internes.

Il reste à déterminer le nombre d'embranchements parmi lesquels choisir aléatoirement. Dietterich rapporte avoir fait ses expériences avec 20 embranchements. Les tests préliminaires effectués sur les ensembles de données n'ont pas été concluants avec ce nombre. Effectivement, comme nous l'avons mentionné plus haut, la plupart des

feuilles classifient des acheteurs, et si l'ensemble d'entraînement est présenté à l'arbre sans changement, seule une feuille, voire deux ou trois, classifient des non-acheteurs. Il peut exister plusieurs combinaisons d'embranchements qui mèneront à ces feuilles, mais il est aussi possible que seul un petit nombre d'embranchements classifie des feuilles comme étant non-acheteurs. Dans ce dernier cas, un choix parmi un trop grand nombre d'embranchements peut réduire de beaucoup la probabilité de trouver l'un des embranchements permettant d'isoler des non-acheteurs. Bien sûr, générer des arbres divers en jouant sur les embranchements est la base de la méthode. Cependant, dans le cas de nos données, permettre un choix parmi un trop grand nombre d'embranchements revient à condamner plusieurs des arbres à n'avoir que des feuilles acheteurs. À l'opposé, un nombre trop petit réduit la diversité des arbres.

Le nombre possible d'embranchements varie en fonction de deux choses : le nombre de variables et le nombre de valeurs que peuvent prendre chaque variable. Le premier est connu, alors que le second ne l'est pas. Nous avons donc décidé de déterminer le nombre d'embranchements comme étant un pourcentage du nombre de variables, selon la règle suivante : si le nombre de variables est supérieur à 100, alors le nombre d'embranchements choisi est de 5% du nombre de variables; dans le cas contraire, le nombre d'embranchement est de 10% du nombre de variables.

Forêts aléatoires

De même que pour la randomisation, peu de choses ont besoin d'être spécifiées pour les forêts aléatoires. Une simple modification est effectuée sur l'algorithme de génération d'arbres : un nombre de variables spécifié au départ, et à chaque

embranchement, les variables sont choisies aléatoirement. Par la suite, le meilleur embranchement est choisi. Comme pour le boosting, l'ensemble d'entraînement est composé d'un échantillon bootstrap différent pour chaque arbre construit.

Il faut ici aussi déterminer le nombre de variables candidates à l'embranchement. Breiman suggère deux mesures : soit la racine carrée du nombre de variables, soit le log de ce nombre auquel on additionne un. Encore une fois, les tests préliminaires ont permis de conclure que ces mesures n'étaient pas adaptées à la structure des données. Pour les ensembles ayant un petit nombre de variables, les résultats étaient intéressants, ce qui était loin d'être le cas pour les ensembles ayant un grand nombre de variables explicatives. Le problème qui se pose ici est l'inverse de celui abordé pour la randomisation : un nombre trop petit de variables réduit de beaucoup la probabilité de trouver une combinaison d'embranchements qui mènent à une feuille classifiée non-acheteurs, et un nombre trop grand réduit la diversité. Deux séries de tests ont été effectuées, respectivement avec un choix parmi 10% et 20% du nombre de variables explicatives. La forêt donnant les meilleurs résultats est conservée.

Étant donné que le bagging et la randomisation sont, par définition, des forêts aléatoires elles aussi, nous appellerons notre implantation de la méthode de Breiman ForetM.

Combinaisons de méthodes

Lors des tests préliminaires, la méthode de randomisation n'a pas donné de très bons résultats, alors que le bagging, quand à lui, a fourni des résultats beaucoup plus prometteurs. Nous avons donc tenté d'améliorer la méthode en lui ajoutant une autre

source d'aléatoire : des échantillons bootstrap. Ainsi, une méthode qui, avec les données présentes, ne semble pas très bien fonctionner, devrait donner de meilleurs résultats si elle est jumelée avec le bagging, ce dernier semblant être capable de bien isoler une partie des non-acheteurs.

Comparaisons

Les comparaisons seront faites à l'aide d'un tableau des gains, construit de la façon suivante. Les observations sont regroupées par tranches de 5% selon leur probabilité d'achat. Par la suite, le profit généré par l'envoi d'offres à chacune des tranches est calculé. Un point de coupure est choisi en fonction du coût d'envoi d'une offre et du profit réalisé lorsque la réponse à l'offre est positive ; il indique le pourcentage de réponse nécessaire pour générer du profit. Ce point sert à déterminer le pourcentage des clients qui recevront des offres, les tranches dont le taux de réponse est supérieur à notre point de coupure. Il est donc possible de calculer le profit espéré en fonction du nombre d'offres envoyées et des réponses escomptées. Les différents modèles seront comparés sur ces profits et sur le pourcentage d'offres à envoyer, le but étant, bien entendu, de générer le profit maximal, et s'il faut départager entre plusieurs modèles, en envoyant des offres à un minimum de clients. La raison de ce deuxième critère est simple : nous souhaitons minimiser l'envoi d'offres à des personnes n'allant pas acheter, donc à des individus mal classifiés par le modèle. Si deux modèles génèrent le même profit mais avec un nombre d'envois différent, alors le modèle permettant l'envoi du plus petit nombre d'offres incluera moins d'individus mal classifiés que l'autre modèle.

Notons avant de terminer cette section, un point qui concerne toutes les méthodes : le nombre d'arbres généré pour chaque méthode. Pour les trois méthodes de forêt aléatoire, soit le bagging, la randomisation avec échantillons bootstrap et ForetM, 1000 arbres seront générés. Pour ce qui est du boosting, 10 roulements de 50 arbres chacun seront utilisés.

Conclusion

Cette section visait à présenter la méthodologie de la recherche. Le but de cette dernière est de comparer trois catégories de méthodes : la régression logistique seule, la méthode de classification actuellement utilisée par l'entreprise, et des méthodes ensemblistes appliquées à des arbres : le bagging, le boosting, la randomisation et les forêts aléatoires. Une nouvelle méthode sera aussi testée, la randomisation bâtie à partir d'échantillons bootstrap. Toutes les méthodes ensemblistes ont été codées par l'auteure. La comparaison s'effectuera sur le profit généré par chacune des méthodes.

Le fait que ce mémoire soit effectué en collaboration avec une entreprise privée implique certaines contraintes: il n'est pas possible de décrire en détail les données et méthodes de la compagnie, et il n'est pas non plus possible de décrire en détail les résultats qui sont obtenus. Mais l'important est présenté : la comparaison des performances des différentes méthodes.

Résultats

Présentation

Comme nous l'avons déjà mentionné, les résultats sont comparés à l'aide des mesures obtenues sur un tableau des gains. Il est possible d'effectuer ces comparaisons à l'aide de deux résultats, le taux de réponse de l'intervalle ou les profits, et ce pour des tranches de 5% d'envois. Ainsi, le premier 5% regroupe les premiers individus à qui une offre sera envoyée. Il est bien entendu souhaitable de rejoindre le plus d'acheteurs possible en envoyant le moins d'offres possible. On souhaite donc que le taux de réponse soit le plus grand possible dans les premières tranches de 5% ; il en est de même pour le profit. Pour faciliter les choses, les comparaisons sont faites à l'aide de graphiques. Étant donné que les graphiques des taux de réponses et de profits, bien qu'étant différents, mènent à la même conclusion, seul l'un des deux résultats est présenté. Comme les tendances sont plus faciles à voir à l'aide des profits, ce sont à partir de ceux-ci que les analyses ont été faites. Les données ayant servi à générer ces graphiques sont présentées en annexe A.

Les méthodes ne sont pas toutes présentes sur les prochains graphiques et ce pour deux raisons. D'une part, nous voulons éviter de surcharger les résultats avec un trop grand nombre de méthodes. D'autre part, les résultats de la randomisation de base sont relativement mauvais, nous avons donc jugé qu'il n'était pas nécessaire de les présenter. Les méthodes présentes dans les graphiques sont les suivantes.

Tableau 2 : Légende des noms des méthodes ensemblistes

Légende	Méthode
Sans	Régression logistique de base
Base	Méthode de l'entreprise
Améliorée	Tentative d'amélioration de la méthode précédente
Bagging	Bagging
Boosting	Boosting
Randbag	Randomisation avec échantillons bootstrap
ForetM10(20)%	ForetM avec le pourcentage de variables sélectionnées pour chaque embranchement

Rappelons-nous que deux modèles ont été construits pour la méthode ForêtM, le premier avec 10% des variables explicatives, le second avec 20%. Afin de ne pas surcharger les résultats, seule la méthode donnant les meilleurs résultats est présentée. Pour les personnes qui seraient intéressées à comparer les deux modèles, les résultats sont présents en annexe.

Ensembles utilisés

Sept ensembles de données ont été utilisés pour construire les modèles, chacun correspondant à un segment de la clientèle qu'il n'est pas possible de décrire étant donné la confidentialité des données. Les résultats de ceux-ci seront d'abord présentés. Par la suite, deux nouveaux ensembles de données sont utilisés afin de vérifier si les modèles sont stables dans le temps. Ce dont des données provenant de segments ayant servi à la construction de deux des premiers modèles, mais deux mois plus tard dans le temps. Ces résultats seront présentés à la suite des premiers. Par la suite, le temps d'exécution de chaque méthode sera analysé.

Les premiers résultats ont été générés à l'aide des données qui ont servi à bâtir les modèles. Nous avons décidé de ne pas utiliser deux ensembles différents pour le test et l'entraînement, car la méthode de l'entreprise est ainsi construite, les méthodes doivent être générées de façon semblable afin que la comparaison se fasse de façon juste.

Premiers résultats

Avant de présenter les résultats, prenons quelques lignes pour bien comprendre les graphiques. Rappelons-nous que chaque modèle induit une probabilité d'achat. Les observations peuvent donc être ordonnées à partir de cette probabilité. Une fois cet ordonnancement fait, les observations sont regroupées en tranches de 5%. Il est possible que quelques groupes ne contiennent pas exactement 5% du nombre de données, deux probabilités égales étant placées dans le même groupe. Ainsi, si 7% des observations ont une prédiction de 1, le premier groupe formé contiendra 7% des données, et le second seulement 3%. Une fois les données ainsi regroupées, il est possible de calculer le profit que générera l'envoi aux clients de chaque groupe. Les graphiques présentent les profits cumulés. Regardons la Figure 2. En envoyant l'offre à seulement 5% des individus, c'est la méthode ForêtM qui génère le plus de profits, environ 37 000$. Par contre, si l'offre est envoyée à 10% des individus, c'est le boosting qui permet de générer le plus de profits.

Notons une particularité des courbes du bagging, de la randomisation avec échantillons bootstrap et de ForêtM sur le premier graphique : la ligne droite au début de la courbe. Ceci s'explique par le rassemblement, dans le même groupe, des probabilités égales. Dans les trois cas, la probabilité d'achat de 1 a été attribuée à environ 30% des

observations ; celles-ci ont donc été toutes mises dans le même groupe et le profit a ainsi été calculé pour l'envoi à ces 30% du nombre de clients, et non pas à 5% comme dans les autre cas.

Pour comparer rapidement les modèles, il est possible de prendre la règle suivante : faire le plus de profits le plus rapidement possible, en utilisant, s'il faut départager deux méthodes, le critère du nombre d'envois : entre deux modèles qui donnent des résultats semblables, on choisira celui qui permet l'envoi d'offres au plus petit nombre de personnes possible. Voyons maintenant les résultats sur les sept ensembles.

Figure 2 : Profits pour l'ensemble 1

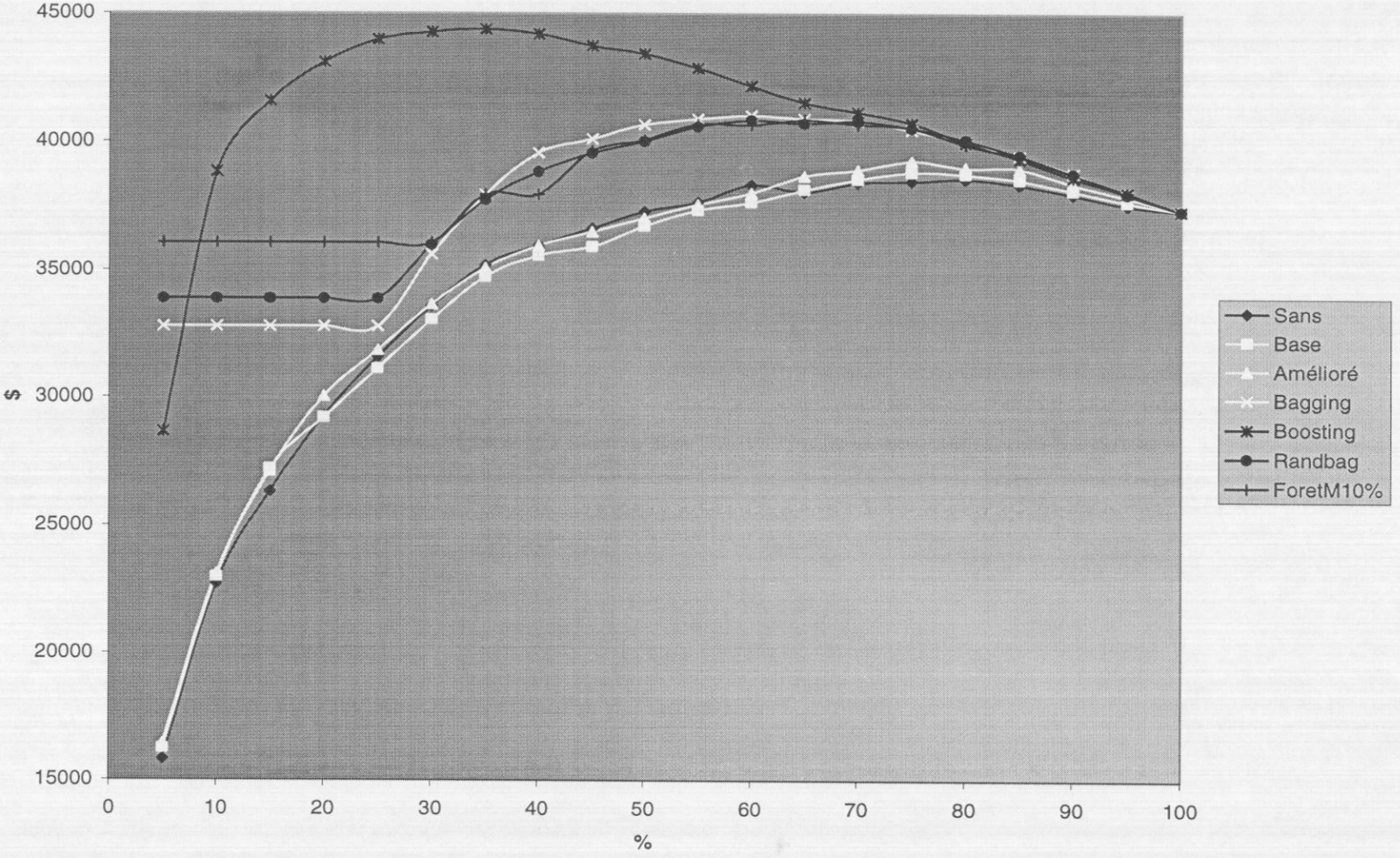


Figure 3 : Profits pour l'ensemble 2

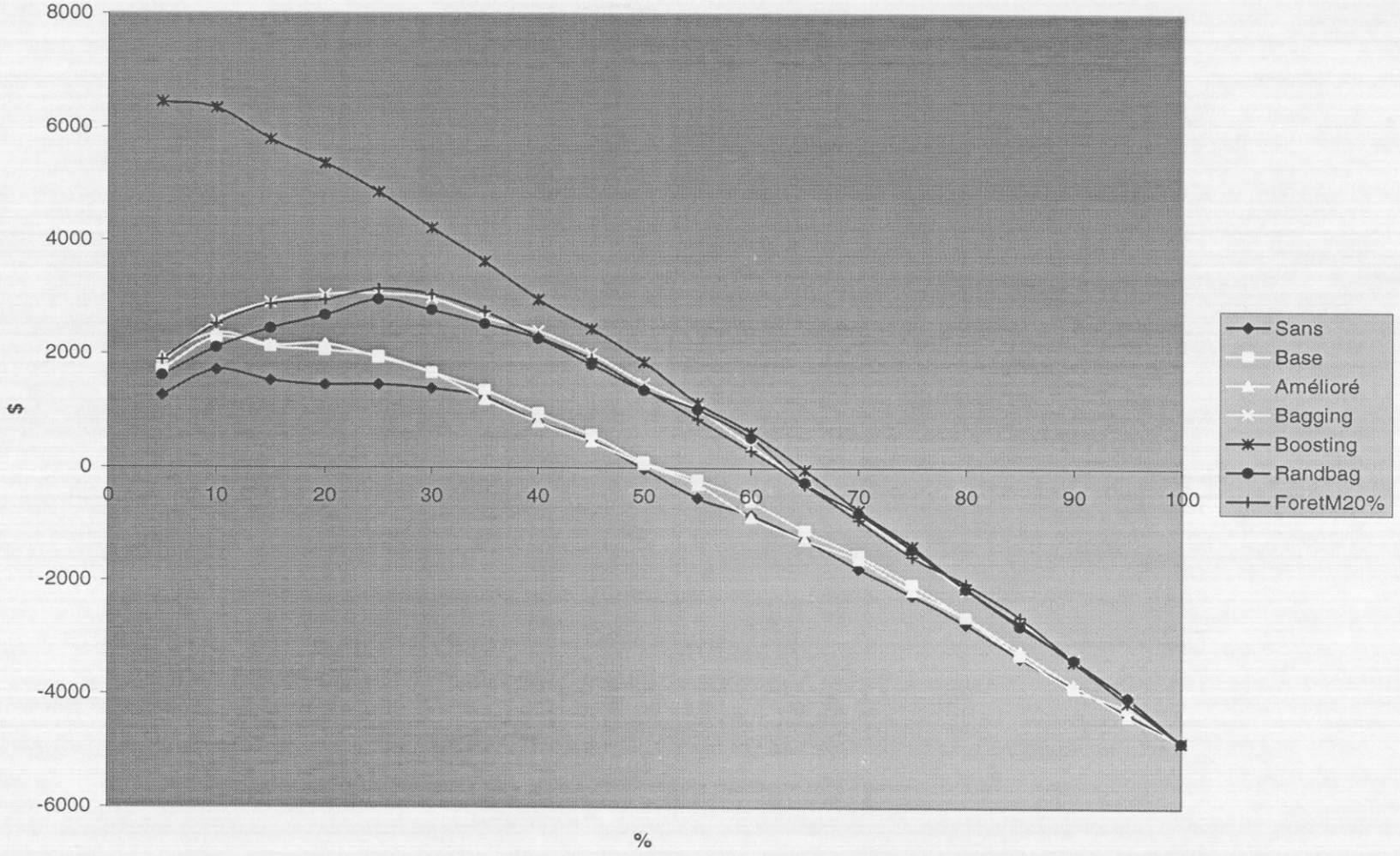


Figure 4 : Profits pour l'ensemble 3

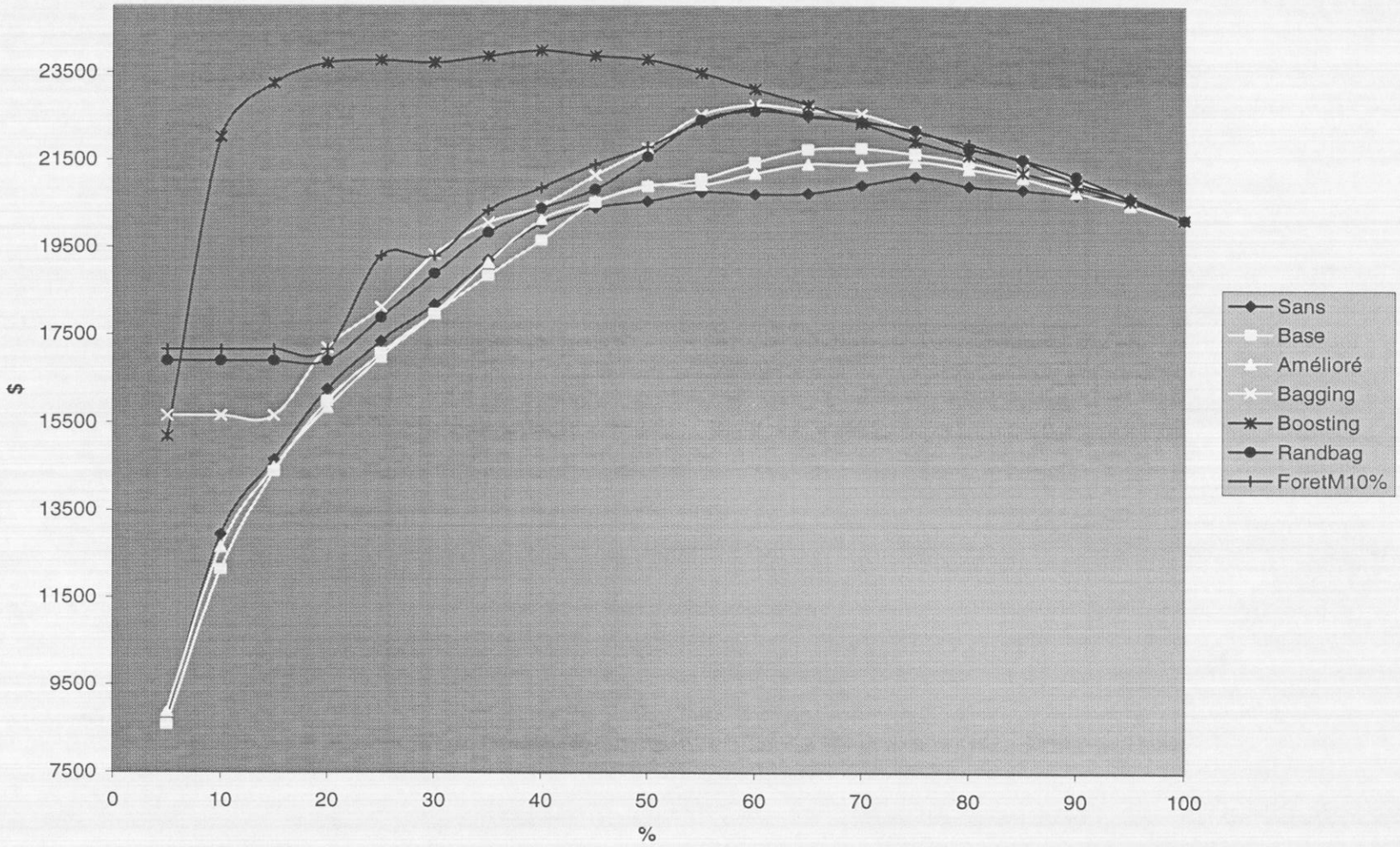


Figure 5 : Profits pour l'ensemble 4

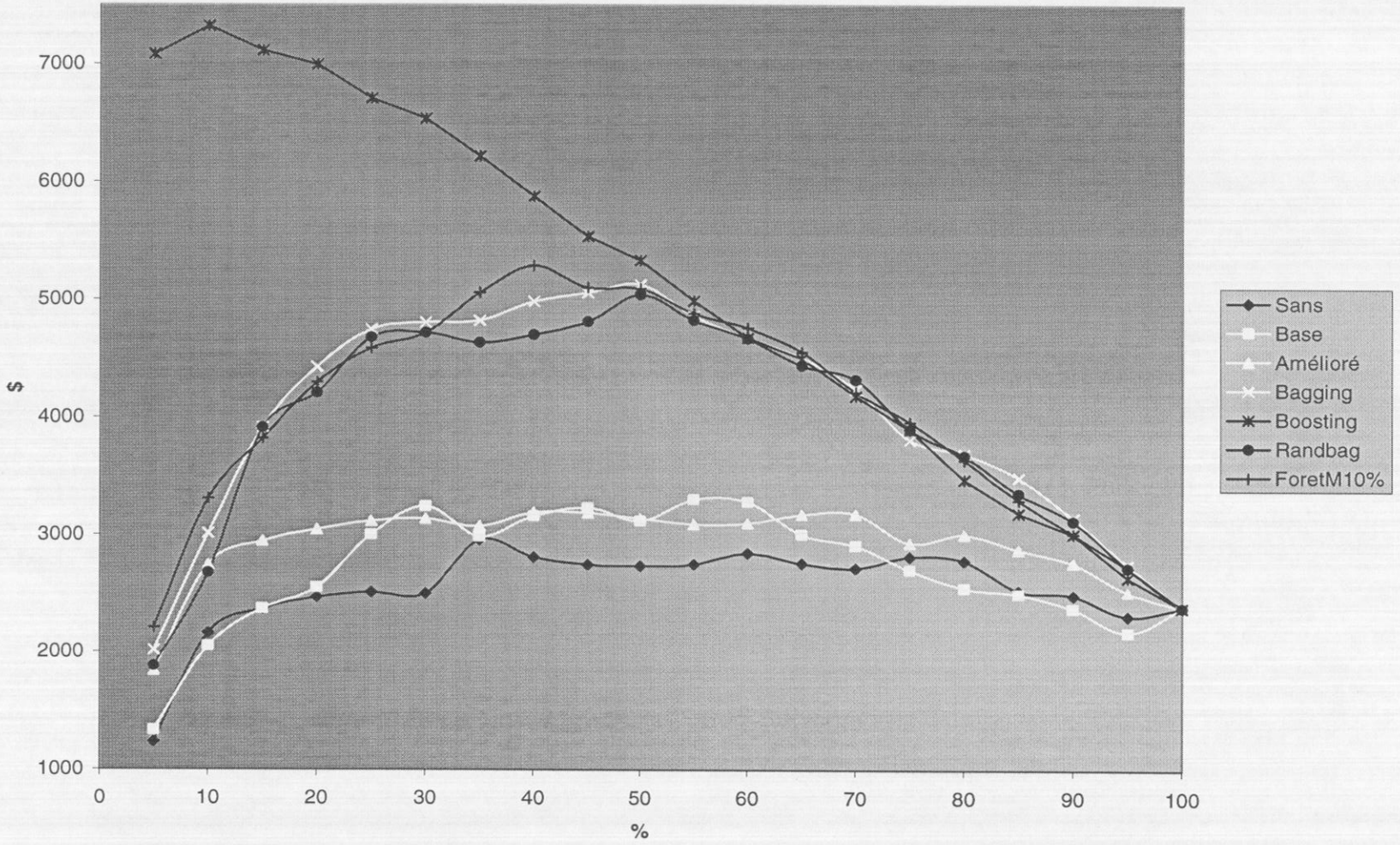


Figure 6 : Profits pour l'ensemble 5

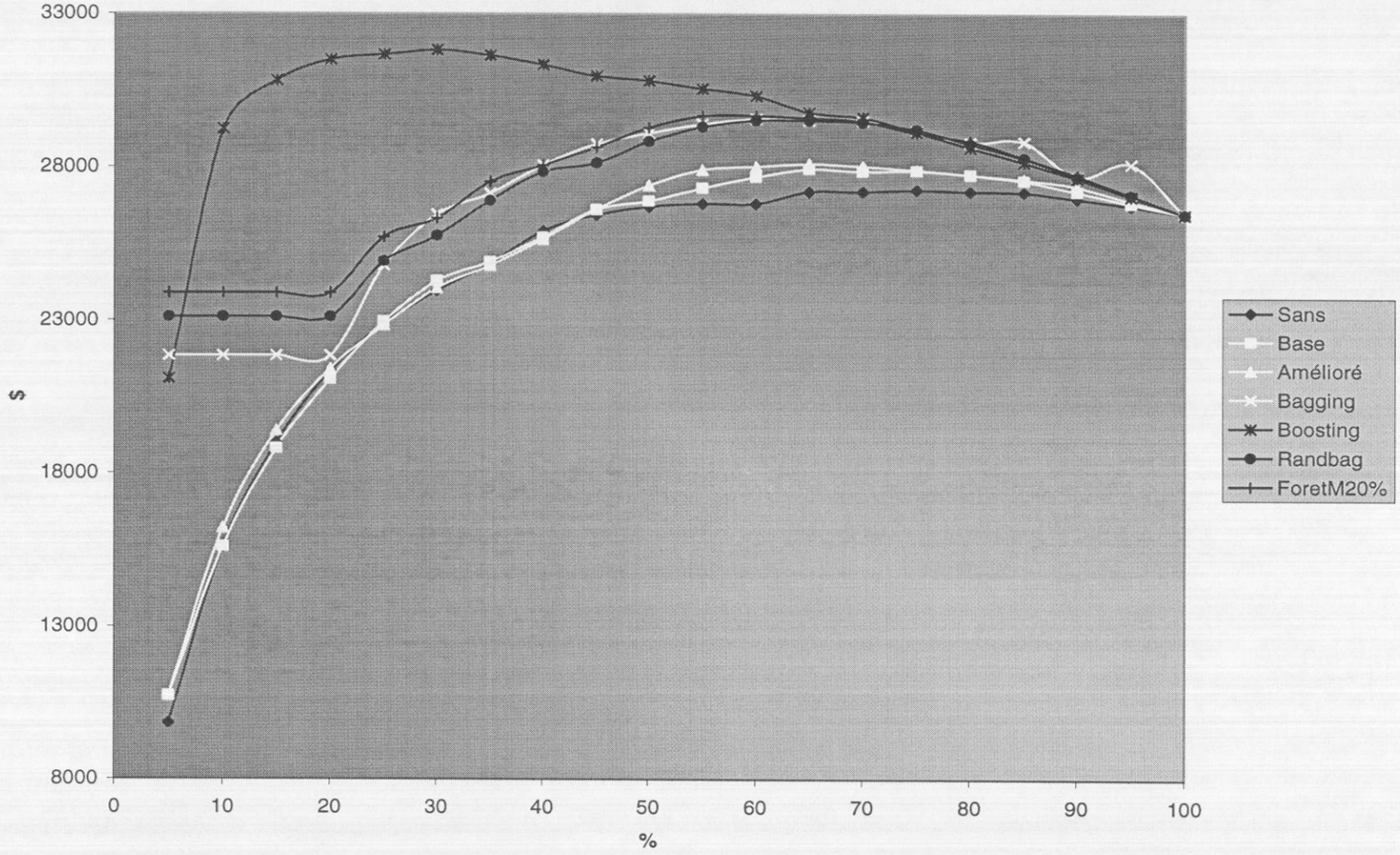


Figure 7 : Profits pour l'ensemble 6

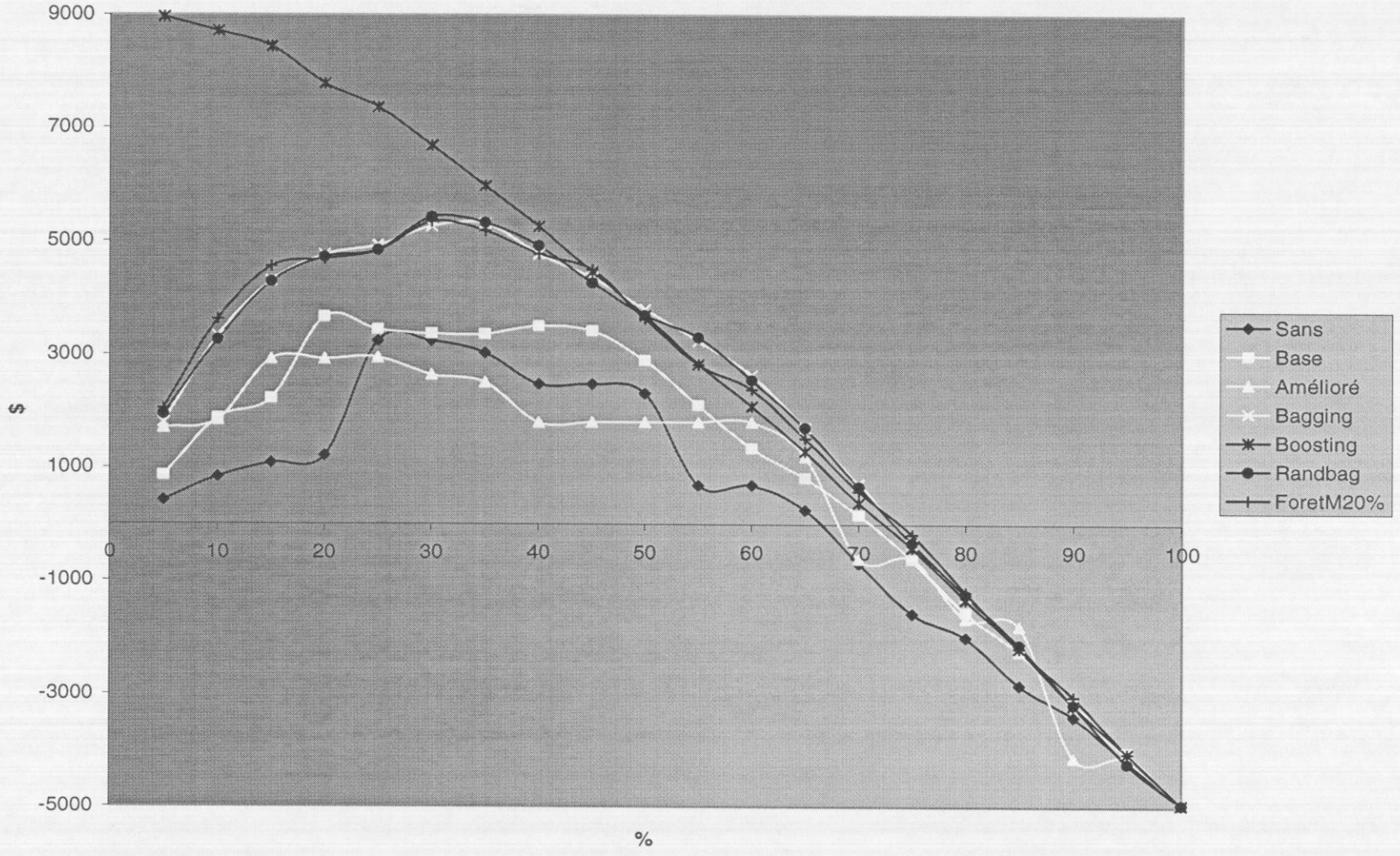
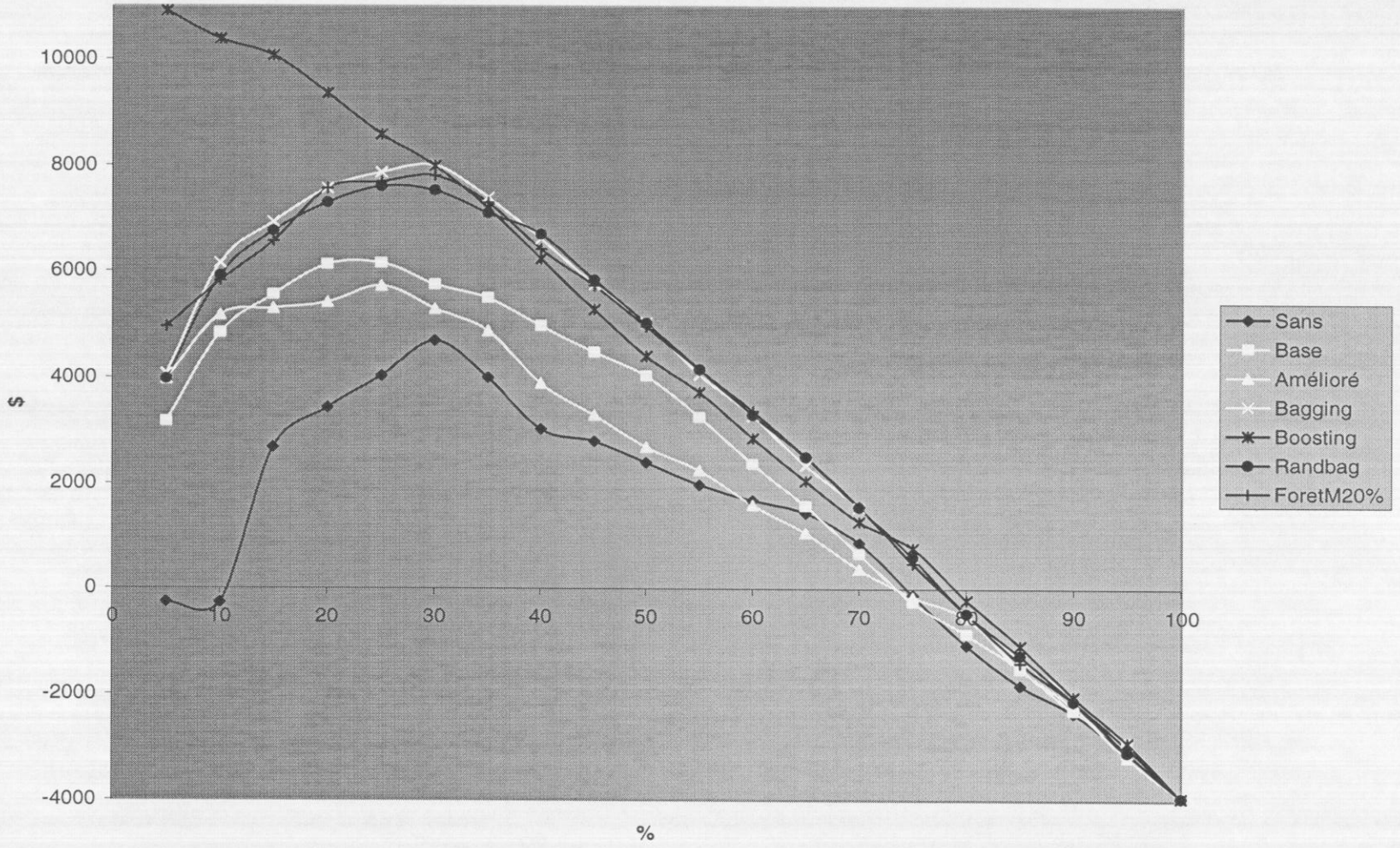


Figure 8 : Profits pour l'ensemble 7



Au regard des graphiques, une tendance très nette se dessine : le boosting vient en tête, puis les trois autres méthodes ensemblistes se suivent de près, et viennent finalement les trois méthodes non ensemblistes, qui donnent des résultats semblables entre elles, mais moins bons que les autres. Il faut toutefois mettre les résultats en perspective, car les profits ont été estimés avec les mêmes observations que celles ayant servi à ajuster les modèles. Par leur nature, les méthodes ensemblistes utilisées, à l'exception du boosting, sont en quelque sorte immunisées contre le sur-ajustement. Le boosting par contre ne l'est pas et il est possible que sa performance soit surestimée dans certains cas même si, rappelons-le, une condition d'un minimum de 5% des observations dans chaque feuille est appliquée. Ceci sera exploré partiellement dans la prochaine section à l'aide de deux échantillons de validation prélevés plus loin dans le temps.

Le boosting se distingue des autres méthodes sur deux points : d'abord, il donne des résultats définitivement supérieurs à ceux des autres méthodes lorsque la performance est jugée sur l'échantillon d'entraînement; ensuite, la forme de sa courbe diffère nettement de celle des autres méthodes et le maximum est atteint plus rapidement. Ces deux particularités sont causées par le même phénomène : le boosting réussit à identifier plus rapidement les acheteurs ; ceux-ci sont alors classifiés dans les premières tranches de 5%, ce qui donne lieu à des profits plus élevés. Lorsque la majeure partie des acheteurs est identifiée dans les cinq ou dix premiers pour-cent, le profit maximal est alors atteint dès l'envoi d'offres aux individus ainsi identifiés. Cette meilleure performance du boosting s'explique par sa tendance à mettre l'accent sur les données mal classifiées : puisqu'au départ les acheteurs sont tous mal classifiés, alors l'accent est mis sur eux lors de la construction des arbres et les acheteurs sont graduellement mieux classifiés.

L'utilisation des coûts de mauvaise classification n'est alors pas nécessaire, ni même souhaitable. Rappelons que dans le cas des autres méthodes, il faut utiliser les coûts de mauvaise classification afin de forcer les arbres à assigner aux feuilles la caractéristique d'acheteur, mais que, ce faisant, la plupart des observations sont classifiées comme acheteurs et peu de feuilles seront classifiées comme étant non-acheteurs.

Après un certain pourcentage, toutes les méthodes auront identifié sensiblement la même proportion de non-acheteurs, et cette proportion sera supérieure à celle obtenue à l'aide des méthodes non ensemblistes, étant donné que dans les deux cas les individus qui n'ont aucune chance de payer seront bien identifiés et relégués dans les derniers pourcentages. Par contre, le boosting, en raison de sa tendance naturelle à mettre l'accent sur les acheteurs étant donné leur petit nombre, sera capable d'identifier plus précisément ces acheteurs. La différence entre le boosting et les autres méthodes peut s'expliquer de la façon suivante : le boosting isole les acheteurs, alors que les autres méthodes isolent les non-acheteurs, d'où la meilleure classification du boosting sur l'échantillon d'entraînement.

Comparons maintenant les autres méthodes ensemblistes. Elles donnent toutes trois donnent des résultats comparables. Dans trois des cas étudiés (ensembles 1, 3 et 7), le bagging permet de générer légèrement plus de profits. Dans trois autres cas (ensembles 2, 4 et 5), ce sont les forêts aléatoires qui mènent, et finalement, dans seulement l'un des cas (ensemble 6), c'est la randomisation qui donne de meilleurs résultats. Par contre, dans les ensembles 1, 5 et 6, les résultats sont tellement rapprochés qu'il est hasardeux d'affirmer qu'une des trois méthodes est meilleure que les autres. Les trois méthodes ont en commun l'utilisation d'échantillons bootstrap. Nous pourrions donc conclure

qu'ajouter une autre source d'aléatoire au bagging (tel que nous l'avons fait pour nos implantations de la randomisation et des forêts aléatoires), n'est pas nécessaire puisque la plupart du temps le boosting prédomine sur l'échantillon d'entraînement. Mais cette conclusion serait hâtive. Effectivement, afin d'accélérer le processus, le nombre optimal d'embranchements ou de variables à utiliser pour la randomisation et les forêts aléatoires n'a pas été utilisé. En effet, nous avons tenté de trouver une règle qui convenait bien à tous les ensembles, mais il se peut que pour chaque ensemble, il existe un nombre optimal de variables à choisir, dans le cas des forêts aléatoires, ou un nombre optimal d'embranchements pour la randomisation, qui donnerait de meilleurs résultats que ceux obtenus. Ceci pourrait expliquer les meilleurs résultats qu'ont donné les forêts aléatoires dans trois des cas ; rappelons-nous que dans le cas des forêts aléatoires, deux forêts différentes ont été construites par ensemble, contre une seule pour la randomisation. Ainsi, avec le nombre optimal de variables ou d'embranchements, il se pourrait que les résultats des forêts aléatoires ou la randomisation donnent de meilleurs résultats et surpassent ceux du bagging. Notons finalement que les trois méthodes atteignent leur optimum au même point : même si l'une des méthodes génère plus de profits que les deux autres, elle ne permet pas d'envoyer l'offre à moins de clients.

Il reste à comparer les trois méthodes non ensemblistes. La régression logistique donne de moins bons résultats que la méthode de l'entreprise, ce qui était à prévoir, mais la tentative d'amélioration de la méthode n'a pas donné les résultats escomptés. Étant donné la confidentialité entourant la méthode utilisée par l'entreprise, il n'est pas possible de discuter davantage de ces résultats.

Validation

Voyons maintenant ce que donne la validation. On appelle ici validation l'application des modèles à des données qui ont été recueillies deux mois après les autres données. Effectivement, étant donné que les modèles probabilistes sont utilisés après que les données ayant servi à les construire aient été recueillies, il est souhaitable que leur performance soit stable à travers le temps. Les ensembles de validation 1 et 2 contiennent des données provenant des mêmes segments que les ensembles 1 et 2, mais recueillies deux mois plus tard. Ainsi, en soumettant les nouvelles données aux modèles construits à l'aide des ensembles 1 et 2, nous pourrons savoir jusqu'à quel point les modèles sont stables à travers le temps. Les résultats obtenus sont présentés dans les deux graphiques suivants.

Figure 9 : Profits pour l'ensemble de validation 1

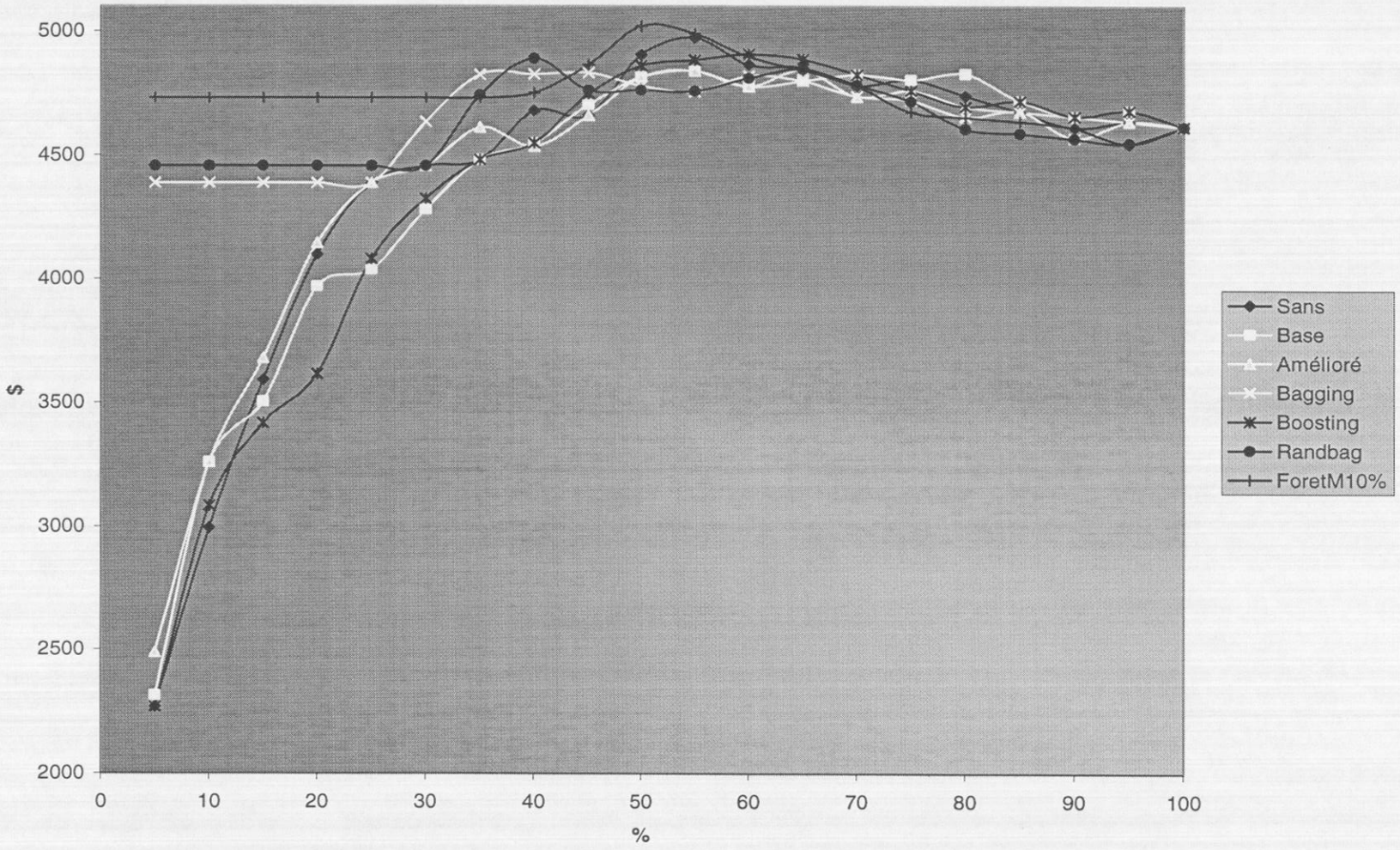
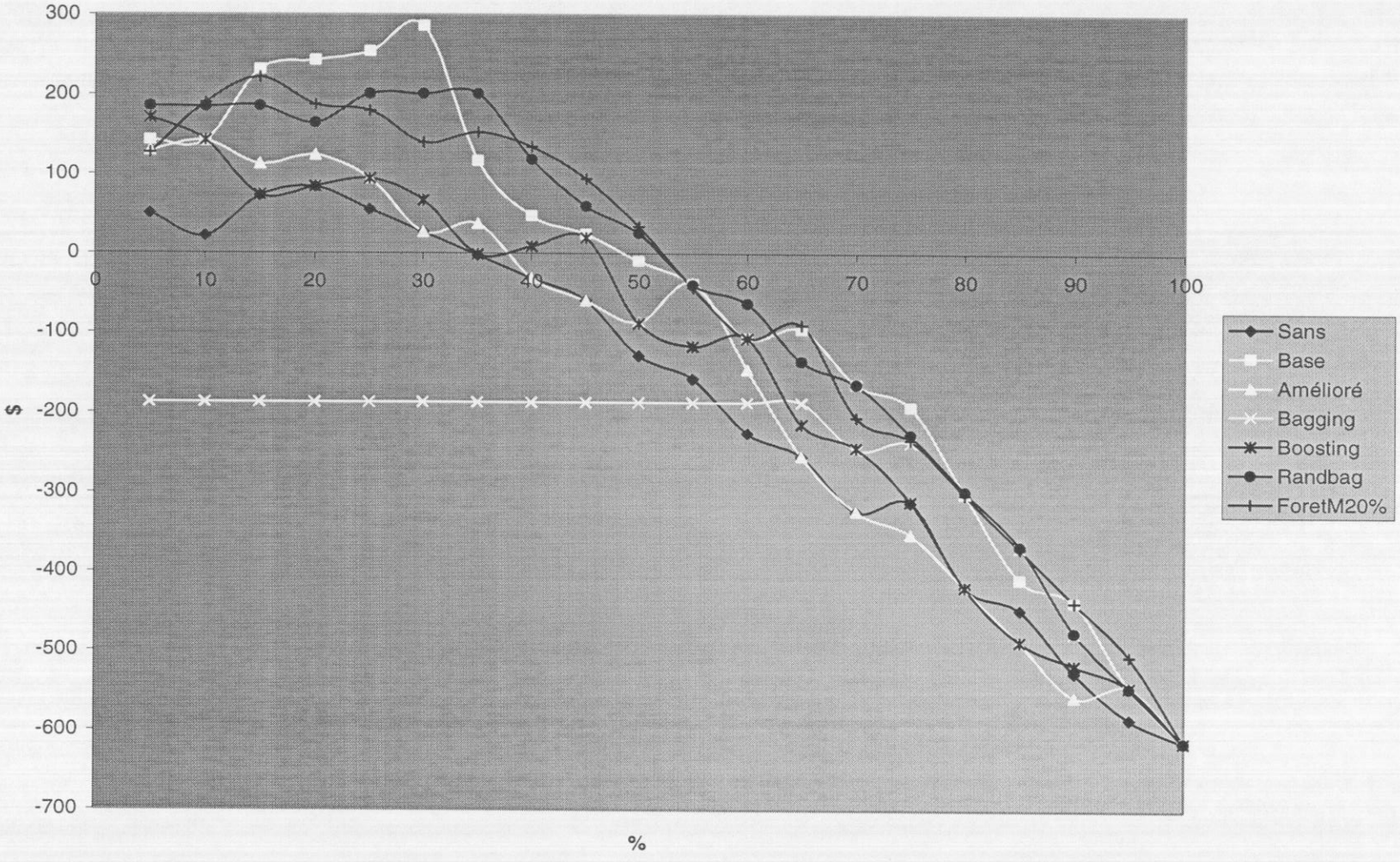


Figure 10 : Profits pour l'ensemble de validation 2



Les résultats sont différents de ceux obtenus précédemment. Effectivement, les méthodes ensemblistes sont moins prédominantes. Dans le premier cas, c'est la méthode ForetM qui génère le plus de profits, suivie de près par la régression logistique, puis le boosting. Dans le second cas, c'est la méthode de base qui donne de meilleurs résultats, suivie de ForetM, puis de la randomisation avec échantillons bootstrap. Les méthodes ensemblistes donnent donc des résultats maintenant comparables aux autres méthodes. Notons aussi que le boosting ne se distingue plus comme auparavant. Ces résultats signifient que les modèles ne sont pas stables à travers le temps pour les jeux de données considérés, alors que la régression logistique et la méthode de la compagnie le sont davantage. Par contre, ils sont comparables à ceux de la méthode déjà utilisée par l'entreprise et sont meilleurs que la régression logistique. Des espoirs sont donc encore permis : des modifications qui leur permettraient d'être plus stable à travers le temps pourraient peut-être leur permettre de battre les modèles utilisés actuellement.

Temps d'exécution

Il reste un point à traiter en ce qui a trait à la comparaison des méthodes ensemblistes : leur temps d'exécution. Effectivement, une méthode qui est légèrement moins performante peut être plus attrayante que celle plus performante si son temps d'exécution est moindre. Les tests ont été faits à l'aide de l'ensemble numéro 4 et 25 arbres ont été construits pour chacune des méthodes. Les méthodes ont été générées sur un Pentium 3, un portable Inspiron 4100 de Dell, ayant 128 Mo de mémoire vive et un disque dur de 1 GHz. Le temps d'exécution moyen, pour construire un arbre avec chaque méthode, est présenté dans le tableau 3, suivi du temps relatif de construction, par rapport

à la construction d'un arbre de classification sans modification.

Tableau 3 : Temps d'exécution des méthodes ensemblistes

Méthode	Arbre	Bagging	Boosting	Rangbag	RF10%bag	RF20%bag
Temps	7.6	6	8.4	7	0.6	0.96
Temps relatif	100	78.9	110	92.1	7.9	12.6

Avant de comparer les temps d'exécution, il faut souligner que l'opération qui prend le plus de temps lors de la construction des arbres est le calcul des différents embranchements. Effectivement, le nombre d'embranchements est la somme des différentes valeurs de toutes les variables. Ainsi, une méthode qui permettra de faire diminuer le nombre d'embranchements à tester sera probablement plus rapide.

L'implantation des forêts aléatoires est beaucoup plu rapide que les autres méthodes, ce qui représente l'un de ses grands avantages. Ceci s'explique par le fait que cette méthode considère moins de variables que les autres méthodes pour le choix de l'embranchement ; par conséquent, beaucoup moins d'embranchements doivent être calculés et comparés. Vient ensuite le bagging. Nous aurions pu nous attendre à ce qu'un arbre avec échantillon bootstrap soit moins rapide à construire qu'un arbre de base, en considérant le temps demandé pour construire l'échantillon bootstrap. Mais dans la mesure où toutes les observations ne se retrouvent pas dans le nouvel échantillon et que certaines observations sont en double, le nombre d'embranchements à tester se trouve ainsi diminué. Le nombre de valeurs différentes que peut prendre une variable étant moindre, le temps de construction diminue. La randomisation avec échantillons bootstrap est plus rapide qu'un arbre de base pour cette même raison, mais est plus lente que le

bagging. La raison de ce comportement vient surtout de l'implémentation qui a été faite ; une autre implantation pourrait favoriser la randomisation. Notons par contre que si l'échantillon utilisé pour construire l'arbre de randomisation était l'échantillon d'entraînement et non pas un échantillon bootstrap, alors la vitesse serait inférieure à celle du bagging, et plus semblable à celle de construction d'un arbre de base. Le boosting, finalement, est la méthode la moins rapide de toutes, étant donné le calcul des nouveaux poids après chaque arbre construit.

Conclusion

Cette recherche visait à comparer plusieurs méthodes ensemblistes à la régression logistique et à la méthode déjà utilisée par le Sélection du Reader's Digest afin de vérifier si une méthode ensembliste utilisant les arbres de décision peut surpasser la méthode actuellement utilisée pour identifier les acheteurs potentiels. Plusieurs méthodes ont été construites, et les résultats de quatre d'entre elles ont été présentés, ces méthodes étant le bagging, le boosting, les forêts aléatoires et la randomisation, cette dernière méthode utilisant des échantillons bootstrap plutôt que les ensembles d'entraînement de base.

Lors des premiers tests, où la validation se fait avec les mêmes données que celles ayant servies à l'ajustement des modèles, le boosting est sorti grand gagnant, surpassant, et de loin, les autres méthodes lors de la comparaison des profits générés : les profits étaient supérieurs à ceux générés par les autres méthodes, et ceci en envoyant les offres à un nombre restreint de clients. Venaient ensuite les trois autres méthodes ensemblistes, dont les résultats différaient peu, suivies par la méthode utilisée par l'entreprise et, pour finir, la régression logistique. Par contre, la validation à l'aide de données prises plus tard dans le temps a montré que les méthodes ensemblistes étaient moins stables à travers le temps. Elles donnent tout de même des résultats quelque peu supérieurs à la régression logistique et comparables à la méthode utilisée par l'entreprise. La méthode ForetM s'est distinguée des autres méthodes, donnant de bons résultats lors des deux validations et réussissant à surpasser la méthode de l'entreprise lors de la première. Une dernière comparaison des méthodes ensemblistes en fonction du temps d'exécution a montré que les forêts aléatoires étaient beaucoup plus rapides que les autres méthodes, et que le

boosting était une méthode plus lente que la construction de simples arbres de décision.

La conclusion à tirer de ces résultats est la suivante : si le modèle doit servir à classifier immédiatement après avoir été construit, alors les méthodes ensemblistes semblent prometteuses. Par contre, si les données sont prises plus tard dans le temps, il n'est pas nécessaire de remplacer la méthode existante. La méthode ForetM est prometteuse et, avec un peu d'ajustements pour entre autres augmenter sa stabilité, elle pourrait éventuellement donner de meilleurs résultats que la méthode de l'entreprise. La stabilité à travers le temps est une propriété qui est importante dans plusieurs domaines et il serait intéressant de voir s'il est possible de tirer parti des arbres en augmentant leur robustesse aux variations longitudinales.

Bibliographie

Bauer, Eric; Kohavi, Ron; *An Empirical Comparison of Voting Classification Algorithms: Bagging, Boosting and Variants*, Machine Learning, Vol 36(1), p 105-139, July1999;

Bay, Stephen D; Pazzani, Micheal J; *Detecting Change in Categorical Data: Mining Contrast Sets*, Proceedings of the Fifth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, p 302-306, 1999;

Bhattacharyya, Siddhartha; *Evolutionary Algorithms in Data Mining : Multi-Objectives Performance Modeling for Direct Marketing*, Proceedings of the Sixth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, p 465-473, August 2000;

Bounsaythip, Catherine; Rinta-Runsala, Esa; *Overview of Data Mining for Customer Behavior Modeling*, Research Report, VTT Information Technology, 53 pp, 2001;

Breiman, Leo; Friedman,, Jerome H.; Olshen, Richard A.; Stone, Charles J.; *Classification and Regression Trees*, Chapman & Hall/CRC, Boca Raton, 535 pp, 1984;

Breiman, Leo; *Bagging Predictors*, Machine Learning, Vol 24(2), p 133-140, Août 1996;

Breiman, Leo; *Arcing Classifiers*, Annals of Statistics, Vol 26(3), p 801-849, Septembre 1998;

Breiman, Leo; *Randomizing Outputs to Increase Prediction Accuracy*, Machine Learning, Vol 40(3), p 229-242, Septembre 2000;

Breiman, Leo; *Random Forests*, Machine Learning, Vol 45(1), p 5-32, Octobre 2001;

Chou, Paul B.; Grossman, Edna; Gunopulos, Dimitros; Kamesam, Pasumarti; *Identifying Prospective Customers*, Proceedings of the Sixth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, p 447-456, August 2000;

Dietterich, Thomas G.; *An Experimental Comparison of Three Methods for Constructing Ensemble of Decision Trees*, Machine Learning, Vol 40(2), p 139-157, Août 2000a;

Dietterich, Thomas G.; *Ensemble Methods in Machine Learning*, Multiple Classifier Systems, p 1-15, 2000b;

Freund, Y; Shapire, R; *Experiments with a new boosting algorithm*. Proceedings if the International Conference on Machine Learning, pp 148-156, Morgan Kaufmann, 1996;

Freund, Y; Shapire, R; *A Brief introduction to Boosting*, Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence, 1999;

Gallant, Steve; Piatetsky-Shapiro, Gregory; Pyle, Dorian; *Successful Customers Relationship Management in Financial Applications*, Tutorial Notes of the Sixth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, p. 165-241, August 2000;

Hosmer, D. W.; Lemeshow, S.; *Applied Logistic Regression*, Wiley series in probability and mathematical statistics. 307 p, 1989;

Kass, G.V.; *An exploratory technique for investigating large quantities of categorical data*, *Applied Statistics*, p 119-127, 1980.

Kleingerg, Jon; Papadimitriou, Christos; Raghavan, Prabhakar; *Segmentation Problems*, Proceedings of the Thirtieth Annual ACM Symposium on Theory of Computing, p473-482, May 1998;

Liu, Bing; Xia, Yiyuan; Yu, Philip, S. Clustering Through Decision Tree Construction, Proceedings of the Ninth International Conference on Information and Knowledge Management, p20-29, November 2000;

Ng, KianSing; Liu, Huam; Kwah, HweeBong; A Data Mining Application: Customer Retention at the Port of Singapore Authority (PSA), ACM SIGMOD Record, Proceedings of the 1998 ACM SIGMOD International Conference on Management of Data, Vol 27(2), p522-525, June 1998;

Opitz, David; Maclin, Richard; *Popular Ensemble Methods: An Empirical Study*, Journal of Artificial Intelligence Research, Vol 11, p 169-198, 1999;

Padmanabhan, Balaji; Zheng, Zhiqiang; Kimbrough, Steven O; *Personalization for Incomplete Data : What You Don't Know Can Hurt*, Proceedings of the Seventh ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, p. 154-163, 2001;

Piatetsky-Shapiro, Gregory; Masand, Brij; *Estimating Campain Benefits and Modeling Lifts*, Proceedings of the Fifth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, p 185-193 , August 1999;

Quinlan, J. R.; *C4.5: Programs for Machine Learning*, Morgan Kaufmann, San Mateo, 302 pp, 1993;

Quinlan J. R.; *Bagging, Boosting and C4.5*, Proceedings of the Thirteenth National Conference on Artificial Intelligence, p725-730, Cambridge, MA, AAAI Press / MIT Press, 1996;

Annexe A: Tableaux des profits

Cette première annexe regroupe les tableaux contenant les données relatives aux profits. Ce sont ces données qui ont servi à générer les graphiques présentés dans la section résultats. De plus, les résultats des deux forêts aléatoires y sont présents, à l'inverse des graphiques où seulement la forêt présentant les meilleurs résultats avait été incluse.

Tableau 1 : Profits pour l'ensemble 1

%	Sans	Base	Amélioré	Bagging	Boosting	Randbag	ForetM 10%	ForetM 20%
5	15807	16247	16423	32788	28667	33873	36080	35818
10	22723	22988	23120	32788	38835	33873	36080	35818
15	26340	27220	27132	32788	41586	33873	36080	35818
20	29252	29208	30044	32788	43092	33873	36080	35818
25	31592	31108	31856	32788	43972	33873	36080	35818
30	33579	33051	33667	35607	44276	35985	36080	35818
35	35216	34732	35084	37945	44369	37731	37956	37978
40	35971	35575	35971	39564	44192	38834	37956	38839
45	36638	35934	36506	40109	43737	39575	39667	39504
50	37307	36734	37042	40686	43425	40024	40087	39940
55	37666	37358	37576	40900	42880	40610	40705	40392
60	38333	37673	38025	41043	42184	40863	40668	40433
65	38076	38164	38692	40920	41539	40742	40888	40471
70	38430	38568	38920	40840	41169	40828	40702	40585
75	38487	38839	39280	40479	40759	40549	40556	40412
80	38583	38759	39023	40033	40006	40092	39922	40063
85	38362	38546	38986	39481	39362	39500	39436	39472
90	37981	38113	38377	38820	38641	38779	38770	38786
95	37549	37681	37857	37984	38049	37988	38059	37947
100	37336	37336	37336	37336	37336	37336	37336	37336

Tableau 2 : Profits pour l'ensemble 2

%	Sans	Base	Amélioré	Bagging	Boosting	Randbag	ForetM 10%	ForetM 20%
5	1279	1717	1933	1826	6448	1620	1844	1894
10	1723	2280	2360	2572	6336	2111	1844	2518
15	1531	2127	2156	2894	5785	2445	2688	2866
20	1458	2054	2174	3040	5354	2678	2708	2946
25	1465	1956	1934	3057	4844	2951	2985	3132
30	1392	1670	1670	2973	4215	2762	2833	3041
35	1240	1364	1199	2644	3625	2523	2502	2738
40	808	967	808	2400	2955	2273	2263	2304
45	497	587	462	2005	2445	1820	1854	1914
50	-13	105	67	1487	1856	1364	1347	1390
55	-523	-206	-284	1038	1147	1030	886	856
60	-834	-596	-874	392	637	537	235	307
65	-1266	-1106	-1266	-241	-33	-253	-166	-291
70	-1776	-1537	-1617	-833	-742	-800	-824	-917
75	-2246	-2047	-2166	-1508	-1371	-1434	-1324	-1562
80	-2756	-2642	-2637	-2160	-2120	-2135	-2056	-2045
85	-3346	-3306	-3226	-2705	-2749	-2790	-2738	-2632
90	-3856	-3896	-3817	-3372	-3419	-3383	-3393	-3381
95	-4327	-4247	-4367	-4045	-4129	-4051	-4039	-4043
100	-4837	-4837	-4837	-4837	-4837	-4837	-4837	-4837

Tableau 3 : Profits pour l'ensemble 3

%	Sans	Base	Amélioré	Bagging	Boosting	Randbag	ForetM 10%	ForetM 20%
5	8742	8592	8862	15664	15198	16897	17154	16457
10	12950	12139	12650	15664	22019	16897	17154	16457
15	14665	14395	14485	15664	23254	16897	17154	16457
20	16260	15990	15840	17202	23708	16897	17154	16457
25	17344	17074	16984	18132	23771	17894	19301	17894
30	18189	17979	17979	19347	23715	18892	19301	19328
35	19214	18853	19154	20075	23868	19850	20358	20067
40	20058	19667	20147	20455	23992	20399	20869	20720
45	20422	20542	20572	21157	23876	20837	21408	21209
50	20576	20906	20936	21780	23789	21581	21814	21666
55	20789	21089	20944	22543	23492	22431	22380	22419
60	20733	21454	21214	22778	23136	22627	22705	22478
65	20766	21757	21426	22648	22778	22541	22505	22507
70	20950	21791	21400	22567	22361	22390	22419	22578
75	21133	21664	21494	22184	21944	22189	22148	22149
80	20927	21468	21318	21801	21618	21786	21886	21766
85	20840	21201	21111	21534	21231	21533	21501	21525
90	20723	21054	20784	21112	20934	21146	21023	21109
95	20637	20577	20487	20638	20577	20637	20666	20669
100	20160	20160	20160	20160	20160	20160	20160	20160

Tableau 4 : Profits pour l'ensemble 4

%	Sans	Base	Amélioré	Bagging	Boosting	Rand bag	ForetM 10%	ForetM 20%
5	1232	1333	1840	2015	7083	1881	2207	2143
10	2160	2049	2746	3012	7321	2682	3312	3069
15	2370	2370	2949	3916	7117	3912	3823	3626
20	2470	2553	3049	4429	6996	4210	4296	4245
25	2515	3005	3119	4746	6710	4680	4586	4747
30	2504	3249	3140	4803	6535	4716	4716	4609
35	2963	2990	3075	4822	6221	4633	5059	4846
40	2814	3166	3194	4983	5879	4703	5288	5084
45	2750	3237	3191	5059	5537	4813	5103	5070
50	2740	3126	3145	5128	5334	5043	5093	5061
55	2756	3308	3098	4861	4992	4824	4883	5019
60	2851	3288	3105	4674	4678	4669	4760	4875
65	2763	3011	3181	4510	4502	4444	4558	4582
70	2725	2914	3184	4243	4188	4327	4219	4214
75	2824	2707	2937	3816	3901	3901	3963	3957
80	2786	2553	3010	3697	3478	3678	3644	3617
85	2529	2501	2882	3493	3191	3359	3301	3327
90	2491	2378	2766	3152	3015	3123	3016	3044
95	2315	2174	2522	2756	2646	2727	2727	2756
100	2387	2387	2387	2387	2387	2387	2387	2387

Tableau 5 : Profits pour l'ensemble 5

%	Sans	Base	Amélioré	Bagging	Boosting	Randbag	ForetM 10%	ForetM 20%
5	9848	10743	10743	21845	21094	23116	23400	23898
10	15610	15610	16223	21845	29230	23116	23400	23898
15	18997	18802	19386	21845	30799	23116	23400	23898
20	21333	21061	21411	21845	31494	23116	23400	23898
25	22891	22969	22853	24806	31663	24935	25198	25735
30	23983	24294	24061	26455	31807	25773	26109	26344
35	24840	24918	24802	27144	31625	26914	27032	27520
40	25931	25736	25658	28063	31329	27842	27863	28056
45	26438	26633	26633	28745	30963	28133	28113	28643
50	26712	26907	27413	29098	30808	28837	28832	29277
55	26830	27336	27920	29408	30541	29318	29406	29655
60	26832	27688	28038	29572	30310	29530	29438	29677
65	27222	27961	28117	29522	29781	29519	29497	29606
70	27223	27885	28041	29505	29622	29459	29450	29497
75	27264	27886	27925	29172	29138	29215	29134	29108
80	27226	27771	27771	28810	28641	28757	28773	28856
85	27189	27578	27617	28831	28166	28312	28311	28326
90	26995	27228	27384	27699	27665	27689	27691	27760
95	26763	26841	26918	28120	27067	27107	27118	27112
100	26492	26492	26492	26492	26492	26492	26492	26496

Tableau 6 : Profits pour l'ensemble 6

%	Sans	Base	Amélioré	Bagging	Boosting	Randbag	ForetM 10%	ForetM 20%
5	422	859	1707	1800	8959	1947	2364	2044
10	843	1881	1847	3333	8713	3265	3598	3628
15	1091	2225	2929	4297	8437	4280	4196	4557
20	1217	3669	2929	4760	7784	4720	4531	4689
25	3247	3441	2949	4924	7363	4844	5050	4838
30	3247	3369	2647	5259	6681	5419	5172	5341
35	3032	3360	2515	5302	5971	5318	4706	5178
40	2482	3513	1803	4781	5259	4915	4254	4776
45	2482	3419	1814	4355	4461	4270	3807	4502
50	2313	2902	1814	3807	3691	3702	3327	3632
55	695	2110	1814	3248	2835	3307	2645	2829
60	695	1353	1814	2644	2095	2554	1992	2389
65	263	830	1206	1714	1296	1717	1440	1527
70	-690	171	-615	726	382	671	614	591
75	-1576	-621	-615	-340	-416	-335	-210	-176
80	-1993	-1559	-1665	-1269	-1330	-1249	-1256	-1200
85	-2832	-2252	-1802	-2183	-2189	-2129	-2090	-2184
90	-3392	-3160	-4121	-3194	-3189	-3184	-3036	-3040
95	-4197	-4022	-4121	-4101	-4045	-4223	-3887	-4049
100	-4959	-4959	-4959	-4959	-4959	-4959	-4959	-4959

Tableau 7 : Profits pour l'ensemble 7

%	Sans	Base	Amélioré	Bagging	Boosting	Randbag	ForetM 10%	ForetM 20%
5	-258	3173	3974	4071	10911	3973	4171	4952
10	-258	4834	5173	6151	10383	5914	6355	5816
15	2682	5546	5292	6925	10072	6754	6297	6567
20	3432	6123	5413	7571	9359	7297	7484	7568
25	4030	6150	5718	7871	8586	7601	7133	7689
30	4688	5744	5285	8002	7996	7529	7152	7807
35	3998	5492	4873	7389	7192	7097	6658	7297
40	3030	4969	3889	6605	6232	6696	6140	6415
45	2796	4472	3297	5741	5274	5832	5639	5725
50	2402	4034	2692	5031	4406	5025	4953	4961
55	1981	3258	2256	4062	3725	4161	4156	4149
60	1684	2374	1599	3287	2859	3305	3286	3372
65	1448	1583	1074	2345	2054	2509	2333	2520
70	874	672	380	1547	1280	1556	1474	1573
75	-111	-247	-143	603	783	599	493	490
80	-1065	-850	-543	-464	-206	-461	-362	-468
85	-1825	-1504	-1502	-1308	-1072	-1256	-1319	-1401
90	-2338	-2300	-2219	-2034	-2032	-2118	-2209	-2118
95	-3097	-2899	-3176	-3080	-2899	-3080	-2901	-2988
100	-3950	-3950	-3950	-3950	-3950	-3950	-3950	-3950

Tableau 8 : Profits pour l'ensemble de validation 1

%	Sans	Base	Amélioré	Bagging	Boosting	Randbag	ForetM 10%	ForetM 20%
5	2269	2313	2489	4390	2269	4457	4732	4690
10	2997	3261	3261	4390	3085	4457	4732	4690
15	3593	3505	3681	4390	3417	4457	4732	4690
20	4101	3969	4145	4390	3617	4457	4732	4690
25	4389	4037	4389	4390	4081	4457	4732	4690
30	4457	4281	4457	4635	4325	4457	4732	4690
35	4481	4481	4613	4826	4481	4742	4732	4829
40	4681	4549	4537	4827	4549	4890	4752	4871
45	4661	4705	4661	4833	4749	4764	4865	4841
50	4905	4817	4817	4780	4861	4764	5020	4781
55	4973	4841	4841	4756	4885	4760	4990	4746
60	4866	4778	4778	4822	4910	4813	4898	4916
65	4854	4801	4845	4802	4889	4849	4845	4841
70	4782	4826	4738	4739	4826	4785	4778	4737
75	4805	4805	4761	4722	4761	4719	4675	4717
80	4741	4829	4697	4656	4697	4608	4653	4693
85	4677	4721	4677	4677	4721	4589	4636	4632
90	4613	4657	4657	4570	4657	4568	4613	4613
95	4637	4681	4637	4637	4681	4549	4548	4551
100	4617	4617	4617	4617	4617	4617	4617	4617

Tableau 9 : Profits pour l'ensemble de validation 2

%	Sans	Base	Amélioré	Bagging	Boosting	Randbag	ForetM 10%	ForetM 20%
5	51	143	130	-187	171	186	93	127
10	22	142	142	-187	143	185	208	190
15	73	232	113	-187	74	186	196	222
20	84	243	124	-187	85	165	247	188
25	56	254	95	-187	95	201	185	180
30	27	286	27	-187	67	201	133	140
35	-2	117	38	-187	-1	201	94	153
40	-31	48	-31	-187	9	119	110	135
45	-59	25	-59	-187	20	60	-1	95
50	-128	-9	-86	-187	-88	26	-35	34
55	-157	-38	-37	-187	-116	-39	-54	-43
60	-225	-106	-145	-187	-106	-63	-129	-106
65	-254	-95	-254	-187	-214	-135	-95	-90
70	-323	-164	-323	-245	-243	-164	-141	-205
75	-312	-192	-352	-237	-311	-227	-204	-233
80	-419	-301	-419	-301	-419	-298	-314	-302
85	-448	-409	-488	-369	-488	-367	-336	-370
90	-527	-438	-557	-477	-517	-476	-428	-438
95	-585	-545	-545	-545	-545	-546	-504	-506
100	-614	-614	-614	-614	-614	-614	-614	-614

Annexe B : Codes

Cette section présente les codes qui ont été utilisés pour générer les méthodes ensemblistes. Le premier code est le code permettant de générer un arbre selon la méthodologie CART. Les autres méthodes sont construites à partir de ce code, et quelques changements y ont été faits lorsque nécessaire. Étant donné que la méthode de randomisation a été utilisée sans et avec échantillon bootstrap, mais que seuls les résultats de celle utilisant les échantillons bootstrap ont été présentés dans le présent mémoire, seul ce code sera présenté. Par contre, pour retrouver la méthode de base, il suffit de construire les arbres avec l'échantillon d'entraînement plutôt que les échantillons bootstrap.

Étant donné la longueur considérable des codes, ils ne seront pas présentés dans leur intégralité, à l'exception du code pour la construction d'un arbre. Pour les méthodes ensemblistes, les en-têtes de fonctions et la fonction principale seront présentés, de même que toutes fonctions ayant subi des changements. Mais les fonctions n'ayant subi aucun changement ne seront pas reproduites en entier à nouveau.

Arbre CART

```
/*      Concepteure: Mélanie La Barre
        Date: 15 novembre 2002
Objet: programme d'implantation d'un arbre de classification binaire suivant la
méthodologie CART, dont la fonction d'impureté est Gini et n'utilisant pas d'élagage,
dans la mesure où les feuille ne doivent pas contenir moins de 5% du nombre de données.
L'arbre est bâti avec tout l'ensemble de données, et c'est ce même ensemble qui est utilisé
pour la val validation.
*/

#include <iostream.h>
#include <fstream.h>
#include <iomanip.h>
#include <string.h>
#include <stdlib.h>
#include <time.h>
#include <math.h>

//using namespace std;

struct noeud
        {
                int split[3];     //pour contenir l'indice de la variable et le point de coupure
                                  //le troisième est pour indiquer le traitement des valeurs
                                  //manquantes

                int nobs;         //pour contenir le nombre d'observations dans le noeud
                bool pred;        //pour contenir la classe majoritaire (prédite)
                bool position;    //pour indiquer si c'est un fils gauche ou droit, 0=gauche
                bool feuille;     //pour indiquer si c'est une feuille, 1=oui;
                noeud * filsg;    //pointeur vers le fils gauche
                noeud * filsd;    //pointeur vers le fils droit
        };
int nbarbres=0;

void creer(int**, int, int, int**, int, int**, int, int*);
void principal(int**, int, int**, int, int**, int, int, int, int, double*,ofstream&);
bool prediction(int, int, int**, double*);
int arbre(noeud*, int, int, int**, int, int, int, int, double*);
        int embranchement(noeud*, int, int, int, int**, int, int, int, double*);
                double gini(int , int , int , int , int , int , int, double*);
                void quicksort(int**, int, int);
        int separation(noeud*, int, int, int, int**);
```

```
                void echange(int**, int, int, int, int, int);
                void echange2(int**, int, int, int, int, int);
void prevision(noeud*, int, int**, int**, int);
void Detruire (noeud*);

void write(ofstream&, noeud*);
void writefly(ofstream&, noeud*);

void main()
{
        int narbre=25; // nombre d'arbres à faire

        int n;   //contient le nombre d'enregistrements de l'ensemble initial
        int x;   //contient le nombre de variables

        double C0;//=0.881;   //cout de mal classifier un 0
        double C1;//=44.01;   //cout de mal classifier un 1
        double util=1; //=0 si on ne veut pas utiliser les coûts dans l'embranchement

        char* fichierarbres=new char[];

        char* probabilites=new char[];

        char* classification=new char[];

        classification="classification101.txt";
        //fichier dans lequel les % de bonne classification des arbres seront écrits

        fichierarbres="arbres101.txt";
        //fichier dans lequel les arbres seront écrits

        probabilites="t101.txt";
        //fichier dans lequel les pourcentages seront écrits

        ifstream fiin("test.txt",ios::in);
        fiin >> n;
        fiin >> x;
        fiin >> C0;
        fiin >> C1;

        int choix=x;
        //Création d'un pointeur de pointeur INT
        //Pour obtenir un tableau dynamic a deux dimensions
        int **donnees;
        //Création de la premiere dimension
        donnees = (int **) new int*[n];
```

```
//Création des deuxiemes dimensions
for (int i=0;i<n;i++)
  donnees[i] = (int *) new int[x];

//Lecture dans le fichier des données
for( i=0;i<n;i++)
{
        for(int j=0;j<x;j++)
        {
                fiin >> donnees[i][j];
        }
}
fiin.close();

//si on veut séparer les données en deux ensembles

//int ntrain=n*0.75;
//int ntest=n-ntrain;
int ntrain=n;
int ntest=n;


//ensemble pour contenir l'ensemble d'entraînement
//Dans la mesure où l'ensemble test et l'ensemble d'entraînement sont les mêmes,
//et qu'on ne joue pas dans l'ensemble d'entraînement, on peut s'en passer pour
//ici...

/*int **train;
train = (int **) new int*[ntrain];
for (i=0;i<ntrain;i++)
  train[i] = (int *) new int[x];
*/
//matrice pour contenir l'ensemble test
int **test;
test = (int **) new int*[ntest];
for (i=0;i<ntest;i++)
  test[i] = (int *) new int[x];

//initialisation des ensembles
/*for (int s=0; s<ntrain; s++)
{
        for (int t=0; t<x; t++)
                train[s][t]=donnees[s][t];
}*/

for (int s=0; s<ntest; s++)
```

```
{
        for (int t=0; t<x; t++)
                test[s][t]=donnees[s][t];
}

delete[] donnees;

//creer(donnees, n, x, train, ntrain, test, ntest, tst);

//donnees nécessaires pour le calcul des couts
double N0=0;  //nombre de 0 dans l'ensemble d'entraînement
double N1=0;  //nombre de 1 dans l'ensemble d'entraînement
double N=ntrain;

for (i=0; i<ntrain; i++)
{
        if (test[i][0]==0)
                N0++;
        else
                N1++;
}

double Pi0=(C0*N0/N)/((C0*N0/N)+(C1*N1/N));
double Pi1=(C1*N1/N)/((C0*N0/N)+(C1*N1/N));

double couts[7]={C0,C1,N0,N1,Pi0,Pi1,util};

int min=int(ntrain*0.1);//nombre d'obs min qu'il faut dans une feuille pour séparer

//ensemble pour contenir les prévisions à chaque arbre
int **prev;
prev = (int **) new int*[ntest];
for (i=0;i<ntest;i++)
  prev[i] = (int *) new int[narbre];

//initialisation
for (s=0; s<ntest; s++)
{
        for (int t=0; t<narbre; t++)
                prev[s][t]=-1;
}

//création d'un ensemble pour contenir l'échantillon bootstrap

//tableau pour contenir le % de 1
double *pour=new double[ntest];
```

```
ofstream fopen(fichierarbres);
ofstream open(classification);

srand(time(NULL));
for (int iter=0; iter<narbre; iter++)
{
        cout << "arbre no: " << iter << endl;

        principal(test, ntrain, test, ntest, prev, iter, min, x, choix, couts, fopen);

        // calcul du taux de bonne pred
        double correct=0;

        double nb=ntest;
        for (int t=0; t<ntest; t++)
        {
                if (test[t][0]==prev[t][iter])
                        correct++;
        }

        double taux=correct/nb*100;
        open << taux << endl;
}

fopen.close();
open.close();

double nbuns=0;
double nb=narbre;

for( i=0;i<ntest;i++)
{
        for(int j=0;j<narbre;j++)
        {
                if (prev[i][j]==1)
                        nbuns++;
        }
        pour[i]=nbuns/nb*100;
        nbuns=0;
}

ofstream fout(probabilites);

for( i=0;i<ntest;i++)
{
```

```
                fout << pour[i];
                fout << endl;
        }
        fout.close();
}

//fonction qui fait le travail principal
void principal(int **train, int ntrain, int **test, int ntest, int **prev, int iter, int min, int x,
int choix, double *couts, ofstream &output)
{
        //définition de l'intervalle de départ
        int debut=0;
        int fin=ntrain-1;

        //création du premier élément
        noeud* racine;//pointeur qui indiquera la position de la racine
        noeud* courant;         //pointeur qui pointera toujours sur le dernier élément

        courant=new noeud;   //création du noeud
        racine = courant;

        nbarbres++;
        //on peut mettre dans le premier noeud les informations qu'on connaît

        courant->nobs=ntrain;
        courant->pred=prediction(debut, fin, train, couts);
        courant->position=0;  //de peu d'importante, le premier noeud écrit est la racine
        courant->feuille=0;

        arbre(courant, 0, ntrain-1, train, min, x, ntrain, choix, couts);

        //reste ensuite à prédire les valeurs pour l'ensemble test, que l'on mettra dans prev

        prevision(racine, ntest, prev, test, iter);

        write(output, racine);
        output << "fin" << endl ;

        Detruire(racine);
        courant=NULL;
}

/* fonction qui retourne la classe majoritaire des données contenues dans le noeud
courant; Compte donc, dans l'intervalle définit par debut et fin, le nombre de 0 et le
nombre de 1 de la variable dépendante, et retourne la classe la classe qui minimise le coût
de mauvaise classification*/
```

```
bool prediction(int debut, int fin, int **tab, double *couts)
{
	int nb0=0;
	int nb1=0;

	for (int i=debut; i<=fin; i++)
	{
		if (tab[i][0]==0)
			nb0++;
		else
			nb1++;
	}

	double C0=couts[0];
	double C1=couts[1];

	double N0=couts[2];
	double N1=couts[3];
	double Pi0=couts[4];
	double Pi1=couts[5];

	double p0=Pi0*nb0/N0;		//p(0,t)
	double p1=Pi1*nb1/N1;		//p(1,t)
	double p=p0+p1;			//p(t)

	double p0st=p0/p;		//p(0|t)
	double p1st=p1/p;		//p(1|t)

	double i0=C1*p1st;		//coût de classifier en 0
	double i1=C0*p0st;		//coût de classifier en 1

	if (i0<i1)
		return 0;
	else
		return 1;

}
```

/*Fonction récursive qui construit l'arbre. Prend en paramètre l'intervalle définissant le sous-ensemble de données à classifier et le pointeur pointant vers le noeud courant, met les pointeurs du noeud courant à NULL s'il n'est pas possible de construire deux nouvelle feuilles, et retourne 0. S'il est encore possible de construire, les deux nouvelles feuilles sont construites. Finalement, min est le nombre minimum qu'il faut dans un noeud pour pouvoir splitter
*/

```
int arbre(noeud *courant, int debut, int fin, int **tab, int min, int x, int ntrain, int choix,
double *couts)
{
        //il faut d'abord vérifier si on a assez d'obs pour diviser

        if(!embranchement(courant, debut, fin, x, tab, min, ntrain, choix, couts))
                {
                        return 0;
                }
                //il faut ensuite réparer l'ensemble de données selon le nouveau split
                //et retourner l'indice de séparation

                int sep=separation(courant, debut, fin, x, tab);
                //sep est l'indice du dernier el < au point de coupure

                //on connait une information des nouveaux noeuds:

                        noeud* nouveaug;
//pointeur qui pointera sur les éléments à ajouter à gauche
                        noeud* nouveaud;
//pointeur qui pointera sur les éléments à ajouter à droite

                        nouveaug=new noeud;
                        nouveaud=new noeud;
                        nbarbres++;
                        nbarbres++;

                        nouveaug->nobs=sep-debut+1;
                        nouveaud->nobs=fin-sep;

                        nouveaug->position=0;       //indique que c'est un fils de gauche
                        nouveaud->position=1;       //indique que c'est un fils de droite

                        nouveaug->feuille=0;        //initialisé à 0: n'est pas une feuille
                        nouveaud->feuille=0;

                        nouveaug->pred=prediction(debut, sep, tab, couts);
                        nouveaud->pred=prediction(sep+1, fin, tab, couts);

                        courant->filsg=nouveaug;
                        courant->filsd=nouveaud;

                        nouveaug=NULL;
                        nouveaud=NULL;

                        arbre(courant->filsg, debut, sep, tab, min, x, ntrain, choix, couts);
```

```
                    arbre(courant->filsd, sep+1, fin, tab, min, x, ntrain, choix, couts);

                    return 1;
}

/*fonction qui trouve le meilleur embranchement possible selon l'indice de Gini. Inscrit
dans le noeud courant l'indice de la variable ainsi que le point de coupure.
*/

int embranchement(noeud *courant, int debut, int fin, int x, int **tab, int min, int ntrain,
int choix, double *couts)
{
        int nbpere=courant->nobs;     //nombre d'obs dans le noeud

        double GiniMin[3]={100,0,-1};
        //contient l'indice min et son point de coupure pour une variable
        /doit être réinitialisé lors du test pour chaque nouvelle variable
        //le troisième indique le traitement des valeurs manquantes:
        //-1=>pas de vm, 0=>vm à gauche, 1=>vm à droite

        bool ok=0;



        double **index;
        index = (double **) new double*[4];
        for (int i=0;i<4;i++)
                index[i] = (double *) new double[choix];
        //contient en première ligne l'indice min pour chaque variable,
        // et en seconde le point de coupure correspondant.
        //en troisième, l'indice de valeur manquante et en dernier l'indice de la variable

        for ( i=0; i<4; i++)
        {
                for (int j=0; j<choix; j++)
                        index[i][j]=100;
        }

        //pour vérifier si la variable a déjà été utilisée...

        int* ch= new int[x];
        for (int h=0; h<x; h++)
                ch[h]=0;

        //Traitement pour chaque variable
```

```
int p=0;                //point de coupure courant
double ginmin=100;  //indice Gini minimum du moment

int indicevar=0;

//Il faut d'abord vérifier si on a assez d'observations...
if (courant->nobs>=min)
{

        //il faut ensuite vérifier si le noeud est pur...
        bool pur=1;
        for (i=debut; i<fin; i++)
                {
                        if (tab[i][0]!=tab[i+1][0])      //alors le noeud n'est pas pur
                                pur=0;
                }

        if (pur==0)
        {
                //pour chaque variable...
                for(int v=0; v<x; v++)
                {
                        int var=v;
                        index[3][v]=v;

                        int dernier=0;
                        //indice du dernier élément dans le tableau
                        int *coup=new int[nbpere];
                        //pour mettre les différentes valeurs
                        for (i=0; i<nbpere; i++)
                                coup[i]=-1;

                        //il faut d'abord trouver tous les points de coupure possible

                        coup[0]=tab[debut][var];
                        bool present=0;

                        for (i=debut+1; i<=fin; i++)
                        {
                        //on veut regarder si la valeur contenue dans tab[i][v]
                        //appartient à coup
                                int j=0;
                                while (coup[j]<tab[i][var] && coup[j]!=-1)
                                        j++;

                                if (tab[i][var]!=coup[j])
```

```
                    //alors il faut l'ajouter à j-1
                    {
                            if (tab[i][var]>coup[j])
                            {
                                    coup[dernier+1]=tab[i][var];
                            }
                            else
                            {
                                    for (int k=dernier; k>=j; k--)
                                            coup[k+1]=coup[k];
                                            //on incrémente les éléments
                                            //de 1 position
                                    coup[j]=tab[i][var];
                            }
                            dernier++;
                    }
                    j=0;

            }

            if (dernier!=0)
            {
/*on a plus besoin d'un tableau de longueur nbpere: on peut transférer les observations
dans un tableau plus petit, qui en première ligne contiendra les valeurs des points de
coupure, en seconde ligne le nombre de 0 et en troisième ligne le nombre de 1 associé à
chaque point de coupure */

                    int **points;

                    points = (int **) new int*[3];
                    for (i=0;i<3;i++)
                            points[i] = (int *) new int[dernier+1];

                    for (int u=0; u<=dernier; u++)
                    {
                            points[0][u]=coup[u];
                            points[1][u]=0;
                            points[2][u]=0;
                    }

                    delete []coup;

                    //il faut ensuite trouver le nombre de 0 et 1 associés
                    //à chaque point
```

```
int j=0;
for (int i=debut; i<=fin; i++)
{
        j=0;
        if(tab[i][var]!=points[0][j])
        {
        while (tab[i][var]!=points[0][j]
                        && j<dernier+1)
                {
                        j++;
                }
        }
        if(tab[i][0]==0)
                points[1][j]++;
        else
                points[2][j]++;
}

//si on a des valeurs manquantes, points[0][0]=-100
int indicep=0;
index[2][v]=-1;
if(points[0][0]==-100)
{
        index[2][v]=1;
        indicep=1;
}
indicep++;      //parce que la première valeur n'est
                //pas candidate à cause du >=

int p;
for (int ip=indicep; ip<=dernier; ip++)
//pour chaque valeur possible du point de coupure
{
        //point de coupure du moment
        p=points[0][ip];

        int g0=0;
        int g1=0;
        int d0=0;
        int d1=0;

        for (int i=indicep-1; i<ip; i++)
        {
                g0=g0+points[1][i];
                g1=g1+points[2][i];
        }
```

```
for (i=ip; i<=dernier; i++)
{
        d0=d0+points[1][i];
        d1=d1+points[2][i];
}

//S'il n'y a pas de vm...
if (index[2][v]==-1)
{
        int g=g0+g1;
        int d=d0+d1;

        //on a un split, mais il faut vérifier si
        //on a plus de 5% de chaque côté...

        if (g>=min/2 && d>=min/2)
        {
                ok=1;

        double igini=gini(g, d, g0, g1, d0,
                        d1, nbpere, couts);
                if (igini < GiniMin[0])
                {
                        GiniMin[0]=igini;
                        GiniMin[1]=p;
                }
        }
}
else
{
        //valeurs manquantes à gauche
        int g=g0+g1+points[1][0]
                        +points[2][0];
        int d=d0+d1;

        //on a un split, mais il faut vérifier si
        //on a plus de 5% de chaque côté...

        if (g>=min/2 && d>=min/2)
        {
                ok=1;
                int tg0=g0+points[1][0];
                int tg1=g1+points[2][0];
        double igini=gini(g, d, tg0, tg1, d0,
                        d1, nbpere, couts);
```

```
                    if (igini < GiniMin[0])
                    {
                        GiniMin[0]=igini;
                        GiniMin[1]=p;
                        GiniMin[2]=0;
                    }
                }
                //valeurs manquantes à droite
                g=g0+g1;
                d=d0+d1+points[1][0]+points[2][0];

                //on a un split, mais il faut vérifier si
                //on a plus de 5% de chaque côté...

                if (g>=min/2 && d>=min/2)
                {
                    ok=1;
                    int td0=d0+points[1][0];
                    int td1=d1+points[2][0];
                    double igini=gini(g, d, g0,
                     g1, td0, td1, nbpere, couts);

                    if (igini < GiniMin[0])
                    {
                        GiniMin[0]=igini;
                        GiniMin[1]=p;
                        GiniMin[2]=1;
                    }
                }
            }

}//fin du for pour les p

//on a trouvé les indices pour tous les p, reste à
//regarder, s'il y a des valeurs manquantes, si les
//mettre toutes à gauche

if (index[2][v]!=-1)    //alors on a des vm
{

    int nb0=0;
    int nb1=1;

    for (int i=1; i<=dernier; i++)
    {
        nb0=nb0+points[1][i];
```

```
                        nb1=nb1+points[2][i];
                }
                int miss0=points[1][0];
                int miss1=points[2][0];

                int g=points[1][0]+points[2][0];
                int d=nb0+nb1;

                if (g>=min/2 && d>=min/2)
                {
                        ok=1;
                        double igini=gini(g, d, points[1][0],
                        points[2][0], nb0, nb1, nbpere,
                                                        couts);

                        if (igini < GiniMin[0])
                        {
                                GiniMin[0]=igini;
                                GiniMin[1]=0;
                                //0 indique une sep par vm
                                GiniMin[2]=2;
                                //2 indique seulement vm à
                                //gauche
                        }
                }
        }

        index[0][v]=GiniMin[0];
        index[1][v]=GiniMin[1];
        index[2][v]=GiniMin[2];
        GiniMin[0]=100;
        GiniMin[1]=0;
        GiniMin[2]=-1;
        p=0;
    }
}

// reste à trouver le minimum dans index...
for (v=0; v<choix; v++)
{
    if(index[0][v]<GiniMin[0])
    {
        GiniMin[0]=index[0][v];
        GiniMin[1]=index[1][v];
        GiniMin[2]=index[2][v];
        indicevar=v;
```

```
                        }
                    }
                }
            }

            if (ok==0)
            {
                //alors il n'y a pas eu de split trouvé
                courant->filsd=NULL;
                courant->filsg=NULL;
                courant->split[0]=-1;  //pas de variable
                courant->split[1]=-1;  //pas de point de coupure
                courant->split[2]=-1;  //pas de valeur manquante
                courant->feuille=1;              //c'est une feuille
                return 0;
            }
            else
            {
                courant->split[0]=indicevar;
                courant->split[1]=int(GiniMin[1]);
                courant->split[2]=int(GiniMin[2]);
            }

            delete []ch;
            delete []index;
            return 1;
}

void quicksort(int **tab,int gauche,int droite)
{

    int g,d,pivot,x,y;

    //Si le tableau comporte plus d'un élément.
    if(gauche<droite)
    {
        //On sélectionne l'élément du premier espace du tableau comme pivot.
        pivot = tab[0][(int)((gauche+droite)/2-0.5)];

        //Initialisation des variables g et d qui changeront de valeur dans
        //l'algorithme. gauche et droite garderont toujours la même valeur
        //dans l'algorithme
        g=gauche;
        d=droite;

        //Tant que g et d ne se sont pas croisés, on boucle
```

```
		while(g<=d)
		{
			//Tant que l'élément en position g du tableau est plus petit que
			//le pivot ET que l'indice g n'est pas au dernier élément du
			//tableau : on passe à l'élément suivant du tableau (incrément
			//de l'indice g).
			while((g<droite) && (tab[0][g]<pivot))
			{
				g++;
			}

			//Tant que l'élément en position d du tableau est plus grand que
			//le pivot ET que l'indice d est plus grand que l'indice gauche
			//on passe à l'élément suivant du tableau (décrément de l'indice d).
			while((d>gauche) && (tab[0][d]>pivot))
			{
				d--;
			}
			//Si l'indice g est plus petit que l'indice d après le passage
			//dans les deux boucles Tant Que, alors les deux éléments
			//(tab[g] et tab[d]) doivent être échangés.
			//On sélectionne l'indice suivant du tableau pour g et d (incrément
			//et décrément de 1 respectivement).
			if (g<=d)
			{
				x=tab[0][g];
				y=tab[1][g];
				tab[0][g]=tab[0][d];
				tab[1][g]=tab[1][d];
				tab[0][d]=x;
				tab[1][d]=y;
				g++;
				d--;
			}
		}

		//On exécute le quicksort récursivement sur les parties gauche
		//et droite du tableau.
		quicksort(tab,gauche,d);
		quicksort(tab,g,droite);
	}
}

double gini(int g, int d, int g0, int g1, int d0, int d1, int nbpere, double *couts)
{
	double C0=couts[0];
```

```
double C1=couts[1];
double util=couts[6];

double tg=g;
double td=d;
double tg0=g0;
double tg1=g1;
double td0=d0;
double td1=d1;
double tnbpere=nbpere;

double igini=0;

if (util==0 || (C0==1 && C1==1))
{

igini=(((tg/tnbpere)*2*(tg0/tg)*(tg1/tg))+((td/tnbpere)*2*(td0/td)*(td1/td)));
}
else
{
        double N0=couts[2];
        double N1=couts[3];
        double Pi0=couts[4];
        double Pi1=couts[5];

        double p0g=Pi0*tg0/N0;      //p(0,t) au noeud gauche
        double p1g=Pi1*tg1/N1;      //p(1,t) au noeud gauche
        double pg=p0g+p1g;          //p(t) au noeud gauche

        double p0d=Pi0*td0/N0;      //p(0,t) au noeud droit
        double p1d=Pi1*td1/N1;      //p(1,t) au noeud droit
        double pd=p0d+p1d;          //p(t) au noeud droit

        double p0sg=p0g/pg;         //p(0|t) au noeud gauche
        double p1sg=p1g/pg;         //p(1|t) au noeud gauche

        double p0sd=p0d/pd;         //p(0|t) au noeud droit
        double p1sd=p1d/pd;         //p(1|t) au noeud droit

        igini=(tg/tnbpere)*p0sg*p1sg+(td/tnbpere)*p0sd*p1sd;

        }
return igini;

}
```

```
int separation(noeud *courant, int debut,int fin, int x, int **tab)
{
//fonction qui sépare le sous-ensemble de données selon le point de coupure et la variable
//du noeud courant.Tous les enregistrements plus grands ou égaux au point de coupure
//seront en bas de l'indice de séparation, et tous les autres en haut.
//On retourne l'indice du dernier enregistrement plus petit au point de coupure

//si le point de coupure=0, c'est dire que ce sont les vm qui définissent la séparation

        int var=courant->split[0];
        int coupure=courant->split[1];
        int vm=courant->split[2];

        int k=debut;
        if (vm!=1)
        {
                echange(tab, debut, fin, var, x, coupure);
                while(tab[k][var]<coupure)
                        k++;
        }
        else
        {
                echange2(tab, debut, fin, var, x, coupure);
                while(tab[k][var]<coupure && tab[k][var]!=-100 )
                        k++;
        }
        k--;
        return k;
}


//fonction qui écrit dans un tableau prev les valeurs prédites pour l'ensemble test
void prevision(noeud *racine, int ntest, int **prev, int **test, int iter)
{
        noeud* pointeur;        //pointeur qui permet de suivre le chemin

        pointeur=new noeud;  //création du noeud
        pointeur=racine;

        for (int i=0; i<ntest; i++)
        {
                pointeur=racine;
                while(pointeur->filsd!=NULL && pointeur->filsg!=NULL)
                {
                        int var=pointeur->split[0];
                        int coupure=pointeur->split[1];
```

```
                int miss=pointeur->split[2];

                if(miss==-1)
                {
                        if (test[i][var]<coupure)
                                pointeur=pointeur->filsg;
                        else
                                pointeur=pointeur->filsd;
                }
                else
                {

                        if (miss==2)    //les vm à gauche, le reste à droite
                        {
                                if (test[i][var]==-100)
                                        pointeur=pointeur->filsg;
                                else
                                        pointeur=pointeur->filsd;
                        }
                        else
                        {
                                //on a deux choix: on a une vm ou pas
                                if (test[i][var]!=-100)         //pas de vm
                                {
                                        if (test[i][var]<coupure)
                                                pointeur=pointeur->filsg;
                                        else
                                                pointeur=pointeur->filsd;
                                }
                                else
                                {
                                        if (miss==0)
                                                pointeur=pointeur->filsg;
                                        else
                                                pointeur=pointeur->filsd;
                                }
                        }
                }
        }
        prev[i][iter]=pointeur->pred;
    }
    pointeur=NULL;
}

void echange(int **tab, int debut, int fin, int var, int x, int coupure)
{
```

```
//fonction qui, pour une variable donnée, entre les observations délimitées par
//début et fin, fait en sorte que toutes les données inférieures au point de coupure
//se retrouvent en haut, et toutes celles supérieures ou égales en bas.

        int i=debut;
        int j=fin;

        int* temp= new int[x];

        //d'abord, trouver le premier élément supérieur ou égal en partant du haut
        if(tab[i][var]<coupure)
        {
                while (i<=fin && tab[i][var]<coupure)
                        i++;
        }
        //et trouver le premier élément inférieur en partant du bas
        if(tab[j][var]>=coupure)
        {
                while (j>=debut && tab[j][var]>=coupure)
                        j--;
        }

        while (i<j)
        {
                //reste à les échanger

                for (int t=0; t<x; t++)
                {
                        temp[t]=tab[i][t];
                        tab[i][t]=tab[j][t];
                        tab[j][t]=temp[t];
                }
                i++;
                j--;

                //puis trouver les prochains
                if(tab[i][var]<coupure)
                {
                        while (i<=fin && tab[i][var]<coupure)
                                i++;
                }
                if(tab[j][var]>=coupure)
                {
                        while (j>=debut && tab[j][var]>=coupure)
                                j--;
                }
```

```
        }
        delete[]temp;
}

void echange2(int **tab, int debut, int fin, int var, int x, int coupure)
{
//fonction qui, pour une variable donnée, entre les observations délimitées par
//début et fin, fait en sorte que toutes les données inférieures au point de coupure
 //se retrouvent en haut, et toutes celles supérieures ou égales en bas.

        int i=debut;
        int j=fin;

        int* temp= new int[x];

        //d'abord, trouver le premier élément supérieur ou égal en partant du haut
        if(tab[i][var]<coupure && tab[i][var]!=-100 )
        {
                while (i<=fin && tab[i][var]<coupure && tab[i][var]!=-100)
                        i++;
        }
        //et trouver le premier élément inférieur en partant du bas
        if(tab[j][var]>=coupure || tab[j][var]==-100)
        {
                while (j>=debut &&(tab[j][var]>=coupure || tab[j][var]==-100))
                        j--;
        }

        while (i<j)
        {
                //reste à les échanger

                for (int t=0; t<x; t++)
                {
                        temp[t]=tab[i][t];
                        tab[i][t]=tab[j][t];
                        tab[j][t]=temp[t];
                }
                i++;
                j--;

                //puis trouver les prochains
                if(tab[i][var]<coupure && tab[i][var]!=-100 )
                {
                        while (i<=fin && tab[i][var]<coupure && tab[i][var]!=-100 )
                                i++;
```

```
                }
                if(tab[j][var]>=coupure || tab[j][var]==-100)
                {
                        while (j>=debut && (tab[j][var]>=coupure || tab[j][var]==-100))
                                j--;
                }
        }
        delete[]temp;
}

void Detruire (noeud *arbre)
{
        if (arbre->filsd != NULL)
                Detruire(arbre->filsd);
        if(arbre->filsg!=NULL)
                Detruire(arbre->filsg);
        delete arbre;
        nbarbres--;

}

void write(ofstream &output, noeud *arbre)
{
//Ecrit l'arbre a partir de le feuille courante.

output << arbre->feuille << " " << arbre->position << " " << arbre->split[0]
 << " " << arbre->split[1] << " " << arbre->split[2] << " "
<< arbre->nobs << " " << arbre ->pred<< endl;

        if(arbre->filsg != NULL)
                writefly(output, arbre->filsg);
        if(arbre->filsd!= NULL)
                writefly(output, arbre->filsd);
}

void writefly(ofstream &output, noeud *arbre)
{
output << arbre->feuille << " " << arbre->position << " " << arbre->split[0]
<< " " << arbre->split[1] << " " << arbre->split[2] << " " << arbre->nobs << " " << arbre
->pred<< endl;

        if(arbre->filsg!= NULL)
                writefly(output, arbre->filsg);
        if(arbre->filsd!= NULL)
                writefly(output, arbre->filsd);
}
```

```
//fonction qui crée les ensembles train et test
void creer(int **donnees, int n, int x, int **train, int ntrain, int **test, int ntest, int *tst)
{
	//copie de la matrice donnees dans une matrice temp pour pouvoir la modifier
	int **temp;
	temp = (int **) new int*[n];
	//Creation des deuxiemes dimensions
	for (int i=0;i<n;i++)
	 temp[i] = (int *) new int[x];

	for( i=0;i<n;i++)
	{
		for(int j=0;j<x;j++)
		{
			temp[i][j]=donnees[i][j];
		}
	}
	srand(time(NULL));

	int aleat;

	for ( i=0; i<ntrain; i++)
	{
		aleat=rand()%n;

		//si on a déjà copié cette ligne, on passe à la suivante.
		if (temp[aleat][0]==-1)
		{
			while(temp[aleat][0]==-1)
			{
				aleat=(aleat+1)%ntrain;
			}
		}
		//copie de la ligne
		for (int k=0; k<x; k++)
		{
			train[i][k]=temp[aleat][k];
			temp[aleat][k]=-1;
		}
	}

	//création de l'ensemble test
	int trace=0;
	int h=0;
```

```
        while(trace<ntest)
        {
                if(temp[h][0]!=-1)
                {
                        for (int k=0; k<x; k++)
                        {
                                test[trace][k]=temp[h][k];
                                temp[h][k]=-1;
                                tst[trace]=h;
                        }
                        trace++;
                }
                h++;
        }
}
```

Bagging

Rappelons-nous que le bagging consiste à construire plusieurs arbres à partir d'échantillons bootstrap différents; seul l'échantillon présenté à l'arbre diffère de la construction d'un simple arbre CART. Ainsi, seulement la fonction principale se trouve légèrement modifiée, aucune des autres fonctions n'ayant besoin de subir de changement. Nous présenterons donc seulement cette fonction.

```
void creer(int**, int, int, int**, int, int**, int, int*);
void principal(int**, int, int**, int, int**, int, int, int, int, double*,ofstream&);
bool prediction(int, int, int**, double*);
int arbre(noeud*, int, int, int**, int, int, int, int, double*);
        int embranchement(noeud*, int, int, int, int**, int, int, int, double*);
                double gini(int , int , int , int , int , int , int, double*);
                void quicksort(int**, int, int);
        int separation(noeud*, int, int, int, int**);
                void echange(int**, int, int, int, int, int);
                void echange2(int**, int, int, int, int, int);
void prevision(noeud*, int, int**, int**, int);
void Detruire (noeud*);

void write(ofstream&, noeud*);
void writefly(ofstream&, noeud*);

void main()
{
        int narbre=100; // nombre d'arbres à construire

        int n;   //contient le nombre d'enregistrements de l'ensemble initial
        int x;   //contient le nombre de variables

        double C0;//=0.881;   //coût de mal classifier un 0
        double C1;//=44.01;   //coût de mal classifier un 1
        double util=1; //=0 si on ne veut pas utiliser les coûts dans l'embranchement

        char* fichierarbres=new char[];

        char* probabilites=new char[];
```

```
char* classification=new char[];

classification="classification101.txt";
//fichier dans lequel les % de bonne class des arbres seront écrits

fichierarbres="arbres101.txt";
//fichier dans lequel les arbres seront écrits

probabilites="t101.txt";
//fichier dans lequel les pourcentages seront écrits

ifstream fiin("test.txt",ios::in);
fiin >> n;
fiin >> x;
fiin >> C0;
fiin >> C1;

int choix=x;

//Creation d'un pointeur de pointeur INT

int **donnees;                          //Creation de la premiere dimension
donnees = (int **) new int*[n];         //Creation des deuxiemes dimensions
for (int i=0;i<n;i++)
  donnees[i] = (int *) new int[x];

//Lecture dans le fichier test.txt des donnees
for( i=0;i<n;i++)
{
        for(int j=0;j<x;j++)
        {
                fiin >> donnees[i][j];
        }
}
fiin.close();

//int ntrain=n*0.75;    //si séparation en deux ensembles
//int ntest=n-ntrain;   //si séparation en deux ensembles
int ntrain=n;
int ntest=n;


//ensemble pour contenir l'ensemble d'entraînement
//Dans la mesure où l'ensemble test et l'ensemble d'entraînement sont les mêmes,
//et qu'on ne joue pas dans l'ensemble d'entraînement, on peut s'en passer pour
//ici...
```

```
/*int **train;
train = (int **) new int*[ntrain];
for (i=0;i<ntrain;i++)
  train[i] = (int *) new int[x];
*/

//matrice pour contenir l'ensemble test
int **test;
test = (int **) new int*[ntest];
for (i=0;i<ntest;i++)
  test[i] = (int *) new int[x];

//initialisation des ensembles
/*for (int s=0; s<ntrain; s++)
{
        for (int t=0; t<x; t++)
                train[s][t]=donnees[s][t];
}*/

for (int s=0; s<ntest; s++)
{
        for (int t=0; t<x; t++)
                test[s][t]=donnees[s][t];
}
delete[] donnees;

//creer(donnees, n, x, train, ntrain, test, ntest, tst);

//donnees nécessaires pour le calcul des coûts

double N0=0;  //nombre de 0 dans l'ensemble d'entraînement
double N1=0;  //nombre de 1 dans l'ensemble d'entraînement
double N=ntrain;
for (i=0; i<ntrain; i++)
{
        if (test[i][0]==0)
                N0++;
        else
                N1++;
}

double Pi0=(C0*N0/N)/((C0*N0/N)+(C1*N1/N));
double Pi1=(C1*N1/N)/((C0*N0/N)+(C1*N1/N));

double couts[7]={C0,C1,N0,N1,Pi0,Pi1,util};
```

```
int min=int(ntrain*0.1);
//nombre d'obs min qu'il faut dans une feuille pour séparer

//ensemble pour contenir les prévisions à chaque arbre
int **prev;
prev = (int **) new int*[ntest];
for (i=0;i<ntest;i++)
  prev[i] = (int *) new int[narbre];

//initialisation
for (s=0; s<ntest; s++)
{
        for (int t=0; t<narbre; t++)
                prev[s][t]=-1;
}

double* poidscum= new double[ntrain];       //pour contenir des poids cumulés

//création d'un ensemble pour contenir l'échantillon bootstrap
int **bag;
bag = (int **) new int*[ntrain];
for (i=0;i<ntrain;i++)
        bag[i] = (int *) new int[x];


//tableau pour contenir le % de 1
double *pour=new double[ntest];

ofstream fopen(fichierarbres);
ofstream open(classification);

srand(time(NULL));
for (int iter=0; iter<narbre; iter++)
{
        cout << "arbre no: " << iter << endl;

        //création de l'ensemble d'entraînement

        for (i=0; i<ntrain; i++)
        {
                int j=0;
                aleat=rand()/max;
                if (aleat==1)
                {
                        while(aleat==1)
                                aleat=rand()/max;
```

```
                }
                else
                {

                        while (poidscum[j]<aleat)
                {
                                j++;
                }
        }

                //copie de la ligne
                for (int k=0; k<x; k++)
                {
                        bag[i][k]=test[j][k];
                }
        }

        principal(bag, ntrain, test, ntest, prev, iter, min, x, choix, couts, fopen);

        double correct=0;

        double nb=ntest;
        for (int t=0; t<ntest; t++)
        {
                if (test[t][0]==prev[t][iter])
                        correct++;
        }


        double taux=correct/nb*100;
        open << taux << endl;
}

fopen.close();
open.close();

double nbuns=0;
double nb=narbre;

for( i=0;i<ntest;i++)
{
        for(int j=0;j<narbre;j++)
        {
                if (prev[i][j]==1)
                        nbuns++;
        }
```

```
            pour[i]=nbuns/nb*100;
            nbuns=0;
        }

        ofstream fout(probabilites);

        for( i=0;i<ntest;i++)
        {
            fout << pour[i];
            fout << endl;
        }
        fout.close();
}
```

Boosting

Le boosting ressemble au bagging, à une différence près, le calcul des poids de chaque observation, et la construction des échantillons bootstrap selon ces poids. Ainsi, encore une fois, seulement la fonction principale se trouve légèrement modifiée, aucune des autres fonctions n'ayant besoin de subir de changement. Nous présenterons donc seulement cette fonction.

```
void creer(int**, int, int, int**, int, int**, int, int*);
void principal(int**, int, int**, int, int**, int, int, int, int, double*,ofstream&);
bool prediction(int, int, int**, double*);
int arbre(noeud*, int, int, int**, int, int, int, int, double*);
        int embranchement(noeud*, int, int, int, int**, int, int, int, double*);
                double gini(int , int , int , int , int , int , int, double*);
                void quicksort(int**, int, int);
        int separation(noeud*, int, int, int, int**);
                void echange(int**, int, int, int, int, int);
                void echange2(int**, int, int, int, int, int);
void prevision(noeud*, int, int**, int**, int);
void Detruire (noeud*);

void write(ofstream&, noeud*);
void writefly(ofstream&, noeud*);

void main()
{
        int narbre=100; // nombre d'arbres à construire

        int n;   //contient le nombre d'enregistrements de l'ensemble initial
        int x;   //contient le nombre de variables

        double C0;//=0.881;   //coût de mal classifier un 0
        double C1;//=44.01;   //coût de mal classifier un 1
        double util=1; //=0 si on ne veut pas utiliser les coûts dans l'embranchement

        char* fichierarbres=new char[];

        char* probabilites=new char[];
```

```
char* classification=new char[];

classification="classification101.txt";
//fichier dans lequel les % de bonne class des arbres seront écrits

fichierarbres="arbres101.txt";
//fichier dans lequel les arbres seront écrits

probabilites="t101.txt";
//fichier dans lequel les pourcentages seront écrits

ifstream fiin("test.txt",ios::in);
fiin >> n;
fiin >> x;
fiin >> C0;
fiin >> C1;

int choix=x;

//Creation d'un pointeur de pointeur INT
//Pour obtenir un tableau dynamic a deux dimensions
int **donnees;
//Creation de la premiere dimension
donnees = (int **) new int*[n];
//Creation des deuxiemes dimensions
for (int i=0;i<n;i++)
  donnees[i] = (int *) new int[x];

//Lecture dans le fichier test.txt des donnees
for( i=0;i<n;i++)
{
        for(int j=0;j<x;j++)
        {
                fiin >> donnees[i][j];
        }
}
fiin.close();

//int ntrain=n*0.75;    //si séparation en deux ensembles
//int ntest=n-ntrain;   //si séparation en deux ensembles
int ntrain=n;
int ntest=n;


//ensemble pour contenir l'ensemble d'entraînement
//Dans la mesure où l'ensemble test et l'ensemble d'entraînement sont les mêmes,
```

```
//et qu'on ne joue pas dans l'ensemble d'entraînement, on peut s'en passer pour
//ici...
/*int **train;
train = (int **) new int*[ntrain];
for (i=0;i<ntrain;i++)
  train[i] = (int *) new int[x];
*/

//matrice pour contenir l'ensemble test
int **test;
test = (int **) new int*[ntest];
for (i=0;i<ntest;i++)
  test[i] = (int *) new int[x];

//initialisation des ensembles
/*for (int s=0; s<ntrain; s++)
{
        for (int t=0; t<x; t++)
                train[s][t]=donnees[s][t];
}*/

for (int s=0; s<ntest; s++)
{
        for (int t=0; t<x; t++)
                test[s][t]=donnees[s][t];
}
delete[] donnees;

//creer(donnees, n, x, train, ntrain, test, ntest, tst);

//donnees nécessaires pour le calcul des coûts

double N0=0;  //nombre de 0 dans l'ensemble d'entraînement
double N1=0;  //nombre de 1 dans l'ensemble d'entraînement
double N=ntrain;
for (i=0; i<ntrain; i++)
{
        if (test[i][0]==0)
                N0++;
        else
                N1++;
}

double Pi0=(C0*N0/N)/((C0*N0/N)+(C1*N1/N));
double Pi1=(C1*N1/N)/((C0*N0/N)+(C1*N1/N));
```

```
double couts[7]={C0,C1,N0,N1,Pi0,Pi1,util};

int min=int(ntrain*0.1);
//nombre d'obs min qu'il faut dans une feuille pour séparer

//ensemble pour contenir les prévisions à chaque arbre
int **prev;
prev = (int **) new int*[ntest];
for (i=0;i<ntest;i++)
  prev[i] = (int *) new int[narbre];

//initialisation
for (s=0; s<ntest; s++)
{
        for (int t=0; t<narbre; t++)
                prev[s][t]=-1;
}
double* poids= new double[ntrain];  //pour contenir des poids des observations
double NB=ntrain;

for (i=0; i<ntrain; i++)
                poids[i]=1/NB;

double* poidscum= new double[ntrain];       //pour contenir des poids cumulés

//création d'un ensemble pour contenir l'échantillon bootstrap
int **bag;
bag = (int **) new int*[ntrain];
for (i=0;i<ntrain;i++)
        bag[i] = (int *) new int[x];


//tableau pour contenir le % de 1
double *pour=new double[ntest];

ofstream fopen(fichierarbres);
ofstream open(classification);

srand(time(NULL));
for (int iter=0; iter<narbre; iter++)
{
        cout << "arbre no: " << iter << endl;
        long double aleat;
        long double max=RAND_MAX;

        //il faut ramener les poids sur 100
```

```
double total=0;

for (int i=0; i<ntrain; i++)
        total=total+poids[i];
//cout << "total: " << total << endl;

for (i=0; i<ntrain; i++)
        poidscum[i]=poids[i]/total;

//calcul des poids cumulés

for (int j=1; j<ntrain; j++)
        poidscum[j]=poidscum[j-1]+poidscum[j];

//création de l'ensemble d'entraînement

for (i=0; i<ntrain; i++)
{
        int j=0;
        aleat=rand()/max;
        if (aleat==1)
        {
                while(aleat==1)
                        aleat=rand()/max;
        }
        else
        {

                while (poidscum[j]<aleat)
        {
                        j++;
        }
}

        //copie de la ligne
        for (int k=0; k<x; k++)
        {
                bag[i][k]=test[j][k];
        }
}

principal(bag, ntrain, test, ntest, prev, iter, min, x, choix, couts, fopen);

// calcul du taux de bonne pred

double erreurnum=0;
```

```
        double erreurden=0;

        for (i=0; i<ntrain; i++)
        {
                if(prev[i][iter]!=test[i][0])
                        erreurnum=erreurnum+poids[i];
                erreurden=erreurden+poids[i];
        }

        double erreur=erreurnum/erreurden;
        double alpha=log((1-erreur)/erreur)/log(10);

        cout << "erreur: " << erreur << " alpha: " << alpha << endl << endl;

        for (i=0; i<ntrain; i++)
        {
                if(prev[i][iter]!=test[i][0])
                        poids[i]=poids[i]*exp(alpha);
        }

        double correct=0;

        double nb=ntest;
        for (int t=0; t<ntest; t++)
        {
                if (test[t][0]==prev[t][iter])
                        correct++;
        }


        double taux=correct/nb*100;
        open << taux << endl;
}

fopen.close();
open.close();

double nbuns=0;
double sommea=0;

for( i=0;i<narbre;i++)
{
        sommea=sommea+alphas[i];
}

for( i=0;i<ntest;i++)
```

```
        {
                for(int j=0;j<narbre;j++)
                {
                        if (prev[i][j]==1)
                                num=num+alphas[j];
                }
                pour[i]=num/sommea*100;
                num=0;
        }

        ofstream fout(probabilites);

        for( i=0;i<ntest;i++)
        {
                fout << pour[i];
                fout << endl;
        }
        fout.close();
}
```

Randomisation avec échantillons bootstrap

La randomisation avec échantillons bootstrap utilise la même fonction principale que le bagging. La différence se trouve au niveau de la fonction qui décide du meilleur embranchement, qui diffère de celle utilisée par CART, étant donné l'aléatoire qui y est ajouté. De plus, cette fonction appelle une nouvelle fonction, appelée randomisation. Nous présenterons donc ici la fonction embranchement modifiée, de même que la nouvelle fonction.

```
void creer(int**, int, int, int**, int, int**, int, int*);
void principal(int**, int, int**, int, int**, int, int, int, double*, ofstream&, int);
bool prediction(int, int, int**, double*);
int arbre(noeud*, int, int, int**, int, int, int, double*, int);
        int embranchement(noeud*, int, int, int, int**, int, int, double*, int);
                double gini(int , int , int , int , int , int , int, double* );
                void randomisation(int, double**, double*);
                void quicksort(int**, int, int);
        int separation(noeud*, int, int, int, int**);
                void echange(int**, int, int, int, int, int);
                void echange2(int**, int, int, int, int, int);
void prevision(noeud*, int, int**, int**, int);
void Detruire (noeud*);

void write(ofstream&, noeud*);
void writefly(ofstream&, noeud*);

int embranchement(noeud *courant, int debut,int fin, int x, int **tab, int min, int ntrain,
double *couts, int nsplit)
{
        int nbpere=courant->nobs;     //nombre d'obs dans le noeud

        double GiniMin[4]={100,0,-1,-1};
        //contient l'indice min et son point de coupure pour une variable
        //doit être réinitialisé lors du test pour chaque nouvelle variable
        //le troisième indique le traitement des valeurs manquantes:
        //-1=>pas de vm, 0=>vm à gauche, 1=>vm à droite, le 4ème conserve l'indice
        //de la variable

        bool ok=0;
```

```
double **index;
index = (double **) new double*[3];
for (int i=0;i<4;i++)
        index[i] = (double *) new double[nsplit];
//contient en première ligne l'indice min pour chaque variable,
// et en seconde le point de coupure correspondant.

for (i=0; i<nsplit; i++)
{
        index[0][i]=-1;
        index[1][i]=-1;
        index[2][i]=-1;
        index[3][i]=-1;
}

//Traitement pour chaque variable
int p=0;                //point de coupure courant
double ginmin=100;  //indice Gini minimum du moment

int indicevar=0;

//Il faut d'abord vérifier si on a assez d'observations...
if (courant->nobs>=min)
{

        //il faut ensuite vérifier si le noeud est pur...
        bool pur=1;
        for (i=debut; i<fin; i++)
                {
                        if (tab[i][0]!=tab[i+1][0])      //alors le noeud n'est pas pur
                                pur=0;
                }

        if (pur==0)
        {
                //pour chaque variable...
                for(int v=1; v<x; v++)
                {

                        int dernier=0;  //indice du dernier élément dans le tableau
                        int *coup=new int[nbpere];   //pour mettre les différentes
                                                     //valeurs

                        for (i=0; i<nbpere; i++)
                                coup[i]=-1;
```

```
//il faut d'abord trouver tous les points de coupure possible

coup[0]=tab[debut][v];
bool present=0;
for (i=debut+1; i<=fin; i++)
{
        //on veut regarder si la valeur contenue dans
        //tab[i][v] appartient à coup
        int j=0;
        while (coup[j]<tab[i][v] && coup[j]!=-1)
                j++;

        if (tab[i][v]!=coup[j]) //alors il faut l'ajouter à j-1
        {
                if (tab[i][v]>coup[j])
                {
                        coup[dernier+1]=tab[i][v];
                }
                else
                {
                        for (int k=dernier; k>=j; k--)
                                coup[k+1]=coup[k];
                                //on incrémente les éléments
                                de 1 //position
                        coup[j]=tab[i][v];
                }
                dernier++;
        }
        j=0;
}

if (dernier!=0)
{
```

/*on a plus besoin d'un tableau de longueur nbpere: on peut transférer les observations dans un tableau plus petit, qui en première ligne contiendra les valeurs des points de coupure, en seconde ligne le nombre de 0 et en troisième ligne le nombre de 1 associé à chaque point de coupure*/

```
        int **points;

        points = (int **) new int*[3];
        for (i=0;i<3;i++)
                points[i] = (int *) new int[dernier+1];

        for (int u=0; u<=dernier; u++)
```

```
{
        points[0][u]=coup[u];
        points[1][u]=0;
        points[2][u]=0;
}
delete []coup;

//il faut ensuite trouver le nombre de 0 et 1 associés
//à chaque point

int j=0;
for (int i=debut; i<=fin; i++)
{
        j=0;
        if(tab[i][v]!=points[0][j])
        {
                while (tab[i][v]!=points[0][j]
                                && j<dernier+1)
                {
                        j++;
                }
        }
        if(tab[i][0]==0)
                points[1][j]++;
        else
                points[2][j]++;
}



//si on a des valeurs manquantes, points[0][0]=-100

int indicep=0;
//index[2][v]=-1;
if(points[0][0]==-100)
{
        indicep=1;
}
indicep++;
//parce que la première valeur n'est pas candidate à
//cause du >=


int p;
for (int ip=indicep; ip<=dernier; ip++)
```

```
//pour chaque valeur possible du point de coupure
{
        //point de coupure du moment

        p=points[0][ip];

        int g0=0;
        int g1=0;
        int d0=0;
        int d1=0;

        for (int i=indicep-1; i<ip; i++)
        {
                g0=g0+points[1][i];
                g1=g1+points[2][i];
        }
        for (i=ip; i<=dernier; i++)
        {
                d0=d0+points[1][i];
                d1=d1+points[2][i];
        }

        //S'il n'y a pas de vm...
        if (points[0][0]!=-100)
        {
                int g=g0+g1;
                int d=d0+d1;

                //on a un split, mais il faut vérifier si
                //on a plus de 5% de chaque côté...

                if (g>=min/2 && d>=min/2)
                {
                        ok=1;

                double igini=gini(g, d, g0, g1, d0,
                                d1, nbpere, couts);

                        GiniMin[0]=igini;
                        GiniMin[1]=p;
                        GiniMin[2]=-1;
                        GiniMin[3]=v;
                randomisation(nsplit,index,
                                        GiniMin);
                }
}
```

```
                else
                {
                        //valeurs manquantes à gauche
                        int g=g0+g1+points[1][0]+
                                                points[2][0];
                        int d=d0+d1;

                //on a un split, mais il faut vérifier si
                //on a plus de 5% de chaque côté...

                        if (g>=min/2 && d>=min/2)
                        {
                                ok=1;
                                int tg0=g0+points[1][0];
                                int tg1=g1+points[2][0];
                                double igini=gini(g, d, tg0,
                                tg1, d0, d1, nbpere, couts);

                                GiniMin[0]=igini;
                                GiniMin[1]=p;
                                GiniMin[2]=0;
                                GiniMin[3]=v;
                                randomisation(nsplit,index,
                                                GiniMin);
                        }

                }


        }//fin du for pour les p

        //on a trouvé les indices pour tous les p, reste à
        //regarder, s'il y a des valeurs manquantes, si les
        //mettre toutes à gauche

        if (points[0][0]==-100)         //alors on a des vm
        {
                int nb0=0;
                int nb1=1;

                for (int i=1; i<=dernier; i++)
                {
                        nb0=nb0+points[1][i];
                        nb1=nb1+points[2][i];
                }
                int miss0=points[1][0];
```

```
                                        int miss1=points[2][0];

                                        int g=points[1][0]+points[2][0];
                                        int d=nb0+nb1;

                                        if (g>=min/2 && d>=min/2)
                                        {
                                                ok=1;
                                                double igini=gini(g, d, points[1][0],
                                                points[2][0], nb0, nb1, nbpere,
                                                                        couts);

                                                GiniMin[0]=igini;
                                                GiniMin[1]=0; //0 indique une sep
                                                                //par vm
                                                GiniMin[2]=2;//2 indique seulement
                                                                //vm à gauche
                                                GiniMin[3]=v;
                                                randomisation(nsplit,index,
                                                                        GiniMin);
                                        }
                                }
                        }
                }
        }
}


        if (ok==0)
        {
                //alors il n'y a pas eu de split trouvé
                courant->filsd=NULL;
                courant->filsg=NULL;
                courant->split[0]=-1;  //pas de variable
                courant->split[1]=-1;  //pas de point de coupure
                courant->split[2]=-1;  //pas de valeur manquante
                courant->feuille=1;             //c'est une feuille
//              cout << "fin embranchement feuille" << endl;
                return 0;
        }
        else
        {
                int j=0;
                if (index[0][nsplit-1]==-1)     //le tableau n'est pas plein
                {
                        while(index[0][j]!=-1)
```

```
                        j++;
                }
                else
                {
                        j=nsplit;
                }

                int aleat=rand()%j;

                courant->split[0]=int(index[3][aleat]);
                //cout << "var: " <<index[3][aleat] << endl;
                courant->split[1]=int(index[1][aleat]);
                //cout <<  "point: " <<index[1][aleat] << endl;
                courant->split[2]=int(index[2][aleat]);
                //cout << "vm: " <<index[2][aleat] << endl;
        }
        return 1;
}

void randomisation(int nsplit, double **index, double *GiniMin)
{
        if(index[0][nsplit-1]==-1)      //alors le tableau n'est pas plein
        {
                int j=0;
                while(index[0][j]!=-1)
                        j++;

                for (int i=0; i<4; i++)
                        index[i][j]=GiniMin[i];
        }
        else
        {
                //il faut trouver le maximum
                int j=0;
                double max;

                max=index[0][0];
                for (int i=1; i<nsplit; i++)
                {
                        if(index[0][i]>max)
                        {
                                max=index[0][i];
                                j=i;
                        }
                }
                if (GiniMin[0]<index[0][j])
```

```
                {
                        for (int i=0; i<4; i++)
                                index[i][j]=GiniMin[i];
                }
                //int tata;
                //if (index[1][0]==-1)
                //      cin >> tata;
        }
}
```

Forêt aléatoire (ForetM)

Comme dans le cas de la randomisation, la fonction principale de base est ici aussi celle du bagging. Seule la fonction d'embranchement est modifiée.

```
void creer(int**, int, int, int**, int, int**, int, int*);
void principal(int**, int, int**, int, int**, int, int, int, int, double*,ofstream&);
bool prediction(int, int, int**, double*);
int arbre(noeud*, int, int, int**, int, int, int, int, double*);
        int embranchement(noeud*, int, int, int, int**, int, int, int, double*);
                double gini(int , int , int , int , int , int , int, double*);
                void quicksort(int**, int, int);
        int separation(noeud*, int, int, int, int**);
                void echange(int**, int, int, int, int, int);
                void echange2(int**, int, int, int, int, int);
void prevision(noeud*, int, int**, int**, int);
void Detruire (noeud*);

void write(ofstream&, noeud*);
void writefly(ofstream&, noeud*);

int embranchement(noeud *courant, int debut,int fin, int x, int **tab, int min, int
 ntrain, int choix, double *couts)
{
        int nbpere=courant->nobs;     //nombre d'obs dans le noeud

double GiniMin[3]={100,0,-1};         //contient l'indice min et son point de
                                      //coupure pour une variable
                                      //doit être réinitialisé lors du test pour
                                      //chaque nouvelle variable
                                      //le troisième indique le traitement des
                                      //valeurs manquantes:
                                      //-1=>pas de vm, 0=>vm à gauche, 1=>vm à droite
bool ok=0;



double **index;
index = (double **) new double*[4];
for (int i=0;i<4;i++)
index[i] = (double *) new double[choix];
        for ( i=0; i<4; i++)
        {
```

```
            for (int j=0; j<choix; j++)
                    index[i][j]=100;
    }
```

//contient en première ligne l'indice min pour chaque variable, et en seconde le //point de coupure correspondant, en troisième, l'indice de valeur manquante et en //dernier l'indice de la variable pour vérifier si la variable a déjà été utilisée...

```
    int* ch= new int[x];
    for (int h=0; h<x; h++)
            ch[h]=0;

    //Traitement pour chaque variable
    int p=0;                    //point de coupure courant
    double ginmin=100;  //indice Gini minimum du moment

    int indicevar=0;

    //Il faut d'abord vérifier si on a assez d'observations...
    if (courant->nobs>=min)
    {

            //il faut ensuite vérifier si le noeud est pur...
            bool pur=1;
            for (i=debut; i<fin; i++)
                    {
                            if (tab[i][0]!=tab[i+1][0])     //alors le noeud n'est pas pur
                                    pur=0;
                    }

            if (pur==0)
            {
                    //pour chaque variable...
                    for(int v=0; v<choix; v++)
                    {
                            int var=rand()%x;       //choix d'une variable

                            if (var==0)
                            {
                                    while (var==0)
                                            var=rand()%x;
                            }

                            if(ch[var]!=0)  //alors la variable a déjà été utilisée
                            {
                                    while (ch[var]!=0)
```

```
                {
                        var=(var+1)%x;
                        if (var==0)
                        {
                                while (var==0)
                                        var=rand()%x;
                        }
                }
        }
        ch[var]=1; //la variable a déjà été utilisée

        index[3][v]=var;

int dernier=0;
//indice du dernier élément dans le tableau

int *coup=new int[nbpere];
//pour mettre les différentes valeurs

        for (i=0; i<nbpere; i++)
                coup[i]=-1;

                //il faut d'abord trouver tous les points de coupure
                //possible

        coup[0]=tab[debut][var];
        bool present=0;

        for (i=debut+1; i<=fin; i++)
        {
                //on veut regarder si la valeur contenue dans
                //tab[i][v] appartient à coup
                int j=0;
                while (coup[j]<tab[i][var] && coup[j]!=-1)
                        j++;

                if (tab[i][var]!=coup[j])
                //alors il faut l'ajouter à j-1
                {
                        if (tab[i][var]>coup[j])
                        {
                                coup[dernier+1]=tab[i][var];
                        }
                        else
                        {
```

```
                        for (int k=dernier; k>=j; k--)
                                coup[k+1]=coup[k];
                                //on incrémente les
                                //éléments de 1 position
                        coup[j]=tab[i][var];
                }
                dernier++;
        }
        j=0;
}
if (dernier!=0)
{
                //on a plus besoin d'un tableau de longueur
                //nbpere: on peut transférer les
                //observations dans un tableau plus petit, qui
                //en première ligne contiendra
                //les valeurs des points de coupure, en
                //seconde ligne le nombre de 0 et en
                //troisième ligne le nombre de 1 associé à
                //chaque point de coupure

        int **points;

        points = (int **) new int*[3];
        for (i=0;i<3;i++)
                points[i] = (int *) new int[dernier+1];

        for (int u=0; u<=dernier; u++)
        {
                points[0][u]=coup[u];
                points[1][u]=0;
                points[2][u]=0;
        }

        delete []coup;

                //il faut ensuite trouver le nombre de 0 et 1
                //associés à chaque point

        int j=0;
        for (int i=debut; i<=fin; i++)
        {
                j=0;
                if(tab[i][var]!=points[0][j])
                {
                        while (tab[i][var]!=points[0][j] &&
```

```
                                        j<dernier+1)
                {
                        j++;
                }
        }
        if(tab[i][0]==0)
                points[1][j]++;
        else
                points[2][j]++;
}
                //si on a des valeurs manquantes,
                //points[0][0]=-100
int indicep=0;
index[2][v]=-1;
if(points[0][0]==-100)
{
        index[2][v]=1;
        indicep=1;
}
indicep++;              //parce que la première valeur
                        //n'est pas candidate à cause
                        //du >=
int p;
//pour chaque valeur possible du point de
//coupure
for (int ip=indicep; ip<=dernier; ip++)
{
        //point de coupure du moment

        p=points[0][ip];

        int g0=0;
        int g1=0;
        int d0=0;
        int d1=0;

        for (int i=indicep-1; i<ip; i++)
        {
                g0=g0+points[1][i];
                g1=g1+points[2][i];
        }
        for (i=ip; i<=dernier; i++)
        {
                d0=d0+points[1][i];
                d1=d1+points[2][i];
        }
```

```
//S'il n'y a pas de vm...
if (index[2][v]==-1)
{
        int g=g0+g1;
        int d=d0+d1;

                //on a un split, mais il faut
                //vérifier si on a plus de 5%
                //de chaque côté...

        if (g>=min/2 && d>=min/2)
        {
        ok=1;

        double igini=gini(g, d, g0, g1, d0,
                        d1, nbpere, couts);
        if (igini < GiniMin[0])
        {
                GiniMin[0]=igini;
                GiniMin[1]=p;
        }
        }
}
else
{
                //valeurs manquantes à
                //gauche
        int g=g0+g1+points[1][0]+
                                points[2][0];
        int d=d0+d1;

                //on a un split, mais il faut
                //vérifier si on a plus de 5%
                //de chaque côté...

        if (g>=min/2 && d>=min/2)
        {
        ok=1;
        int tg0=g0+points[1][0];
        int tg1=g1+points[2][0];
        double igini=gini(g, d, tg0, tg1, d0,
                        d1, nbpere, couts);

        if (igini < GiniMin[0])
        {
```

```
                    GiniMin[0]=igini;
                    GiniMin[1]=p;
                    àGiniMin[2]=0;
                }
                }
        }

}//fin du for pour les p

        //on a trouvé les indices pour tous les p, reste
        //à regarder, s'il y a des valeurs manquantes,
        //si les mettre toutes à gauche

if (index[2][v]!=-1)    //alors on a des vm
{
        int nb0=0;
        int nb1=1;

        for (int i=1; i<=dernier; i++)
        {
                nb0=nb0+points[1][i];
                nb1=nb1+points[2][i];
        }
        int miss0=points[1][0];
        int miss1=points[2][0];

        int g=points[1][0]+points[2][0];
        int d=nb0+nb1;

        if (g>=min/2 && d>=min/2)
        {
        ok=1;
        double igini=gini(g, d, points[1][0],
        points[2][0], nb0, nb1, nbpere, couts);

        if (igini < GiniMin[0])
        {
                GiniMin[0]=igini;
                GiniMin[1]=0;//0 indique une
                                //sep par vm
                GiniMin[2]=2;//2 indique
                                //seulement    vm    à
                                //gauche
        }
        }
}
```

```
                                index[0][v]=GiniMin[0];
                                index[1][v]=GiniMin[1];
                                index[2][v]=GiniMin[2];
                                GiniMin[0]=100;
                                GiniMin[1]=0;
                                GiniMin[2]=-1;
                                p=0;
                        }
                }

                // reste à trouver le minimum dans index...
                for (v=0; v<choix; v++)
                {
                        if(index[0][v]<GiniMin[0])
                        {
                                GiniMin[0]=index[0][v];
                                GiniMin[1]=index[1][v];
                                GiniMin[2]=index[2][v];
                                indicevar=int(index[3][v]);
                        }
                }
        }
}

        if (ok==0)
        {
                //alors il n'y a pas eu de split trouvé
                courant->filsd=NULL;
                courant->filsg=NULL;
                courant->split[0]=-1;  //pas de variable
                courant->split[1]=-1;  //pas de point de coupure
                courant->split[2]=-1;  //pas de valeur manquante
                courant->feuille=1;             //c'est une feuille
                return 0;
        }
        else
        {
                courant->split[0]=indicevar;
                courant->split[1]=int(GiniMin[1]);
                courant->split[2]=int(GiniMin[2]);
        }

        delete []ch;
        delete []index;
        return 1;       }
```