1 Objectif

Stratégie de swap pour le traitement des très grands fichiers avec R.

Le traitement des grands fichiers est un problème récurrent du data mining[1]. Pour s'en sortir, il faut éviter de charger toutes les données en mémoire. L'idée est relativement simple : (1) nous devons écrire (swaper) les données sur disque, dans un format binaire afin de permettre un accès indexé ; (2) les algorithmes devront chercher les informations sur le disque et non plus en mémoire centrale lors de la phase de construction des modèles. L'avantage est évident. Les données ne sont jamais simultanément totalement montées en mémoire, les capacités de la machine ne constituent plus un goulot d'étranglement. L'inconvénient est un temps de calcul qui peut se révéler prohibitif. En effet, chaque accès à une valeur se traduit en un accès disque.

Plusieurs pistes sont possibles pour produire une solution viable : d'une part, il faut adopter une organisation des données efficace sur le disque dur ; d'autre part, il faut mettre en place un système de cache c.-à-d. monter en mémoire une fraction des informations lors de la lecture de manière à ce que la prochaine lecture se fasse en mémoire, on minimise ainsi les accès disques.

Dans ce didacticiel, nous étudierons une solution mise en place dans R sous la forme d'une libraire. Le package « **filehash** » permet de copier (de « dumper » carrément) tous types d'objets sur le disque, les données mais aussi les modèles. Il utilise un format de type base de données. Il présente un avantage énorme, il est possible d'utiliser les fonctions statistiques standards ou issus d'autres packages sans avoir à procéder à une quelconque adaptation. Au lieu de manipuler des data.frame en mémoire, elles travaillent sur des data.frame stockés sur le disque, de manière totalement transparente. C'est assez épatant, il faut l'avouer. Les capacités de traitement sont largement améliorées et, dans le même temps, la dégradation du temps de calcul n'est pas rédhibitoire.

Néanmoins, nous constaterons que les fonctions R n'étant pas spécifiquement conçus pour l'appréhension des grands ensembles de données, lorsque nous augmentons encore nos exigences, les calculs ne sont plus possibles alors que les ressources ne sont pas entièrement utilisées. C'est un peu la limite des approches génériques. La modification des algorithmes d'apprentissage est souvent nécessaire pour exploiter au mieux les particularités du contexte. Il faudrait même aller plus loin. Pour obtenir des résultats réellement probants, il faudrait à la fois adapter les algorithmes d'apprentissage et organiser en conséquence les données sur le disque. Une solution qui conviendrait à tout type d'analyse paraît difficile, voire illusoire.

Pour évaluer la solution apportée par le package « filehash », nous étudierons le temps de calcul et l'occupation mémoire, avec ou sans swap sur le disque, lors du calcul de statistiques descriptives, de l'induction d'un arbre de décision avec **rpart** du package du même nom, et de la modélisation à l'aide de l'analyse discriminante avec la fonction **lda** de la librairie MASS.

[1] Généralement plusieurs millions de lignes et plusieurs dizaines de variables lorsque nous traitons des données marketing ; à l'inverse, plusieurs centaines d'observations et plusieurs dizaines de milliers de variables lorsque l'on traite des données initialement non structurées comme en bioinformatique, traitement d'images, text mining, etc.

Nous réaliserons les mêmes opérations dans SIPINA. En effet, ce dernier propose également une solution de swap pour l'appréhension des très grandes bases de données. Nous pourrons ainsi comparer les performances des stratégies implémentées.

2 Données

Nous utilisons le fichier **wave** de Breiman et al. (1984) comportant 2.000.000 d'observations, déjà utilisé par ailleurs (http://tutoriels-data-mining.blogspot.com/2009/10/sipina-accelerer-par-lechantillonnage.html). Nous le mettons au format texte avec séparateur tabulation. Nous incluons dans l'archive le code source R (http://eric.univ-lyon2.fr/~ricco/tanagra/fichiers/wave2M.txt.zip).

3 Analyse standard avec R

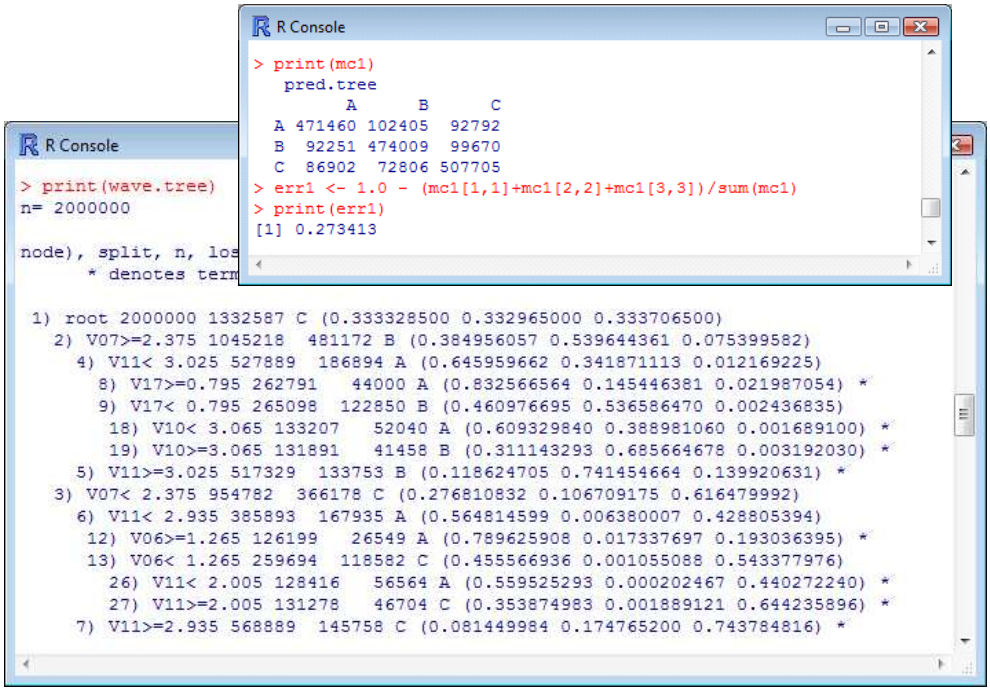
3.1 Traitements en mémoire centrale (5 variables prédictives)

Dans un premier temps, nous chargeons les données dans un data.frame et nous calculons les statistiques descriptives. Dans un deuxième temps, nous procédons à la modélisation à l'aide de **rpart** et **lda**. Nous n'utilisons que 5 variables prédictives (V07, V11, V17, V06, V10) parmi les 21 présentes dans le fichier. Nous calculons les prédictions des modèles sur les mêmes données. Nous produisons enfin les matrices de confusion et taux d'erreur.

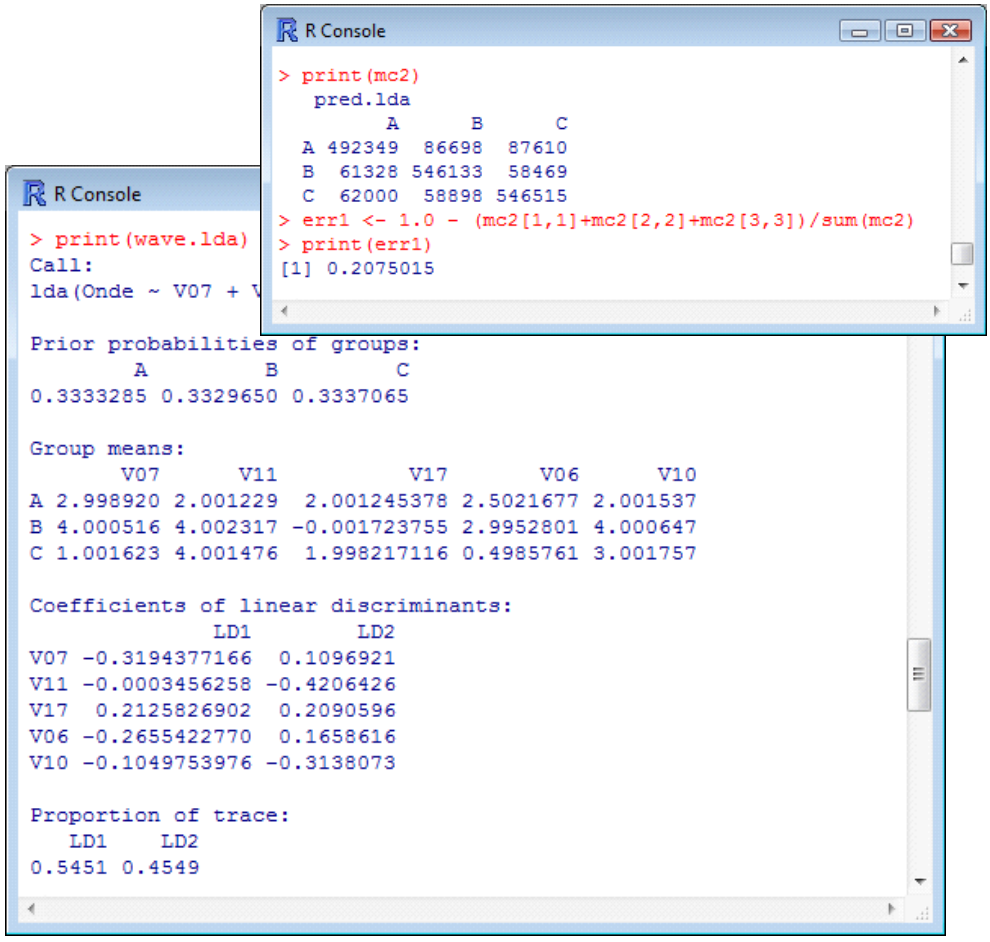
Le code source R est le suivant.

```
#clear the memory
rm (list=ls())
#loading the data file
setwd("D:/DataMining/Databases_for_mining/dataset_for_soft_dev_and_comparison/swap_strategy")
wave.data <- read.table(file = "wave2M.txt", sep = "\t", dec = ".", header = T)
#checking the data
print(summary(wave.data))
#learning a decision tree - rpart
library(rpart)
settings <- rpart.control(xval=0,minsplit=100000,minbucket=100000,maxsurrogate=0,cp=0)
wave.tree <- rpart(Onde ~ V07+V11+V17+V06+V10, data = wave.data, control = settings)
print(wave.tree)
#linear discriminant analysis
library(MASS)
wave.lda <- lda(Onde ~ V07+V11+V17+V06+V10, data = wave.data)
print(wave.lda)
#prediction
pred.tree <- predict(wave.tree, newdata = wave.data, type = "class")
pred.lda <- predict(wave.lda, newdata = wave.data)$class
#confusion matrix
mc1 <- table(wave.data$Onde,pred.tree)
err1 <- 1.0 - (mc1[1,1]+mc1[2,2]+mc1[3,3])/sum(mc1)
mc2 <- table(wave.data$Onde,pred.lda)
err2 <- 1.0 - (mc2[1,1]+mc2[2,2]+mc2[3,3])/sum(mc2)
```

Première information très importante, le calcul a été possible (sur ma machine en tous les cas). Nous obtenons l'arbre suivant avec un taux d'erreur de 27.34%.



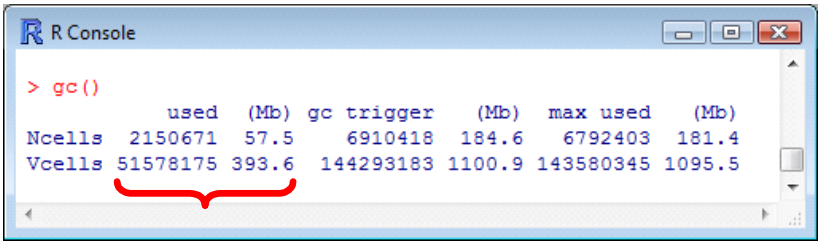
Pour l'analyse discriminante, nous avons un taux d'erreur de 20.75%



Nous récapitulons dans un tableau (Figure 1) le temps de traitement et l'occupation mémoire à chaque étape du processus. Concernant la mémoire, nous donnons prioritairement l'occupation mesurée par R lui-même. A la fin des calculs, nous fournissons l'occupation mémoire de R dans Windows, indiquée par le gestionnaire de tâches.

	Traitement en mémoire	
Apprentissage	Temps (sec.)	Occ. mémoire Mo (Ncells+Vcells)
Connexion/chargement données	51.57	331.8
Statistique descriptive	5.54	332
Création arbre (rpart)	38.52	427.7
Création analyse discriminante (lda)	11.79	428.1
Sauvegarde arbre	--	--
Sauvegarde lda	--	--
Prédiction		
Connexion/chargement données	--	--
Prédiction arbre	3.9	443.4
Prédiction analyse discriminante	38.41	451.1
Calcul matrices de confusion	1.15	451.1
Occupation mémoire Windows (Mo)	--	591

Figure 1 - Performances - Données en mémoire



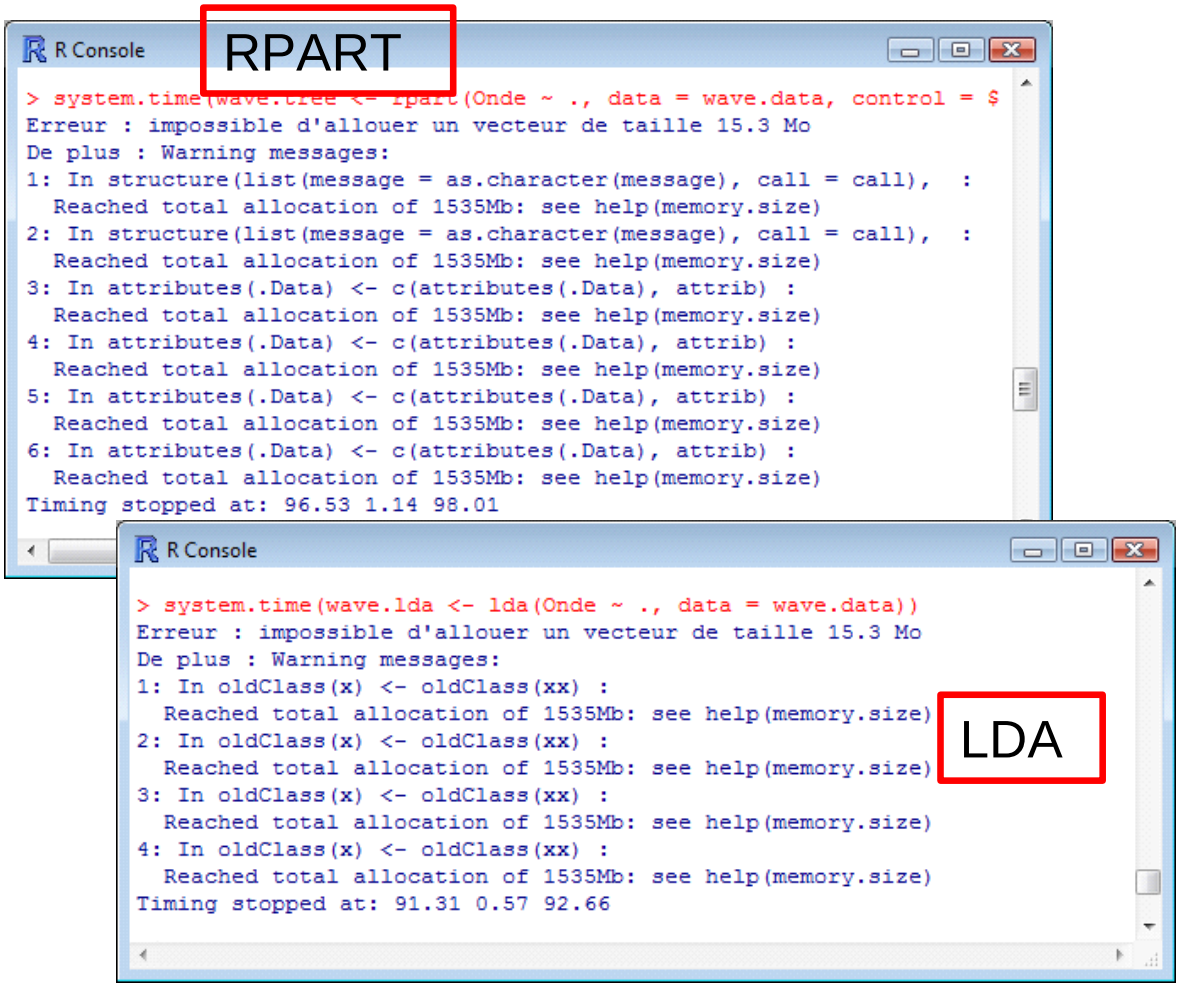
Les indications de la fonction **gc()** laissent à penser qu'il reste encore de la marge. En comparant la mémoire utilisée et le maximum autorisé, il semble que nous puissions aller plus loin encore.

3.2 Traitement en mémoire (toutes les variables prédictives)

Nous avons voulu reproduire le processus en incluant toutes les variables lors de la construction des modèles (arbre et analyse discriminante). Le code est le suivant.

```
#clear the memory
rm (list=ls())
#loading the data file
setwd("D:/DataMining/Databases_for_mining/dataset_for_soft_dev_and_comparison/swap_strategy")
wave.data <- read.table(file = "wave2M.txt", sep = "\t", dec = ".", header = T)
#checking the data
print(summary(wave.data))
#learning a decision tree - rpart
library(rpart)
settings <- rpart.control(xval=0,minsplit=100000,minbucket=100000,maxsurrogate=0,cp=0)
wave.tree <- rpart(Onde ~ ., data = wave.data, control = settings)
print(wave.tree)
#linear discriminant analysis
library(MASS)
wave.lda <- lda(Onde ~ ., data = wave.data)
print(wave.lda)
```

R nous annonce que l'opération n'est pas possible pour les deux algorithmes. Une tentative d'allocation mémoire trop importante déclenche une exception.



Il faudrait une connaissance étendue de R (que je n'ai pas) pour décrypter le message d'erreur et pouvoir proposer des solutions pour contourner la limitation. Pour notre part, nous allons nous tourner vers le package « filehash » qui nous permet de réaliser directement les traitements sur disque, sans avoir à charger toutes les données simultanément en mémoire.

4 Solution de swap pour R – Le package « filehash »

Le package « filehash » (http://cran.r-project.org/web/packages/filehash/index.html) permet de « dumper » tout type d'objet dans un fichier. Dans un premier temps, nous allons détailler sa mise en œuvre lors de la modélisation à 5 variables prédictives. Notre objectif est de mesurer le gain en mémoire ainsi obtenu et, en contrepartie, la perte en rapidité de calcul. Est-ce que l'un justifie l'autre ? Puis, dans un second temps, nous allons essayer de mettre en œuvre le package pour le traitement de la totalité des variables prédictives.

4.1 Création d'un fichier de stockage «.db »

Il nous faut tout d'abord installer la librairie « filehash » (cf. la description de la procédure d'installation des packages sous R – http://tutoriels-data-mining.blogspot.com/2009/05/installation-des-packages-sous-r.html). Ensuite, le code suivant est destiné à importer les données du fichier

texte vers un data.frame, puis copier cette dernière dans la base de données. **Attention, l'opération dure plus d'une heure !** Plusieurs fois, je l'ai moi-même interrompu, pensant à un plantage. Il faut être patient simplement.

```
#clear the memory
rm (list=ls())
#loading the data file into a data.frame "wave.data" in memory
setwd("D:/DataMining/Databases_for_mining/dataset_for_soft_dev_and_comparison/swap_strategy")
wave.data <- read.table(file = "wave2M.txt", sep = "\t", dec = ".", header = T)
#creating a database and insert the data.frame into the database
#loading the "filehash" package
library(filehash)
#creating a database with the name "wave.dataset.db"
dbCreate("wave.dataset.db")
#initialize the database
db <- dbInit("wave.dataset.db")
#insert the data.frame into the database - the key is "wave.data"
dbInsert(db,"wave.data",wave.data)
```

Un petit mot sur la taille des fichiers : le fichier texte « wave2M.txt » occupe 213.988 Ko sur le disque, le fichier binaire fait 335.939 Ko. On devine la raison de cette inflation : les valeurs réelles sont codées sur 8 octets, les valeurs entières associées au type « factor » (variable catégorielle), sur 4. Pour 2.000.000 d'observations avec 21 variables continues et une catégorielle, nous aurons

$$(2000000 \times 21 \times 8 + 2000000 \times 1 \times 4) / 1024 \approx 335.937{,}5 \text{ Ko}$$

Nous devons y ajouter l'espace dévolu à la structure de liste du data.frame.

4.2 Traitement via la base au format « .db » et sauvegarde des modèles

Maintenant, nous allons réalisons les mêmes traitements que précédemment (section 3.1), avec les 5 variables prédictives. Le code R est le suivant.

```
#clear the memory
rm (list=ls())
#connecting the database
#loading the "filehash" package
library(filehash)
#initialize the database
setwd("D:/DataMining/Databases_for_mining/dataset_for_soft_dev_and_comparison/swap_strategy")
db <- dbInit("wave.dataset.db")
#attach the database to an environnement
wave.env <- db2env(db)
#checking the dataset
print(summary(wave.env$wave.data))
#learning a decision tree - rpart
library(rpart)
settings <- rpart.control(xval=0,minsplit=100000,minbucket=100000,maxsurrogate=0,cp=0)
wave.tree <- rpart(Onde ~ V07+V11+V17+V06+V10, data = wave.env$wave.data, control = settings)
print(wave.tree)
#linear discriminant analysis
```

```
library(MASS)
wave.lda <- lda(Onde ~ V07+V11+V17+V06+V10, data = wave.env$wave.data)
print(wave.lda)
#saving the models into a database
#saving the tree
dumpObjects(wave.tree, dbName = "wave.models.db")
#saving the lda
dumpObjects(wave.lda, dbName = "wave.models.db")
```

Plusieurs éléments doivent attirer notre attention :

- Les données ne sont jamais explicitement chargées, nous nous connectons à la base de données avec l'instruction **dbInit()**.
- Nous l'associons à un environnement avec **db2env()**.
- A la fin du processus, grande nouveauté ici, nous pouvons sauvegarder les modèles dans un autre fichier binaire « .db » avec l'instruction **dumpObjects()**. Cet aspect est très intéressant, il nous permettra par la suite de les déployer dans d'autres contextes, voire de les distribuer, sans avoir à refaire à chaque fois le calcul des paramètres sur les données d'apprentissage. Nous y reviendrons dans la section suivante.

Le reste du processus est identique au traitement en mémoire, notamment les instructions destinées à la construction des modèles (**rpart** et **lda**). Pour les utilisateurs, le passage par une base de données est transparent. **Les résultats (arbre, coefficients de lda) obtenus sont exactement les mêmes !!! Ouf ! Le contraire aurait été très ennuyeux.**

Concernant les temps de traitement et l'occupation mémoire, nous obtenons :

	Traitement sur disque (filehash)	
Apprentissage	Temps (sec.)	Mémoire (Ncells + Vcells) Mo
Connexion/chargement données	0.02	6
Statistique descriptive	13.44	6.3
Création arbre (rpart)	48.67	101.9
Création analyse discriminante (lda)	23.61	102.2
Sauvegarde arbre	250.82	102.2
Sauvegarde lda	0.02	102.2

Le temps de calcul a été multiplié par deux (à peu près) pour ce qui est des statistiques descriptives et de l'analyse discriminante, par 1.26 pour l'arbre de décision. Mais, a contrario, l'occupation mémoire a été divisée par 4 à la fin du processus.

La taille mémoire occupée par R dans Windows a été fortement réduite aussi, elle est passée à 111.9 Mo (contre 591 Mo précédemment).

Tout est affaire d'arbitrage, bien évidemment, mais il est évident que le package « filehash » constitue une solution tout à fait valable pour l'appréhension des très grandes de données. Seule la création initiale du fichier binaire, opération excessivement longue, peut poser problème (section 4.1). Mais elle n'est effectuée qu'une seule fois. Par la suite, nous pourrons accéder directement autant que l'on souhaitera aux données du fichier « .db ».

4.3 Déploiement des modèles

Autre aspect très réjouissant de la libraire « filehash », nous pouvons sauvegarder les modèles dans des fichiers binaires « .db » et les réutiliser dans d'autres contextes. Dans ce qui suit, nous produisons un code qui permet de déployer l'arbre et le classifieur issu de la LDA sur un fichier stocké, lui aussi, dans un fichier au format « .db ». Nous utilisons notre fichier initial ici afin de comparer les matrices de confusion, mais cela pourrait être un autre fichier. L'important est que les variables de l'étude y soient présentes. Le code R est le suivant.

```
#clear the memory
rm (list=ls())
#connecting the dataset into a database
#loading the "filehash" package
library(filehash)
#initialize the database
setwd("D:/DataMining/Databases_for_mining/dataset_for_soft_dev_and_comparison/swap_strategy")
db.data <- dbInit("wave.dataset.db")
#attach the database to an environnement
wave.env.data <- db2env(db.data)
#loading the packages associated to the models
library(rpart)
library(MASS)
#connecting the models into the database
db.models <- dbInit("wave.models.db")
#attach the db to an environnement
wave.env.models <- db2env(db = db.models)
#printing the models
print(wave.env.models$wave.tree)
print(wave.env.models$wave.lda)
#prediction on the dataset
pred.tree <- predict(wave.env.models$wave.tree, newdata = wave.env.data$wave.data, type = "class")
pred.lda <- predict(wave.env.models$wave.lda, newdata = wave.env.data$wave.data)$class
#confusion matrix
mc1 <- table(wave.env.data$wave.data$Onde,pred.tree)
print(mc1)
err1 <- 1.0 - (mc1[1,1]+mc1[2,2]+mc1[3,3])/sum(mc1)
print(err1)
mc2 <- table(wave.env.data$wave.data$Onde,pred.lda)
print(mc2)
err2 <- 1.0 - (mc2[1,1]+mc2[2,2]+mc2[3,3])/sum(mc2)
print(err2)
```

Encore une fois, nous obtenons les mêmes matrices de confusion que lors du traitement en mémoire. Cette vérification était primordiale. Mais, ici, nous pouvons déployer un modèle dans tout type de contexte, sans avoir à le construire juste avant. L'avantage est énorme en termes de réutilisabilité.

Concernant le temps de traitement et l'occupation mémoire, nous avons :

	Traitement sur disque (filehash)	
Prédiction	Temps (sec.)	Mémoire (Ncells + Vcells) Mo
Connexion/chargement données	0.02	6
Prédiction arbre	22.45	94.3
Prédiction analyse discriminante	46.49	102
Calcul matrices de confusion	15.36	102

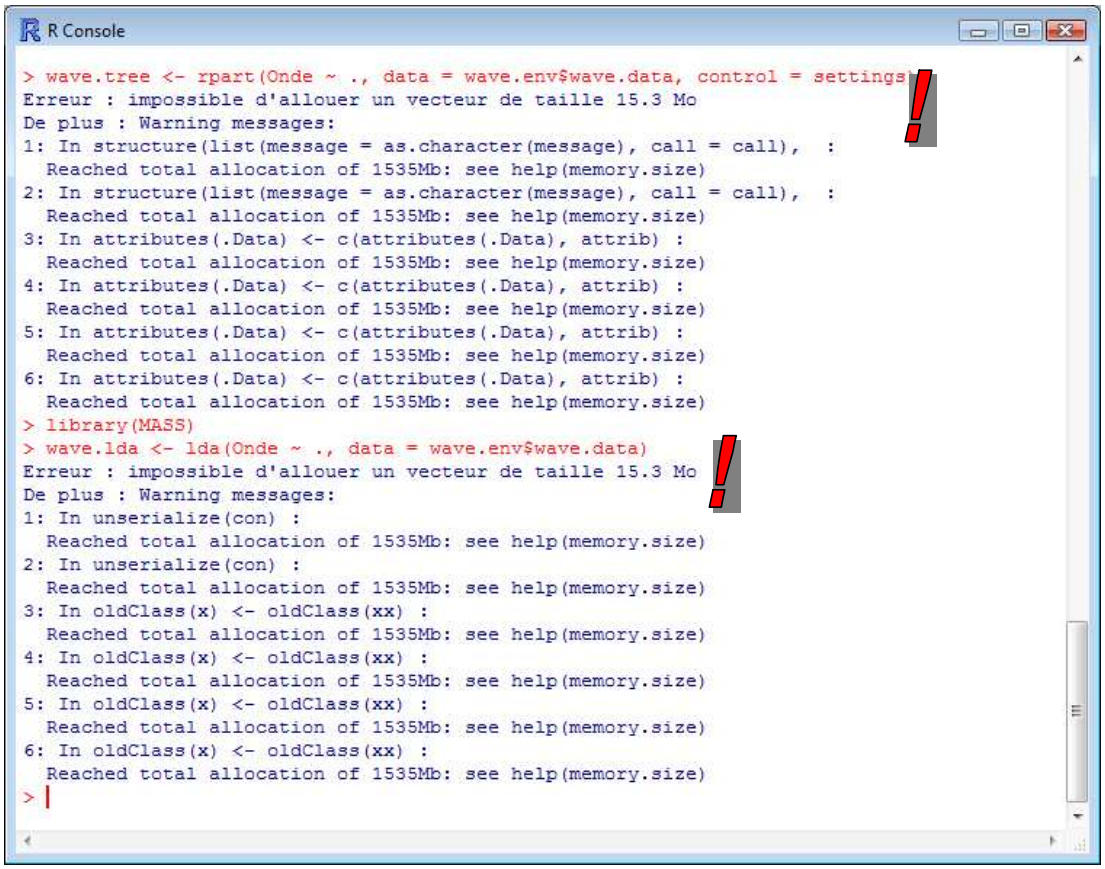
Ici également, nous sommes loin de saturer les ressources. Par rapport au traitement exclusivement en mémoire (Figure 1), l'augmentation du temps de calcul est plus important pour l'arbre de décision que pour l'analyse discriminante. Il en est de même pour la construction de la matrice de confusion où, pourtant, on n'utilise que la variable cible « Onde ». Cela tient vraisemblablement au mode de stockage du data.frame dans le fichier binaire. Les accès par colonne sont pénalisés.

4.4 Introduction de toutes les variables prédictives

Puisque le package « filehash » nous donne de la marge sur les ressources, nous pouvons légitimement penser qu'il est maintenant possible de traiter la totalité des variables prédictives de la base. Nous avons donc lancé l'apprentissage avec les 21 descripteurs.

```
#clear the memory
rm (list=ls())
#connecting the database
#loading the "filehash" package
library(filehash)
#initialize the database
setwd("D:/DataMining/Databases_for_mining/dataset_for_soft_dev_and_comparison/swap_strategy")
db <- dbInit("wave.dataset.db")
#attach the database to an environnement
wave.env <- db2env(db)
#checking the dataset
print(summary(wave.env$wave.data))
#learning a decision tree - rpart
library(rpart)
settings <- rpart.control(xval=0,minsplit=100000,minbucket=100000,maxsurrogate=0,cp=0)
wave.tree <- rpart(Onde ~ ., data = wave.env$wave.data, control = settings)
print(wave.tree)
#linear discriminant analysis
library(MASS)
wave.lda <- lda(Onde ~ ., data = wave.env$wave.data)
print(wave.lda)
```

Une énorme déception nous attend. Le calcul a echoué, **R nous envoie exactement les mêmes messages d'erreur que lors du traitement en mémoire**.



C'est très décevant. Surtout que nous n'utilisons pas la totalité de la mémoire disponible. La seule raison que l'on peut invoquer est que les fonctions **rpart** et **lda** n'ont pas été prévues pour traiter de tels volumes. J'imagine qu'elles procèdent mécaniquement à des allocations mémoires en relation avec la taille des bases, déclenchant ainsi des exceptions. C'est la seule explication qui me semble plausible.

Cela rejoint la remarque que nous formulions plus haut. Pour traiter de grandes bases sur disque, il est indispensable de mettre en place une solution efficace de swap des données sur disque, mais il faut aussi adapter en conséquence l'implémentation des algorithmes d'apprentissage.

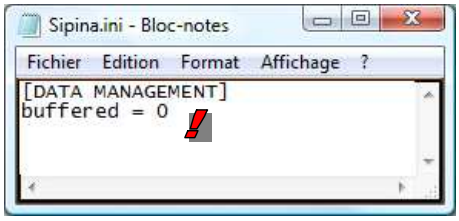
5 Solution de swap pour Sipina

Nous avions déjà présenté par ailleurs une solution de swap pour Sipina lors de l'induction des arbres de décision. Dans cette section, nous reproduisons l'expérimentation menée avec R, avec et sans swap. Par rapport au précédent tutoriel, nous évaluons le comportement de Sipina pour l'induction d'un arbre, mais aussi pour l'analyse discriminante. Les résultats seront particulièrement intéressants car la solution de Sipina a été avant tout conçue et optimisée pour l'élaboration des arbres de décision.

Nous réaliserons d'abord l'analyse à 5 variables prédictives. Nous verrons par la suite s'il est possible de passer à la totalité des variables incluses dans la base, chose que R n'a pas réussi à faire.

Enfin, Sipina ne donne pas d'indications sur l'occupation mémoire interne de l'application, nous nous fierons principalement aux informations fournies par le gestionnaire de tâches de Windows.

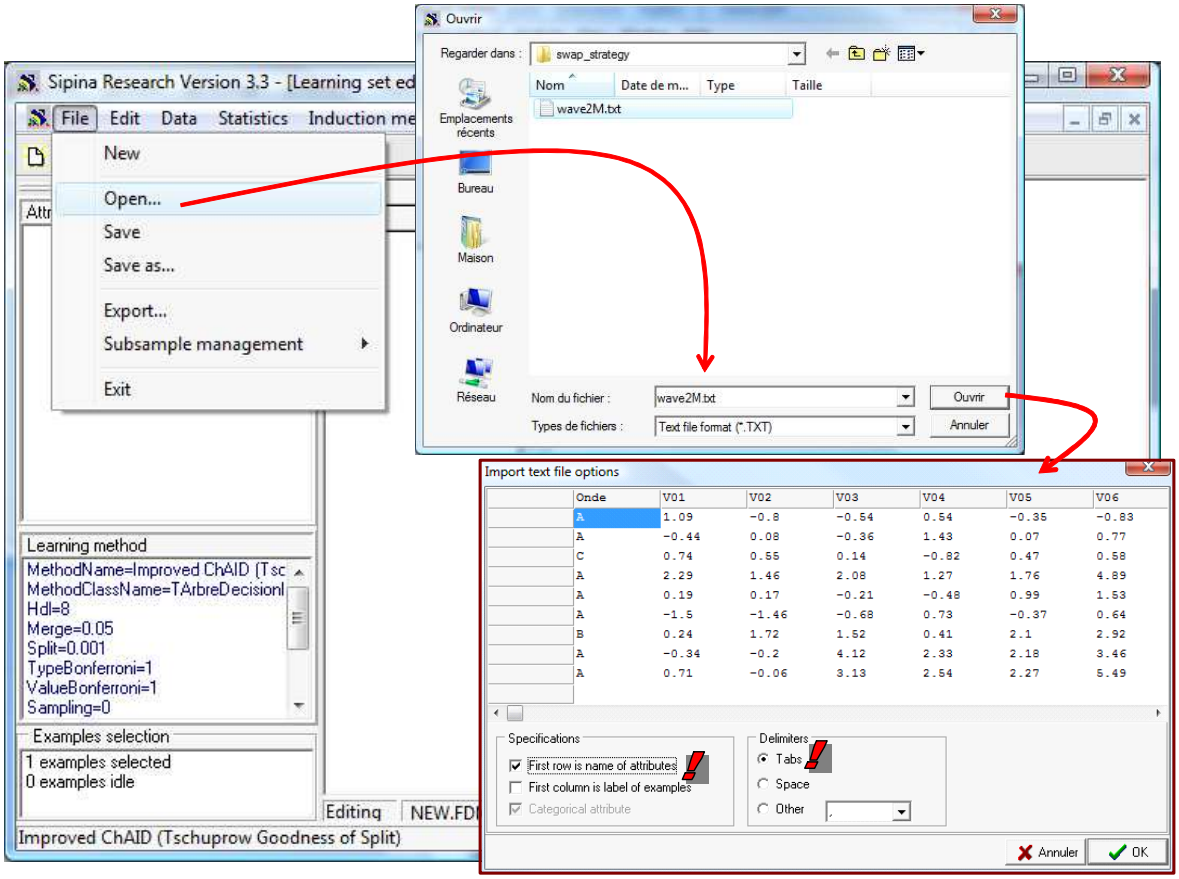
5.1 Traitement en mémoire – 5 variables prédictives



Pour connaître la configuration courante de Sipina, nous devons consulter le fichier SIPINA.INI. Lorsque l'option « buffered » est égale à 0, l'application charge toutes les données en mémoire centrale, sans création de fichiers temporaires.

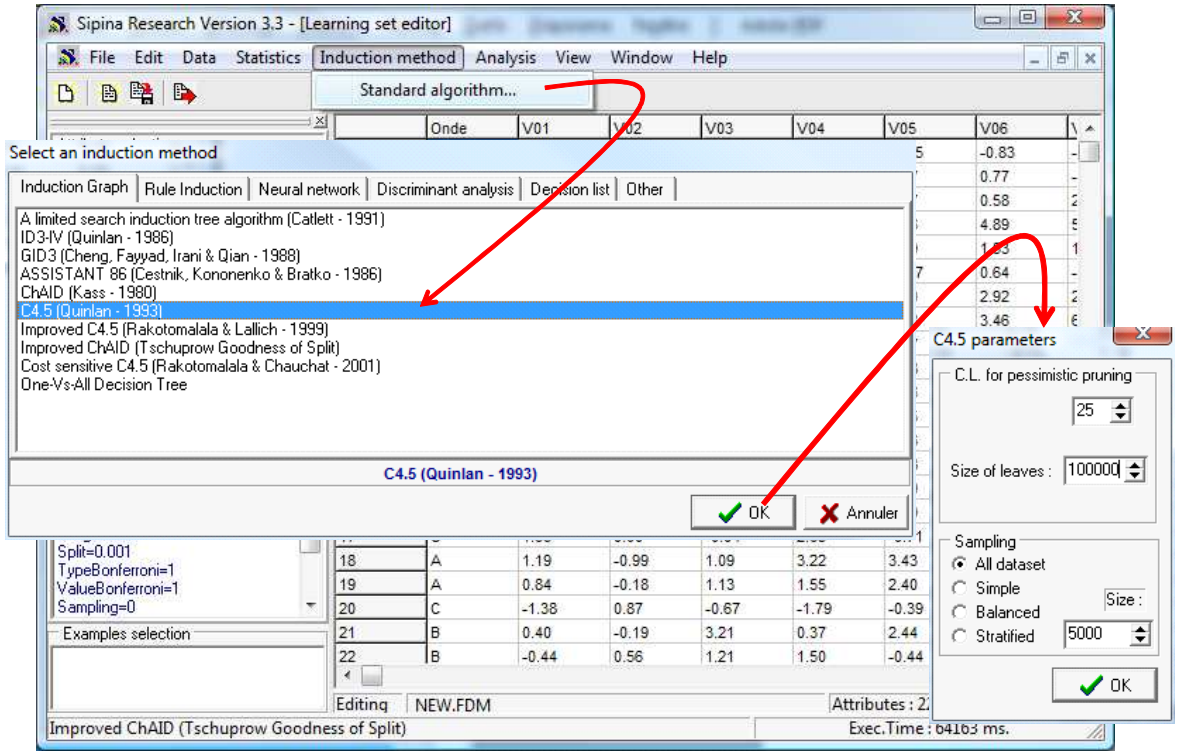
5.1.1 Importation des données

Nous démarrons SIPINA, nous actionnons le menu FILE / OPEN pour charger le fichier « wave2M.txt ». Une boîte de paramétrage apparaît, nous indiquons au logiciel la configuration du fichier.

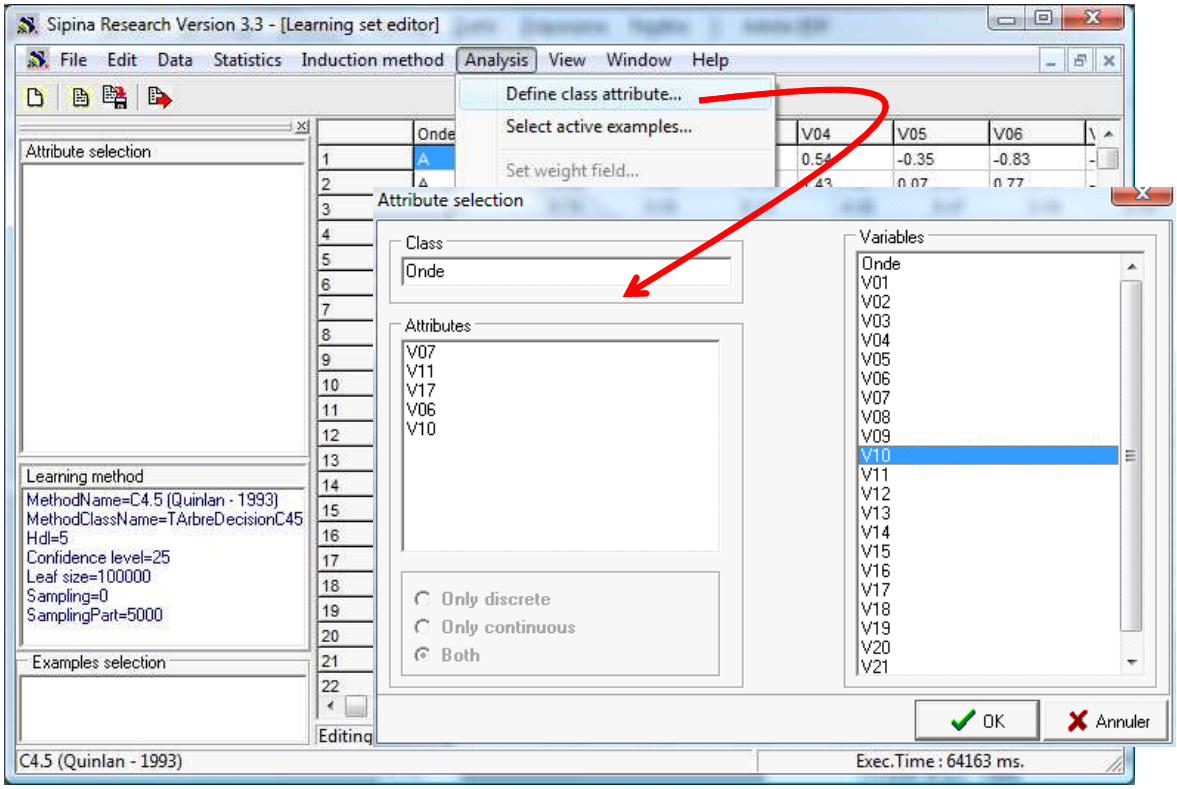


5.1.2 Arbre de décision – Prédiction

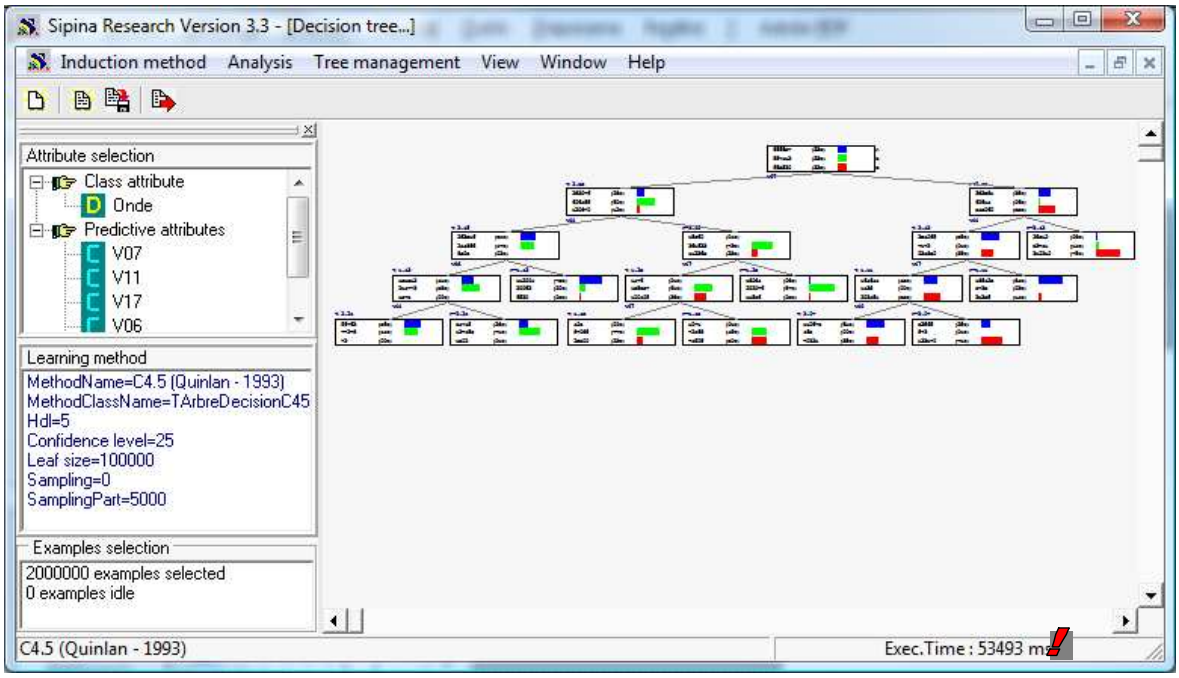
Nous utiliserons la méthode C4.5 avec un paramétrage rendant comparable les temps de calcul avec ceux de R. Pour ce faire, nous actionnons le menu INDUCTION METHOD / STANDARD ALGORITHM. Dans la boîte de dialogue qui apparaît, nous sélectionnons la méthode C4.5. Nous cliquons sur OK. Dans la boîte suivante, nous indiquons à l'algorithme que les feuilles doivent comporter au minimum 100.000 individus.



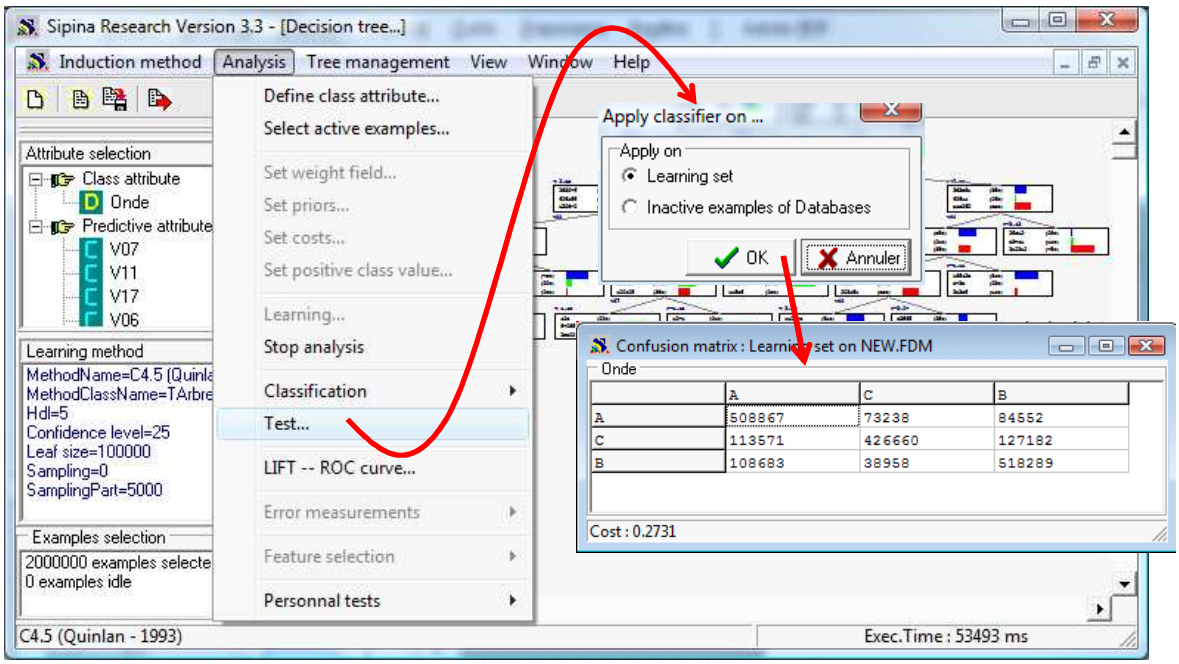
Ensuite, nous devons définir les variables de l'analyse. Nous actionnons le menu ANALYSIS / DEFINE CLASS ATTRIBUTE. Nous plaçons ONDE en CLASS ; V07, V11, V17, V06 et V10 en ATTRIBUTES.



Pour lancer les calculs, nous actionnons le menu ANALYSIS / LEARNING. L'arbre de décision apparaît au bout de 53 secondes.



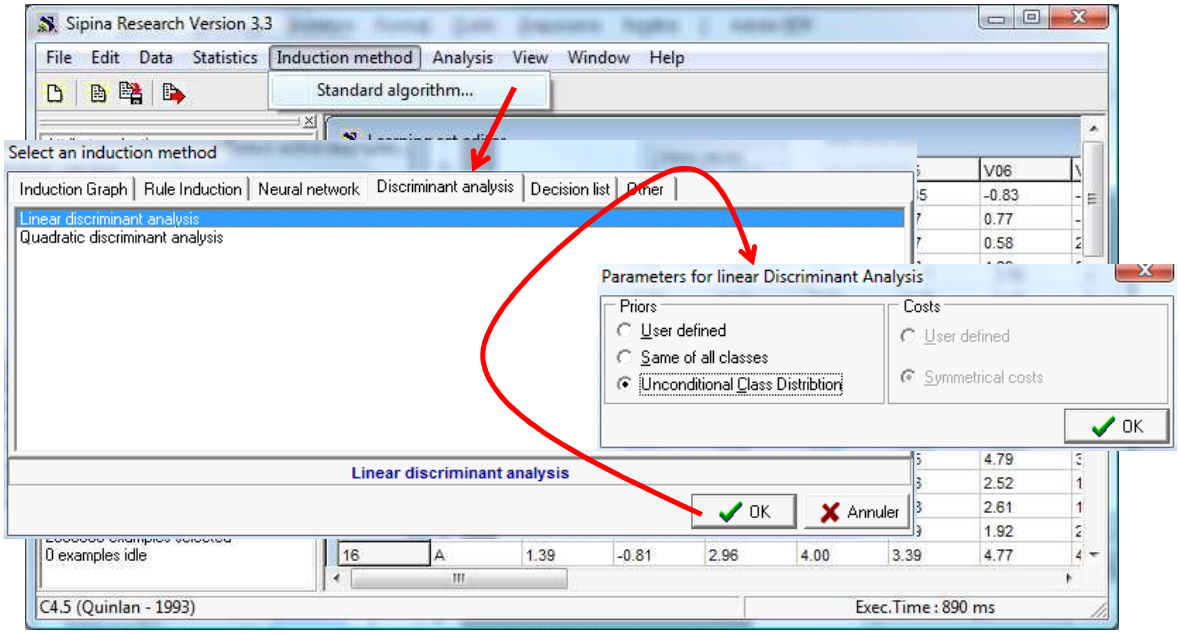
Enfin, pour calculer la matrice de confusion, nous actionnons le menu ANALYSIS / TEST. Nous sélectionnons les observations actives (LEARNING SET).



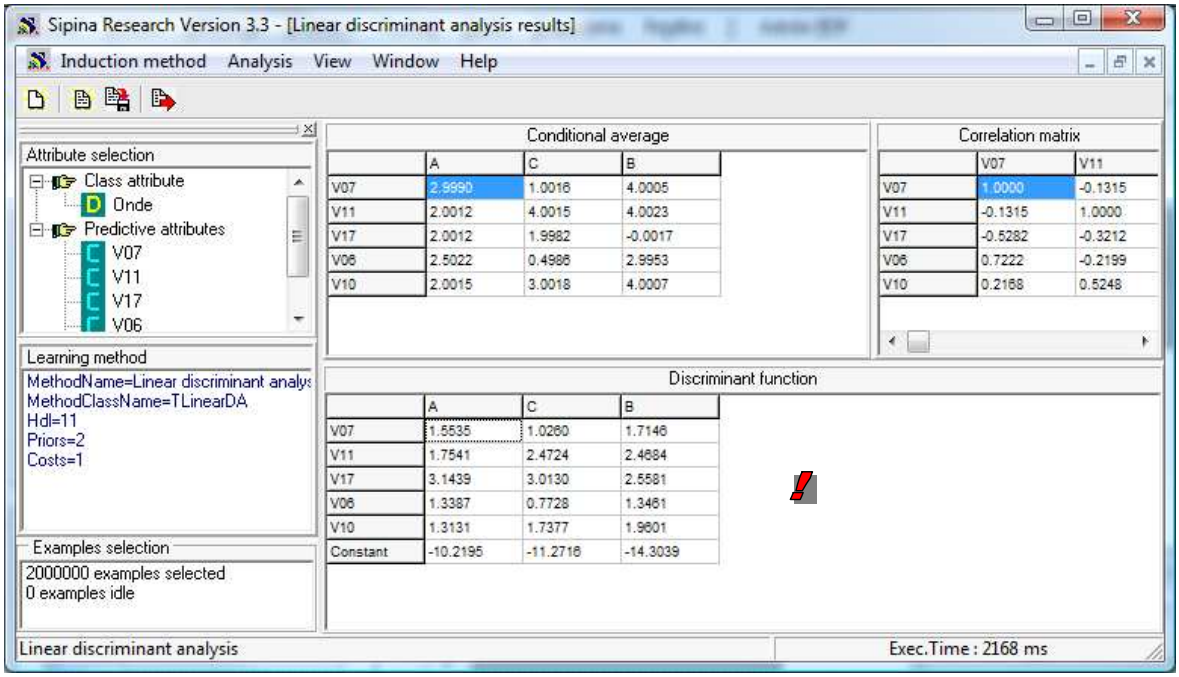
5.1.3 Analyse discriminante – Prédiction

Nous utilisons le même canevas pour l'analyse discriminante. Nous stoppons l'étude en cours en cliquant sur ANALYSIS / STOP ANALYSIS.

Puis nous modifions l'algorithme d'apprentissage en actionnant le menu INDUCTION METHOD / STANDARD ALGORITHM. Nous allons dans l'onglet DISCRIMINANT ANALYSIS cette fois, nous choisissons l'analyse discriminante linéaire. Nous validons les paramètres par défaut.

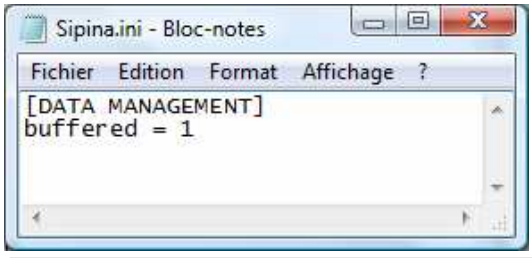


Pour relancer l'apprentissage, nous allons de nouveau sur ANALYSIS / LEARNING. La fenêtre d'affichage du modèle apparaît au bout de 2 secondes.



Nous réitérons les mêmes manipulations que précédemment pour l'obtention de la matrice de confusion.

5.2 Traitement avec swap – 5 variables prédictives



Nous refermons Sipina. De nouveau, nous ouvrons le fichier SIPINA.INI. Nous passons le paramètre « buffered » à 1. Ainsi, pour chaque colonne de données, Sipina crée un fichier temporaire sur disque. Seule une fenêtre des observations est disponible en

mémoire. Le système de cache a été optimisé pour les accès séquentiels. Notre idée était de privilégier la construction des arbres de décision.

Nous recommençons à l'identique l'analyse ci-dessus, nous retraçons ici les temps de calcul et l'occupation mémoire.

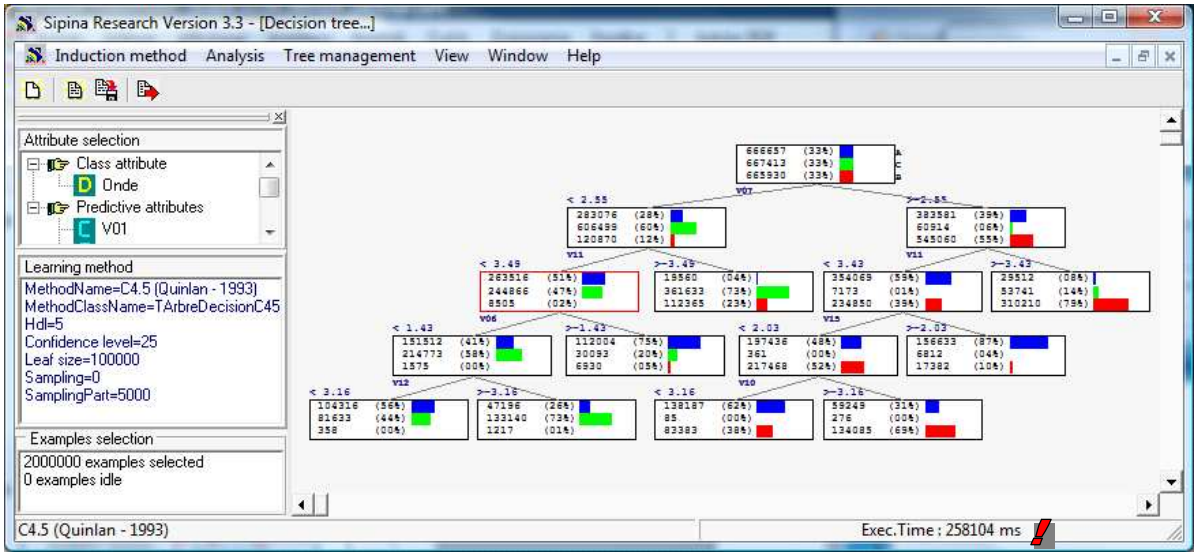
	Traitement en mémoire	Traitement sur disque
Apprentissage	Temps (sec.)	Temps (sec.)
Connexion/chargement données	64	67
Création arbre	53	64
Création analyse discriminante	2	3
Calcul matrices de confusion	3	3
Mémoire Windows après chargement des données (Mo)	229	19

Deux informations importantes sautent aux yeux. (1) L'occupation mémoire est tout autre lorsqu'on utilise des fichiers swap, au cas où on en douterait encore. (2) L'analyse discriminante sait tirer parti de notre implémentation. Après coup cela paraît évident, les données sont lues de manière séquentielle lorsque l'on calcule les matrices de variance covariance conditionnelles. La dégradation du temps de calcul devrait être autrement plus sensible si les individus étaient présentés dans un ordre aléatoire à l'algorithme de calcul, pour un réseau de neurones par exemple.

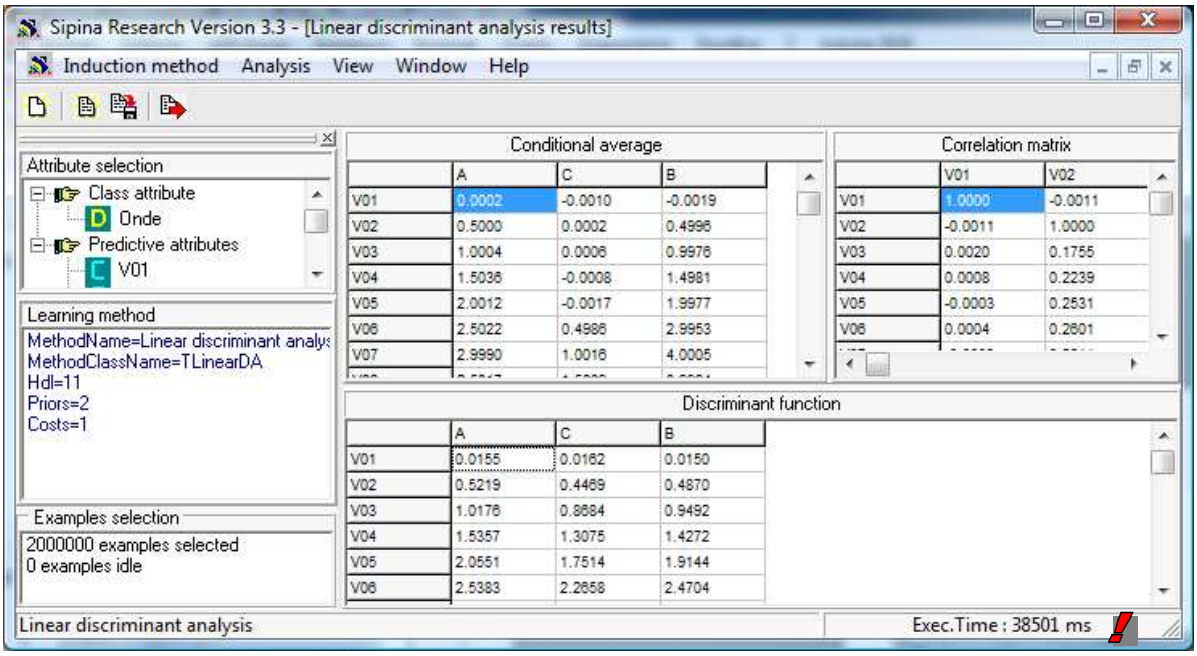
5.3 Traitement avec swap – Toutes les variables prédictives

Contrairement à R, lorsque nous avons voulu intégrer toutes les variables prédictives pour la construction du modèle, Sipina a fonctionné sans problèmes. Nous avons reproduit exactement la démarche ci-dessus, mais en incluant les variables V01 à V21 (ANALYSIS / DEFINE CLASS ATTRIBUTES).

Nous avons obtenu l'arbre suivant en 258 secondes.



Pour ce qui est de l'analyse discriminante, le calcul a duré 38 secondes.



6 Conclusion

Dans ce didacticiel, nous avons présenté le package « filehash » de R. Il permet de manipuler directement les données dans des fichiers binaires sur disque. L'espace mémoire ainsi libéré peut être dédié à l'exécution des algorithmes de data mining, décuplant ainsi les capacité de calcul. La bibliothèque est de surcroît particulièrement efficace. En effet, malgré les accès répétés au disque, la dégradation des temps de calcul est certes sensible mais pas rédhibitoire.

Néanmoins, nous avons constaté que lorsque nous augmentons sensiblement nos exigences, les fonctions (rpart et lda) issus des packages très populaires échouent, faute de mémoire (?). Cela laisse à penser qu'une adaptation de ces fonctions reste nécessaire lorsque nous souhaitons réellement appréhender de très grandes bases de données. « Filehash » n'est pas la solution miracle.

Remarque : Pour les connaisseurs de « filehash », j'ai essayé d'utiliser la commande **dumpDF()** pour sauver le data.frame dans un fichier binaire. Les variables sont stockées individuellement, colonne par colonne. Ainsi, il n'est pas nécessaire de manipuler toute la base à chaque fois que l'on veut accéder à la valeur d'une des variables en lecture (seule la colonne est manipulée).

Au final, la sauvegarde du data.frame sur disque est nettement plus rapide, quelques minutes au lieu de plus d'une heure. En revanche, les fonctions rpart ou lda ont encore échoué lors du traitement de l'ensemble des variables prédictives, *mais avec un message d'erreur différent*. Mes investigations se sont arrêtées à ce stade.