1 Objectif

Programmation multithread de l'analyse discriminante prédictive. Implémentation de la solution dans le logiciel SIPINA 3.10. Impacts sur les temps de traitement.

Le volume des données à traiter ne cesse d'augmenter dans le Data Mining. Avec le phénomène « Big Data », la gestion de la volumétrie devient un critère clé. Dans le même temps, nos machines personnelles sont de plus en plus puissantes, intégrant pour la très grande majorité d'entre eux plusieurs cœurs de calcul, si ce n'est plusieurs processeurs. Et pourtant, rares sont les logiciels généralistes qui exploitent pleinement les possibilités des machines modernes.

Car l'affaire est loin d'être simple. Il est impossible de définir une stratégie générique valable pour tout type de technique d'apprentissage. Mis à part des cas très spécifiques comme les méthodes ensemblistes (ex. random forest[1]), il faut opérer des modifications explicites dans l'organisation des calculs. Cela implique une étude attentive du processus afin de détecter les opportunités de parallélisation, et de transformer le programme en conséquence.

A l'instar d'un précédent document où nous détaillons une solution basée sur les threads pour l'induction des arbres de décision[2], nous proposons une nouvelle stratégie multithread pour l'analyse discriminante linéaire (prédictive) dans **SIPINA 3.10**. Cette méthode, très favorablement connue des statisticiens, est curieusement peu implémentée dans les logiciels d'obédience informatique. Alors que, sauf cas pathologiques que l'on sait identifier[3], elle présente des performances en prédiction comparables aux autres techniques linéaires sur la majorité des données, je pense notamment à la très populaire régression logistique (Saporta, 2006 – page 480 ; Hastie et al., janv. 2013 – page 128). Et, en termes de rapidité de traitements sur les grandes bases, elle est incomparablement plus rapide[4], plus encore lorsque nous exploitons l'architecture multi-cœur (ou multiprocesseur) des machines comme nous le constaterons durant nos expérimentations.

Pour mieux situer les améliorations induites par notre stratégie, nous comparons nos temps d'exécution avec des logiciels reconnus tels que SAS 9.3 (proc discrim), R (lda du package MASS) et Revolution R Community (une version « optimisée » de R).

2 Analyse discriminante linéaire prédictive

Nous ne décrivons que très succinctement l'algorithme de calcul dans cette section. Il existe d'excellentes références sur le web[5]. L'objectif est de prédire les valeurs d'une variable cible Y prenant ses valeurs dans {1, 2, …, K} à partir de p variables prédictives $X = (X_1, X_2, \ldots, X_p)$. Nous disposons d'un échantillon de taille n. Un individu est noté ω, sa valeur pour la variable cible s'écrit

[1] Ex. Le package « bigrf » pour les Random Forest sous R - http://cran.r-project.org/web/packages/bigrf/index.html ; pour RapidMiner http://m-core-wekaext.sourceforge.net/

[2] http://tutoriels-data-mining.blogspot.fr/2010/11/multithreading-pour-les-arbres-de.html

[3] http://tutoriels-data-mining.blogspot.fr/2013/05/classifieurs-lineaires.html

[4] Sur la base MIT FACE IMAGE – voir les expérimentations – la régression logistique de SAS 9.3 (proc logistic) fait 7 mn 08 sec quand l'analyse discriminante (proc discrimin) ne dure que 39,12 secondes !

[5] Ex sur Wikipédia, http://fr.wikipedia.org/wiki/Analyse_discriminante_linéaire

y(ω). Le nombre d'observations appartenant à la classe Y = k est égal à n_k. Dans ce qui suit, nous présentons les étapes susceptibles de représenter un enjeu fort en termes de temps de calcul :

1. Construction des **K** matrices de variance covariance conditionnelles de dimension (p x p).

$$S_k = \left(\frac{1}{n_k} \sum_{\omega : y(\omega)=k} \left[x_i(\omega) - \bar{x}_{i,k}\right] \times \left[x_j(\omega) - \bar{x}_{j,k}\right] \right)_{i,j=1,\cdots,p}$$

Où $\bar{x}_{i,k}$ représente la moyenne de la variable X_i pour les individus de la classe (Y = k).

2. Construction de la matrice de variance covariance intra-classes $S = \frac{1}{n-K} \sum_{k=1}^{K} n_k \times S_k$
3. Puis inversion de cette dernière (S^{-1}).

D'ores et déjà, l'étape (2) ne nécessite pas d'accès aux données. Elle ne pose aucun problème. De même, l'inversion de S à l'étape (3) n'est pas un souci non plus avec les bibliothèques de calcul actuelles, même lorsque 'p' est assez grand (plusieurs milliers)[6]. Reste donc l'étape (1).

Programmation monothread. Dans une programmation séquentielle monothread, le plus simple, et vraisemblablement le plus rapide, est d'effectuer une seule passe sur les données. Cela impose de maintenir en mémoire K matrices S_k de dimension (p x p). Un rapide calcul montre que l'occupation mémoire reste contenue même sur de grandes bases[7]. Il est possible de réduire encore leur encombrement puisque les S_k sont symétriques. De plus, cette stratégie présente l'incommensurable intérêt d'être parfaitement en accord avec notre stockage optionnel des données sur le disque lorsque nous traitons des bases ne pouvant pas tenir en mémoire centrale[8]. La dégradation des temps de traitement est quasiment imperceptible dans ce cas.

Programmation multithread. J'avoue avoir vraiment beaucoup cherché. Après moult essais (et plantages en tous genres), j'en suis venu à une solution simplifiée, voire simpliste. L'idée consiste à créer K index permettant d'associer chaque individu à sa classe d'appartenance (1er passage sur les données). Cela revient à partitionner les données en K blocs, mais sans avoir à les dupliquer en mémoire. Puis de lancer un thread par classes (2nd passage). **Les avantages** sont évidents. La transformation du programme séquentiel est très facile. La synchronisation est réduite à sa plus simple expression. Il suffit d'attendre que le dernier thread ait terminé avant de passer au calcul de la matrice S (étape 2). De plus, mis à part les index, l'organisation et l'encombrement mémoire du dispositif est identique à celui du programme séquentiel. C'était une des contraintes de mon cahier des charges. **Les inconvénients** sont – hélas – évidents également : (a) si le nombre de cœurs ou de processeurs est supérieur à K, le surplus n'est pas exploité[9] ; (b) lorsque les classes sont d'effectifs fortement déséquilibrées, le système est tributaire du calcul sur la modalité la plus représentée. Les

[6] Avec des libraires telles que LAPACK (http://www.netlib.org/lapack/), les temps de traitements sont vraiment époustouflants. C'est une piste possible si d'aventure l'étape (3) devenait un problème.

[7] Pour MIT FACE IMAGE, K = 2 et p = 361. L'occupation en double précision est [(361 x 361) x 8] x 2 ≈ 2 Mo.

[8] http://tutoriels-data-mining.blogspot.fr/2009/10/sipina-traitement-des-tres-grands.html et http://tutoriels-data-mining.blogspot.fr/2010/06/traitement-des-tres-grands-fichiers.html (section 5).

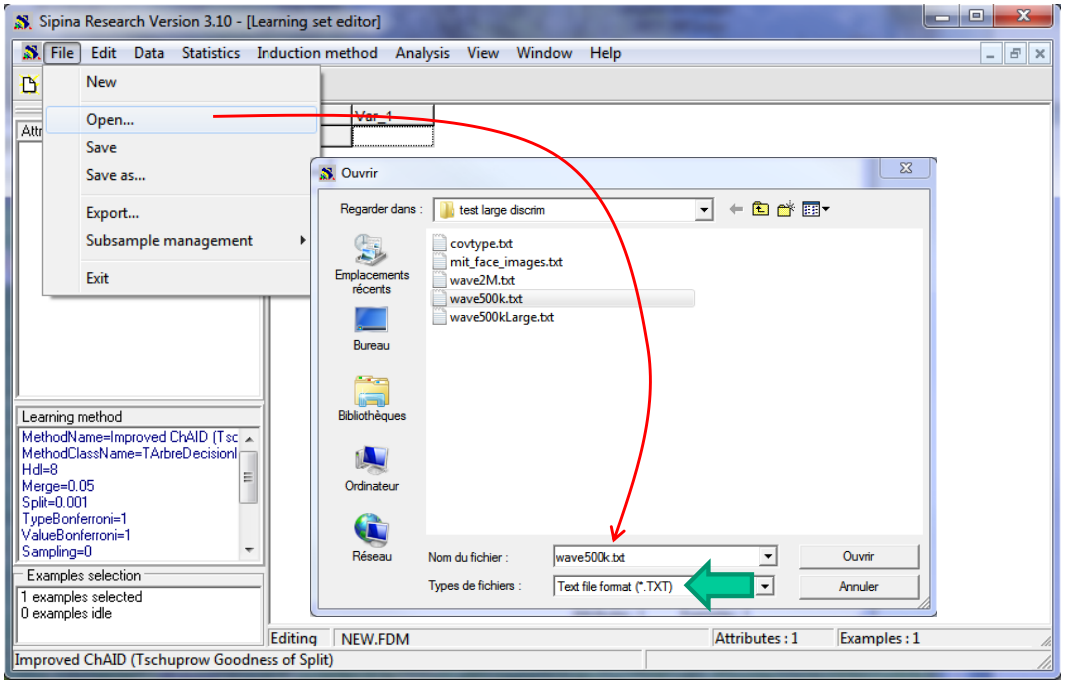
[9] A l'inverse, définir plus de threads que de cœurs (processeurs) n'engendre pas une dégradation des temps de traitement… tant qu'on reste raisonnable (16 threads max par cœur dixit la documentation de Delphi 6.0 que j'utilise).

expérimentations viendront confirmer ces intuitions lorsque nous mettrons en relation les performances avec les caractéristiques des données.

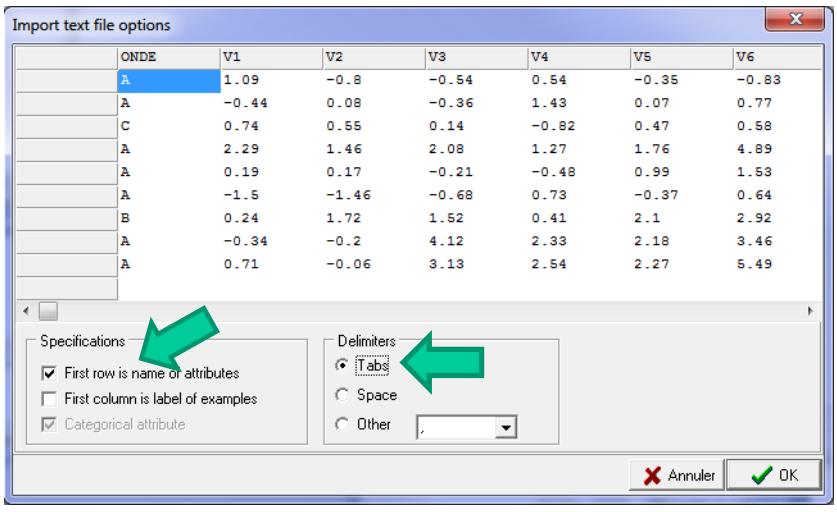
3 Analyse discriminante avec SIPINA

3.1 Importation des données

Le programme de Hastie et al. (http://www-stat.stanford.edu/~tibs/ElemStatLearn/data.html ; **waveform.S**)[10] nous a permis de générer la base WAVE500K avec n = 500.000 observations, p = 21 variables prédictives, et une cible à K = 3 modalités. Après avoir démarré SIPINA, nous actionnons le menu FILE / OPEN et nous sélectionnons le fichier WAVE500K.TXT au format texte.

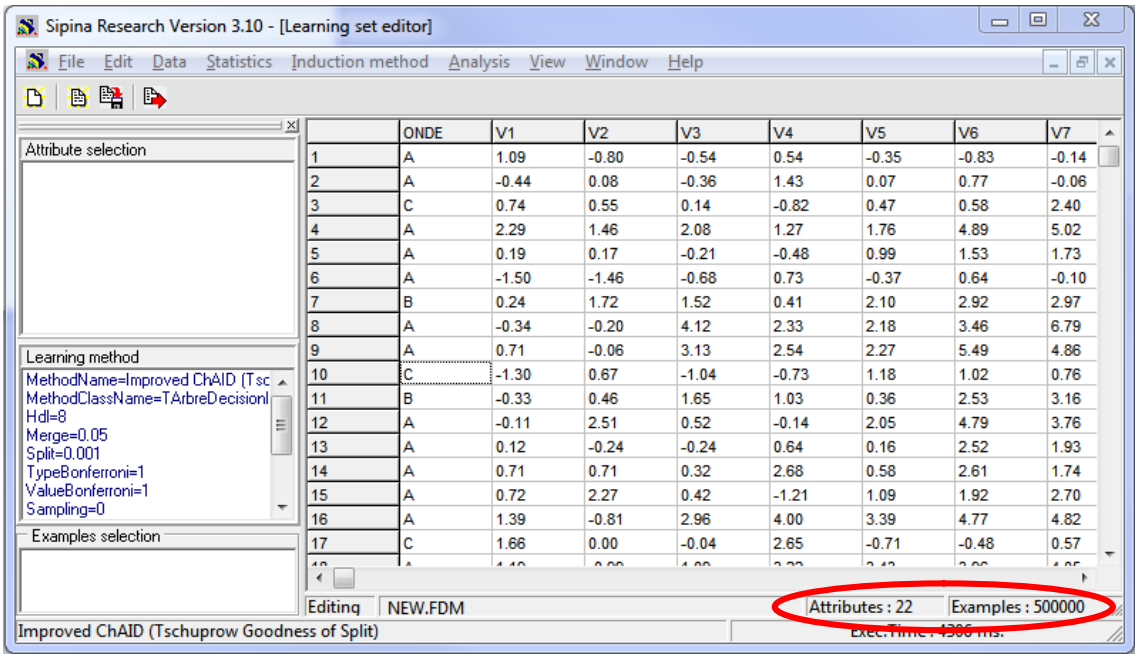


Un assistant apparaît. Il vous permet de définir les spécificités du fichier (type de séparateur, la première ligne correspond aux noms des variables).



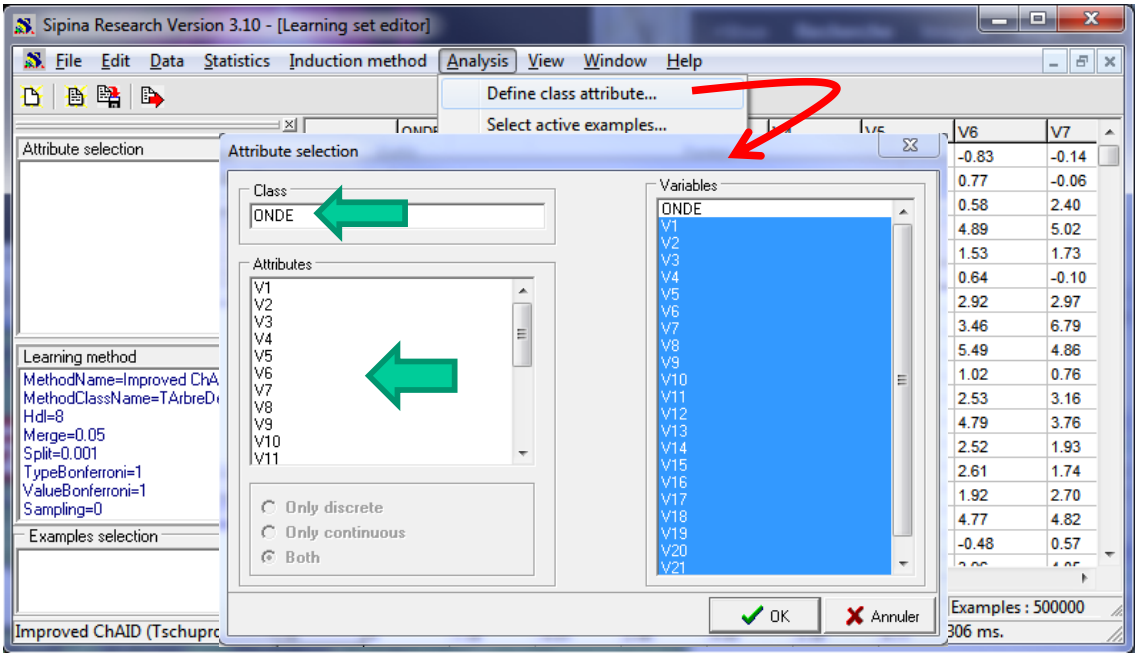
[10] Nous décrivons une version de ce programme http://tutoriels-data-mining.blogspot.fr/2012/01/regression-logistique-sur-les-grandes.html ; notre version permet de générer des variables supplémentaires.

Nous validons. Les valeurs apparaissent dans la grille principale de l'application.



3.2 Choix des variables

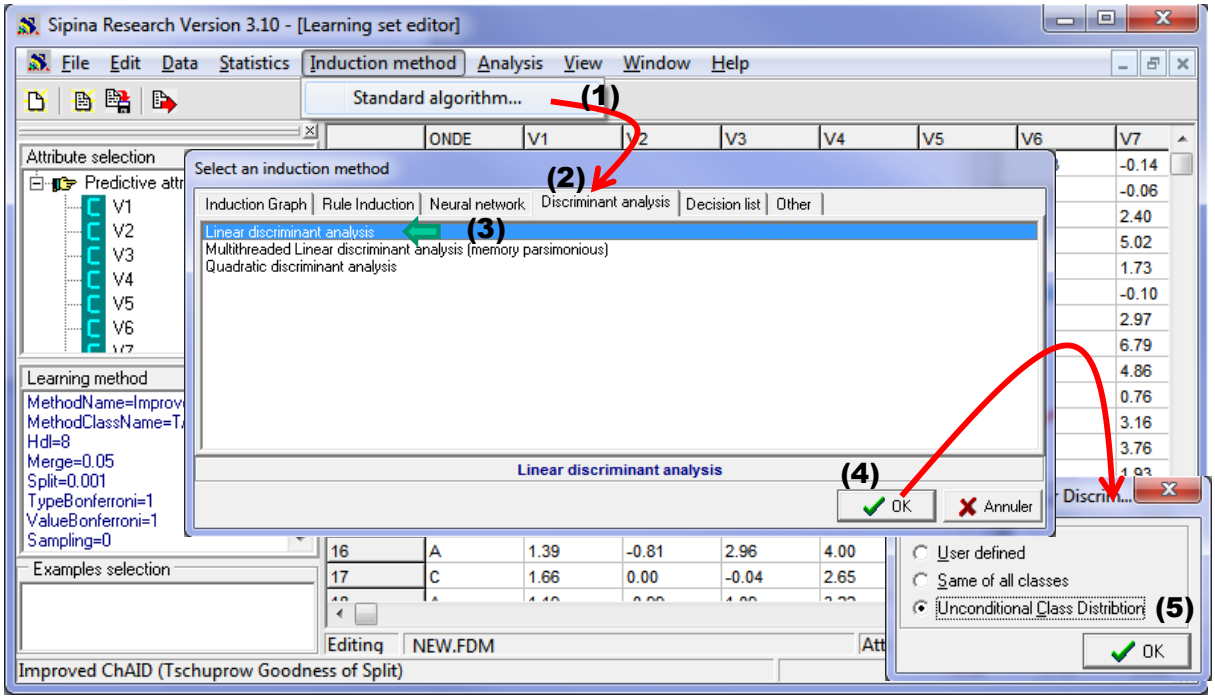
Pour définir le rôle des variables, nous actionnons le menu ANALYSIS / DEFINE CLASS ATTRIBUTE. Par « glisser-déposer », nous plaçons ONDE en CLASS, toutes les autres variables en ATTRIBUTES.



Nous validons en cliquant sur OK. Les variables sélectionnées apparaissent dans la partie gauche de la fenêtre principale. Le symbole « D » désigne une variable discrète (catégorielle), « C » une variable continue (quantitative).

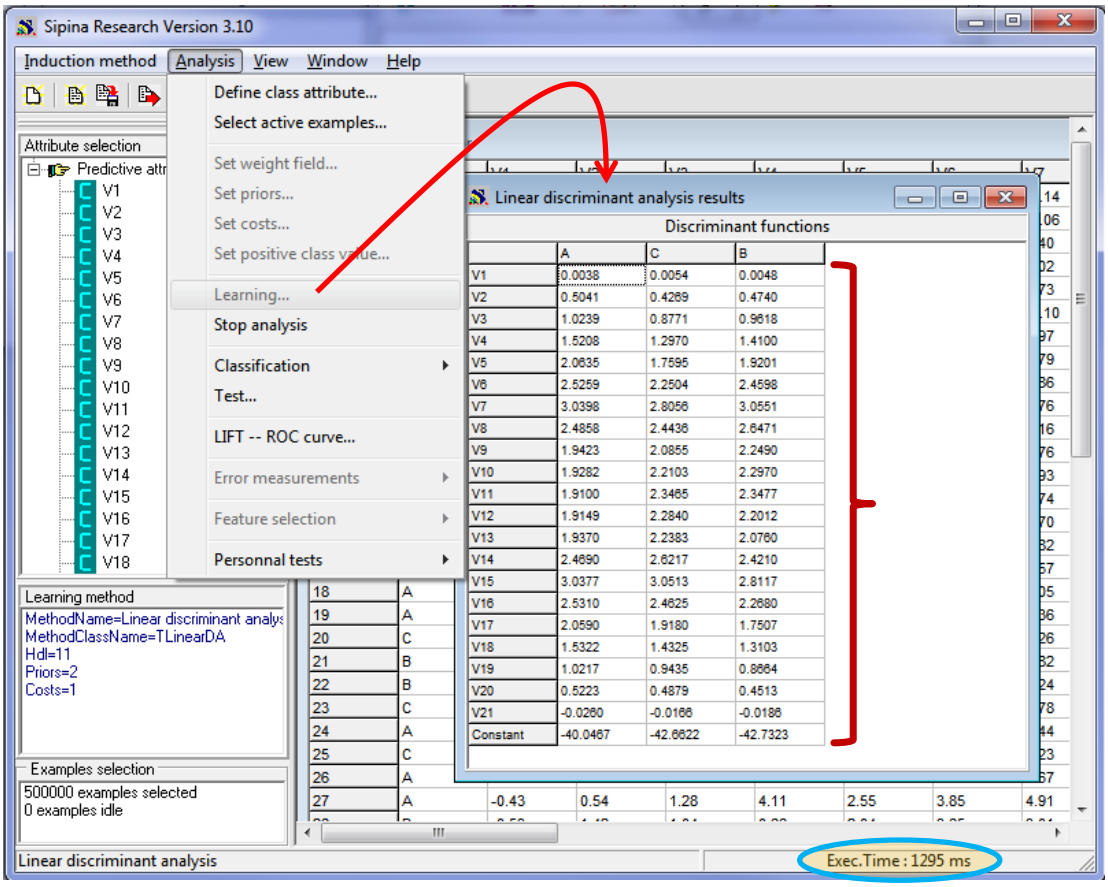
3.3 Algorithme séquentiel

Nous souhaitons modéliser à l'aide de l'analyse discriminante linéaire. Pour sélectionner la méthode, nous actionnons le menu INDUCTION METHOD / STANDARD ALGORITHM.



Dans l'onglet DISCRIMINANT ANALYSIS, nous avons choisi l'algorithme standard (monothread) LINEAR DISCIMINANT ANALYSIS. Nous validons les paramètres par défaut (les probabilités a priori sont estimées à partir des fréquences des classes).

Il ne nous reste plus qu'à cliquer sur ANALYSIS / LEARNING pour lancer les calculs. Au bout de **1295 millisecondes**, les fonctions discriminantes sont affichées dans une nouvelle fenêtre.



A titre de comparaison, nous mettons côte à côte les coefficients fournis par SIPINA et SAS. Ils sont strictement identiques.

Label	A	B	C
Constant	-40.04668	-42.73233	-42.66217
V1	0.00377	0.00476	0.00541
V2	0.50408	0.47400	0.42693
V3	1.02392	0.96181	0.87710
V4	1.52079	1.41004	1.29700
V5	2.06354	1.92008	1.75948
V6	2.52591	2.45984	2.25037
V7	3.03977	3.05506	2.80563
V8	2.48582	2.64713	2.44356
V9	1.94234	2.24896	2.08549
V10	1.92816	2.29702	2.21027
V11	1.91005	2.34766	2.34649
V12	1.91492	2.20122	2.28404
V13	1.93703	2.07603	2.23833
V14	2.46897	2.42098	2.62170
V15	3.03766	2.81172	3.05126
V16	2.53097	2.26796	2.46251
V17	2.05902	1.75070	1.91801
V18	1.53216	1.31029	1.43251
V19	1.02175	0.86640	0.94353
V20	0.52232	0.45131	0.48785
V21	-0.02602	-0.01864	-0.01655

SIPINA

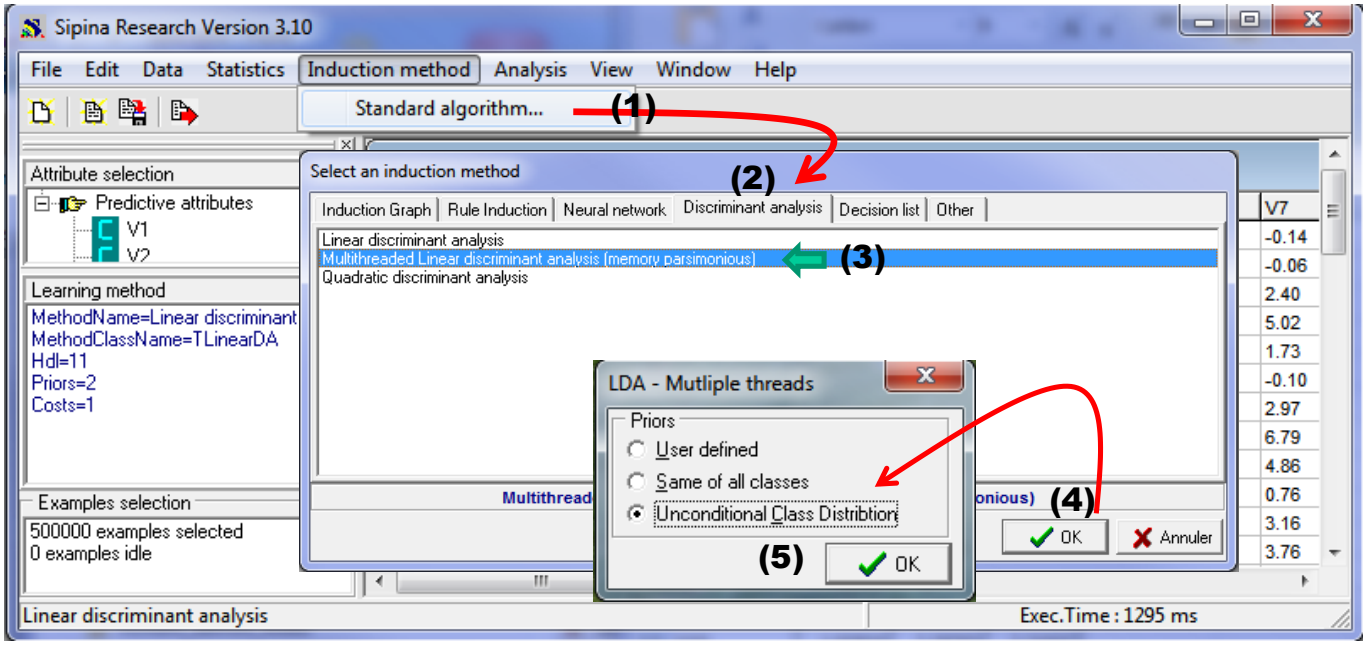
Linear Discriminant Function for ONDE

Variable	A	B	C
Constant	-40.04668	-42.73233	-42.66217
V1	0.00377	0.00476	0.00541
V2	0.50408	0.47400	0.42693
V3	1.02392	0.96181	0.87710
V4	1.52079	1.41004	1.29700
V5	2.06354	1.92008	1.75948
V6	2.52591	2.45984	2.25037
V7	3.03977	3.05506	2.80563
V8	2.48582	2.64713	2.44356
V9	1.94234	2.24896	2.08549
V10	1.92816	2.29702	2.21027
V11	1.91005	2.34766	2.34649
V12	1.91492	2.20122	2.28404
V13	1.93703	2.07603	2.23833
V14	2.46897	2.42098	2.62170
V15	3.03766	2.81172	3.05126
V16	2.53097	2.26796	2.46251
V17	2.05902	1.75070	1.91801
V18	1.53216	1.31029	1.43251
V19	1.02175	0.86640	0.94353
V20	0.52232	0.45131	0.48785
V21	-0.02602	-0.01864	-0.01655

SAS

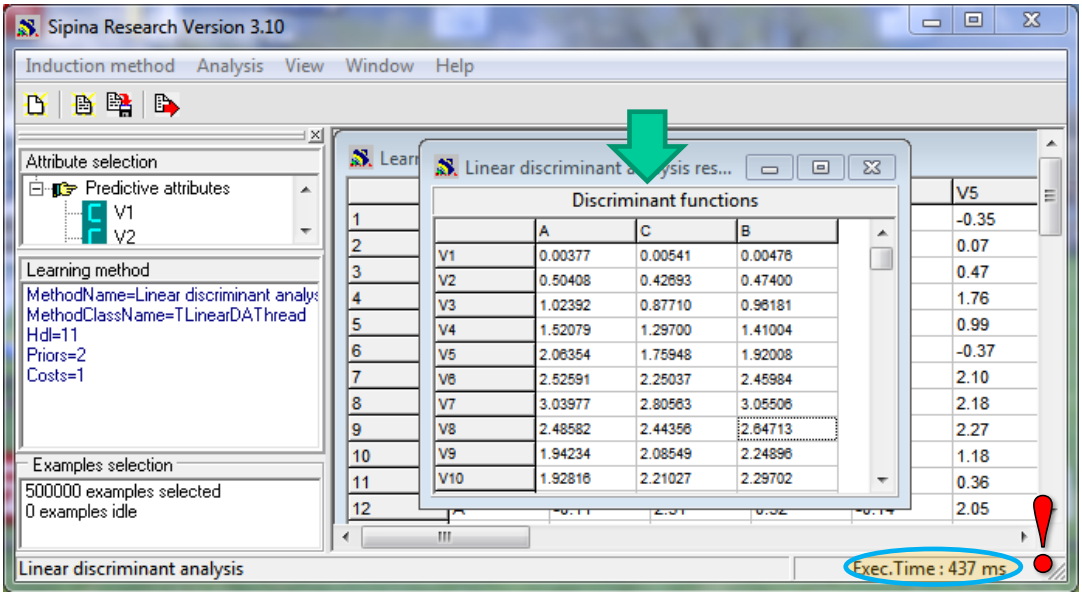
3.4 Algorithme multithread

Pour accéder à l'algorithme multithread, nous devons d'abord stopper l'analyse existante (ANALYSIS / STOP ANALYSIS), puis sélectionner la méthode (INDUCTION METHOD / STANDARD ALGORITHM…).

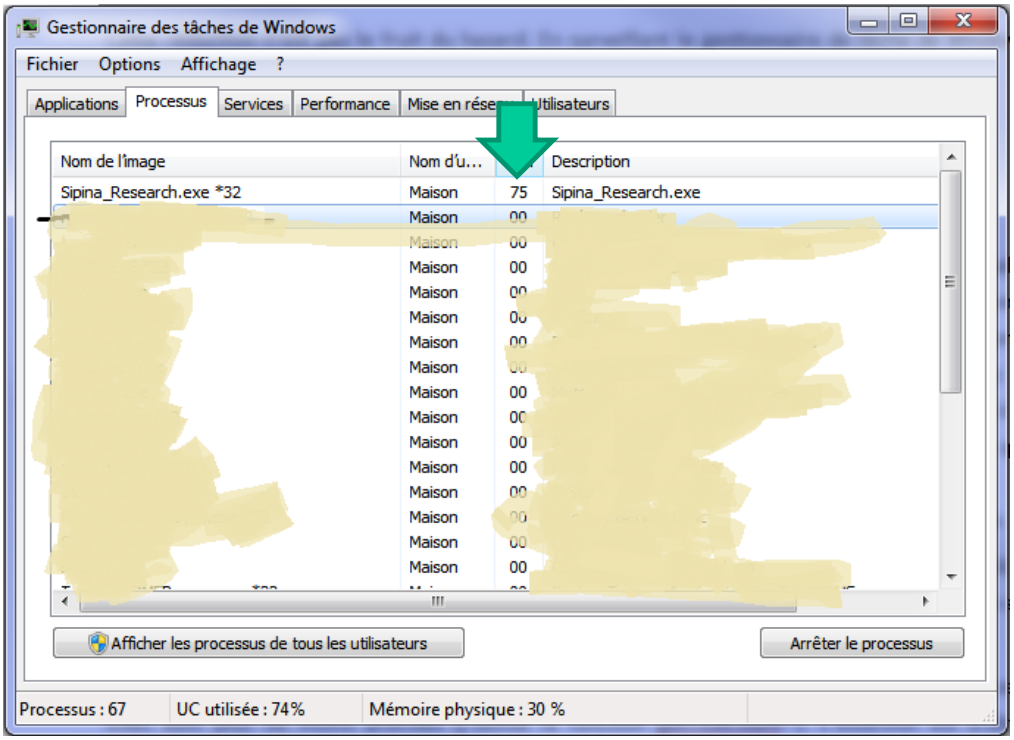


Nous choisissons MULTITHREADED LINEAR DISCRIMINANT ANALYSIS (MEMORY PARSIMONIOUS) dans l'onglet DISCRIMINANT ANALYSIS. Les paramètres sont identiques à ceux de l'approche usuelle.

Nous lançons de nouveau les calculs avec le menu ANALYSIS / LEARNING. Nous obtenons les mêmes coefficients que précédemment (heureusement), mais le temps de calcul est passé à **437 ms**.



Cette réduction n'est pas le fruit du hasard. Notre machine est équipée d'un processeur Q9400 Quad-Core (4 cœurs). En surveillant le gestionnaire de tâche de Windows, nous constatons que K = 3 cœurs parmi les 4 (75%) sont exploités durant les calculs.



3.5 Expérimentations sur plusieurs bases

Pour évaluer le comportement de la solution multithread face à l'implémentation séquentielle, nous avons mesuré les temps de traitements sur différentes bases. Pour la série de WAVE, le nombre de classes est toujours K = 3, et surtout elles sont équilibrées (les effectifs par classes sont quasi-identiques) : WAVE500K que nous avons décrit ci-dessus ; WAVE500KLARGE, toujours avec 500.000

observations mais avec p = 121 descripteurs (100 variables supplémentaires par rapport à la base originelle) ; WAVE2M avec n = 2.000.000 et p = 21. Ce sont toutes des bases artificielles que nous pouvons faire évoluer à notre guise. L'idée est d'étudier l'impact de l'évolution du nombre d'observations et du nombre de descripteurs par rapport à la référence WAVE500K ci-dessus.

Nous avons ensuite traité les bases réelles COVTYPE avec K = 7 classes, mais assez déséquilibrées (2 modalités de la variable cible concentrent une grande partie des observations) ; et MIT FACE IMAGE, binaire (K = 2), avec des effectifs très déséquilibrés.

Nous recensons dans le tableau ci-dessous les temps de traitement. Les durées sont en secondes. Elles sont plus ou moins précises (j'utilise la fonction getTickCount[11]). L'essentiel est d'obtenir un ordre d'idée sur l'évolution des opérations[12]. « Ratio » indique la réduction du temps d'exécution lors du passage au multithread (ex. 3.0 = 1.30 / 0.44). Notre machine d'expérimentation étant un Quad-Core, le meilleur ratio possible est 4 quelle que soit la base traitée. Lorsque K ≤ 4, il serait égal à K.

Dataset	K	n	p	SIPINA (std)	SIPINA (threads)	Ratio
Wave 500k	3	500000	21	1.30	0.44	**3.0**
Wave 500k Large	3	500000	121	19.19	6.13	**3.1**
Wave 2M	3	2000000	21	5.16	1.83	**2.8**
Covtype	7	581012	52	5.30	2.67	**2.0**
Face Images	2	513455	361	142.73	135.46	**1.1**

Ces résultats nous inspirent plusieurs pistes de réflexion :

- Le passage par les threads améliore toujours les temps de traitement. Le double passage sur les données (construction des index + threads pour le calcul des matrices S_k) n'est pas pénalisant.

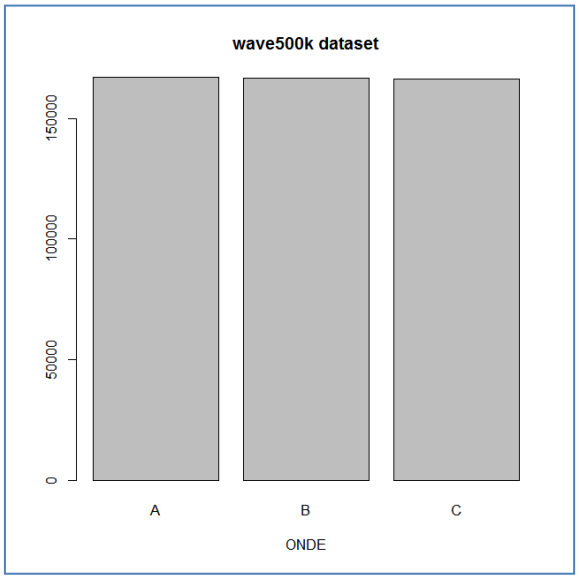


Figure 1 - Répartition des classes pour la base "wave500K"

[11] http://msdn.microsoft.com/en-us/library/windows/desktop/ms724408%28v=vs.85%29.aspx

[12] En toute rigueur, il aurait fallu également répéter l'expérimentation est calculer la moyenne des durées d'exécution.

- Par rapport à WAVE500K, la solution est bénéfique lorsque qu'augmente le nombre de variables (WAVE500KLarge) ou le nombre d'observations (WAVE2M). Il faut dire que nous sommes dans une configuration particulièrement favorable ici. D'une part, le nombre de classes K = 3 est proche du nombre de cœurs de ma machine (4 cœurs). D'autre part, les effectifs sont équilibrés (Figure 1). Les cœurs sont utilisés conjointement jusqu'au terme des calculs des matrices conditionnelles. De fait, nous divisons par un facteur proche de 3 le temps de traitement puisque K = 3 cœurs sont mobilisés.
- La situation est plus contrastée quand il s'agit de traiter la base COVTYPE. Pourtant, avec K = 7 classes, nous devrions exploiter pleinement le potentiel de la machine. Mais le ratio de réduction est de 2. On le comprend mieux en étudiant la répartition des classes (Figure 2). Les performances dépendent du traitement des modalités « spruce » et « lodgepole » qui vont monopoliser 2 cœurs. Le calcul de la matrice intra-classes **S** (étape 2 du processus) n'est possible que lorsque « lodgepole » est finalisée.

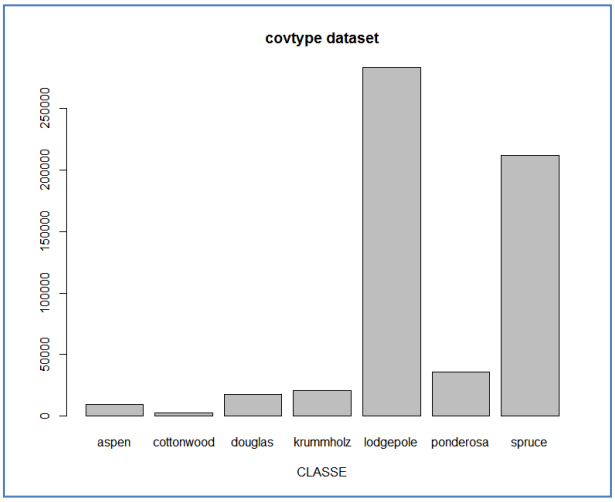


Figure 2 - Répartition des classes pour la base "covtype"

- Le bilan est franchement décevant pour la base MIT_FACE_IMAGE. La réduction est très faible. On comprend mieux lorsque l'on constate qu'il y a seulement K = 2 classes, et surtout qu'elles sont très fortement déséquilibrées (Figure 3).

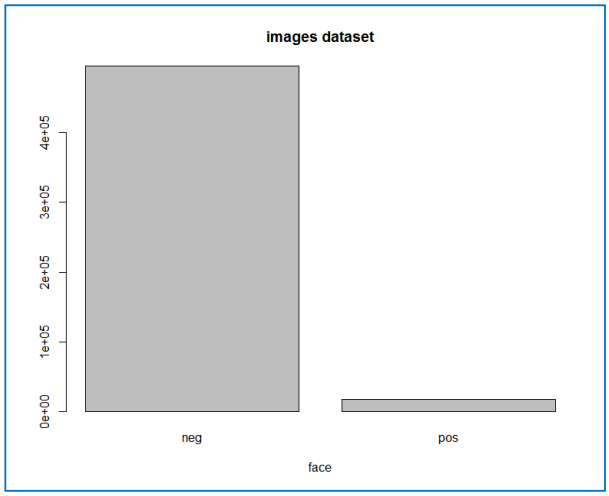


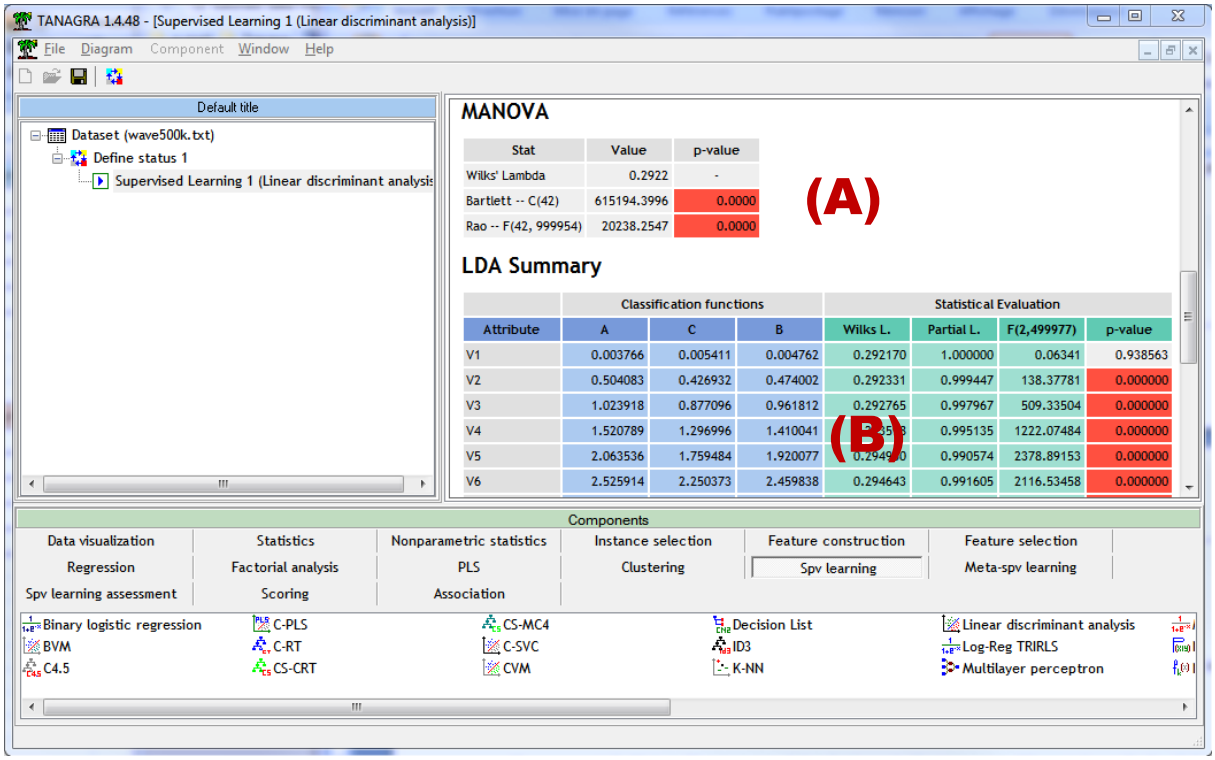
Figure 3 - Répartition des classes pour la base "mit face images"

Conclusion. Mieux exploiter les capacités de la machine, ici en passant par une meilleure utilisation des cœurs disponibles, ne peut être que bénéfique en termes de performance. Dans tous les cas de figure, nous avons amélioré les durées de traitement, y compris dans la configuration la plus défavorable (MIT_FACE_IMAGE). Mais il est clair également qu'on peut faire mieux. Nous en reparlerons dans la conclusion générale.

4 Comparaison avec d'autres logiciels

4.1 Comparaison avec Tanagra

L'analyse discriminante prédictive est disponible dans Tanagra. Nous montrons ci-dessous une copie d'écran du logiciel lors du traitement de la base WAVE500K[13].



Par rapport à Sipina, Tanagra produit des indicateurs supplémentaires qui sont gourmands en calculs : **(A)** le lambda de Wilks permet d'évaluer la significativité globale du modèle, il implique le calcul de la matrice de variance covariance globale, puis des déterminants ; **(B)** les lambda partiels qui permettent d'évaluer la pertinence des variables, là aussi plusieurs déterminants de matrices doivent être calculés. Ainsi, la durée des traitements est de 5.25 sec pour WAVE500K. Tanagra est presque 10 fois plus lent que SIPINA en ce qui concerne l'analyse discriminante linéaire. On sait pourquoi maintenant.

4.2 Comparaison avec SAS, R et Revolution R

Nous avons testé les performances de SAS 9.3 (proc discrim), de R 3.0.0 (64 bits) et Revolution R Community 6.2.0 (64 bits). Cette dernière est une variante optimisée de R. Il semble qu'elle sache

[13] Voir http://tutoriels-data-mining.blogspot.fr/2012/07/analyse-discriminante-lineaire.html pour l'importation des données et la construction du diagramme de traitements dans Tanagra. De nombreux autres tutoriels sont disponibles sur ce site.

tirer parti des processeurs multi-cœur durant les calculs matriciels. L'analyse discriminante en bénéficie dixit le site de l'éditeur[14]. Pour ces outils, nous nous sommes contentés d'une analyse à minima, produisant uniquement les coefficients des modèles.

Pour SAS, voici le code utilisé. Rien d'optionnel n'est demandé. Je ne connais pas en revanche les objets intermédiaires que SAS produit en interne, je ne sais s'ils ont un impact sur les durées d'exécution :

```
proc discrim data = mesdata.wave500k;
  class onde;
  priors proportional;
run;
```

Pour R et Revolution R, nous utilisons la procédure **lda()** du package MASS :

```
wave500k <- read.table(file="wave500k.txt",header=T,dec=".",sep="\t")
system.time(model.1 <- lda(ONDE ~ ., data = wave500k))
print(model.1)
```

Nous recensons les durées d'exécution dans le tableau suivant. Nous avons placés les logiciels de gauche à droite selon leurs performances :

Dataset	K	n	p	SIPINA (threads)	SAS	SIPINA (std)	Revol. R 6.2.0	R 3.0.0
Wave 500k	3	500000	21	**0.44**	1.65	1.30	33.89	37.68
Wave 500k Large	3	500000	121	**6.13**	9.09	19.19	187.29	264.89
Wave 2M	3	2000000	21	**1.83**	6.19	5.16	137.05	148.42
Covtype	7	581012	52	**2.67**	4.29	5.30	67.29	84.69
Face Images	2	513455	361	135.46	**39.12**	142.73	ERR	ERR

Nous constatons plusieurs résultats :

- Visiblement, passer par des threads est bénéfique. SIPINA multithread devance SAS dans 4 configurations sur 5.
- La base « MIT FACE IMAGE » est particulière. Déjà, notre implémentation ne tire pratiquement pas parti du multithread parce que les classes sont très déséquilibrées (section 3.5). De plus, c'est la base qui comporte le plus de colonnes. Or, nous avions déjà constaté dans d'autres comparatifs que SIPINA (et TANAGRA puisqu'ils reposent sur la même philosophie) est très avantagé pour le traitement colonne par colonne des données (typiquement les arbres de décision[15]). Il est très mal à l'aise en revanche lorsqu'il s'agit de les parcourir en ligne (ex. les SVM[16]). Or c'est ce qu'il se passe pour l'analyse discriminante. Il n'est pas étonnant dès lors que les performances soient désastreuses par rapport à SAS. Pour nous, il ne s'agit surtout pas de modifier les structures internes en fonction des méthodes. Il faut accepter des compromis. Notons cependant que les calculs ont été menés à leur terme contrairement à R et Revolution R.
- Les logiciels R et Revolution R semblent très en retrait. C'est parce qu'ils reposent sur une autre stratégie de calcul en réalité. Les fonctions discriminantes sont déduites des facteurs produits par

[14] http://www.revolutionanalytics.com/why-revolution-r/benchmarks.php

[15] http://tutoriels-data-mining.blogspot.fr/2011/10/arbres-de-decision-sur-les-grandes.html (section 6).

[16] http://tutoriels-data-mining.blogspot.fr/2008/10/svm-comparaison-de-logiciels.html (section 3.5).

l'analyse factorielle discriminante (Venables & Ripley, 2002 ; page 334). La procédure lda() s'appuient sur la décomposition en valeur singulière durant l'apprentissage. De fait, les temps d'exécution ne sont pas directement comparables.

- Revolution R Community est systématiquement plus rapide que R, assurément. Mais les écarts ne sont pas aussi spectaculaires qu'annoncés sur le site de l'éditeur. Je l'avais déjà noté dans un précédent comparatif[17].
- Revolution R Community et R ont planté lors du traitement de la base MIT FACE IMAGES. Les deux outils ont affiché la même erreur : « Reached total allocation of 8127Mb ». Il faudrait augmenter la mémoire qui leur est allouée pour dépasser cette limitation.

A cause de ses structures internes, Sipina est moins performant que SAS lorsque le nombre de variables est élevé (ex. 'Face Image', 'Wave500K Large' en mono-thread). A contrario, à plus forte raison avec la stratégie multithread, Sipina se démarque nettement lorsque n >> p (ex. 'Wave2M'). SIPINA et SAS sont largement plus rapides que R et Revolution R Community sur les bases testés. Tout simplement parce que les algorithmes utilisés ne sont pas les mêmes vraisemblablement.

5 Conclusion

La parallélisation des algorithmes de data mining n'est pas un domaine nouveau. De très nombreux travaux existent. Mais il s'agit souvent de solutions élaborées spécifiquement pour certaines architectures. Elles sortent rarement des laboratoires et ne parviennent pas aux logiciels grand public. Notons également que de très nombreux efforts sont menés sous R pour le « calcul haute performance » et la parallélisation[18]. Mais à de très rares exceptions près[19], il s'agit le plus souvent de mise en place d'environnements permettant de programmer des algorithmes parallèles plutôt que de modifications de méthodes existantes.

Dans ce tutoriel, nous avons mis en avant une solution légère pour l'analyse discriminante. Elle tire parti du surcroit de performances que proposent les processeurs multi-cœurs installés dans les ordinateurs grand public. Par rapport à un programme séquentiel, l'organisation et l'encombrement mémoire des structures de calcul sont inchangés. C'est un atout pour le traitement des très grandes bases. Nous l'avons implanté dans SIPINA 3 .10. Les expérimentations montre que, pour simple et perfectible que soit notre solution, elle permet de réduire dans d'importantes proportions les temps d'exécution, notamment pour certaines configurations des données.

Après, la porte n'est certainement pas fermée. On doit pouvoir faire mieux encore. Deux pistes me paraissent intéressantes, encore faut-il pouvoir y parvenir : (1) exploiter pleinement le potentiel de la machine en utilisant tous les cœurs disponibles (le dispositif doit être paramétrable selon les caractéristiques de la machine) ; (2) mieux équilibrer les charges c.-à-d. niveler les durées de traitement sur les cœurs.

[17] http://tutoriels-data-mining.blogspot.fr/2012/04/revolution-r-community-50.html

[18] http://cran.r-project.org/web/views/HighPerformanceComputing.html

[19] Signalons le package SPRINT (http://cran.r-project.org/web/packages/sprint/index.html) qui propose des implémentations parallèles des SVM, du calcul de la matrice des corrélations, ... Un prochain tutoriel en perspective.

6 Références

Hastie T., Tibshirani R., Friedman J., « The Elements of Statistical Learning », Springer, 10th printing, January 2013.

Saporta G., « Probabilités, Analyse de Données et Statistique », Technip, 2006.

Venables W.N., Ripley B.D., « Modern Applied Statistics with S », Springer, 2002.