1 Topic

Load balanced multithreaded algorithm for linear discriminant analysis (LDA). The number of threads used is a parameter of the method. This new approach is incorporated in Sipina 3.11.

In a previous paper, we described a multithreading strategy for the linear discriminant analysis[1]. The aim was to take advantage of the multicore processors of the recent computers. We noted that for the same memory occupation than the standard implementation, we can decrease dramatically the computation time according to the dataset characteristics. The solution however had two drawbacks: the number of cores used was dependent on the number of classes K of the dataset; the load of the cores depended on classes' distributions. For instance, for one of dataset with K = 2 highly unbalanced classes, the gain was negligible compared to the single-threaded version.

In this paper, we present a new approach for the multithreaded implementation of the linear discriminant analysis, available in **Sipina 3.11**. It allows to overcome the two bottlenecks of the previous version. The capacity of the machine is fully used. More interesting, the number of used threads (cores) becomes customizable, allowing the user to adapt the machines resources used to process the database. But this is not without consideration. The memory occupation is increased. It depends on both the characteristics of the data and the number of cores that we want to use.

To evaluate the improvement introduced in this new version, we use various benchmark datasets to compare its computation time with those of the previous multithreaded approach, the single-threaded version, and the state-of-the-art proc discrim of SAS 9.3.

2 Linear discriminant analysis

There are many references which describe the linear discriminant analysis on the web (e.g. https://onlinecourses.science.psu.edu/stat505/node/89). It deals with a classification problem. It aims to assign the instances described by a set of *p* quantitative measurements ($X_1, X_2, \ldots, X_p$) to a predefined group described by a categorical variable Y. There are K groups {1, 2, …, K}. We dispose of a learning sample of size *n* for the construction of the model. Let ω an instance, $y(\omega)$ correspond to the class value of this instance. The absolute frequency of the class *k* is n_k.

In our previous paper, we have established that the computation of the **K** conditional covariance matrices (p x p) is the most resource-intensive step of the process.

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

Where $\bar{x}_{i,k}$ corresponds to the mean of X_i for the individuals belonging to the group (Y = k).

Single-threaded implementation. For the single-threaded strategy, the fastest way to implement the approach is to make the calculations in a single pass over the dataset. For that, we need to maintain in memory several objects. With double precision values (8 bytes), we have:

[1] Tanagra Tutorials, « Multithreading for linear discriminant analysis », may 2013.

- « 8 x (p x K) » for the vector of conditional means ;
- « 8 x [(p x p) x K] »[2] for the conditional covariance matrices S_k.

All in all, we have:

$$8 \times [(p \times K) + (p \times p) \times K]$$

$$= 8 \times K \times p \times (p + 1)$$

For instance, for the MIT FACE IMAGES dataset with K = 2 classes and p = 361 descriptors:

$$8 \times 2 \times 361 \times (361 + 1) \approx 2 \text{ Mo}$$

Multithreaded implementation (memory parsimonious). In Sipina 3.10, we create first K index vectors which allow to associate each instance to its group membership. Then, we start K threads for the computation of the covariance matrices S_k. Thus, apart from the index vectors that can be stored on disk, the memory occupation and the objects used for the calculations are the same compared to the single-threaded version.

Multithread implementation (load balancing). **We want to subdivide the calculations in M units, independently to the number K of classes**. To do that, we describe below one of the distinctive feature of the covariance matrix. The entries of S_k can be rewritten as follows:

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

$$= \left(\frac{1}{n_k} \sum_{\omega : y(\omega) = k} x_i(\omega) x_j(\omega) - \frac{1}{n_k^2} \sum_{\omega : y(\omega) = k} x_i(\omega) \times \sum_{\omega : y(\omega) = k} x_j(\omega) \right)_{i,j=1,\ldots,p}$$

All entries are additive! In fact, it is possible to create arbitrarily M groups of individuals regardless of their belonging to the classes, perform the calculations in parallel, to carry out consolidations for each group "k" prior to the calculation of the conditional average and covariance. But the objects for the calculations are duplicated M times. The memory occupation becomes:

$$\mathbf{M \times} [8 \times K \times p \times (p + 1)]$$

Thus, with M = 4 threads for the MIT FACE IMAGES dataset, the memory occupation for the calculations is:

$$4 \times [8 \times 2 \times 361 \times (361 + 1)] \approx 8 \text{ Mo}$$

It remains quite reasonable.

The interest of this strategy is that: (1) we can set the value of M according to the resources that we want to devote to the calculations; (2) by sending the same number of instances (n / M) for each thread, the utilization of the cores is perfectly balanced.

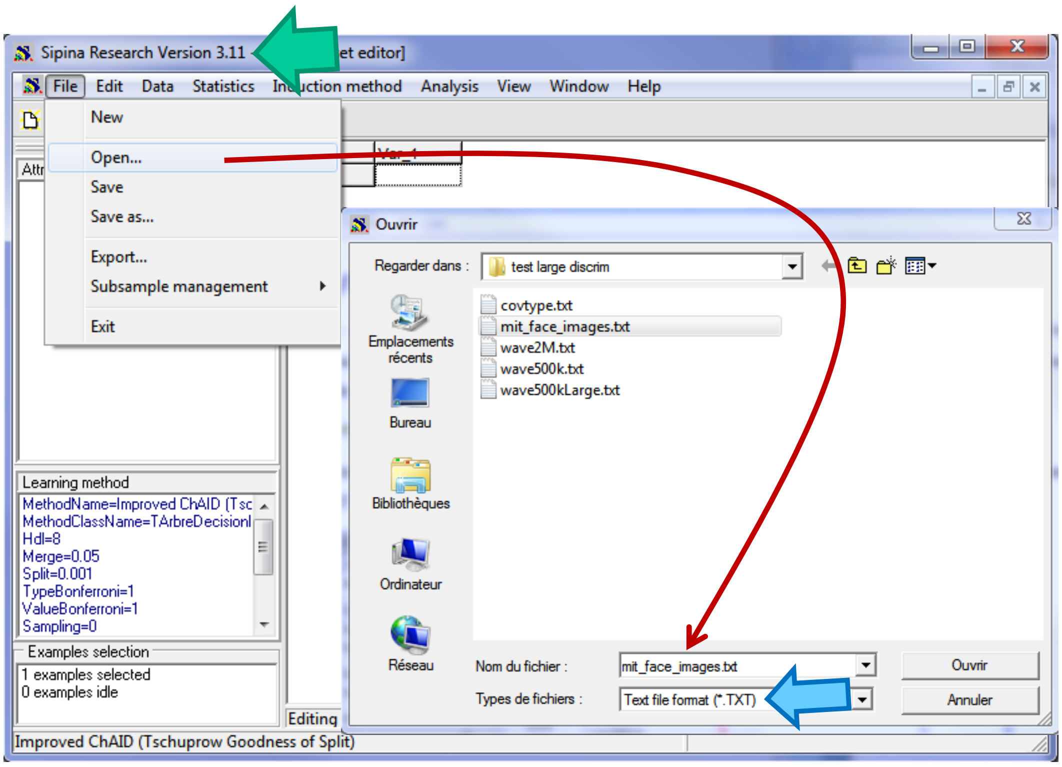
[2] Considering the fact that the matrix is symmetric, we can reduce the size of S_k to $[\frac{p \times (p+1)}{2} \times 8]$.

3 LDA with SIPINA

The use of the single-threaded and the memory parsimonious multithreaded approaches is described in the previous tutorial[3]. We focus on the load balanced version available in **Sipina 3.11** here.

3.1 Importing the dataset

We want to process the MIT FACE IMAGE dataset. First, we import the data file (TXT, tab delimited text file format).

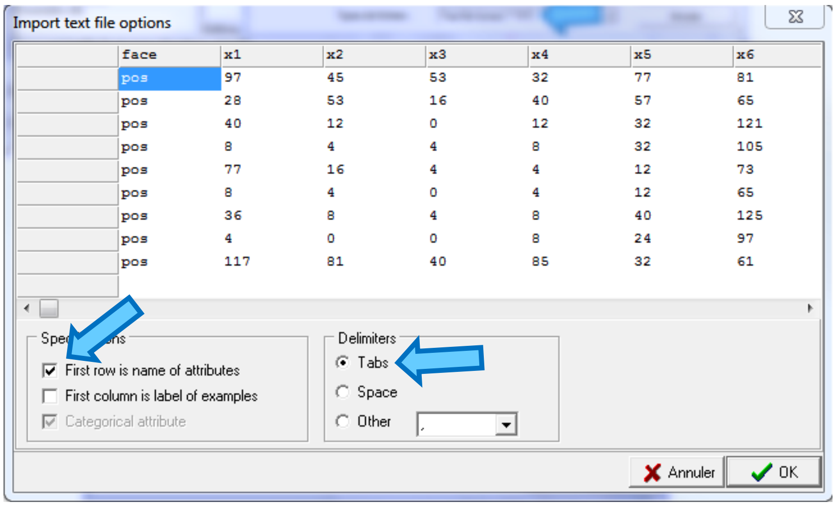


Learning method
MethodName=Improved ChAID (Tsc
MethodClassName=TArbreDecisionI
Hdl=8
Merge=0.05
Split=0.001
TypeBonferroni=1
ValueBonferroni=1
Sampling=0

Examples selection
1 examples selected
0 examples idle

Improved ChAID (Tschuprow Goodness of Split)

We specify the settings into the wizard. We validate.



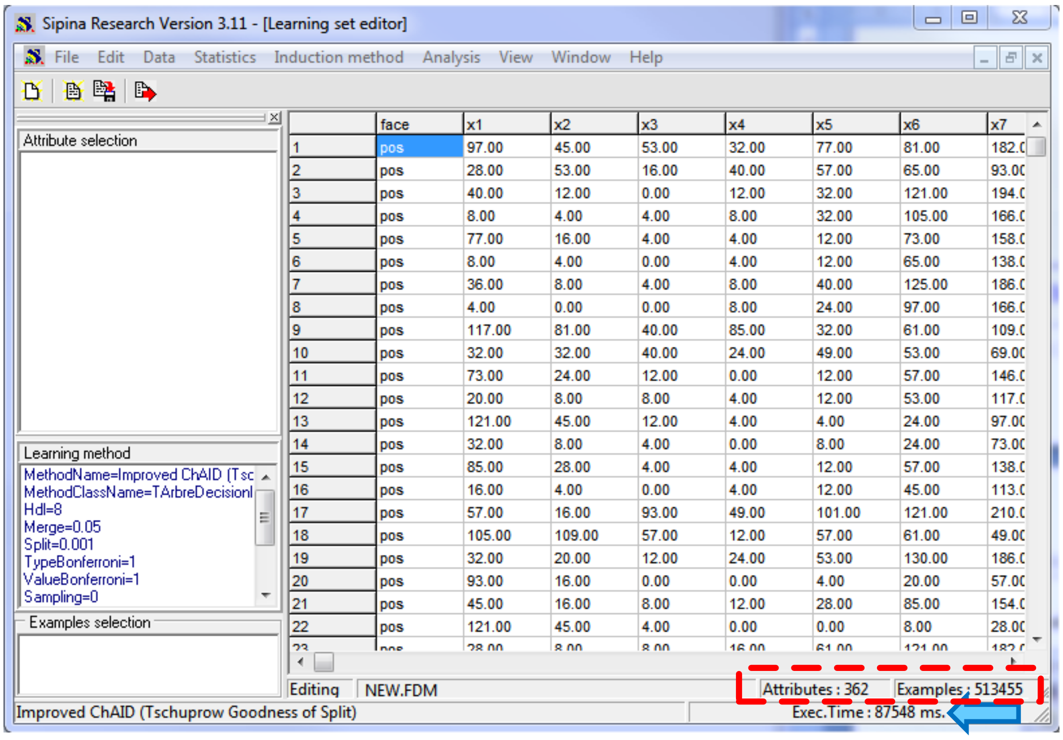
	face	x1	x2	x3	x4	x5	x6
	pos	97	45	53	32	77	81
	pos	28	53	16	40	57	65
	pos	40	12	0	12	32	121
	pos	8	4	4	8	32	105
	pos	77	16	4	4	12	73
	pos	8	4	0	4	12	65
	pos	36	8	4	8	40	125
	pos	4	0	0	8	24	97
	pos	117	81	40	85	32	61

Specifications
- [x] First row is name of attributes
- [ ] First column is label of examples
- [x] Categorical attribute

Delimiters
[✓ Tabs] Space Other

The duration of the processing of the 513,455 individuals and the 362 variables is 87 seconds.

[3] Tanagra Tutorials, « Multithreading for linear discriminant analysis », may 2013.


Sipina Research Version 3.11 - [Learning set editor]

Attribute selection

Learning method

MethodName=Improved ChAID (Tsc
MethodClassName=TArbreDecisionI
Hdl=8
Merge=0.05
Split=0.001
TypeBonferroni=1
ValueBonferroni=1
Sampling=0

Examples selection

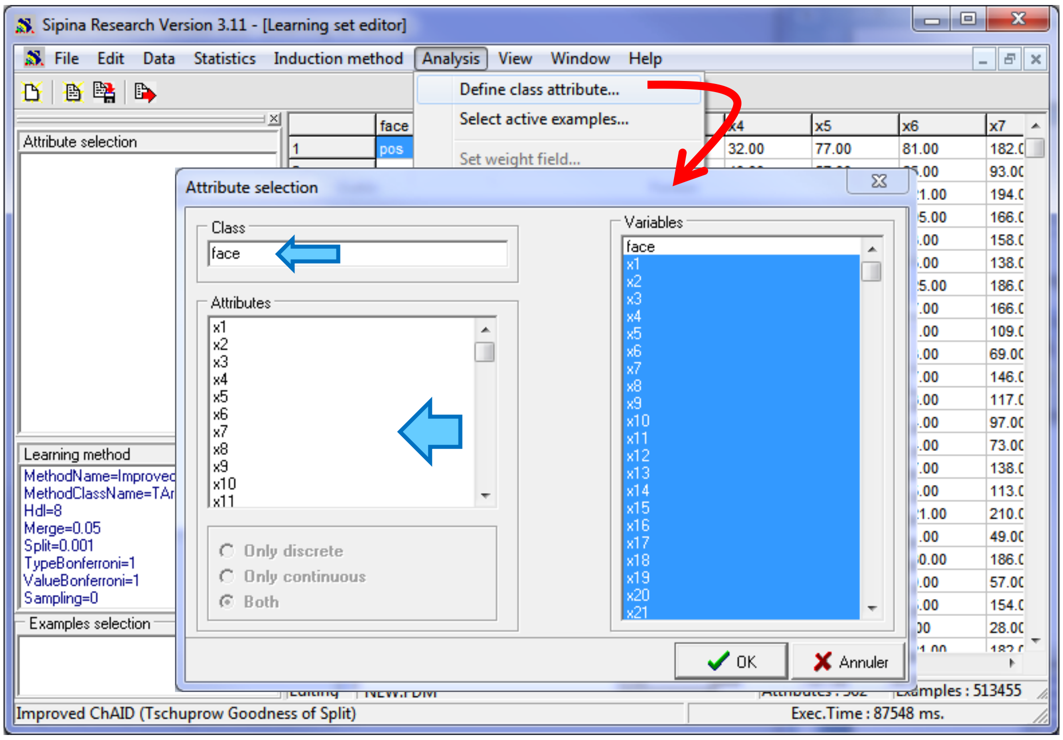
	face	x1	x2	x3	x4	x5	x6	x7
1	pos	97.00	45.00	53.00	32.00	77.00	81.00	182.
2	pos	28.00	53.00	16.00	40.00	57.00	65.00	93.0
3	pos	40.00	12.00	0.00	12.00	32.00	121.00	194.
4	pos	8.00	4.00	4.00	8.00	32.00	105.00	166.
5	pos	77.00	16.00	4.00	4.00	12.00	73.00	158.
6	pos	8.00	4.00	0.00	4.00	12.00	65.00	138.
7	pos	36.00	8.00	4.00	8.00	40.00	125.00	186.
8	pos	4.00	0.00	0.00	8.00	24.00	97.00	166.
9	pos	117.00	81.00	40.00	85.00	32.00	61.00	109.
10	pos	32.00	32.00	40.00	24.00	49.00	53.00	69.0
11	pos	73.00	24.00	12.00	0.00	12.00	57.00	146.
12	pos	20.00	8.00	8.00	4.00	12.00	53.00	117.
13	pos	121.00	45.00	12.00	4.00	4.00	24.00	97.0
14	pos	32.00	8.00	4.00	0.00	8.00	24.00	73.0
15	pos	85.00	28.00	4.00	4.00	12.00	57.00	138.
16	pos	16.00	4.00	0.00	4.00	12.00	45.00	113.
17	pos	57.00	16.00	93.00	49.00	101.00	121.00	210.
18	pos	105.00	109.00	57.00	12.00	57.00	61.00	49.0
19	pos	32.00	20.00	12.00	24.00	53.00	130.00	186.
20	pos	93.00	16.00	0.00	0.00	4.00	20.00	57.0
21	pos	45.00	16.00	8.00	12.00	28.00	85.00	154.
22	pos	121.00	45.00	4.00	0.00	0.00	8.00	28.0

Editing NEW.FDM Attributes : 362 Examples : 513455

Improved ChAID (Tschuprow Goodness of Split) Exec.Time : 87548 ms.

3.2 Specifying the role of the variables

To define the role of the variables, we click on the ANALYSIS / DEFINE CLASS ATTRIBUTE menu. By drag-and-drop, we set FACE as class-attribute, and all the others as predictive attributes.


Sipina Research Version 3.11 - [Learning set editor]

Attribute selection

Class: face

Attributes: x1 x2 x3 x4 x5 x6 x7 x8 x9 x10 x11

Variables: face x1 x2 x3 x4 x5 x6 x7 x8 x9 x10 x11 x12 x13 x14 x15 x16 x17 x18 x19 x20 x21

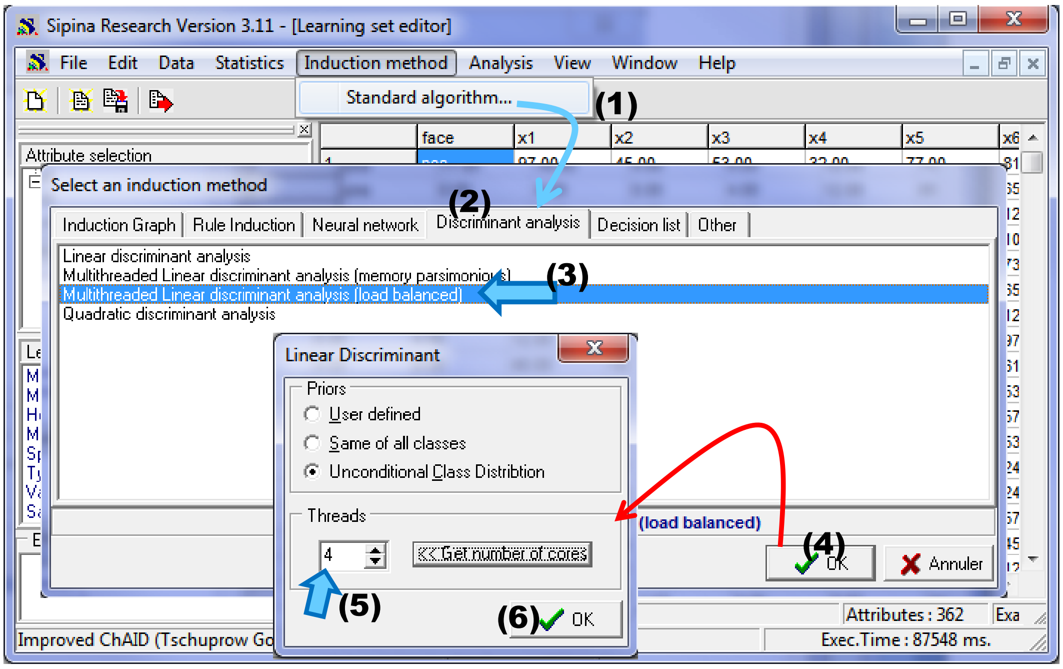
Only discrete
Only continuous
[✓ Both]

Improved ChAID (Tschuprow Goodness of Split) Exec.Time : 87548 ms.

We click on the OK button. The variables included in the analysis appear of the left part of the main window (D for discrete attribute, C for continuous one).

3.3 Load balancing of the multithreaded algorithm

To select the method, we click on the INDUCTION METHOD / STANDARD ALGORITHM **(1)** menu.


Sipina Research Version 3.11 - [Learning set editor]
File Edit Data Statistics Induction method Analysis View Window Help
Standard algorithm...
(1)
Select an induction method
Induction Graph | Rule Induction | Neural network | Discriminant analysis | Decision list | Other
(2)
(3)
(4)
OK
Annuler
(load balanced)
Linear Discriminant
(5)
(6)
OK
Get number of cores
Attributes : 362
Exec.Time : 87548 ms.
Improved ChAID (Tschuprow Go

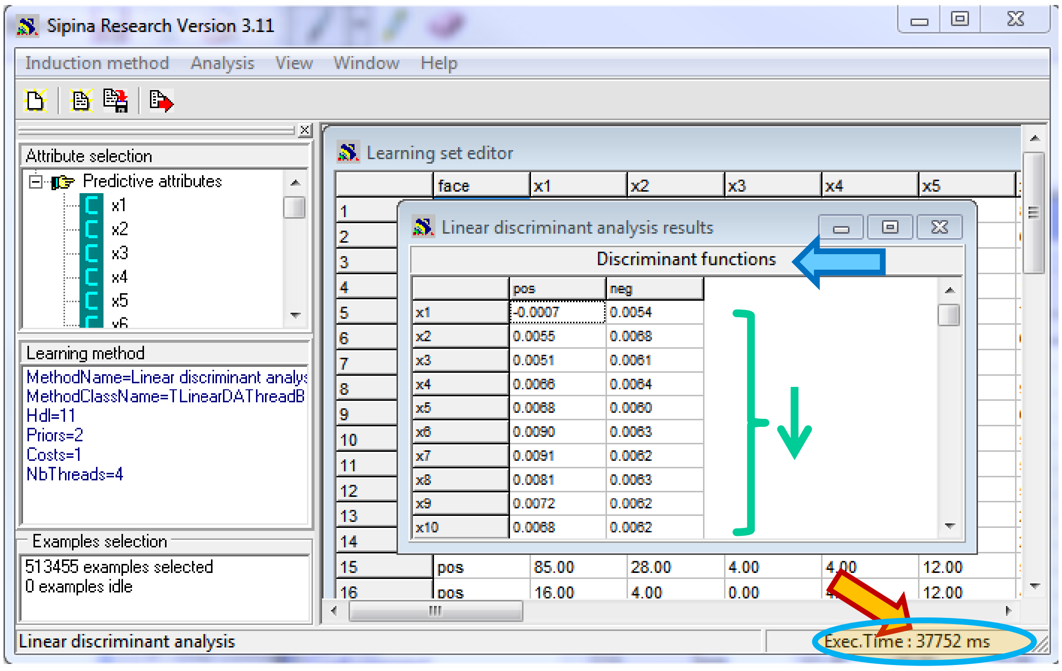
Linear discriminant analysis
Multithreaded Linear discriminant analysis (memory parsimonious)
Multithreaded Linear discriminant analysis (load balanced)
Quadratic discriminant analysis

Priors
( ) User defined
( ) Same of all classes
[✓ Unconditional Class Distribtion]

Threads
4

Into the DISCRIMINANT ANALYSIS tab **(2)**, we pick the MULITHREAD LINEAR DISCRIMINANT ANALYSIS (LOAD BALANCED) algorithm **(3)**. We validate the choice **(4)**. Into the dialog settings, we can set the number of threads to use **(5)**. We set **4 threads**, knowing that it is possible to know the number of available cores on the machine by clicking the button "Get number of cores". We confirm with the OK button **(6)**.

We can launch the analysis by clicking on the ANALYSIS / LEARNING menu. We obtain the results in **37.752** seconds. The classification functions are displayed in a new window.


Sipina Research Version 3.11
Induction method Analysis View Window Help
Attribute selection
Predictive attributes
x1
x2
x3
x4
x5
x6
Learning set editor
Linear discriminant analysis results
Examples selection
Linear discriminant analysis
Exec.Time : 37752 ms

Learning method
MethodName=Linear discriminant analys
MethodClassName=TLinearDAThreadB
Hdl=11
Priors=2
Costs=1
NbThreads=4

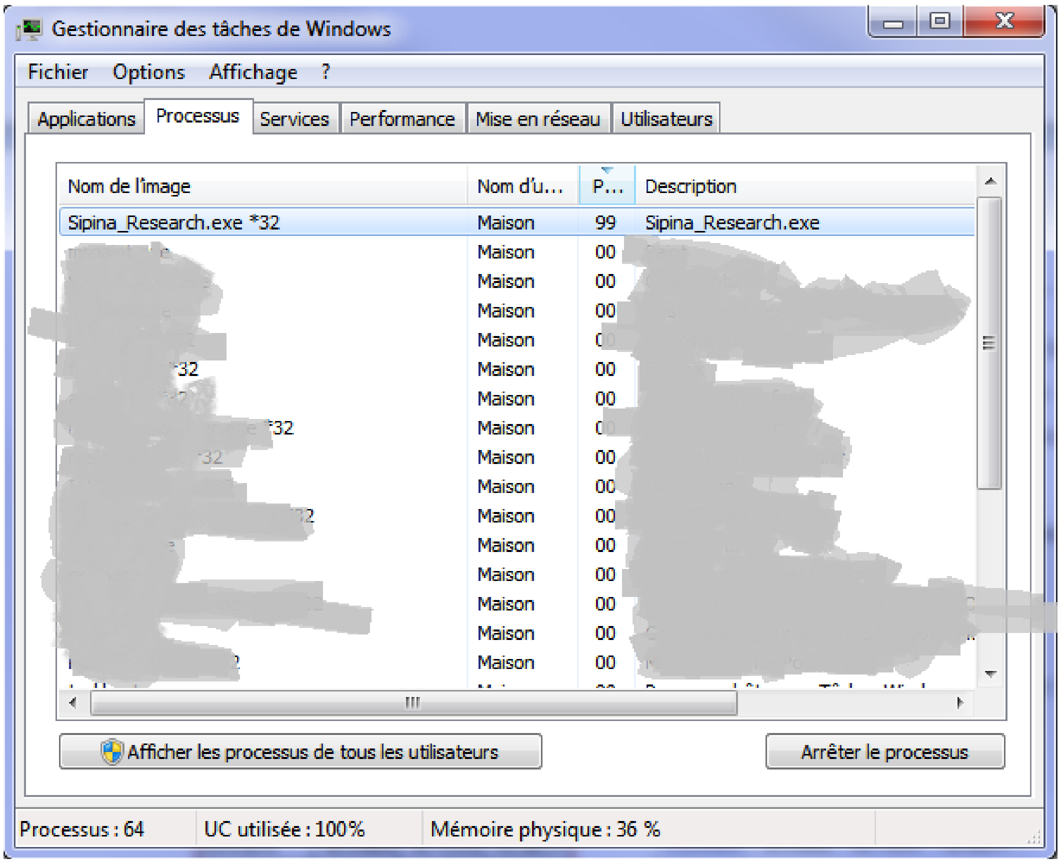
513455 examples selected
0 examples idle

Discriminant functions

	pos	neg
x1	-0.0007	0.0054
x2	0.0055	0.0068
x3	0.0051	0.0061
x4	0.0066	0.0064
x5	0.0068	0.0060
x6	0.0090	0.0063
x7	0.0091	0.0062
x8	0.0081	0.0063
x9	0.0072	0.0062
x10	0.0068	0.0062

Note: We use the user time, by computing the difference between two calls to the GetTickCount() function. Thus, if all the cores are used, and in the same time another application is running (e.g. flash player in a browser), the measurement may be disturbed.

We monitored Windows Task Manager. Really, all the resources (99%) are allocated to the calculation under Sipina.


Gestionnaire des tâches de Windows
Fichier Options Affichage ?
Applications Processus Services Performance Mise en réseau Utilisateurs
Afficher les processus de tous les utilisateurs
Arrêter le processus

Nom de l'image	Nom d'u...	P...	Description
Sipina_Research.exe *32	Maison	99	Sipina_Research.exe
[redacted]	Maison	00	[redacted]
[redacted]	Maison	00	[redacted]
[redacted]	Maison	00	[redacted]
[redacted]	Maison	00	[redacted]
[redacted]	Maison	00	[redacted]
[redacted]	Maison	00	[redacted]
[redacted]	Maison	00	[redacted]
[redacted]	Maison	00	[redacted]
[redacted]	Maison	00	[redacted]
[redacted]	Maison	00	[redacted]
[redacted]	Maison	00	[redacted]
[redacted]	Maison	00	[redacted]
[redacted]	Maison	00	[redacted]
[redacted]	Maison	00	[redacted]
[redacted]	Maison	00	[redacted]

Processus : 64 UC utilisée : 100% Mémoire physique : 36 %

3.4 Experiments on various databases

To evaluate the behavior of this new approach against the two previous ones (memory parsimonious multithreaded approach, single-threaded approach), we measure the computation time on various databases described in the preceding paper[4]. Let us remember their characteristics. For the WAVE datasets, we have always K = 3 balanced classes: WAVE500K with n = 500,000 instances and p = 21 descriptors; WAVE500KLARGE with n = 500,000 instances and p = 121 descriptors (100 additional descriptors); WAVE2M with n = 2,000,000 instances and p = 21 descriptors. These are all artificial databases that we can modify as we want. The idea is to study the impact of varying the number of instances and the number of descriptors on the calculation time.

We have also process the COVTYPE[5] dataset with K = 7 classes, but rather unbalanced (2 classes of the target variable concentrated a large part of the observations); and MIT FACE IMAGE[6] dataset, with K = 2 very unbalanced classes.

[4] http://data-mining-tutorials.blogspot.fr/2013/05/multithreading-for-linear-discriminant.html

[5] http://archive.ics.uci.edu/ml/datasets/Covertype

[6] http://c2inet.sce.ntu.edu.sg/ivor/cvm.html (Extended MIT face + non-face images data set).

We described below the computing time (in seconds). We set as reference the single-threaded version. The ratio, corresponding to the reduction of the calculation time, is defined as follows:

$$Ratio = \frac{Duration\ singlethread}{Duration\ multithread}$$

For instance, Ratio = 3 = (1.30 / 0.44) for the WAVE500K dataset means that the memory parsimonious multithreaded approach is 3 times faster than the single-threaded program (the execution time is divided by 3).

Our testing machine has a **Quad-core** Q9400 processor. For the load balanced multithreaded approach, we use systematically M = 4 threads.

Dataset	K	n	p	SIPINA (multithread) Load balanced	SIPINA (multithread) Memory Parsimonious	SIPINA (single thread)
Wave 500k	3	500000	21	0.38	0.44	1.30
Wave 500k Large	3	500000	121	4.87	6.13	19.19
Wave 2M	3	2000000	21	1.39	1.83	5.16
Covtype	7	581012	52	1.44	2.67	5.30
Face Images	2	513455	361	37.75	135.46	142.73

	Ratio against single thread	
Wave 500k	3.5	3.0
Wave 500k Large	3.9	3.1
Wave 2M	3.7	2.8
Covtype	3.7	2.0
Face Images	3.8	1.1

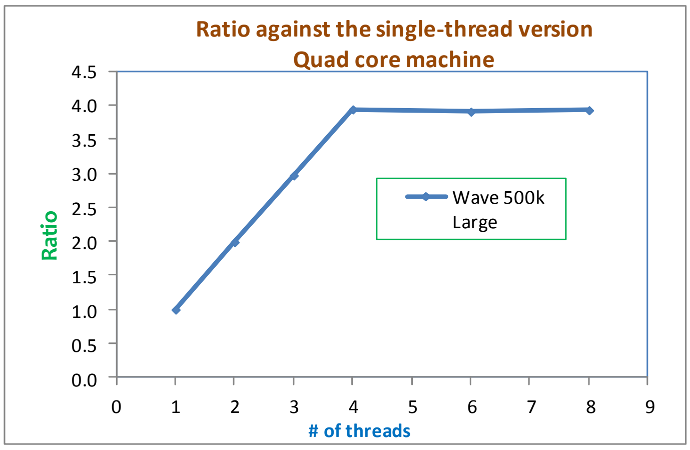
These results suggest some comments:

- The performances of the multithreaded version parsimonious in memory, especially their relationships with the characteristics of the data, have widely been detailed in our previous paper. We note mainly that in particularly unfavorable configurations (e.g. ratio = 1.1 for MIT FACE IMAGES), the gain is almost zero.
- It is otherwise with regard to the "load balanced" version. The computation time is disconnected from the dataset characteristics. The ratio is almost the same regardless of the base. The execution time reduction for MIT FACE IMAGES is particularly impressive.
- But the ratio is not equal to 4 because (1) we use the user time to measure the execution time, the system has continuously performed other tasks in the same time; (2) a part of the treatments is in a single-threaded mode (e.g. the inversion of the pooled covariance matrix).

Conclusion. Compared to the previous version (memory parsimonious approach), we obtained exactly what we wanted now: the number of threads to use is configurable, the loads are well balanced. But, the additional cost is an increase of the memory occupation. It seems however that this constraint is quite reasonable on the majority of database.

3.5 Influence of the number of threads used

To what extent using additional threads can improve performance? In the graph below, we show ratios depending on the number of thread used for the "WAVE500K Large" dataset.



On a Quad-core machine (4 cores), adding a thread improves the processing time while M < 4. Note that the ratio's evolution is linear according to the number of threads i.e. the overall calculation is well divided into as many cores used. The ratio is not perfect however. It will be very slightly less than M because, among other reasons, a part of calculations is single-threaded (e.g. the calculation of the pooled covariance matrix and its inversion).

Beyond M = 4, one additional thread brings nothing. But it does not degrade the performances, which are really limited by the number of available cores.

3.6 Comparison with SAS

SAS is a state-of-the-art statistical tool which can handle easily large datasets. We have used the proc discrim of SAS 9.3 on the same databases. E.g.

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

We compare the computation time with those of Sipina (load balanced multithreaded version).

Dataset	K	n	p	SIPINA (threads) Load balanced	SAS
Wave 500k	3	500000	21	0.38	1.65
Wave 500k Large	3	500000	121	4.87	9.09
Wave 2M	3	2000000	21	1.39	6.19
Covtype	7	581012	52	1.44	4.29
Face Images	2	513455	361	37.75	39.12

We observe that:

- Sipina is systematically better. Use of all available machine resources using the threads is beneficial.
- This result is all the more remarkable that SIPINA, because of its internal structures, is disadvantaged when the database includes a large number of variables (e.g. WAVE500KLARGE, MIT FACE IMAGES...). The use of the multithreaded approach allows to overcome this drawback.

Note: I know that it is possible to switch to multithreading under SAS using the THREADS option. But it is available only for some methods at the moment (SAS 9.2). I think that the modification of the 'proc discrim' will occur at one time or another.

4 Conclusion

Very excited by a first multithreaded version of the discriminant analysis implemented in SIPINA 3.10, I tried to improve the procedure by disconnecting its performance to the characteristics of dataset to be processed. Now available in **SIPINA 3.11**, this version uses better the machine resources by using all available cores, and by better dividing up the loads.