GENERATEUR DE DONNEES ET SUPERVISEUR D'EXPERIENCES

Gilles Verley et Eric Ramat[1]
Laboratoire d'**I**nformatique
Ecole d'**I**ngénieurs en **I**nformatique pour l'**I**ndustrie
64, avenue Jean Portalis
37200 - TOURS
☎ 02 47 36 14 14
e-mail: ramat,verley@univ-tours.fr

Résumé

Générer des jeux de données complexes conformes à un plan d'expériences et faciliter la mise en oeuvre de ces données sur les programmes informatiques que l'on veut expérimenter sont les deux objectifs que nous nous sommes donnés pour réaliser l'atelier informatique d'expérimentation sur des programmes que nous présentons d'abord au travers d'une expérimentation sur les classifieurs supervisés et, ensuite, de manière théorique.

1. Introduction

Disposer de données numériques générées artificiellement est une nécessité chaque fois que l'on veut observer le comportement externe d'un programme informatique qui est censé résoudre un problème particulier, c'est-à-dire produire une sortie aussi proche que possible d'une sortie désirée lorsqu'on lui fournit certaines données en entrée. Ces simulations ou expériences sont justifiées de différentes manières selon la nature des problèmes à résoudre:

- lorsqu'il existe des solutions déterministes, les programmes informatiques qui les implémentent sont, en général et en pratique, très difficilement prouvables. Il est donc nécessaire de les vérifier empiriquement avant de les mettre en situation réelle de fonctionnement. De plus, même lorsqu'ils sont formellement équivalents, certaines de leurs performances, comme le temps de calcul, peuvent varier.
- lorsqu'il n'existe pas de solution déterministe, les heuristiques qui en font office sont, par nature, gourmandes en paramètres d'optimisation. Ceci nécessite de vastes plans d'expérience pour estimer et comparer les performances sur des ensembles de données suffisamment représentatifs. Cette problématique est très courante dans des domaines tels que la reconnaissance des formes et l'ordonnancement.

Face à cette situation, le concepteur d'un programme doit réaliser un second programme, sur mesure, permettant de valider ou optimiser le premier. Ce second programme, outre le temps et le coût que son développement occasionne, est une nouvelle source potentielle d'erreurs. De plus, la reproduction éventuelle des expériences par des tierces personnes suppose qu'elles disposent également de ce second programme ou, pour le

[1] Allocataire de Recherche du Conseil Régional du Centre - Moniteur CIES

moins, des jeux de données produits par celui-ci. La solution globale à tous ces inconvénients consiste alors à utiliser des jeux de données standards plus ou moins adaptés à la situation mais qui ont l'avantage d'être reconnus par une communauté de personnes travaillant dans le même domaine. En définitive, ces jeux de données sont censés être représentatifs du domaine traité sans que cette représentativité soit toujours bien assurée. On peut alors penser que l'on sélectionne, de manière très arbitraire, des programmes ad hoc à ces jeux de données. Cette situation est loin d'être utopique. En reconnaissance des formes, on cite souvent l'exemple de la conception d'un programme informatique qui devait reconnaître les chars russes et les chars américains à partir de leurs photos respectives. Parmi les nombreux programmes testés, on en trouva un qui distinguait parfaitement les deux catégories. On s'aperçut, plus tard, que le programme distinguait, en fait, la finesse du grain des photos qui n'était pas la même sur les photos des deux types de chars. Toute expérience comporte des biais méthodologiques et il est souvent difficile d'associer généralité et reproductibilité dans les expériences. Le fait que les expériences portent sur des programmes informatiques ne simplifie pas la méthodologie expérimentale. Cela facilite seulement l'automatisation des expériences.

La description détaillée de notre atelier informatique d'expérimentation se trouve dans le troisième paragraphe de cet article qui peut-être lu avant le second paragraphe qui, pour des raisons pédagogiques, est placé avant. De manière succincte, l'atelier d'expérimentation possède deux fonctions principales. L'une consiste à générer des données d'une manière aussi universelle que possible autant en ce qui concerne la structure même de ces données qu'en ce qui concerne les valeurs affectées à ces données. La spécification de la structure des données générées ainsi que de la procédure permettant d'affecter une valeur à ces données s'effectue au travers d'un langage. La deuxième fonction consiste à élaborer des plans expérimentaux à partir des échantillons de données générés précédemment et des programmes ou variantes de programmes qui vont recevoir ces données en entrée. Les sorties calculées par les programmes testés sont enregistrées d'une manière permettant leur exploitation statistique. La spécification de ces plans expérimentaux ainsi que des données expérimentales utiles s'effectue également au travers d'un langage. Tout programme exécutable, qui s'apparente à une procédure ayant trois sortes de paramètres: une structure de données en entrée, une structure de paramètres qui modifient l'exécution du programme, une structure de données en sortie, peut être intégré dans notre atelier d'expérimentation.

L'application de l'atelier que nous décrivons dans le paragraphe suivant est la réalisation d'une expérience dans laquelle on cherche à comparer les performances de différents classifieurs supervisés utilisés en reconnaissance des formes et à vérifier certaines hypothèses corrélatives [Verley 96].

2. Application à l'étude comparative des classifieurs supervisés

2.1 Cadre théorique

On se place maintenant dans le cadre de la classification avec apprentissage supervisé. Schématiquement, un classifieur est un programme capable de classer automatiquement des points dans un espace de représentation. On dit qu'il est supervisé lorsque ce classement s'effectue après qu'on lui ait fourni un échantillon de points déjà classés (phase d'apprentissage). Supervisés ou non, les classifieurs dépendent de paramètres dont la détermination est, bien souvent, heuristique. C'est le cas des classifieurs supervisés à base de réseaux de neurones formels (MLP). Les deux principaux paramètres à fixer a priori sont le nombre de cellules cachées du réseau et le nombre de cycles (ou durée) de la phase d'apprentissage.

2.2 Hypothèses

On cherche à savoir s'il existe, en toute généralité, un paramètre plus intéressant que l'autre à optimiser et, de manière plus fine, comment optimiser ces paramètres en fonction de certaines caractéristiques du problème de classification.

2.3 Méthodologie expérimentale

Pour la première hypothèse, il faut donc produire des problèmes de classification supervisée avec la plus grande généralité possible. Cela signifie faire varier le nombre de dimensions de l'espace de représentation, le nombre de classes, les probabilités a priori des classes mais également et, plus difficilement, les lois de probabilité conditionnelles à chaque classe. Les aspects théoriques de cette dernière question sont traités par ailleurs [Verley 94].

Schématiquement, on contrôle les deux derniers facteurs de la manière suivante. On considère un maillage régulier plus ou moins fin de l'espace de représentation. On affecte, par tirage uniforme, certains des noeuds du maillage précédent aux différentes classes dans la proportion exacte des probabilités a priori des classes que l'on souhaite. A chaque noeud affecté à une classe, on associe une loi normale centrée sur ce noeud et de même écart-type. La liste **S** des couples (noeud, classe) des noeuds affectés aux classes détermine alors formellement une forme analytique du problème ainsi qu'un algorithme permettant de produire les échantillons correspondants.

<u>Point de vue analytique:</u>
La fonction de densité de probabilité conditionnelle à une classe ω_c est:

$$f^{(c)}(\underline{x}) = \frac{1}{card(k_c)} \sum_{i \in k_c} N\left(\underline{m}_i ; \sigma_c^2 \underline{I}\right)$$

où k_C est l'ensemble des indices des noeuds affectés à la classe c dans la liste **S**
et $\underline{m}_i$ est le noeud d'indice i dans la classe c

et σ_C est l'écart-type des lois normales **N**
Les probabilités a priori des classes ω_c sont :

$$P_c = \frac{card(k_c)}{\sum_c card(k_c)}$$

Point de vue algorithmique:
La procédure suivante assure un tirage d'un point (et donc d'un échantillon par simple répétition) selon les données du problème défini analytiquement au paragraphe précédent à partir de la simple connaissance de la liste **S** :
1 on tire un noeud dans la liste **S** selon une loi uniforme.
2 on tire ensuite un point selon une loi normale centrée sur le noeud tiré en 1

En faisant varier seulement les trois paramètres suivants (hormis la dimension de l'espace de représentation, le nombre de classes et les probabilités a priori des classes), on assure une grande généralité aux problèmes de classification produits par le générateur. Il s'agit de:
- la finesse du maillage
- la proportion des noeuds affectés à aucune classe
- l'écart-type de la loi normale associée aux noeuds affectés à une classe

Cette généralité a été démontrée théoriquement en s'appuyant sur la démonstration des estimateurs à noyaux de Parzen [Verley 94]. On illustre graphiquement sur deux exemples à deux dimensions, le type de problème de classification pouvant être produit selon les spécifications des paramètres. Les limites entre les classes sont les frontières bayésiennes qui peuvent être déterminées à partir de la connaissance analytique des lois de probabilité des classes.

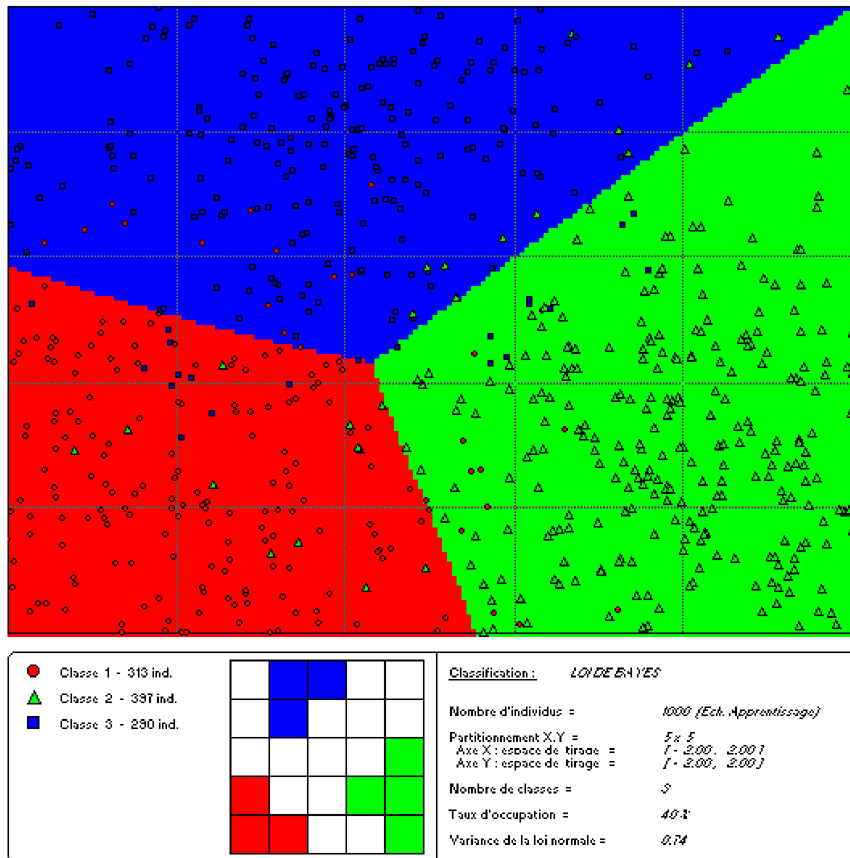

Figure 1. Problème simple à trois classes.7

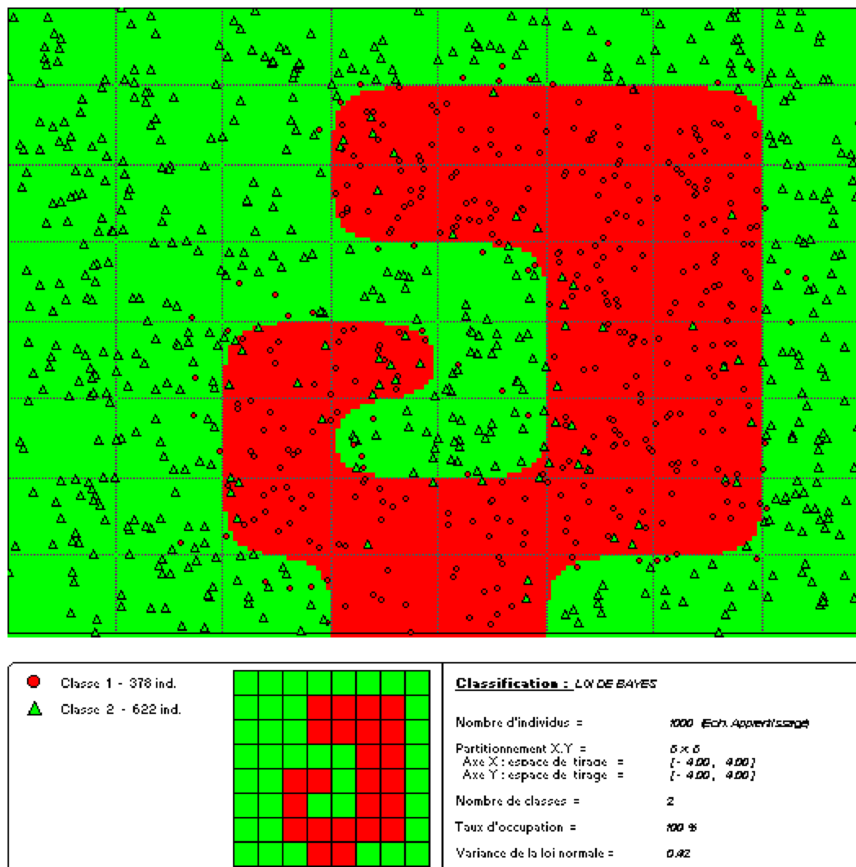

Figure 2. Problème complexe à deux classes.

2.4 *Spécifications du plan d'expérience*

On montre maintenant comment réaliser notre expérience grâce à notre atelier d'expérimentation dont la description compléte est détaillée dans la troisième partie.
Il s'agit de:

- spécifier la structure des données à générer
- définir la méthode de génération des données afin de produire les échantillons nécessaires à la validation de l'hypothèse
- intégrer le classifieur à expérimenter à l'atelier d'expérimentation et spécifier la forme des fichiers résultats
- déterminer les différentes valeurs des paramètres du classifieur étudié et lancer l'expérience
- exploiter les résultats

2.4.1 spécifier la structure des données à générer

La copie d'écran suivante illustre la définition de la structure de données permettant de générer les échantillons de données nécessaires à notre expérimentation:

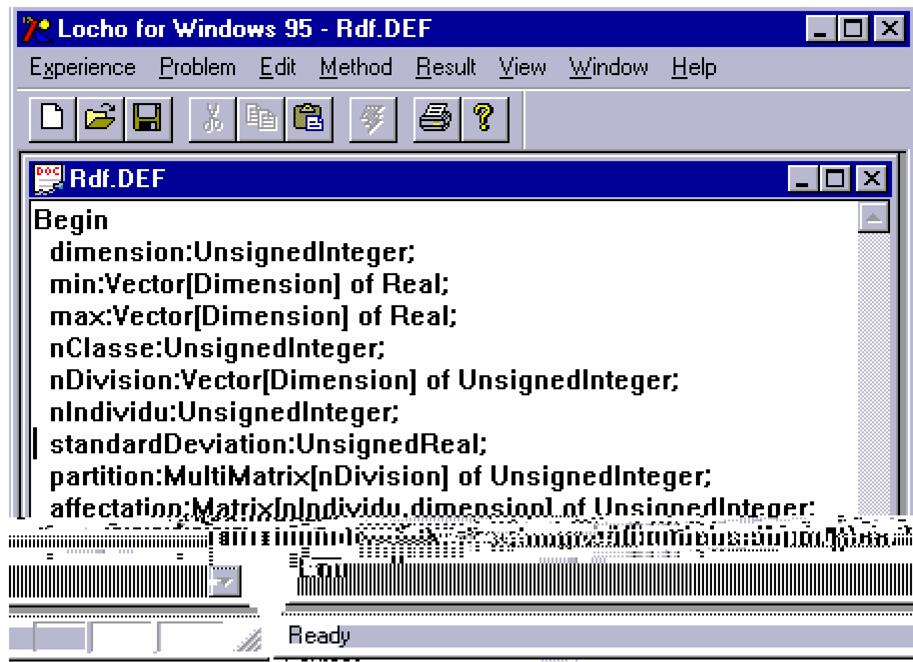

Figure 3. Spécification du problème.

Quelques explications sur la signification de ces variables :

- **dimension** : dimension de l'espace.
- **min** et **max** sont les vecteurs qui déterminent les bornes de l'espace dans chaque dimension.
- **nClasse** : nombre de classes.

- **nDivision** est le vecteur qui contient le nombre de mailles dans chaque dimension.
- **nIndividu** : nombre d'individus par échantillon généré.
- **standardDeviation** : écart-type des lois normales associées aux noeuds .
- **partition** est la manière retenue pour coder la liste **S**. C'est une hypermatrice où chaque élément (repéré par les indices dans la matrice) représente un noeud du maillage de l'espace et aura pour valeur le numéro de la classe associée à ce noeud (0 si le noeud n'est affecté à aucune classe).
- **affectation** est la matrice qui contiendra, pour chaque individu de l'échantillon, les indices du noeud auquel il sera associé. Ce sont les indices de l'élément de la matrice **partition** qui représente le noeud en question.
- **individu** est la matrice des individus de l'échantillon qui contiendra les coordonnées

modalités). Il aurait été possible de faire varier dans ce même plan les facteurs tels que la dimension de l'espace, le nombre de classes, les probabilités a priori des classes. Cela n'a pas été fait ici afin de ne pas alourdir l'écriture de l'exemple présenté.

La Section « Data » permet de définir les expressions qui vont déterminer les valeurs que doivent prendre les variables déclarées dans le paragraphe 2.4.1 .
Certaines expressions sont des simples littéraux scalaires comme les expressions de la dimension ou du nombre de classes. Ils sont affectés tout simplement à des variables de même type.

Les expressions du **min** et du **max** sont également des littéraux scalaires alors que ces variables sont des vecteurs. Il y a donc conversion implicite de type comme on le verra en détail dans la troisième partie. Les éléments de chaque vecteur auront donc ici la même valeur.

L'expression de la variable **partition** est un tout petit plus complexe à comprendre. Elle renvoie la valeur scalaire 0 ou 1 ou 2 ou 3 (valeurs qui représentent conventionnellement les classes que l'on va affecter aux noeuds) selon un tirage discret respectant certaines probabilités a priori déterminées dans la même expression (0.1 pour la classe 1, 0.6 pour la classe 2, et ainsi de suite). La variable **partition** étant une multimatrice, il y a encore conversion implicite de type et chaque élément de la matrice sera le résultat d'un tirage discret effectué dans les conditions décrites ci-dessus.

L'expression de la variable **affectation** est plus complexe. Elle renvoie un vecteur d'entiers dont chaque élément est le résultat d'un tirage uniforme entre 1 et le nombre de mailles dans la dimension correspondante. Ceci est déterminé par les deux premiers paramètres de la fonction **Uniforme**. Le vecteur ainsi généré détermine donc une maille qui va être affectée à un individu de l'échantillon par la spécification des indices de cette maille dans les différentes dimensions. Les paramètres de la variable **affectation** indiquent simplement que l'expression devra être évaluée à nouveau pour chaque ligne de la matrice (formellement, cela correspond au caractère souligné figurant en premier paramètre) et que le vecteur qui résulte de l'expression doit être affecté dans les différentes colonnes de la ligne considérée (formellement, cela est précisé par la forme <1> du second paramètre qui indique que la première dimension du résultat de l'expression sera affectée à la deuxième dimension de la matrice, c'est-à-dire les colonnes). Il reste à préciser la signification du dernier paramètre de la fonction **Uniforme**. Ce paramètre est facultatif. Il permet l'introduction de contraintes sur le tirage. Ici, on précise que la maille qui va être affectée par tirage à un individu doit, elle-même, avoir été affectée à une classe (<> 0).

L'expression de l'objet **individu** renvoie un vecteur tiré selon une loi normale multidimensionnelle dont le vecteur-moyenne est le centre de la maille associée à l'individu auquel le résultat de l'expression sera affecté. Le calcul des coordonnées de ce vecteur-moyenne est possible en récupérant les indices de la maille correspondante par référence à la variable **affectation** (cf. référence partielle à une donnée expliquée en 3.2.4) et en effectuant une simple translation en fonction des bornes de l'espace (s'il s'agit de la 3éme maille selon la première dimension de l'espace et de la 5éme maille selon la deuxième dimension, la maille est référencée par le couple (5,3) et les

coordonnées du centre de cette maille sont obtenues immédiatement à partir des bornes de l'espace selon les deux dimensions).

Lors de l'exécution du plan expérimental ainsi défini, des fichiers vont être créés pour stocker les différentes valeurs générées pour les variables définies dans le plan expérimental. De manière standard, il est <u>créé un fichier par variable et par modalité du plan expérimental</u>. Les coordonnées des points seront, par exemple, stockées dans les fichiers **individu00001.dat** à **individu00045.dat**. Les classifieurs à tester (ici les MLP) iront chercher leurs données directement dans ces fichiers ou au travers de passerelles ad hoc sous forme de DLL.

2.4.3 intégrer le classifieur à étudier

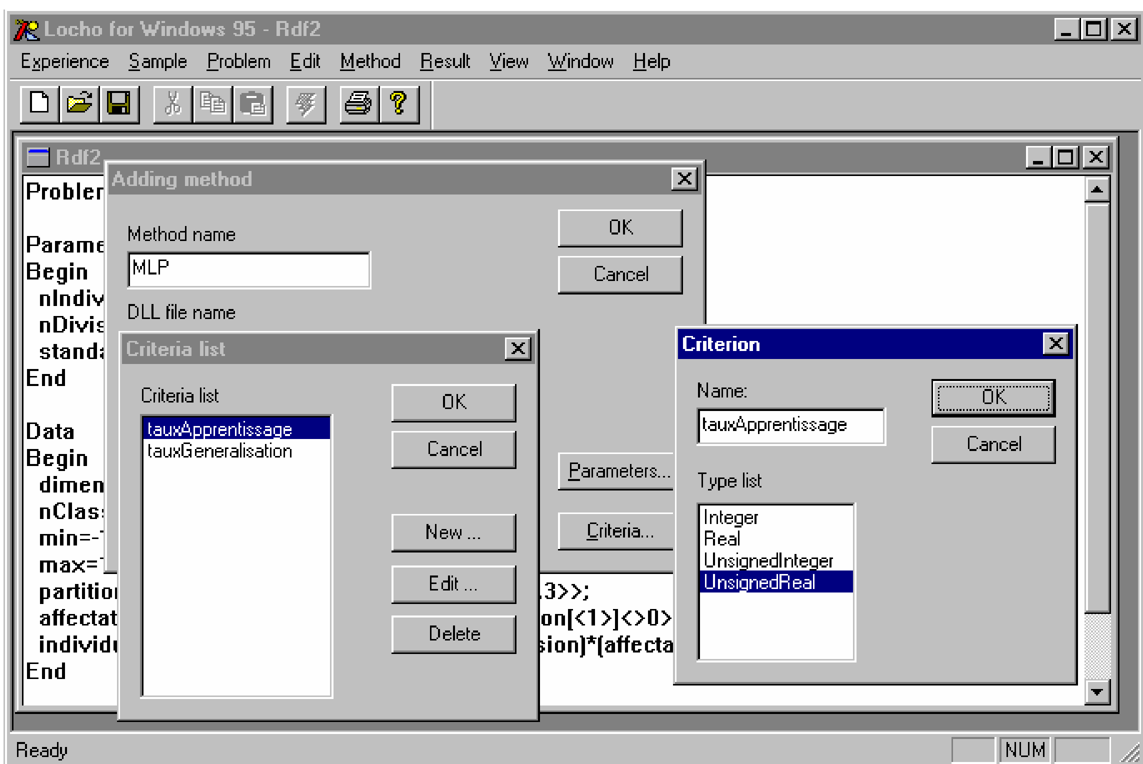

Figure 5. Intégration d'une méthode.

Le classifieur à base de réseaux de neurones (MLP) que nous voulons expérimenter doit donc être compilé sous forme de DLL, puis être simplement déclaré à notre atelier d'expérimentation. Les données produites en sortie par le MLP et qui devront être stockées dans les fichiers de résultats doivent être également déclarées. Il s'agit ici du taux d'apprentissage, c'est-à-dire le taux de bonne classification sur l'échantillon ayant servi à l'apprentissage du MLP et du taux de généralisation, , c'est-à-dire le taux de bonne classification sur un échantillon de test généré dans les mêmes conditions que l'échantillon d'apprentissage.

2.4.4 spécifier les différentes valeurs des paramétres du classifieur étudié

Cette partie ne pose pas de problèmes particuliers. Comme on peut le voir dans la copie d'écran suivante, il suffit, pour chaque paramètre testé, de lister les valeurs qu'il prendra successivement au cours de l'expérience. Dans l'expérience décrite ici, on croise les modalités du facteur nombre de cellules cachées avec celles du facteur nombre de cycles d'apprentissage.

Il ne reste plus qu'à lancer l'expérience[2] et à exploiter les résultats grâce à des logiciels externes.

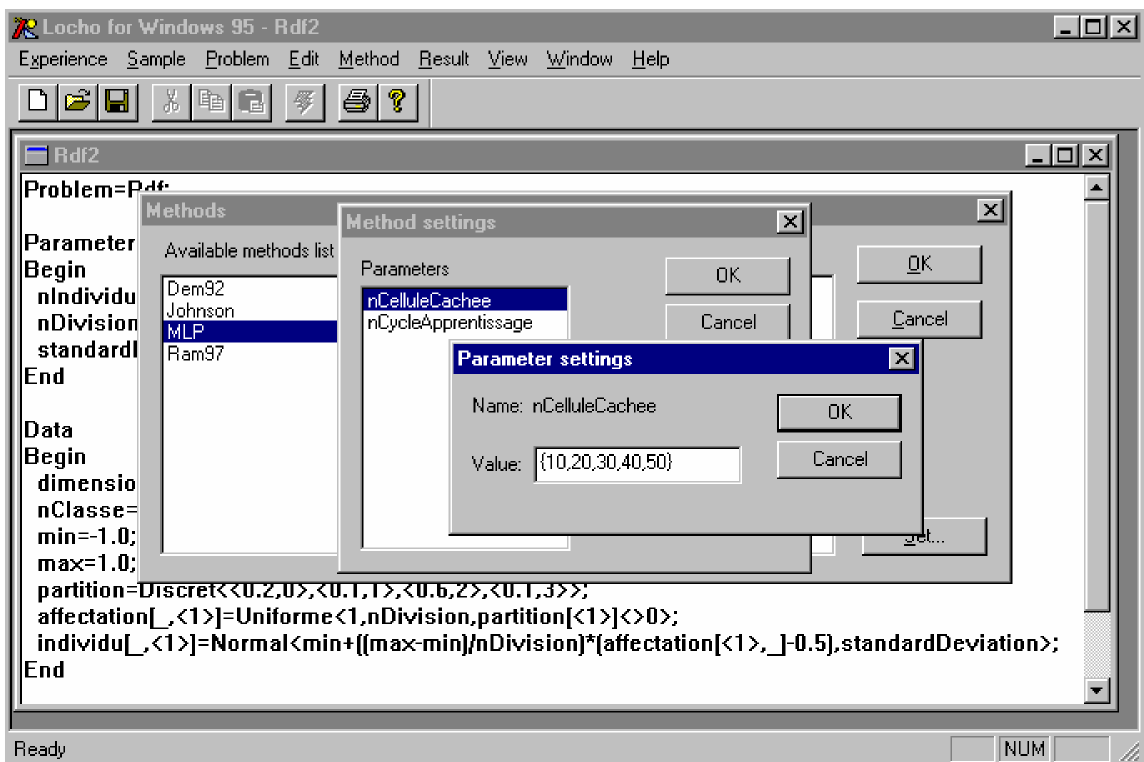

Figure 6. Spécification des paramétres de la méthode.

2.5 Résultats expérimentaux

Les résultats expérimentaux détaillés sont décrits par ailleurs [Verley 96]. Globalement, nous avons pu montrer que le facteur durée d'apprentissage est plus intéressant à optimiser que le facteur nombre de cellules et cela, de manière d'autant plus nette, que l'ambiguïté des classes est grande, facteur qui était contrôlé dans notre expérience par le paramètre « écart-type ». Cette manière de contrôler l'apprentissage dans ce type de classifieur est connue sous l'anglicisme « early stopping ».

---

[2] Une fonctionnalité permettra très prochainement de superviser l'expérience sur plusieurs PC en réseau lorsqu'il s'agira de grands plans expérimentaux.

3. Description détaillée de l'atelier informatique de génération de données et de supervision d'expériences

L'atelier d'expérimentation dont nous avons présenté une application particulière est donc une réponse aux besoins de validation que nous avons dû satisfaire dans les domaines de la reconnaissance des formes et de l'ordonnancement [GOThA 93] [Taillard 93] [Nadreau 96]. Il est structuré en trois modules (Figure 7). Le premier module (3.1) sert à définir formellement la structure de données que l'algorithme à tester doit recevoir en entrée (spécification du problème). Le second module (3.2) permet de définir complètement les lois de génération (aléatoires ou non) des données dont la structure a été définie dans le premier module (spécification de l'expérience). Le dernier module (3.3) sert à gérer les paramètres des algorithmes à tester, l'exécution de ceux-ci sur les données générées ainsi que la structure des fichiers de résultats. Nous présentons maintenant chaque élément de l'outil en nous appuyant le plus souvent possible sur des exemples.

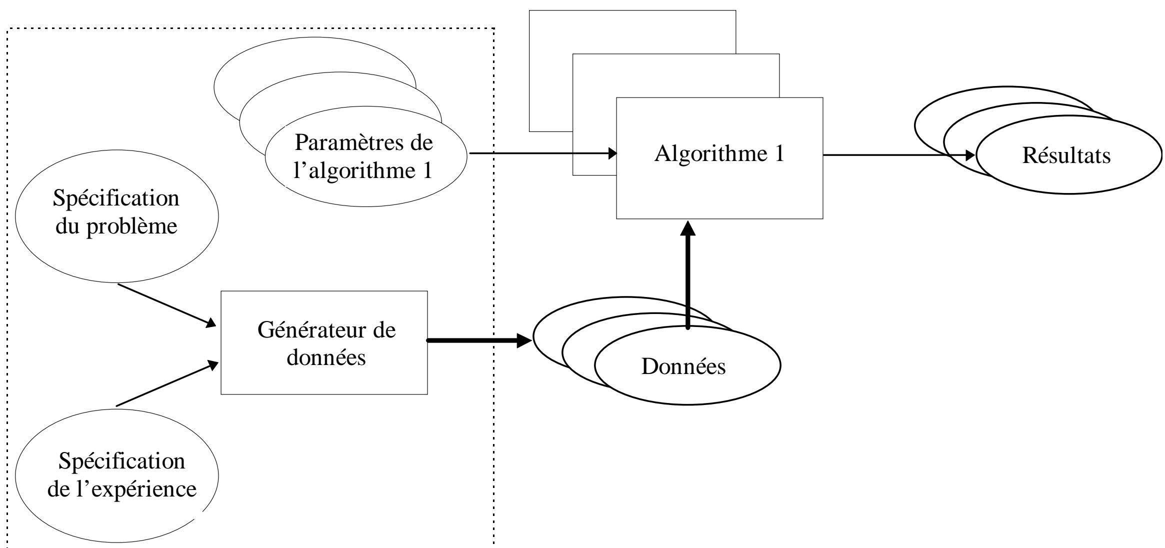

Figure 7. Architecture logique de l'atelier de validation

3.1 Le langage de spécification de problèmes

Les données d'entrée d'un algorithme sont de deux types: celles dépendantes du problème que l'on doit résoudre et celles dépendantes uniquement de l'algorithme de résolution (ce second ensemble de données sera nommé, par la suite, paramètres de l'algorithme). Le langage de spécification de problèmes s'attache pour sa part à définir les données du premier type, c'est-à-dire spécifiques du problème.

Prenons un exemple. Soit le problème d'analyse numérique suivant : rechercher les zéros d'un polynôme. Les données nécessaires à la validation d'un algorithme résolvant ce problème se limitent à deux données élémentaires : le degré et les coefficients du polynôme. Ce sont, en effet, ces deux éléments que l'on désire générer aléatoirement soit

pour valider la complétude de l'algorithme développé soit pour mesurer ses performances vis à vis d'un ensemble d'algorithmes résolvant ce même problème.
Le langage est basé sur quatre grands types de données : les types simples (entier, entier non signé, réel et réel non signé), les vecteurs, les matrices et les matrices multidimensionnelles (généralisation des matrices aux dimensions supérieures à 2). Il est à noter que les trois derniers types définis sont construits uniquement à partir des types simples. Par exemple, on peut définir une matrice d'entiers non signés mais en revanche, la définition d'une matrice de vecteurs d'entiers n'est pas autorisée. Nous avons, en effet, considéré que le type matrice multidimensionnelle était suffisant pour la définition de structures matricielles complexes.

Cet ensemble de types n'est qu'une base élémentaire. En effet, l'outil étant complètement conçu et développé en objet (OMT pour la modélisation et C++ pour le langage), l'ajout de types supplémentaires est possible à condition de vérifier les propriétés des classes de base de la hiérarchie actuelle (Figure 8 et Figure 9). L'exemple du type Graph [Kolisch 95] illustre parfaitement cette dynamique. Lors de la spécification d'un type de problème en ordonnancement, le besoin d'une structure très spécifique s'est fait sentir. Or aucune des structures de base ne permettaient sa modélisation. Nous avons donc défini un nouveau type (dérivé de la classe de base TType) ainsi qu'une représentation de sa valeur (dérivée de la classe TValue).

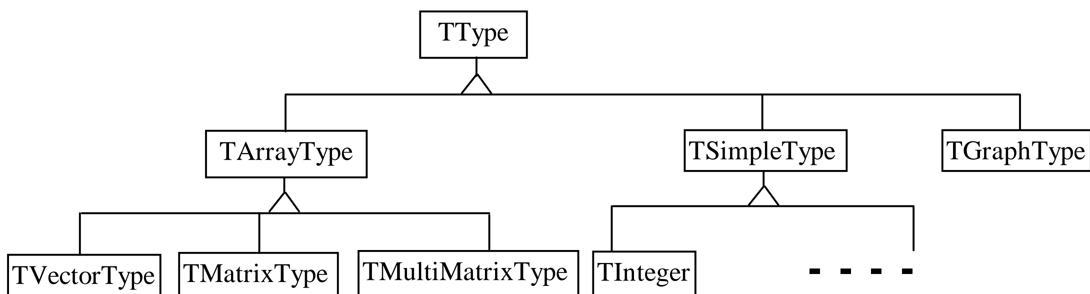

Figure 8. Hiérarchie partielle des types

Revenons maintenant à notre langage. Celui-ci ne permet que la déclaration des données. En effet, pour définir les données du problème, nous devons en énumérer la liste. Dans les langages structurés classiques (C ou Pascal), les données, lors de leur déclaration, ne peuvent être définis qu'en fonction de données statiques (par exemple, T est un tableau de 10 entiers). Nous avons donc étendu la syntaxe d'une déclaration, en permettant de définir une donnée en fonction de données préalablement définies. La taille d'un vecteur peut alors être le résultat d'une expression arithmétique définie à partir de données entières. Ceci n'est qu'un exemple ; l'approche adoptée permet de fixer tout paramètre d'une structure par le résultat d'une expression, au sens large (cette expression étant évaluée à la génération).

Quelques exemples simples.

L'exemple suivant montre un paramétrage simple du nombre de lignes d'une matrice par une donnée entière.

N:UnsignedInteger; *   /* Nombre de points dans* $\Re^2$ **/*

P:Matrix[N,2] of UnsignedInteger; / Matrice de points dans $\Re^2$ */*

Un peu plus complexe : la dimension et la taille de la matrice multidimensionnelle *partition* sont paramètrées par un vecteur qui est lui-même fonction, pour sa taille, d'une donnée entière.

dimension:UnsignedInteger;
nDivision:Vector[Dimension] of UnsignedInteger;
partition:MultiMatrix[nDivision] of UnsignedInteger;

Un dernier petit exemple. La taille du vecteur *D* est une expression arithmétique. La taille ne sera donc connue que lors de la génération.

Na:UnsignedInteger;
D:Vector[Na+2] of UnsignedInteger;

Les types de données disponibles ne se limitent pas aux types élémentaires présentés jusqu'ici. Nous avons étendu la notation relative à une structure matricielle. La taille d'une matrice, au sens large, est un vecteur d'entiers où la taille du vecteur définit la dimension de la matrice et les valeurs entières du vecteur, le nombre de valeurs de la matrice selon les différentes dimensions. Ceci constitue la définition la plus classique. Or, nous avons autorisé que l'une des tailles selon une dimension soit un vecteur. On peut alors générer des listes de matrices. Prenons un exemple:

v:Vector[2] of UnsignedInteger;
m:Matrix[v,3] of Real;

Lors de la génération, la donnée m aura pour valeur une liste de deux matrices dont les tailles seront v[1] lignes et 3 colonnes, et v[2] lignes et 3 colonnes. Naturellement, les valeurs du vecteur v ne seront connues que lors de la génération. On peut aller plus loin en définissant la taille même du vecteur à l'aide d'une donnée dont la valeur sera, bien entendu, connue seulement à la génération. On obtient alors une liste de matrices dont la longueur peut être générée aléatoirement et dont le nombre de lignes des matrices varie d'une instance à l'autre et suivant une loi spécifique. Cette possibilité est disponible pour toutes les structures matricielles. Il suffit de définir l'une des tailles par une expression dont l'évaluation donnera une valeur de type vecteur.

Un dernier complément sur la définition des données, il est souvent intéressant de générer une valeur de type vecteur où les différentes valeurs contenues dans le vecteur sont distinctes. La notion de gamme que nous manipulons dans les problèmes d'ordonnancement de type Job- shop par exemple, nécessite cette propriété. En effet, une gamme, pour ce problème particulier, définit l'ordre de passage d'un produit sur les différentes machines d'un atelier. Or un produit passe sur toutes les machines une et une seule fois. La gamme d'une pièce est donc un vecteur de m entiers (où m est le nombre de machines) distincts compris entre 1 et m.

gamme : Vector[m] of UnsignedInteger {Unique};

De manière générale, on peut spécifier qu'une structure matricielle ou une partie de celle-ci peut être à valeurs distinctes.

En conclusion, cette première version du langage de spécification de données est un point de départ consistant. Il serait néanmoins, intéressant d'introduire des types complémentaires tels que les booléens, les caractères, les chaînes de caractères ou tout autre type spécifique au domaine d'utilisation.

3.2 Le langage de spécification d'expériences

La seconde étape nécessaire à la validation d'algorithmes consiste à définir les règles de génération des données d'entrée. De manière générale, il faut pour chaque donnée du problème définir la valeur à lui affecter. On distingue deux types de données d'entrée: les données générées aléatoirement ou à partir d'une expression et les données qui prendront successivement plusieurs valeurs. On les distinguera dans deux sections distinctes: une section "data" pour le premier type et une section "parameter" pour le second. Si la donnée doit prendre plusieurs modalités, on écrit l'expression de l'ensemble des valeurs à prendre. Voyons la syntaxe à travers un petit exemple:

```
/* structure des données */
taille:UnsignedInteger;
m:Matrix[taille,3] of Real;
/* parameter */
taille={3,4,5};
/* data */
m=Uniforme<-1,1>;
```

La donnée *m* est une matrice dont le nombre de lignes est paramétré par *taille*. La petite spécification d'expérience indiquée ci-dessus permet de générer trois jeux de données où le nombre de lignes de la matrice sera égal successivement à 3, 4 et 5. La notation se généralise à tout type de donnée (liste de valeurs de type vecteur, par exemple). La section data permet, comme nous le verrons plus en détail après, d'affecter à chacun des éléments des matrices générées une valeur réelle tirée selon une loi uniforme entre -1 et +1. Une autre manière de spécifier la liste des valeurs à prendre pour une donnée de type simple est de l'écrire sous forme d'intervalle : *taille=<3~4,1>;* (<min~max,pas>). La donnée taille prendra les valeurs 3, 4 et 5. Si la spécification de l'expérience dispose de plusieurs données à modalités multiples, le nombre de jeux de données issu de la génération sera égal au produit des nombres de modalités

3.2.1 Syntaxe

Après plusieurs versions du langage, nous avons fini par aboutir à la grammaire suivante:

Affectation	*::=**Nom**=Expression; |*
	***Nom[**Localisation**]**=Expression; |*
	***Nom[Entier][**Localisation**]**=Expression;*
Localisation	*::=Index**,**Localisation | Index*
Index	*::=**_** | **<Entier>** | Expression*
Expression	*::=Terme **+** Expression | Terme **-** Expression | **-** Expression*
Terme	*::=Facteur ***** Terme*
Facteur	*::=Atome **/** Facteur*
Atome	*::=**(**Expression**)** | **Valeur** | **Nom** | SpécifLoi |*
	Nom[**Localisation**]** | **Fonction(**ListeExpressions**)
SpécifLoi	*::=**Loi<**ListeExpressions**>** | Ensemble | SpécifLoiDiscrète |*
	Loi<**ListeExpressions**,**ExpressionConditionnelle**>
Ensemble	*::=**{**ListeValeurs**}***
SpécifLoiDiscrète	*::=**{**ListeProbaValeur**}***
ListeProbaValeur	*::=**<Réel,Valeur>,**ListeProbaValeur | **<Réel,Valeur>***

Détaillons maintenant les règles afin d'en montrer les possibilités. La syntaxe de l'affectation que nous avons adoptée est classique : une référence à la variable à affecter et une expression. Il existe trois formes de référence à une variable. La première est appelée globale : la variable, toute entière, sera affectée par le résultat de l'évaluation de l'expression, sachant que l'évaluation sera effectuée autant de fois que nécessaire pour "remplir" la variable dans le cas matricielle.

M:Matrix[3,3] of Real;
M=Normal<0,2>;

La matrice M sera constituée de valeurs qui seront générées suivant une loi normale de moyenne 0 et d'écart-type 2. Par principe et quelque soit la référence utilisée, une expression est évaluée autant de fois que nécessaire.

La seconde forme de référence à une variable est relative uniquement aux structures matricielles. L'objectif est de spécifier une sous-structure matricielle (une colonne, par exemple) pour indiquer quelle expression lui sera affecté. La syntaxe retenue consiste à indiquer entre crochets une liste d'index où un index peut être de trois formes : un entier, un souligné ou un entier entre < >. L'entier permet d'identifier de manière stricte une ligne, par exemple, tandis que le souligné a pour sémantique quelque soit. Si on veut donc faire référence à la première ligne de la matrice M, on écrira : *M[1,_]*. De manière générale, si on veut référencer une sous-matrice de dimension *d'* d'une matrice de dimension *d*, il faudra utiliser *d'* soulignés. La troisième forme possible pour un index sera décrite plus tard (3.2.3).

La dernière forme de référence à une variable est, quant à elle, une généralisation de la précédente applicable aux variables de type liste de structures matricielles. En effet, le premier [] permet d'indiquer à quelle matrice dans la liste on veut s'adresser.

La seconde partie de l'affectation est constituée d'une expression. La syntaxe de celle-ci est classique et fait intervenir les opérateurs usuels (addition, soustraction, multiplication et division), les références sur les variables (forme identique à celle présentée pour la partie gauche de l'affectation) et les fonctions usuelles sur les réels (racine carrée, exponentielle, ...) et les vecteurs (moyenne, somme, produit, ...). Nous avons ajouté à cette liste les générateurs aléatoires. En effet, la génération aléatoire fait partie intégrante des expressions. L'atelier dispose des lois les plus usitées [Asselin de Beauville 73] [L'Ecuyer 96][Wallace 96] : uniforme, triangulaire, rampe, normale, log-normale, exponentielle, gamma, bêta, F, Student, Cauchy, Poisson et binomiale. De plus et dans un souci d'être complet, nous avons ajouté deux autres générateurs que l'on qualifie de discret. Ces derniers générateurs sont définis sous la forme d'une liste de valeurs (valeurs simples ou matricielles). Les processus de génération sont les suivants :

1. le choix d'une valeur parmi les autres est équiprobable;
2. chaque valeur est pondérée par sa probabilité d'être choisie.

3.2.2 Génération

Après avoir passé en revue la syntaxe de l'affectation, nous allons maintenant préciser le processus d'évaluation et d'affectation.

Syntaxiquement, il est possible de construire des expressions de type simple ou matricielle. Une expression de type simple est bâtie autour de variables de type simple, de générateurs aléatoires dont tous les paramètres sont de type simple, et de variables matricielles dont on adresse qu'une case. Les expressions matricielles sont, quant à elle, composées de variables matricielles, de générateurs aléatoires dont au moins un paramètre est une expression matricielle et de variables matricielles dont on adresse une partie. Si l'expression ou l'un des paramètres d'un générateur contient au moins un terme de type matriciel, son évaluation délivrera une valeur matricielle. Ceci a impliqué le développement de règles de conversion implicite entre type de données. Il existe deux familles de règles de conversion : l'un implicite lors des opérations usuelles sur les types simples et l'autre, plus complexe, lors d'opérations entre valeurs de type simple et de type matriciel ou de type matriciel entre eux (mais à dimensions différentes). Les premières règles de conversion entre types simples sont celles couramment développées : entier non signé → entier, entier non signé → réel non signé, entier → réel, réel non signé → réel, ... selon les opérations. Quant aux règles de conversion sur les structures matricielles, le principe est le suivant: on construit la structure matricielle de dimension d par recopie de la valeur à convertir par rapport aux différentes dimensions. Par exemple, la conversion d'un réel vers un vecteur de n réels c'est le vecteur composé de n fois le même réel. Ce type de conversion a été adopté pour permettre l'évaluation de l'opérateur addition (mais aussi de tous les autres). En effet, la somme d'un réel et d'un vecteur c'est un vecteur dont chaque élément est égal à la somme entre le réel et l'élément correspondant du vecteur initial. Les règles de conversion sont utilisées pour effectuer l'évaluation des opérateurs et des fonctions mais aussi pour la génération aléatoire. Si les paramètres d'un générateur

aléatoire sont de type matricielle (après conversion, si nécessaire), la valeur générée doit être aussi de type matricielle.

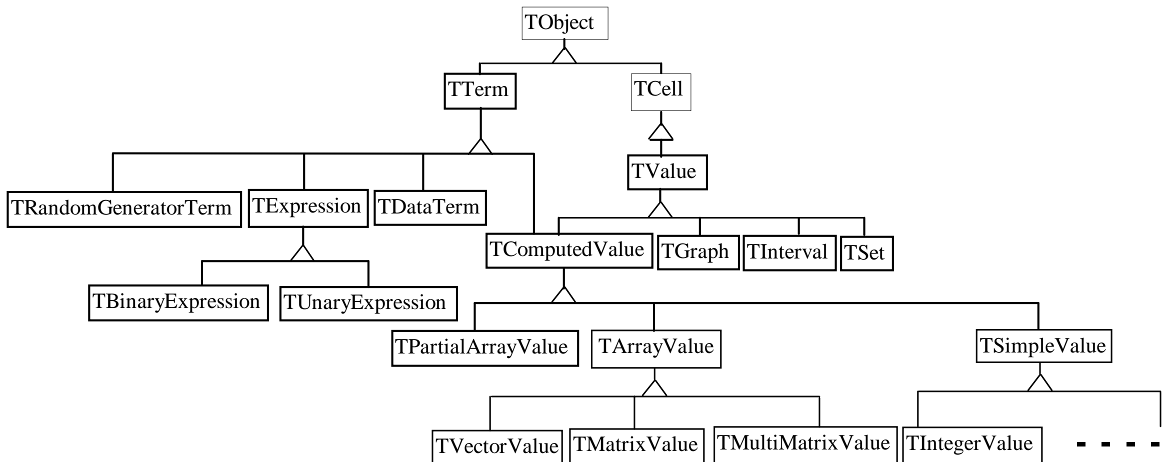

Figure 9. Hiérarchie partielle des valeurs

3.2.3 Affectation

Nous avons montré au 3.2.1 que l'on spécifiait l'affectation en référençant partiellement la donnée à affecter. De plus, le résultat de l'évaluation de la partie droite de l'affectation peut être de type simple ou matricielle. Il faut donc pouvoir indiquer de manière non ambiguë l'affectation.

Prenons un exemple:

C:Vector[3] of Real;
M:Matrix[3,3] of Real;

C=Uniforme<-10,10>;
M[_,<1>]=Normal<C,1>;

Le vecteur C représente un point dans $\Re^3$. Les valeurs de C suivent une loi uniforme entre -10 et 10, ce qui signifie que C représente un point dans le cube $[-10,10]^3$.La matrice M, quant à elle, représente un simple tableau de trois points de $\Re^3$ qui seront générés autour du point C suivant un noyau gaussien. L'expression *Normal<C,1>* génère donc un vecteur de la même taille que C (soit 3) dont chaque valeur suit une loi normale de moyenne C[i] et d'écart-type 1. Si on utilise la syntaxe suivante *M[_,_]* pour spécifier la partie de la matrice à affecter, on ne peut pas savoir si le vecteur généré est à affecter à une ligne ou à une colonne (étant donné que celles-ci ont la même taille). Il faut donc spécifier selon quelle dimension (ou axe) on réalise l'affectation d'où l'écriture *M[_,<1>]* qui signifie : les valeurs selon la première dimension de la structure matricielle générée seront affectées selon les colonnes de la matrice. Autrement dit, ce petit exemple permet de tirer un point uniformément dans un cube qui nous servira de mode pour tirer trois

points selon une loi normale centrée sur ce mode. Les colonnes de M coderont les coordonnées des différents points représentés par les lignes.

On peut généraliser de la manière suivante : *M[_,<1>,<2>]* signifie que l'expression à évaluer va générer une matrice (de dimension 2) et que les valeurs selon la première dimension de celle-ci serviront à affecter la deuxième dimension de la matrice 3D qui doit recevoir le résultat et les valeurs selon la deuxième dimension, la troisième dimension. Le souligné, quant à lui, signifie que l'on doit effectuer l'évaluation de l'expression et l'affectation autant de fois que la taille de la matrice M selon la première dimension.

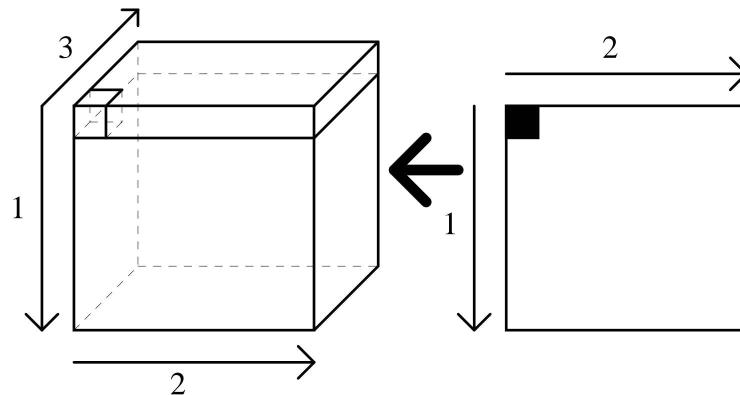

Figure 10. Exemple d'affectation

3.2.4 Référence partielle à une donnée

Lors de l'établissement d'une expression, il est indispensable de pouvoir adresser une variable. Or les variables peuvent être de deux types. Les types simples ne posent aucun problème. En revanche, si on fait référence à une variable de type matrice, deux approches sont possibles : soit on fait référence à la variable toute entière, soit à une partie de celle-ci (une ligne particulière, par exemple). Dans un premier temps, on peut se contenter de la notation utilisée au paragraphe précédent en utilisant les soulignés et en généralisant les entiers aux expressions entières. Mais cette notation est insuffisante dans le cas où l'un des indices pour la spécification de la référence partielle est dépendant de l'un des indices utilisés pour la structure à affecter.

Un petit exemple s'impose:

```
N : Integer;
V1 : Vector [3] of Real;
V2 : Vector [3] of Real;
M1 : Matrice[N,3] of Real;
M2 : Matrice[N,3] of Real;

N=200;
V1=[-4,-4,-4];
V2=[4,4,4];
M1[_,<1>]=Discret<<0.5,V1><0.5,V2>>;
M2[_,<1>]=Normal<M1[<1>,_],2>;
```

La méthode de construction de la matrice M1 a été étudiée dans un exemple précédent. Chaque ligne de M1 est un des vecteurs V1 ou V2, le choix étant réalisé par un tirage discret entre les deux vecteurs avec des probabilités a priori égales à 0.5.

La matrice M2 est, elle aussi, construite par l'affectation de N vecteur-ligne. Mais l’expression qui définit M2 dépend d'une autre matrice M1, dans le sens où une ligne de la matrice M2 est le vecteur qui résulte du tirage selon une loi normale de vecteur-moyenne égale aux valeurs de la ligne correspondante dans M1. On note donc ceci par <1> sur la première dimension de M1 ce qui signifie que l'indice 1 de M1 est égal à l'indice 1 de M2 lors de l'affectation. On peut généraliser cette approche aux structures matricielles de dimension quelconque. L’exemple précédent permet ainsi de générer 200 points dans un cube, chacun de ces points étant tiré selon une loi normale de vecteur-moyenne l’un des deux points V1 et V2. Cela correspond à un échantillon issu d’une loi de densité égale à la moyenne de deux lois normales.

3.3 Définition et gestion d’une méthode

Cette troisième partie concerne l'intégration dans l'atelier des algorithmes à tester Dans l'expérience sur les réseaux de neurones décrite précédemment les algorithmes à tester étaient des classifieurs à base de réseaux de neurones.

Du point de vue de l'environnement, l'algorithme à tester est donc définissable à travers ce que nous avons appelé méthode. Une méthode n'est qu'un nom logique, qui couplé à un nom de fichier DLL[3], nous permet de faire le lien avec le code exécutable de l'algorithme. En effet, tout algorithme doit être encapsulé dans une DLL qui, du point de vue de l'atelier, livre une fonction appelée « run » et prenant deux paramètres : le répertoire où sont disponibles les fichiers contenant les données d'entrée et le répertoire où doit être sauvegardé le fichier de résultat. Le contenu de ce fichier de résultat est spécifié lors de la définition d'une nouvelle méthode à l'aide de la rubrique *Critères*.

---

[3] Une DLL est une bibliothèque de fonctions autonomes qui est lié dynamiquement à un exécutable (ou à une autre DLL) lors de l'exécution de celui-ci.

La définition d'une nouvelle méthode est complétée par trois éléments supplémentaires: le format de données que désire l'algorithme, le nom du problème que résout l'algorithme et la liste des paramètres de l'algorithme.
Il est à noter que le format des données est considéré comme une caractéristique de l'algorithme à tester. En effet, si le format "standard" n'est pas satisfaisant, il est possible d'étendre les formats disponibles par ajout de modules DLL.
Attardons-nous sur la définition des paramètres d'un algorithme. Un algorithme peut nécessiter pour son paramètrage un ensemble de valeurs. L'outil permet lors de la définition logique du lien avec l'algorithme, de lister les paramètres et leurs types respectifs (uniquement de type simple). Cette définition sera utilisée lors de la sélection des algorithmes à tester. L'utilisateur devra, en effet, spécifier l'ensemble des valeurs que prendront les paramètres lors de l'exécution de ceux-ci. La syntaxe utilisée est identique à celle développée pour la définition des valeurs à affecter aux données à multiples modalités.

L'exécution d'un plan d'expérience, après définition du plan d'expérience, nécessite tout simplement la sélection des algorithmes à tester. Au passage, l'outil demande de fixer des valeurs aux paramètres ou de laisser les valeurs par défaut. Ensuite, l'atelier va générer, de manière successive, les données et lancer les algorithmes.

4. Conclusion

L'utilisation de cet atelier informatique d'expérimentation sur les logiciels demande sûrement un minimum de formation. Une fois maîtrisée le langage puissant de déclaration et de génération de données, il est extrêmement aisé de produire des données selon des modèles complexes comme c'est le cas dans l'application présentée. Les possibilités de l'outil ont été également exploitées pour tester des heuristiques en ordonnancement. Il appartient aux lecteurs d'en imaginer d'autres utilisations et de nous en faire part, ce dont nous les remercions d'avance.

5. Références

[Asselin de Beauville 73] ASSELIN de BEAUVILLE J. P., *Les sous-programmes usuels de Simulation Statistique*, Note Interne, UER Aménagement-Géographie-Informatique, Mai 1973

[GOThA 93] GOThA, Les problèmes d'ordonnancement, Recherche opérationnelle/ Operation Research, vol 27, n°1, p77-150, 1993

[Kolisch 95] KOLISCH R., SPRECHER A. et DREXL A., Characterization and generation of a general class of Resource-constrained project scheduling problems, Management Science, vol 41, n°10, 1995

[L'Ecuyer 96] L'ECUYER P. et CORDEAU J.F., Tests sur les points rapprochés pour des générateurs pseudo-aléatoires, actes des XXVIIIèmes journées de Statistiques, p479-482, Québec,1996

[Nadreau 96] NADREAU J.C., RAMAT E. et VIGNIER A., Super Locho, Rapport de fin d'études, Laboratoire d'Informatique/E3i, Tours, 1996.

[Taillard 93] TAILLARD E., Benchmarks for basic scheduling problems, European Journal of Operational Research, n°64, p279-285, 1993

[Verley 1994] VERLEY, G., ASSELIN DE BEAUVILLE, J.P. Validation des systèmes de reconnaissance des formes: une contribution méthodologique. *Actes du deuxième colloque Automatique et Génie Informatique AGI '94, Poitiers.* Juin 1994

[Verley 1994] VERLEY, G., ASSELIN DE BEAUVILLE, J.P., RAMAT E., LECLERCQ N., Différences théoriques et expérimentales entre les réseaux de neurones multicouches et le classifieur bayésien optimal. *Actes du colloque sur le Neuromimétisme, Lyon*, p. 217-220. Juin 1994

[Verley 1996] VERLEY, G., ASSELIN DE BEAUVILLE, J.P , Multilayer Perceptrons Learning Control, Lectures Notes in Computer Sciences, Vol. 1124, tII, Ed. Springer Verlag, ISBN 3-540-61627-6, pp 377-386, 1996.

[Verley 1996] VERLEY, G., ASSELIN DE BEAUVILLE, J.P, LENTE C., SLIMANE M., Prise en compte de connaissances a priori pour améliorer la fiabilité des classifieurs supervisés, actes des XXVIIIèmes journées de Statistiques, Québec,1996

[Wallace 96] WALLACE C. S., Fast pseudorandom generator for normal and exponential variantes, ACM Transactions on Mathematical Software, vol 22, n°1, p119-127, 1996

GASCUEL, O., GALLINARI P. Statistiques et apprentissage: application aux réseaux de neurones. *Tutoriel 3 de RFIA'9*, Paris, 36 p. 1994.

HAMPSHIRE, J.B., PEARLMUTTER, B.A. Equivalence proofs for multilayer perceptron classifiers and the Bayesian discriminant function. *Proc. 1990 Connectionnist Models Summer School*, Morgan Kaufmann, 1991.

PARZEN, E. An estimation of a probability density function and mode. *Ann. Math. Statist.,* Vol. 33, p. 1065-1076. 1962.

PAUGAM-MOISY, H. A Selecting and parallelizing neural networks for improving performances. *Artificial Neural Networks*, Kohonen & al. editors, Elsevier Sc. Pub., North-Holland, vol. I, p. 659-664. Juin 1991.

PERNOT, E. Problèmes posés par l'évaluation et la comparaison de classifieurs. *R.I. Thomson-CSF/LCR*, 8 p. 1994

PERSONNAZ, L. *Etude de réseaux de neurones formels: conception, proptiétés et applications*. Thèse de doctorat d'état. Univ. Paris 6. 1986.

VAPNIK, V. N. *Estimation of dependences based on empirical data.* Springer series in statistics, Springer Verlag. 1982.

WHITE, H. Learning in artificial neural networks: a statistical perspective. *Neural comp. 1*, p. 425-464, 1989.