1 Objectif

Configurer un perceptron multicouche.

Au premier abord, les réseaux de neurones artificiels semblent toujours un peu énigmatiques. Justement parce qu'il est question de « neurones », et que la plupart des références qui les présentent commencent quasiment toujours par des grandes envolées sur les métaphores biologiques, faisant miroiter la perspective de doter les machines d'une intelligence. Pour l'étudiant que j'étais, tout ça me paraissait bien inaccessible… et surtout ça ne répondait pas du tout aux questions simples – mais bon on est là pour réaliser des traitements après tout – que je me posais : Est-ce que la de méthode est applicable à l'analyse prédictive ? Pour quelles configurations de données et classes de problèmes est-elle la plus performante ? Comment fixer les valeurs des paramètres – s'ils existent – associés à la méthode pour la rendre plus efficace sur mes données ?

Avec le temps, et une lecture assidue de la documentation y afférente, j'ai fini par comprendre que le perceptron multicouche[1] était l'approche la plus populaire en apprentissage supervisée, qu'elle était très performante pourvu que l'on sache définir correctement le nombre de neurones dans les couches cachées. Deux extrêmes tout aussi funestes sont à éviter : si on n'en met pas assez, le modèle est peu compétitif, incapable de retranscrire la relation (le concept) entre la variable cible et les prédicteurs ; si on en met trop, le modèle devient trop performant sur l'échantillon d'apprentissage, ingérant les particularités de ce dernier, non transposables dans la population.

Ceci étant posé, on vient alors nous annoncer qu'il n'y a pas de méthode universelle pour définir le bon paramétrage et qu'il faut en réalité tâtonner en s'appuyant sur des heuristiques plus ou moins heureuses. Tout ça paraît assez obscur. L'incompréhension vient souvent du fait que, d'une part, on a du mal à appréhender le rôle des neurones de la couche intermédiaire – si l'on s'en tient à une seule couche cachée - dans la modélisation ; d'autre part, on perçoit mal également l'impact de l'augmentation ou la diminution de leur nombre.

Dans ce tutoriel, nous allons essayer d'expliciter le rôle des neurones de la couche cachée du perceptron. Nous utiliserons des données générées artificiellement pour étayer notre propos. Nous nous en tenons à 2 variables explicatives afin de pouvoir projeter les données dans le plan et ainsi décrire concrètement le comportement de l'approche. Nous travaillons avec **Tanagra 1.4.48** dans un premier temps puis, dans un second temps, nous proposerons une petite routine de détection automatique du nombre optimal de neurones pour **R 2.15.2** (package **nnet**).

2 Données

Le fichier « **artificial2d.xls** » contient deux jeux de données de n = 2000 observations décrites par p = 2 variables X1 (définie dans [-0.5 ; 0.5]) et X2 ([0 ; 1]). La variable cible Y est binaire {pos, neg}.

Le premier ensemble « **data2** » correspond à une liaison fonctionnelle relativement simple, elle est définie de la manière suivante :

[1] Voir http://eric.univ-lyon2.fr/~ricco/cours/slides/reseaux_neurones_perceptron.pdf ; http://www.grappa.univ-lille3.fr/polys/apprentissage/sortie005.html.

Si (0.1 * X2 > X1²) **Alors** Y = pos **Sinon** Y = neg

Le concept sous-jacent à « **data4** » est un peu plus complexe

Si (X1 < 0)

Alors Si (0.5 * X2 > 0.5 - |X1|) **Alors** Y = pos **Sinon** Y = neg

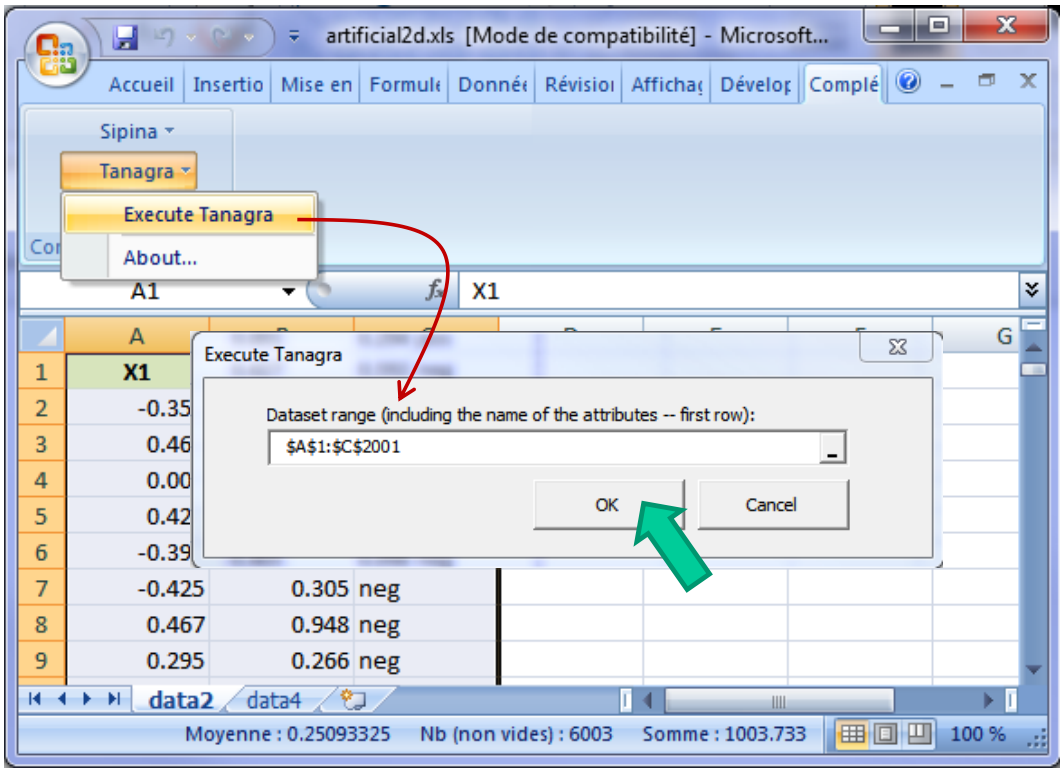
Sinon Si (X2 > X1 et 1 – X2 > X1) **Alors** Y = pos **Sinon** Y = neg

Bien évidemment, nous ne disposons pas de ces informations dans les études réelles. Nous comptons justement sur le système d'apprentissage pour tenter de les reconstituer ou, tout du moins, de les approximer. L'idée de ce tutoriel est de situer le comportement du perceptron sur des données dont on connaît les tenants et aboutissants.

3 Perceptron simple avec Tanagra

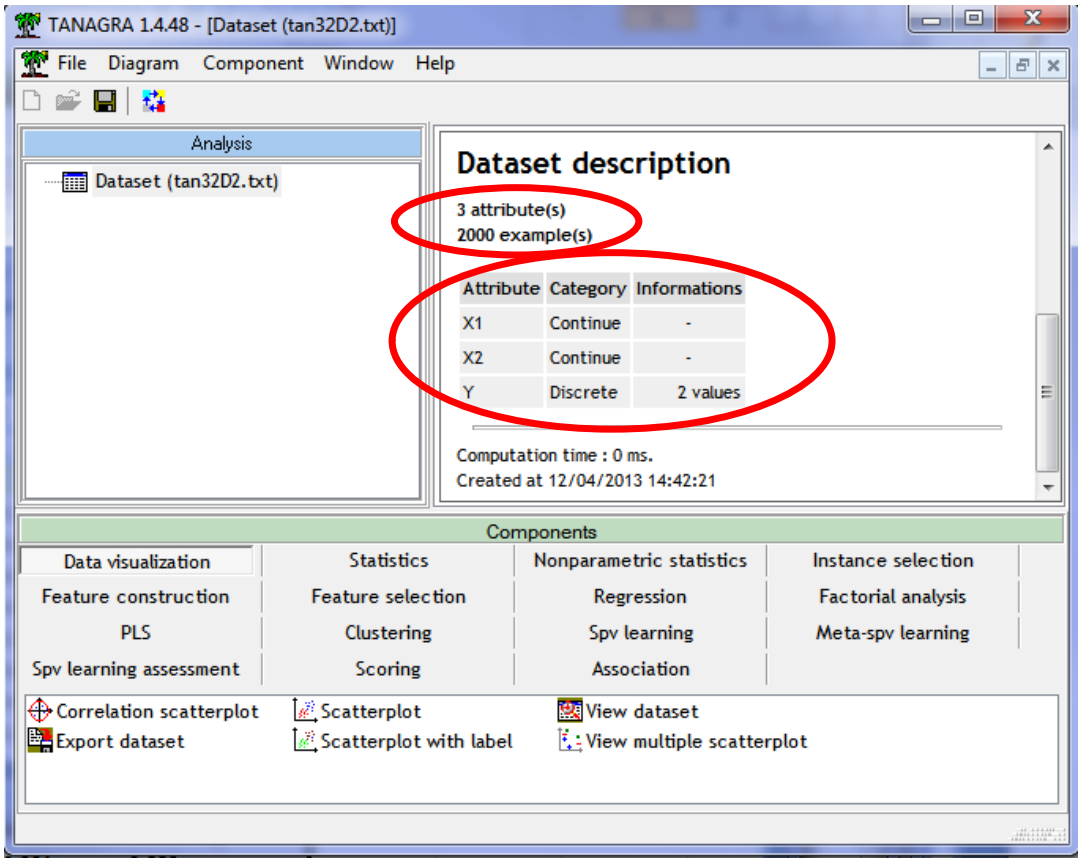
3.1 Importation des données

Nous ouvrons le fichier « **artificial2d.xls** » dans le tableur Excel. Nous sélectionnons la première feuille « **data2** » et, après avoir sélectionné la plage de données, nous les envoyons à Tanagra via la macro-complémentaire « tanagra.xla »[2].



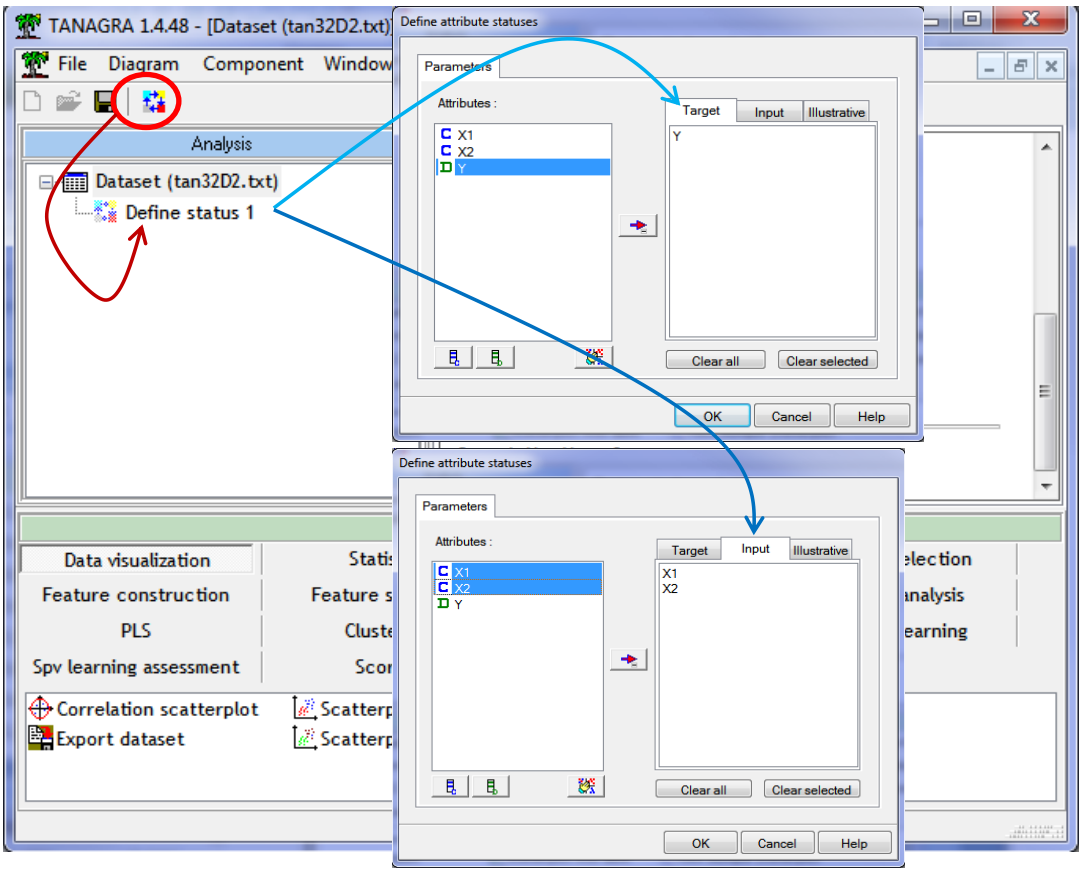
Tanagra est automatiquement démarré et les données chargées, nous disposons de n = 2000 observations et 3 variables, dont p = 2 prédicteurs.

[2] Voir http://tutoriels-data-mining.blogspot.fr/2010/08/ladd-in-tanagra-pour-excel-2007-et-2010.html



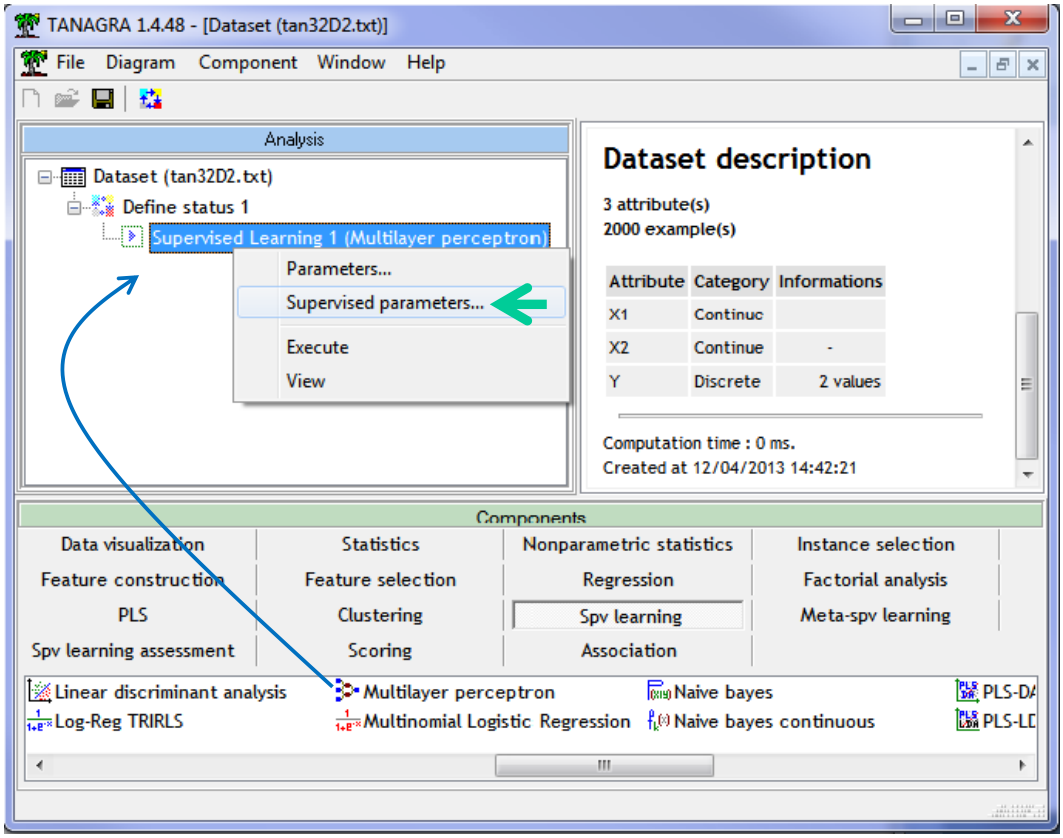
3.2 Perceptron sans couche cachée – Perceptron simple

Avant de lancer l'apprentissage, nous devons définir le rôle des variables.

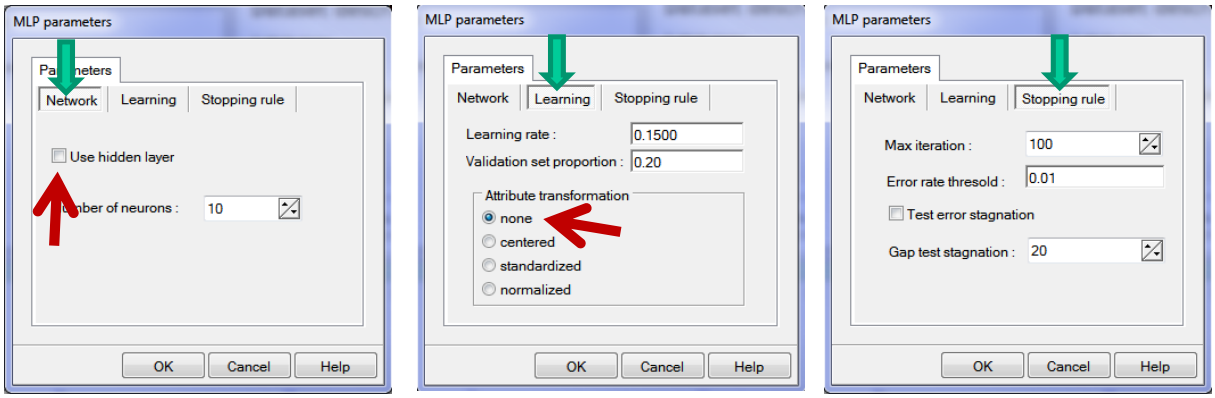


Nous utilisons le composant DEFINE STATUS pour ce faire. Nous cliquons sur le raccourci de la barre d'outils. Nous plaçons Y en TARGET, X1 et X2 en INPUT.

Pour construire le réseau de neurones, nous insérons le composant MULTILAYER PERCEPTRON (onglet SPV LEARNING) dans le diagramme.



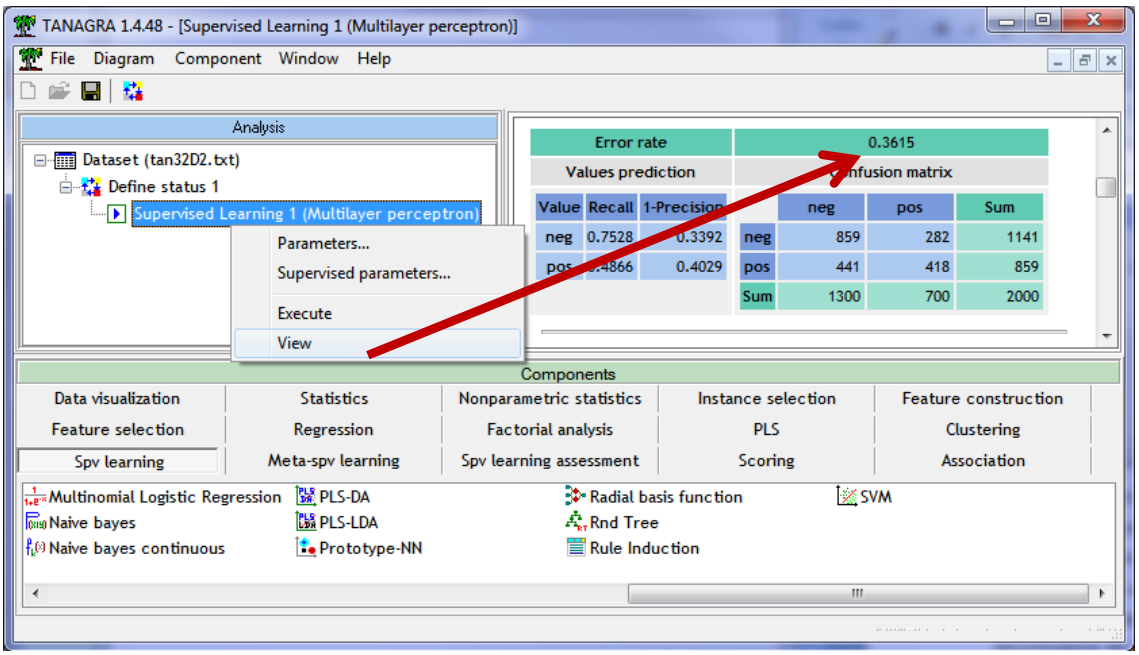
Nous le paramétrons en actionnant le menu contextuel **SUPERVISED PARAMETERS**.



Nous souhaitons mettre en œuvre un perceptron simple. Pour cela, nous désactivons la couche cachée en décochant l'option « **Use Hidden Layer** » dans l'onglet NETWORK. Dans l'onglet LEARNING, nous choisissons de ne pas transformer les données « **Attribute Transformation = none** ». Notons au passage que, par défaut, Tanagra scinde les données en 2 parties : la première, 80%, sert à l'apprentissage ; la seconde, 20% (Validation set proportion), sert à surveiller la décroissance de l'erreur. Cette dernière n'intervenant pas dans le processus de calcul des poids, il permet d'obtenir une estimation plus fidèle des performances en généralisation du réseau et éviter

ainsi le phénomène de surapprentissage. Dans l'onglet STOPPING RULE, nous notons que le processus est stoppé après 100 itérations ou lorsque le taux d'erreur est inférieur à 1%.

Après avoir validé ces paramètres, nous lançons les calculs en actionnant le menu VIEW.



Le taux d'erreur en resubstitution, calculée sur les n = 2000 observations, est de 36.15%. Plus loin, nous constatons qu'elle se décompose en 36.19% sur les 80% des observations ayant servi au calcul des poids synaptiques, et 36% sur l'échantillon de validation (20% des observations)[3].

Learning characteristics

Epochs	100
Last train error rate	0.3619
Last validation error rate	0.3600
Last train mse	361.1898

Nous avons ensuite les poids reliant la couche d'entrée à la couche de sortie. La variable cible étant binaire, les coefficients sont identiques avec des signes contraires sur les deux neurones de sortie.

Weigths

From INPUT to OUTPUT layer

-	neg	pos
X1	-0.08557568	0.08557568
X2	-2.10453600	2.10453600
bias	1.36340152	-1.36340152

[3] ATTENTION, les poids étant initialisés aléatoirement, les données étant également mélangées tout aussi aléatoirement, il y a de fortes chances que vous n'ayez pas exactement les mêmes valeurs. Vous devriez obtenir le même ordre de grandeur néanmoins.

3.3 Commentaires – Insuffisances de la frontière linéaire

Manifestement, les classes ne sont pas séparables linéairement. Puisque nous n'avons que 2 variables, nous pouvons les représenter dans le plan. Nous insérons le composant SCATTERPLOT (onglet DATA VISUALIZATION) à la racine du diagramme. Nous spécifions X1 en abscisse, X2 en ordonnée, et **nous utilisons la cible observée Y pour distinguer les points**.

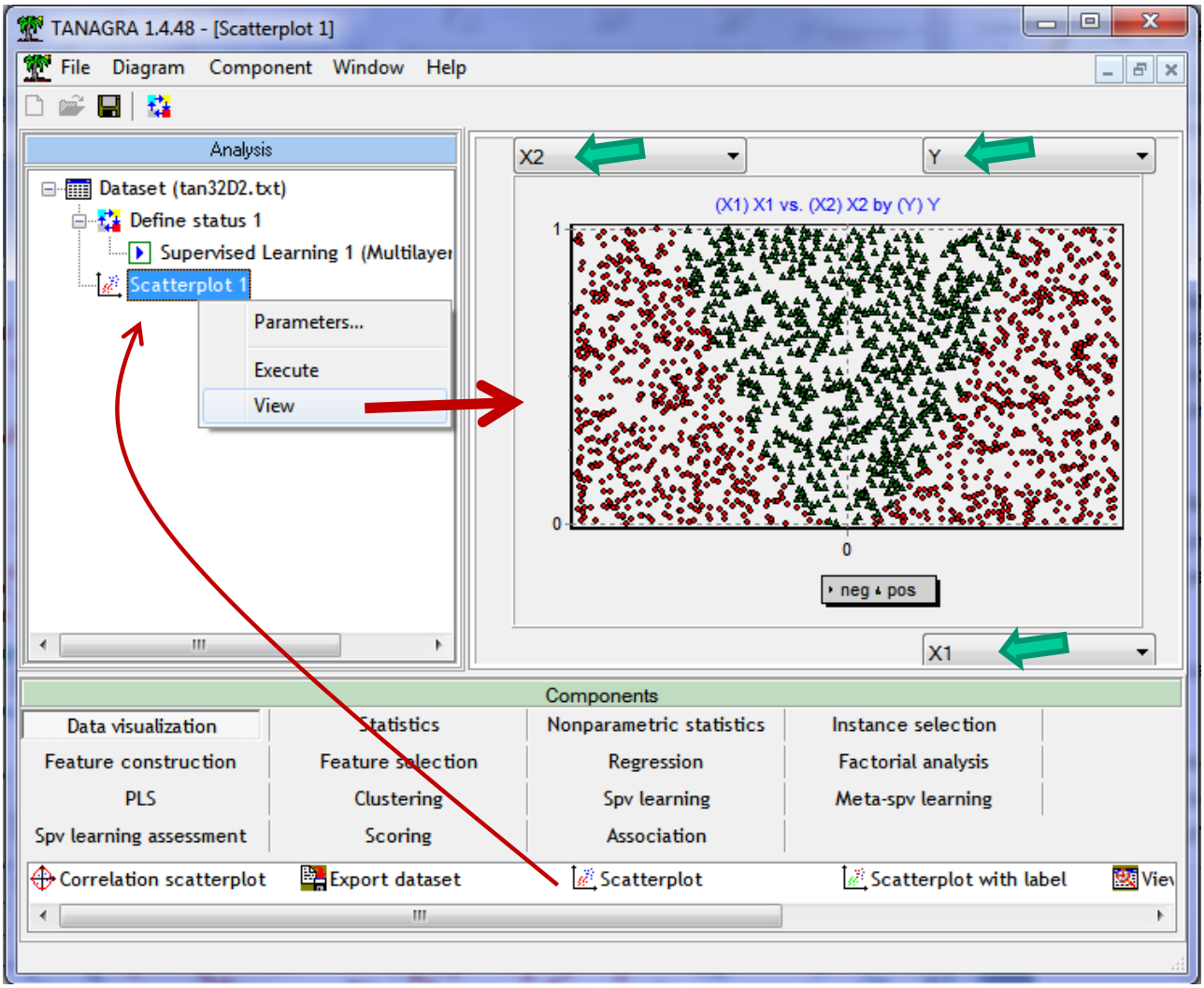


Figure 1 - Projection des points dans (X1, X2) - Etiquettes observées

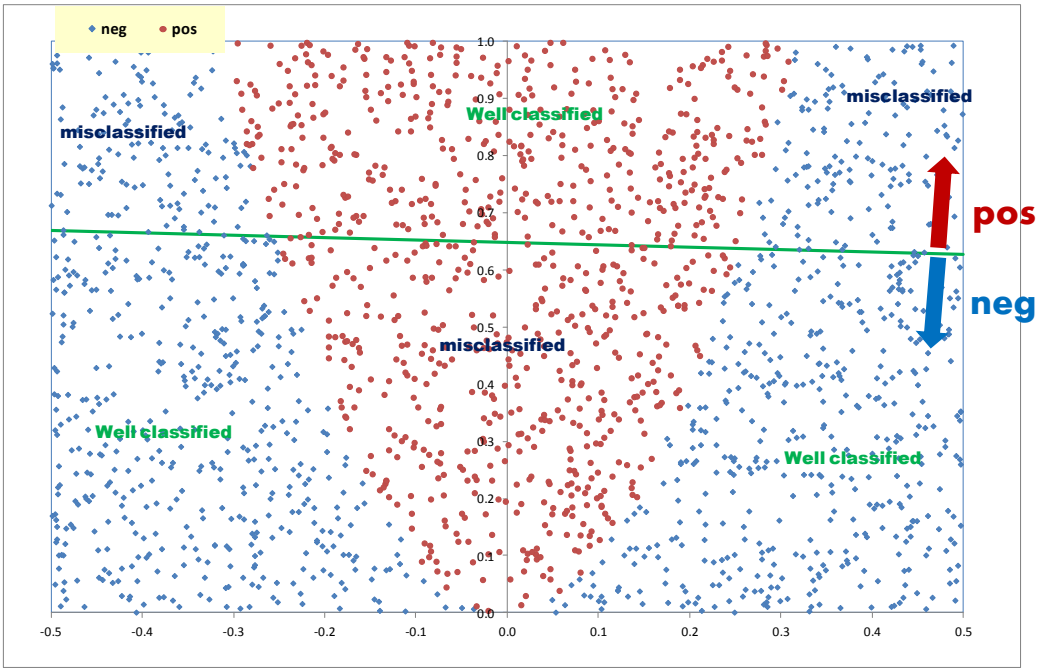
Nous aurions du commencer par là. La frontière est plutôt de forme parabolique, il est impossible de produire une droite (unique) permettant d'isoler les « pos » des « neg ». Dans notre cas, le perceptron simple a induit la droite séparatrice dont voici l'équation implicite :

$$-0.0856 * X1 - 2.1045 * X2 + 1.3634 = 0$$

La forme explicite s'écrit :

$$X2 = -0.0407 * X1 + 0.6478$$

Nous pouvons la tracer dans l'espace de représentation. Nous constatons que nous sommes vraiment loin du compte. Nous distinguons nettement les individus mal étiquetés « misclassified » de part et d'autres de la frontière, ils sont légions.



Il est possible d'obtenir un graphique similaire sous Tanagra. Nous plaçons le composant SCATTERPLOT à la suite du réseau de neurones dans le diagramme. Nous plaçons X1 en abscisse, X2 en ordonnées, puis **nous illustrons les points à l'aide de la prédiction** (PRED_SPVINSTANCE_1). Nous distinguons au-dessus (resp. en-dessous) de la frontière les individus classés « pos » (resp. « neg »), certains à tort (les mal classés), d'autres à raison (les bien classés).

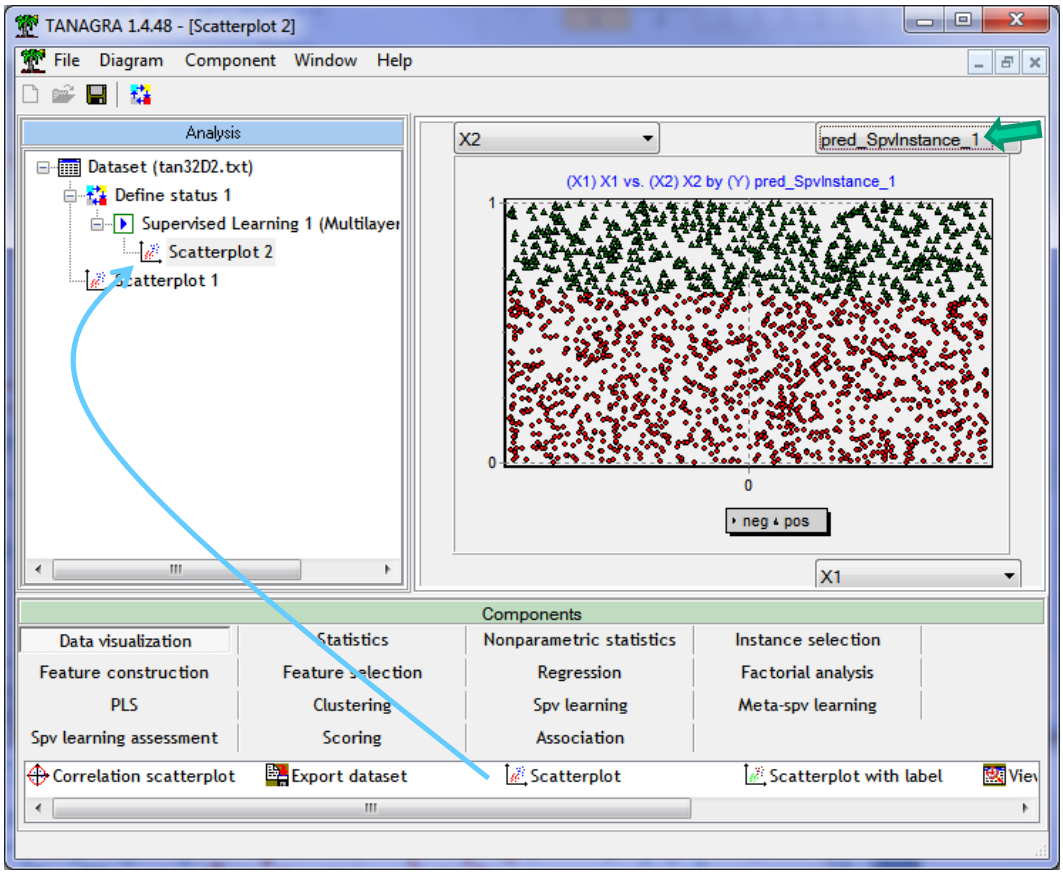


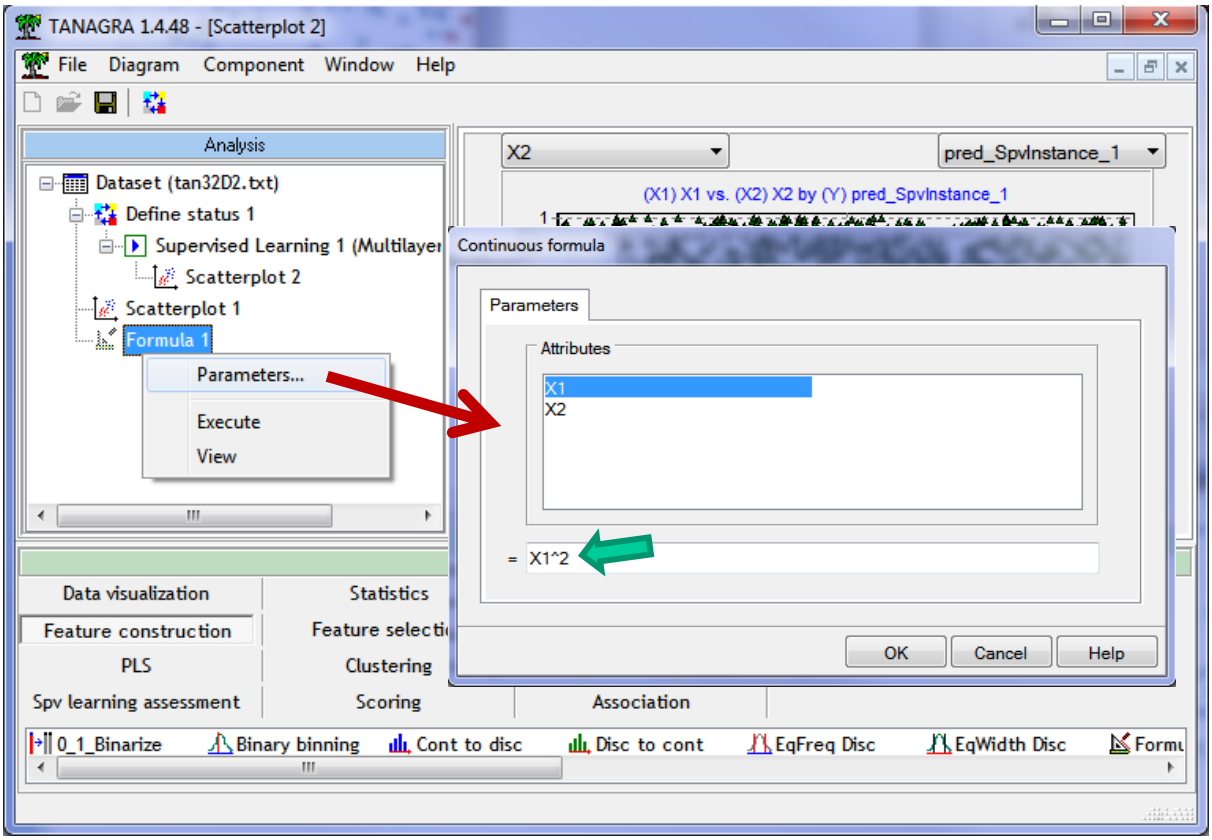
Figure 2 - Projection des points dans (X1, X2) - Etiquettes prédites

Remarque : De fait, tout séparateur linéaire serait incapable de produire un classifieur efficace pour nos données : que ce soit une régression logistique, une analyse discriminante linéaire, un SVM (support vector machine) linéaire. Il faut modifier le biais de représentation c.-à-d. la capacité du modèle à représenter des concepts plus complexes.

3.4 Transformation de variables

Une solution possible, qui tombe sous le sens au vu du graphique précédent[4], est d'opérer une transformation de variable. Nous modifions ainsi l'espace de représentation des données.

Nous choisissons de construire la variable Z1 = X1², puis la substituer à X1 dans le modèle. Dans Tanagra, nous pouvons la produire à l'aide du composant FORMULA (onglet FEATURE CONSTRUCTION). Nous l'insérons à la racine du diagramme, puis nous le paramétrons comme suit :



Une nouvelle variable avec la formule « X1^2 » est générée pour la branche du diagramme. Mais est-ce vraiment bénéfique ? Pour le savoir, nous replaçons l'outil SCATTERPLOT à la suite de « FORMULA 1 ». Nous mettons en abscisse la variable calculée FORMULA_1 (Z1), en ordonnée X2, nous illustrons les points avec la cible Y. Miracle ! Les classes sont maintenant linéairement séparables. Il doit être possible de tracer une droite pour séparer les « pos » des « neg » dans le nouveau repère.

[4] Dans le plan oui, la solution est évidente. Si nous avons plus de variables (p > 2), la situation devient compliquée. Il faut se tourner vers des outils similaires aux « résidus partiels » de la régression logistique. Voir « Pratique de la Régression Logistique – Régression logistique binaire et polytomique » (section 8.2.4) ; http://eric.univ-lyon2.fr/~ricco/cours/ouvrages.html

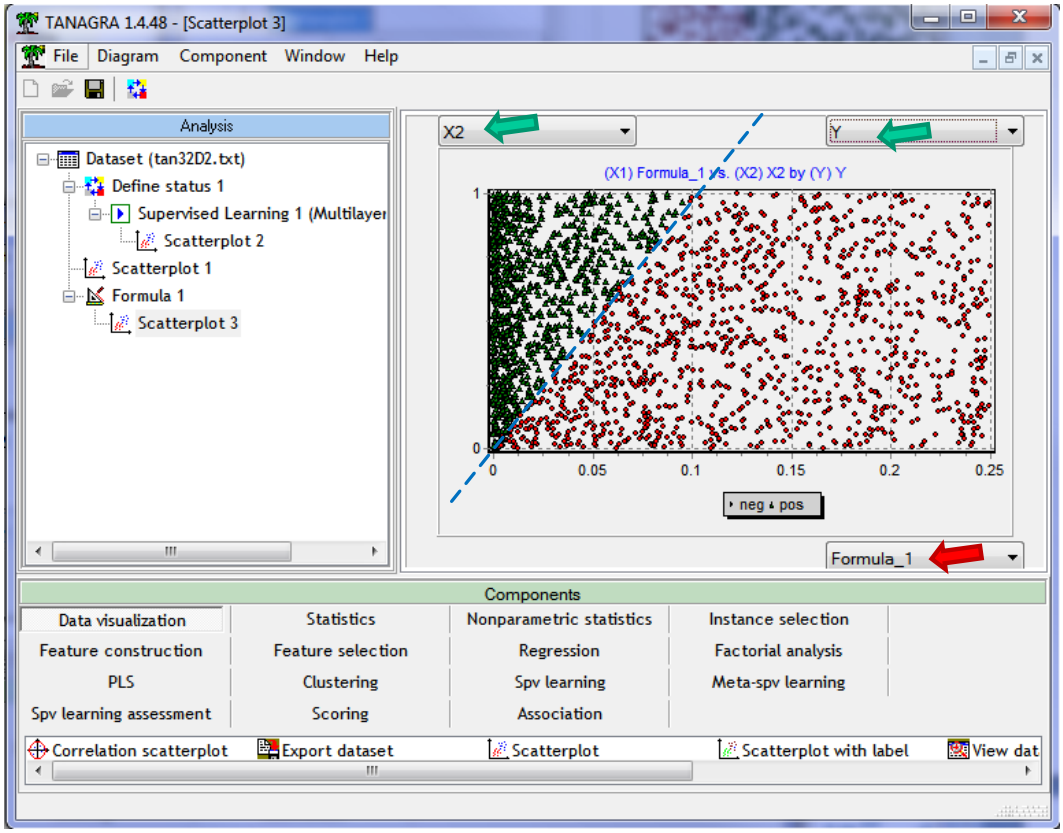
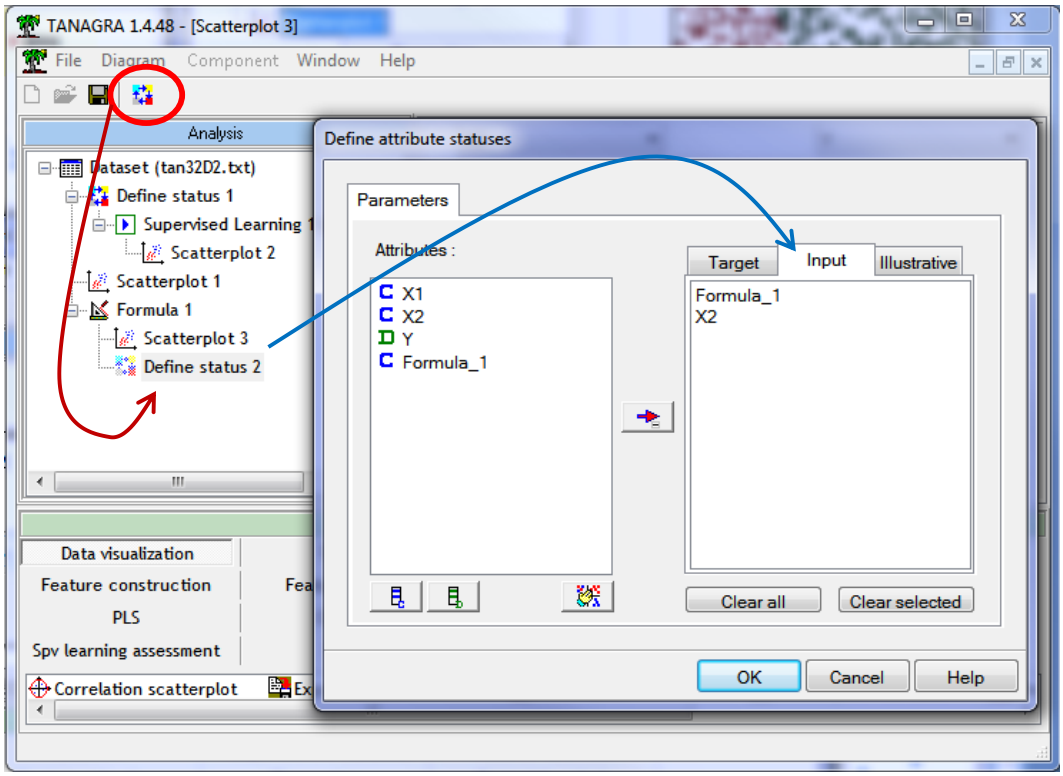
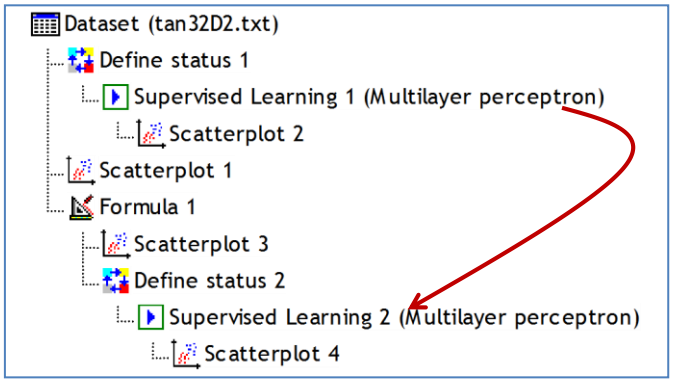


Figure 3 - Projection des points dans (Z1, X2) - Etiquettes observées

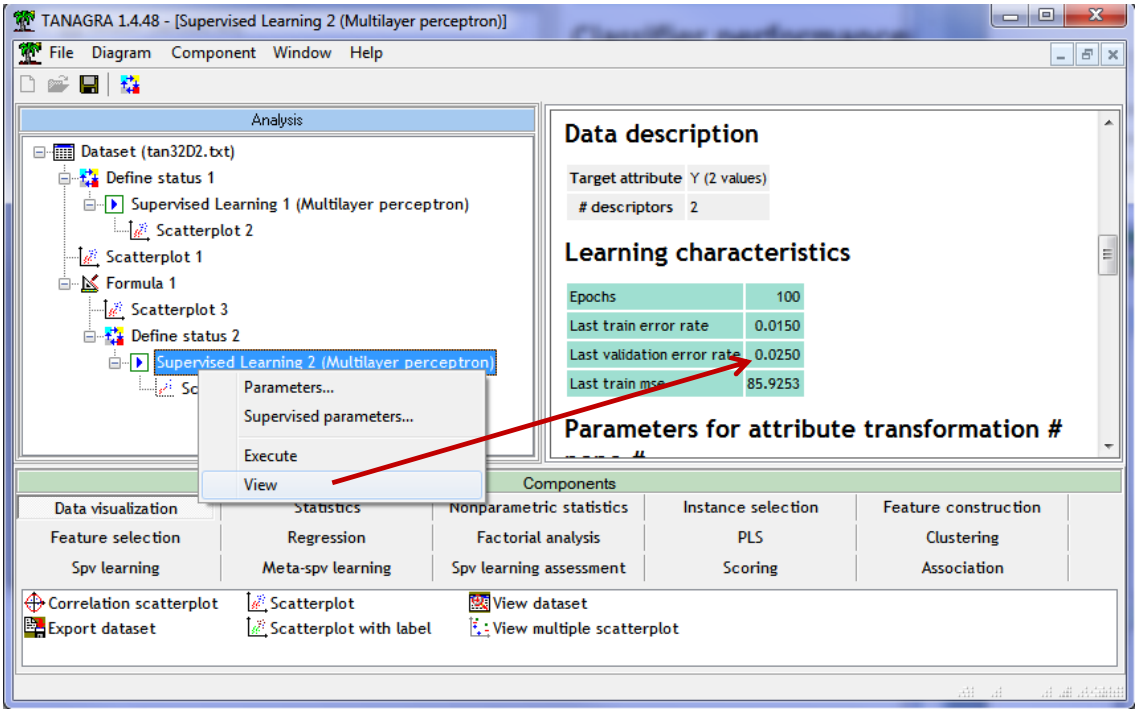
Voyons si le perceptron simple va savoir tirer profit de cette nouvelle configuration. Nous insérons le composant DEFINE STATUS à la suite de « FORMULA 1 ». Comme précédemment, nous plaçons toujours Y en TARGET, en INPUT nous choisissons maintenant FORMULA_1 (Z1) et X2.



Nous souhaitons insérer de nouveau le perceptron **en veillant à le paramétrer à l'identique**. Le plus simple pour cela consiste à déplacer à l'aide la souris le composant déjà inséré dans le diagramme. Il est dupliqué avec exactement les mêmes paramètres (Tanagra a aussi copié l'outil SCATTERPLOT qui lui était lié, ce n'est pas un souci).



Nous cliquons sur le menu VIEW du composant.



La transformation opérée porte ses fruits avec un taux d'erreur de 2.5% sur l'échantillon de validation. Le perceptron sans couche cachée est parfaitement adapté dans (Z1, X2). Nous obtenons ci-contre les poids synaptiques estimés.

Weigths

From INPUT to OUTPUT layer

-	neg	pos
X2	-3.90226725	3.90225817
Formula_1	36.07647058	-36.07636690
bias	-0.01056363	0.01056308

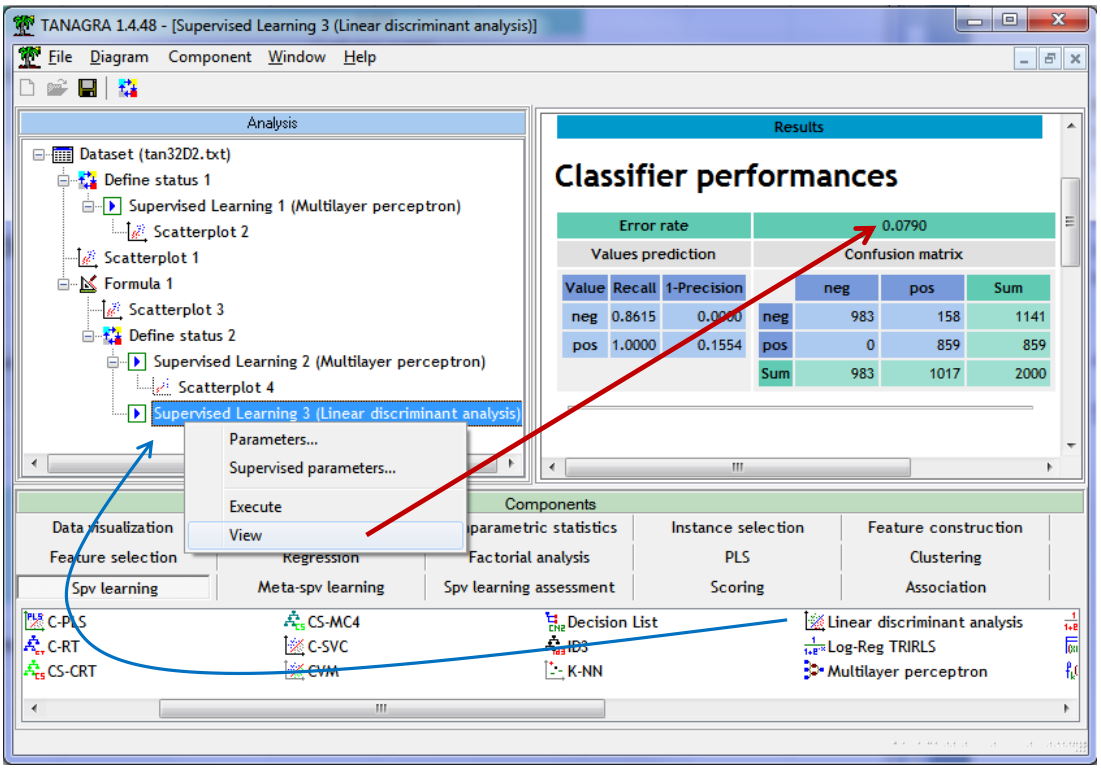
En passant à l'équation explicite de la frontière (avec Z1 = X1²), nous avons :

$$X2 = 0.1082 * X1^2 + 0.0003$$

L'estimation est de très bonne qualité[5]. En effet, rappelons que la frontière artificiellement utilisée lors de la génération des données était (voir Section 2) :

$$X2 = 0.1 * X1^2$$

Remarque : Ce n'est pas parce qu'un classifieur sait représenter une frontière qu'il sait forcément la trouver. Prenons le cas de l'analyse discriminante qui est aussi un séparateur linéaire (biais de représentation). La méthode s'appuie sur une hypothèse (biais d'apprentissage) – les formes des nuages de points conditionnels sont identiques – qui à l'évidence n'est pas respectée (Figure 3). De fait, si nous l'appliquons sur nos données transformées (Z1, X2) (composant LINEAR DISCRIMINANT ANALYSIS, onglet SPV LEARNING), le classifieur a une moins bonne tenue avec un taux d'erreur en resubstitution décevant de 7.9% (ça ne sera pas meilleur sur un échantillon test).



4 Perceptron multicouche avec Tanagra

4.1 Spécifier le nombre de neurones de la couche cachée

Trouver la bonne transformation de variables n'est pas aisée, voire impossible dans certaines situations. L'opération est de toute manière très fastidieuse dès que le nombre de variables augmente. Le perceptron multicouche permet d'améliorer le pouvoir de représentation du modèle en introduisant une couche intermédiaire, dite « cachée », s'intercalant entre les couches d'entrée et

[5] **Elle serait meilleure encore si nous avions augmenté le nombre d'itérations**. Nous nous sommes contentés d'un apprentissage à minima pour montrer la pertinence de la transformation de variables dans notre cas.

de sortie[6]. Il est dès lors possible d'apprendre n'importe quel type de fonction en mettant suffisamment de neurones dans la couche cachée[7].

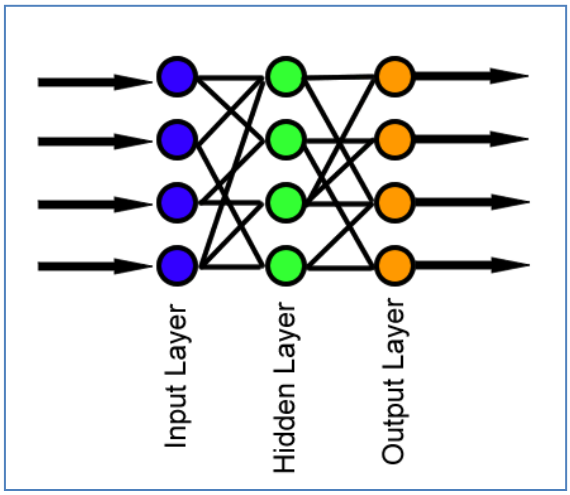


Figure 4 - Perceptron avec couche cachée - http://en.wikipedia.org/wiki/Feedforward_neural_network

A priori, « qui peut le plus peut le moins ». Dans le doute, on serait tenté de mettre un maximum de neurones dans la couche cachée. Mais, en exagérant la capacité de représentation de la structure, nous courrons le risque d'ingérer les informations spécifiques à l'échantillon d'apprentissage, qui ne sont pas transposables à la population. On parle de « surapprentissage ». Le modèle semble très performant sur nos données, il s'effondre lamentablement dès qu'on souhaite le déployer. « Point trop n'en faut » donc. Encore faut-il comprendre le fonctionnement du perceptron multicouche, en particulier l'action de la couche cachée, pour pouvoir identifier le bon nombre de neurones à introduire pour un problème donné.

Il y a deux prismes possibles :

1. La couche cachée permet de produire un espace de représentation intermédiaire où les classes deviennent linéairement séparables. On comprend bien l'idée avec la transformation de variables que nous avons effectuée ci-dessus. Mais dans le cas présent, cette piste ne nous donne pas d'indications sur le nombre optimal de neurones cachés à introduire.
2. Chaque neurone de la couche cachée permet de définir une droite séparatrice. La connexion entre la couche intermédiaire et la sortie permet de les combiner. Nous avons donc un modèle linéaire par morceaux. Le modèle dans sa globalité est non linéaire. Cette vision est plus intéressante dans notre situation. En effet, on se rend compte que la combinaison de deux droites permettra d'approcher au mieux la parabole qui sépare les « pos » des « neg » dans l'espace de représentation originel (Figure 1).

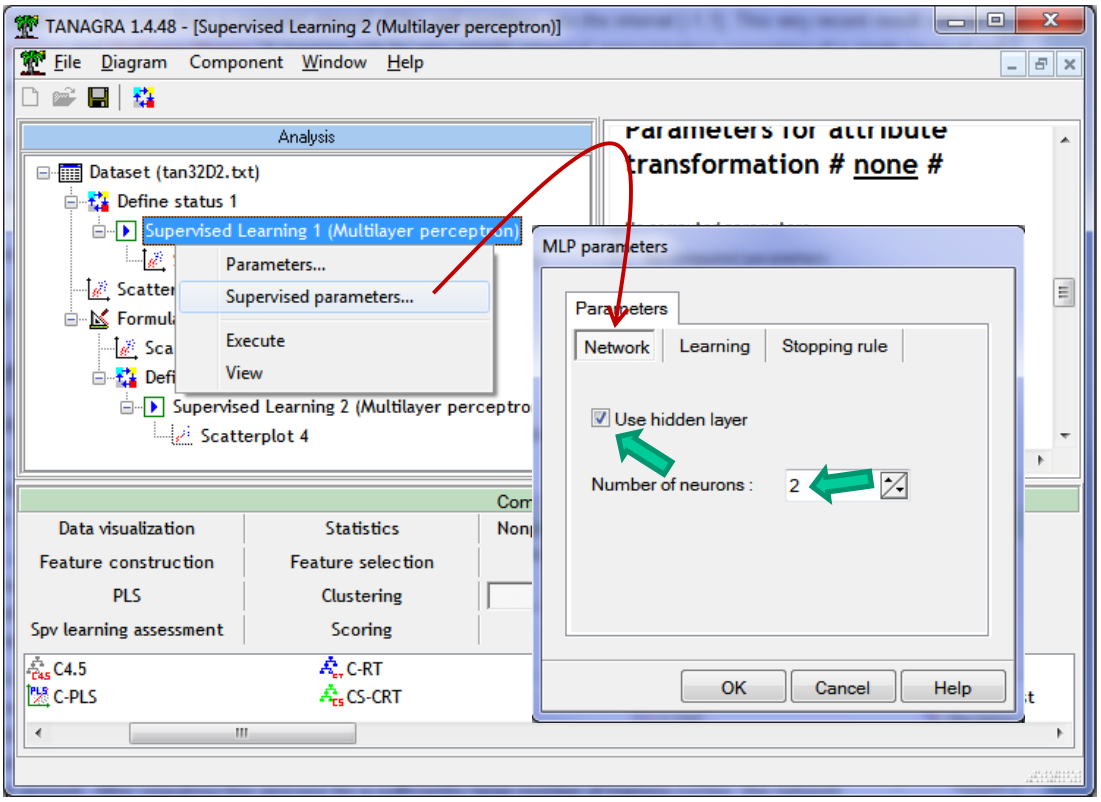
Bref, intégrer 2 neurones dans la couche cachée serait le plus adapté pour nos données.

[6] En théorie, on peut mettre plusieurs couches cachées. En pratique, et dans la très grande majorité des logiciels, on s'en tient à une seule couche cachée avec un nombre de neurones paramétrable.

[7] Voir http://en.wikipedia.org/wiki/Feedforward_neural_network

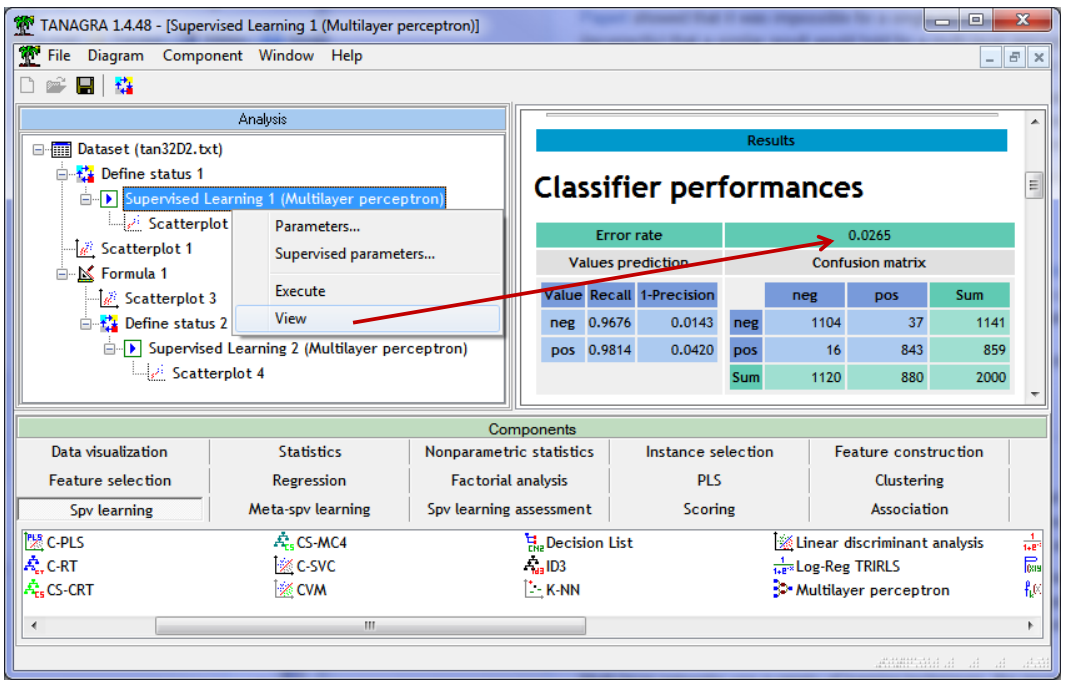
4.2 Perceptron multicouche dans Tanagra

Nous revenons sur l'item SUPERVISED LEARNING 1 (MULTILAYER PERCEPTRON) dans notre diagramme. Nous le paramétrons en actionnant le menu SUPERVISED PAREMETERS. Nous activons la couche cachée (« Use hidden layer ») et nous spécifions 2 neurones.



Nous validons et nous relançons l'outil avec le menu VIEW.

Le taux d'erreur global est 2.65%



Avec 2.62% sur l'échantillon d'apprentissage et 2.75% sur l'échantillon de validation.

Learning characteristics

Epochs	100
Last train error rate	0.0262
Last validation error rate	0.0275
Last train mse	28.6295

Nous retrouvons les poids synaptiques, reliant la couche d'entrée à la couche cachée d'une part, cette dernière à la couche de sortie d'autre part.

Weigths

From INPUT to HIDDEN layer

-	Neuron "1"	Neuron "2"
X1	19.91611619	-19.29262980
X2	-5.03061779	-4.78012858
bias	-1.61075723	-1.98975678

From HIDDEN to OUTPUT layer

-	neg	pos
Neuron "1"	13.09724306	-13.09724287
Neuron "2"	12.87895088	-12.87895070
bias	-6.48310191	6.48310182

4.3 Frontières dans l'espace originel

Reprenons les équations reliant la couche d'entrée à la couche intermédiaire.

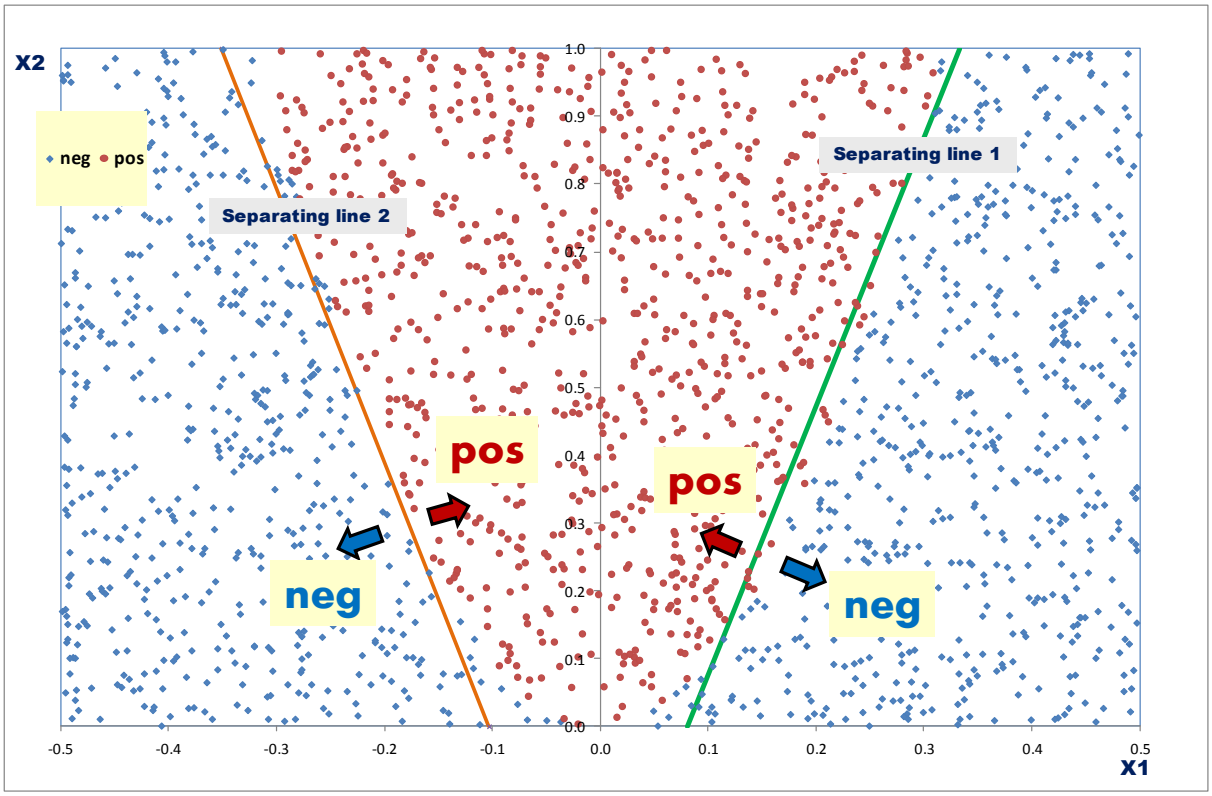
-	Neuron "1"	Neuron "2"
X1	19.9161	-19.2926
X2	-5.0306	-4.7801
bias	-1.6108	-1.9898

Mises sous une forme explicite, elles s'expriment comme suit :

Equation 1 (Frontière 1) : X2 = 3.9590 * X1 – 0.3202

Equation 2 (Frontière 2) : X2 = -4.0360 * X1 – 0.4163

Rapportées dans l'espace de représentation originel (X1, X2), nous distinguons les deux frontières induites par les 2 neurones de la couche cachée pour séparer les « pos » et les « neg ». Nous pouvons dès lors identifier les individus mal classés, situés du mauvais côté des frontières.



4.4 Frontière dans l'espace intermédiaire

L'autre manière d'appréhender le perceptron consiste à plonger les observations dans l'espace intermédiaire (U1, U2) défini par les sorties des neurones de la couche cachée. Les fonctions de transformation s'écrivent :

$$U1 = \frac{1}{1 + e^{-(19.9161 \times X1 - 5.0306 \times X2 - 1.6108)}}$$

$$U2 = \frac{1}{1 + e^{-(-19.2926 \times X1 - 4.7801 \times X2 - 1.9898)}}$$

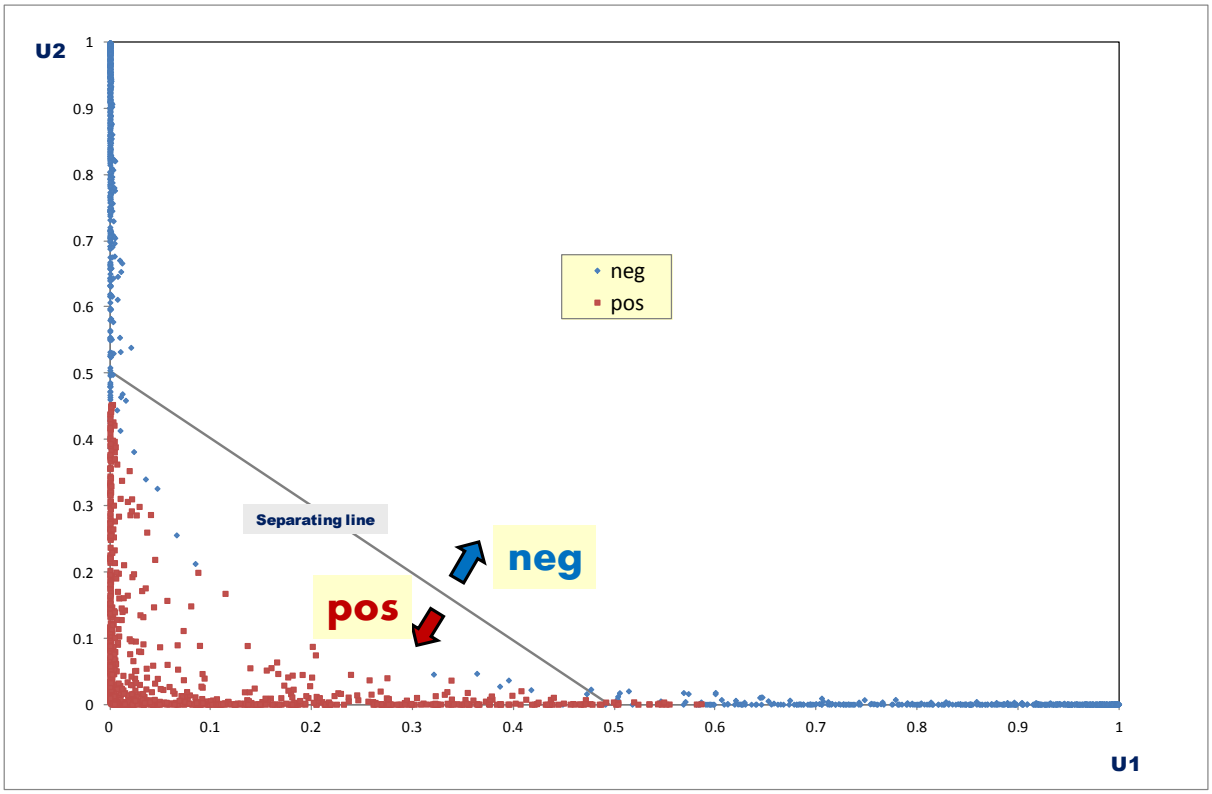
La droite séparatrice dans ce nouvel espace est spécifiée par les poids synaptiques reliant la couche intermédiaire à la sortie :

-	neg	pos
Neuron "1"	13.0972	-13.0972
Neuron "2"	12.8790	-12.8790
bias	-6.4831	6.4831

Soit, sous une forme explicite :

Frontière – Espace intermédiaire : U2 = -1.0169 * U1 + 0.5034

Voyons ce qu'il en est dans le plan défini par les variables (U1, U2) :



Les points mal étiquetés sont rares, ce sont les « neg » (resp. les « pos ») situés en dessous (resp. «au dessus) de la frontière. D'où le faible taux d'erreur.

Conclusion : Tout ça est quand même passionnant. Personnellement, j'ai mis du temps à intégrer ces mécanismes. On présente rarement le perceptron sous cet angle dans les ouvrages. Oui, le principe de l'apprentissage par correction d'erreur est important. Mais s'appesantir outre mesure sur l'algorithme du rétropropragation du gradient – que nous n'aurons jamais à calculer à la main de toute manière, c'est au logiciel de le faire – n'est pas toujours très heureux. Tout dépend du public auquel on s'adresse. En revanche, pouvoir identifier le bon nombre de neurones dans la couche cachée est un enjeu fort qui concerne tout le monde. A l'évidence, il sera très difficile de réaliser les bons choix si l'on perçoit mal les conséquences de l'adjonction de neurones dans la couche cachée.

5 Perceptron avec R – Package « nnet »

Notre étude est rendue facile par le très faible nombre de descripteurs (p = 2) qui permet de projeter les observations dans le plan et, ainsi, d'identifier le bon paramétrage. En pratique, cette démarche n'est pas tenable, surtout **lorsque le nombre de descripteurs augmente (p > 2). Il nous définir une stratégie générique qui permet d'identifier automatiquement le bon nombre de neurones sur un jeu de données**. R est un outil privilégié pour cela. Tout simplement parce qu'il est très facile de programmer des actions complexes avec quelques lignes de codes bien senties.

Dans cette section, dans un premier temps, nous reproduisons l'étude ci-dessus (sur « **data2** ») pour décrire la mise en œuvre du perceptron multicouche avec le package **nnet**. Puis, dans un second temps, nous mettons en place un dispositif pour spécifier le nombre adéquat de neurones de la couche cachée pour le fichier « **data4** » où le concept à apprendre est plus complexe c.-à-d. il faut plus de droites séparatrices, mais combien ?

5.1 La procédure nnet du package éponyme

5.1.1 Importation et définition des échantillons

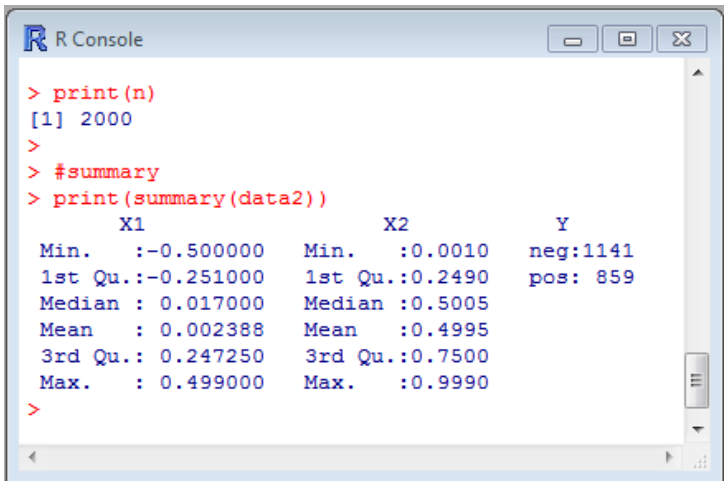
Nous avons transformé la feuille Excel « **data2** » en un fichier texte avec séparateur tabulation. Nous l'importons dans R avec la commande **read.table()**. Nous affichons les caractéristiques des variables avec la commande **summary()**.

```
#loading the data file "data2"
data2 <- read.table(file="artificial2d_data2.txt",sep="\t",dec=".",header=T)

#number of instances
n <- nrow(data2)
print(n)

#summary
print(summary(data2))
```

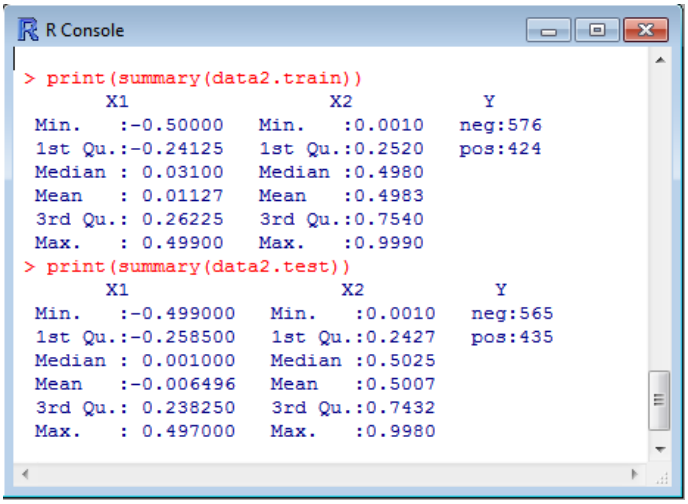
Nous obtenons :



Nous partitionnons aléatoirement les données en deux échantillons de taille égale : la première, dite d'apprentissage, servira à la construction du réseau ; la seconde, dite de test, sera utilisée pour évaluer les performances en généralisation.

```
#splitting in train and test samples
set.seed(5)
index <- sample(n,1000)
data2.train <- data2[index,]
data2.test <- data2[-index,]
print(summary(data2.train))
print(summary(data2.test))
```

La commande **set.seed()** permet de fixer la valeur d'initialisation du générateur de nombres aléatoires. Ainsi, lecteur devrait obtenir des résultats identiques à ceux décrits dans ce tutoriel.



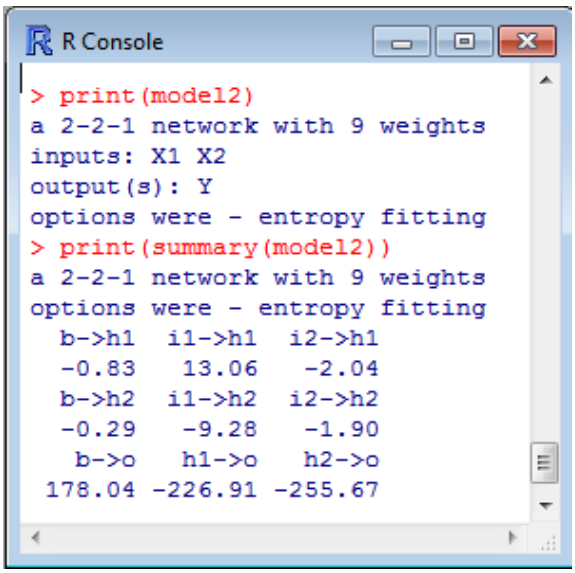
Nous constatons que les caractéristiques des deux échantillons sont très proches. C'est heureux, la subdivision ayant été réalisée de manière aléatoire. Il faudrait s'inquiéter dans le cas contraire.

5.1.2 Construction du modèle

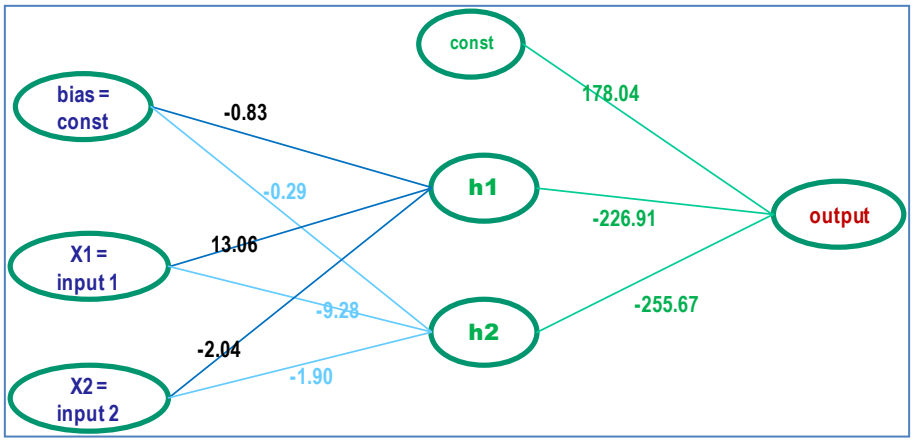
Nous utilisons le package « nnet » dans ce tutoriel. Il présente l'avantage d'être particulièrement simple à utiliser. Ses sorties sont en revanche très sobres, voire laconiques. Nous pouvons quand même obtenir les poids synaptiques du perceptron.

```
#loading the nnet package
library(nnet)
#multilayer perceptron, 2 neurons into the hidden layer
set.seed(10)
model2 <- nnet(Y ~ .,skip=FALSE,size=2,data=data2.train,maxit=300,trace=F)
print(model2)
print(summary(model2))
```

La commande **nnet()** construit le réseau à partir de l'échantillon d'apprentissage. Nous fixons le nombre d'itérations maximum à 300 (maxit = 300) pour nous assurer de la stabilité des résultats. La valeur par défaut (maxit = 100) n'est apparemment pas suffisante pour nos données. L'option « skip = FALSE » indique la présence d'une couche cachée, « size = 2 » définit le nombre de neurones.



La variable cible étant binaire, nous avions constaté que les deux sorties étaient strictement complémentaires avec Tanagra. De fait, R n'en décrit qu'une seule. Nous obtenons le réseau suivant :



Les coefficients semblent très différents de ceux de Tanagra. Mais lorsque l'on passe aux équations explicites, nous obtenons bien (quasiment, il y a une part d'aléatoire dans l 'apprentissage des poids du réseau) les mêmes frontières.

5.1.3 Evaluation

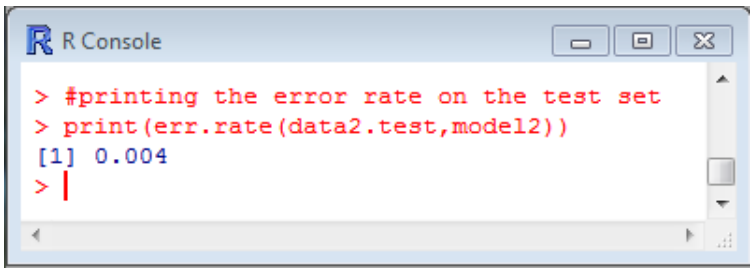
Nous élaborons une fonction pour l'évaluation. Elle prend en entrée les données à utiliser et le modèle. Ce dernier se charge de produire la prédiction à l'aide de la commande **predict()**. Nous construisons la matrice de confusion à partir de la confrontation entre la cible observée et la prédiction. Nous en déduisons le taux d'erreur (taux de mauvais classement).

```
#evaluating the model
err.rate <- function(test.data,model){
 #prediction on the test set
 pred <- predict(model,newdata=test.data,type="class")
 #confusion matrix
 mc <- table(test.data$Y,pred)
 #error rate
 error <- 1-sum(diag(mc))/sum(mc)
 #output
 return(error)
}
```

Appliquée sur l'échantillon test (1000 observations) et le réseau construit dans la section précédente,

```
#printing the error rate on the test set
print(err.rate(data2.test,model2))
```

Nous obtenons un taux d'erreur de 0.004.



5.2 Algorithme pour le paramétrage automatique du réseau

Notre objectif dans cette section est de mettre en place une procédure très simple de détection du nombre « optimal » de neurones de la couche cachée. Elle doit être générique c.-à-d. s'appliquer quel que soit le nombre de variables prédictives. Nous l'utiliserons alors pour apprendre la fonction – le concept - associant (X1, X2) à Y pour les données « **data4** ».

5.2.1 Algorithme de paramétrage automatique

La démarche est très similaire à la stratégie « wrapper » utilisée pour la sélection de variables en apprentissage supervisé[8]. Les données sont subdivisées en 2 échantillons : apprentissage et validation. Nous construisons différentes hypothèses sur l'échantillon d'apprentissage, dans notre cas il s'agit de nombre de neurones différents dans la couche cachée (1, ce qui revient à un perceptron simple ; puis 2, puis 3, etc.), que nous évaluons sur l'échantillon test. La solution correspond à la configuration la plus simple, c.-à-d. avec le plus petit nombre de neurones, permettant de minimiser le taux d'erreur en généralisation.

Le programme tient en une boucle passant en revue différentes valeurs de « k », nombre de neurones dans la couche cachée. Nous avons fixé *ex ante* le nombre maximal de neurones à tester (K = 20). C'était le plus simple à implémenter. Mais nous pouvons également imaginer des stratagèmes plus élaborés (ex. s'arrêter dès que le taux d'erreur ne décroît plus).

```
#detecting the right number of neurons in the hidden layer
K <- 20
res <- numeric(K)
for (k in 1:K){
 set.seed(10)
 model <- nnet(Y ~ .,skip=FALSE,size=k,data=data4.train,maxit=300,trace=F)
 error <- err.rate(data4.test,model)
 res[i] <- error
}
```

Le nom de la variable cible doit être Y dans ce petit programme. En revanche, il n'y a aucune limitation concernant le nombre et le nom des prédicteurs.

5.2.2 Application sur les données « data4 »

Pour vérifier l'efficacité du programme, nous l'appliquons sur les données « datat4 » dont nous ne sommes pas censés connaître le nombre adéquat de neurones (voir Section 2). Nous montrons ici le processus complet : l'importation des données, leur subdivision en apprentissage-validation, la mise en œuvre de la détection, l'affichage des résultats (nombre de neurones vs. taux d'erreur en validation) dans un graphique.

```
#loading the data file "data4"
data4 <- read.table(file="artificial2d_data4.txt",sep="\t",dec=".",header=T)
```

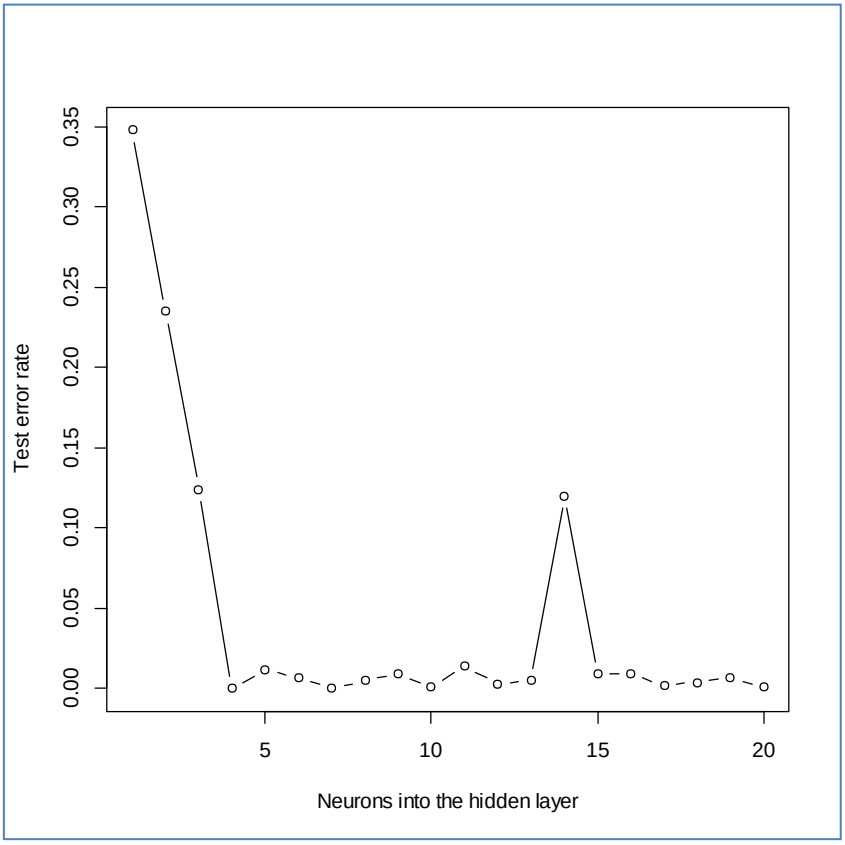
[8] Décrite dans http://tutoriels-data-mining.blogspot.fr/2009/05/strategie-wrapper-pour-la-selection-de.html pour **Sipina** et **R** ; et http://tutoriels-data-mining.blogspot.fr/2010/01/wrapper-pour-la-selection-de-variables.html pour **Knime**, **Weka** et **RapidMiner**.

```
#splitting in train and test samples
data4.train <- data4[index,]
data4.test <- data4[-index,]

#detecting the right number of neurons in the hidden layer
K <- 20
res <- numeric(K)
for (k in 1:K){
 set.seed(10)
 model <- nnet(Y ~ .,skip=FALSE,size=k,data=data4.train,maxit=300,trace=F)
 #print(model)
 error <- err.rate(data4.test,model)
 res[k] <- error
}

plot(1:K,res,type="b",xlab="Neurons into the hidden layer",ylab="Test error rate")
```

Nous obtenons une courbe particulièrement édifiante. A partir de « **k = 4** » neurones dans la couche intermédiaire, nous reproduisons parfaitement le concept.



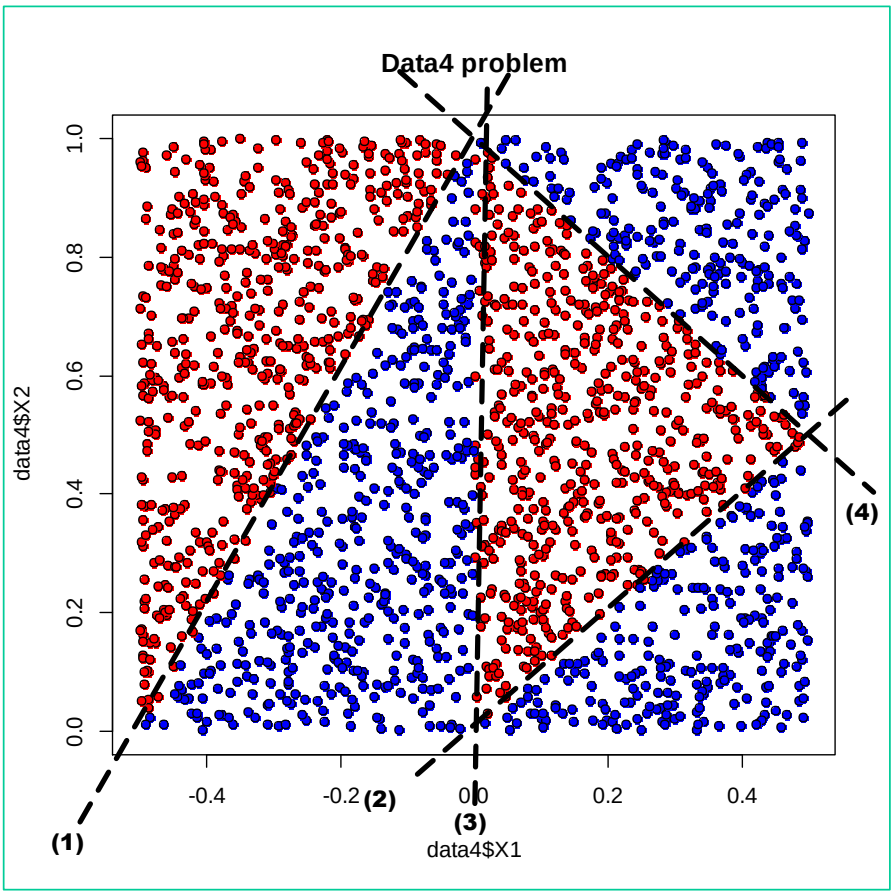
On serait tenté de parler de surapprentissage pour k = 14. Il faut surtout y voir un artefact lié aux fluctuations d'échantillonnage. Il serait plus approprié de passer par la validation croisée pour obtenir des résultats plus stables, mieux lissées. Mais, je ne souhaitais pas alourdir l'exposé au risque de semer le lecteur en cours de route dans le contexte de ce tutoriel.

5.2.3 Vérification Représentation graphique

Est-ce que vraiment k = 4 neurones est la bonne solution ici ? Pour le savoir, et puisque que nous n'avons que 2 descripteurs, nous projetons les observations étiquetées dans le plan.

```
#graphical representation of the instances
plot(data4$X1,data4$X2, pch = 21, bg = c("blue","red")[unclass(data4$Y)], main="Data4 problem")
```

Oui, il faut clairement une combinaison de 4 droites séparatrices pour isoler les « pos » des « neg ».



5.3 Sélection de variables et paramétrage du réseau

Nous nous sommes placés dans une configuration très favorable dans ce tutoriel. Nous avions un nombre réduit de variables prédictives et nous savions à l'avance qu'elles étaient toutes pertinentes. Dans les études réelles, la situation est un peu plus corsée. Il faut à la fois détecter les variables adéquates et identifier la bonne architecture du réseau.

A priori, la généralisation de notre approche à ce type de situation ne pose pas de problème. Il suffit de mettre en place une double boucle imbriquée permettant de combiner la recherche des bonnes variables prédictives et du nombre de neurones de la couche intermédiaire. Attention cependant au surapprentissage. En multipliant les hypothèses à tester, nous augmentons le risque de trouver des solutions faussement optimales. De plus, l'échantillon de test tel que nous l'avons définie dans notre démarche devient partie prenante de l'apprentissage puisqu'il sert à discerner la meilleure solution parmi les très nombreuses combinaisons « variables – architecture » qu'on lui a soumises.

De fait, il faudrait plutôt prévoir 3 échantillons dans ce cas : le premier, **apprentissage**, sert à l'apprentissage des modèles ; le second, **validation** (ou « sélection » même serait-on tenté de dire), sert à sélectionner le meilleur modèle ; le troisième, **test**, ne sert qu'à mesurer les performances en prédiction du modèle sélectionné. Ce dernier, n'intervenant qu'en dernier recours et n'ayant jamais participé au processus d'apprentissage/sélection, permet ainsi d'obtenir une estimation non biaisée des performances en prédiction du modèle sélectionné.

6 Conclusion

Le perceptron bénéficie d'une aura très favorable en apprentissage supervisé, à juste titre au regard de ses performances. Même si ces dernières années, si je réfère aux publications dans les revues et les conférences spécialisées, les méthodes ensemblistes et les SVM (support vector machine, de manière plus générique les méthodes à noyaux « kernel machines »[9]) semblent bénéficier d'une plus grande popularité au sein de la communauté du data mining (hé oui, les effets de mode sont partout, y compris dans la recherche). Il semble que ce soit moins vrai en ce qui concerne leur utilisation dans les études réelles[10].

Deux reproches reviennent souvent lorsque l'on parle des réseaux de neurones : c'est un modèle « boîte noire » dans le sens où l'on identifie mal l'influence respective des variables prédictives dans la prédiction /explication de la variable cible ; son paramétrage est difficile, définir le nombre de neurones dans la couche cachée n'est pas évident. Pour ce qui est de ce second inconvénient, nous montrons dans ce tutoriel qu'il est possible de mettre en place des solutions simples qui tiennent en quelques lignes de code sous R.

[9] http://www.kernel-machines.org/

[10] Classement 2011 des méthodes utilisées dans le data mining, sondage Kdnuggets : http://www.kdnuggets.com/polls/2011/algorithms-analytics-data-mining.html. Les trois approches se tiennent.