

1 Introduction

Determining the right number of neurons and layers in a multilayer perceptron.

At first glance, artificial neural networks seem mysterious. The references I read often spoke about biological metaphors, which were very interesting of course, but did not answer the simple questions I asked myself: Is the method applicable to predictive analysis? For which data configurations and kind of problems does it perform best? How to set parameter values - if they exist - associated with the method to make it more efficient on my data?

Over time, I have come to understand that the multilayer perceptron was one of the most popular neural network approach for supervised learning, and that it was very effective if we know to determine the number of neurons in the hidden layers. Two equally disastrous opposite choices are to be avoided: if we set too little neurons, the model is not very competitive, unable to discover the relationship between the target variable and the predictors, the bias of the error is high; if we set too much neurons, the model becomes too dependent to the learning sample, the variance of the error is too high. The difficulties come often that we do not really understand the influence of the numbers of the neurons in the hidden layer (if we have only one hidden layer for instance) on the characteristics of the classifier.

In this tutorial, we will try to explain the role of neurons in the hidden layer of the perceptron. We will use artificially generated data. We treat a two-dimensional problem in order to use graphical representations to describe concretely the behavior of the perceptron. We work with [Tanagra 1.4.48](#) in a first step. Then, we use R ([R 2.15.2](#)) to create a program to determine automatically the right number of neurons into the hidden layer. We use the [nnet](#)¹ package for R. The French version of this tutorial was written in April 2013, the versions of the tools are a little old, but the results, the comments and the conclusion remain up-to-date.

2 Dataset

The data file "[artificial2d.xls](#)" contains two datasets with $n = 2000$ instances. We have $p = 2$ descriptors X_1 (defined in $[-0.5, +0.5]$) and X_2 ($[0, 1]$). The target Y is binary {pos, neg}.

1. The first dataset "**data2**" corresponds to a problem defined as follows:

IF $(0.1 * X_2 > X_1^2)$ **THEN** $Y = \text{pos}$ **ELSE** $Y = \text{neg}$

¹ <https://cran.r-project.org/web/packages/nnet/index.html>

2. The underlying concept for “**data4**” is more complicated

If ($X1 < 0$)

THEN IF ($0.5 * X2 > 0.5 - |X1|$) THEN $Y = \text{pos}$ ELSE $Y = \text{neg}$

ELSE IF ($X2 > X1$ et $1 - X2 > X1$) THEN $Y = \text{pos}$ ELSE $Y = \text{neg}$

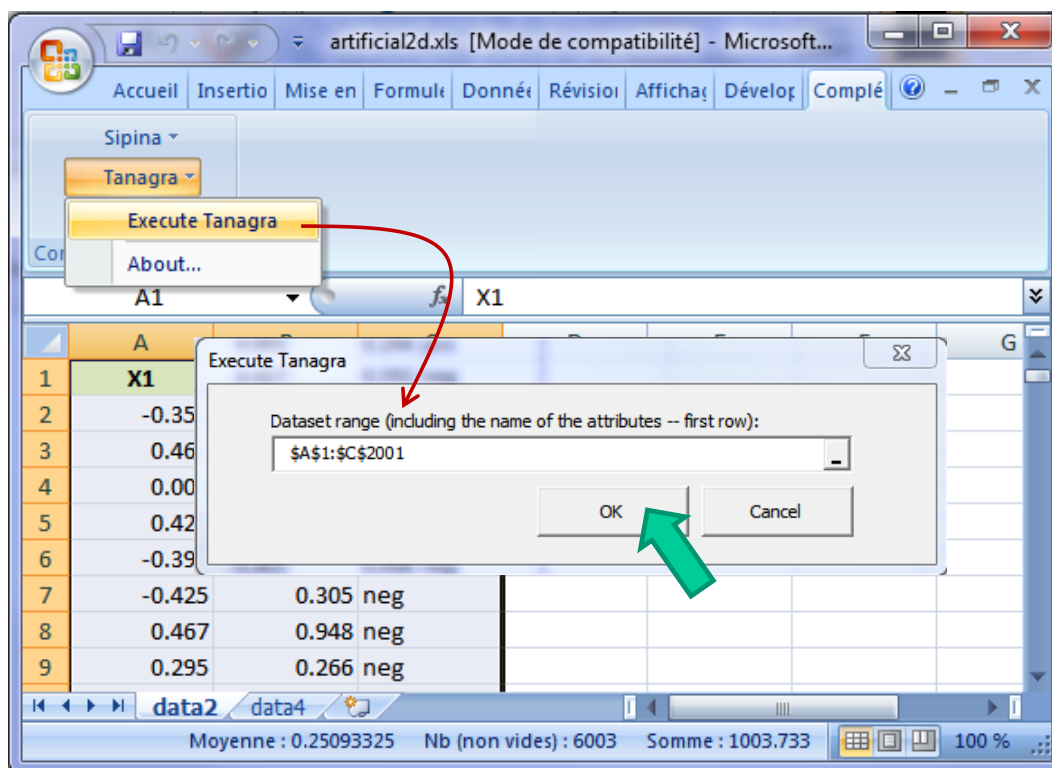
Of course, we do not have this information in real studies. The aim of this tutorial is above all to understand the behavior of the perceptron.

3 Single layer perceptron with Tanagra

A single layer perceptron has not a hidden layer. We have thus a linear classifier. We analysis its behavior on our datasets.

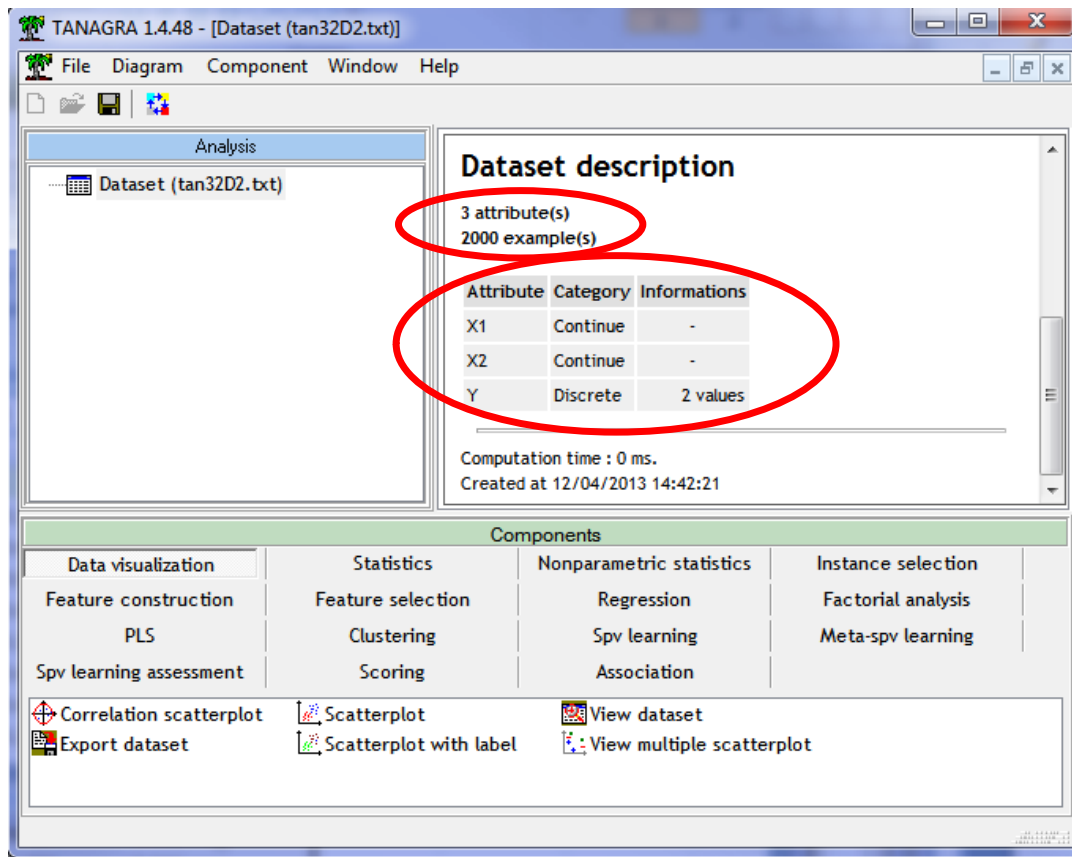
3.1 Data Importation

We open the **artificial2d.xls** data file into Excel. We select the data range in the first sheet **data2**, that we send to Tanagra using the Tanagra.xla add-in².



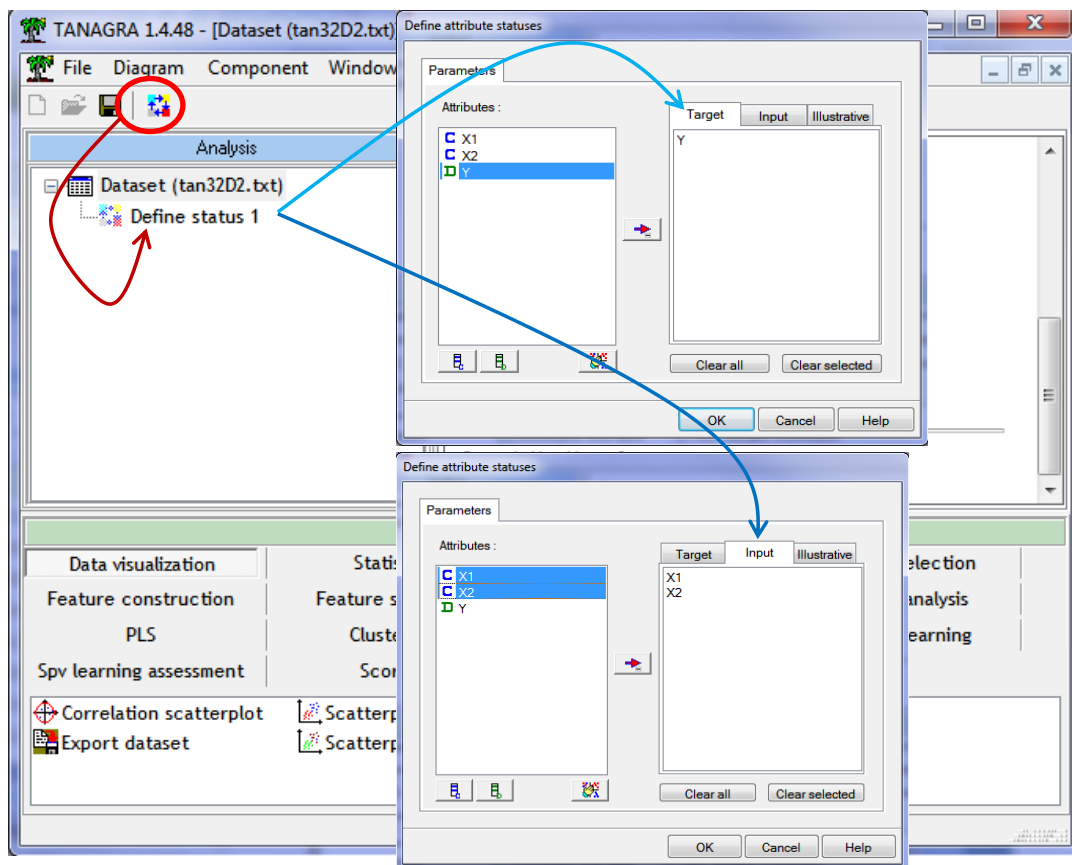
Tanagra is automatically launched. We have $n = 2000$ instances with 3 variables, of which $p = 2$ predictor variables.

² <http://data-mining-tutorials.blogspot.fr/2010/08/tanagra-add-in-for-office-2007-and.html>

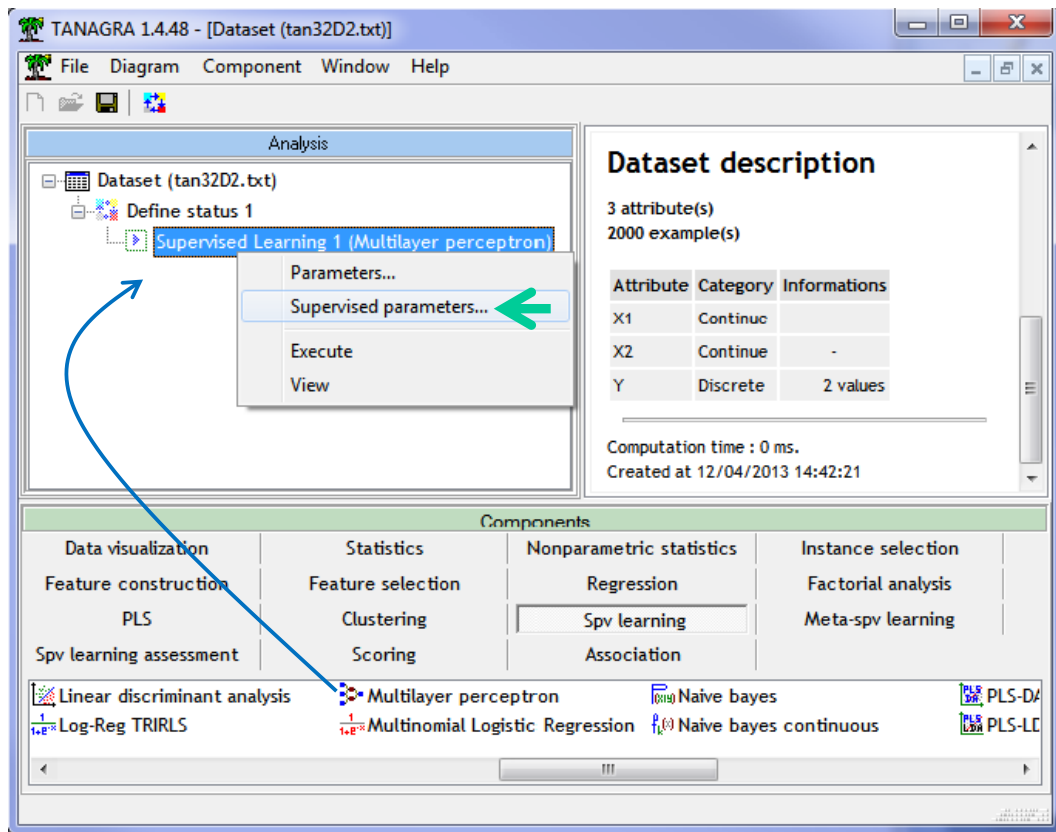


3.2 Single Layer Perceptron

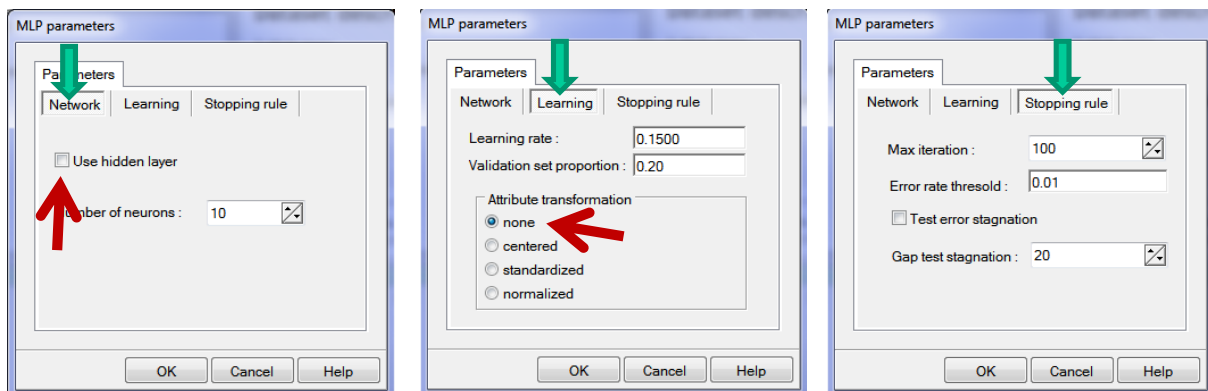
The DEFINE STATUS component allows to define the role of the variables.



We click on the shortcut in the toolbar. We set Y as TARGET, X1 and X2 as INPUT. Then, we insert the MULTILAYER PERCEPTRON (SPV LEARNING tab) component into the diagram.

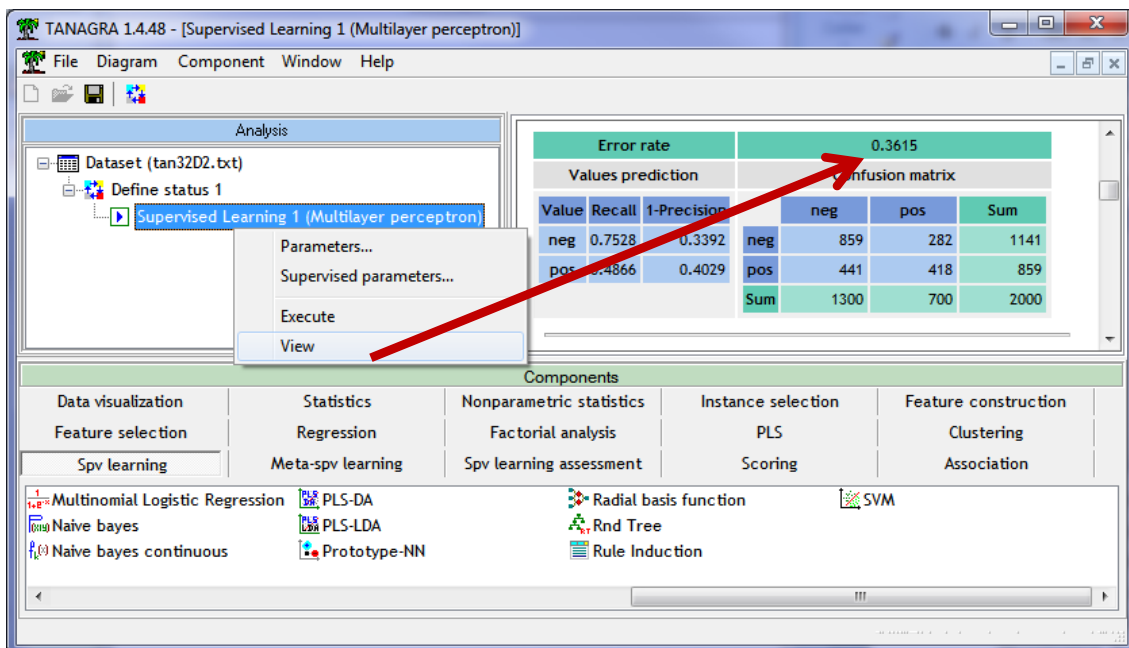


We set the settings by clicking on the **SUPERVISED PARAMETERS** menu.



We want to perform a single layer perceptron. For that, we unselect the **Use Hidden Layer** option into the NETWORK tab. Into the LEARNING tab, we set **Attribute Transformation = none**. We note that, by default, Tanagra subdivides the dataset into learning set (80%) and validation set (20%). This last one is used to monitor the decreasing of the error. Because it is not used for the calculations of the weights, it gives a good approximation of the generalization error rate. Into the STOPPING RULE tab, we observe that the learning process is stopped after 100 iterations or when the error rate is lower than 1%.

After we validate these settings, we launch the calculations by clicking on the VIEW menu.



The resubstitution error rate computed on the $n = 2000$ instances is 36.15%. Thereafter, we can see that it is decomposed into 36.19% for the 80% of the observations used to calculate the weights of the neural network, and 36% for the validation sample³.

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

Then, we can visualize the weights of the network. Since we have a binary problem, we have the same values for the two outputs, but in the opposite signs.

Weights		
From INPUT to OUTPUT layer		
-	neg	pos
X1	-0.08557568	0.08557568
X2	-2.10453600	2.10453600
bias	1.36340152	-1.36340152

³ Because the initial weights are settled randomly in the learning process, you may obtain a slightly different result.

3.3 Comments – Inadequacy of the linear classifier

Clearly, the classes are not linearly separable. Since we have only 2 variables, we can represent the dataset in a scatter chart. We add the SCATTERPLOT (DATA VISUALIZATION tab) into the root of the diagram. We set X1 in abscissa, X2 in ordinate, Y for defining the colors of the points.

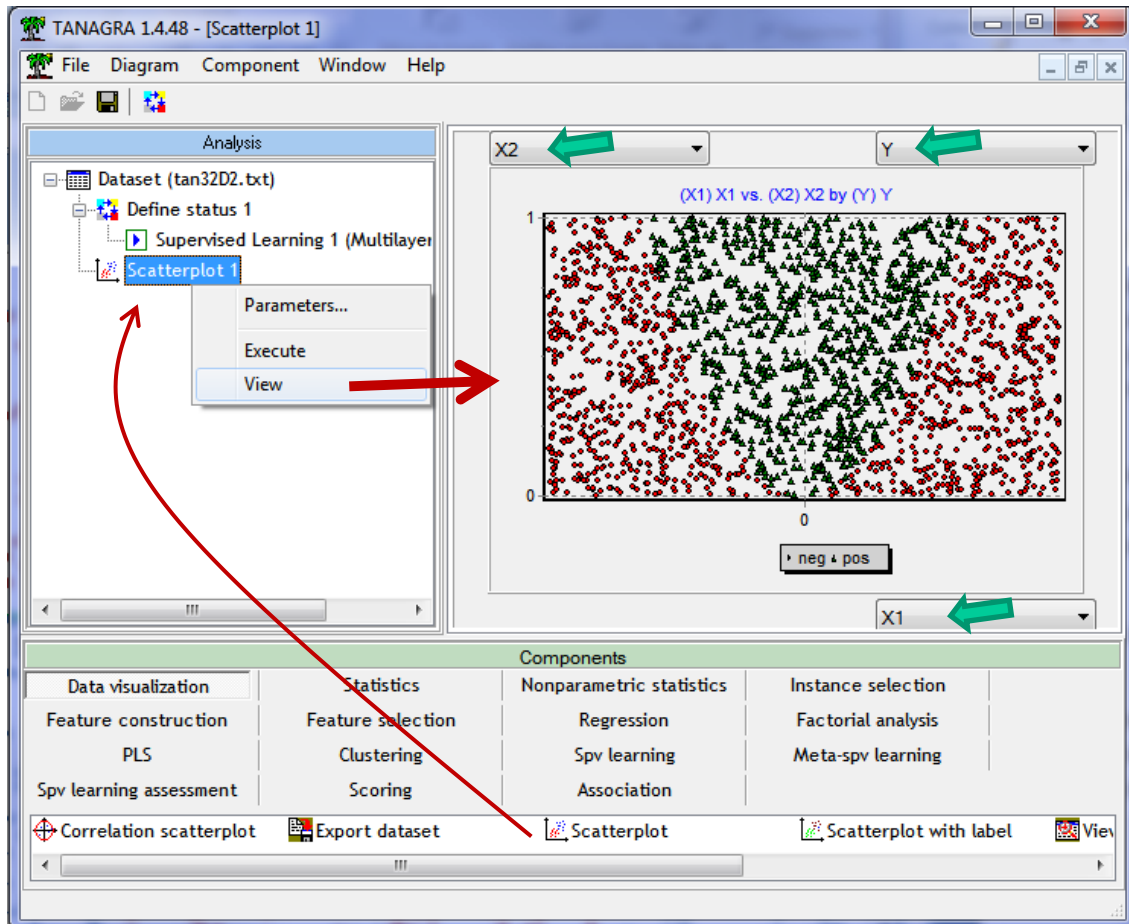


Figure 1 – Visualization of the data points into (X1, X2) – Colors defined by the observed labels

We should have started there. The frontier is rather of parabolic shape, it is not possible to produce a unique straight line to discriminate the classes.

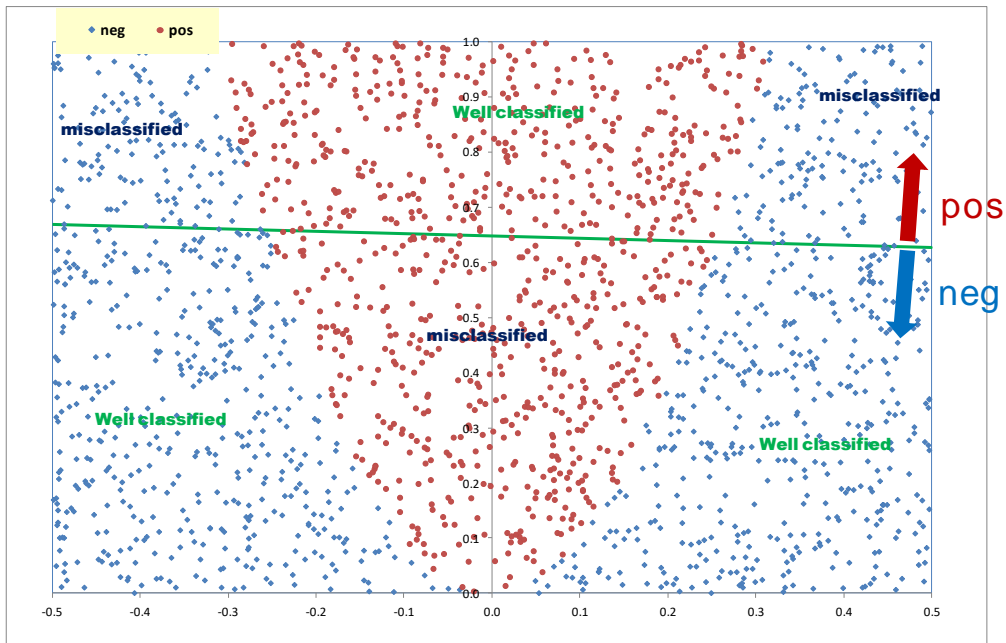
For our dataset, we frontier is defined by the following equation:

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

The explicit formula of the separation line is:

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

We can draw it in the representation space. We see that the solution is not appropriate. We distinguish between misclassified instances on both sides of the frontier. There are numerous.



It is possible to obtain a similar chart under Tanagra. We insert the SCATTERPLOT component after the neural network into the diagram. We place X1 in abscissa, X2 in ordinate, **predicted labels** for defining the colors of the points (PRED_SPVINSTANCE_1). We observe the frontier defined by the predicted labels.

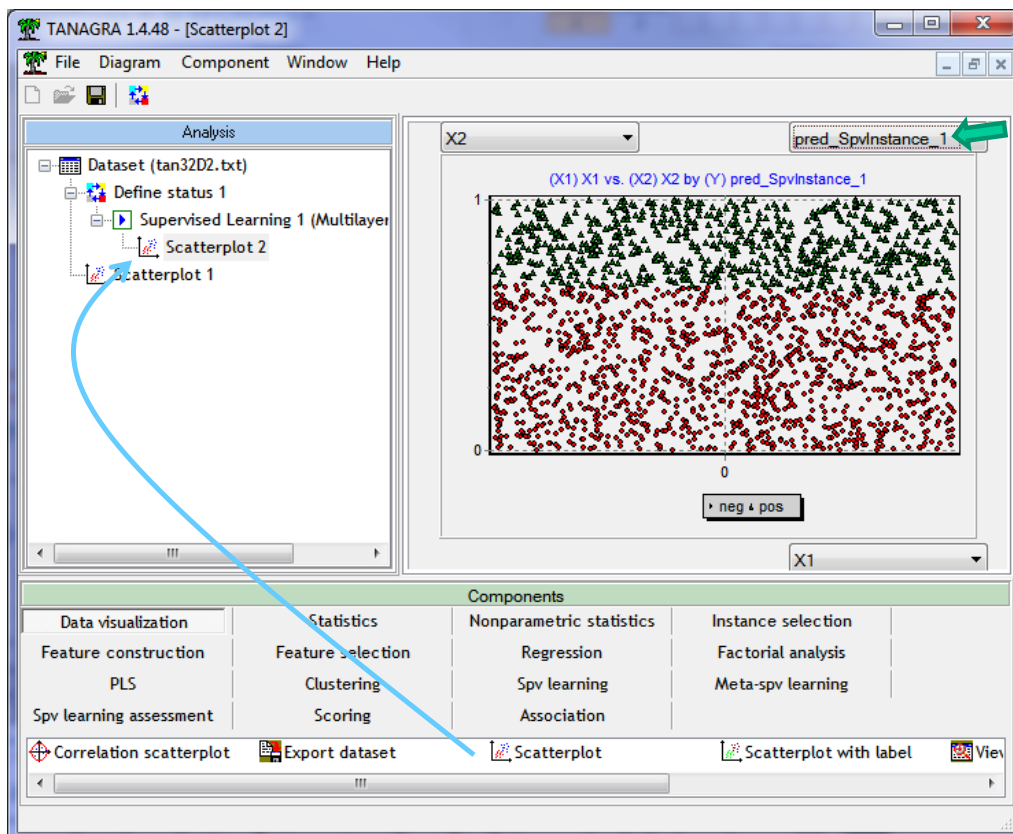


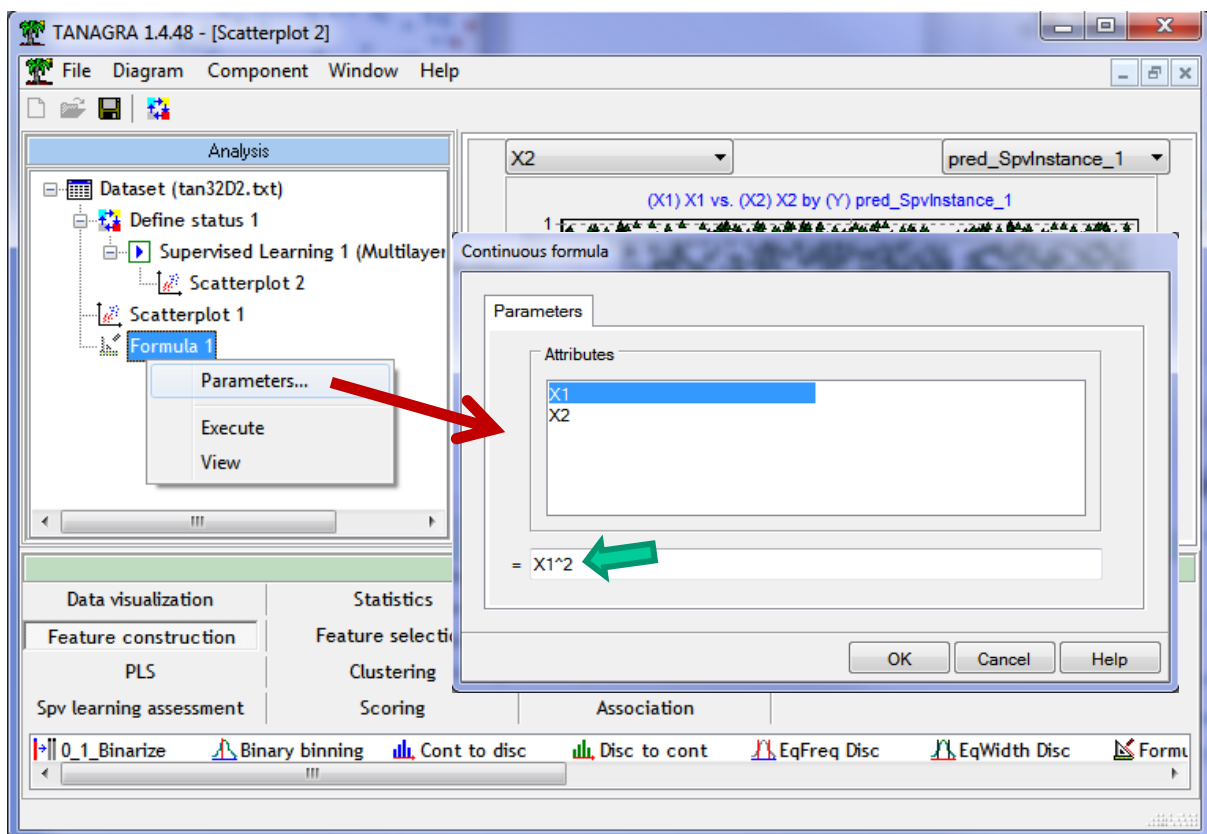
Figure 2 - Visualization of the data points into (X1, X2) - Colors defined by the predicted labels

Note: Any linear separator would be unable to produce an efficient classifier for our data: whether the logistic regression, the linear discriminant analysis, or the linear SVM (vector machine support). We need to change the representation bias i.e. the model's ability to represent more complex underlying concepts.

3.4 Variable transformation

One possible solution, which makes sense in view of the previous scatter chart, is to perform a variable transformation. That modifies the representation space. We choose to construct the variable $Z1 = X1^2$, we replace $X1$ by this one in the model.

Into Tanagra, we add the FORMULA component (FEATURE CONSTRUCTION tab) into the diagram. We set the parameters as follows:



The new variable $X1^2$ is now available in the subsequent part of the diagram.

To know if this transformation is relevant, we add the SCATTERPLOT tool after FORMULA 1. We set in abscissa the computed variable FORMULA_1 ($Z1$), in ordinate $X2$, the data points are colored according to the observed labels (Y). Now, we observe that the classes are linearly separable. We can draw a straight line to discriminate positive and negative instances.

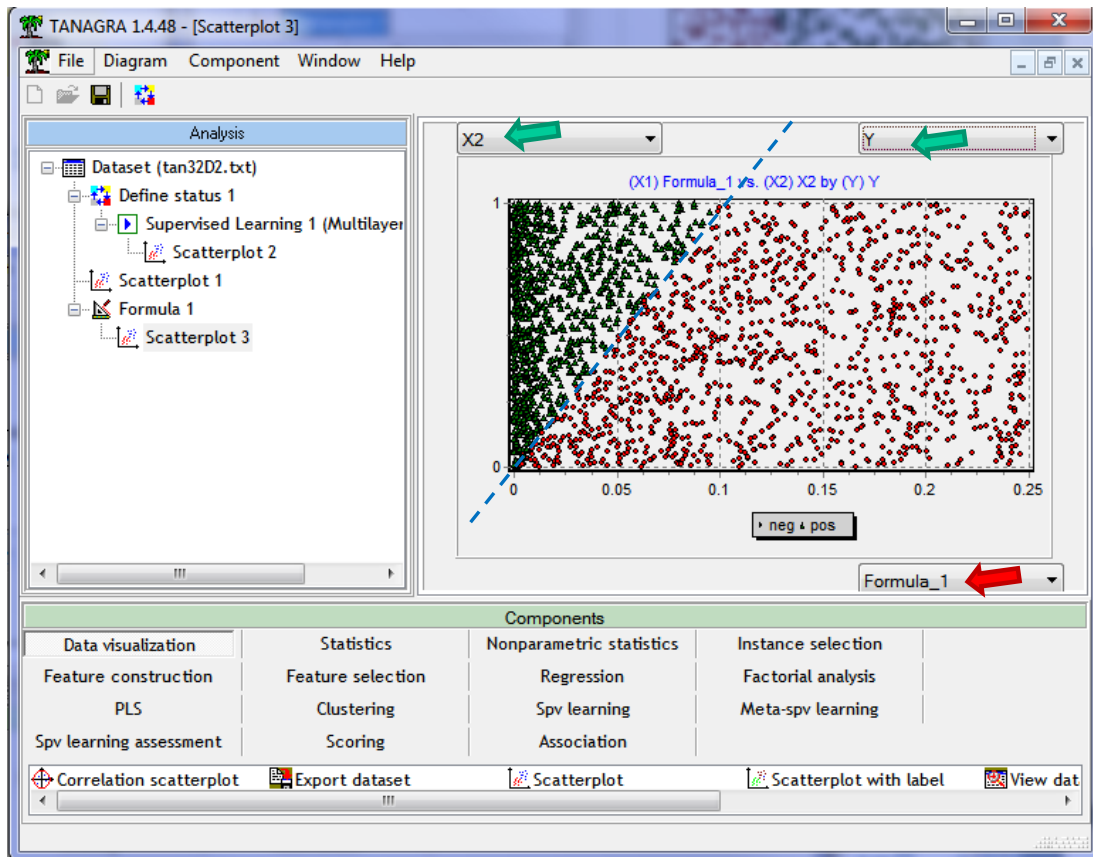
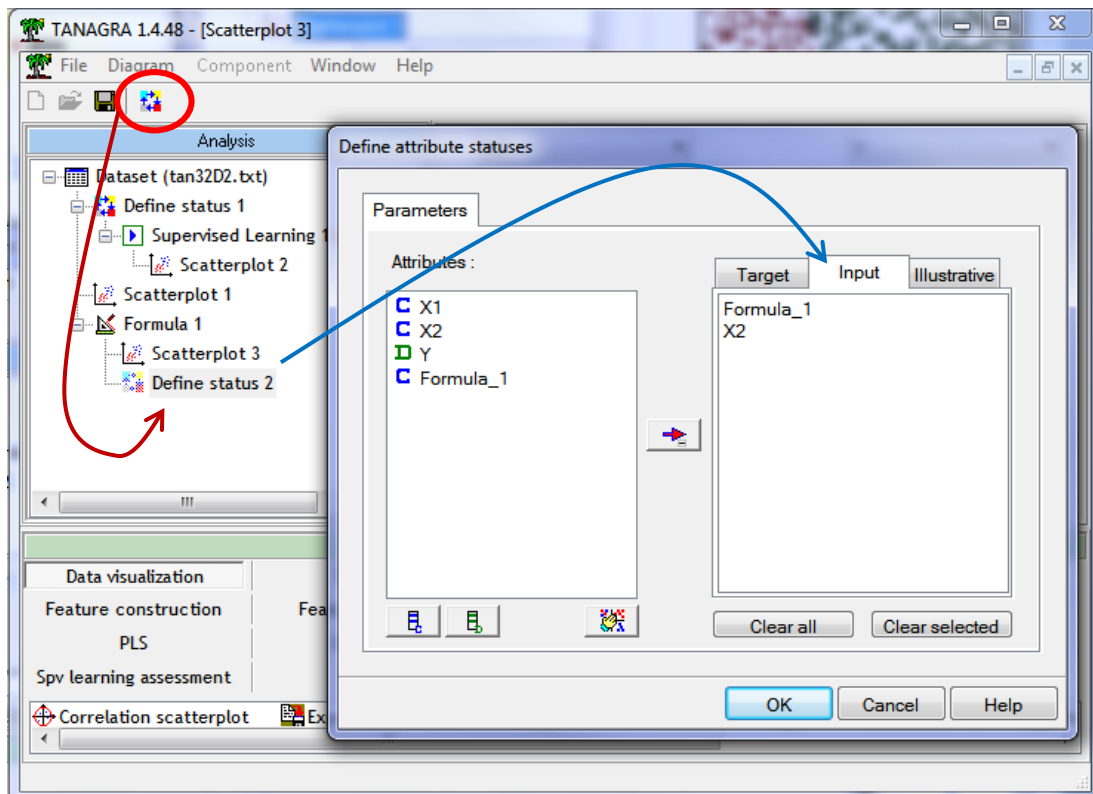
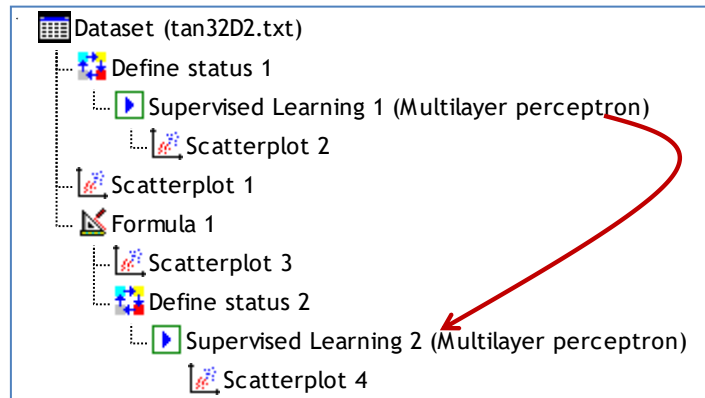


Figure 3 - Visualization of the data points into (Z1, X2) - Colors defined by the observed labels

Let us see if the single layer perceptron can benefit to this configuration. We insert the DEFINE STATUS component, we set Y as TARGET, FORMULA_1 (Z1) and X2 as INPUT.



We want to insert the **perceptron** again, making sure that it is used **the same settings as above**. The easiest way to do this is to move the component already inserted in the diagram with the mouse. It is duplicated with the same parameters⁴.



We click on the VIEW contextual menu to obtain the results.

Learning characteristics	
Epochs	100
Last train error rate	0.0150
Last validation error rate	0.0250
Last train mse	85.9253

The transformation carried out is fruitful with an error rate of 2.5% on the validation sample. The single layer perceptron is perfectly adapted in (Z_1, X_2) . Here are the estimated weights.

⁴ See <http://data-mining-tutorials.blogspot.fr/2009/06/copy-paste-feature-into-diagram.html>

Weights

From INPUT to OUTPUT layer

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

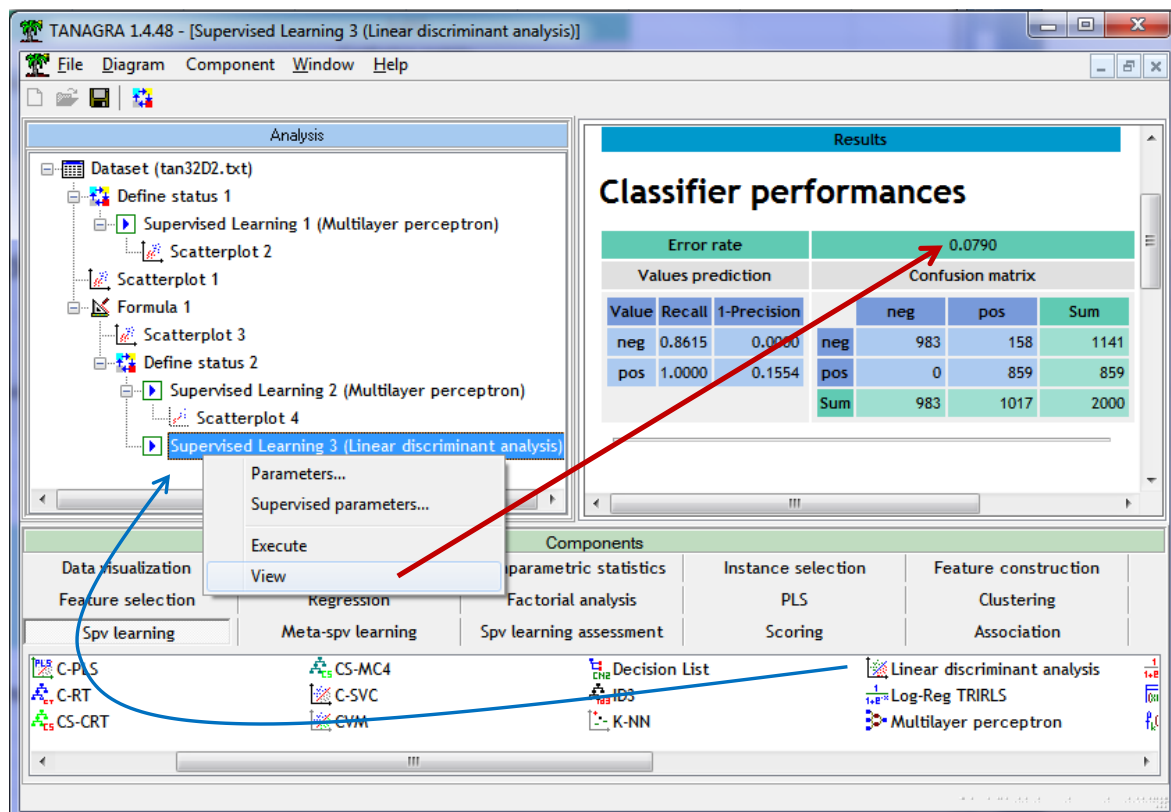
The explicit form of the equation of the frontier is (with $Z_1 = X_1^2$):

$$X_2 = 0.1082 * X_1^2 + 0.0003$$

The estimate is of very good quality. Indeed, it should be recalled that the frontier artificially used when generating the data was (see Section 2):

$$X_2 = 0.1 * X_1^2$$

Note: Just because a classifier knows how to represent a frontier does not mean that it can find it necessarily. A linear discriminant analysis for instance, which is also a linear classifier, assumes the gaussian conditional distribution. This assumption is not meet to our dataset (Figure 3). Thus, if we apply this approach (LINEAR DISCRIMINANT ANALYSIS component, SPV LEARNING tab) on the transformed representation space (Z_1, X_2), we obtain a disappointing resubstitution error rate (7.9%). This will not be better on the test set.



4 Multilayer perceptron (MLP) with Tanagra

4.1 Specifying the right number of neurons into the hidden layer

Finding the right variable transformation is not easy or impossible in real problems. The operation is very tedious as soon as the number of variables increases. The multilayer perceptron enables to improve the model's representational power by introducing a so-called "hidden" intermediate layer between the input and output layers (Note: we study the network with only one hidden layer in this tutorial). It is possible to learn any type of function by setting enough neurons in the hidden layer.

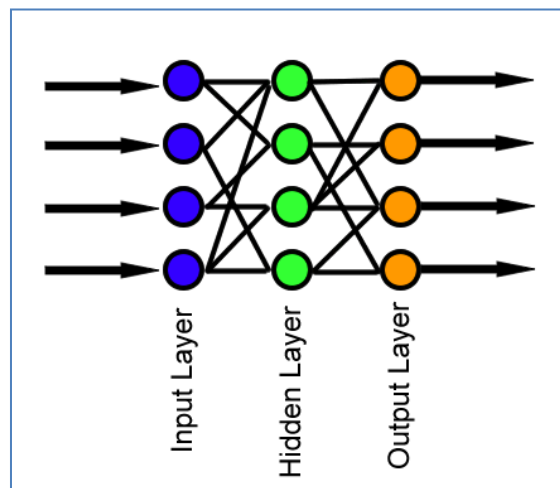


Figure 4 - Perceptron with a hidden layer - http://en.wikipedia.org/wiki/Feedforward_neural_network

If in doubt, we would be tempted to put many neurons in the hidden layer. However, by exaggerating the representational capacity of the structure, we risk the overfitting issue. The model corresponds too closely to the learning set and it is not able to generalize well.

Actually, setting the "right" number of the neurons is not easy in real situations. To do this, it would already be necessary to understand what are the ideas underlying a hidden layer in a neural network.

There are two ways to consider the hidden layer of a multilayer perceptron:

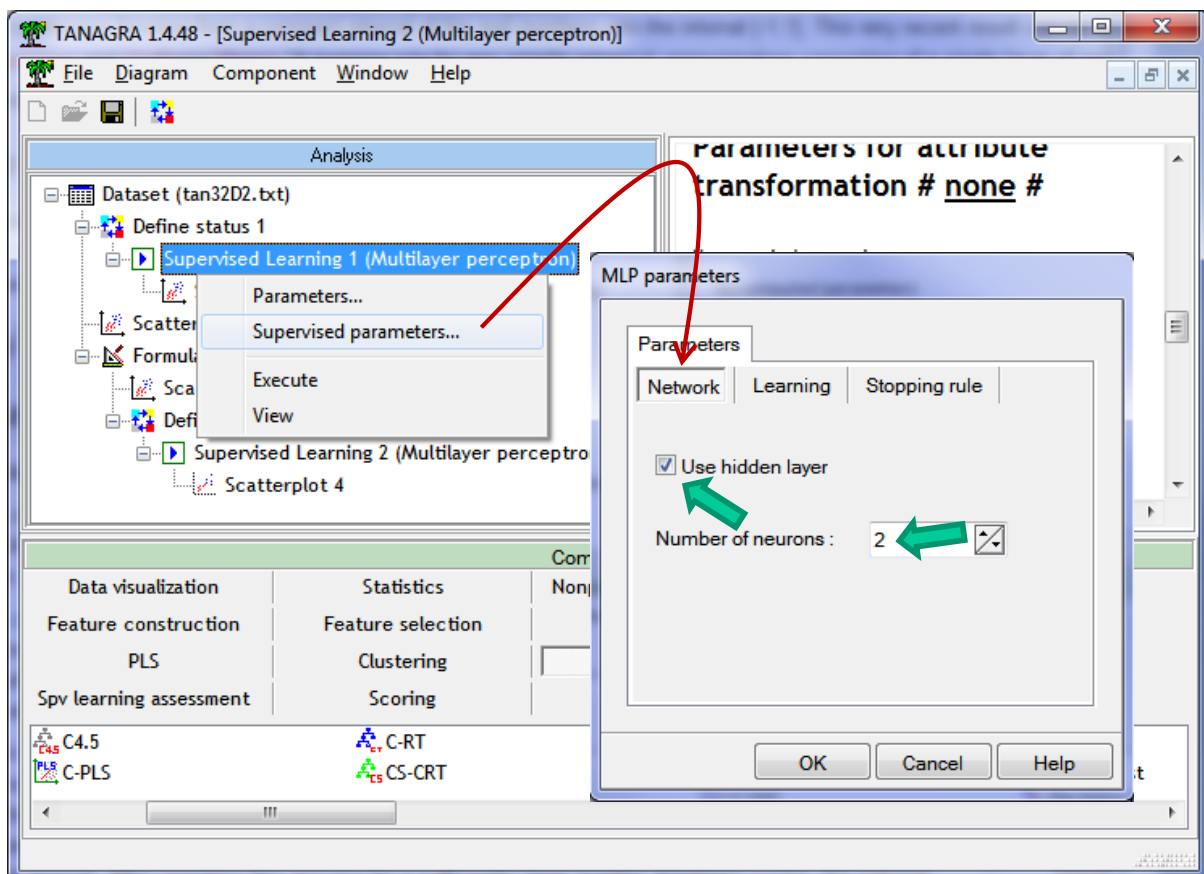
1. The hidden layer allows to produce an intermediate representation space where classes become linearly separable. The idea is well understood with the transformation of variables that we have done above. But in this case, this approach does not give us indication on the optimal number of hidden neurons to use.
2. Each neuron of the hidden layer defines a separator line. The connection between the hidden layer and the output allows to combine them. The model as a whole is non-linear.

This vision is more interesting in our situation. Indeed, we realize that the combination of two straight lines will allow us to approach as closely as possible the parable that separates the "positive" from the "negative" instances in the original representation space (Figure 1).

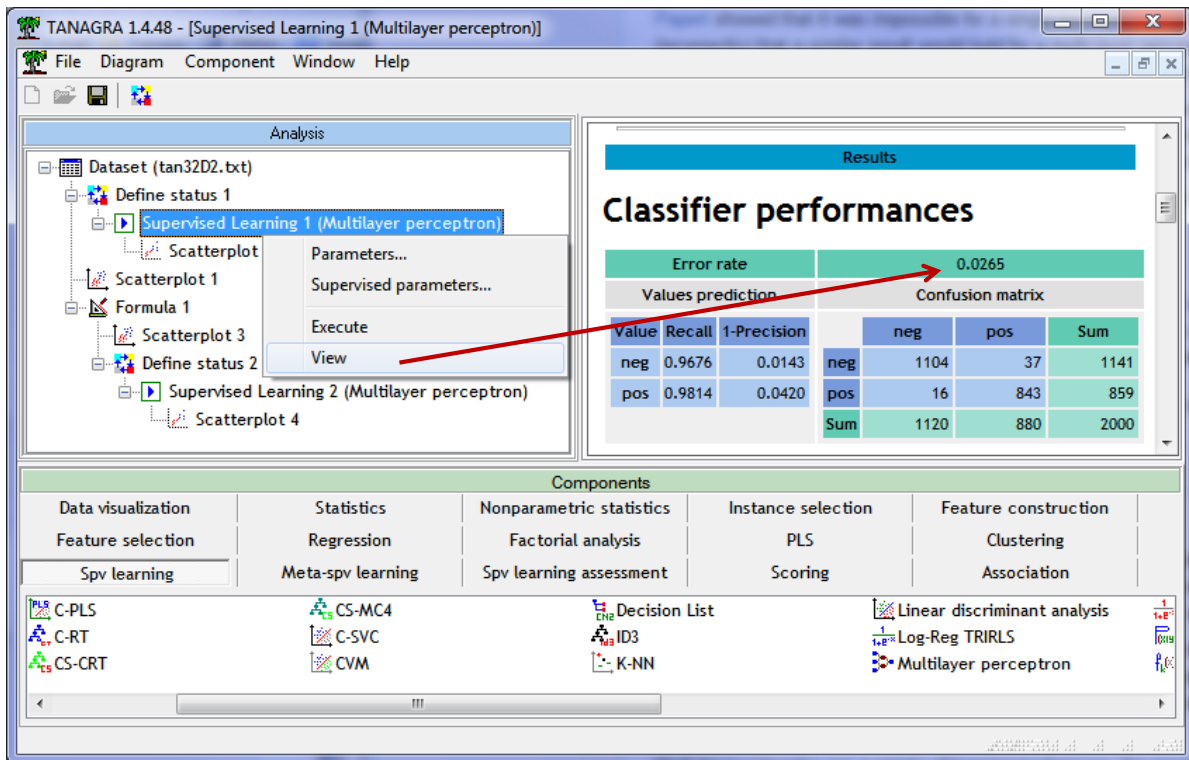
In short, incorporating 2 neurons in the hidden layer would be the most appropriate for our data.

4.2 Multilayer perceptron with Tanagra

We come back to the SUPERVISED LEARNING 1 (MULTILAYER PERCEPTRON) component into our diagram. We modify the settings by clicking on the SUPERVISED PAREMETERS menu. We select the hidden layer (Use hidden layer) and we specify 2 neurons.



We validate, and we click on the VIEW menu. The error rate is 2.65% now.



With 2.62% on the training set and 2.75% on the validation set.

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

We have the weights, connecting the input layer to the hidden layer on the one hand, and the hidden layer to the output layer on the other hand.

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 Frontiers into the original representation space

Let us consider the equations connecting the input layer to the hidden layer.

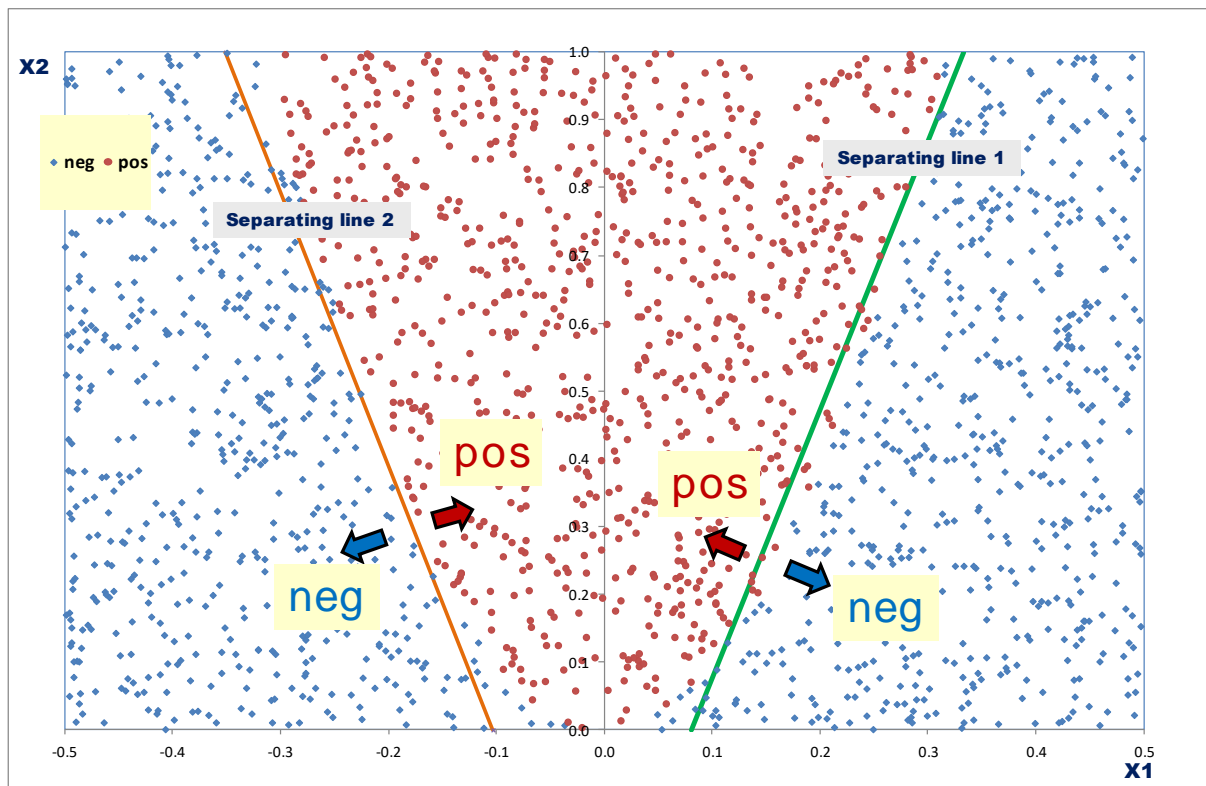
-	Neuron "1"	Neuron "2"
X1	19.9161	-19.2926
X2	-5.0306	-4.7801
bias	-1.6108	-1.9898

In the explicit form:

$$\text{Equation 1 (Frontier 1): } X2 = 3.9590 * X1 - 0.3202$$

$$\text{Equation 2 (Frontier 2): } X2 = -4.0360 * X1 - 0.4163$$

When we draw the corresponding straight lines into the original representation space (X1, X2), we observe the frontiers defined by the two neurons of the hidden layer to discriminate the classes. We can therefore identify misclassified individuals, which are on the wrong side of borders.



4.4 Frontier into the intermediate representation space

Another approach to understand the MLP is to plot the data points into the intermediate representation space (U1, U2) defined by the neurons of the hidden layer. By applying the activation function (sigmoid), the intermediate variables are defined as follows:

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

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

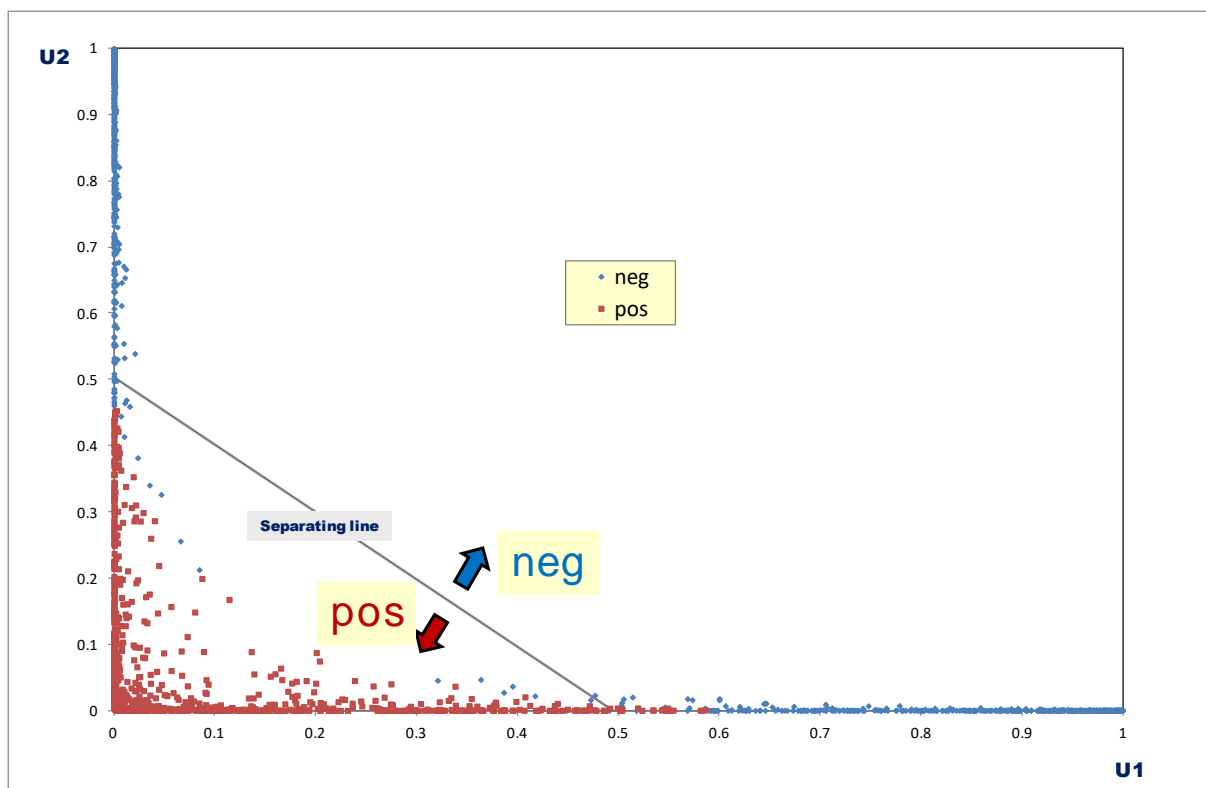
The separation line in this intermediate representation space is defined the weights between the hidden layer and the output layer.

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

In the explicit form:

$$\text{Frontier - Intermediate representation space: } U2 = -1.0169 * U1 + 0.5034$$

Let us see the data points in the representation space (U1, U2):



Here also, we can identify the misclassified instances.

Conclusion: It is all very inspiring. Personally, it took me a long time to really understand these mechanisms. MLP is rarely presented from this point of view in books. Yet, being able to identify the right number of neurons in the hidden layer is a major challenge that concerns everyone. It

will be very difficult to make the right choices if the consequences of adding neurons into the hidden layer are not well understood.

5 Perceptron with R – “nnet” package

Our study is easy to lead because we have only $p = 2$ descriptors. We can use graphical representations. When ($p > 2$), we need to define a generic strategy that allows us automatically identifies the right number of neurons for a dataset. We use R because we can program complex sequences of instructions. It is a privileged tool for that.

In this section, we first reproduce the above study (on **data2**) to describe the implementation of the multilayer perceptron with the **nnet** package. Then, in a second time, we set up an approach to determine the appropriate number of neurons of the hidden layer for the **data4** file where the concept to learn is more difficult.

5.1 The package nnet for R

5.1.1 Importation and subdivision of the dataset

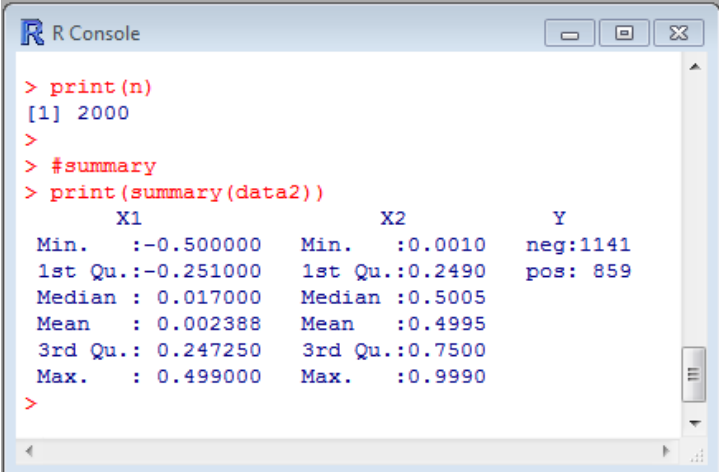
We have transformed the Excel sheet **data2** in a tabulation-separated text file. We import this data file into R with the **read.table()** command. We display the characteristics of the variables using the **summary()** command.

```
#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))
```

We obtain:

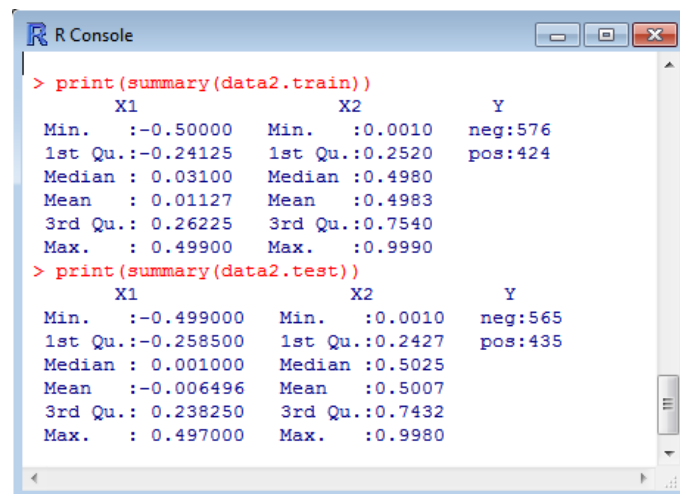


```
> print(n)
[1] 2000
>
> #summary
> print(summary(data2))
      X1          X2          Y
Min.   :-0.500000  Min.    :0.0010  neg:1141
1st Qu.: -0.251000  1st Qu.:0.2490  pos: 859
Median :  0.017000  Median :0.5005
Mean   :  0.002388  Mean    :0.4995
3rd Qu.:  0.247250  3rd Qu.:0.7500
Max.   :  0.499000  Max.    :0.9990
>
```

We randomly partition the data into two samples of equal size: the first, the learning sample, is used to build the network; the second, the test sample, is used to evaluate generalization performance.

```
#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))
```

set.seed() allows to generate the same sequence of random values. So, the reader will obtain the same results as those described in this tutorial.



```
> print(summary(data2.train))
      X1          X2          Y
Min.   :-0.50000   Min.   :0.0010   neg:576
1st Qu.: -0.24125   1st Qu.:0.2520   pos:424
Median :  0.03100   Median :0.4980
Mean    :  0.01127   Mean    :0.4983
3rd Qu.:  0.26225   3rd Qu.:0.7540
Max.    :  0.49900   Max.    :0.9990

> print(summary(data2.test))
      X1          X2          Y
Min.   :-0.499000   Min.   :0.0010   neg:565
1st Qu.: -0.258500   1st Qu.:0.2427   pos:435
Median :  0.001000   Median :0.5025
Mean    : -0.006496   Mean    :0.5007
3rd Qu.:  0.238250   3rd Qu.:0.7432
Max.    :  0.497000   Max.    :0.9980
```

The subdivision was done randomly. We note that the characteristics of the two samples are very similar.

5.1.2 Learning process

We use the [nnet](#) package in this tutorial. It is easy to use. But, its drawback is the brevity of its outputs. We can however get the weights of the 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))
```

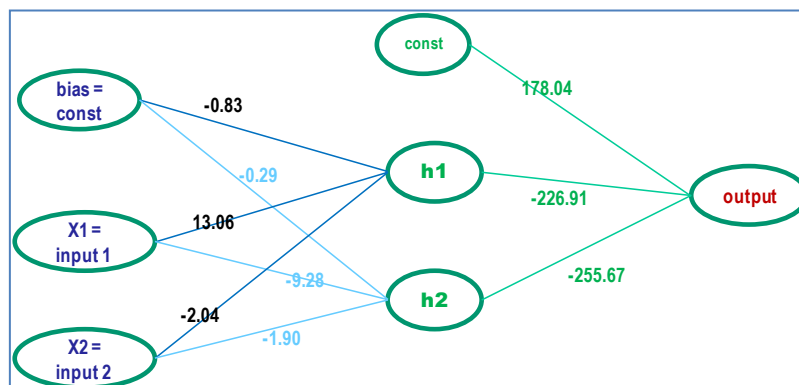
The **nnet()** command launches the learning process. We set the maximum number of iterations to 300 (`maxit = 300`). The option "`skip = FALSE`" indicates that we want to include a hidden layer in our network; "`size = 2`" specifies the number of neurons into the hidden layer.

```

R Console
> print(model2)
a 2-2-1 network with 9 weights
inputs: X1 X2
output(s): Y
options were - entropy fitting
> print(summary(model2))
a 2-2-1 network with 9 weights
options were - entropy fitting
b->h1  i1->h1  i2->h1
-0.83  13.06  -2.04
b->h2  i1->h2  i2->h2
-0.29  -9.28  -1.90
b->o   h1->o   h2->o
178.04 -226.91 -255.67

```

Because we have a binary target attribute, the weights of the two outputs neurons are strictly the same with the opposite signs. Therefore, **nnet()** shows only one of them. Here is the network:



The coefficients seem very different from those of Tanagra. But when we calculate the explicit equations, we get the same boundaries (almost, there is a part of randomness in the learning of the weights of the network).

5.1.3 Evaluation

We program a function for evaluation. It takes the data to be used and the model as input. The latter calculates the prediction using the `predict()` command. We construct the confusion matrix by comparing the observed values of the target attribute and the predicted values of the model. We then compute the error rate (misclassification rate).

```

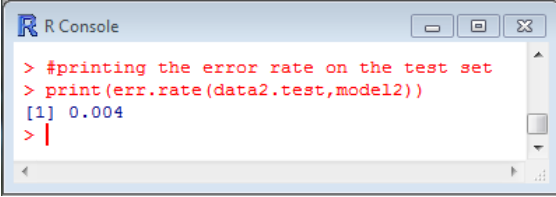
#evaluating the model
err.rate <- function(test.data,model){
  #prediction on the test set
  pred <- predict(model,newdata=test.data,type="class")
  #confusion matrix
  mc <- table(test.data$Y,pred)
  #error rate
  error <- 1-sum(diag(mc))/sum(mc)
  #output
  return(error)
}

```

Applied on the test set (1000 observations), the test error rate of the model...

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

... is **0.004**.



```
R Console
> #printing the error rate on the test set
> print(err.rate(data2.test,model2))
[1] 0.004
> |
```

5.2 Approach for determining automatically the right parameters

Our objective in this section is to program a very simple procedure for detecting the "optimal" number of neurons in the hidden layer. It must be generic i.e. can be applied regardless of the number of predictors. We will then use it to learn the function - the concept - associating (X1, X2) with Y for the "data4" dataset.

5.2.1 Program in R

The approach is very similar to the wrapper strategy used for the variable selection⁵. The dataset is partitioned into training sample and test sample. We try different hypotheses on the learning sample. In our case we modify the number of neurons in the hidden layer (1, which corresponds to a simple perceptron; then 2, then 3, etc.). We evaluate the models on the test sample. We select the solution corresponding to the most parsimonious model (the lowest number of neurons into the hidden layer) with the lowest error rate.

The program tries in a loop different values of "k", number of neurons in the hidden layer. We have set the maximum number of neurons to be tested (K = 20). It was the easiest to program. But we could also imagine more elaborate schemes (e. g. stopping as soon as the error rate no longer decreases).

```
#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
}
```

⁵ See <http://data-mining-tutorials.blogspot.fr/2010/03/wrapper-for-feature-selection.html> and <http://data-mining-tutorials.blogspot.fr/2010/04/wrapper-for-feature-selection.html>

The name of the target variable must be Y in this small program. But, on the other hand, there is no limitation on the number and name of predictors.

5.2.2 Application on the **data4** dataset

To check the effectiveness of the program, we apply it to **datat4** dataset for which we are not supposed to know the correct number of neurons (see Section 2). We show below the complete process: importing data, subdividing it into learning and validation, implementing detection of the number of neurons, displaying results (number of neurons vs. error rate in validation) in a chart.

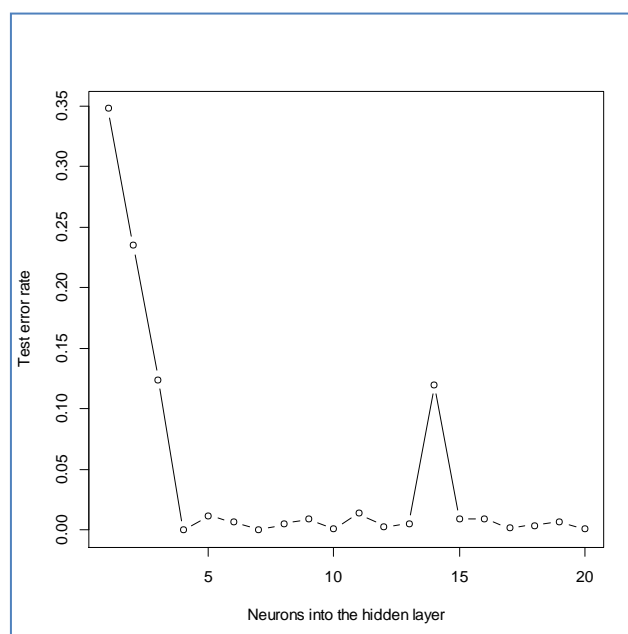
```
#loading the data file "data4"
data4 <- read.table(file="artificial2d_data4.txt", sep="\t", dec=".", header=T)

#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")
```

We obtain a particularly enlightening curve. From **k = 4** neurons into the hidden layer, we have a perfect model.



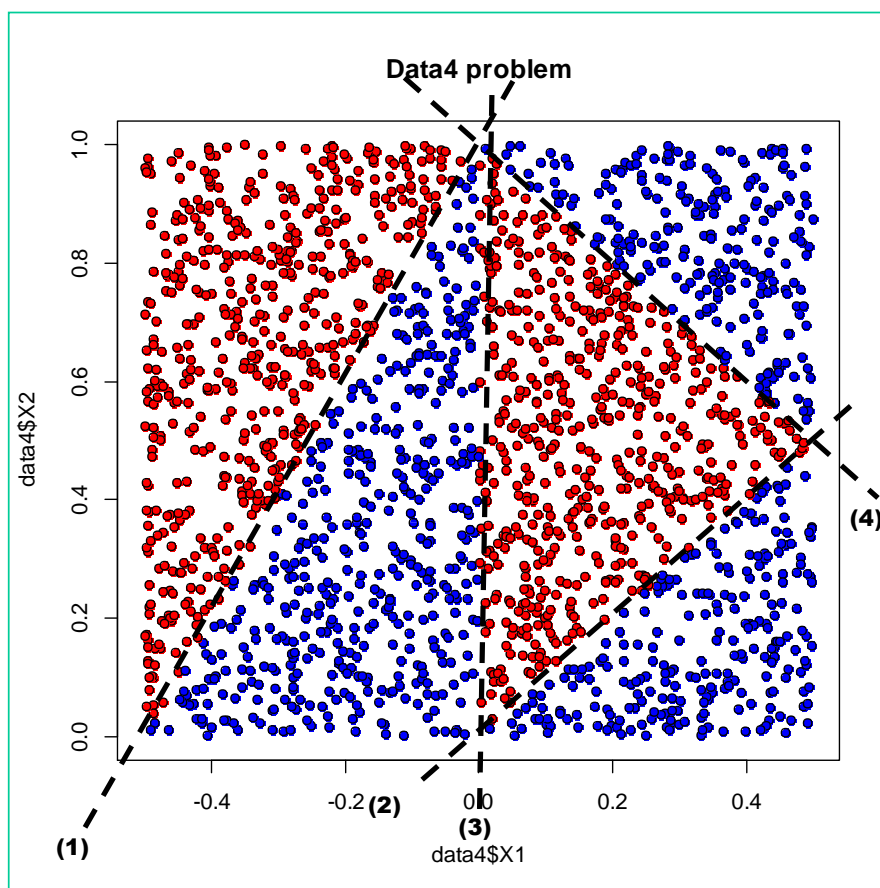
For $k = 14$, this is primarily an artifact caused by the sampling fluctuations. It would be more appropriate to use cross-validation to obtain more stable results. This is a possible improvement of our program.

5.2.3 Graphical representation of the concept

The solution “ $k = 4$ ” neurons is it really the right one here? To find out this, and since we only have 2 descriptors, we draw the labelled observations in a scatter graph.

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

Yes, we need a combination of 4 straight lines to discriminate the classes.



5.3 Variable selection and configuration of the network

We have placed ourselves in a very favorable situation in this tutorial. We had a small number of predictive variables and we knew in advance that they were all relevant. In real studies, the circumstances are a little more difficult. It is necessary to both detect the right variables and identify the right network architecture.

A priori, the generalization of our approach to this type of situation is a possible approach. We can set a double nested loop to combine the search for the right predictive variables and the number of neurons in the intermediate layer. But beware of overfitting. By multiplying the hypotheses to be tested, we increase the risk of finding false optimal solutions. In addition, the test sample as we defined it in our approach becomes part of the learning process since it serves to identify the best solution among the many "variable-architecture" combinations which are submitted. Last, the computation time becomes rapidly a serious bottleneck.

The search for the best combination of the variables and the number of neurons remains a difficult problem.

6 Conclusion

The multilayer perceptron is a well-known method, for a long time. After having been masked by ensemble methods and support vector machine (SVM) approaches in the publications during a moment, it has been revived with the success of the deep learning in the recent years.

Two issues are often stated when we talk about neural networks: it is a "black box" model in the sense that we cannot identify the influence of predictive variables in the prediction/explanation of the target variable; specifying the right values of the parameters is difficult, especially when we want to determine the number of neurons in the hidden layer. As for this second drawback, we show in this tutorial that it is possible to implement simple solutions that correspond to a few lines of instructions in R.