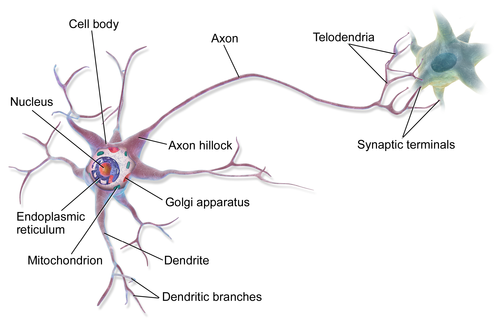
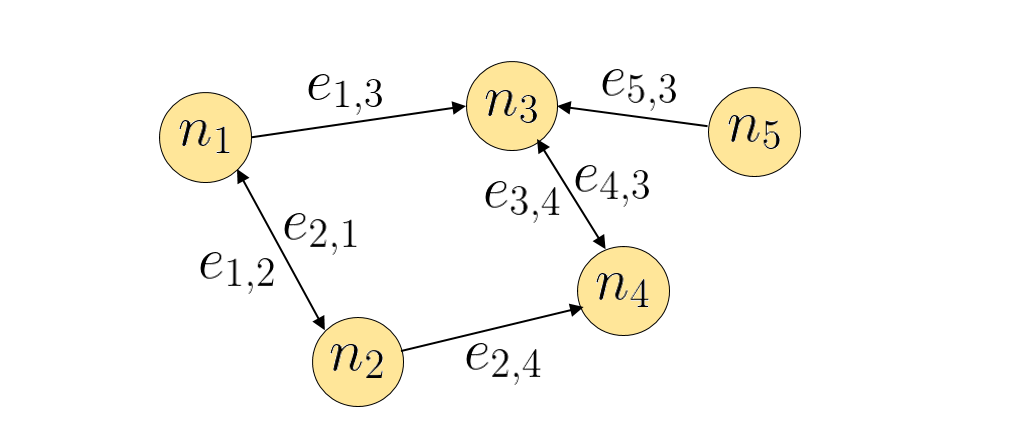
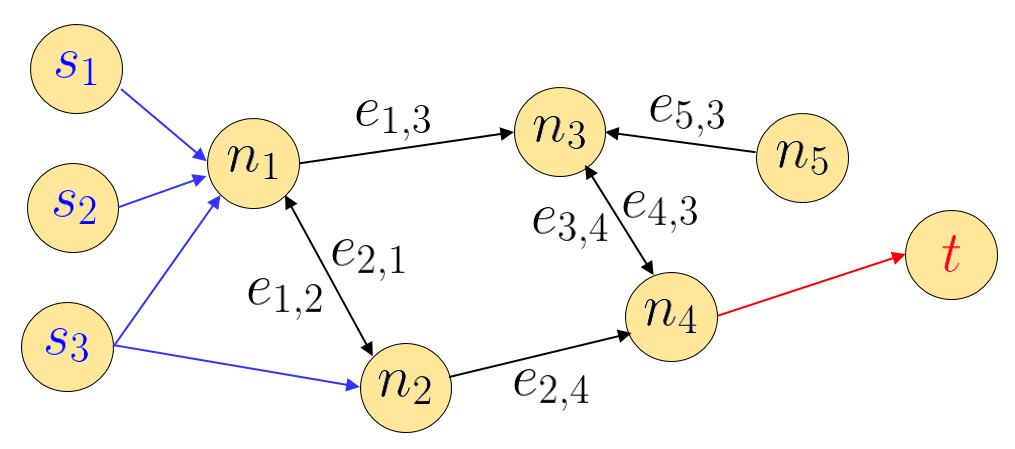
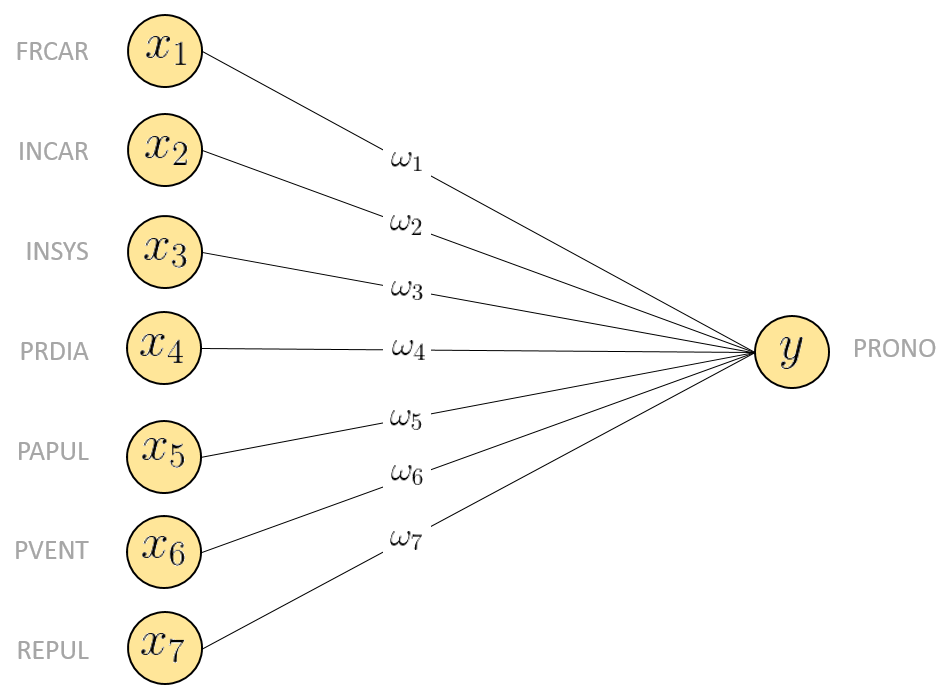
From now on, we will discuss technique that did not originate from those probabilistic models, even if they might still have a probabilistic interpretation. Somehow. Today, we will start with neural nets.

The goal is not to replicate well designed R functions, used for predictive modeling. It is simply to get a basic understanding of what’s going on.

**Networs, nodes and edges**

First of all, neurals nets are nets, or networks. I will skip the parallel with “neural” [stuff](https://en.wikipedia.org/wiki/Neuron) because it does not help me understanding what is happening (all apologies for my poor knowledge on biology, and cells)  
[](https://en.wikipedia.org/wiki/Neuron)  
So, it’s about some network. Networks have nodes, and edges (possibly connected) that connect nodes,  
  
or maybe, to more specific (at least it helped me understanding what’s going on), some sort of [flow network](https://en.wikipedia.org/wiki/Flow_network),  
  
In such a network, we usually have sources (here multiple) sources (here {s1,s2,s3}\color{red}\{s\_1,s\_2,s\_3\}{s1​,s2​,s3​}), on the left, on a sink (here {t}\{\color{blue}t\}{t}), on the right. To continue with this metaphorical introduction, information from the sources should reach the sink. An usually, sources are explanatory variables, {x1,⋯ ,xp}\{\mathbf{x}\_1,\cdots,\mathbf{x}\_p\}{x1​,⋯,xp​}, and the sink is our variable of interest y\mathbf{y}y. And we want to create a graph, from the sources to the sink. We will have directed edges, with only one (unique) direction, where we will put weights. It is not a flow, the parallel with flow will stop here. For instance, the most simple network will be the following one, with no layer (i.e no node between the source and the sink)  
  
The output here is a binary variable y∈{0,1}y\in\{0,1\}y∈{0,1} (it can also be y∈{−1,+1}y\in\{-1,+1\}y∈{−1,+1} but here, it’s not a big deal). In our network, our output will be y∈(0,1)y\in(0,1)y∈(0,1), because it is more easy to handly. For instance, consider y=f(y=f(y=f(something))), for some function fff taking values in (0,1)(0,1)(0,1). One can consider the [sigmoid](https://en.wikipedia.org/wiki/Sigmoid_function) functionf(x)=11+e−x=exex+1f(x)=\frac{1}{1+e^{-x}}=\frac{e^{x}}{e^{x}+1}f(x)=1+e−x1​=ex+1ex​which is actually the [logistic](https://en.wikipedia.org/wiki/Logistic_function) function (so we should not be surprised to have results somehow close the logistic regression…). This function fff is called the [activation function](https://en.wikipedia.org/wiki/Activation_function), and there are thousands of such functions. If y∈{−1,+1}y\in\{-1,+1\}y∈{−1,+1}, people consider the hyperbolic tangentf(x)=tanh⁡(x)=(ex−e−x)(ex+e−x)f(x)=\tanh(x)={\frac {(e^{x}-e^{-x})}{(e^{x}+e^{-x})}}f(x)=tanh(x)=(ex+e−x)(ex−e−x)​or the inverse tangent function  
f(x)=tan⁡−1(x)f(x)=\tan ^{-1}(x)f(x)=tan−1(x)And as input for such function, we consider a weighted sum of incoming nodes. So hereyi=f(∑j=1pωjxj,i)y\_i=f\left(\sum\_{j=1}^p\omega\_j x\_{j,i}\right)yi​=f(j=1∑p​ωj​xj,i​)We can also add a constant actuallyyi=f(ω0+∑j=1pωjxj,i)y\_i=f\left(\omega\_0+\sum\_{j=1}^p\omega\_j x\_{j,i}\right)yi​=f(ω0​+j=1∑p​ωj​xj,i​)So far, we are not far away from the logistic regression. Except that our starting point was a probabilistic model, in the sense that the later was interpreted as a probability (the probability that Y=1Y=1Y=1) and we wanted the model with the highest likelihood. But we’ll talk about selection of weights later on. First, let us construct our first (very simple) neural network. First, we have the sigmoid function

|  |
| --- |
| sigmoid = **function**(x) 1 / (1 + **exp**(-x)) |

The consider some weights. In our model with seven explanatory variables, with need 7 weights. Or 8 if we include the constant term. Let us consider ω=1\mathbf{\omega}=\mathbf{1}ω=1,

|  |
| --- |
| weights\_0 = **rep**(1,8)  X = **as.matrix**(**cbind**(1,myocarde[,1:7]))  y\_5\_1 = sigmoid(X %\*% weights\_0) |

that’s kind of stupid because all our predictions are 1, here. Let us try something else. Like ω=β^ols\mathbf{\omega}=\widehat{\mathbf{\beta}}^{ols}ω=β

​ols. It is optimized, somehow, but we needed something to visualize what’s going on

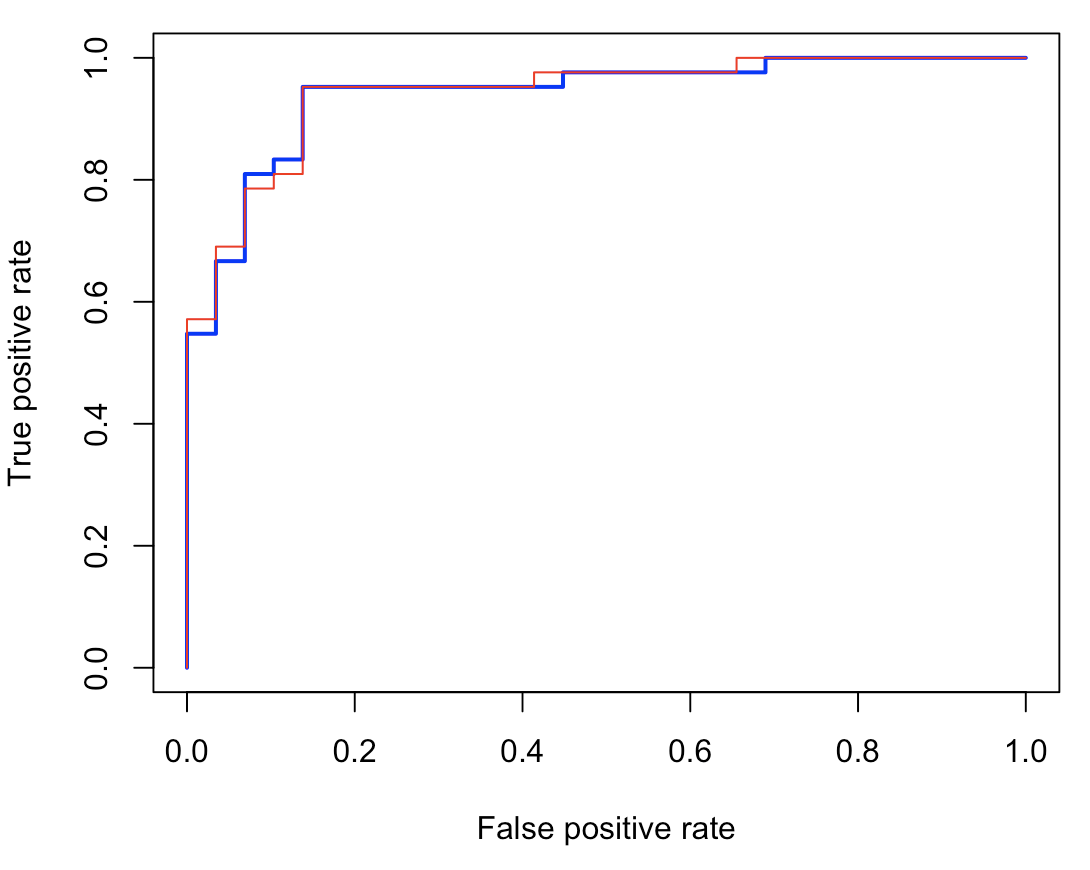
|  |
| --- |
| weights\_0 = **lm**(PRONO~.,**data**=myocarde)$coefficients |

then use

|  |
| --- |
| y\_5\_1 = sigmoid(X %\*% weights\_0) |

In order to see if we get a “good” prediction, let use plot the [ROC curve](https://en.wikipedia.org/wiki/Receiver_operating_characteristic), and compare it with the one we got with a (simple) logistic regression

|  |
| --- |
| **library**(ROCR)  pred = ROCR::prediction(y\_5\_1,myocarde$PRONO)  perf = ROCR::performance(pred,"tpr", "fpr")  **plot**(perf,**col**="blue",lwd=2)  reg = **glm**(PRONO~.,**data**=myocarde,**family**=**binomial**(link = "logit"))  y\_0 = **predict**(reg,type="response")  pred0 = ROCR::prediction(y\_0,myocarde$PRONO)  perf0 = ROCR::performance(pred0,"tpr", "fpr")  **plot**(perf0,add=TRUE,**col**="red") |

  
That’s not bad for a very first attempt. Except that we’ve been cheating here, since we did use ω=β^ols\mathbf{\omega}=\widehat{\mathbf{\beta}}^{ols}ω=β

​ols. How, for real, should we choose those weights?

**Using a loss function**

Well, if we want an “optimal” set of weights, we need to “optimize” an objective function. So we need to quantify the loss of a mistake, between the prediction, and the observation. Consider here a quadratic loss function

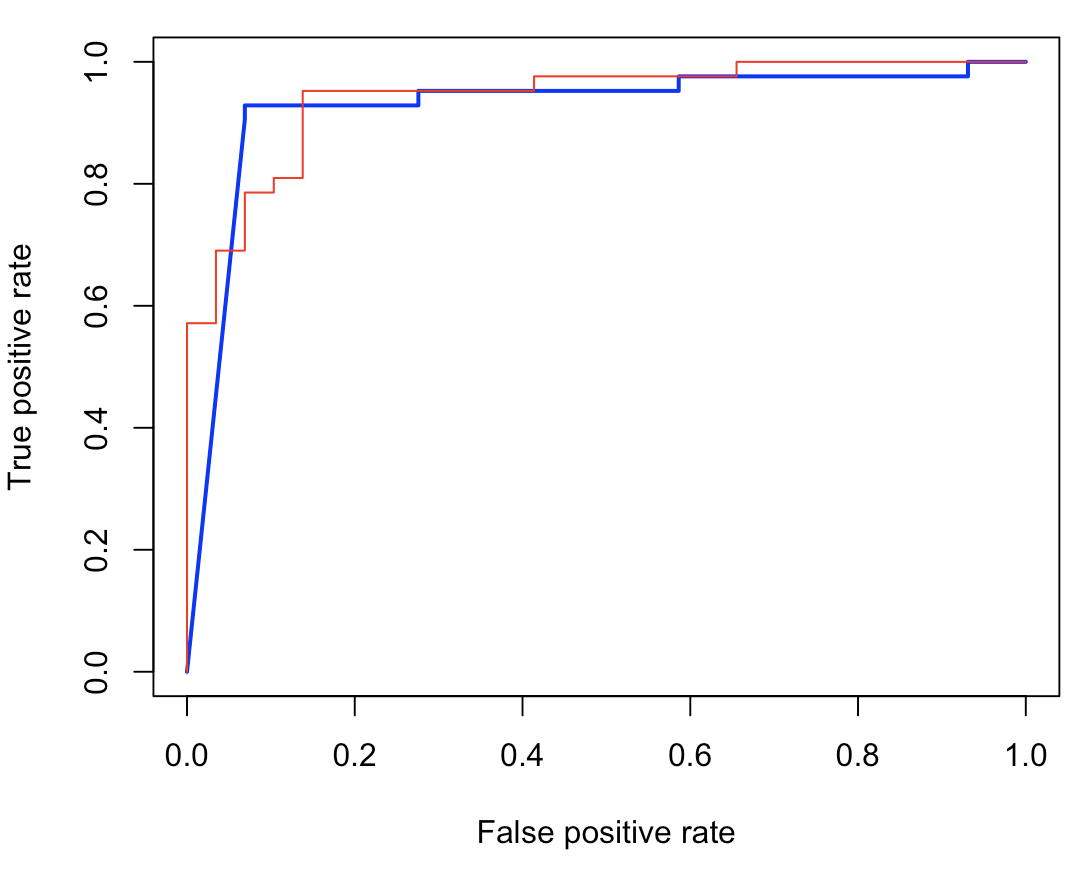
|  |
| --- |
| loss = **function**(**weights**){  **mean**( (myocarde$PRONO-sigmoid(X %\*% **weights**))^2) } |

It might be stupid to use a quadratic loss function for a classification, but here, it’s not the point. We just want to understand what is the algorithm we use, and the loss function ℓ\ellℓ is just one parameter. Then we want to solveω⋆=argmin{1n∑i=1nℓ(yi,f(ω0+xiTω))}\mathbf{\omega}^\star=\text{argmin}\left\lbrace\frac{1}{n}\sum\_{i=1}^n\ell\left(y\_i,f(\omega\_0+\mathbf{x}\_i^T\mathbf{\omega})\right)\right\rbraceω⋆=argmin{n1​i=1∑n​ℓ(yi​,f(ω0​+xiT​ω))}Thus, consider

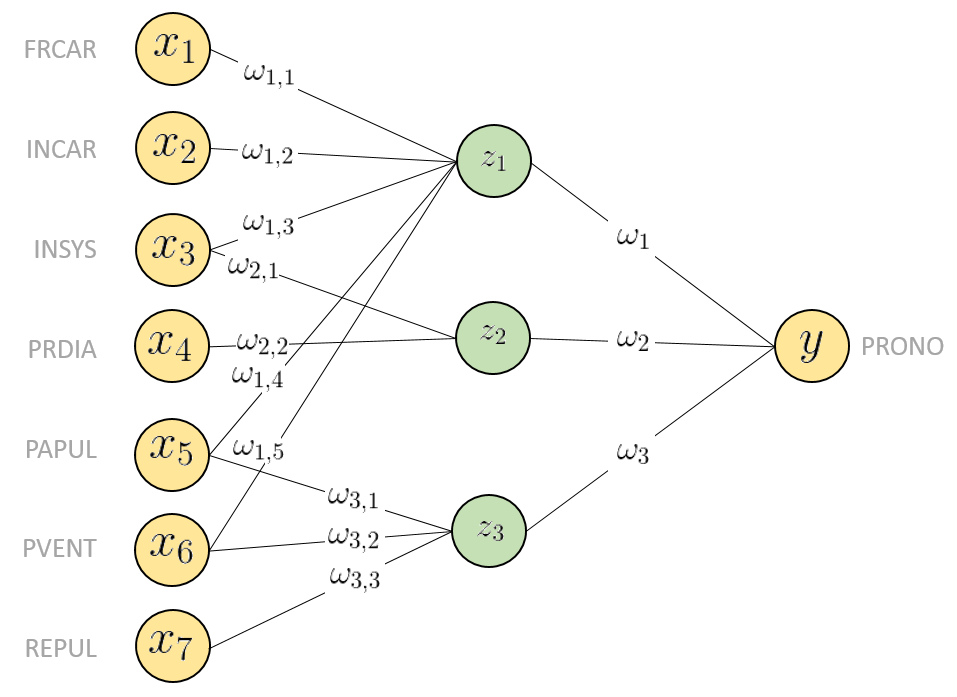
|  |
| --- |
| weights\_1 = **optim**(weights\_0,loss)$par |

(where the starting point is the OLS estimate). Again, to see what’s going on, let us visualize the ROC curve

|  |
| --- |
| y\_5\_2 = sigmoid(X %\*% weights\_1)  pred = ROCR::prediction(y\_5\_2,myocarde$PRONO)  perf = ROCR::performance(pred,"tpr", "fpr")  **plot**(perf,**col**="blue",lwd=2)  **plot**(perf0,add=TRUE,**col**="red") |

  
That’s not amazing, but again, that’s only a first step.

**A single layer**

Let us add a single layer in our network.  
  
Those nodes are connected to the sources (incoming from sources) from the left, and then connected to the sink, on the right. Those nodes are not inter-connected. And again, for that network, we need edges (i.e series of weights). For instance, on the network above, we did add one single layer, with (only) three nodes.

For such a network, the prediction formula is y=f(ω0+∑h=13ωhfh(ωh,0+∑j=1pωh,jxj))\mathbf{y}=f\left( \omega\_0+ \sum\_{h=1}^3\omega\_h f\_h\left(\omega\_{h,0}+ \sum\_{j=1}^p \omega\_{h,j} x\_j\right)\right)y=f(ω0​+h=1∑3​ωh​fh​(ωh,0​+j=1∑p​ωh,j​xj​))or more syntheticallyy=f(ω0+∑h=13ωhfh(ωh,0+xTωh))\mathbf{y}=f\left( \omega\_0+ \sum\_{h=1}^3 \omega\_hf\_h\left(\omega\_{h,0}+ \mathbf{x}^T\mathbf{\omega}\_h\right)\right)y=f(ω0​+h=1∑3​ωh​fh​(ωh,0​+xTωh​))Usually, we consider the same activation function everywhere. Don’t ask me why, I find that weird.

Now, we have a lot of weights to choose. Let us use again OLS estimates

|  |
| --- |
| weights\_1 &lt;- **lm**(PRONO~1+FRCAR+INCAR+INSYS+PAPUL+PVENT,**data**=myocarde)$coefficients  X1 = **as.matrix**(**cbind**(1,myocarde[,**c**("FRCAR","INCAR","INSYS","PAPUL","PVENT")]))  weights\_2 &lt;- **lm**(PRONO~1+INSYS+PRDIA,**data**=myocarde)$coefficients  X2=**as.matrix**(**cbind**(1,myocarde[,**c**("INSYS","PRDIA")]))  weights\_3 &lt;- **lm**(PRONO~1+PAPUL+PVENT+REPUL,**data**=myocarde)$coefficients  X3=**as.matrix**(**cbind**(1,myocarde[,**c**("PAPUL","PVENT","REPUL")])) |

In that case, we did specify edges, and which sources (explanatory variables) should be used for each additional node. Actually, here, other techniques could be have been used, like using a [PCA](https://en.wikipedia.org/wiki/Principal_component_analysis). Each node will then be one of the components. But we’ll use that idea later on…

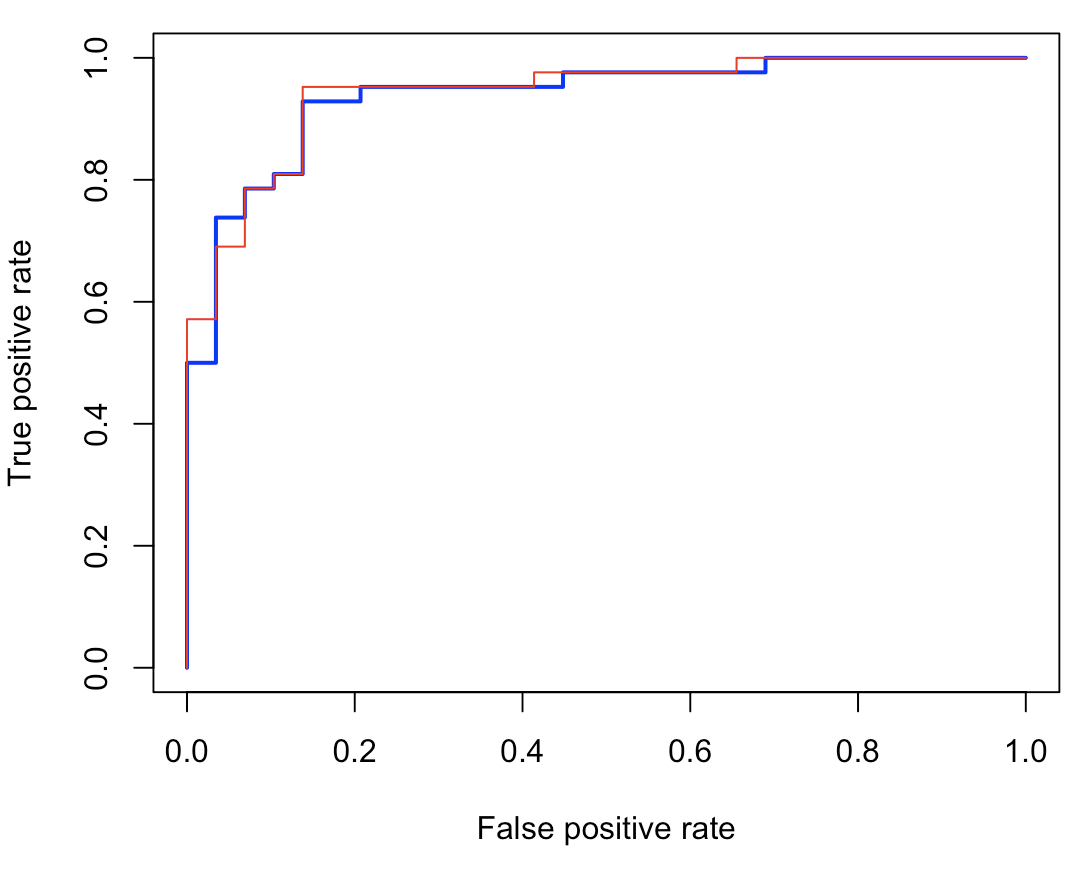
|  |
| --- |
| X = **cbind**(sigmoid(X1 %\*% weights\_1), sigmoid(X2 %\*% weights\_2), sigmoid(X3 %\*% weights\_3)) |

But we’re not done here. Those were weights from the source to the know nodes, in the layer. We still need the weights from the nodes to the sink. Here, let use use a simple average

|  |
| --- |
| **weights** = **c**(1/3,1/3,1/3)  y\_5\_3 &lt;- sigmoid(X %\*% **weights**) |

Again, we can plot the ROC curve to see what we’ve done…

|  |
| --- |
| pred = ROCR::prediction(y\_5\_3,myocarde$PRONO)  perf = ROCR::performance(pred,"tpr", "fpr")  **plot**(perf,**col**="blue",lwd=2)  **plot**(perf0,add=TRUE,**col**="red") |



**On back propagation**

Now, we need some optimal selection of those weights. Observe that with only 3 nodes, there are already (7+1)×3+3=27(7+1)\times3+3=27(7+1)×3+3=27 parameters in that model! Clearly, parcimony is not the major issue when you start using neural nets! If p(x)=f(ω0+∑h=13ωhfh(ωh,0+xTωh))p(\mathbf{x})=f\left( \omega\_0+ \sum\_{h=1}^3 \omega\_hf\_h\left(\omega\_{h,0}+ \mathbf{x}^T\mathbf{\omega}\_h\right)\right)p(x)=f(ω0​+h=1∑3​ωh​fh​(ωh,0​+xTωh​))we want to solveω⋆=argmin{1n∑i=1nℓ(yi,p(xi))}\mathbf{\omega}^\star=\text{argmin}\left\lbrace\frac{1}{n}\sum\_{i=1}^n\ell\left(y\_i,p(\mathbf{x}\_i)\right)\right\rbraceω⋆=argmin{n1​i=1∑n​ℓ(yi​,p(xi​))}for some loss function, which isω⋆=argmin{1n∑i=1n(yi−p(xi))2}\mathbf{\omega}^\star=\text{argmin}\left\lbrace\frac{1}{n}\sum\_{i=1}^n (y\_i-p(\mathbf{x}\_i))^2 \right\rbraceω⋆=argmin{n1​i=1∑n​(yi​−p(xi​))2}for the quadratic norm, orω⋆=argmin{1n∑i=1n(yilog⁡p(xi)+[1−yi]log⁡[1−p(xi)])}\mathbf{\omega}^\star=\text{argmin}\left\lbrace\frac{1}{n}\sum\_{i=1}^n (y\_i\log p(\mathbf{x}\_i)+[1-y\_i]\log [1-p(\mathbf{x}\_i)]) \right\rbraceω⋆=argmin{n1​i=1∑n​(yi​logp(xi​)+[1−yi​]log[1−p(xi​)])}if we want to use [cross-entropy](https://en.wikipedia.org/wiki/Cross_entropy).

For convenience, let us center all the variable we create, otherwise, we get numerical problems.

|  |
| --- |
| center = **function**(z) (z-**mean**(z))/**sd**(z)  loss = **function**(**weights**){  weights\_1 = **weights**[0+(1:7)]  weights\_2 = **weights**[7+(1:7)]  weights\_3 = **weights**[14+(1:7)]  weights\_ = **weights**[21+1:4]  X1=X2=X3=**as.matrix**(myocarde[,1:7])  Z1 = center(X1 %\*% weights\_1)  Z2 = center(X2 %\*% weights\_2)  Z3 = center(X3 %\*% weights\_3)  X = **cbind**(1,sigmoid(Z1), sigmoid(Z2), sigmoid(Z3))  **mean**( (myocarde$PRONO-sigmoid(X %\*% weights\_))^2)} |

Now that we have our objective function, consider some starting points. We can consider weights from a PCA, and then use a gradient descent algorithm,

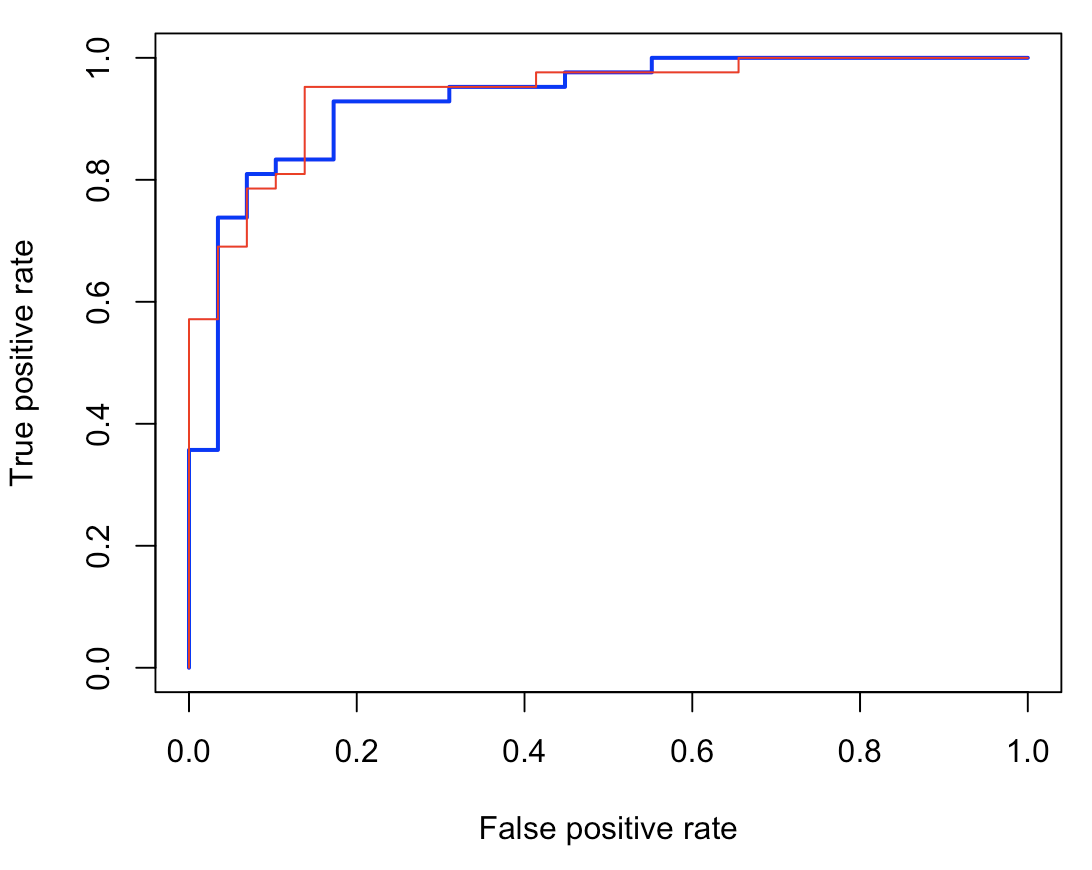
|  |
| --- |
| pca = **princomp**(myocarde[,1:7])  W = get\_pca\_var(pca)$contrib  weights\_0 = **c**(W[,1],W[,2],W[,3],**c**(-1,**rep**(1,3)/3))  weights\_opt = **optim**(weights\_0,loss)$par |

The prediction is then obtained using

|  |
| --- |
| weights\_1 = weights\_opt[0+(1:7)]  weights\_2 = weights\_opt[7+(1:7)]  weights\_3 = weights\_opt[14+(1:7)]  weights\_ = weights\_opt[21+1:4]  X1=X2=X3=**as.matrix**(myocarde[,1:7])  Z1 = center(X1 %\*% weights\_1)  Z2 = center(X2 %\*% weights\_2)  Z3 = center(X3 %\*% weights\_3)  X = **cbind**(1,sigmoid(Z1), sigmoid(Z2), sigmoid(Z3))  y\_5\_4 = sigmoid(X %\*% weights\_) |

And as previously, why not plot the ROC curve of that model

|  |
| --- |
| pred = ROCR::prediction(y\_5\_4,myocarde$PRONO)  perf = ROCR::performance(pred,"tpr", "fpr")  **plot**(perf,**col**="blue",lwd=2)  **plot**(perf,add=TRUE,**col**="red") |

  
That’s not too bad. But with 27 coefficients, that’s what we would expect, no?

**Using nnet() function**

That’s more or less what is done in neural nets functions. Let us now have a look at some dedicated R functions.

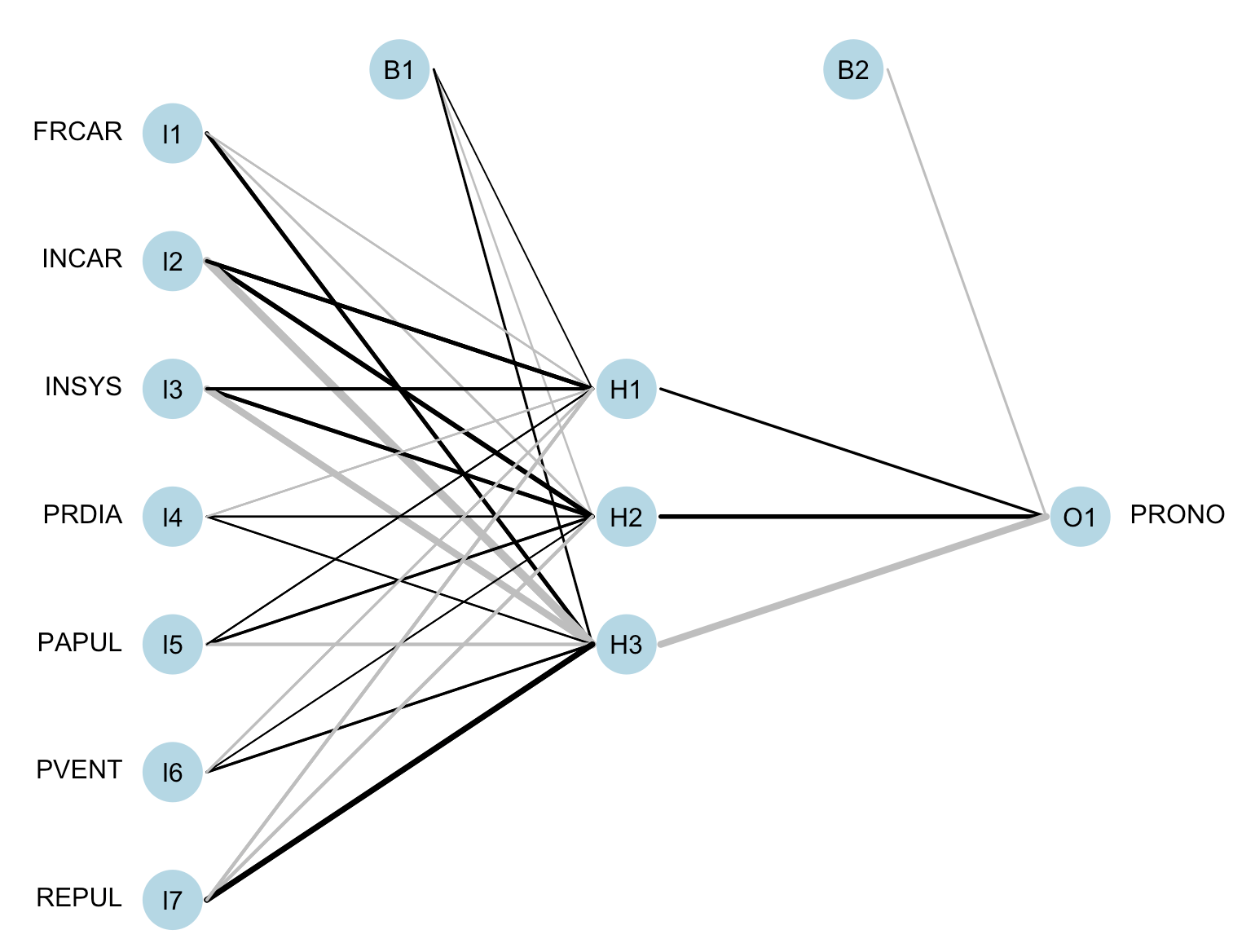
|  |
| --- |
| **library**(nnet)  myocarde\_minmax = myocarde  minmax = **function**(z) (z-**min**(z))/(**max**(z)-**min**(z))  **for**(j **in** 1:7) myocarde\_minmax[,j] = minmax(myocarde\_minmax[,j]) |

Here, variables are linearly transformed, to take values in (0,1)(0,1)(0,1). Then we can construct a neural network with one single layer, and three nodes,

|  |
| --- |
| model\_nnet = nnet(PRONO~.,**data**=myocarde\_minmax,size=3)  **summary**(model\_nnet)  a 7-3-1 network **with** 28 **weights**  **options** were -  b-&gt;h1 i1-&gt;h1 i2-&gt;h1 i3-&gt;h1 i4-&gt;h1 i5-&gt;h1 i6-&gt;h1 i7-&gt;h1  -9.60 -1.79 21.00 14.72 -20.45 -5.05 14.37 -17.37  b-&gt;h2 i1-&gt;h2 i2-&gt;h2 i3-&gt;h2 i4-&gt;h2 i5-&gt;h2 i6-&gt;h2 i7-&gt;h2  4.72 2.83 -3.37 -1.64 1.49 2.12 2.31 4.00  b-&gt;h3 i1-&gt;h3 i2-&gt;h3 i3-&gt;h3 i4-&gt;h3 i5-&gt;h3 i6-&gt;h3 i7-&gt;h3  -0.58 -6.03 25.14 18.03 -1.19 7.52 -19.47 -12.95  b-&gt;o h1-&gt;o h2-&gt;o h3-&gt;o  -1.32 29.00 -10.32 26.27 |

Here, it is the complete full network. And actually, there are (online) some functions that can he used to visualize that network

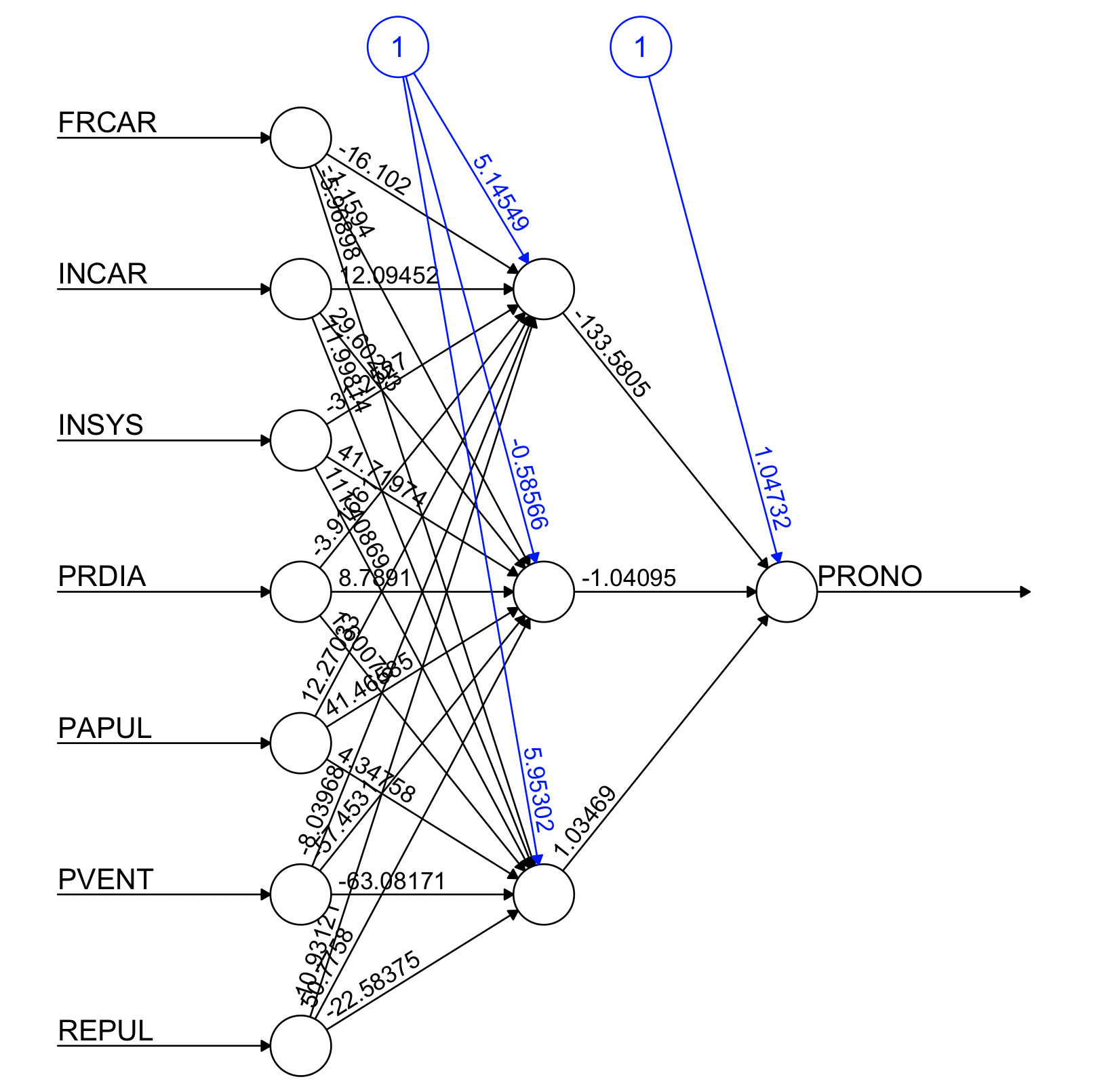
|  |
| --- |
| **library**(devtools)  source\_url('https://gist.githubusercontent.com/fawda123/7471137/raw/466c1474d0a505ff044412703516c34f1a4684a5/nnet\_plot\_update.r')  plot.nnet(model\_nnet) |

  
Nice, isn’t it? We clearly see the intermediary layer, with three nodes, and on top the constants. Edges are the plain lines, the darker, the heavier (in terms of weights).

**Using neuralnet()**

Other R functions can actually be considered.

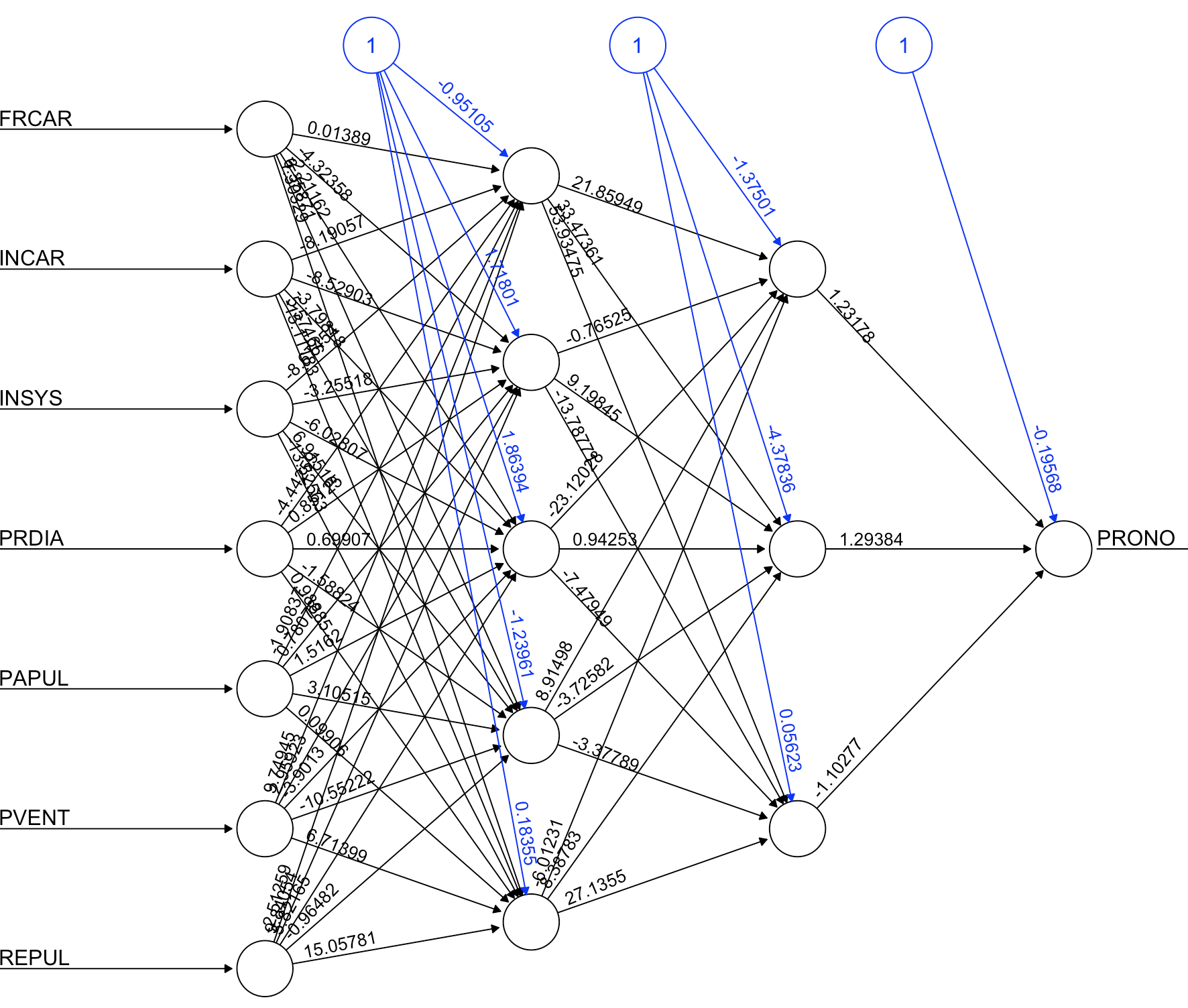
|  |
| --- |
| **library**(neuralnet)  model\_nnet = neuralnet(**formula**(**glm**(PRONO~.,**data**=myocarde\_minmax)),  myocarde\_minmax,hidden=3, act.fct = sigmoid)  **plot**(model\_nnet) |

  
Again, for the same network structure, with one (hidden) layer, and three nodes in it.

**Network with multiple layers**

The good thing is that it’s not possible to add more layers. Like two layers. Nodes from the first layer are no longuer connected with the sink, but with nodes in the second layer. And those nodes will then be connected to the sink. We now have something like  
p(x)=f(ω0+∑h=13ωhfh(ωh,0+zhTωh))p(\mathbf{x})=f\left( \omega\_0+ \sum\_{h=1}^3 \omega\_h f\_h\left(\omega\_{h,0}+ \mathbf{z}\_h^T\mathbf{\omega}\_h\right)\right)p(x)=f(ω0​+h=1∑3​ωh​fh​(ωh,0​+zhT​ωh​))wherezh=f(ωh,0+∑j=1khωh,jfh,j(ωh,j,0+xTωh,j))\mathbf{z}\_h=f\left( \omega\_{h,0}+ \sum\_{j=1}^{k\_h} \omega\_{h,j} f\_{h,j}\left(\omega\_{h,j,0}+ \mathbf{x}^T\mathbf{\omega}\_{h,j}\right)\right)zh​=f(ωh,0​+j=1∑kh​​ωh,j​fh,j​(ωh,j,0​+xTωh,j​))I may be rambling here (a little bit) but that’s a lot of parameters. Here is the visualization of such a network,

|  |
| --- |
| **library**(neuralnet)  model\_nnet = neuralnet(**formula**(**glm**(PRONO~.,**data**=myocarde\_minmax)),  myocarde\_minmax,hidden=3, act.fct = sigmoid)  **plot**(model\_nnet) |



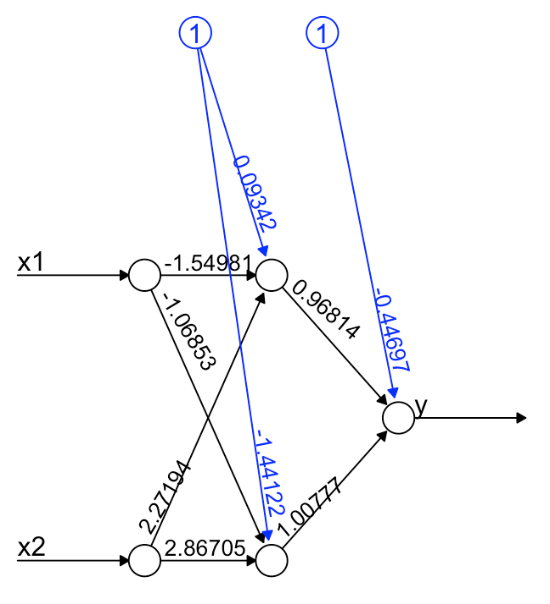
**Application**

Let us get back on our simple dataset, with only two covariates.

|  |
| --- |
| **library**(neuralnet)  df\_minmax =**df**  df\_minmax$y=(df\_minmax$y=="1")\*1  minmax = **function**(z) (z-**min**(z))/(**max**(z)-**min**(z))  **for**(j **in** 1:2) df\_minmax[,j] = minmax(**df**[,j])  X = **as.matrix**(**cbind**(1,df\_minmax[,1:2])) |

Consider only one layer, with two nodes

|  |
| --- |
| model\_nnet = neuralnet(**formula**(**lm**(y~.,**data**=df\_minmax)),  df\_minmax,hidden=**c**(2))  **plot**(model\_nnet) |

  
Here, we did not specify it, but the activation function is the sigmoid (actually, it is called logistic here)

|  |
| --- |
| model\_nnet$act.fct  **function** (x)  {  1/(1 + **exp**(-x))  }    **attr**(,"type")  [1] "logistic"  f=model\_nnet$act.fct |

The weights (on the figure) can be obtained using

|  |
| --- |
| w0 = model\_nnet$weights[[1]][[2]][,1]  w1 = model\_nnet$weights[[1]][[1]][,1]  w2 = model\_nnet$weights[[1]][[1]][,2] |

Now, to get our prediction,  
we should usep(x)=f(ω0+ω1f(ω1,0+xhTω1,1:2)+ω1f(ω2,0+xhTω2,1:2))p(\mathbf{x})=f\left( \omega\_0+ \omega\_1 f(\omega\_{1,0}+ \mathbf{x}\_h^T\mathbf{\omega}\_{1,1:2})+\omega\_1 f(\omega\_{2,0}+ \mathbf{x}\_h^T\mathbf{\omega}\_{2,1:2})\right)p(x)=f(ω0​+ω1​f(ω1,0​+xhT​ω1,1:2​)+ω1​f(ω2,0​+xhT​ω2,1:2​))which can be obtained using

|  |
| --- |
| f(**cbind**(1,f(X%\*%w1),f(X%\*%w2))%\*%w0)  [,1]  [1,] 0.7336477343  [2,] 0.7317999050  [3,] 0.7185803540  [4,] 0.7404005280  [5,] 0.7518482779  [6,] 0.4939774149  [7,] 0.4965876378  [8,] 0.7101714888  [9,] 0.5050760026  [10,] 0.5049877644 |

Unfortunately, it is not the output of the model here,

|  |  |
| --- | --- |
| neuralnet::prediction(model\_nnet)  Data Error: 0;  $rep1  x1 x2 y  1 0.1250 0.0000000000 0.02030470787  2 0.0625 0.1176470588 0.89621706711  3 0.9375 0.2352941176 0.01995171956  4 0.0000 0.4705882353 1.10849420363  5 0.5000 0.4705882353 -0.01364966058  6 0.3125 0.5294117647 -0.02409150561  7 0.6875 0.8235294118 0.93743057765  8 0.3750 0.8823529412 1.01320924782  9 1.0000 0.9058823529 1.04805134309  10 0.5625 1.0000000000 1.00377379767 | |
| If anyone has a clue, I’d be glad to know what went wrong here… I find that odd to have outputs outside the (0,1)(0,1)(0,1) interval, but the output is neitherp(x)=ω0,0+ω0,1f(ω1,0+xhTω1,1:2)+ω0,2f(ω2,0+xhTω2,1:2)p(\mathbf{x})=\omega\_{0,0}+ \omega\_{0,1} f(\omega\_{1,0}+ \mathbf{x}\_h^T\mathbf{\omega}\_{1,1:2})+\omega\_{0,2} f(\omega\_{2,0}+ \mathbf{x}\_h^T\mathbf{\omega}\_{2,1:2})p(x)=ω0,0​+ω0,1​f(ω1,0​+xhT​ω1,1:2​)+ω0,2​f(ω2,0​+xhT​ω2,1:2​)**cbind**(1,f(X%\*%w1),f(X%\*%w2))%\*%w0  [,1]  [1,] 1.01320924782  [2,] 1.00377379767  [3,] 0.93743057765  [4,] 1.04805134309  [5,] 1.10849420363  [6,] -0.02409150561  [7,] -0.01364966058  [8,] 0.89621706711  [9,] 0.02030470787  [10,] 0.01995171956 |