Neural Nets

Rob McCulloch

1. The Neural Net Model for Numeric Y
2. Size and Decay
3. Iterative Fitting and Random Starting Values
4. How Does it Work?
5. Neural Nets for Binary Y
6. How Does it Work Again, XOR

1. The Neural Net Model for Numeric Y

We will look at two examples of fitting neural nets: the zagat data and the tabloid data.

zagat has numeric y and tabloid has binary y.

zagat is simple enough that it is a good place to get acquainted with the basic ideas.

Tabloid is more realistic.

the x's must be numeric !!!

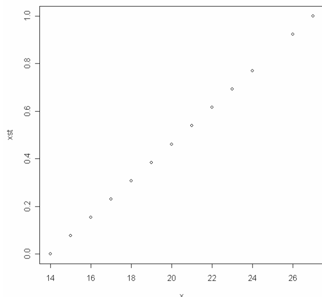
Since we will be regularizing we will have to standardize (as usual).

Here is the zagat data:

```
zag = read.table("zagat.txt",header=T)
summary(zag)
      food           decor          service          price      
 Min.   :14.00   Min.   : 2.00   Min.   :10.00   Min.   :11.00  
 1st Qu.:18.00   1st Qu.:14.00   1st Qu.:16.00   1st Qu.:25.00  
 Median :20.00   Median :16.50   Median :18.00   Median :32.50  
 Mean   :19.61   Mean   :16.58   Mean   :17.77   Mean   :33.32  
 3rd Qu.:21.00   3rd Qu.:20.00   3rd Qu.:20.00   3rd Qu.:41.00  
 Max.   :27.00   Max.   :28.00   Max.   :26.00   Max.   :65.00  
```

Let's rescale so that each x is in (0,1).

```
> x = zag$food
> xst = (x-min(x))/(max(x)-min(x))
> summary(xst)
  Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
0.0000  0.3077  0.4615  0.4312  0.5385  1.0000
> plot(x,xst)
```



Subtract off the min and divide by max-min.

The other obvious way to standardize is to subtract off the mean and divide by the standard deviation.

5

Standardize each of the tree zag x's:

```
minv = rep(0,3)
maxv = rep(0,3)
zagsc = zag
for(i in 1:3) {
minv[i] = min(zag[[i]])
maxv[i] = max(zag[[i]])
zagsc[[i]] = (zag[[i]]-minv[i])/(maxv[i]-minv[i])
}
```

First, I will just use the one x, food, to keep things simple.

First you have to load the neural net library, nnet:

```
> library(nnet)
```

Here is the command:

```
> znn = nnet(price~food,zagsc,size=3,decay=.1,linout=T)
```

As usual, a data structure is returned containing
(in some possibly obscure way!!)
the results.

The first two arguments are familiar.

linout=T is appropriate for a numeric y.

size and decay, are the two key parameters for controlling the flexibility of the neural net fit.

After we understand the basic structure of the model we will discuss these.

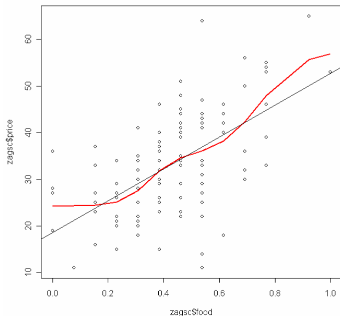
These will be the parameters that control the complexity of the model like k in KNN and λ in the LASSO.

Let's have a look at the fits.
Just as with trees and regression,
we use the predict command:

```
> fznn = predict(znn,zagsc)
> plot(zagsc$food,zagsc$price)
> oo = order(zagsc$food)
> lines(zagsc$food[oo],fznn[oo],col="red",lwd=2)
> abline(lm(price~food,zagsc)$coef)
```

zzn: nnet fit
zagsc: data frame with scaled x's.

The red is the nn fit and the straight line is linear regression.



What is the structure of the model ?

```
> summary(znn)
a 1-3-1 network with 10 weights
options were - linear output units  decay=0.1
 b->h1 i1->h1
  4.35  -0.24
 b->h2 i1->h2
 -7.42  21.41
 b->h3 i1->h3
 -9.93  13.28
 b->o h1->o h2->o h3->o
12.33 12.09 10.70 22.74
```

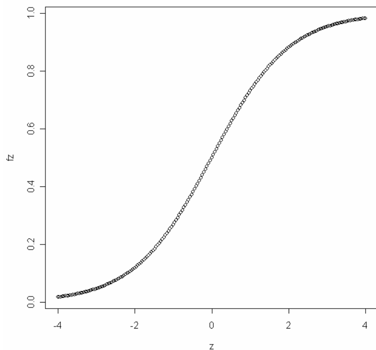
What does this mean ?

First note:

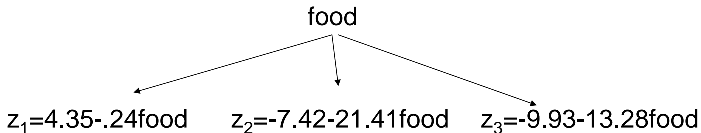
Let, $F(z) = \frac{e^z}{1+e^z}$

```
> z =  (-100:100)/25
> fz  = exp(z)/(1+exp(z))
> plot(z,fz)
```

This F is often called the logistic function.



Let, $F(z) = \frac{e^z}{1 + e^z}$



$y = 12.33 + 12.09F(z_1) + 10.70F(z_2) + 22.74F(z_3)$

1. Form several different linear functions of the x's.
2. Apply the logistic function to each.
3. Take a linear combination of the results of 2.

The z's are called the *hidden layer*.

Each of the z's (linear function) is called a unit.

In the call to nnet the parameter "size" is the number of units in the hidden layer.

Here is the fit of a neural net with 5 units in the hidden layer.

```
> znn = nnet(price~food,zagsc,size=5,decay=.1,linout=T)
> summary(znn)
a 1-5-1 network with 16 weights
options were - linear output units  decay=0.1
 b->h1 i1->h1
  2.70   0.40
 b->h2 i1->h2
 -9.69  13.02
 b->h3 i1->h3
  0.71  -5.99
 b->h4 i1->h4
 -6.63  19.76
 b->h5 i1->h5
  2.70   0.39
 b->o h1->o h2->o h3->o h4->o h5->o
 7.41  7.00 23.30  7.62 13.56  6.99
```

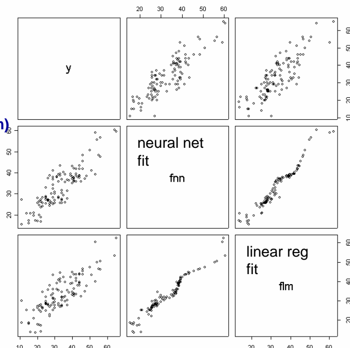
the coefficients for the 5 linear functions of x, the 5 z's.

the coefficients for the five f(z)

14

All three x's

```
> znn = nnet(price~.,zagsc,size=5,decay=.1,linout=T)
> fznn = predict(znn,zagsc)
>
> zlm = lm(price~.,zagsc)
> fzlm = predict(zlm,zagsc)
>
> temp = data.frame(y=zagsc$price,fnn=fznn,flm=fzlm)
> pairs(temp)
>
> print(cor(temp$y,temp$fnn))
[1] 0.867858
> print(cor(temp$y,temp$flm))
[1] 0.829138
> print(cor(temp$fnn,temp$flm))
[1] 0.9705388
```



Using R^2, nnet fit is a little better in sample, but quite similar to the linear regression fit.

15

The fitted model with 3 x's and 5 units in the hidden layer.

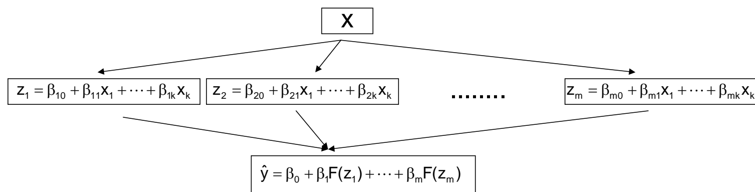
a 3-5-1 network. 3 x's, 5 units, 1 y.

```
> summary(znn)
a 3-5-1 network with 26 weights
options were - linear output units  decay=0.1
 b->h1 i1->h1 i2->h1 i3->h1
 -5.64  -2.64  11.48   8.31
 b->h2 i1->h2 i2->h2 i3->h2
-18.09  20.98  19.53  -0.64
 b->h3 i1->h3 i2->h3 i3->h3
  1.45  -4.79   1.95   1.00
 b->h4 i1->h4 i2->h4 i3->h4
  1.44  -0.94  -7.64   3.35
 b->h5 i1->h5 i2->h5 i3->h5
-20.40   9.93  14.09   4.48
  b->o  h1->o  h2->o  h3->o  h4->o  h5->o
  5.15  13.15  13.33   6.75  12.51  24.42
```

# of weights = 4*5 + 6.

The General Model

A k-m-1 network.



k x's
m hidden units.

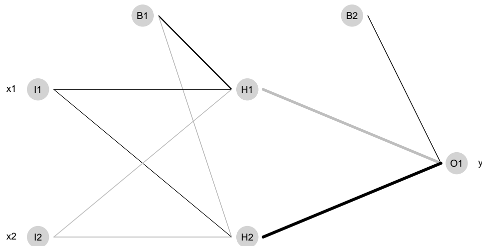
Why on earth, would this work ?

If you find this a bit hard to grasp, you are not alone.

A basic knock on neural nets is that it is hard to interpret the model !!

Another way to draw the model has a node and a connect for each x and the intercept.

A 2-2-1 net:



In neural-net world the intercepts are called the biases and the coefficients are called the weights.

2. Size and Decay

The size of the neural net is the number of units in the hidden layer.

Clearly, the more units the richer the model.

The more we are able to fit the data.

The more we are able to overfit the data.

The decay parameter is the L2 regularlization parameter.

Fit minimizes:

$$\text{error} + \text{decay} \; * \; \sum \text{coefficient}^2$$

where, for example,

$$\text{error} = \sum (y_i - \hat{y}_i)^2.$$

Whether a coefficient is large or small depends on the units of the x's.

This is the fundamental reason we rescale the x's.

Only if the x's are on the same scale does the decay parameter work properly.

People have found that in practice the decay parameter is useful for walking the fit/overfit line.

21

Let's do a little experiment with size and decay to see how they affect the fit with price on food (so we can see what happens).

Four different fits with size = 3 and 50, decay = .5 and .00001.

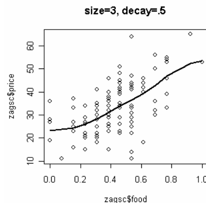
```
znn1 = nnet(price~food,zagsc,size=3,decay=.5,linout=T)
znn2 = nnet(price~food,zagsc,size=3,decay=.00001,linout=T)
znn3 = nnet(price~food,zagsc,size=50,decay=.5,linout=T)
znn4 = nnet(price~food,zagsc,size=50,decay=.00001,linout=T)
temp = data.frame(price = zagsc$price, food = zagsc$food)
znnf1 = predict(znn1,temp)
znnf2 = predict(znn2,temp)
znnf3 = predict(znn3,temp)
znnf4 = predict(znn4,temp)
```

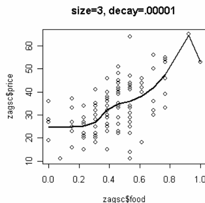
For each model fit we'll plot y vs x and then the fits on top:

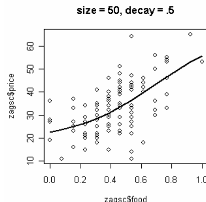
```
> par(mfrow=c(2,2))
> plot(zagsc$food,zagsc$price)
> lines(zagsc$food[oo],znnf1[oo],lwd=2)
> title("size=3, decay=.5")
> plot(zagsc$food,zagsc$price)
> lines(zagsc$food[oo],znnf2[oo],lwd=2)
> title("size=3, decay=.00001")
> plot(zagsc$food,zagsc$price)
> lines(zagsc$food[oo],znnf3[oo],lwd=2)
> title("size = 50, decay = .5")
> plot(zagsc$food,zagsc$price)
> lines(zagsc$food[oo],znnf4[oo],lwd=2)
> title("size = 50, decay = .00001")
```

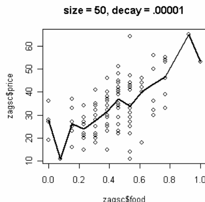
Left to right we can see that lower decay means a more flexible fit, the coefficients are freer.

With low decay (right two plots) increasing the size really frees up the fit.









With high decay adding more units does not seem to hurt !!

We also see that even with 50 hidden units,
a large decay parameter can restrain the fit.

In practice, this has lead to the following strategy
for fitting neural nets.

1. Fix a large number of hidden units.

2. Use the three set approach or cross validation
   to choose the decay parameter.

Of course, you could use cv or three sets to
choose both size and decay.

"Generally speaking it is better to have too many hidden units than too few. With too few hidden units, the model might not have enough flexibility to capture the nonlinearities in the data; with too many hidden units, the extra weights can be shrunk toward zero if appropriate regularization (decay) is used. Typically the number of hidden units is somewhere in the range of 5 to 100, with the number increasing with the number of of inputs and the number of training cases. It is most common to put down a reasonably large number of units and train them with regularization. Some researchers have used cross-validation to estimate the optimal number, but this seems unnecessary if cross-validation is used to estimate the regularization parameter. Choice of the number of hidden layers is guided by background knowledge and experimentation."

"The Elements of Statistical Learning,

Data Mining, Inference, and Prediction"

26

3. Iterative Fitting and Random Starting Values

There are some very things about fitting neural nets to data that don't come up in a lot of our other models.

Except for some linear algebra, we know how do least squares: $\hat{\beta} = (X'X)^{-1}X'y$.

Computing the logit MLE is an iterative optimization, but, for moderately size problems, it converges pretty fast.

Numerically fitting neural nets is trickier.

What does our single layer model with *m* units, a numeric y and just one x look like?

$$\hat{y} = f(x, b) = b_0 + \sum_{j=1}^{m} b_j F(b_0^j + b_1^j x)$$

e.g. $m = 2$

$$f(x, b) = b_0 + b_1 F(b_0^1 + b_1^1 x) + b_2 F(b_0^2 + b_1^2 x)$$

$$b = (b_0, b_1, b_2, b_0^1, b_1^1, b_0^2, b_1^2)$$

More generally,

$$\hat{y} = f(x, b) = b_0 + \sum_{j=1}^{m} b_j \, F(b_0^j + (b^j)' x)$$

$b = (b_0, b_1, \ldots, b_m, b_0^1, b_0^2, \ldots b_0^m, b^1, b^2, \ldots, b^m).$

So,

to fit we solve

$$\min_b \; \sum_{i=1}^{n} (y_i - f(x_i, b))^2 + d \, ||b||^2$$

where d is the decay and we are summing over observations in our training data.

Given the size and the decay, it is no joke to fit a neural net.

It is an iterative optimization.

The `nnet` package uses the BFGS option of the the `R` `optim` command (BroydenFletcherGoldfarbShanno algorithm).

BFGS is a version of Newton's methods (more later?).

We say the iterative fitting has converged if
there is little difference between subsequent fits.

Sometimes it can iterate a long time and
not converge !!

Up to now,
I have cut
out the output
about “iter”.

The algorithm
iterated
100 times
and then quit.

It had not
“converged yet”
in the sense
that the fit was
still changing
after 100 iterations.

```
> znn3 = nnet(price~food,zagsc,size=50,decay=.5,linout=T)
# weights:  151
initial  value 149902.637210
iter  10 value 9483.187518
iter  20 value 9233.424055
iter  30 value 9065.532626
iter  40 value 8987.139367
iter  50 value 8960.219511
iter  60 value 8933.476681
iter  70 value 8922.799791
iter  80 value 8919.391429
iter  90 value 8915.018123
iter 100 value 8911.423203
final  value 8911.423203
stopped after 100 iterations
```


32

You can control the maximum number of iterations.

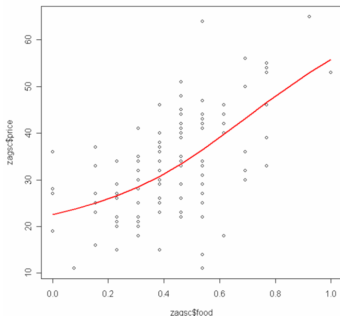
```
> znn3 = nnet(price~food,zagsc,size=50,decay=.5,linout=T,maxit=20)
# weights:  151
initial  value 143500.875215
iter  10 value 9043.417735
iter  20 value 8940.665729
final  value 8940.665729
stopped after 20 iterations
```

You can also control the convergence criterion but let’s not get into that.

33

For the zagat data with size=50 and decay = .5 it takes 300 iterations to converge !

```
> znn3 = nnet(price~food,zagsc,size=50,decay=.5,linout=T,maxit=1000)
# weights:  151
initial  value 143671.085017
iter  10 value 10210.153040
iter  20 value 9412.824067
iter  30 value 9141.437946
iter  40 value 9050.086653
iter  50 value 8987.998898
iter  60 value 8945.542710
iter  70 value 8923.052784
iter  80 value 8913.481971
iter  90 value 8909.269247
iter 100 value 8906.842213
iter 110 value 8905.212133
iter 120 value 8904.352179
iter 130 value 8903.167414
iter 140 value 8901.816040
iter 150 value 8901.066403
iter 160 value 8900.600340
iter 170 value 8900.283331
iter 180 value 8900.073722
iter 190 value 8899.822352
iter 200 value 8899.591261
iter 210 value 8899.409385
iter 220 value 8899.309109
iter 230 value 8899.211293
iter 240 value 8899.131651
iter 250 value 8899.049650
iter 260 value 8898.994058
iter 270 value 8898.953519
iter 280 value 8898.921090
iter 290 value 8898.903097
iter 300 value 8898.884537
final  value 8898.880799
converged
```



Does not look too different from what we had before (maybe a bit smoother).

Starting Values

Where do you start the iterations ?

R (and lots of other software) starts at randomly chosen coefficient values.

The default is that each coefficient is drawn from the uniform distribution of [-.7,.7].

This only makes sense if we standardize the x's.

So,

You can fit a neural net.

Do it again,

and get a different answer !!!!!!!!!!!!!!!!!!!!

Unless, you set the seed that is !

```
> set.seed(23)
> temp  = nnet(price~food,zagsc,size=2,decay=.001)
# weights:  7
initial  value 136686.571683
iter  10 value 133022.696234
iter  20 value 133014.160947
final  value 133014.130896
converged
> summary(temp)
a 1-2-1 network with 7 weights
options were - decay=0.001
 b->h1 i1->h1
 -0.93   0.55
 b->h2 i1->h2
  1.60  -0.40
 b->o h1->o h2->o
 9.21  4.52  3.68
> temp = nnet(price~food,zagsc,size=2,decay=.001)
# weights:  7
initial  value 136030.030297
iter  10 value 133022.799207
iter  20 value 133014.161079
final  value 133014.127113
converged
> summary(temp)
a 1-2-1 network with 7 weights
options were - decay=0.001
 b->h1 i1->h1
 -0.21  -0.10
 b->h2 i1->h2
  1.64   1.45
 b->o h1->o h2->o
 8.49  5.04  2.40
```

First fit is exactly the same.

second fit is different !

```
> set.seed(23)
> temp = nnet(price~food,zagsc,size=2,decay=.001)
# weights:  7
initial  value 136686.571683
iter  10 value 133022.696234
iter  20 value 133014.160947
final  value 133014.130896
converged
> summary(temp)
a 1-2-1 network with 7 weights
options were - decay=0.001
 b->h1 i1->h1
 -0.93   0.55
 b->h2 i1->h2
  1.60  -0.40
 b->o h1->o h2->o
 9.21  4.52  3.68
```

37

The optimization problem in neural nets is very difficult.

There are many local minimima and there is no way to know if you are at a good one.

The solutions you iterate to can be radically different!!

This makes using neural nets quite tricky in practice.

4. How Does it Work?

How could this possibly work???!!!

Let's fit a few simple examples and see how the pieces add up to the overall fit.

What does our single layer model with a numeric y and just one x look like?

$$\hat{y} = f(x, b) = b_0 + \sum_{j=1}^{m} b_j F(b_0^j + b_1^j x)$$

e.g. $m = 2$

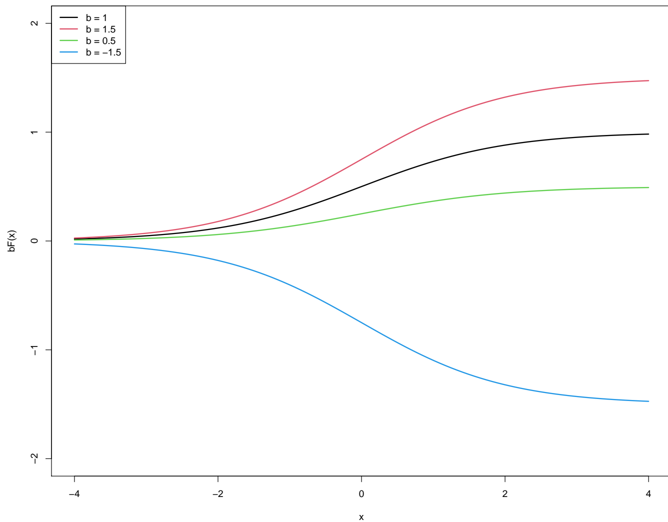
$$f(x, b) = b_0 + b_1 F(b_0^1 + b_1^1 x) + b_2 F(b_0^2 + b_1^2 x)$$

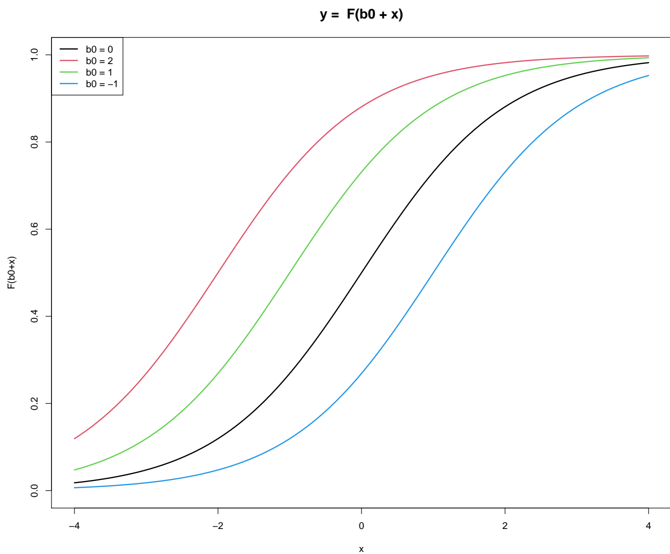
$$b = (b_0, b_1, b_2, b_0^1, b_1^1, b_0^2, b_1^2)$$

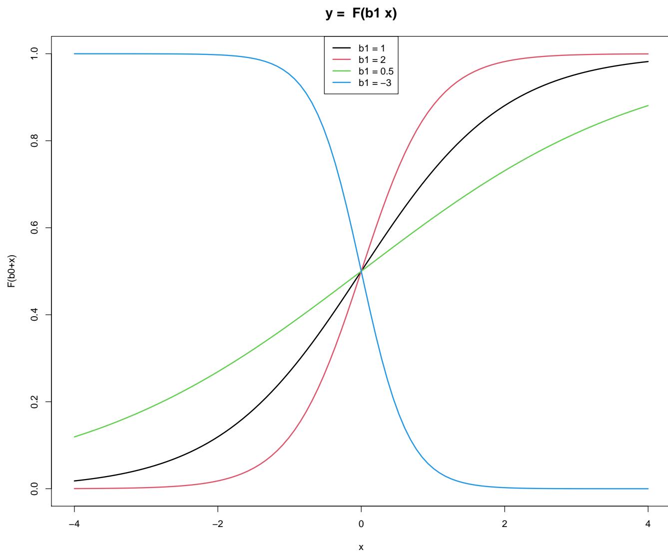
So,

we are just adding up functions of the form

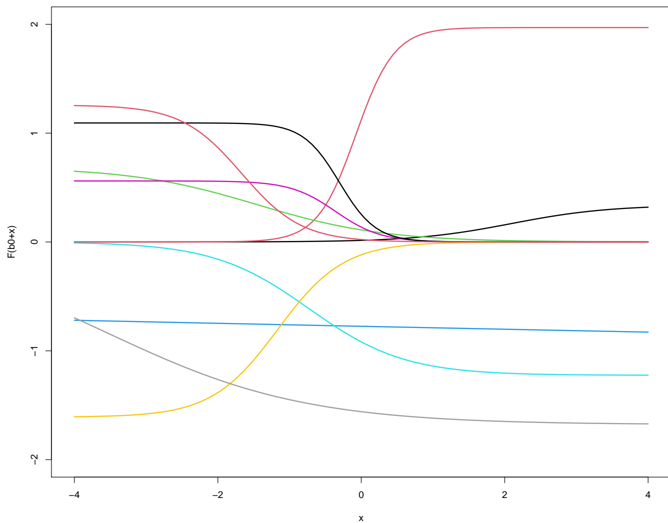
$$g(x) = b\,F(b_0 + b_1\,x)$$


y = b F(x)
b = 1
b = 1.5
b = 0.5
b = −1.5
bF(x)
2
1
0
−1
−2
−4
−2
0
2
4
x


y = F(b0 + x)
b0 = 0
b0 = 2
b0 = 1
b0 = −1
F(b0+x)
0.0
0.2
0.4
0.6
0.8
1.0
−4
−2
0
2
4
x



y = F(b1 x)
b1 = 1
b1 = 2
b1 = 0.5
b1 = −3
F(b0+x)
x
0.0
0.2
0.4
0.6
0.8
1.0
−4
−2
0
2
4


y = b F(b0 + b1 x)
F(b0+x)
x
2
1
0
−1
−2
−4
−2
0
2
4

Let's calculate the pieces for a simple example and see how they can add up to a decent fit.

```
x = zagsc$food
y = zagsc$price

z1 = 4.35  -0.24 *x
z2 = -7.42 +21.41*x
z3 = -9.93 +13.28*x
```

The three linear functions of x, the z's.

```
f1 = 12.09*exp(z1)/(1+exp(z1))
f2 = 10.7*exp(z2)/(1+exp(z2))
f3 = 22.74*exp(z3)/(1+exp(z3))
```

*coefficent * f(z)*

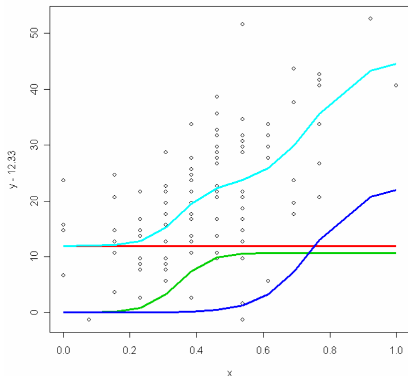
```
plot(x,y-12.33)
lines(x[oo],f1[oo],col=2)
lines(x[oo],f2[oo],col=3)
lines(x[oo],f3[oo],col=4)
lines(x[oo],(f1+f2+f3)[oo],col=5)
```

```
b->h1 i1->h1
 4.35  -0.24
b->h2 i1->h2
-7.42  21.41
b->h3 i1->h3
-9.93  13.28
b->o h1->o h2->o h3->o
12.33 12.09 10.70 22.74
```

cyan:fit

red:first component
blue:second
green:third

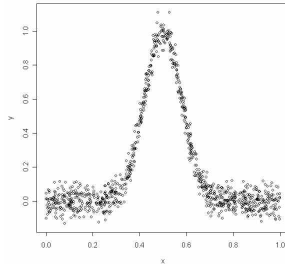
Wow,
scary and
cool !



39

How would you fit a bump?

```
set.seed(23)
x = runif(1000)
x = sort(x)
y = exp(-80*(x-.5)*(x-.5)) + .05*rnorm(1000)
plot(x,y)
df = data.frame(y=y,x=x)
```

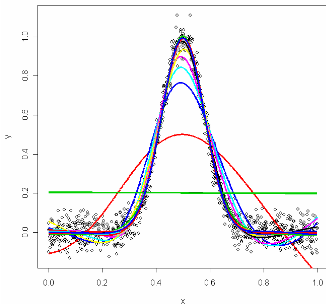


```
plot(x,y)
sz =  3
for(i in 1:20) {
nnsim = nnet(y~x,df,size=sz,decay = 1/2^i,linout=T,maxit=1000)
simfit = predict(nnsim,df)
lines(x,simfit,col=i,lwd=3)
print(i)
readline()
}
```

Try various decay values.

With 3 units
it takes a
small decay.

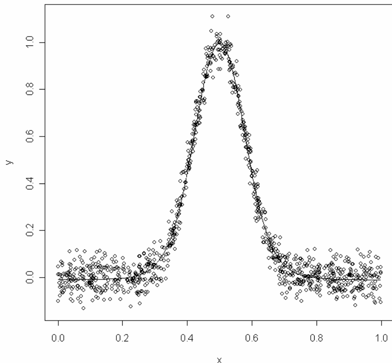
decay = 1/2^12
works.



41

```
nnsim = nnet(y~x,df,size=3,decay=1/2^12,linout=T,maxit=1000)
thefit = predict(nnsim,df)
plot(x,y)
lines(x,thefit)
```

Plot with
nn fits.
Pretty good.



42

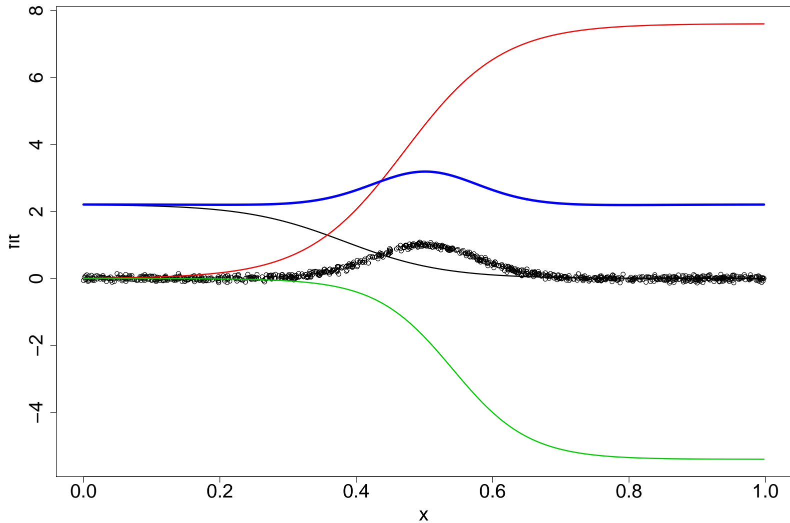
Here is the fitted model:

```
> summary(nnsim)
a 1-3-1 network with 10 weights
options were - linear output units  decay=0.0002441406
 b->h1 i1->h1
  5.26 -13.74
 b->h2 i1->h2
 -6.58  13.98
 b->h3 i1->h3
 -9.67  17.87
  b->o  h1->o  h2->o  h3->o
 -2.20   2.21   7.61  -5.40
```

Add up the pieces:

```
F = function(x) {return(exp(x)/(1+exp(x)))}

z1 =  5.26 - 13.74*x
z2 = -6.58 + 13.98*x
z3 = -9.67 + 17.87

f1 = 2.21*F(z1)
f2 = 7.61*F(z2)
f3 = -5.40*F(z3)
```

```
rx=range(x)
ry = range(c(f1,f2,f3,y))
plot(rx,ry,type="n",xlab="x",ylab="fit",cex.axis=2,cex.lab=2)
points(x,y)
lines(x,f1,col=1,lwd=2)
lines(x,f2,col=2,lwd=2)
lines(x,f3,col=3,lwd=2)
lines(x,f1+f2+f3,col=4,lwd=4)
```

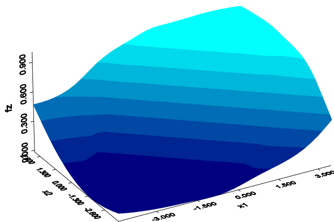


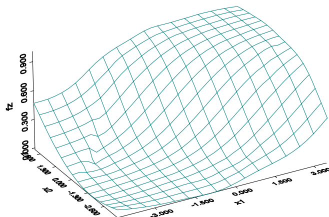
Awesome !!!

With more than one x it is a little harder to see how this works.

For each z, we get “ridge functions”.

F(x1+x2)





Elements of Statistical Learning page 394:

$$\begin{aligned} Z_m &= \sigma(\alpha_{0m} + \alpha_m^T X), m = 1, 2, \ldots, M \\ T_k &= \beta_{0k} + \beta_k^T Z, k = 1, 2, \ldots, K \\ \hat{y}_k(X) \equiv f_k(X) &= g_k(T), k = 1, 2, \ldots, K \end{aligned}$$

e.g. $K = 1$,$g(T) = T$ for the single linear output case.

Finally, we note that the name "neural networks" derives from the fact that they were first developed as models for the human brain. Each unit represents a neuron, and the connections represent synapses. In early models, the neurons fired when the total signal passed to that unit exceeded a certain threshold. In the model above, this corresponds to the use of a step function for $\sigma(Z)$ and $g_m(T)$. Later the neural network was recognized as a useful tool for nonlinear statistical modeling, and for this purpose the step function is not smooth enough for optimization. Hence the step function was replaced by a smoother threshold function, the sigmoid.

5. Neural Nets for Binary Y

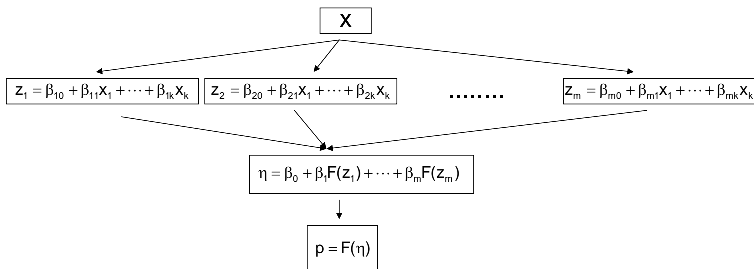
We have to adapt our model for the case of binary y.

What does it mean to predict a binary y?

As usual, we want $P(Y = 1|x)$.

The General Model

A k-m-1 network.
(x is numeric and k dimensional)



Everything is the same, except, at the end we apply the logistic function.
This gives us a number between 0 and 1, which will be our probability.

Beer Data

Data is from an mba class.
Ask each student for nbeer(#beers can drink)
weight, height, age, gender.

```
> beer = read.table("nbeer.txt",header=T)
> beer$gender = as.factor(beer$gender)
```

```
> summary(beer)
   nbeer            weight          height            age           gender
 Min.    : 0.00   Min.    :100.0   Min.    :60.00   Min.    :21.00   0:41
 1st Qu.: 5.00   1st Qu.:136.3   1st Qu.:66.63   1st Qu.:25.00   1: 9
 Median : 7.75   Median :160.0   Median :70.00   Median :26.66
 Mean    : 7.45   Mean    :155.8   Mean    :69.00   Mean    :27.18
 3rd Qu.:10.00   3rd Qu.:170.0   3rd Qu.:71.00   3rd Qu.:28.00
 Max.    :20.00   Max.    :230.0   Max.    :76.00   Max.    :40.00
```

As a simple (and somewhat silly) example,
we ask: given the # of beers what is the prob of a female ?

First, rescale the numeric x and then,

to fit the nnet,

no linout=T

```
> beer$nbeer = (beer$nbeer-min(beer$nbeer))/(max(beer$nbeer)-min(beer$nbeer))
> nnbeer = nnet(gender~nbeer,beer,size=5,decay=.01,maxit=1000)
```

The basic structure of the model is the same as with numeric y.

```
> summary(nnbeer)
a 1-5-1 network with 16 weights
options were - entropy fitting  decay=0.01
 b->h1 i1->h1
 -3.24   7.26
 b->h2 i1->h2
  0.01  -0.06
 b->h3 i1->h3
  0.01  -0.06
 b->h4 i1->h4
  0.01  -0.06
 b->h5 i1->h5
  0.01  -0.06
 b->o h1->o h2->o h3->o h4->o h5->o
 0.19 -6.71  0.13  0.13  0.13  0.13
```

Now let's think about fitting/predicting:

```
> pbeer = predict(nnbeer,beer)
> length(pbeer)
[1] 50
> is.array(pbeer)
[1] TRUE
> dim(pbeer)
[1] 50  1
> mode(pbeer)
[1] "numeric"
```

The predict command has its usual format, but we have to be careful about the format of the results.

What does it give us !!??

The predict command returns a 50 x 1 array.
In general, it would be n x1, where n is the sample size.

Since the fits are are n by 1 array we can just index the row or index the row and the first column.

```
> pbeer[1:5]
         1          2          3          4          5
0.00982357 0.00982357 0.29811830 0.29811830 0.14363053
> pbeer[1:5,1]
         1          2          3          4          5
0.00982357 0.00982357 0.29811830 0.29811830 0.14363053
```

The fit is the prob of the second level, which in this case is "being female".

```
> beer$gender[1:5]
[1] 0 0 0 0 0
Levels: 0 1
```

The first three observations are guys.

```
> beer$nbeer[1:5]
[1] 0.60 0.60 0.25 0.25 0.35
```

The third guy is only .25 up the number of beers scale, so he has a .298 choice of being female

```
> pbeer[1:5]
          1           2           3           4           5 
0.009810964 0.009810964 0.298126896 0.298126896 0.143612251
```

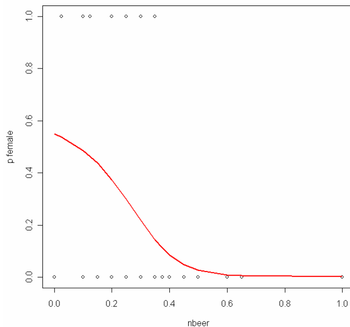
53

Since, we have only one x, we can plot the fits.

```
plot(beer$nbeer,as.numeric(beer$gender)-1,xlab="nbeer",ylab="p female")
oo = order(beer$nbeer)
lines(beer$nbeer[oo],pbeer[oo],col=2,lwd=2)
```

Red line is the neural net fit.

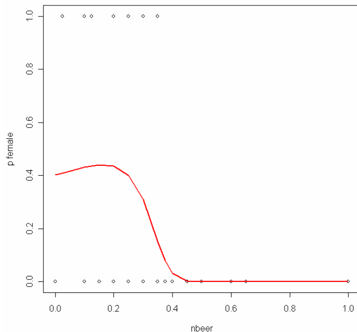
For each x (# of beers) gives Pr(female | x).



Do it again, but change size and decay:

```
> nnbeer = nnet(gender~nbeer,beer,size=10,decay=.001,maxit=1000)
```

Does not make sense in that we do not expect that first part where P(female|nbeer) increases.



55

Tabloid Data

```
> tab = read.table("tabdat9n20.txt",header=T)
> tab$purchase = as.factor(tab$purchase)
>
> tab =  tab[,1:3]
> tab$nTab = tab$nTab/81
> tab$moCbook = tab$moCbook/50
```

read the data in, set purchase to be a factor

Let's just use 2 = nTab and 3 = moCbook. As always, we have to rescale them.

```
> summary(tab)
 purchase        nTab               moCbook
 0:19509   Min.   :0.00000   Min.   :0.02365
 1:   491  1st Qu.:0.00000   1st Qu.:1.00000
           Median :0.00000   Median :1.00000
           Mean   :0.02263   Mean   :0.95201
           3rd Qu.:0.02469   3rd Qu.:1.00000
           Max.   :1.00000   Max.   :1.00000
```

Let's do this one more seriously and look at out of sample performance.

We use sets1 and 2 to fit the model, including choice of decay.

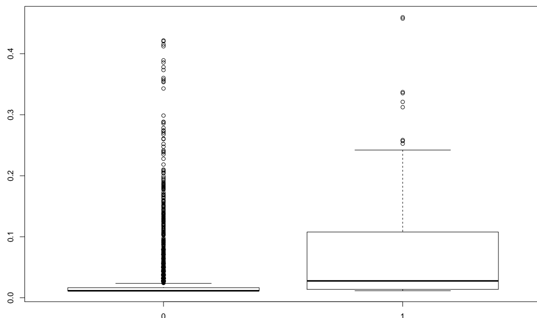
We use set three to assess out of sample performance.

```
nob = nrow(tab)
n1 = floor(.5*nob)
n2 = floor(.25*nob)
n3 = nob - n1 - n2
set.seed(19)
perm = sample(1:nob,nob)
set1 = tab[perm[1:n1],]
set2 = tab[perm[(n1+1):(n1+n2)],]
set3 = tab[perm[(n1+n2+1):nob],]
```

The usual three set stuff

Let's try a quick fit and see if we get anything.

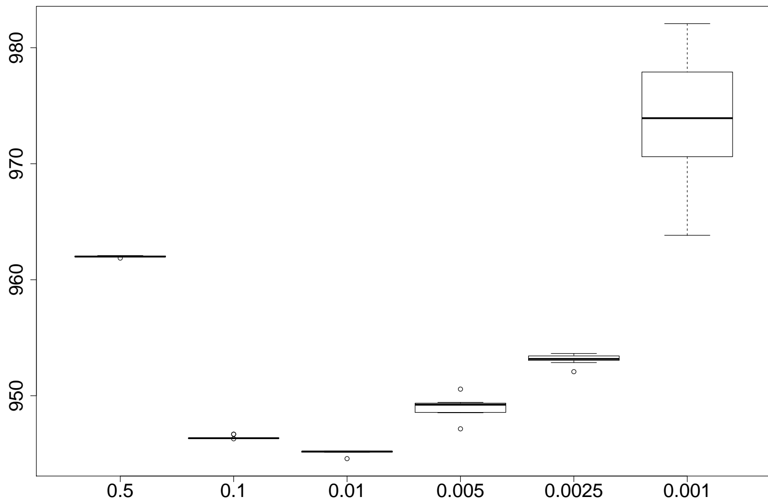
```
set.seed(99)
tempnn = nnet(purchase~.,set1,size=20,decay=.1,maxit=10000)
nnout = predict(tempnn,set2)[,1]
boxplot(nnout~set2$purchase)
```



looks promising !!!

```
##let's try a bunch of decay valuse
#This takes a while !!
#Im fitting length(decv)*nstart neural nets
#each with 10,000 observations.

if(file.exists("lossl.RData")) {
   cat("******reading in loosl from file lossl.RData\n")
   load("lossl.RData")
} else {
   cat("******running loop to compute lossl\n")
   decv = c(.5,.1,.01,.005,.0025,.001)
   nstart = 10
   lossl = list()

   set.seed(99)
   for(i in 1:length(decv)) {
      temploss<-rep(0,nstart)
      for(j in 1:nstart){
         cat("on dev: ",i," and start: ",j,"\n")
         tempnn = nnet(purchase~.,set1,size=20,decay=decv[i],maxit=10000)
         nnout = predict(tempnn,set2)[,1]
         temploss[j] = loss(set2$purchase,nnout,wht=.0001)
      }
      lossl[[i]] = temploss
   }
   save(lossl,decv,nstart,file="lossl.RData")
}
```

```
names(lossl) = as.character(decv)
boxplot(lossl,cex.lab=2,cex.axis=2)
```

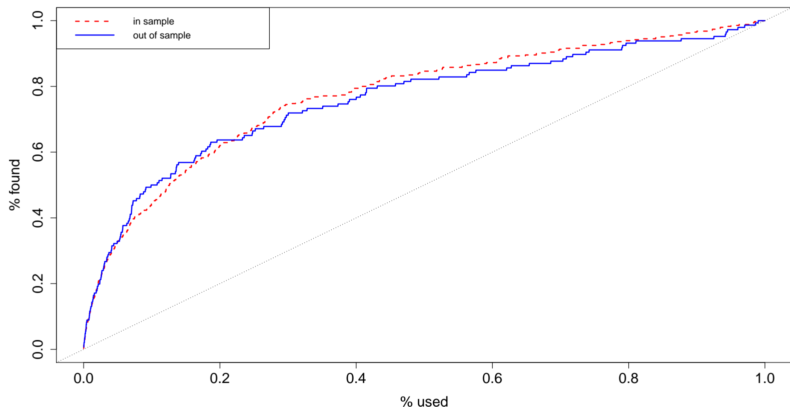


Ok, let's use decay = .01 and refit on sets 1 and 2 combined, and then predict on set 3.

```
set12 = rbind(set1,set2)
nns12 = nnet(purchase~.,set12,size=20,decay =.01,maxit=10000)

nnfit12 = predict(nns12,set12)[,1]
nnfit3 = predict(nns12,set3)[,1]
```

Now let's plot the lift, in and out of sample.

```
par(mfrow=c(1,1))
sy12 = liftf(set12$purchase,nnfit12,dopl=FALSE)
ii12 = (1:length(sy12))/length(sy12)
sy3 = liftf(set3$purchase,nnfit3,dopl=FALSE)
ii3 = (1:length(sy3))/length(sy3)
plot(c(0,1),c(0,1),type="n",xlab="% used",ylab="% found",
               cex.axis=1.5,cex.lab=1.5)
lines(ii12,sy12,col="red",lty=2,lwd=2)
lines(ii3,sy3,col="blue",lty=3,lwd=2)
abline(0,1,lty=3)
legend("topleft",legend=c("in sample","out of sample"),
           col=c("red","blue"),lwd=c(2,2),lty=c(2,3))
```



It's fun when it works !!

6. How Does it Work Again, XOR

Let's look again at how a neural net works by playing around with the famous XOR example.

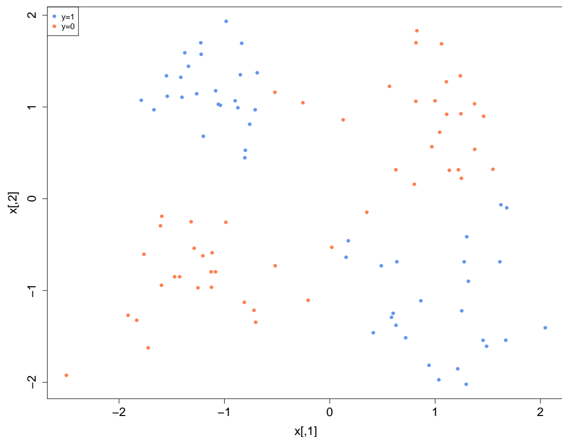
This is example is famous because it is a simple example where linear classification:

$$y = 1 \text{ if } a + b_1 x_1 + b_2 x_2 > 0$$

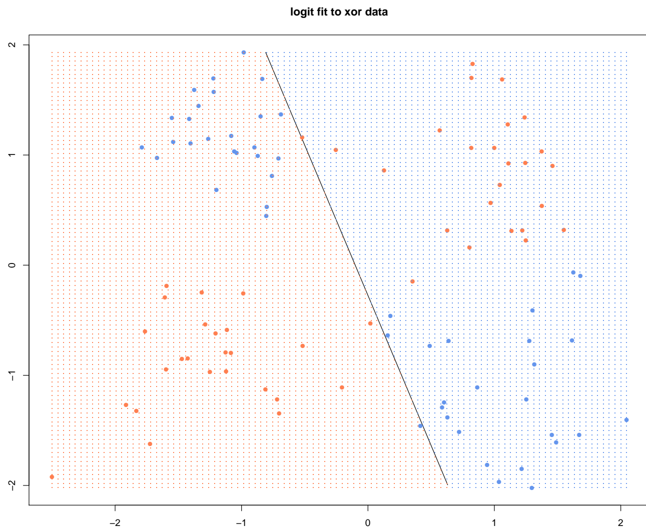
cannot work.

Basically, y is 1 if the $sign(x_1) \neq sign(x_2)$ but I added noise so a few points cross the boundaries.

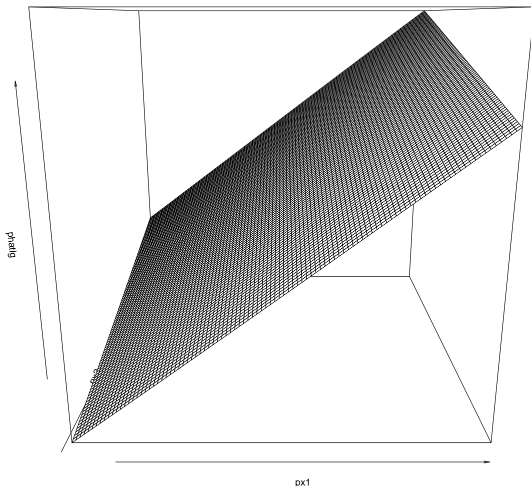
Here is a plot of the (simulated) data.



Here is the decision boundary ($\hat{y} = 1$ if $\hat{p} > .5$) for a linear logit fit.



Here is a plot of $\hat{p}(x_1, x_2)$ from the logit fit.



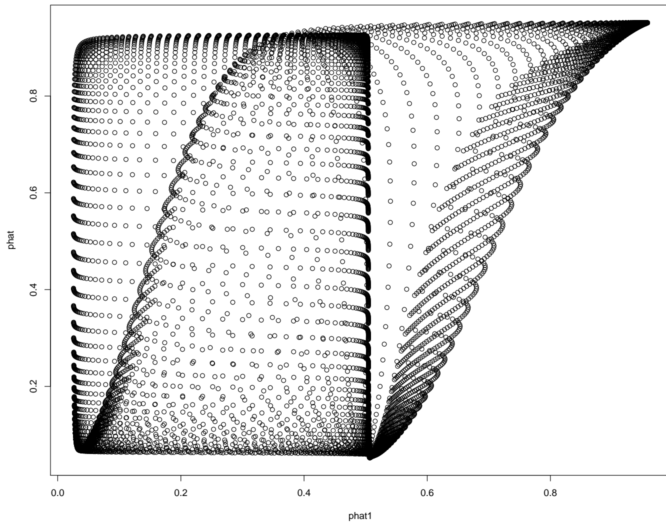
Really all the $\hat{p}$ are close to .5 !!

```
> print(summary(lgfit))

Call:
glm(formula = y ~ ., family = binomial, data = dfd)

Deviance Residuals:
     Min        1Q    Median        3Q       Max
-1.25921  -1.17512   0.02788   1.17894   1.23320

Coefficients:
            Estimate Std. Error z value Pr(>|z|)
(Intercept)  0.01013    0.20113   0.050    0.960
x1           0.10058    0.17129   0.587    0.557
x2           0.03688    0.18028   0.205    0.838

(Dispersion parameter for binomial family taken to be 1)

    Null deviance: 138.63  on 99  degrees of freedom
Residual deviance: 138.27  on 97  degrees of freedom
AIC: 144.27

Number of Fisher Scoring iterations: 3

> summary(phatl)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
 0.4217  0.4676  0.4964  0.4964  0.5253  0.5713
```

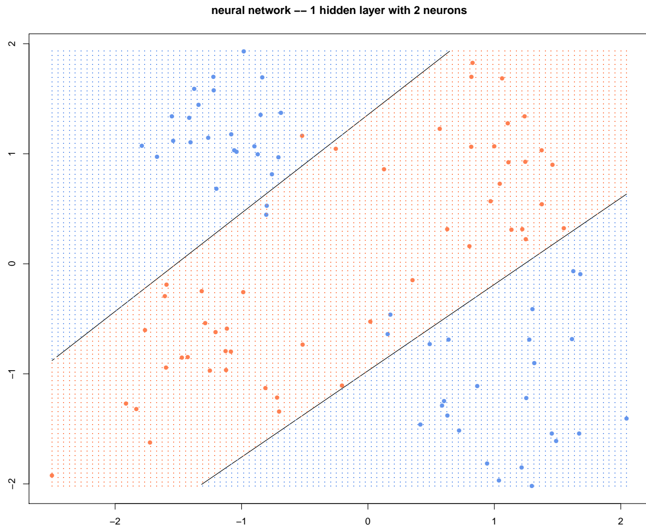
Let's try a nn fit.

```
#uses random starting values for iterative optimization
set.seed(99) #misses
xnn = nnet(y~.,dfd,size=2,decay=.1)
phat1 = predict(xnn,gd)[,1]

set.seed(14) #works
xnn = nnet(y~.,dfd,size=2,decay=.1)
phat = predict(xnn,gd)[,1]

#plot fits, far out!!
plot(phat1,phat)
```

Far out.

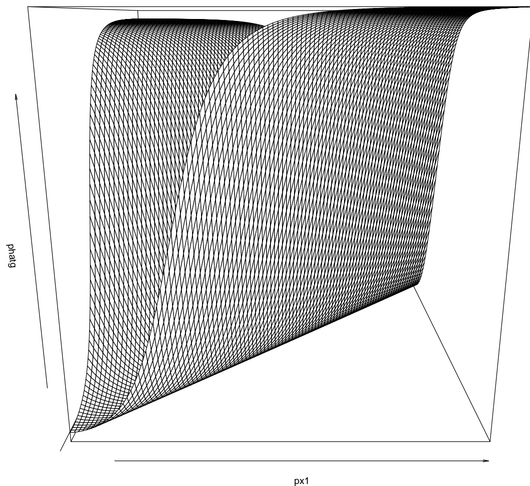


Here is the nn decision boundary (from the one that worked).



Beautiful !!!

Here is a plot of $\hat{p}(x_1, x_2)$ from the nn fit.



Obvious !!!!????

(see plot3d in xor.R).

```
> summary(xnn)
a 2-2-1 network with 9 weights
options were - entropy fitting  decay=0.1
 b->h1 i1->h1 i2->h1
  3.35   2.38  -2.66
 b->h2 i1->h2 i2->h2
 -2.73   2.28  -2.90
 b->o h1->o h2->o
 2.54 -5.84  6.30
```

Basically uses $x_1 - x_2$!!!!.

A plot of `xnn`: