Introduction to Neural Networks

1 Introduction

In this lab we are going to have a look at some very basic neural networks on a new data set which relates various covariates about cheese samples to a taste response.

2 Cheese Data Set

2.1 Getting Setup

Exercise: Download the cheese data from eLearning.

Exercise: Load the cheese data into a variable called `cheese` in your workspace.

```
>
```

Exercise: Load the `nnet` package, which contains the functions to build neural networks in `R`.

2.2 Explore the Data

Since we're looking at a new data set it's good practice to get a feel for what we're looking at.

```
> cheese
> names(cheese)
> summary(cheese)
```

These two commands tell us respectively the names of the variables and some summary statistics about the data contained therein.

Exercise: Do a box plot and a pairs plot of all the data.

```
>
>
```

Exercise: To reinforce the new programming learned in the last lab and lecture, adapt the for-loop code from lab 4.

```
> par(mfrow=c(2, 2))
>
```

2.3 Train, Validate and Test Subsets

You might expect that we would next split the data into cross-validation sets. However, one of the first things that probably struck you about the data when looking at it is how few observations there are (only 30). Consequently, it is questionable whether it would be sensible to split the data three ways to get train, validate and test sets: we would end up with so little data in each set that our analysis would lack any power.

In lectures you will see/have seen two possible options for assessing models without splitting the data: the Akaike Information Criterion (AIC) and Schwarz Bayesian Information Criterion (SBIC). Under the assumption of normally distributed errors, these can be written:

$$\text{AIC} = 2k + n \log\left(\tfrac{\text{SSE}}{n}\right)$$

$$\text{SBIC} = k \log n + n \log\left(\tfrac{\text{SSE}}{n}\right)$$

where k is the number of parameters being estimated, n is the number of observations in the full data set and $\text{SSE} = \sum \varepsilon_i^2 = \sum (\hat{y}_i - y_i)^2$. We will use these instead of train, test and validate splits for this small data set.

3 Neural Networks

3.1 Simple Neural Net, Linear Activation Fn, No Hidden Layer

3.1.1 Fitting the Neural Net

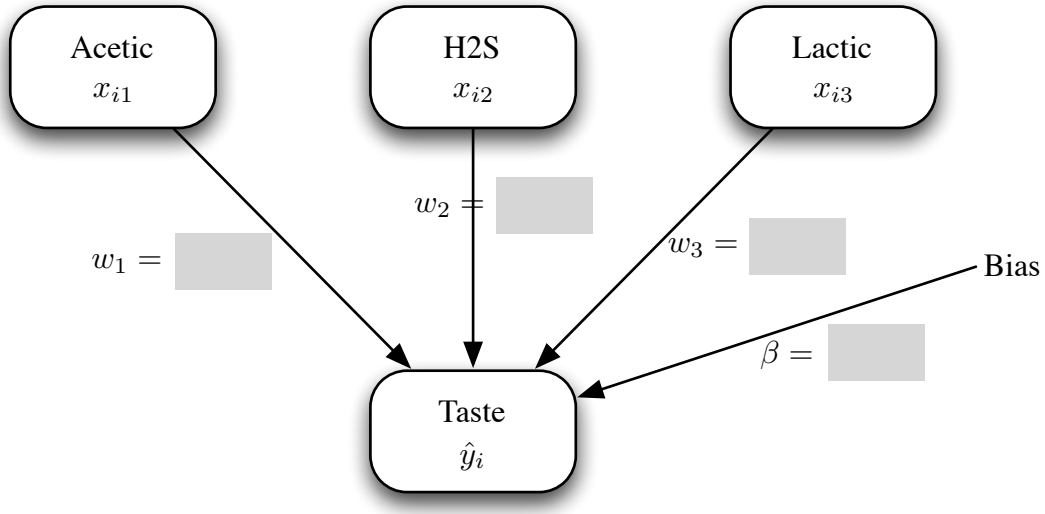
We are going to first fit the simplest possible neural network to the cheese data, to predict taste from acetic, H2S and lactic. This is the neural network with the input layer directly connected to the output.

```
> fitnn1 = nnet(taste ~ Acetic + H2S + Lactic, cheese, size=0,
                skip=TRUE, linout=TRUE)
> summary(fitnn1)
```

The `size` argument specifies how many nodes to have in the hidden layer, `skip` indicates that the input layer has a direct connection to the output layer and `linout` specifies the simple identity activation function.

The output from summary gives us the detail of the neural network. `i1`, `i2` and `i3` are the input nodes (so acetic, H2S and lactic respectively); `o` is the output node (so taste); and `b` is the bias.

Exercise: Fill in the shaded boxes on the neural network diagram below with the values which `R` has fitted.



$$\implies \hat{y}_i = \beta + w_1 x_{i1} + w_2 x_{i2} + w_3 x_{i3}$$

$$\implies \hat{y}_i =$$

We can compare this to the fit we get by doing standard least squares regression:

```
> lm(taste ~ Acetic + H2S + Lactic, cheese)
```

Exercise: Upto a reasonable level of accuracy, do you see any difference between the linear model and neural network fit?

3.1.2 Model Evaluation

We can now compute the AIC for this fitted model quite simply:

```
> SSE1 = sum(fitnn1$residuals^2)
> AIC1 = 2*5 + 30*log(SSE1/30)
> AIC1
```

Exercise: Using the SSE already calculated in `SSE1`, compute the SBIC and store it in a variable called `SBIC1`.

```
>
>
```

A scientist working on the project may believe there is reason to query whether acetic is a good linear predictor for taste. To test this conjecture, we can refit a linear model without acetic in and compare the AIC and SBIC of the two models.

```
> fitnn2 = nnet(taste ~ H2S + Lactic, cheese, size=0, skip=TRUE,
                linout=TRUE)
> summary(fitnn2)
> SSE2 = sum(fitnn2$residuals^2)
> AIC2 = 2*4 + 30*log(SSE2/30)
> AIC2
> AIC2 < AIC1
```

Exercise: On the basis of the above analysis, what is your advice to the scientist? Should the model with or without acetic be preferred?

3.2 Adding a Hidden Layer With a Single Node

We now add a hidden layer in between the input and output neurons, with a single node.

```
> fitnn3 = nnet(taste ~ Acetic + H2S + Lactic, cheese, size=1,
                linout=TRUE)
```

You may (or may not) see output with numbers of similar magnitude to the following:

```
# weights:  6
initial  value 25372.359476
final  value 7662.886667
converged
```

The numbers being printed here are the SSE $= \sum \varepsilon_i^2 = \sum (\hat{y}_i - y_i)^2$ for the fit on the current iteration of the neural network fitting algorithm. If you look back to the previous `nnet()` calls, you should see that the final SSE ≈ 2700. It seems very strange that the SSE has increased so dramatically when we are *increasing* the model complexity! If you keep rerunning the last command, you should find that sometimes you get SSE < 2700 and other times not (try this until you see it change). This is because the algorithm chooses random starting weights for the neural net and for some choices is getting stuck in a local minima. The writers of `nnet()` have designed the algorithm to be more robust to this kind of issue when scaled data is used ($\bar{x}_i = 0, s_{x_i}^2 = 1$).

So, we'll scale the data and refit:

```
> cheese2 = scale(cheese)
> fitnn3 = nnet(taste ~ Acetic + H2S + Lactic, cheese2, size=1,
                linout=TRUE)
> summary(fitnn3)
```

Aside: *Note that we could not now compare an AIC/SBIC from this model to the earlier ones because the data were on different scales which affects the SSE. We would have to refit the earlier model on the scaled data to do a comparison.*

Exercise: Use the space below to draw the full neural network we have fitted. Make sure you show all appropriate connections; all the weights for every connection; and write the full model equation after the diagram.

4 Neural Networks for Classification

4.1 Getting Setup

Exercise: Download the Titanic data from the course website if you don't already have a local copy saved.

Exercise: Load the titanic data into a variable called `data` in your workspace.

```
>
```

Exercise: Load the `nnet` package, which contains the functions to build neural networks in `R`.

```
>
```

4.2 Train and Test Subsets

We are back in the nice situation of having plenty of data, so we can do the usual train and test splits in the data for model checking.
Exercise: Split the data in `data` into two variables, one called `train` and another called `test`. The split should be $\frac{1}{3}$ for the testing data and $\frac{2}{3}$ for the training data.

```
>
>
>
```

Exercise: Confirm that we did not get a freakish random split of the Survived variable. In other words, check that there are a decent number of survivors and non-survivors in both `train` and `test`. (Hint: see lab 4, §2.3 if you've forgotten)

```
>
>
```

4.3 Fitting the Neural Network

The `nnet` package has a slightly strange requirement that the target variable of the classification (ie Survived) be in a particular format. At the moment, if you look at `data$Survived` you will see it is a vector of `No`/`Yes` values, but when using `nnet()` for classification we must instead provide a two-column matrix — one column for `No` and the other for `Yes` — with a `1`/`0` as appropriate. In other words, we need to convert:

$$\begin{pmatrix} \texttt{Yes} \\ \texttt{Yes} \\ \texttt{No} \\ \texttt{Yes} \\ \texttt{No} \\ \texttt{No} \\ \vdots \end{pmatrix} \rightarrow \begin{pmatrix} 0 & 1 \\ 0 & 1 \\ 1 & 0 \\ 0 & 1 \\ 1 & 0 \\ 1 & 0 \\ \vdots & \vdots \end{pmatrix}$$

Fortunately there is an easy to use utility function `class.ind` to do this for us built-in to the package:

```
> train$Surv = class.ind(train$Survived)
> test$Surv = class.ind(test$Survived)
```

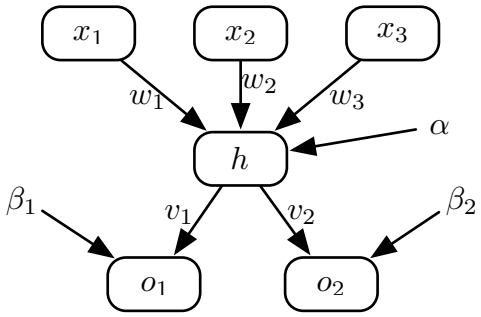
Now we are ready to fit a neural network for classification purposes:

```
> fitnn = nnet(Surv~Sex+Age+Class, train, size=1, softmax=TRUE)
> fitnn
> summary(fitnn)
```

Exercise: Draw the neural network below. Remember to label each node carefully (particularly pay attention to the fact we have categorical nodes here) and label every arc with the weights.

4.4 Calculating Probabilities

Calculating the probabilities from the neural network can seem a bit difficult, so we recap how the `nnet` function is doing it (which might be subtly different from lectures, but you should see the high-level similarity). Consider the following example of a classification neural network:



Then,

$$\mathbb{P}(\text{category 1}) = \frac{e^{o_1}}{e^{o_1} + e^{o_2}} \qquad \text{and} \qquad \mathbb{P}(\text{category 2}) = \frac{e^{o_2}}{e^{o_1} + e^{o_2}}$$

where

$$o_1 = v_1 h + \beta_1 \qquad , \qquad o_2 = v_2 h + \beta_2$$

and

$$h = \frac{e^{\alpha + \sum w_i x_i}}{1 + e^{\alpha + \sum w_i x_i}}$$

Exercise: Using the neural network you drew above, manually calculate the probability that an adult female passenger in first class does not survive.

4.5 Neural Network Performance

We can now look at the overall performance of the neural network by looking at a table of how predictions using the test set fare:

```
> table(data.frame(predicted=predict(fitnn, test)[,2] > 0.5,
                   actual=test$Surv[,2]>0.5))
```

How do you think it is doing? In reality the answer to this question will have an element of randomness in it due to different people's train/test splits, because once again the algorithm is not entirely stable.

4.6 Adding Decay

You have discussed in lectures how decay can help to improve the stability of neural networks. The first thing we can do is scale data. See above (Section 3.2) for how to do this. The `nnet` function accepts an argument called decay which implements this automatically for you (ie we can put, for example, `decay=0.02` in the `nnet()` function call). We can look at how decay affects the sum of squared errors by calculating various decay values upto 1 in a for loop.

Exercise: Fill in the missing parts of the following code and so compute the sum of squared error over a range of values between 0.0001 and 1 which is then plotted.

```
err = vector("numeric", 100)
d = seq(0.0001, 1, length.out=100)
for(i in ???) {
    fit = ???
    err[i] = sum(???)
}
plot(d, err)
```

NB: If you get obvious outliers in the y-axis direction, rerun your code until you get something resembling a curve.

4.7 The Hessian

Remember at lectures we looked at the Hessian matrix as an indicator of the stability of the solution,. To obtain and print the Hessian matrix we can do the following:

```
> fitnn = nnet(Surv~Sex+Age+Class, train, size=1, softmax=TRUE, Hess=TRUE)
> fitnn$Hess
```

The function `eigen()` in `R` can be used to find the eigenvectors and eigenvalues of a matrix. Remember that the Hessian matrix should be positive definite (all eigenvalues greater than 0).

```
> eigen(fitnn$Hess)
```

5 Comparing Neural Networks and Trees

5.1 A Disadvantage of Neural Networks

Neural networks do not, by default, pick up any interactions between variables in the data. However, you can add variable interactions manually. So, say we imagine that there is interaction between Age and Class in the Titanic data, we simply modify our `nnet()` call to:

```
> fitnn = nnet(Surv~Sex+Age+Class+Age*Class, train, size=1, softmax=TRUE)
> fitnn
> summary(fitnn)
> table(data.frame(predicted=predict(fitnn, test)[,2] > 0.5,
                   actual=test$Surv[,2]>0.5))
```

5.2 A Longer Exercise ...

Exercise: If you have time, go back to the previous labs and remember how to fit a tree to the `train` data. Do so and compare the confusion matrix to the one you obtained just now from the neural network. Which performed better without extensive tuning?