HW3

Norman Hong

February 21, 2019

Number 28 is at the end of this document.

Extra 30

Suppose two different ANNs have been trained on a training set for a classification problem, and the responses, scaled to have values in [0,1], have been computed for all training instances for both networks. Assume that the responses for network 2 are related to those from network 1 by a monotone function, such as in the plot in the hw pdf document. Explain carefully why the two ANNs have the same ROC curve.

A monotone function $f : x \to y$ between two topological spaces satisfies the properties for homomorphism because f is a bijection, f is continuous, and the inverse function of f is also continuous. This means that the predicted responses for the two different ANN models have the same topological properties, which implies that the ROC curve are the same. Since the topological properties are preserved, the two models must have the same decision boundary. Also, each true negatives and true positives in model 1 can be mapped to each true positives and true negatives in model 2. This implies that there is no change in the number of true positives and true negatives. Therefore the ROC curve should be the same.

Extra 34

Make a dataframe with $k = 11$ columns and $N = 100$ observations, where all entries are independent standard normal random sample. Let z be the last column. Use set.seed(20305).

```
set.seed(20305)
values <- rnorm(1100,mean=0,sd=1)
data <- data.frame(matrix(values, nrow=100))
names(data)[11] <- 'z'
```

(a)

Fit z to the other 10 columns using multiple regression. What is the sum of squares of the residuals?

The sum of squares residuals is 98.8. None of the estimated coefficients are statistically significant. The F-statistic is also not statistically significant because it is less than 1.

```
lin.fit <- lm(z~., data=data)
summary(lin.fit)
```

```
##
## Call:
## lm(formula = z ~ ., data = data)
##
## Residuals:
##      Min       1Q   Median       3Q      Max
## -2.35816 -0.67302  0.04407  0.74991  2.70681
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept) -0.17797    0.11162  -1.594   0.1144
## X1           0.20702    0.11184   1.851   0.0675 .
## X2          -0.02103    0.12071  -0.174   0.8621
## X3           0.08535    0.11437   0.746   0.4575
## X4           0.03585    0.10940   0.328   0.7439
## X5          -0.01923    0.10478  -0.184   0.8548
## X6           0.13443    0.10667   1.260   0.2108
## X7           0.02458    0.11090   0.222   0.8251
## X8          -0.03366    0.11685  -0.288   0.7740
## X9          -0.15123    0.10347  -1.462   0.1474
## X10         -0.08584    0.10481  -0.819   0.4150
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 1.054 on 89 degrees of freedom
## Multiple R-squared:  0.09073,    Adjusted R-squared:  -0.01143
## F-statistic: 0.8881 on 10 and 89 DF,  p-value: 0.5475
```

```r
sum(residuals(lin.fit)**2)
```

```
## [1] 98.80704
```

(b)

Fit z to the other 10 columns using a neural network with 2 hidden units and setting maxit=2000 and decay=.01. Does this model fit the data better? How do you know?

It doesn't make sense to compare cross entropy measure to sum of squared residuals. Therefore, I should calculate the sum of squared residuals for this model and compare it to the multiple regression moddel. The sum of squares residuals is 91.45. The sum of squared residuals decreased from 98.80 to 91.45, which means that this model fits the data better than the multiple regression model.

```r
set.seed(20305)
nn1 <- nnet(z~., data=data, size=2, maxit=2000, decay=.01)
```

```
## # weights:  25
## initial  value 159.852387
## iter  10 value 111.418145
## iter  20 value 110.563089
## iter  30 value 100.365861
## iter  40 value 97.048956
## iter  50 value 95.146648
## iter  60 value 94.745303
## iter  70 value 94.472068
## iter  80 value 94.429607
## iter  90 value 94.423817
## iter 100 value 94.423505
## final  value 94.423501
## converged
```

```r
pred <- predict(nn1, data, type='raw')
sum((data$z-pred)**2)
```

```
## [1] 91.45336
```

(c)

Redo this experiment with the same data and with 5 and 10 hidden units and explain what you see.

I notice that the cross entropy is at a lower value as the number of hidden units increase. This means that the model is better optimized. The sum of squared residuals decreased as the hidden units increased, showing the ANN model with more hidden units fit the data better.

```
set.seed(20305)
nn1 <- nnet(z~., data=data, size=5, maxit=2000, decay=.01)
```

```
## # weights:  61
## initial  value 159.028740
## iter  10 value 111.959009
## iter  20 value 110.636584
## iter  30 value 101.864955
## iter  40 value 97.805681
## iter  50 value 93.545945
## iter  60 value 91.916157
## iter  70 value 91.663413
## iter  80 value 91.586132
## iter  90 value 91.572110
## iter 100 value 91.534036
## iter 110 value 91.516110
## iter 120 value 91.513886
## iter 130 value 91.511979
## iter 140 value 91.509896
## iter 150 value 91.509260
## iter 160 value 91.509170
## iter 170 value 91.509150
## final  value 91.509145
## converged
```

```
pred <- predict(nn1, data, type='raw')
sum((data$z-pred)**2)
```

```
## [1] 88.74601
```

```
nn1 <- nnet(z~., data=data, size=10, maxit=2000, decay=.01)
```

```
## # weights:  121
## initial  value 198.957615
## iter  10 value 111.670189
## iter  20 value 110.477734
## iter  30 value 101.076277
## iter  40 value 99.283662
## iter  50 value 98.212682
## iter  60 value 97.146123
## iter  70 value 96.953896
## iter  80 value 95.192540
## iter  90 value 92.044269
## iter 100 value 91.686650
## iter 110 value 91.572723
## iter 120 value 91.424354
## iter 130 value 91.255804
## iter 140 value 91.169590
## iter 150 value 91.102919
## iter 160 value 91.056192
## iter 170 value 91.029237
## iter 180 value 91.009618
## iter 190 value 90.018524
## iter 200 value 87.654367
## iter 210 value 86.743681
## iter 220 value 86.299671
## iter 230 value 85.936706
## iter 240 value 85.597168
## iter 250 value 85.373113
## iter 260 value 85.302489
## iter 270 value 85.209043
## iter 280 value 84.607562
## iter 290 value 83.756667
## iter 300 value 82.483032
## iter 310 value 81.098546
## iter 320 value 80.913227
## iter 330 value 80.725396
## iter 340 value 80.522883
## iter 350 value 80.439355
## iter 360 value 80.397194
## iter 370 value 80.378981
## iter 380 value 80.342843
## iter 390 value 80.335223
## iter 400 value 80.308757
## iter 410 value 80.291721
## iter 420 value 80.275634
## iter 430 value 80.230392
## iter 440 value 80.176773
## iter 450 value 80.166552
## iter 460 value 80.165113
## iter 470 value 80.164932
## final  value 80.164926
## converged
```

```r
pred <- predict(nn1, data, type='raw')
sum((data$z-pred)**2)
```

```
## [1] 75.71666
```

Extra 33

We'll use the MINST image classification data, available at mnist_all.RData that were used in class during the last two weeks. We want to distinguish between 4 and 7. Extract the relevant training data and place them in a data frame.

```r
mnistTrainX <- train$x[(train$y == 4) | (train$y == 7),]
mnistTrainY <- train$y[train$y == 4 | train$y == 7]
# let class 1 represent number 7 and class 0 represent
# number 4.
mnistTrainY <- as.numeric(mnistTrainY == 7)
mnist <- data.frame(x=mnistTrainX, y=mnistTrainY)

mnistTestX <- test$x[(test$y == 4) | (test$y == 7),]
mnistTestY <- test$y[test$y == 4 | test$y == 7]
```

```r
# let class 1 represent number 7 and class 0 represent
# number 4.
mnistTestY <- as.numeric(mnistTestY == 7)
mnistTest <- data.frame(x=mnistTestX, y=mnistTestY)
```

(a)

Pick two features (variables) that have large variances and low correlation. Fit a logistic regression model with these two features. Evaluate the model with the AUC score.

variable 430 has highest variance. The variable with the lowest correlation with this variable is 266. The area under the curve for the training data is 0.946. The area under the curve for the test data is .9551. There does not seem to be any evidence of overfitting.

```r
# Determine highest variance variables.
vars <- apply(mnistTrainX, MARGIN=2, var)
sortedHighVar <- sort(vars, decreasing=TRUE, index.return=TRUE)
sortedHighVar$ix[1:20]
```

```
##  [1] 430 402 240 241 239 267 266 401 431 268 374 238 429 574 269 346 242
## [18] 602 262 373
```

```r
cor(mnistTrainX[, sortedHighVar$ix[1:20]])
```

```
##             [,1]       [,2]       [,3]       [,4]       [,5]       [,6]
##  [1,]  1.0000000  0.8319160 -0.4865487 -0.4315866 -0.4595349 -0.4494881
##  [2,]  0.8319160  1.0000000 -0.4655716 -0.4178829 -0.4355189 -0.4106246
##  [3,] -0.4865487 -0.4655716  1.0000000  0.8452650  0.8748572  0.6694308
##  [4,] -0.4315866 -0.4178829  0.8452650  1.0000000  0.6612988  0.5044696
##  [5,] -0.4595349 -0.4355189  0.8748572  0.6612988  1.0000000  0.6984342
##  [6,] -0.4494881 -0.4106246  0.6694308  0.5044696  0.6984342  1.0000000
##  [7,] -0.3709382 -0.3273713  0.5463082  0.3920093  0.6771763  0.8635413
##  [8,]  0.7114484  0.7730539 -0.4473480 -0.4067663 -0.4214086 -0.3820796
##  [9,]  0.8699569  0.7252710 -0.4762600 -0.4146700 -0.4514632 -0.4570005
## [10,] -0.4782857 -0.4407790  0.7184191  0.6291973  0.5975966  0.8688270
## [11,]  0.5904984  0.8303583 -0.4051429 -0.3663087 -0.3735466 -0.3335277
## [12,] -0.3854139 -0.3561813  0.6682881  0.4937367  0.8549177  0.5345247
## [13,]  0.8331437  0.7169042 -0.4612396 -0.4237148 -0.4299249 -0.4052212
## [14,] -0.1993871 -0.1573053  0.2159543  0.1710816  0.2639520  0.1946320
## [15,] -0.4239214 -0.3914963  0.6057085  0.7438772  0.4556655  0.6266692
## [16,]  0.3776278  0.5551994 -0.3375291 -0.3032091 -0.3231028 -0.2463192
## [17,] -0.3145999 -0.3006798  0.5580820  0.7754232  0.4325060  0.3177499
## [18,] -0.1903027 -0.1503496  0.2142006  0.1739563  0.2552604  0.1946303
## [19,] -0.2275021 -0.2471380  0.2028899  0.2112018  0.1499649  0.2186108
## [20,]  0.4896249  0.5915535 -0.3866558 -0.3450613 -0.3756134 -0.3162502
##             [,7]       [,8]       [,9]      [,10]       [,11]       [,12]
##  [1,] -0.3709382  0.7114484  0.8699569 -0.4782857  0.59049840 -0.38541390
##  [2,] -0.3273713  0.7730539  0.7252710 -0.4407790  0.83035831 -0.35618131
##  [3,]  0.5463082 -0.4473480 -0.4762600  0.7184191 -0.40514292  0.66828815
##  [4,]  0.3920093 -0.4067663 -0.4146700  0.6291973 -0.36630871  0.49373669
##  [5,]  0.6771763 -0.4214086 -0.4514632  0.5975966 -0.37354660  0.85491773
##  [6,]  0.8635413 -0.3820796 -0.4570005  0.8688270 -0.33352773  0.53452466
##  [7,]  1.0000000 -0.3252623 -0.3769705  0.6736813 -0.25088726  0.67239829
##  [8,] -0.3252623  1.0000000  0.5478086 -0.4108733  0.72665171 -0.36497449
##  [9,] -0.3769705  0.5478086  1.0000000 -0.4746833  0.49537137 -0.37716838
## [10,]  0.6736813 -0.4108733 -0.4746833  1.0000000 -0.35689554  0.43216293
## [11,] -0.2508873  0.7266517  0.4953714 -0.3568955  1.00000000 -0.29359876
## [12,]  0.6723983 -0.3649745 -0.3771684  0.4321629 -0.29359876  1.00000000
## [13,] -0.3388569  0.8584967  0.6315465 -0.4438358  0.56865223 -0.37128783
## [14,]  0.2442433 -0.1730377 -0.1821099  0.1387688 -0.09565478  0.29901690
## [15,]  0.4712565 -0.3776415 -0.4067043  0.8172005 -0.31668508  0.32968009
## [16,] -0.1974555  0.5911709  0.3041910 -0.2503633  0.81796332 -0.26485009
## [17,]  0.2445725 -0.3067894 -0.2957251  0.3761351 -0.26979527  0.31625786
## [18,]  0.2375173 -0.1618029 -0.1751256  0.1551939 -0.08875086  0.29776013
## [19,]  0.1271364 -0.1324226 -0.2269724  0.2650435 -0.22136398  0.08883622
## [20,] -0.2870658  0.8477477  0.3824701 -0.3245272  0.71168251 -0.33672211
##            [,13]       [,14]       [,15]       [,16]      [,17]
##  [1,]  0.8331437 -0.19938709 -0.42392136  0.37762781 -0.3145999
##  [2,]  0.7169042 -0.15730525 -0.39149627  0.55519943 -0.3006798
##  [3,] -0.4612396  0.21595434  0.60570851 -0.33752908  0.5580820
##  [4,] -0.4237148  0.17108163  0.74387720 -0.30320912  0.7754232
##  [5,] -0.4299249  0.26395200  0.45566548 -0.32310277  0.4325060
##  [6,] -0.4052212  0.19463196  0.62666923 -0.24631918  0.3177499
##  [7,] -0.3388569  0.24424327  0.47125651 -0.19745554  0.2445725
##  [8,]  0.8584967 -0.17303771 -0.37764153  0.59117085 -0.3067894
##  [9,]  0.6315465 -0.18210988 -0.40670426  0.30419105 -0.2957251
## [10,] -0.4438358  0.13876878  0.81720048 -0.25036334  0.3761351
## [11,]  0.5686522 -0.09565478 -0.31668508  0.81796332 -0.2697953
## [12,] -0.3712878  0.29901690  0.32968009 -0.26485009  0.3162579
## [13,]  1.0000000 -0.18950090 -0.41392432  0.41112850 -0.3255772
## [14,] -0.1895009  1.00000000  0.08884313 -0.05467683  0.1636068
## [15,] -0.4139243  0.08884313  1.00000000 -0.22210554  0.5743884
## [16,]  0.4111285 -0.05467683 -0.22210554  1.00000000 -0.2202220
## [17,] -0.3255772  0.16360680  0.57438844 -0.22022204  1.0000000
## [18,] -0.1806984  0.86488023  0.11573564 -0.03723094  0.1612746
## [19,] -0.1856695 -0.03758950  0.25680177 -0.07101419  0.1929299
## [20,]  0.6213510 -0.14850027 -0.29205494  0.72975242 -0.2546752
##             [,18]       [,19]       [,20]
##  [1,] -0.19030271 -0.22750209  0.48962488
##  [2,] -0.15034956 -0.24713800  0.59155350
##  [3,]  0.21420059  0.20288986 -0.38665575
##  [4,]  0.17395630  0.21120177 -0.34506125
##  [5,]  0.25526038  0.14996491 -0.37561341
##  [6,]  0.19463034  0.21861081 -0.31625020
##  [7,]  0.23751727  0.12713638 -0.28706585
##  [8,] -0.16180286 -0.13242258  0.84774769
##  [9,] -0.17512557 -0.22697239  0.38247015
## [10,]  0.15519391  0.26504349 -0.32452722
## [11,] -0.08875086 -0.22136398  0.71168251
## [12,]  0.29776013  0.08883622 -0.33672211
## [13,] -0.18069842 -0.18566947  0.62135096
## [14,]  0.86488023 -0.03758950 -0.14850027
## [15,]  0.11573564  0.25680177 -0.29205494
## [16,] -0.03723094 -0.07101419  0.72975242
## [17,]  0.16127456  0.19292986 -0.25467524
## [18,]  1.00000000  0.03427560 -0.13049144
## [19,]  0.03427560  1.00000000 -0.01269927
## [20,] -0.13049144 -0.01269927  1.00000000
```

```
# 430 and 266 have low correlation.
glm.fit <- glm(y~x.430+x.266, data=mnist, family=binomial)
summary(glm.fit)
```

```
##
## Call:
## glm(formula = y ~ x.430 + x.266, family = binomial, data = mnist)
##
## Deviance Residuals:
##     Min       1Q   Median       3Q      Max
## -2.5953  -0.2807   0.2663   0.3316   3.0051
##
## Coefficients:
##               Estimate Std. Error z value Pr(>|z|)
## (Intercept)  0.4044324  0.0429966   9.406   <2e-16 ***
## x.430       -0.0192498  0.0003447 -55.846   <2e-16 ***
## x.266        0.0114840  0.0003068  37.432   <2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## (Dispersion parameter for binomial family taken to be 1)
##
##     Null deviance: 16769.1  on 12106  degrees of freedom
## Residual deviance:  7241.2  on 12104  degrees of freedom
## AIC: 7247.2
##
## Number of Fisher Scoring iterations: 6
```

```
pred <- predict(glm.fit, mnist, type='response')
auc(mnist$y, pred)
```

```
## Area under the curve: 0.946
```

```
predTest <- predict(glm.fit, mnistTest, type='response')
auc(mnistTest$y, predTest)
```

```
## Area under the curve: 0.9551
```

(b)

Create a neural net with 1 unit in the hidden layer. Train the neural net with the same two features as the previous part and evaluate the model with AUC. Compare the results from (a) and explain.

The AUC score for the training and test is about the same as the AUC score from the logistic model. This makes sense because the nnet package creates single layer neural networks. nnet also uses a logistic function as its activation function. When we use a single node and single hidden layer, only a single logistic transformation is occuring on the linear combination on the sum of the inputs. Then the output layer does another logistic transformation on the output from the hidden layer, which means the output behaves like a logistic regression. The output from the hidden layer should be a single value for each observation.

```
set.seed(20305)
nn1 <- nnet(y~x.430+x.266, data=mnist, size=1, decay=.1)
```

```
## # weights:  5
## initial  value 3020.426151
## iter  10 value 1199.844737
## iter  20 value 1128.762794
## iter  30 value 1114.673084
## iter  40 value 1092.277283
## iter  50 value 1086.842480
## final  value 1086.827237
## converged
```

```
pred <- predict(nn1, mnist, type='raw')
pred <- as.vector(pred)
auc(mnist$y, pred)
```

```
## Area under the curve: 0.946
```

```
predTest <- predict(nn1, mnistTest, type='raw')
predTest <- as.vector(predTest)
auc(mnistTest$y, predTest)
```

```
## Area under the curve: 0.9551
```

(c)

With the same two features, train three different neural nets, each time using more units in the hidden layer. How do the results improve, using the AUC?

Using more hidden units increases the AUC metric slightly. This corresponds to models that better fit the data and more accurate models. With 20 hidden units, it can be seen that the training AUC increased, but the test AUC decreased slightly. This corresponds to the start of overfitting the model.

```
for (i in c(2, 3, 20)){
set.seed(20305)
nn1 <- nnet(y~x.430+x.266, data=mnist, size=i, decay=.1, maxit=200)
pred <- predict(nn1, mnist, type='raw')
pred <- as.vector(pred)
cat(auc(mnist$y, pred), '\n')
pred <- round(pred, 0)

predTest <- predict(nn1, mnistTest, type='raw')
predTest <- as.vector(predTest)
cat(auc(mnistTest$y, predTest), '\n')
predTest <- round(predTest, 0)
}
```

```
## # weights:  9
## initial  value 3067.840318
## iter  10 value 1433.554536
## iter  20 value 1105.124819
## iter  30 value 1085.356598
## iter  40 value 1063.791578
## iter  50 value 1037.291379
## iter  60 value 1029.084649
## iter  70 value 1014.027830
## iter  80 value 1010.954872
## iter  90 value 1010.297741
## final  value 1010.297553
## converged
## 0.9498296
## 0.9556838
## # weights:  13
## initial  value 3860.025097
## iter  10 value 1164.770018
## iter  20 value 1104.548420
## iter  30 value 1067.837718
## iter  40 value 1052.542075
## iter  50 value 1020.719109
## iter  60 value 1005.309755
## iter  70 value 999.296744
## iter  80 value 998.587670
## iter  90 value 998.338122
## iter 100 value 998.241233
## final  value 998.239092
## converged
## 0.9517139
## 0.9593361
## # weights:  81
## initial  value 3019.274415
## iter  10 value 1058.869088
## iter  20 value 1027.876370
## iter  30 value 1021.090080
## iter  40 value 1015.743007
## iter  50 value 1012.324681
## iter  60 value 1009.143209
## iter  70 value 1003.876674
## iter  80 value 1003.209835
## iter  90 value 1002.852439
## iter 100 value 1001.767632
## iter 110 value 1001.030574
## iter 120 value 1000.658301
## iter 130 value 998.400720
## iter 140 value 995.350144
## iter 150 value 993.579149
## iter 160 value 991.877260
## iter 170 value 990.645351
## iter 180 value 990.100211
## iter 190 value 989.630205
## iter 200 value 989.066690
## final  value 989.066690
## stopped after 200 iterations
## 0.9522033
## 0.9585972
```

(d)

Is there evidence for overfitting in your results in (c)? Use the test data to find out.

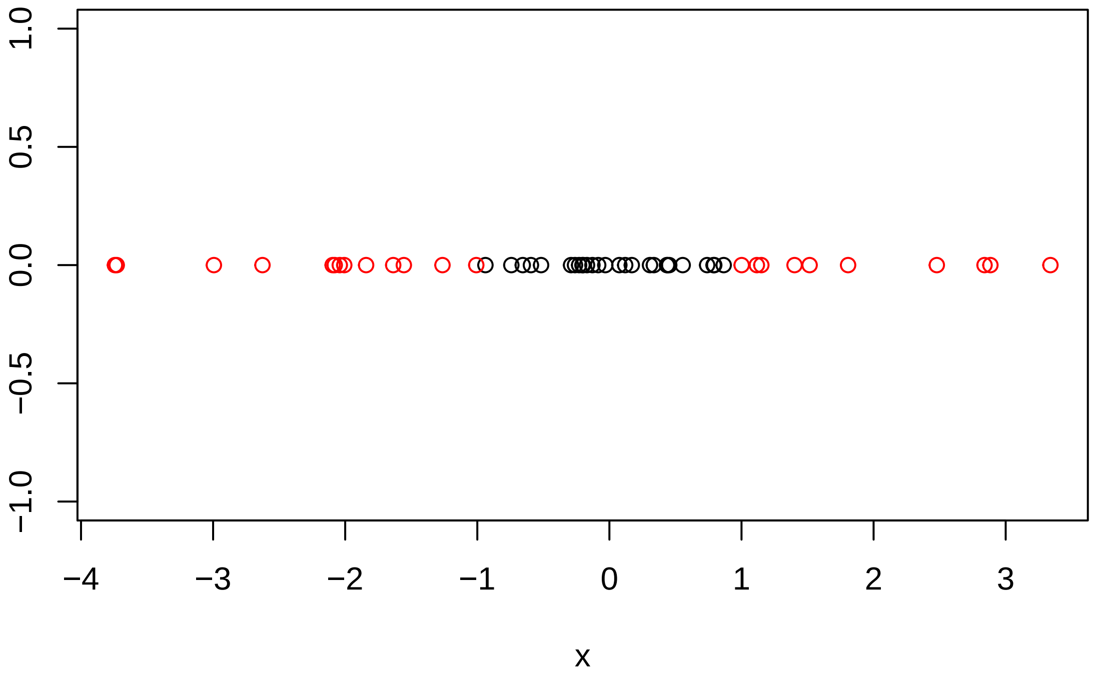
There is some evidence of overfitting in the model. With 20 hidden units, it can be seen that the training AUC increased, but the test AUC decreased slightly.

36

In the Tensorflow Playground, we can use a 'bullseye' dataset to demonstrate non-linear dicision boundaries that would be impossibly difficult for logistic regression. Here we're going to explore that kind of data set in a simplified version.

A 1-dimensional bullseye dataset would be like the following. Notice that one of the classes completely surrounds the other.

```r
set.seed(200)
x <- rnorm(50, 0, 2)
y <- rep(1, length(x))
y[abs(x) < 1] = 0
plot(x,rep(0, length(x)), col=y+1, ylab='')
```



(a)

Fit a logistic regression model to this dataset. Verify that the results are not great.

The AUC is .64, which is not great. It is not much better than randomly predicting the class of each observation.

```r
temp <- data.frame(x=x, y=y)
log.fit <- glm(y~x, data=temp, family='binomial')
summary(log.fit)
```

```
##
## Call:
## glm(formula = y ~ x, family = "binomial", data = temp)
##
## Deviance Residuals:
##    Min      1Q  Median      3Q     Max
## -1.168  -1.102  -1.039   1.138   1.529
##
```

```
## Coefficients:
##             Estimate Std. Error z value Pr(>|z|)
## (Intercept)  -0.1924     0.2887  -0.666    0.505
## x            -0.1812     0.1870  -0.969    0.333
##
## (Dispersion parameter for binomial family taken to be 1)
##
##     Null deviance: 68.994  on 49  degrees of freedom
## Residual deviance: 68.028  on 48  degrees of freedom
## AIC: 72.028
##
## Number of Fisher Scoring iterations: 4
```

```r
pred <- predict(log.fit, temp)
auc(temp$y, pred)
```

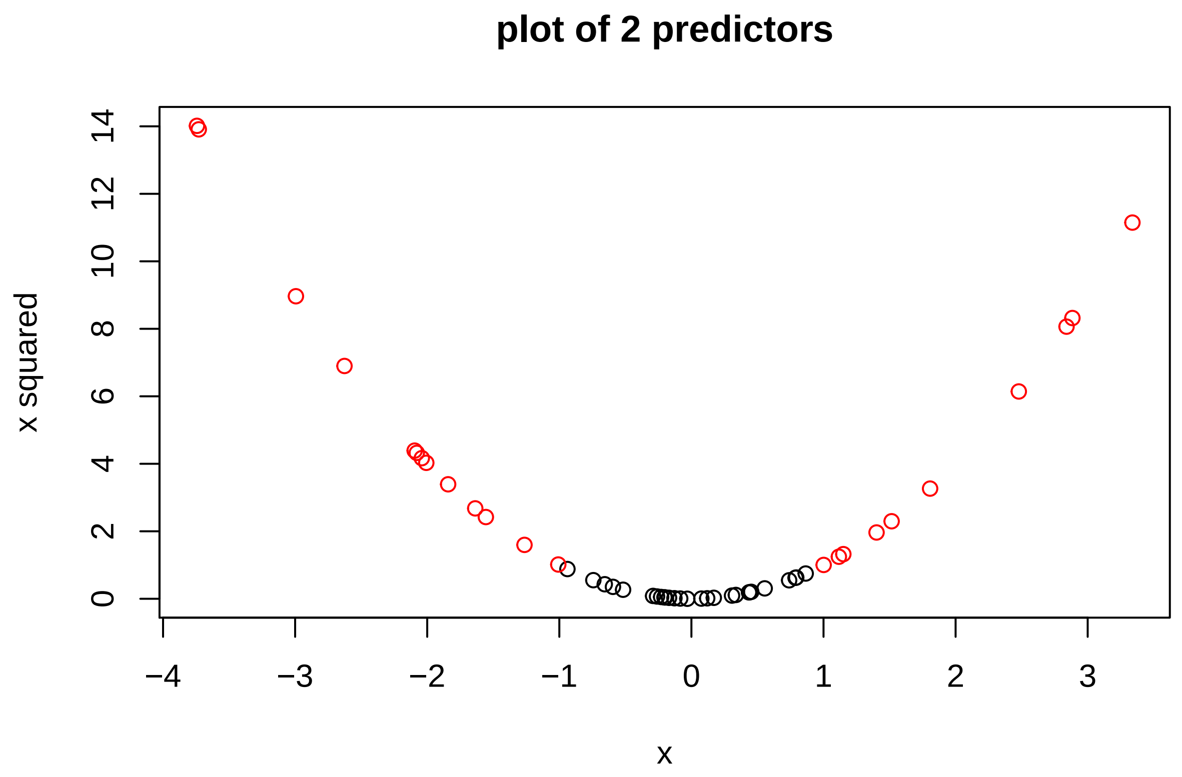
```
## Area under the curve: 0.5652
```

(b)

But we can solve this problem using logistic regression if we employ clever 'feature engineering'. Create a new feature which is just x^2. Make a plot of the two features x and x^2 and color by class label to verify that the two classes are now more easily separable. Fit a logistic regression model and comment on the results.

The area under the curve is 1, which implies the model was perfect. The logistic regression shows the following error message: fitted probabilities numerically 0 or 1 occured. This means that the classes are perfectly separable using the predictor variables $x2$ and x. In other words, achieved perfect separation.

```r
temp$x2 <- temp$x**2
plot(temp$x,temp$x2, col=y+1, xlab='x', ylab='x squared', main='plot of 2 predictors')
```



```
log.fit2 <- glm(y~x2+x, data=temp, family="binomial")
```

```
## Warning: glm.fit: algorithm did not converge
```

```
## Warning: glm.fit: fitted probabilities numerically 0 or 1 occurred
```

```
pred <- predict(log.fit2, temp, type='response')
```

```
auc(temp$y, pred)
```

```
## Area under the curve: 1
```

(c)

If we never thought of this feature engineering, we can also easily solve this problem with a neural network. But importantly, we have to make a network topology such that the hidden layer has higher dimensionality than the input layer. Fit a neural network to $Y\ X$ with two nodes in the hidden layer. Verify that we can achieve perfect classification on the training data.

The area under the curve for the neural network model is 1, which implies perfect classification on the training data.

```
nn2 <- nnet(y~x, size=2, data=temp, decay=.01)
```

```
## # weights:  7
## initial  value 13.079180
## iter  10 value 8.715044
## iter  20 value 4.254165
## iter  30 value 3.345428
## iter  40 value 3.334949
## final  value 3.334941
## converged
```

```
pred <- predict(nn2, temp, type='raw')
pred <- as.vector(pred)
auc(temp$y, pred)
```

```
## Area under the curve: 1
```

(d)

By projecting the data into a higher-dimensional space, we can separate the two classes. In the case of a neural network, the network figured it out for us - we didn't have to do it ourselves. Provide an explanation and intuition into how the network can achieve this goal in this particular case. Your explanation might rely on helpful visualizations.

The neural network maps the input variable x using the logistic function as the activation function. Each input is transformed 2 times and then linearly combined in the hidden layer. The output from hidden layer is passed to the output layer where it is transformed again using a logistic function. It is through this mapping process that the neural network is able to discover a hyperplane that can perfectly separate the two classes.

Extra 37

(a)

Import the data into your R workspace and change all variable names to something simpler. Split the data into a traning set (70%) and a test set (30%).

```
set.seed(207)
names(concrete) <- c('x1', 'x2', 'x3', 'x4', 'x5', 'x6', 'x7', 'x8', 'y')
train.index <- sample(1030, .7*1030)
concrete.train <- concrete[train.index,]
concrete.test <- concrete[-train.index,]
```

(b)

Fit artificial neural networks with a single hidden layer and 2, 3, 4,. . . , 20 nodes to the training data. Compute the root mean squared residuals on the test data and plot them in the same graph.

Root mean squared residuals means root mean squared error

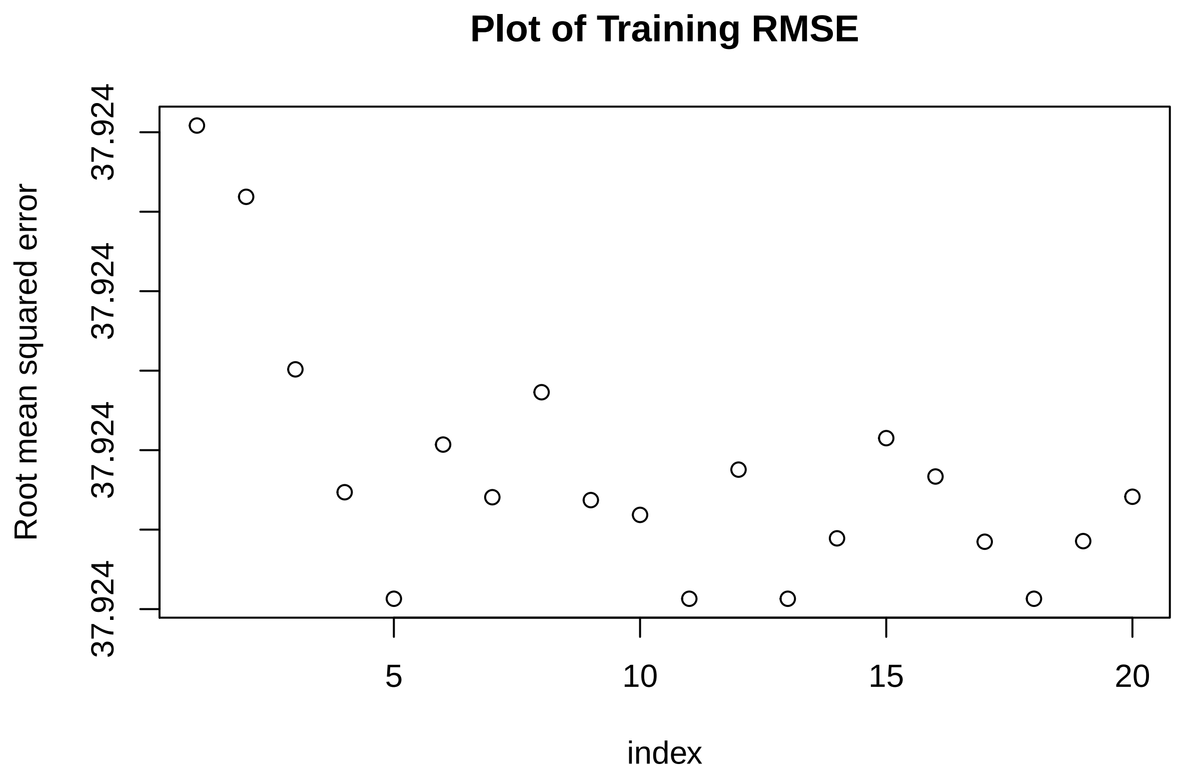
```
set.seed(207)
rmse <- c()
for(i in c(1:20)){
  nn3 <- nnet(y~., data=concrete.train, size=i, decay=.01, maxit=1000)
  pred <- predict(nn3, concrete.train, type='raw')
  residuals <- (pred-concrete.train$y)
  mse <- mean(residuals**2)
  rmse[i] <- sqrt(mse)
}
```

```
## # weights:  11
## initial  value 1063001.813447
## iter  10 value 1036965.212091
## final  value 1036964.660191
## converged
## # weights:  21
## initial  value 1068729.324611
## iter  10 value 1036966.458501
## final  value 1036964.665646
## converged
## # weights:  31
## initial  value 1052024.616874
## iter  10 value 1037210.263870
## iter  20 value 1036981.215699
## iter  30 value 1036965.484563
## final  value 1036964.476172
## converged
## # weights:  41
## initial  value 1069515.434136
## iter  10 value 1036964.850301
## final  value 1036964.241335
## converged
## # weights:  51
## initial  value 1063200.562297
## iter  10 value 1036964.792142
## final  value 1036964.114896
## converged
## # weights:  61
## initial  value 1080646.085735
## iter  10 value 1037098.350263
## iter  20 value 1036964.342728
## final  value 1036964.141443
## converged
## # weights:  71
## initial  value 1062880.297700
## iter  10 value 1037253.311704
## iter  20 value 1036991.243513
## final  value 1036964.065690
## converged
## # weights:  81
## initial  value 1058523.504873
## iter  10 value 1036971.208556
## final  value 1036964.644638
## converged
## # weights:  91
## initial  value 1061876.643174
## iter  10 value 1036977.218244
## iter  20 value 1036964.895433
## final  value 1036964.176433
## converged
## # weights:  101
## initial  value 1059814.264088
## iter  10 value 1036972.759259
## iter  20 value 1036964.052254
## final  value 1036964.031783
## converged
## # weights:  111
## initial  value 1074542.563609
## iter  10 value 1036993.164123
## iter  20 value 1036964.415630
## final  value 1036964.047607
## converged
## # weights:  121
## initial  value 1071845.990398
## iter  10 value 1037201.211712
## iter  20 value 1037024.773864
## iter  30 value 1036964.187280
## final  value 1036964.018030
## converged
## # weights:  131
## initial  value 1048678.315345
## iter  10 value 1037004.117689
## final  value 1036964.136278
## converged
## # weights:  141
## initial  value 1053853.075504
## iter  10 value 1037097.434150
## iter  20 value 1036964.003674
## iter  20 value 1036964.000802
## final  value 1036963.974168
## converged
## # weights:  151
## initial  value 1076896.560074
## iter  10 value 1038099.798164
## iter  20 value 1036971.295600
## iter  30 value 1036964.289990
## final  value 1036964.083454
## converged
## # weights:  161
## initial  value 1079765.827631
## iter  10 value 1036965.630174
## final  value 1036964.070945
## converged
## # weights:  171
## initial  value 1046886.424304
## iter  10 value 1037051.819408
## final  value 1036964.005464
## converged
## # weights:  181
## initial  value 1072982.711006
## iter  10 value 1036987.492280
## iter  20 value 1036964.103938
## final  value 1036963.901275
## converged
## # weights:  191
## initial  value 1059378.045030
## iter  10 value 1037024.349799
## final  value 1036963.872424
## converged
## # weights:  201
## initial  value 1083925.123662
## iter  10 value 1038903.197078
## iter  20 value 1036990.158748
## final  value 1036963.965521
## converged
```

```r
rmse
```

```
##  [1] 37.924 37.924 37.924 37.924 37.924 37.924 37.924 37.924 37.924 37.924
## [11] 37.924 37.924 37.924 37.924 37.924 37.924 37.924 37.924 37.924 37.924
```

```r
plot(c(1:20), rmse, xlab='index', ylab='Root mean squared error', main='Plot of Training RMSE')
```

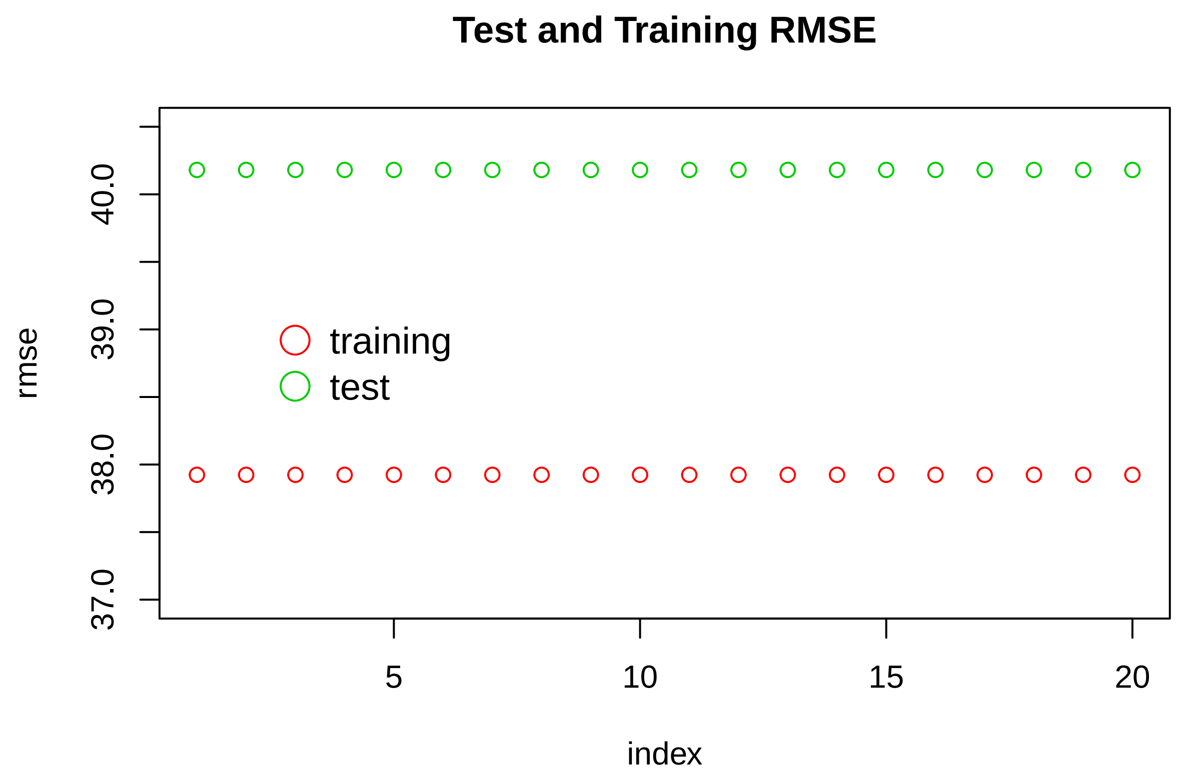


(c)

For the networks in b), compute also the root mean squared residuals on the test data and plot them in the same graph.

```r
rmseTest <- c()
for(i in c(1:20)){
  pred <- predict(nn3, concrete.test, type='raw')
  residuals <- (pred-concrete.test$y)
  mse <- mean(residuals**2)
  rmseTest[i] <- sqrt(mse)
}
```

```r
plot(c(1:20), rmse, col=2, type="p", ylim=c(37,40.5), xlab='index',
     main='Test and Training RMSE')
points(c(1:20), rmseTest, col=3)
legend('left', legend=c('training', 'test'), col=c(2,3), pch=1,
  bty = "n",
  pt.cex = 2,
  cex = 1.2,
  text.col = "black",
  horiz = F ,
  inset = c(0.1, 0.1))
```



(d)

Is there evidence of overfitting? How can you tell?

There is a slight evidence of overfitting because the test rsme is higher than the training rmse.

(e)

Do you think that the ANN is overfitting the data?

Same answer as part d.