Anthony Tummillo

HW Report #7

Problem 1.

Part a.
There did not appear to be a significant change in the mean misclassification error or mean square error while using the traincgf function on the single layer neural network regardless of how I changed the parameters net.trainParam.epochs, net.trainParam.show, and net.trainParam.max_fail.

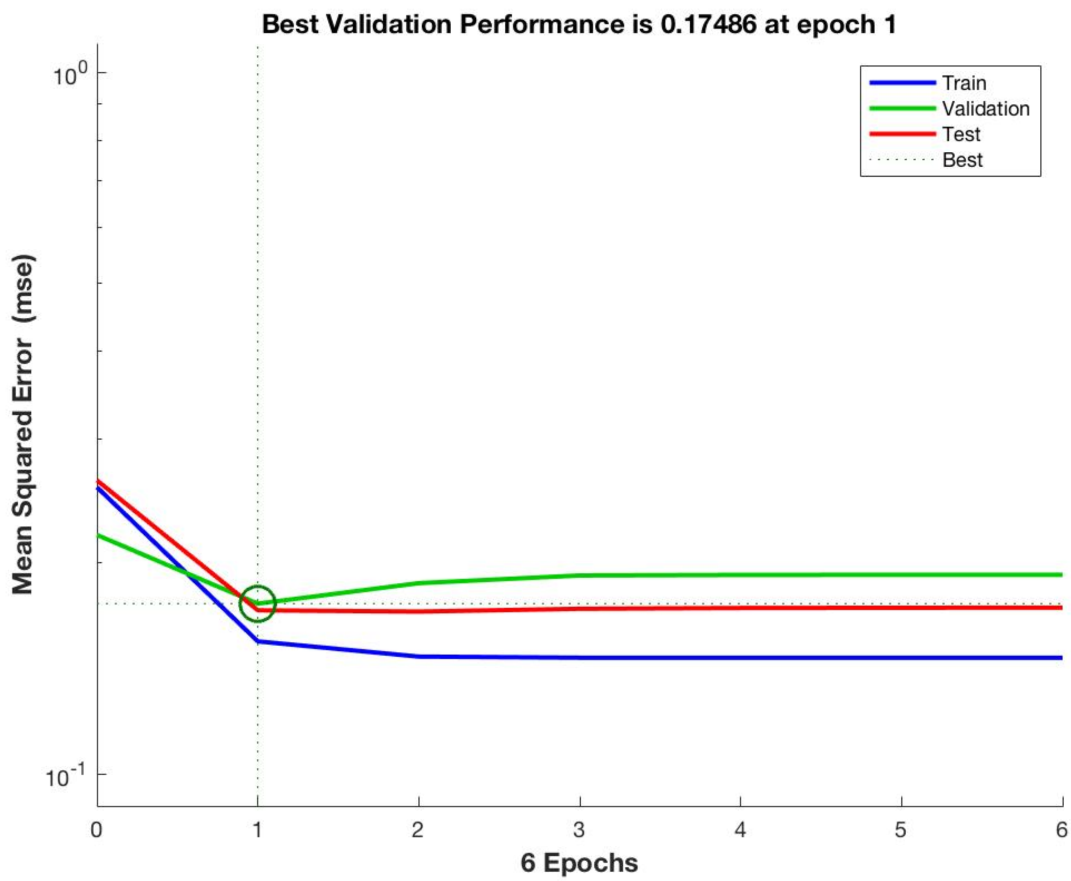
The train and test classification error tended to float around 0.23+/-0.02
The train and test mean square error stayed around 0.46+/-0.02

I did, however, notice that when using the given values of epochs, show, and max_fail (2000, 10, and 5 respectively) with the trainlm function I was able to get a lower mean misclassification error and a significantly lower mean square error. I have included the weights, errors, and performance graph of this model below.

Weights:
w0 = 0.3231
w1 = 0.9712
w2 = 3.4008
w3 = -0.3573
w4 = -0.4936
w5 = 0.1797
w6 = 2.1343
w7 = 1.8285
w8 = 0.5915

Train classification error = 0.2301
Test mean square error = 0.1571

Test classification error = 0.1965
Test mean square error = 0.1478



Part b.
For this part, I set the number of units in the hidden layer to 2, epochs to 2000, show to 10, and max_fail to 2000. Once reaching 2000 epochs I found these results:

Train classification error = 0.2226
Test mean square error = 0.1551

Test classification error = 0.2096
Test mean square error = 0.1485

On average, it appears that by adding two units to a hidden layer in the neural network, the mse and classification error for the train data slightly improves while the mse and classification error for the test data gets slightly worse.

I would say that in this case the first model (the one from part a) is better because it produces lower mse and classification error for the test data. Based on these results I would assume our data has a decently linear decision boundary. This would mean that adding multiple logistic regression units in a hidden layer to try and model nonlinearities would cause overfitting, which I believe is being expressed here.

Part c.

Hidden units = 2:

Train function = trainlm show = 10 epochs = 2000 max_fail = 2000
Train classification error = 0.2301
Train mean square error = 0.1563
Test classification error = 0.2096
Test mean square error = 0.1474

Train function = trainlm show = 10 epochs = 2000 max_fail = 15
Train classification error = 0.2319
Train mean square error = 0.1551
Test classification error = 0.1878
Test mean square error = 0.1460

Train function = traincgf show = 10 epochs = 2000 max_fail = 2000
Train classification error = 0.2263
Train mean square error = 0.4838
Test classification error = 0.2183
Test mean square error = 0.4476

Train function = traincgf show = 10 epochs = 2000 max_fail = 15
Train classification error = 0.2876
Train mean square error = 0.6135
Test classification error = 0.2707
Test mean square error = 0.6288

Hidden units = 3:

Train function = trainlm show = 10 epochs = 2000 max_fail = 2000
Train classification error = 0.2263
Train mean square error = 0.1562
Test classification error = 0.2140
Test mean square error = 0.1457

Train function = trainlm show = 10 epochs = 2000 max_fail = 15
Train classification error = 0.2171
Train mean square error = 0.1519
Test classification error = 0.2314
Test mean square error = 0.1468

Train function = traincgf show = 10 epochs = 2000 max_fail = 2000
Train classification error = 0.2449
Train mean square error = 0.5052
Test classification error = 0.2489

Test mean square error = 0.4654

Train function = traincgf show = 10 epochs = 2000 max_fail = 15
Train classification error = 0.2430
Train mean square error = 0.5005
Test classification error = 0.2314
Test mean square error = 0.4516

Hidden units = 5:
Train function = trainlm show = 10 epochs = 2000 max_fail = 2000
Train classification error = 0.2134
Train mean square error = 0.1417
Test classification error = 0.1921
Test mean square error = 0.1441

Train function = trainlm show = 10 epochs = 2000 max_fail = 15
Train classification error = 0.1967
Train mean square error = 0.1404
Test classification error = 0.1921
Test mean square error = 0.1430

Train function = traincgf show = 10 epochs = 2000 max_fail = 2000
Train classification error = 0.2319
Train mean square error = 0.4778
Test classification error = 0.2052
Test mean square error = 0.4356

Train function = traincgf show = 10 epochs = 2000 max_fail = 15
Train classification error = 0.2597
Train mean square error = 0.5207
Test classification error = 0.2533
Test mean square error = 0.5039

Hidden units = 10:
Train function = trainlm show = 10 epochs = 2000 max_fail = 2000
Train classification error = 0.2115
Train mean square error = 0.1518
Test classification error = 0.2489
Test mean square error = 0.1579

Train function = trainlm show = 10 epochs = 2000 max_fail = 15
Train classification error = 0.2096
Train mean square error = 0.1477
Test classification error = 0.1921

Test mean square error = 0.1432

Train function = traincgf show = 10 epochs = 2000 max_fail = 2000
Train classification error = 0.2319
Train mean square error = 0.4729
Test classification error = 0.2052
Test mean square error = 0.4454

Train function = traincgf show = 10 epochs = 2000 max_fail = 15
Train classification error = 0.2171
Train mean square error = 0.4550
Test classification error = 0.2183
Test mean square error = 0.4484

Analyzing this data seems to be a quite complex task as there is a lot going on. However, there are some fairly consistent trends that should be noted. One is that with trainlm function, train and test error both tended to get better with a lower max_fail value. Another is that with the traincgf function error seems to improve for the train data and get worse for the test data as the value of max_fail decreases. There are some exceptions in the data I found but I can see some evidence of and would hypothesize that as you add more hidden units to the model trained on this data the error gets worse. I believe that the more units you add the more overfitting you get, and thus worse error.

My hypothesis is supported by the fact that the best accuracy and error calculcated in this problem was using the normal logistic regression without any hidden layers.

Problem 2.

Part a.
Between the two trees built by the code provided, assuming no pruning is performed, the second tree built (new_tree) is better for prediction as it performs with a lower misclassification error on the test set. This is likely due to over fitting.

However, when experimenting with pruning I found that the first tree which (built with less restrictions) achieved the lowest overall error between the two trees of 0.2009 at a pruning level of 8. The minimum I found for the second tree (built with more restrictions) was 0.2271 at a pruning level of 2. It was possible to lower the misclassification error of both trees through backpruning, so I believe that we should always try to perform backpruning.

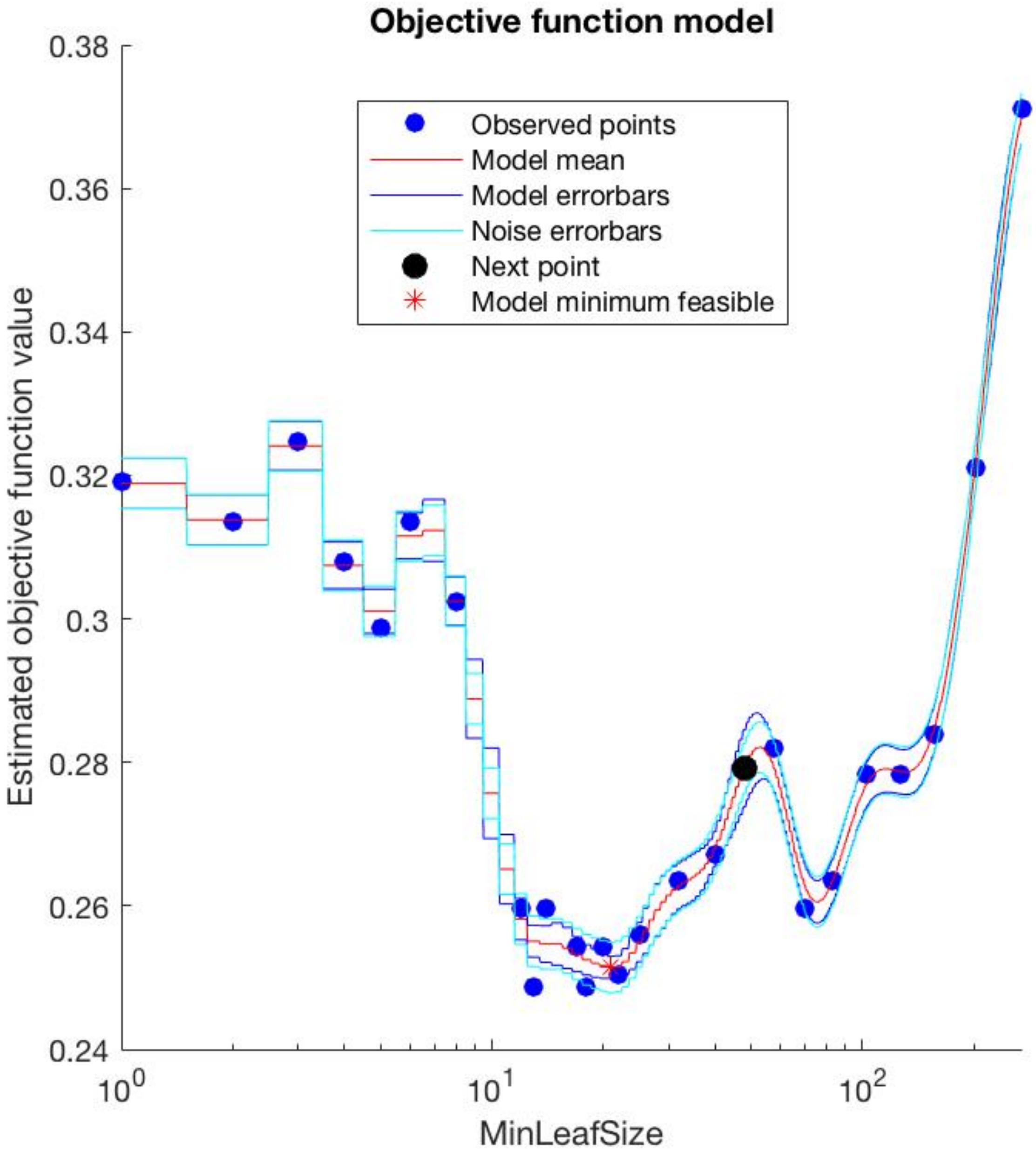
Part b.

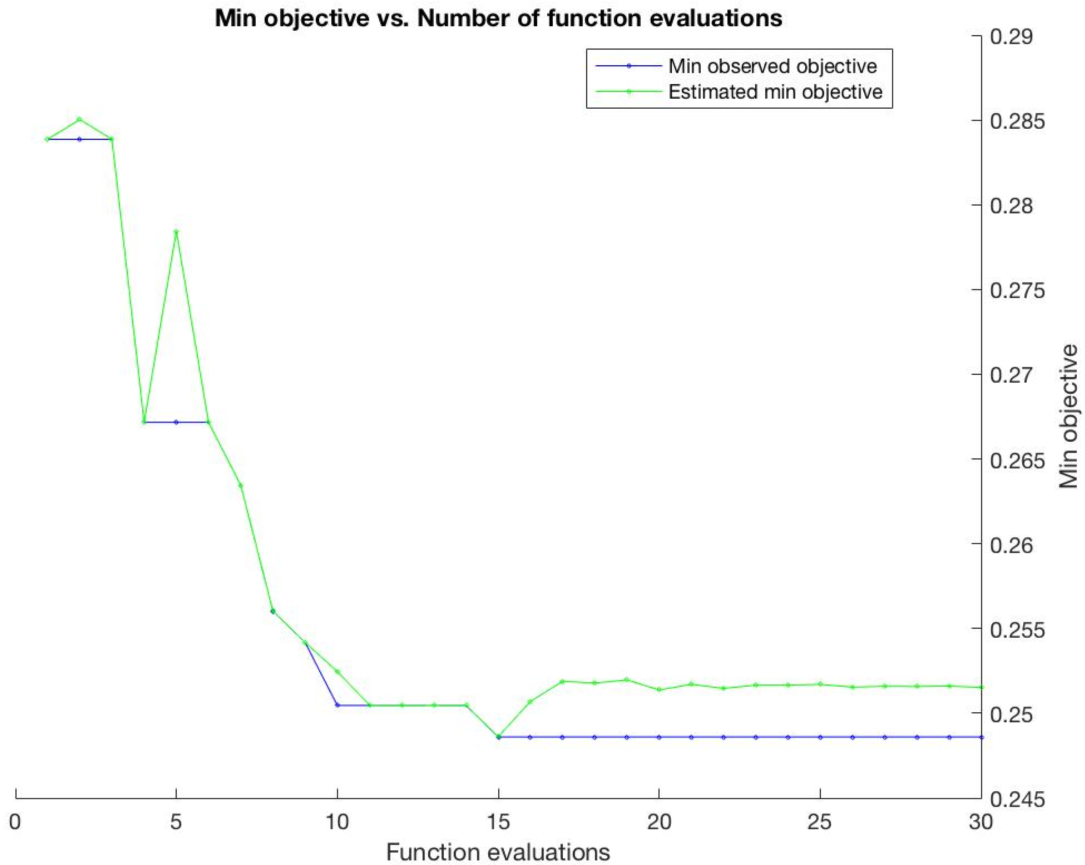
Additional Settings: splitcriterion = twoing

Unrestricted tree test error = 0.2751
Restricted tree test error = 0.2576

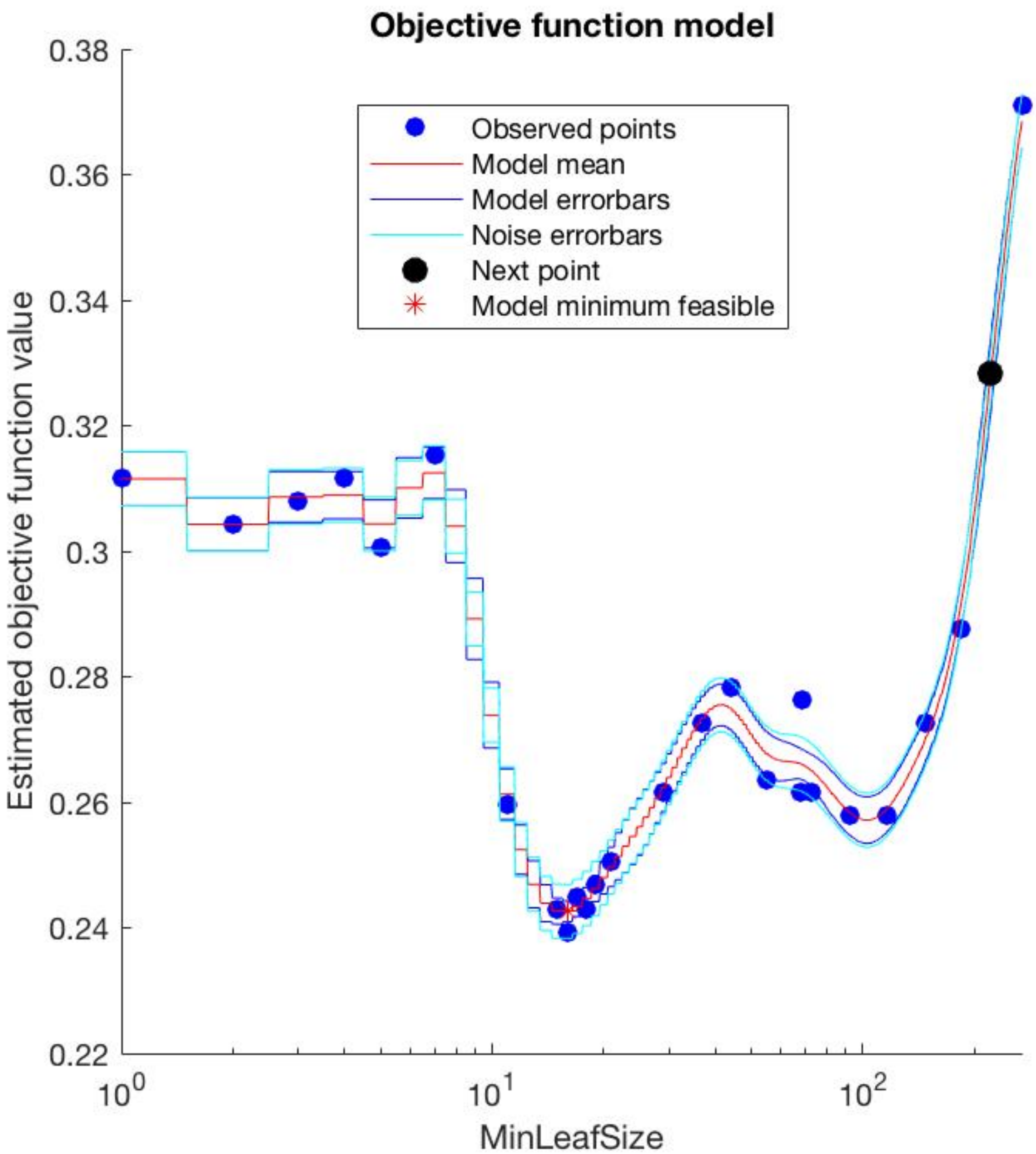
Additional Settings: splitcriterion = deviance
Unrestricted tree test error = 0.2882
Restricted tree test error = 0.2620

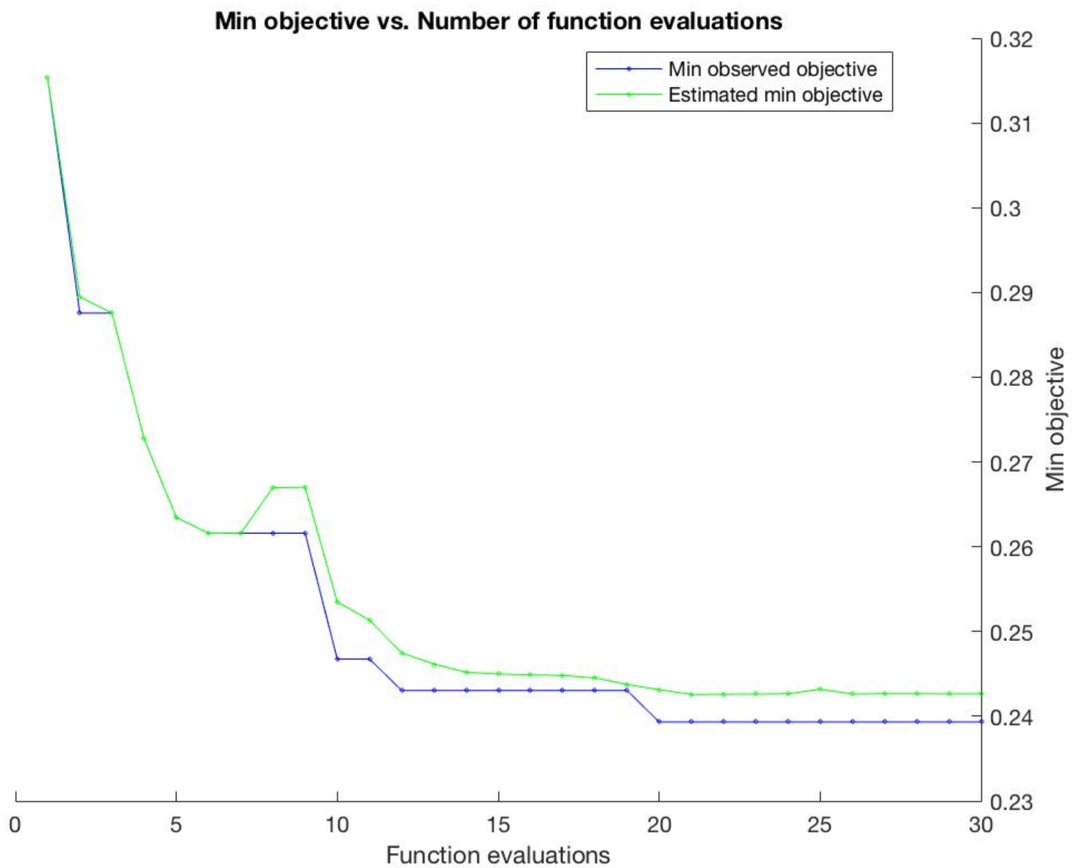
Settings: OptimizeHyperparameters = auto
(graphs obtained from hyperparameter optimization are included below as well)
Unrestricted tree test error = 0.2358


Objective function model
Observed points
Model mean
Model errorbars
Noise errorbars
Next point
Model minimum feasible
Estimated objective function value
0.38
0.36
0.34
0.32
0.3
0.28
0.26
0.24
10^0
10^1
10^2
MinLeafSize



Restricted tree test error = 0.2227


Objective function model
Observed points
Model mean
Model errorbars
Noise errorbars
Next point
Model minimum feasible
Estimated objective function value
MinLeafSize
0.38
0.36
0.34
0.32
0.3
0.28
0.26
0.24
0.22
10^0
10^1
10^2



Problem 3.

Part a.
With neighbors set to 3:
Accuracy = 0.7336
Error = 0.2664

With neighbors set to 1:
Accuracy = 0.7031
Error = 0.2969

With neighbors set to 5:
Accuracy = 0.7773
Error = 0.2227

Part b.
With neighbors set to 3 and normalized data:
Accuracy = 0.7555

Error = 0.2445

With neighbors set to 1 and normalized data:
Accuracy = 0.7118
Error = 0.2882

With neighbors set to 5 and normalized data:
Accuracy = 0.7686
Error = 0.2314

After normalizing both the train and test datasets my results improved when using 3 and 1 neighbors, however, my results got worse after normalization when using 5 neighbors.

Part c.
h = 0.2
error = 0.2882
accuracy = 0.7118

h = 0.5
error = 0.2576
accuracy = 0.7424

h = 2
error = 0.2795
accuracy = 0.7205

h = 10
error = 0.2969
accuracy = 0.7031

h = 50
error = 0.2969
accuracy = 0.7031

For the values of h that I tested it appears that this model tends to perform worse than the model from part b of this question.

It should be noted that at a certain value of h that increases in h's value stop changing the calculated accuracy and error.

Problem 4.

Part a.

1.) P(F | A, D) = P(F | D)   P(F, A | D) = P(F | D)*P(A | D)

2.) P(D | C, E) = P(D | C)   P(D, E | C) = P(D | C)*P(E | C)

3.) P(F | B, D) = P(F | D)   P(F, B | D) = P(F | D)*P(B | D)

4.) P(F | C, D) = P(F | D)   P(F, C | D) = P(F | D)*P(C | D)

5.) P(A, B) = P(A)*P(B)

Part b.

There would be 96 – 1 (due to inference) = **95** variables in the problem domain.

Part c.

P(A, B, C, D, E, F) =
P(F | A, B, C, D, E) * P(A, B, C, D, E) =
<u>P(F | D)</u> * P(D | A, B, C, E) * P(A, B, C, D, E) =
<u>P(F | D)</u> * <u>P(D | A, B, C)</u> * P(E | C, B, A) * P(A, B, C) =
<u>P(F | D)</u> * <u>P(D | A, B, C)</u> * <u>P(E | C)</u> * <u>P(A)</u> * <u>P(B)</u> * <u>P(C)</u>

Part d.

There are **24** parameters needed to define the belief network in the figure.