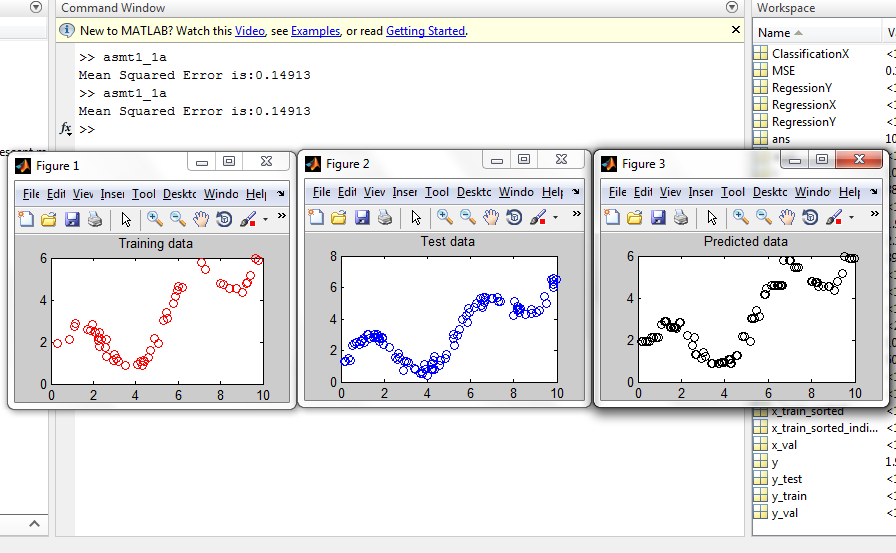
**ECE521 Assignment #1**

**James Schuback**

**Feb 7th, 2014**

***All scripts and algorithms are included in the appendices.***

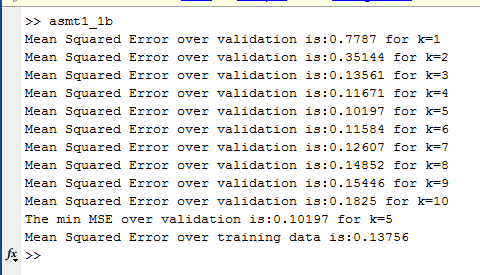
1. a) Nearest neighbors. Use the nearest neighbor method to make predictions for the 100 test cases using the 50 training cases. *Compute and report* the mean squared error (MSE) on the test set.



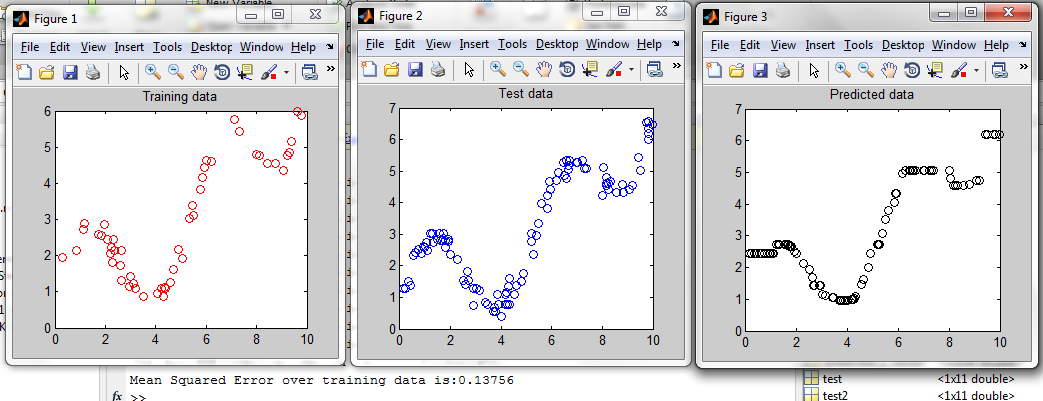
The above image contains the test results. **The Mean Squared Error (MSE) was found to be 0.14913**, which represents approximately 2.42% of the range of Y values.

The above 3 plots show the training data (of 50 (x,y) data points), the test data (of 100 (x,y) data points) and the predicted data showing (test case x, nearest neighbor y) pairs. The MSE was calculated from predicted and test data y values. See the script “asmt1\_1a.m” in the appendix.

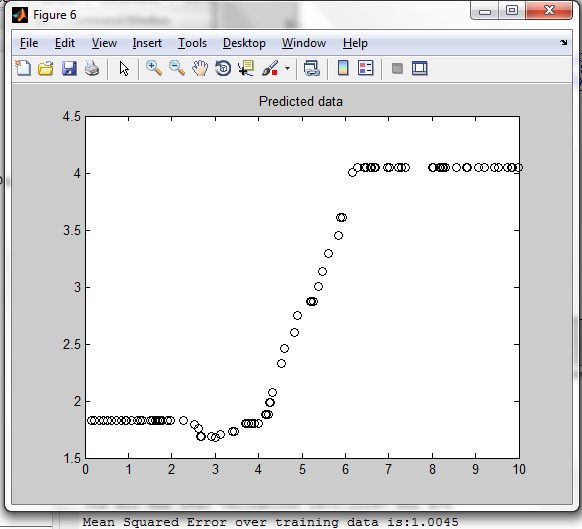
1. b) k-nearest neighbors. For k ranging from 1 to 10, use the k-nearest neighbor method to make predictions for the 50 validation cases. *Compute and report* the 10 corresponding values of the MSE. *Which value of k achieves the lowest validation error?* Use that value of k to make predictions for the 100 test cases and *report the MSE* on the test set. *Discuss how this value compares* to the MSE reported in (1a).



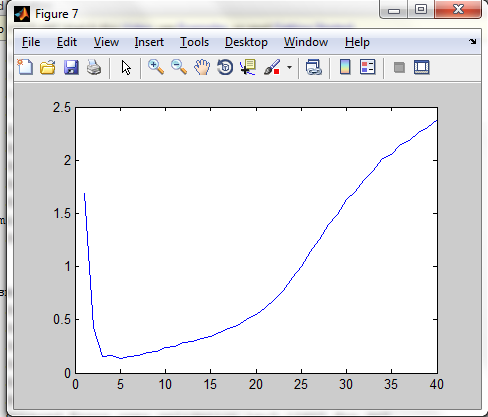
The above figure shows numeric results for the K nearest neighbor tests over the validation data (see script “asmt1\_1b”). **The minimum MSE was achieved for k=5 with a value of 0.10197**. Below we see the plots of training data, test data, and predicted data for nearest neighbors k=5:



**The MSE over training data for k=5 was 0.13756.** This is lower than the above result for k=1 trained over training data, and represents approximately 2.23% of the range of RegressionY data. Interesting to note is the predicted curve shape for k=5 – as k increases, the model tends to average out segments of data producing a flatter and more piecewise-linear-looking shape. A quick test for k=25 showed the following:

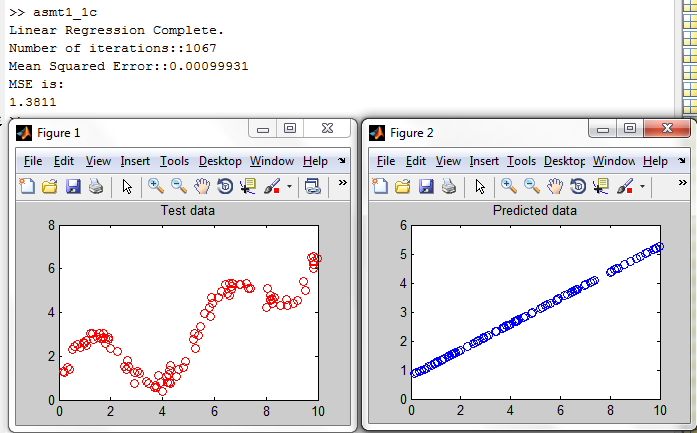


If increasing k serves to linearize the model, it is clear that there reaches a point where more linearization will only increase the MSE; in this example, the answer was k=5. Plotted below is the MSE as a function of k for k=1 to 40:



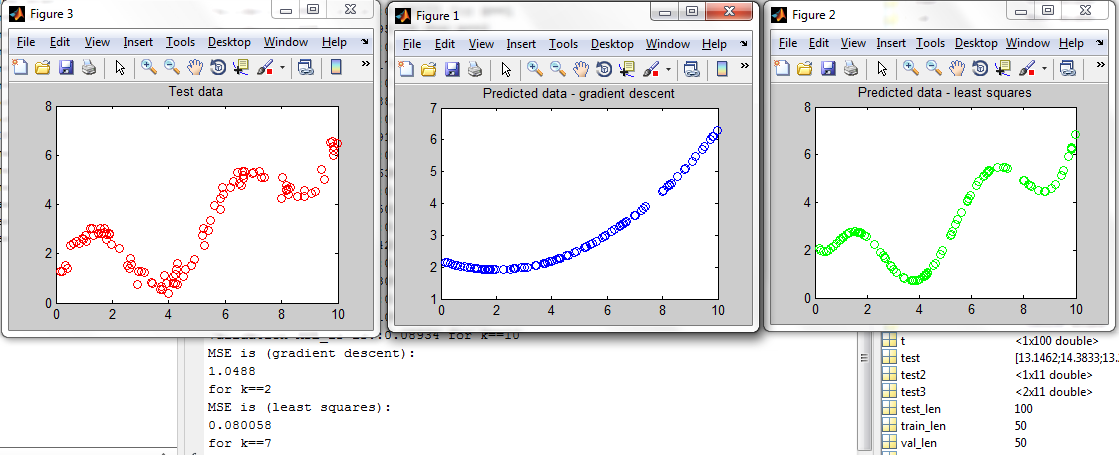
The MSE as a function of k attains a minimum at k=5 (results obtained above) and further diverges with increasing k.

1. c) Linear regression. Use the steepest descent method to fit a linear regression model (two parameters, randomly initialized uniformly between -0.1 and 0.1) to the training data. Make predictions for the test cases and *report the MSE*.



The 2-param linear regression model fits a straight-line to the data. After over 1000 iterations, **the MSE was found to be 1.3811**. Increasing the number of iterations to 200000, the MSE over the test data only changes to 1.3806. The MSE represents about 22% of the full RegressionY data range, a relatively high data. Clearly a more sophisticated model is needed.

1. d) Linear regression using polynomial inputs. Expand the input x to form a vector of inputs (x, x2, x3, …, xk), for a positive integer k. Use steepest descent to fit linear regression models to the training data, for k ranging from 1 to 10. Note that for a given value of k, there will be k+1 parameters. For each model, make predictions for the validation cases. Compute and *report the 10 corresponding values of the MSE*. *Which value of k achieves the lowest validation error?* Use that value of k to make predictions for the 100 test cases and *report the MSE* on the test set. *Discuss how this value compares* to the MSE reported in (1c).



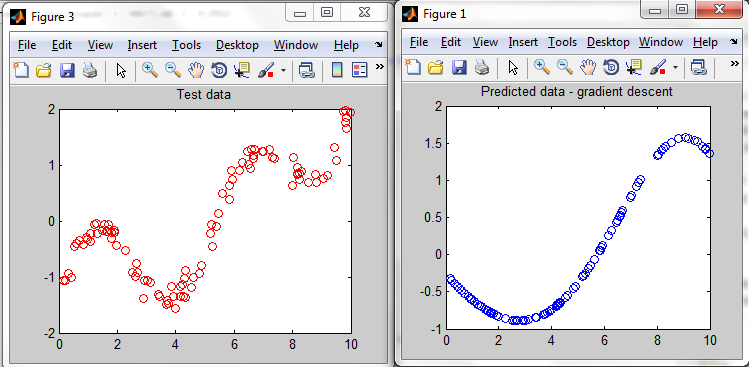
Unfortunately, steepest/gradient descent did not converge for k > 2 despite trying numerous “tricks” such as dynamically adjusting the learning rate, cold restarts of the weight vector upon divergence, dynamically changing weight vector bounds, and increasing the number of iterations. However, the **k = 2** fit was found to be better **with an MSE of 1.0488**. This is at least better than the linear fit above.

Interesting to note is that a simple *least squares* approach (i.e. the analytic solution to the simplest quadratic program) proved to work exceedingly well – and much faster due to the dimensions of this toy problem. On the right in the above image is the **k = 7** curve fit which on the test data achieved **an MSE of 0.08**, or approximately 1.3% of the full scale range of RegressionY.

These results beg the question of when one would ever opt to use gradient descent for curve fitting in place of least squares, as the former is less accurate, non-robust, and has much worse time efficiency.

EDIT – upping to 5 million iterations, gradient descent achieved an MSE of 1.0679 for k = 3 (cubic curve fit). It is not unreasonable to believe that given enough iterations and a low enough learning rate than gradient descent could approach the results of least squares, but given the time required it seems unreasonable to continue trying.

EDIT 2 – Normalizing the data via the transform z = (x – μ)/σ for both x = RegressionX and RegressionY data allowed the polynomial linear regression to converge in much less iterations and at a lower learning rate. With k = 7 (same k of best polynomial fit that least squares found), the **MSE was found to be 0.27289** **under the above normalizing transformation.** This equates to an MSE of approximately **7.35% of the full scale range of RegressionY data under the same transformation**. These results are much better, but the MSE is still 7x greater than least squares results.



1. e) Neural Networks. …*Submit a single figure* that plots the training error vs the logarithm of the number of epochs, for a variety of learningrates (there should be one curve for each learning rate). Select a learning rate that balances thesetwo concerns and *report its value.* Using the model that was trained using the selected learning rate*, report the test error.*

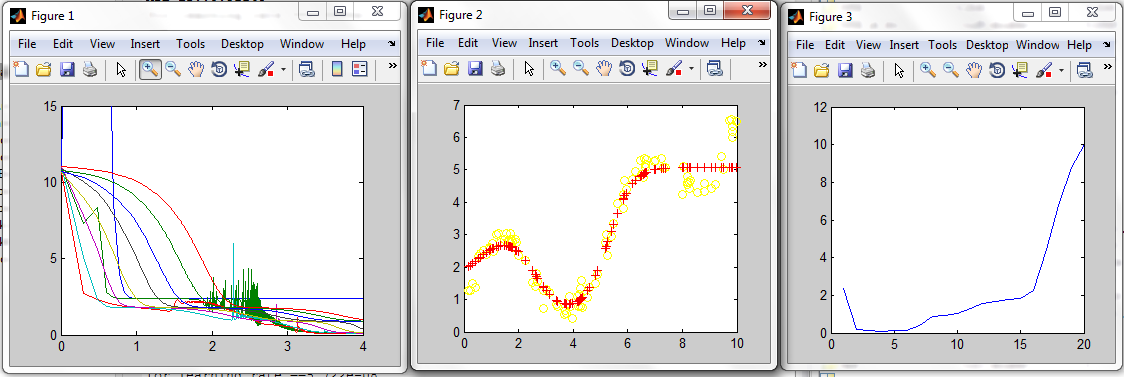


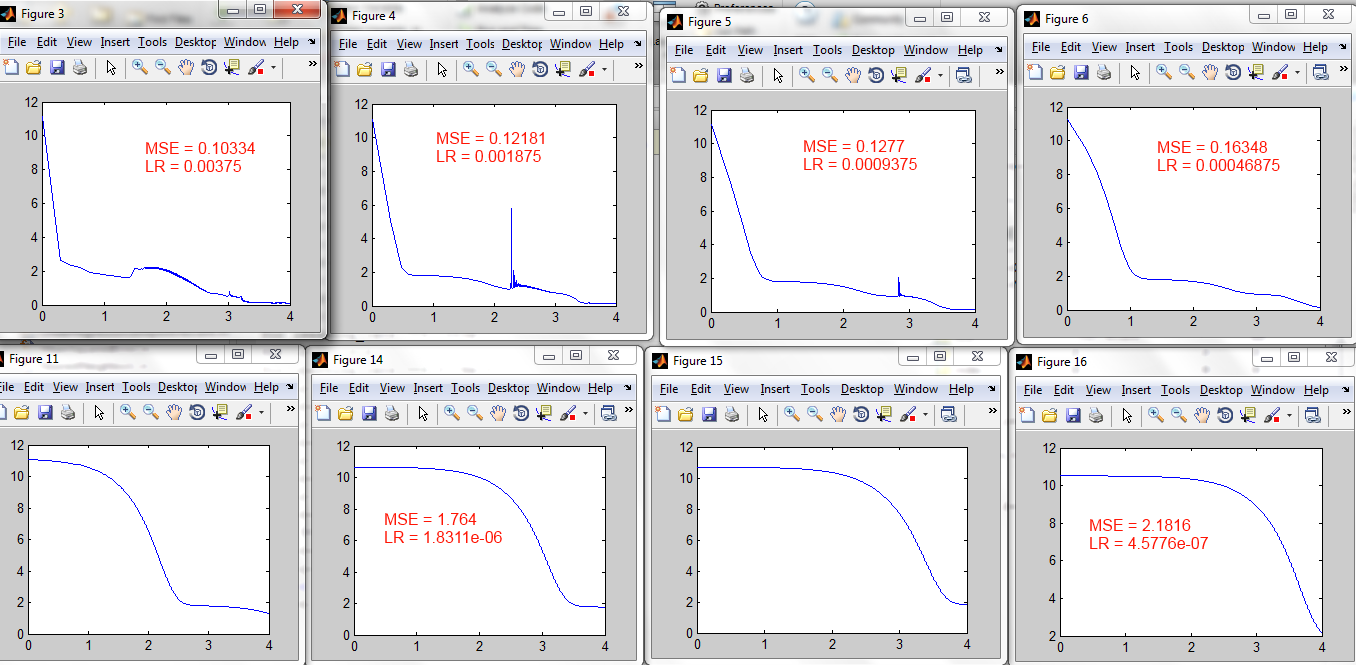
Figure 1 shows the training error (y axis) vs the logarithm of iteration count (at 10000) for 20 different learning rates. The learning rates tested were of the form:

learning\_rate = 0.03/(2^i);

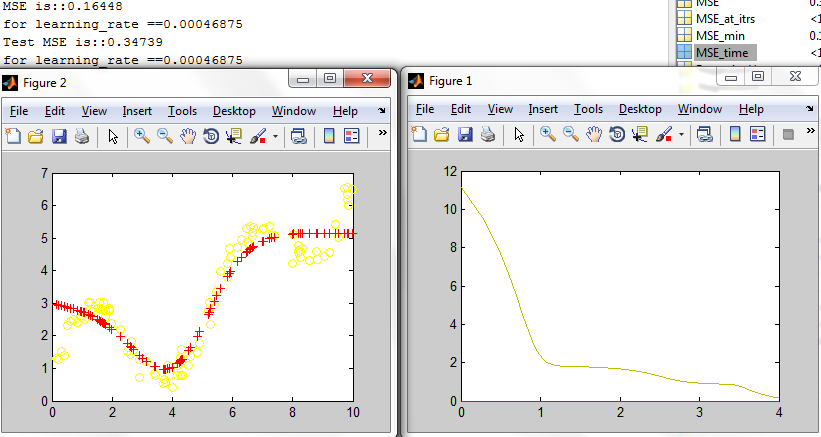
As variable i iterated from 1 to 20. LRs higher than 0.03/2 were found to not converge. As can be seen from figure 1, the training error curve spikes on its way towards lower error as the nonlinear NN model is adjusted.

Figure 2 shows the predicted fit on the test data using a **learning rate of 0.001875** which was found to produce a **test MSE of 0.21415** (the lowest MSE) or about 3.48% of the full scale of RegressionY.

Figure 3 shows the training MSE (y axis) vs. i (which is the learning rate from the above equation). i == 4 (the above learning rate) was found to end with the lowest training MSE after 10000 iterations. However, i == 4 may not be the most stable learning rate as the below figure shows for another trial run showing a few training error vs. log epochs for a few LRs:



The best LRs in terms of training error, i == 3, 4, and 5, have spikes suggesting some instability in convergence due to the large LR. By i == 6 and above, the training error curve becomes monotonic (strictly decreasing) meaning that it may be a “safer” option to train with. The MSE is potentially 60% worse, but it represents some balance. Re-testing on test data after training with lr(i==6), the following curve and MSE are found: (training error vs. log epochs plotted on the right)



The **test MSE of this more “balanced” learning rate is 0.34739**, or about 5.64% of the full scale of RegressionY. Since we “trust” the numeric stability more, let’s try increasing the number of iterations and see if the fit gets better while maintaining this strictly decreasing test error:

The number of iterations for the above LR is now 10^6 rather than 10^4, and the test MSE was found to be 0.2119. Is this better than what would have happened had we selected the “bumpier”/”less smooth” LR from i == 3 at 10^6 itrs? That would be the whole point I would think – don’t waste time doing the full learning/training on an unstable LR. However, the test MSE for LR(i==3) was still better, at 0.13509. But the error vs epoch curve was non-monotonic; suggesting that at certain epoch choices in between, the neural net could have converged to a worse result.

To prove this in the context of this example, I hand-picked an epoch number (about 100 iterations) where the training error of LR(i==3) achieved a local maximum. Would the smoother LR(i==6) perform better for this epoch?

Test MSE for LR(i==3) and num\_itrs==100: 2.1974

Test MSE for LR(i==6) and num\_itrs==100: 2.1313

Yes, it does. This proves that there is an actual balance to strike: the monotonicity of the error curve suggests that lower learning rates are more stable to variations in number of iterations, while higher LRs can be subject to hitting local maxima in error for a given number of iterations. But for the purposes of this toy example, clearly slightly poorer performance (on the order of a few %) for 100 iterations is not something to be concerned about. However it would be interesting to further investigate this phenomenon.

1.f) Neural networks: Random restarts… *Submit a histogram* of the 20 resulting training error values. *How much variability is there?* Compute and *report the test error* and *compare it to the* *test error* obtained in (1e).

The learning rate used is the one above found to have the best test MSE despite having intermittent error curve fluctuation: LR == 0.00375. This LR has a higher probability of showing something interesting in the histogram…

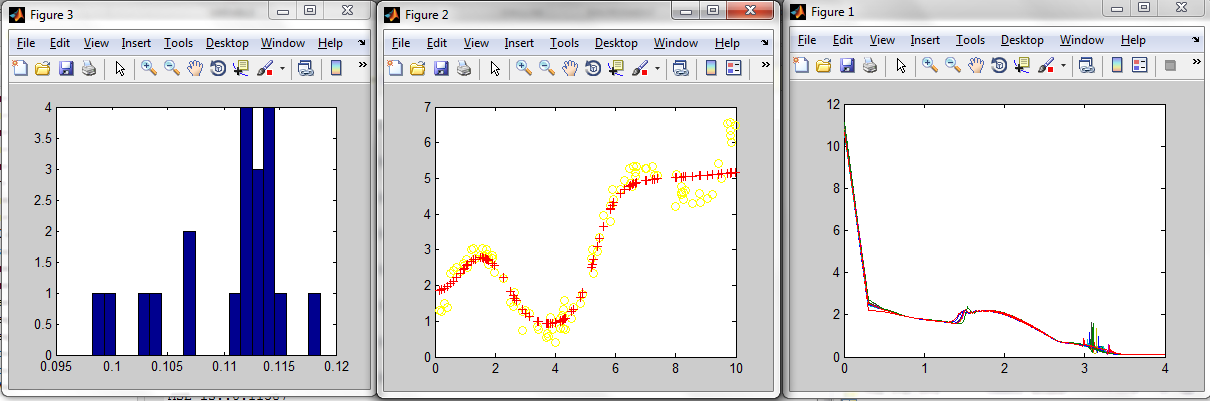
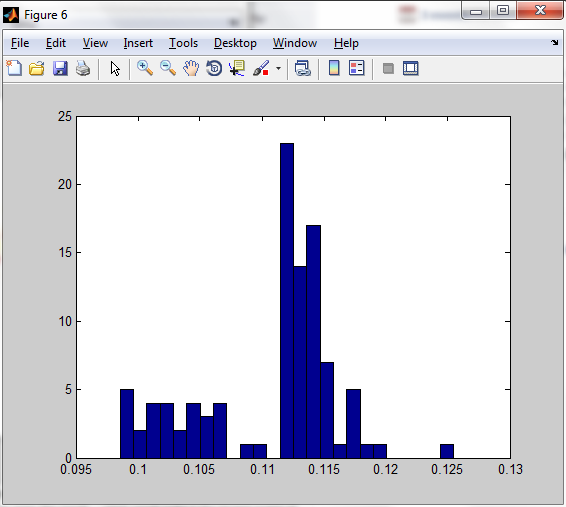


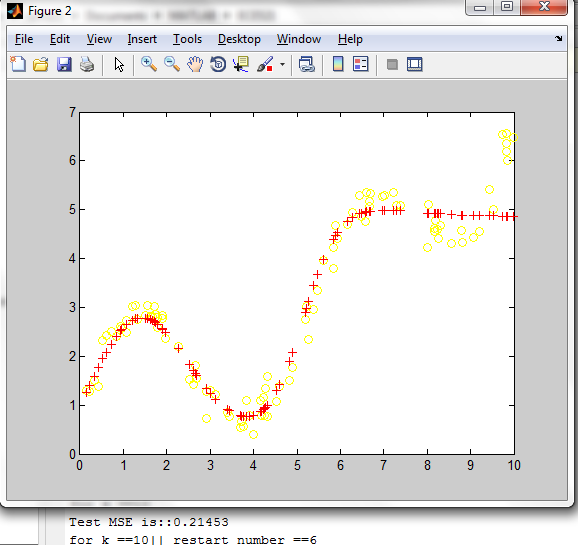
Figure 3 shows the histogram (frequency vs. MSE buckets) of training error values with 20 buckets. **The mean is 0.1104, and the standard deviation is 0.0054. The minimum is 0.0983 while the maximum is 0.1186**. The **test MSE using the best results of the 20 random restarts (fit in figure 2) is 0.19201**, which in terms of full-scale range, is **about 10% better**.

Thinking that the histogram might reveal a more meaningful shape for 100 random restarts, I ran that experiment as well: (histogram with 25 buckets):



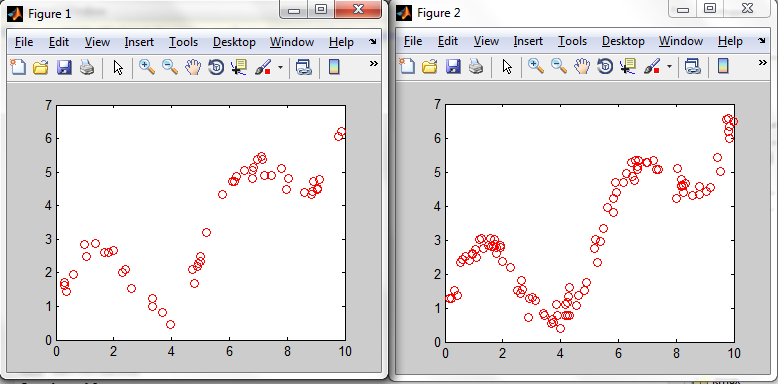
It’s starting to look a little more like an even distribution. Interestingly though, the test MSE using the best of these 100 random restarts was only 0.19364, slightly worse than the result for 20 restarts; however it was still better than the result for no restarts at all (1.e).

1. g) Neural networks: Selecting the number of hidden units. *Compute and report the validation error* for these 10 models. *Which value of k achieves the lowest validation error?* Use the model with lowest validation error to makepredictions for the 100 test cases and *report the MSE* on the test set. *Discuss how this value compares* to the MSE reported in (1a) to (1f). [also uses 20 random restarts]



The **test MSE after training on the validation error was found to be 0.21453.** This occurred the model trained for k = 10 (10 hidden logistic units) on random restart number 6.

This is actually one of the worst results yet for test MSE, beating only linear regression and neural networks using an inferior learning rate. But it’s still of the same order of magnitude. The only other experiment which trained over validation was nearest neighbors, but that achieved 2x less MSE. Let’s examine why this might be by looking at the data itself:



On the left in figure 1 we see validation data, while on the right in figure 2 is the test data.

The curve fit for k = 10 in the image above this suggests that the main accumulation of the error begins for x > 8. From approximately x = 6 to x = 8, the fit is linear, which is very much in line with the validation data for x > 6: it looks like a noisy linear function with slightly negative slope, plus two data points near x = 9.5 that want to drag up the data. Compare this with the test data where there is much more lower ratio of data points between x = [6,9] and x = [9,10].

Fitting based on minimizing MSE to the validation set is then difficult because it’s hard to account for the two data points at x > 9.5 as anything other than noise. Meanwhile there are 15 or so data points forming the negative linear line from x = [6.5, 9.5].

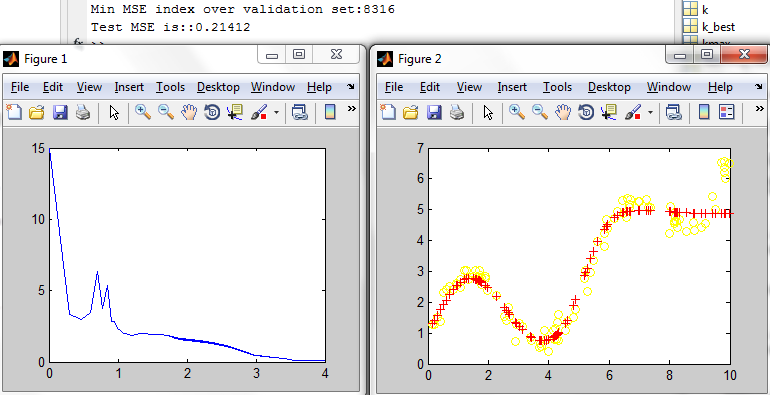
But note that nearest neighbors doesn’t have a problem with this. The test data has a bunch of points x > 9? No problem, they’re close to those two validation points we trained on for x > 9.5.

This suggests a simple heuristic for discriminating between MSE-minimizing learning routines and nearest neighbor routines:

**MSE-minimizing:** trust the density of data – where there’s more, there’s truth.

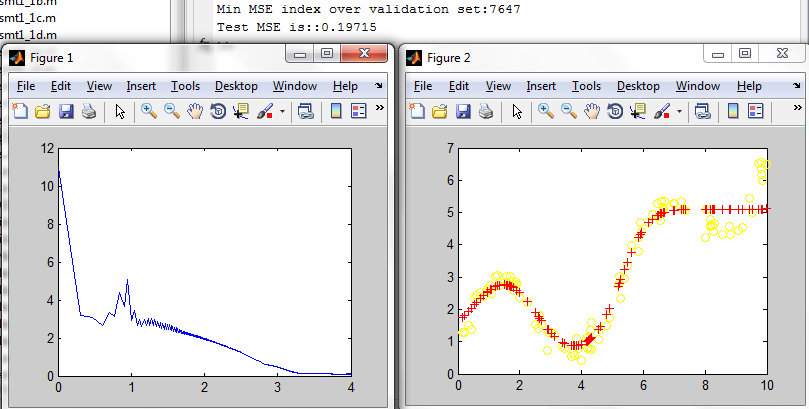
**Nearest Neighbor:** trust the shape/outline of the curve – the dense patches of data are redundant.

1. h) Neural networks: Early stopping. For k=10, generate and *submit a single plot* that includes both the training error and the validation error versus the number of epochs. Determine the number of epochs that achieves the lowest validation error and *report this value*. *Report the test* *error* for this model and discuss how this value compares to those from (1a) to (1g). [no random restarts]



An early stop at iteration 8316 (of 10000) trained on the validation data yielded a **test MSE of 0.21412.**

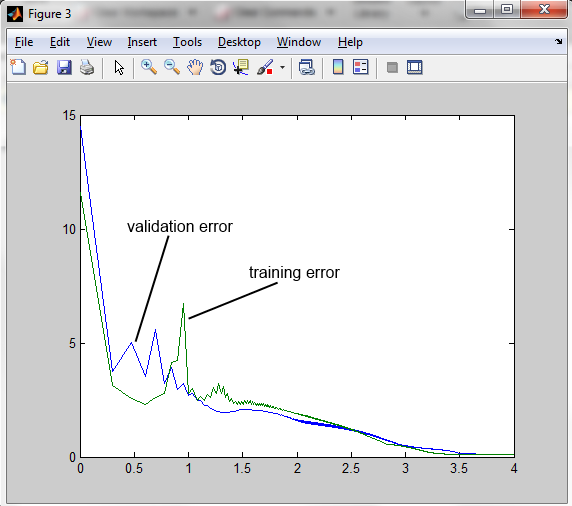
This is ever so slightly better than the above result. Let’s repeat the early stopping experiment for the training data:



An early stop at iteration 7647 (of 10000) trained on the training data yielded a **test MSE of 0.19715**.

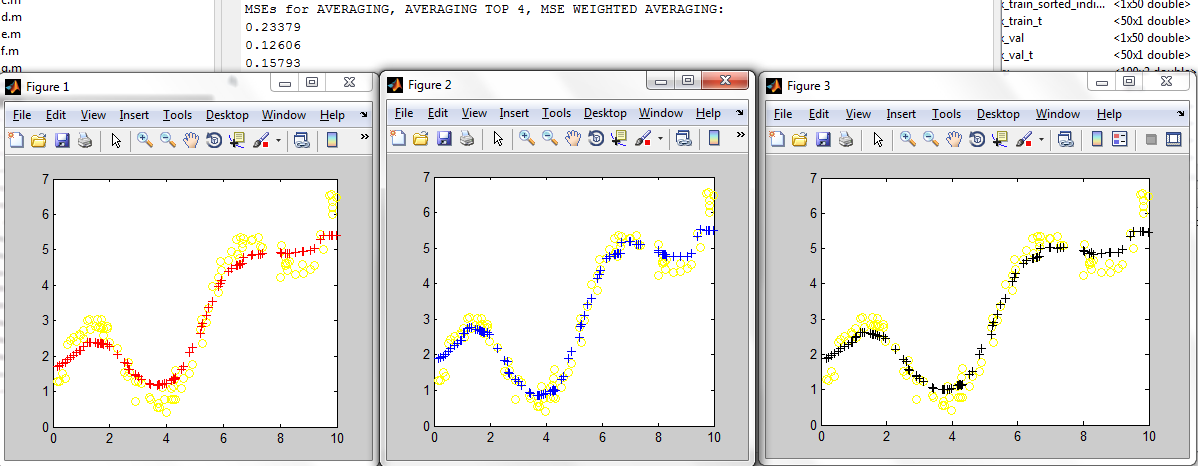
The early stopping with the training data was the second best neural network result so far, second only to the random restart method which slightly outperformed it with a test MSE of 0.19201. Early stopping can only help given that the error curve is not monotonic (i.e. the learning rate is high enough to induce bumps and irregularities as a trade-off for possible faster convergence).

Random restarts can be superior because by starting with different initial system weights, the system can try converging to a solution which *happens* to bias outliers in the training data that better represents the test data (for an example, see the above question which discussed the validation data).



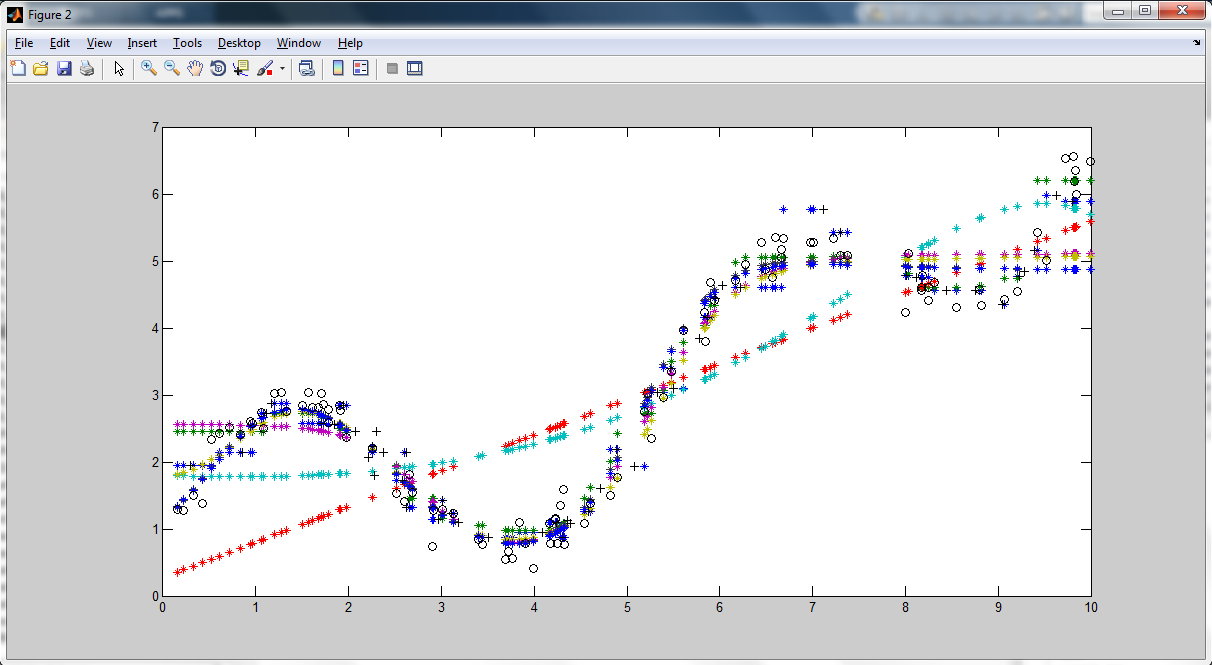
This figure includes the validation error and training error vs. (log10) the epoch number.

1. i) Ensembles. Create an ensemble predictor using all of the above models (including different models obtained from different random restarts, etc) by averaging their predictions. Evaluate the ensemble on the test set and *report the test error*… *Report the test error* of the more selective ensemble method. Finally, when averaging predictions, use a weighted average where the weight for a predictor is higher if it has a lower validation error. *Report the test error* of the weighted ensemble.



The leftmost figure shows **all 8 models averaged together, with an MSE of 0.23379**. The middle figure shows the **top 4 models averaged together, with a test MSE of 0.12606 (the lowest)**. The rightmost figure shows **all 8 models averaged by weighting according to approximately 1/MSE and achieved a** **test MSE of 0.15793.**

1. j) In a single figure, *plot the predicted value for y against x*, for each of the methods (a) to (i), using the best of the three ensemble predictors from (i). There should be *nine curves* in the plot. In the same figure, plot the training data using markers ‘+’ and test data using markers ‘o’. *Discuss* how the different curves match the training and test data and which ones are more or less overfit to the training data.



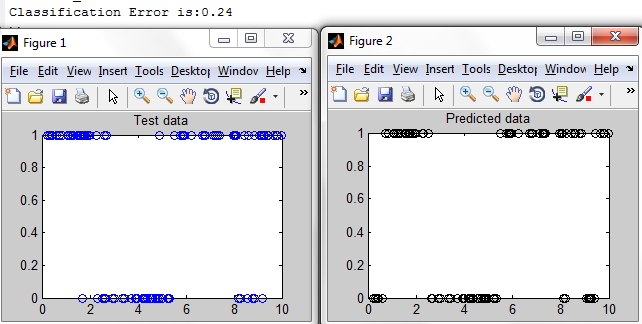
The 9 curves with a training data (+) and test data (o) for 11 curves total.

The linear regression curves (trained using gradient descent) underfit the data, as do the neural networks with less than 5 hidden logistical units. Least squares regression overfits the data by achieving the lowest MSE of any model as found in question 1.d).

With more than 5 logistic hidden units, the neural networks may be overfitting the data. Empirically, each additional logistic hidden unit seems to be able to “capture” a different monotonic section of the curve. Following the test data, there seems to be (eyeballing it) 5 changes in slope of the function y = f(x). Thus adding more degrees of freedom to the model may end up overfitting.

As discussed in question 1.g), MSE-minimizing fitting procedures tend to favor data density over curve shape, while the nearest neighbor fits trust the data shape while treating data density as redundancy.

2.a) Nearest neighbors. Use the nearest neighbor method to make label predictions for the 100 test cases using the 50 training cases. *Compute and report* the classification error rate on the test set. Also, *compute and report* the 2 x 2 confusion matrix for the test data.



The test data is on the left, with the predicted data on the right. **The classification was found to be 0.24.** This means 24 data points out of 100 test data points where classified incorrectly.

Confusion Matrix:

>> C = confusionmat(predicted\_y\_vector, y\_test)

C =

31 16

8 45

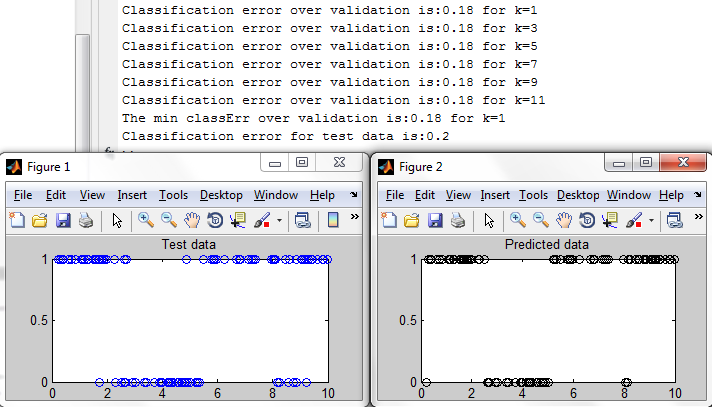
Where: (as will be convention from here on)

C(1,1) = # true positives C(1,2) = # false negatives

C(2,1) = #false positives C(2,2) = # true negatives

classErr = [C(1,2) + C(2,1)] / 100 = 0.24 for this example.

2. b) k-nearest neighbors: Voting. For k=1,3,5,…,11, use the k-nearest neighbor method to make predictions for the 50 validation cases, using voting to decide the most likely label. *Compute and report* the 10 corresponding values of the validation error rate. *Which value of k achieves the lowest validation error?* Use that value of k to make predictions for the 100 test cases and *report the error rate* on the test set. *Discuss how this value compares* to the error rate reported in (2a). *Report* the 2 x 2 confusion matrix for the test data.



The classification error was minimized over the validation data for **all (odd) k up to 11**. Using this parameter, **the classification error was found to be 0.2.** This result is **better** than the above. Interesting to note graphically are the group of false positives in the predicted data (on the right, from x = 8 to 9.5) due to the influence of all the positive close-by neighbors.

The confusion matrix is as follows:

C =

26 7

13 54

2. c) k-nearest neighbors: Probabilities. Modify your software so that instead of using the k-nearest neighbors to vote on the label, they are used to estimate the probabilities that the label is 0 or 1, using the proportion of labels with the value 0 and 1. Include a pseudocount of 0.1 in each class to prevent zero-probabilities; for example, for k=3 if there are two 0’s and one 1, the probabilities of 0 and 1 would be 2.1/3.2 and 1.1/3.2 (note that these sum to one!) For the above values of k, *compute and report* the log-likelihood of the validation set. *Which value of k* *achieves the lowest validation error, and is it the same as the value of k found using voting?* Use that value of k to make predictions for the 100 test cases and *report the test log-likelihood* and also the *test classification error rate*.

For this problem, we must associate with each model, call them k, a (log-)likelihood function, l(k). Our goal in this maximum likelihood learning will then be to select the model k that maximizes the likelihood function of the data. Start with the likelihood function L(k):

L(k) = Product(P(y(t) | x(t), k)) for training/validation cases t = 1, …, T

For binary labels, the probability of y = 1 for a given x, k (input and model) is given by:

p = Pr(y = 1 | x, k) = C1 / (C0 + C1)

and 1 –p = Pr(y = 0 | x, k) = C0 / (C0 + C1)

Where C0 and C1 are the neighbor counts of nearby 1s and 0s, plus the additional pseudocounts of 0.1.

The probability of any y at test case t (each y having a binary label) can be written as one expression:

P(y(t)| x(t), k) = [p(t)^y(t)]\*[(1-p(t))^(1-y(t))]

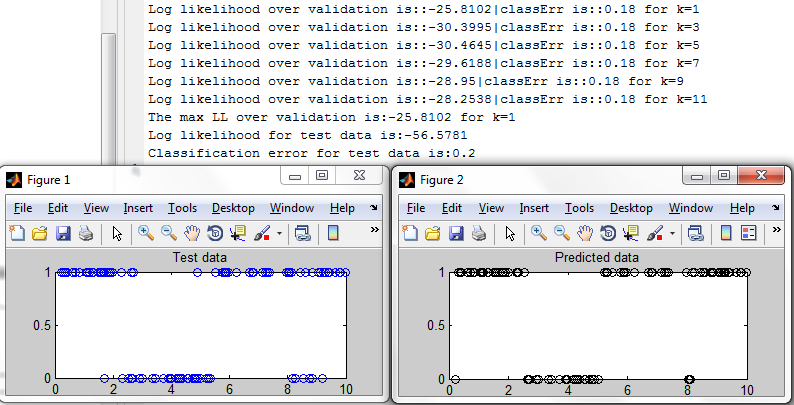
Which is a convenient closed-form expression analogous to that in logistical regression. Computationally, our log-likelihood function can be expressed as:

l(k) = log(Product(P(y(t) | x(t), k)))

l(k) = sum(y(t)\*log(p(t)) + (1-y(t))\*log(1-p(t))), the sum extending over all training cases t = 1, …, T

The NearestNeighborsK function was modified to be NearestNeighborsKprobability to help with this task, and the script in asmt1\_2c implemented the calculating and comparison of l(k) for k = 1,3,…,11.

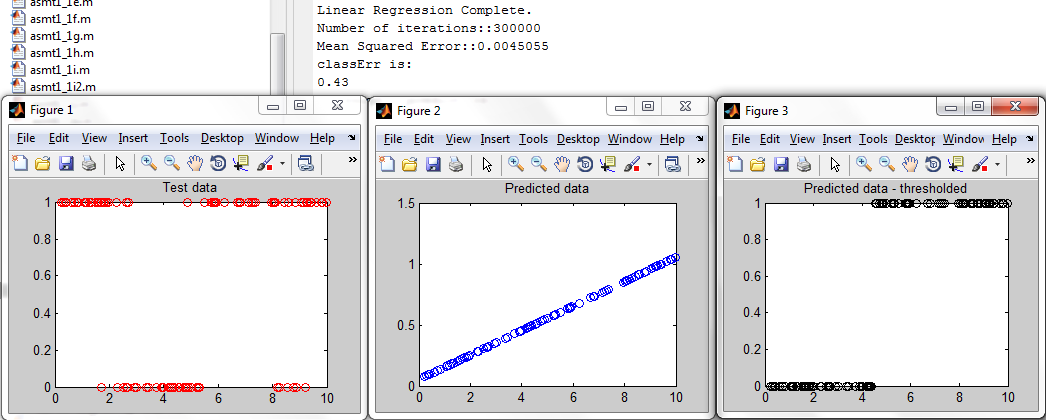
Results on the next page:



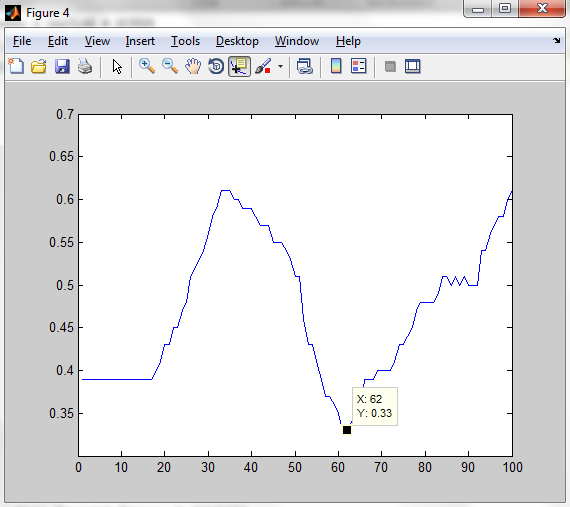
The log-likelihood of the validation set for various k’s is reported in the above figure. The validation error was the same for all odd k’s up to 11 (rerunning the experiment, I see this error increase to 0.44 for k = 17 and up, and then back down to 0.34 for k = 33 to 45). The voting procedure found the same classification errors as this probabilistic procedure. Interesting to note, however, is the validation log-likelihood is greatest for k = 1, meaning that even given the same classification error as other models, we may be more confident in the k = 1 model than say the k = 7 model.

The **test log-likelihood was -56.5781 with a classification error of 0.2**.

2. d) Linear regression. We can cheat and pretend that the labels 0 and 1 are real numbers. Use the steepest descent method to fit a linear regression model (two parameters) to the training data. Apply this model to the test data. The resulting predictions will be real numbers. Apply a threshold of 0.5 to obtain binary predictions and *report the classification error rate*. Compare to the rates obtained in (2a) to (2c). To test the effect of the threshold, repeat the above procedure for a range of thresholds and *plot the test error rate versus the threshold*.



The two parameter model fit to the training data is shown in figure 2, with its threshold version in figure 3. **The classification error is 0.43**. It is obvious from the test data in figure 1 that a 2-param model can only fit bimodal data, while the test data itself has at least 3 modes.



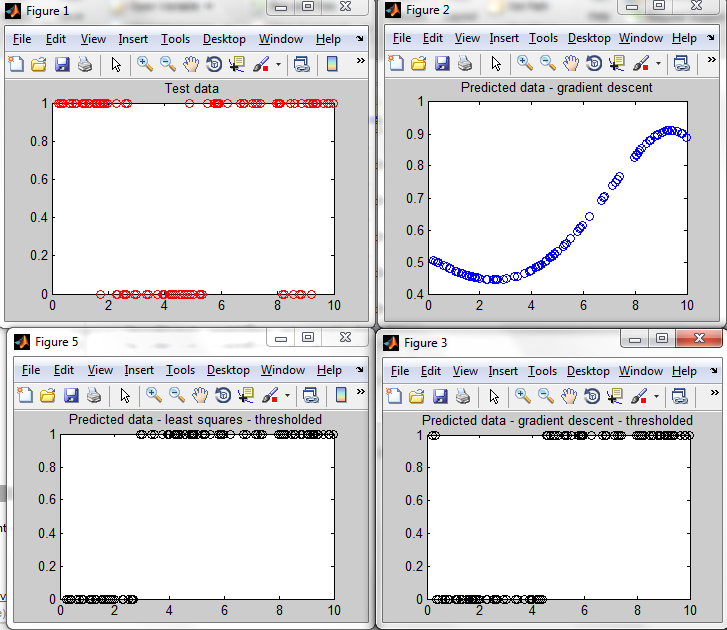
This figure plots test error (y-axis) vs. threshold\*100 on the x-axis. For the given model, it is minimized to a value of 0.33 when the threshold is set to 0.62.

2.e) Linear regression using polynomial inputs. Expand the input x to form a vector of inputs (x, x2, x3, …, xk), for a positive integer k. Use steepest descent to fit linear regression models to the training data, for k ranging from 1 to 10. For each model, threshold the predictions for the validation cases and compute and *report the 10 corresponding values of the validation error* *rate*. *Which value of k achieves the lowest validation error?* Use that value of k to make predictions for the 100 test cases and *report the test error rate*. *Discuss how this value compares* to the error rates reported in (2a) to (2d).

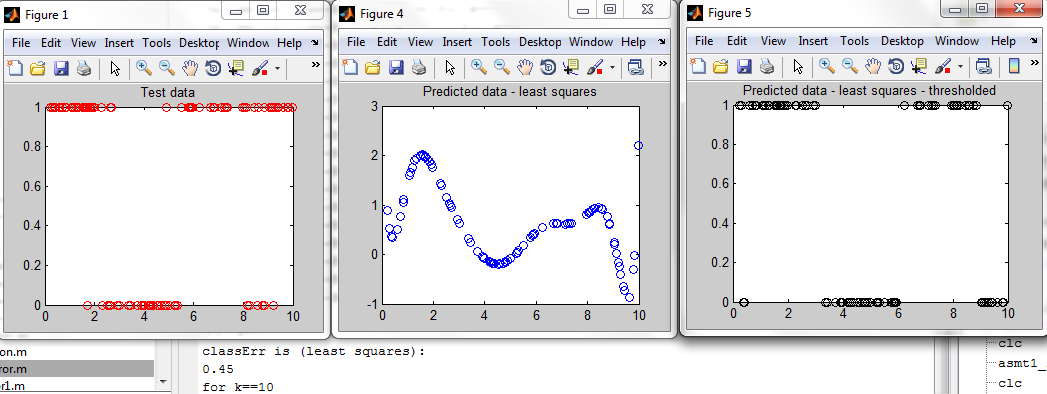
Validation error rates were found to be as follows (note: I also ran the least squares algorithm for comparison):

Validation classErr is::0.56 for k==1  
Validation classErr\_ls is::0.51 for k==1  
Validation classErr is::0.56 for k==2  
Validation classErr\_ls is::0.51 for k==2  
Validation classErr is::0.56 for k==3  
Validation classErr\_ls is::0.51 for k==3  
Validation classErr is::0.56 for k==4  
Validation classErr\_ls is::0.51 for k==4  
Validation classErr is::0.56 for k==5  
Validation classErr\_ls is::0.51 for k==5  
**Validation classErr is::0.54 for k==6** 🡺 gradient descent lowest validation error  
Validation classErr\_ls is::0.51 for k==6  
Validation classErr is::0.54 for k==7  
Validation classErr\_ls is::0.51 for k==7  
Validation classErr is::0.54 for k==8  
Validation classErr\_ls is::0.51 for k==8  
Validation classErr is::0.54 for k==9  
Validation classErr\_ls is::0.51 for k==9  
Validation classErr is::0.54 for k==10  
Validation classErr\_ls is::0.51 for k==10  
**classErr is (gradient descent):   
0.5 for k==6** 🡺 the k=6 polynomial fit, when threholded, was chosen as the best fit, achieveing a classification error of 0.5.  
classErr is (least squares):   
0.55 for k==1

The accompanying graphs with labels are on the following page:

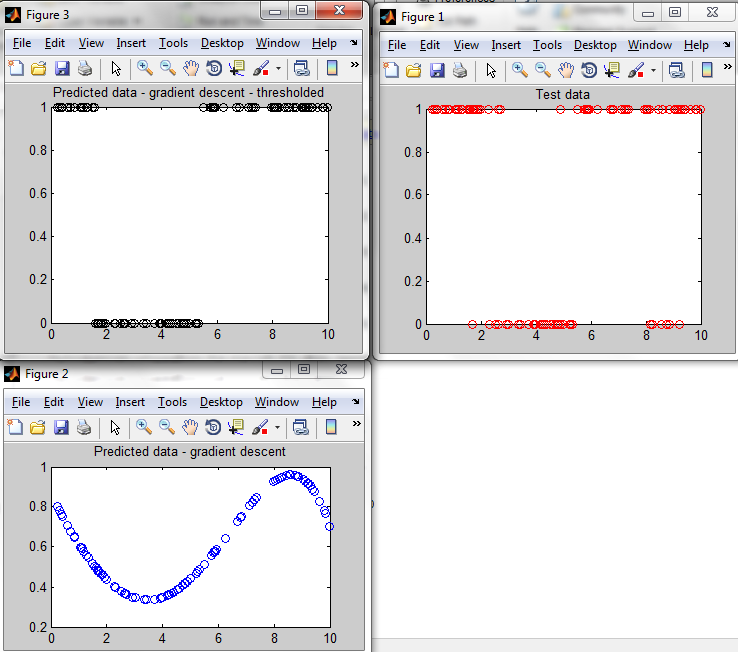


The upper right curve of the k=6 fit begins to fit a 3-phase data regime, more in-line with the test data. The bottom left curve shows thresholded least squares results for k = 1. Below we see the results for the best least squares result at k = 10:



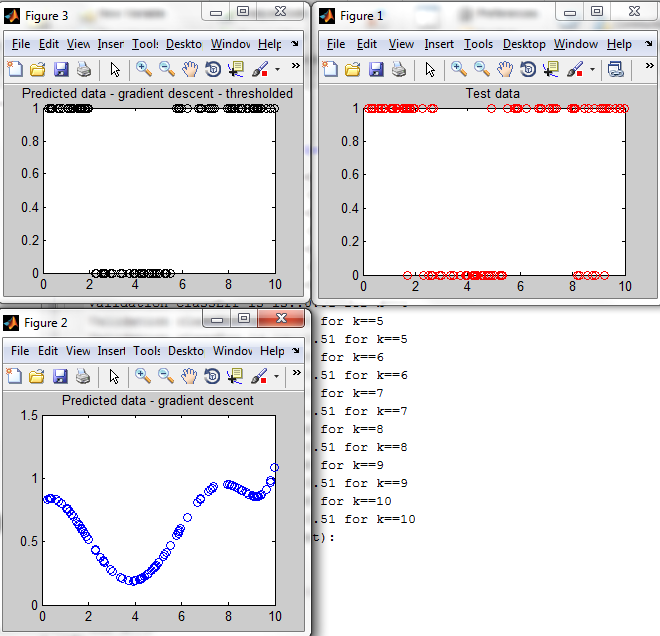
The above figure shows the k = 10 LS curve fit and its thresholded predictions. It achieved a test classification error of 0.45, but an impressive validation classification error of 0.11. Clearly LS overfits in this case.

Note that all the regression results are for 10,000 iterations. For the iterative gradient descent, perhaps using a different scoring metric and having more iterations would better fit the 3-phase test data. In fact, I re-ran the test for 100,000 iterations and found a k=5 curve doing just this, but the thresholded classification test error was 0.52 (larger than above); however the validation classification error was 0.48, the lowest yet for a linear regression method.



The predicted gradient descent curve – and its threshold – better capture multiple data groupings (> 2).

Here’s one more look for 1,000,000 iterations; a k=7 curve was chosen with a classification error of 0.43 and validation error of 0.44 (and this time there were no ties to 2 decimal places):



One might imagine that given enough iterations and enough polynomial features, the 3-4 data groupings of the test data could be captured pretty well. However for now, all the regression results have been inferior to nearest neighbor results.

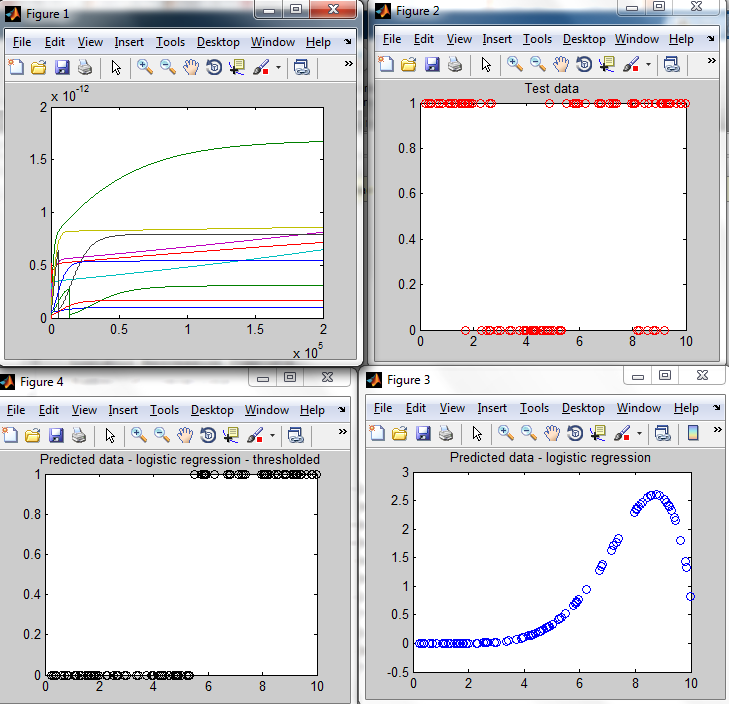
A trade-off that was noticed: minimizing validation classification error clearly didn’t always lead to better test classification error. Regression can overfit.

2. f) Logistic regression using polynomial inputs. Use steepest descent to fit a logistic regression model to the training data, using polynomial inputs with k ranging from 1 to 10. *Submit a figure* that plots the log-likelihood of the training set versus the number of epochs, for each value of k (there will be 10 curves in the plot). For each model, compute and *report the 10 corresponding* *values of the validation log-likelihood*. *Which value of k achieves the lowest validation loglikelihood?* Use that value of k to make predictions for the 100 test cases and *report the test loglikelihood*. *Discuss how this value compares* to the log-likelihood reported in (2c). Threshold the predictions using a probability of 0.5, and *report the test classification error rate*. *Discuss how* *this value compares* to the error rates reported in (2a) to (2e).

Here are the main results:

Validation classErr is::0.5||LL==-352.6957 for k==1  
Validation classErr is::0.36||LL==-199.4749 for k==2  
Validation classErr is::0.4||LL==-386.7074 for k==3  
Validation classErr is::0.4||LL==-369.8299 for k==4  
Validation classErr is::0.46||LL==-341.9668 for k==5  
Validation classErr is::0.44||LL==-189.9257 for k==6  
**Validation classErr is::0.4||LL==-152.789 for k==7**  
Validation classErr is::0.42||LL==-177.1891 for k==8  
Validation classErr is::0.4||LL==-210.2363 for k==9  
Validation classErr is::0.44||LL==-238.1303 for k==10  
**Test classErr is::0.33||LL==-394.6577 for k==7**

Figures on the next page:



In the top left we see likelihood (log-likelihood was too sharp a curve to glean any insight from visually), and almost all the models’ likelihoods are monotone increasing, except for k=7 and k=9. In my software for logistic regression, I check to see if the gradient descent step doesn’t increases the LL; if it does, I re-initialize weights and try iterating again at a lower learning rate. I’ve found this to greatly help with regression convergence, or at least making sure the program doesn’t get stuck in the wrong direction.

Another source of numerical frustration was in calculating *log*-likelihood: in evaluating expressions like log(1-p), if p was >=1, the logarithm would spit out an imaginary number, and an imaginary log-likelihood implies a negative probability. Thus I added fudge factors, e.g. log(1-p) 🡪 log(1-p+e), where e is something small like 1e-10. In practice this only shifts the log-likelihood function; it still behaves the same and thus serves as a comparative yardstick when evaluating different parameterized models.

The test classification error results are better than previous regression results, but still inferior to nearest neighbor methods.

2.g) Neural network classifiers: Real-valued outputs. As with linear regression, we can pretend that the labels 0 and 1 are real numbers and train a neural network that has a real-valued output. Repeat parts (1e) to (1h) from Question 1, using the classification data instead of the regression data. Wherever it is requested that you report the MSE, *report the MSE* and *also report the* *classification error rate*, by applying a threshold of 0.5 to output of the neural network. Compare the error rates to those reported in (2a) to (2f).

Results are given in terms of classErr\_2g\_e, …, classErr\_2g\_h (classification errors for experiments e thru h), MSE\_2g\_e, …, MSE\_2g\_h (and their respective test MSEs):

y\_predict\_2g\_e: classErr --0.17 -- MSE --0.13664

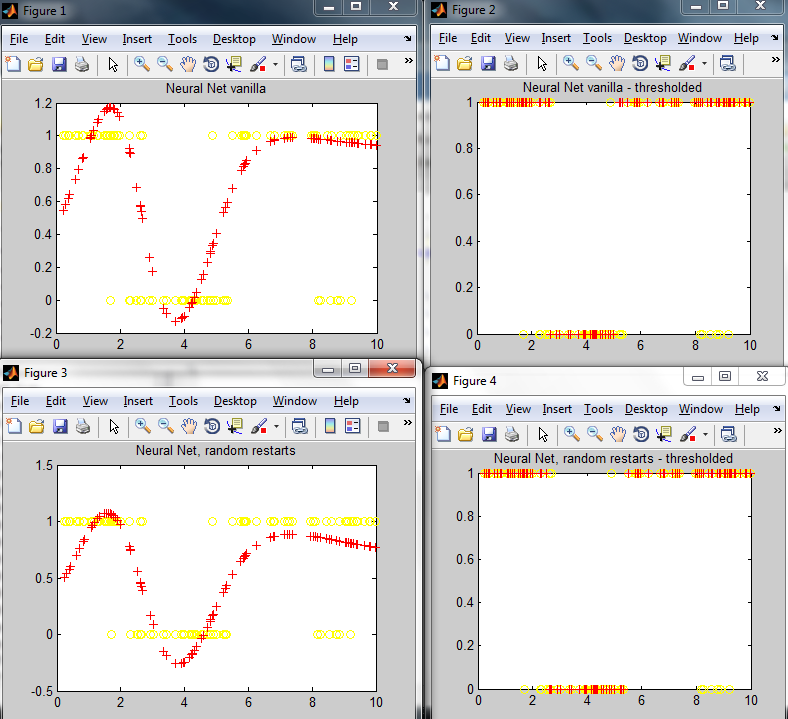
**y\_predict\_2g\_f: classErr --0.13 -- MSE --0.12329**

y\_predict\_2g\_g: classErr --0.39 -- MSE --0.24008

y\_predict\_2g\_h: classErr --0.39 -- MSE --0.24009

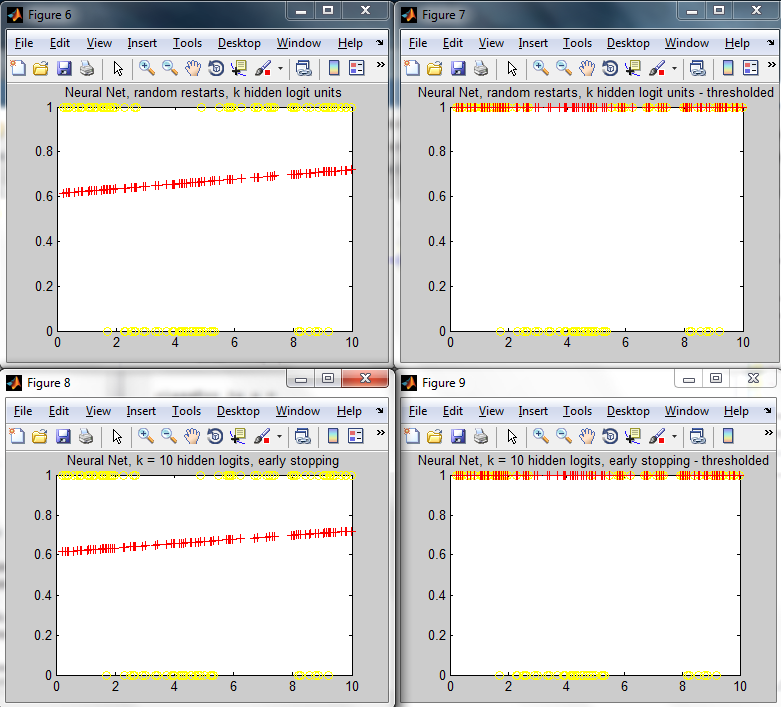
The experiments were run for 10,000 iterations each. Neural network f , random restarts, was found to have a classification error of **0.13**, **the best result yet** from any of the previous classifier models. Its **test MSE was found to be 0.12329**.

Relevant figures are included below:



This figure shows experiments 1e and 1f fits (top and bottom figures respectively). As all models show, the NNs are good at capturing the 3 distinct groupings of data. Perhaps a model of this type is the best one can reasonably do, as capturing the sparser negative grouping around x = 8 to 9.5 would involve a high-degree curve with wild oscillations around those points – which would almost certainly be a case of overfitting.

Experiments g and h, ran over validation data, were not able to do very well over the validation data with a classification error of 0.39; both models chose k = 10 hidden logistic units. This is inferior to logistic regression but superior to linear regression; also inferior to all nearest neighbor models. Their figures are plotted below:



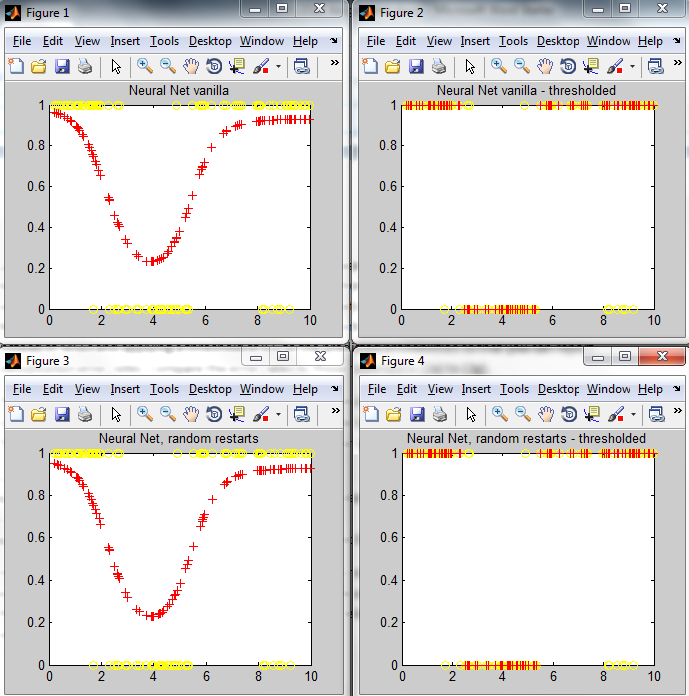
2.h) Neural network classifiers: Probabilistic outputs. Instead of a linear output unit, we can use a logistic output unit so that the output is a real number between 0 and 1. This results in a neural network version of logistic regression (see 2f). Implement a steepest descent method that maximizes the log-likelihood of the training data. Repeat parts (1e) to (1h) from Question 1, *reporting log-likelihood values* instead of MSEs and applying a threshold of 0.5 to the predicted probabilities so that you can *report classification error rates*. Compare the error rates to those reported in (2a) to (2g).

The NeuralNetwork\* programs were modified to have sigmoidal output neurons (and their corresponding error derivatives were modified appropriately). These methods reported loglikelihoods. The following results were obtained, using an analogous notation to the above:

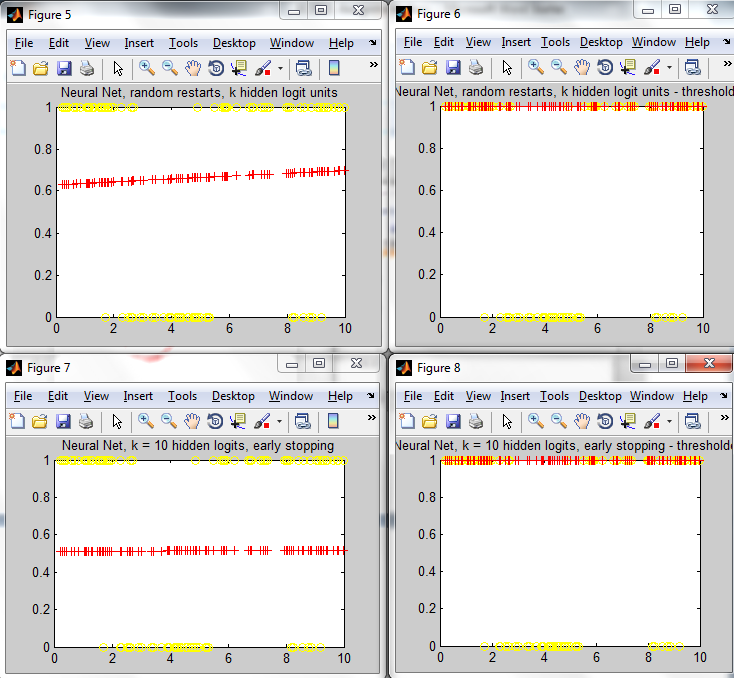
y\_predict\_2h\_e: classErr --0.12 -- LL ---42.3829  
**y\_predict\_2h\_f: classErr --0.12 -- LL ---42.4637** 🡪 lowest classification error (w/ slightly higher LL)  
y\_predict\_2h\_g: classErr --0.39 -- LL ---67.3272  
**y\_predict\_2h\_h: classErr --0.39 -- LL ---68.7246 🡪** the most likely model according to LL

Interesting to note here is that the most likely model does not correspond to the lowest classification error. As the next figures show, these more “likely” models with worst classification rates threshold all their decisions to 1, and their pre-thresholded curves are relatively flat and don’t attempt to fit the data very much.

The random restart NN with probabilistic output has the lowest test classification error of any model yet, including when compared to nearest neighbors.



And the more likely models:



From the comparison above we can gain the following insights on choosing a model for likelihood maximization over classification / mean-squared error (at least in the case were MSE minimization and LL maximization aren’t equivalent, i.e. not over Gaussians):

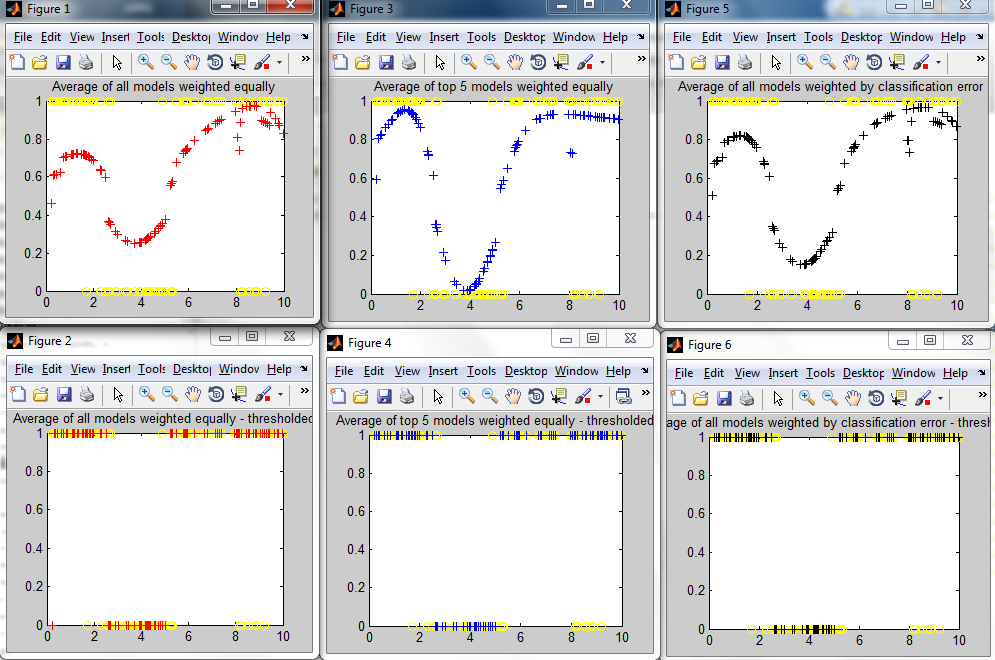
Likelihood maximization is, roughly speaking, about finding a model that’s more conservative with respect to fitting the whole dataset; it’s probably not too right, not too wrong – it underfits. Error minimization, on the other hand, is about bending parameters to find a combination of them which match training data – and with enough iterations and parameters, it’s not hard to imagine overfitting occurring. Note that these intuitive findings are entirely consistent with the experimental results in “Bayesian Prediction” revised lecture notes of ECE521.

Note that the example model fit of y = 0.5 (approximately) that came about due to the k = 10 model not moving much for the 10,000 iterations is a somewhat pathological case of the above example.

The choice of metric with which to choose a model is critically dependent on the application domain, as concerns regarding potential overfitting or underfitting arise.

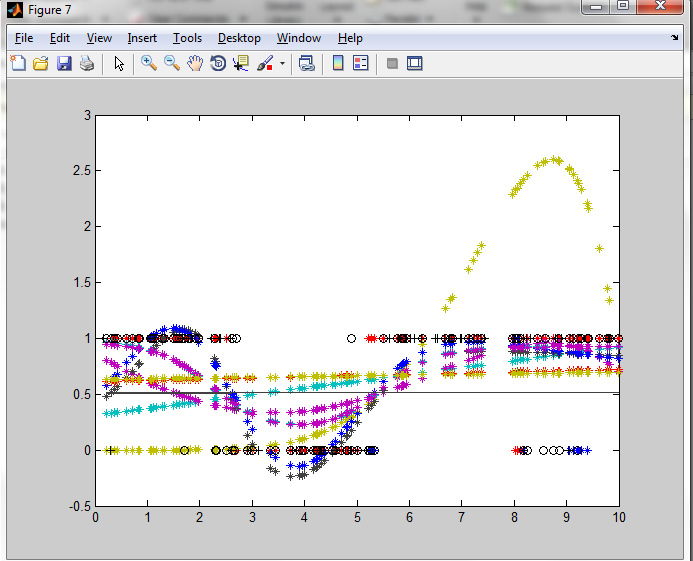
i) Ensembles. As in (1i), build ensemble predictors and report the test classification error. Explore different ways of averaging, eg, averaging binary classification decisions and averaging real-valued outputs (make sure to scale the outputs sensibly so that averaging different predictors makes sense, eg, linear regression produces unbounded outputs, whereas logistic regression produces numbers in [0,1]).

Similar to in question 1, we will explore the averaging of models, taking the top couple of models and averaging them, as well as weighting by a metric – the classification error in this case. Note that averaging will be done by raw model values (including those given as 1/0s already, e.g. nearest neighbors) and then applying the threshold afterwards.



The above figure shows the results of these combinations of models, and their threshold visualizations. Their respective classification error rates were 0.17, 0.16, and 0.16 respectively. These results are second only to the top model, the neural network probabilistic output with random restarts.

2. j) In a single figure, for each of the methods above, *plot curves* for the predicted probability of the label 1 against x. Include the training and test data in the figure, as described in (1j). *Discuss* *the different predictors*.



These are the probability functions over the test cases for all models created. Note that a few of the models (the ones using real-valued outputs) go outside of a PDF range of [0,1] but in practice the were clamped to obtain the prediction.

Relevant discussion pertaining to the different types of predictors has been included throughout this assignment; e.g. 2.h) compares models maximizing likelihood versus those than minimize error. The best models were able to capture 3 distinct groupings of the data. None of the models except for nearest neighbors were able to handle the few interspersed negative samples from x = 8 to 9.5. Other models not fitting this may be interpreted as treating these training examples as noise or incorrect samples from an experiment.

In conclusion, this assignment was worthwhile in the practical coding experience gained as well as the consolidation of knowledge needed to craft the software, particularly the creation of likelihood functions in part 2. However, it my opinion that the assignment was far too long giving the meaningfulness (or lack-thereof) of the test data, and its connection to any real-world machine learning data sets. Stated differently, given that I spent over 60 hours creating machine learning algorithms, it would be nice to have more to show than some fitted X/Y data points that are solved trivially using least squares.

* James Schuback