ABE598 Autonomous Decision Making: Problem Set 2

# Problem 1 – Regression Problem with Wing Rock Dynamics

Collecting a dataset: the data was simulated using file wingrock\_main.m using the following initial conditions x = [x1; x2]

|  |  |  |
| --- | --- | --- |
| Test | x1 | x2 |
| 1 | 1.2 | 1 |
| 2 | 3 | .6 |
| 3 | 5 | 1 |
| 4 | .3 | 1.3 |
| 5 | 3.8 | .5 |

The weights were first learned in the file hw1\_1\_1.m using the Matlab regress function on the input-output data collected from the simulations. This method found the basis

b = [0.798955, 0.232291, 0.689129, -0.623908, 0.013385, 0.021379]

this is very close to the actual basis of the system dynamics

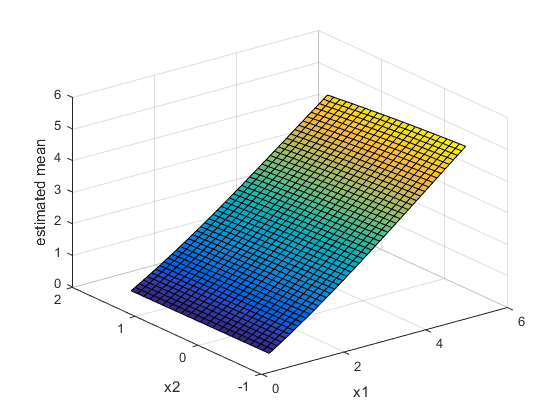
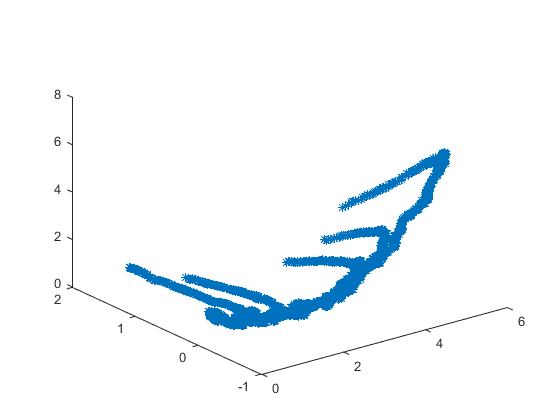
bactual = [.8, .2314, .6918, -.6245, .0095, .0214]

The file hw1\_1\_3.m then implements the GP regression to learn the outputs of the system given the inputs. Using the same data as above I find the estimated state given input data to learn. The MSE between the predicted state and the actual state depends on the range over which x1 and x2 are evaluated. The best predictions occur when the range is small, due to the relatively limited amount and variety of data put into the GP model. Therefore, when the ranges are set at

x1\_range = min(X1\_IN):.1:max(X1\_IN);

x2\_range = min(X2\_IN):.1:max(X2\_IN);

the MSE = 1.9321, and the resulting figures are

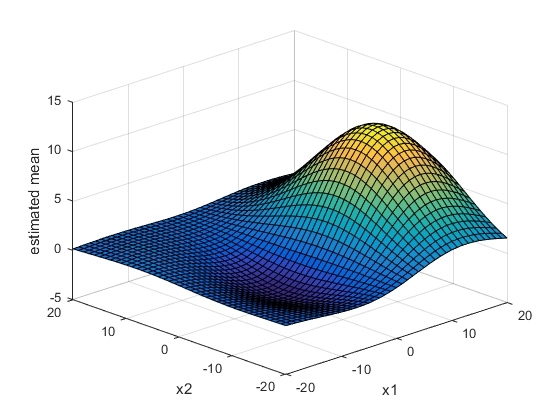


For a plot over a larger range

x1\_range = -20:1:20;

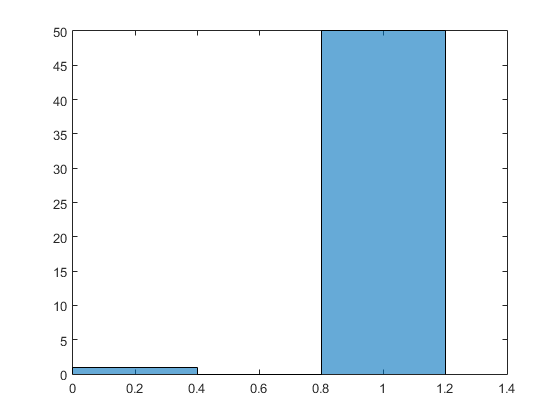
x2\_range = -20:1:20;

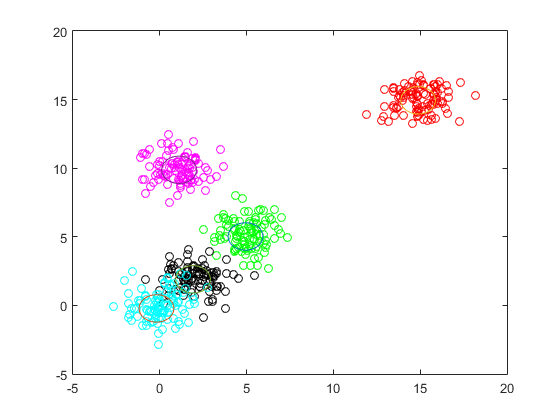
The MSE becomes significantly worse, at MSE = 1.1628e+04, and the figure the program generates looks like the following



# Problem 2 – Clustering with K-Means

Over 20 runs, the algorithm implemented takes an average of 25.2389 seconds to run. The algorithm almost always converged to the correct clusters, as shown in the histogram over 51 runs.



The figure below shows an example of the excellent convergence properties of the algorithm.

# Problem 3 – Dirichlet Means

# Problem 4 – Neural Networks

The neural network used is implemented in the python file ‘single\_hidden\_layer\_neural\_net.py’. A vectorized version was attempted in ‘vectorized\_nn.py’, but I don’t have time to debug for the due date.

The percent change between the cost of the first epoch and the cost of the final epoch is used to show the performance of the neural net. These values were taken from a run of training. Each run was over 50,000 epochs, I made no attempts to tune the system hyperparameters to change performance, and techniques such as data normalization were not implemented.

|  |  |
| --- | --- |
| # layer 2 neurons | Percent change |
| 4 | 80.59 |
| 6 | 52.26 |
| 8 | 47.08 |
| 10 | 34.96 |
| 12 | 20.80 |
| 14 | 18.98 |
| 16 | 8.11 |

The convergence does depend on the initial weights in general, since the backprop algorithm relies on gradient descent. However, by choosing an appropriate convex cost function, this problem can be avoided.

# Problem 5 – Deep Neural Network

The neural network was implemented in the iPython Notebook. The results for 100 epochs (this took about 10 minutes, yay GPUs!) are presented below. A bug was discovered in doing this problem where the iPython Notebook file could not be copied to a different directory without destroying all work done in the notebook. Basically, I did the problem as assigned and it went well, but afterward I tried to copy the file to a “turn in” directory, and the copied version of the notebook only gives a 404 page.

The accuracy of the system is great, and it was a straightforward problem to get done. 