Machine Learning from Data HW12

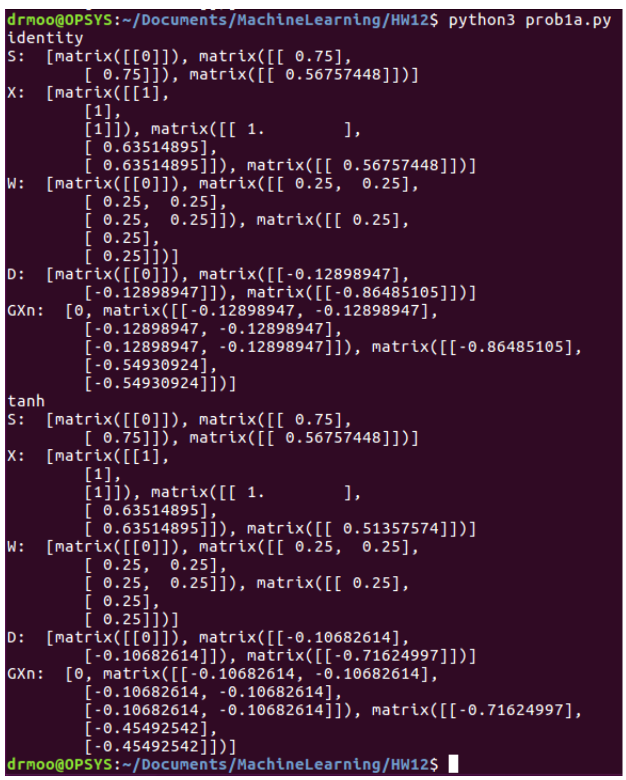
Shane O'Brien

December 2017

Neural Networks and Backpropagation

a

I constructed the neural network with both the identity and tanh output node transformation functions. Below is a picture of the result in terminal:



```
drmoo@OPSYS:~/Documents/MachineLearning/HW12$ python3 prob1a.py
identity
S:  [matrix([[0]]), matrix([[ 0.75],
        [ 0.75]]), matrix([[ 0.56757448]])]
X:  [matrix([[1],
        [1],
        [1]]), matrix([[ 1.        ],
        [ 0.63514895],
        [ 0.63514895]]), matrix([[ 0.56757448]])]
W:  [matrix([[0]]), matrix([[ 0.25,  0.25],
        [ 0.25,  0.25],
        [ 0.25,  0.25]]), matrix([[ 0.25],
        [ 0.25],
        [ 0.25]])]
D:  [matrix([[0]]), matrix([[-0.12898947],
        [-0.12898947]]), matrix([[-0.86485105]])]
GXn:  [0, matrix([[-0.12898947, -0.12898947],
        [-0.12898947, -0.12898947],
        [-0.12898947, -0.12898947]]), matrix([[-0.86485105],
        [-0.54930924],
        [-0.54930924]])]
tanh
S:  [matrix([[0]]), matrix([[ 0.75],
        [ 0.75]]), matrix([[ 0.56757448]])]
X:  [matrix([[1],
        [1],
        [1]]), matrix([[ 1.        ],
        [ 0.63514895],
        [ 0.63514895]]), matrix([[ 0.51357574]])]
W:  [matrix([[0]]), matrix([[ 0.25,  0.25],
        [ 0.25,  0.25],
        [ 0.25,  0.25]]), matrix([[ 0.25],
        [ 0.25],
        [ 0.25]])]
D:  [matrix([[0]]), matrix([[-0.10682614],
        [-0.10682614]]), matrix([[-0.71624997]])]
GXn:  [0, matrix([[-0.10682614, -0.10682614],
        [-0.10682614, -0.10682614],
        [-0.10682614, -0.10682614]]), matrix([[-0.71624997],
        [-0.45492542],
        [-0.45492542]])]
drmoo@OPSYS:~/Documents/MachineLearning/HW12$
```

Or, to stress the important parts:

Identity Output

$$x_2 = 0.5675$$

$$g_1 = \begin{bmatrix} -0.1289 & -0.1289 \\ -0.1289 & -0.1289 \\ -0.1289 & -0.1289 \end{bmatrix}$$

$$g_2 = \begin{bmatrix} -0.8648 \\ -0.5493 \\ -0.5493 \end{bmatrix}$$

Tanh Output

$$x_2 = 0.5135$$

$$g_1 = \begin{bmatrix} -0.1068 & -0.1068 \\ -0.1068 & -0.1068 \\ -0.1068 & -0.1068 \end{bmatrix}$$

$$g_2 = \begin{bmatrix} -0.7162 \\ -0.4549 \\ -0.4549 \end{bmatrix}$$

b

I perturbed the weights to 0.2501 and got this result:

```
drmoo@TheOffice2:~/Documents/MachineLearning/HW12$ python3 prob1a.
py
identity
S:  [matrix([[0]]), matrix([[ 0.7503],
        [ 0.7503]]), matrix([[ 0.56789101]])]
X:  [matrix([[1],
        [1],
        [1]]), matrix([[ 1.        ],
        [ 0.63532789],
        [ 0.63532789]]), matrix([[ 0.56789101]])]
W:  [matrix([[0]]), matrix([[ 0.2501,  0.2501],
        [ 0.2501,  0.2501],
        [ 0.2501,  0.2501]]), matrix([[ 0.2501],
        [ 0.2501],
        [ 0.2501]])]
D:  [matrix([[0]]), matrix([[-0.12889747],
        [-0.12889747]]), matrix([[-0.86421797]])]
GXn:  [0, matrix([[-0.00042966, -0.00042966],
        [-0.00042966, -0.00042966],
        [-0.00042966, -0.00042966]]), matrix([[-0.00288073],
        [-0.00183021],
        [-0.00183021]])]
tanh
S:  [matrix([[0]]), matrix([[ 0.7503],
        [ 0.7503]]), matrix([[ 0.56789101]])]
X:  [matrix([[1],
        [1],
        [1]]), matrix([[ 1.        ],
        [ 0.63532789],
        [ 0.63532789]]), matrix([[ 0.51380874]])]
W:  [matrix([[0]]), matrix([[ 0.2501,  0.2501],
        [ 0.2501,  0.2501],
        [ 0.2501,  0.2501]]), matrix([[ 0.2501],
        [ 0.2501],
        [ 0.2501]])]
D:  [matrix([[0]]), matrix([[-0.10674226],
        [-0.10674226]]), matrix([[-0.71567409]])]
GXn:  [0, matrix([[-0.00035581, -0.00035581],
        [-0.00035581, -0.00035581],
        [-0.00035581, -0.00035581]]), matrix([[-0.00238558],
        [-0.00151563],
        [-0.00151563]])]
drmoo@TheOffice2:~/Documents/MachineLearning/HW12$
```

As expected, the output is very similar. The gradient, however, changes quite a bit. These are the important numbers:

Identity Output

$$x_2 = 0.5678$$

$$g_1 = \begin{bmatrix} -0.0004 & -0.0004 \\ -0.0004 & -0.0004 \\ -0.0004 & -0.0004 \end{bmatrix}$$

$$g_2 = \begin{bmatrix} -0.0029 \\ -0.0018 \\ -0.0018 \end{bmatrix}$$

Tanh Output

$$x_2 = 0.5138$$

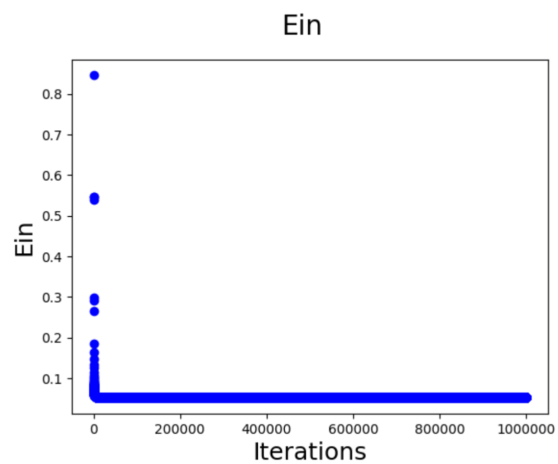
$$g_1 = \begin{bmatrix} -0. & -0.1068 \\ -0.1068 & -0.1068 \\ -0.1068 & -0.1068 \end{bmatrix}$$

$$g_2 = \begin{bmatrix} -0.7162 \\ -0.4549 \\ -0.4549 \end{bmatrix}$$

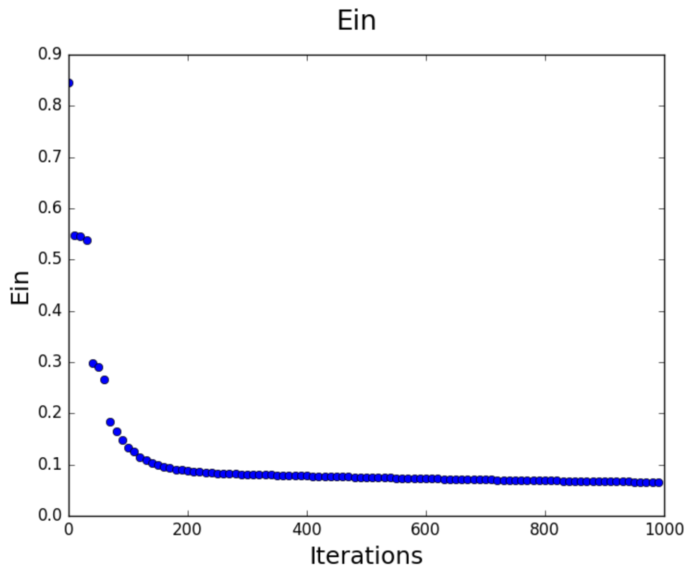
Neural Networks for Digits

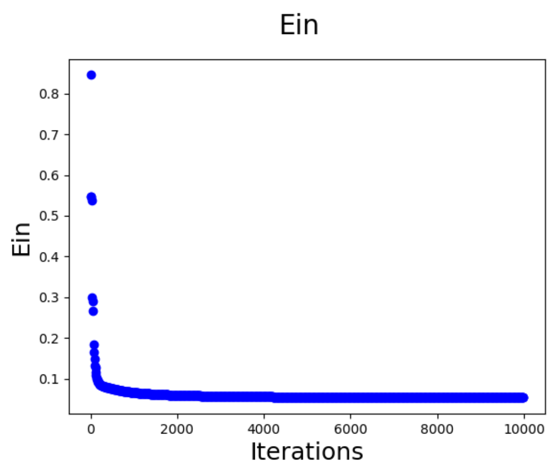
a

Below is a chart plotting $E_{in}(\mathbf{w})$ versus iterations up to 1,000,000. I want to run to 2,000,000, but my processing limitations made me unable. It took almost two days to run up to 1,000,000.



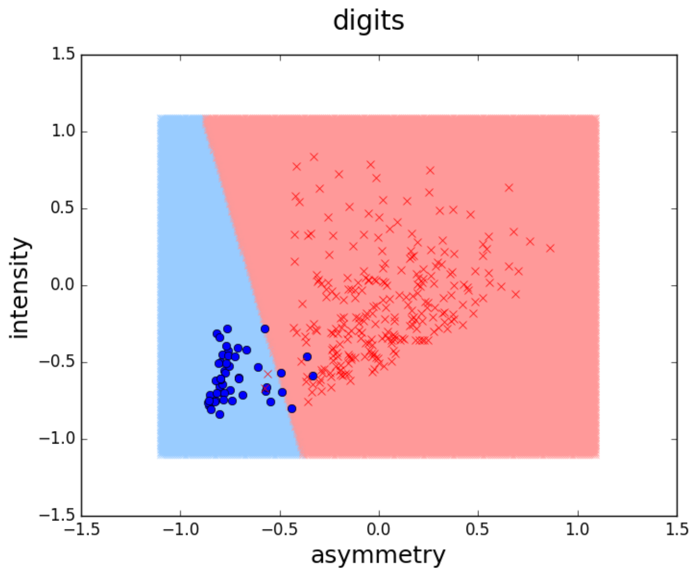
I have also zoomed in on the first 1,000 and 10,000 iterations to show the initial drop better:



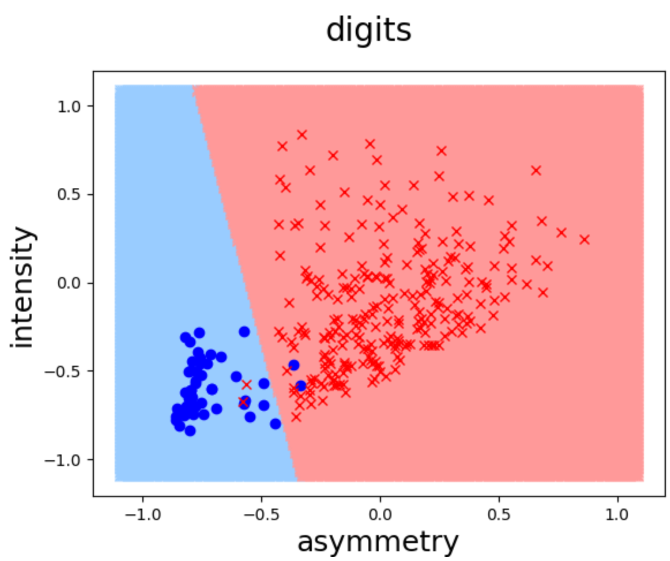


Interestingly, there isn't a very big difference between 10,000 iterations versus 1,000,000. Here, I have plotted the decision boundaries on the training data:

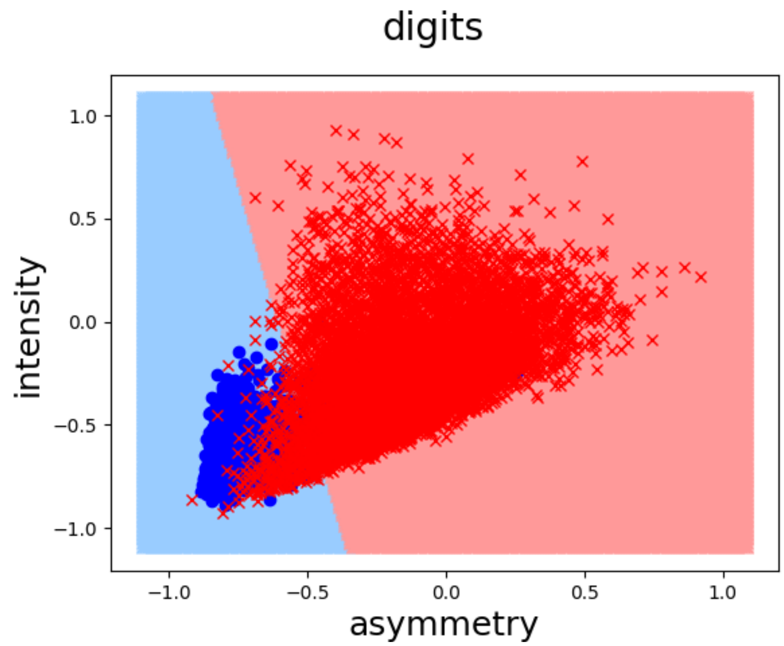
1000 Iterations



1,000,000 Iterations



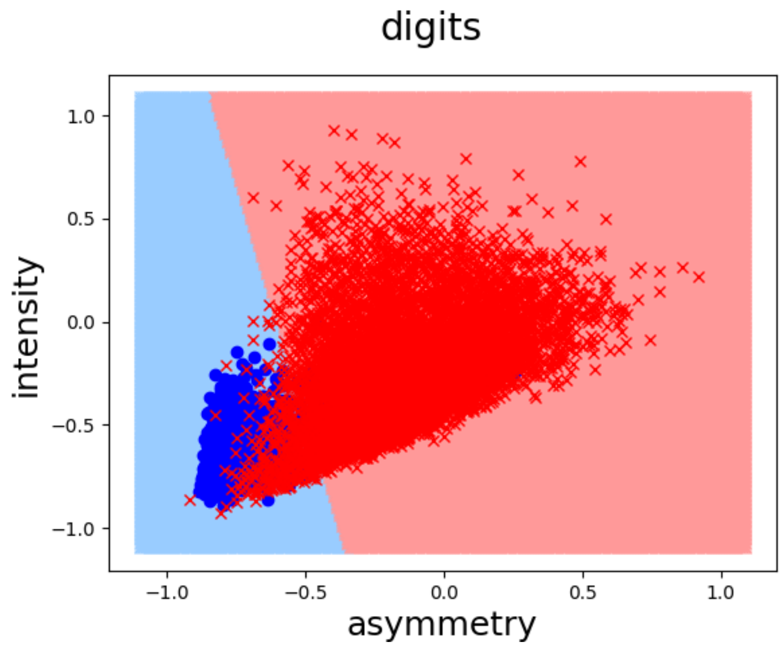
Below, I plot 1,000,000 iterations on my testing data



The results of my neural network was very surprising. After about 80,000 iterations, there was little to no improvement on E_{in}. I believe that this is due to the small size of the network. Also, gradient descent is sporadic in nature. Maybe the classifier becomes overfitted after 5 or 10 million iterations? Unless this is run with that many iterations, there is no way to really know.

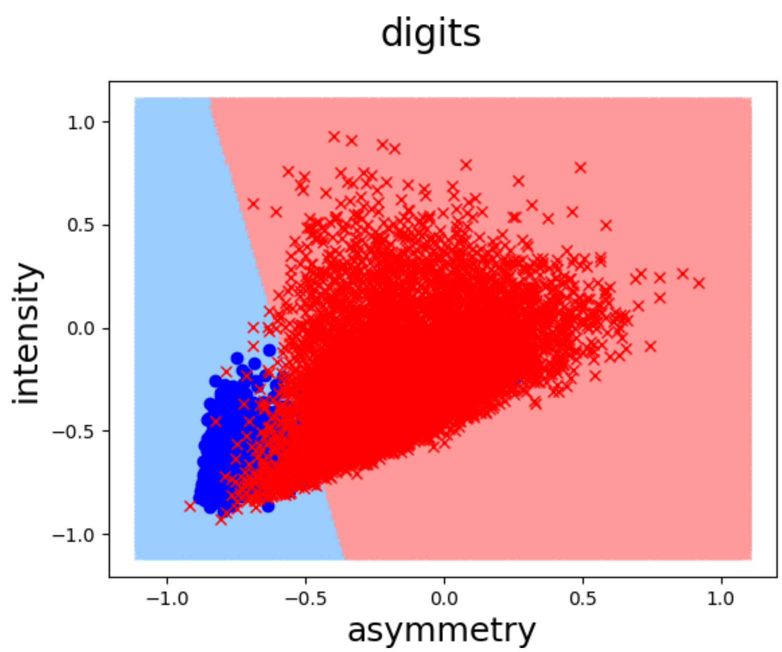
b

As I stated before, the max number of iterations wasn't enough to become overfitted. As a result, I ran weight decay with $\lambda = \frac{0.01}{N}$ up to just 80,000 iterations. This classifier looked pretty similar to my previous graphs, as expected.



c

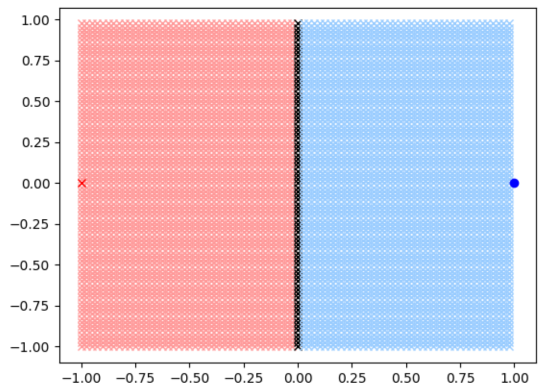
For early stopping, I tested validation up to 50,000 iterations. Validation was lowest at just 1,100 iterations. Here is the resulting classifier on my testing data, with $E_{test} = 0.044$.



Support Vector Machines

a

I ran a SVM algorithm using quadratic programming on this simple data set. Here is the resulting classifier:



The equation for this line is just $x = 0$

b

The data points in this new space are just the same. This is due to the nature of squaring 1's and 0's

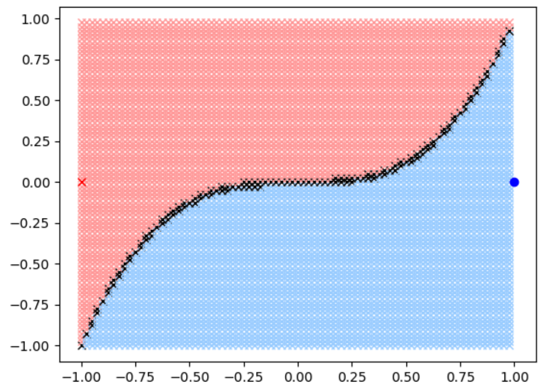
$$x_1 = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$$

$$x_2 = \begin{bmatrix} -1 \\ 0 \end{bmatrix}$$

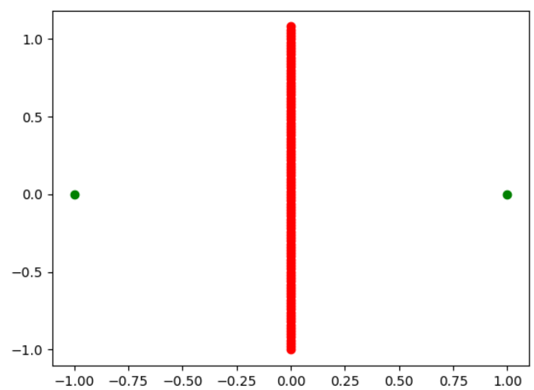
$$z_1 = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$$

$$z_2 = \begin{bmatrix} -1 \\ 0 \end{bmatrix}$$

The graph for this is given below. The equation is $z = 0$

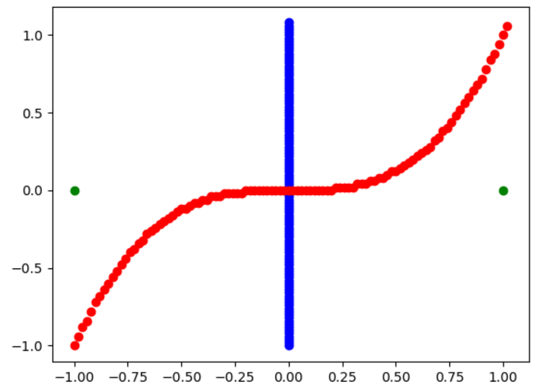


And given below is this same graph except kept in the Z-space. The red classifier is on the left and the blue classifier is on the right



c

On this graph, I have plotted both the boundaries together. The X-space plot is blue, and the Z-space plot is red.



d

We have two points x and y. We will transform them based on the specifications given earlier in the question.

$$x = \begin{bmatrix} x_0 \\ x_1 \end{bmatrix}$$

$$x_z = \begin{bmatrix} x_0^3 - x_1 \\ x_0 x_1 \end{bmatrix}$$

$$y = \begin{bmatrix} y_0 \\ y_1 \end{bmatrix}$$

$$y_z = \begin{bmatrix} y_0^3 - y_1 \\ y_0 y_1 \end{bmatrix}$$

Then we get the dot product

$$x_0^3 y_0^3 - x_0^3 y_1 - x_1 y_0^3 + x_1 y_1 + x_0 x_1 y_0 y_1$$

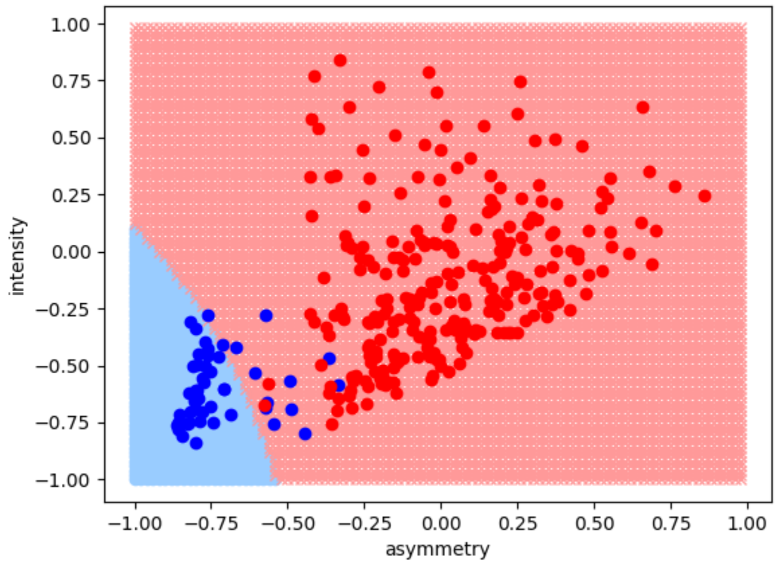
e

We can turn this into a simple classifier as $h(x) = sign(x^3 - y)$.

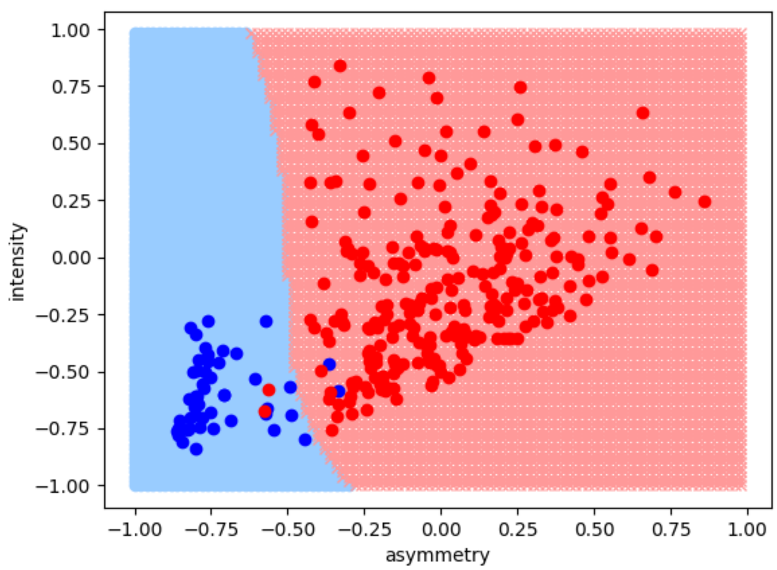
SVM with Digits Data

a

I chose a very small value of $C = 0.1$. This way, the classifier is very simple. It is shown below



I chose a large value of $C = 1000$. The classifier is now more complex.

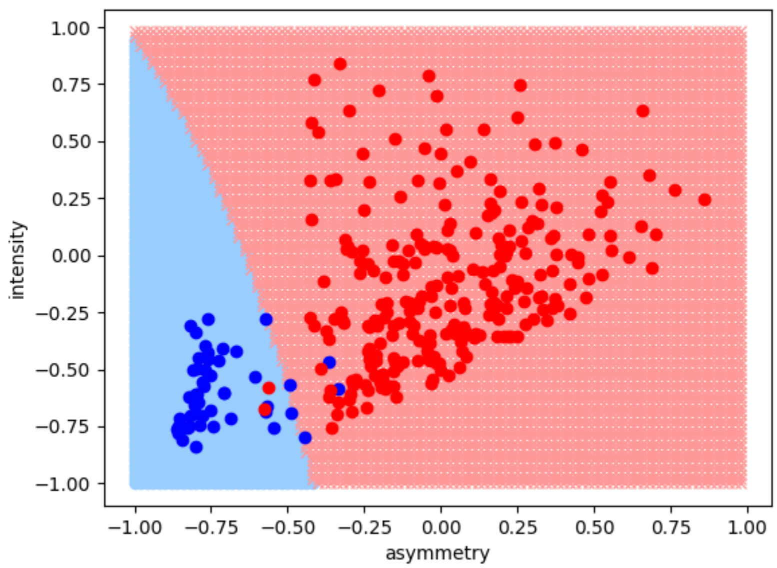


b

As C gets higher, the complexity of the classifier goes up. This is because the line in Z-Space will be forced to change if the costs of error are much higher. The classifier in the small c example is almost linear. The classifier in the large c example looks higher order.

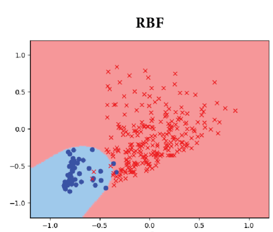
c

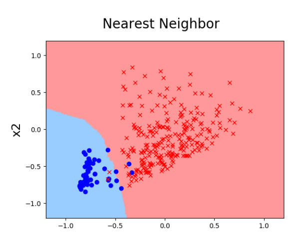
After using cross validation, my best value for C is 1.5. This gave me an E_{cv} of 0.039. When I ran this on my testing data, I achieved an E_{test} of 0.0446. The graph of this with the testing data is below

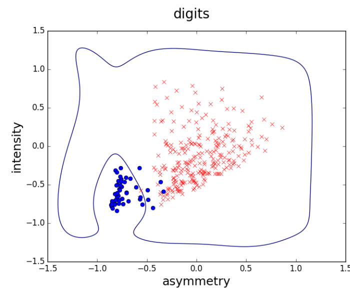


Compare Methods

For reference, I have given the classifiers that aren't part of this homework below. This includes k-NN, k-RBF, and Linear 8th Order.







Model	E_{test}
Linear 8th Order	0.032
k-NN	0.030
k-RBF	0.085
Early-stop NN	0.044
SVM 8th Order	0.047

The similarity of all these methods is very interesting, but expected. All of these methods are powerful, and can fit data in a 2D space very well. So, these are all regularized, with the regularization parameter picked using some type of validation. As a result, the E_{in} is pretty similar across models.

In regards to picking a model that is the best, I would prefer either Neural Networks or SVMs. Although the other methods are powerful, Neural Networks and SVMs are in a league of their own.

As discussed in class, the model (as long as it is properly regularized) doesn't matter that much in regards to E_{test}. It seems that what really matters is feature selection and creation. Most good models will get an E_{test} down to 3 or 4 percent. To get down to 1 percent, excellent features must be selected.