Guthrie Alexander

CS 6316- Machine Learning

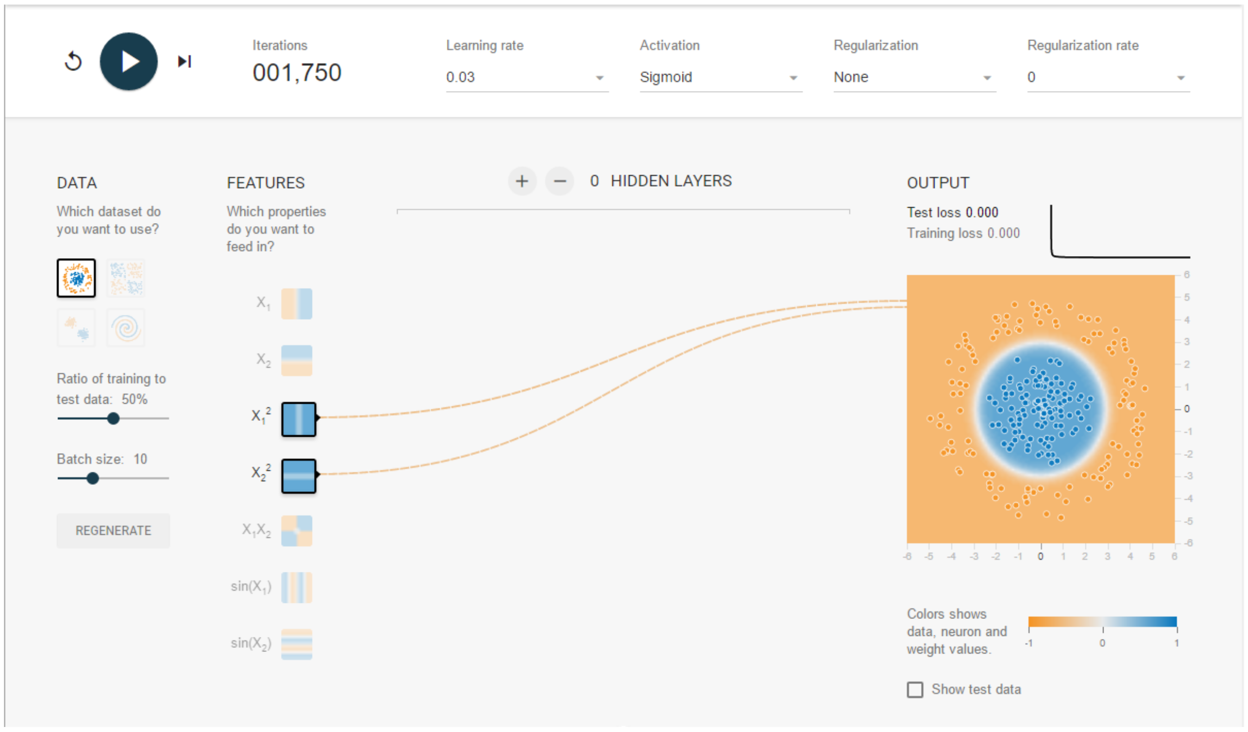
HW6

Question 1

1.1

	Selected Features	Iterations	Test loss
Circle	X1^2, X2^2	1750	0.000
Exclusive Or	X1X2	3662	0.000
Gaussian	X1, X2	135	0.000
Spiral	X2, sin(X2)	1500	0.486

Circle:


Iterations 001,750

Learning rate 0.03

Activation Sigmoid

Regularization None

Regularization rate 0

0 HIDDEN LAYERS

Test loss 0.000

Training loss 0.000

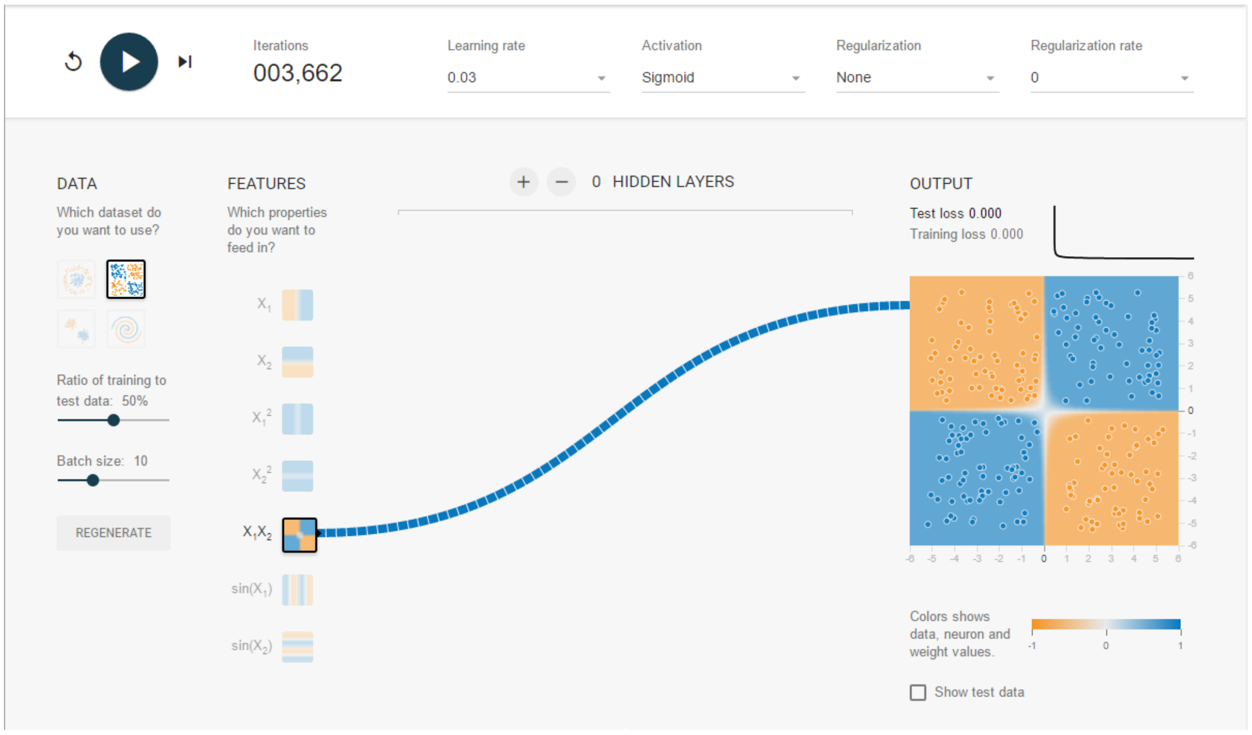
Ratio of training to test data: 50%

Batch size: 10

DATA
Which dataset do you want to use?
FEATURES
Which properties do you want to feed in?
OUTPUT
REGENERATE
Colors shows data, neuron and weight values.
Show test data

X1^2 and X2^2 worked the best, I think because of how the features were aligned. Other combinations with those two features worked, but they always rely on those two features to reach an ideal test loss. This is due to these two features being a quadratic new feature of the original feature sets.

Exclusive Or:


Iterations 003,662
Learning rate 0.03
Activation Sigmoid
Regularization None
Regularization rate 0

DATA
Which dataset do you want to use?
Ratio of training to test data: 50%
Batch size: 10

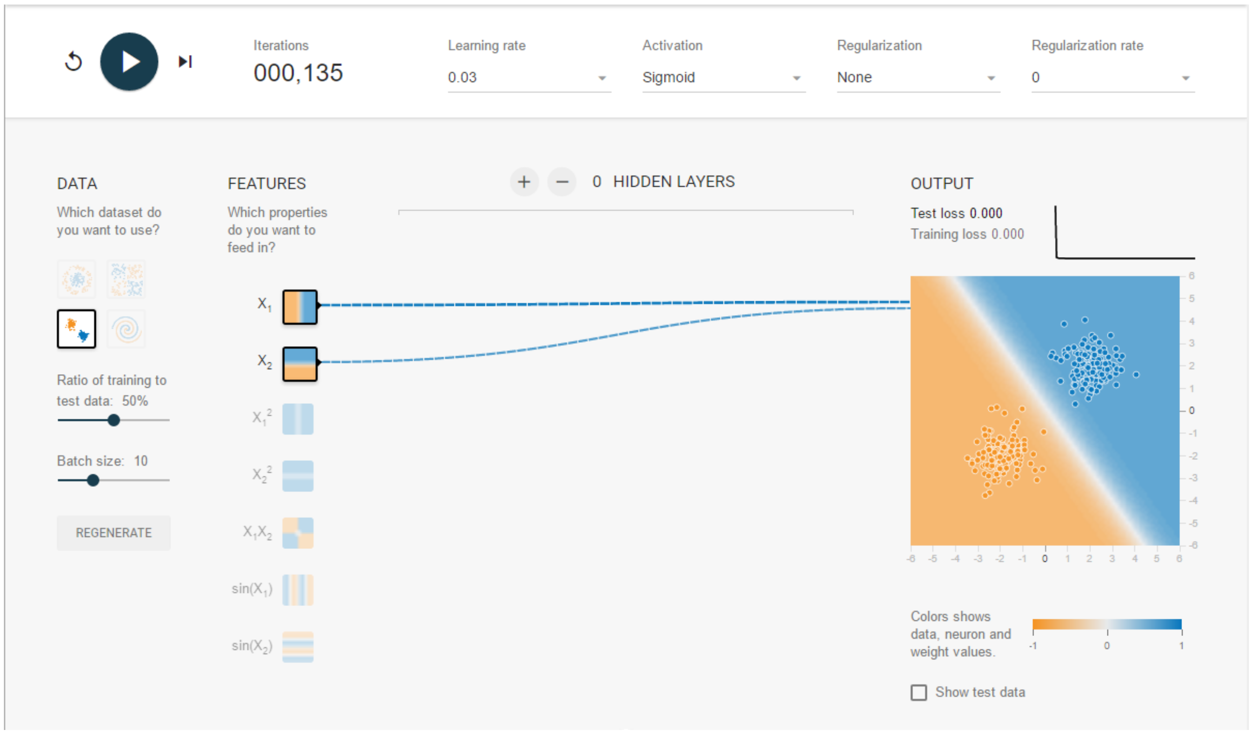
FEATURES
Which properties do you want to feed in?

0 HIDDEN LAYERS

OUTPUT
Test loss 0.000
Training loss 0.000

X1X2 was a good match, simply by looking at the way the points were distributed. It makes sense that none of the other features were needed. This is simply because of the nature of how multiplying the two features to get a new phi() feature set works.

Gaussian:


Iterations 000,135

Learning rate 0.03

Activation Sigmoid

Regularization None

Regularization rate 0

DATA

Which dataset do you want to use?

Ratio of training to test data: 50%

Batch size: 10

FEATURES

Which properties do you want to feed in?

0 HIDDEN LAYERS

OUTPUT

Test loss 0.000

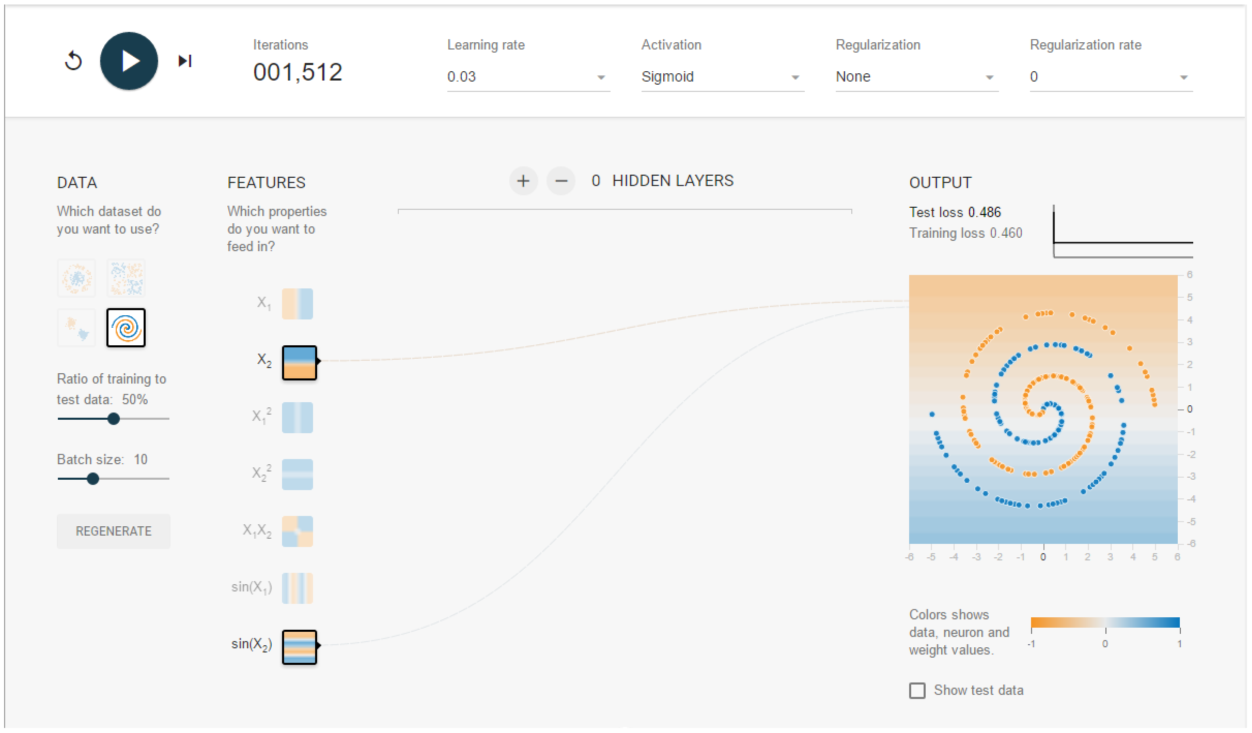
Training loss 0.000

Colors shows data, neuron and weight values.

Show test data

The overlap for positive/negative data points in the two features allows this model to quickly find a boundary line between the two data regions.

Spiral



Iterations 001,512

Learning rate 0.03

Activation Sigmoid

Regularization None

Regularization rate 0

DATA

Which dataset do you want to use?

Ratio of training to test data: 50%

Batch size: 10

FEATURES

Which properties do you want to feed in?

0 HIDDEN LAYERS

OUTPUT

Test loss 0.486

Training loss 0.460

Colors shows data, neuron and weight values.

Show test data

There was no combination of features that allowed for good classification of these data points. This was the best I could do just with a perceptron model (i.e. 1 layer model). The complexity of the data set requires a more complicated model based on more than just a simple perceptron model.

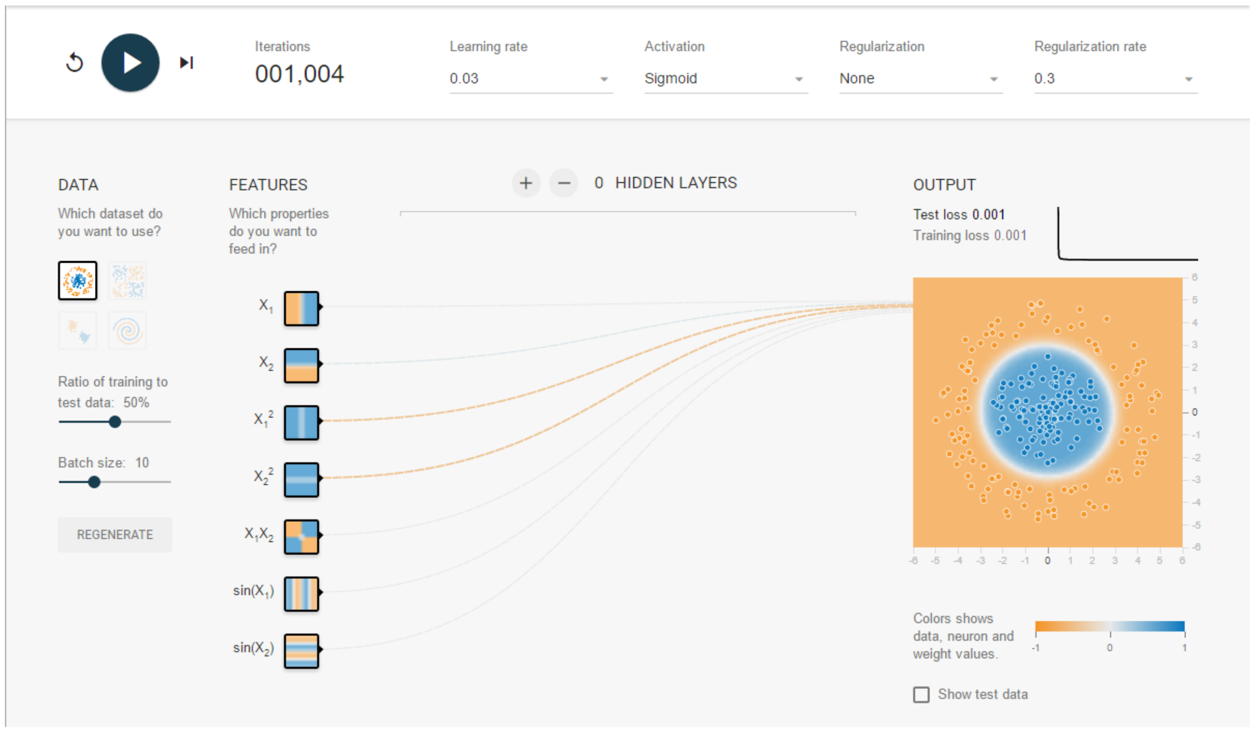
1.2 Regularization

~1000 iterations

	No regularization	L1, rate = 0.1	L2, rate = 0.3
Circle	0.001	0.002	0.003
Exclusive Or	0.001	0.004	0.014
Gaussian	0.001	0.003	0.004

Circle:

No regularization:



Iterations 001,004

Learning rate 0.03

Activation Sigmoid

Regularization None

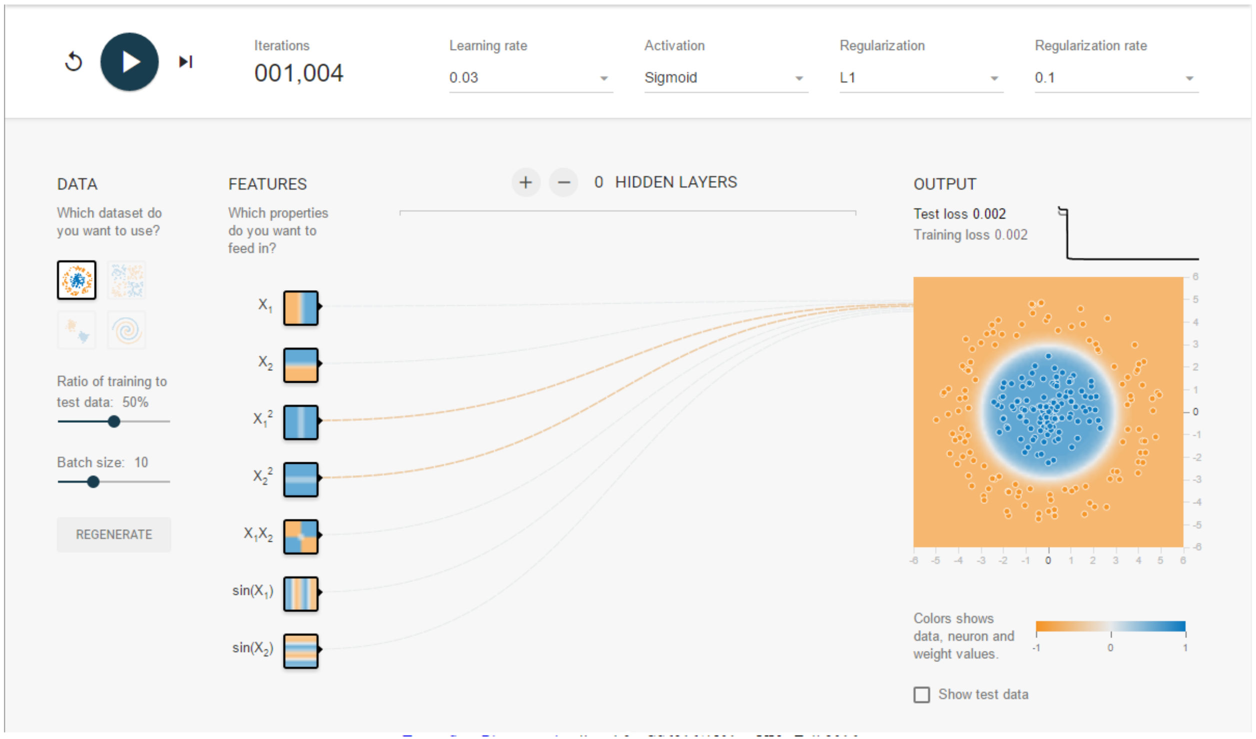
Regularization rate 0.3

DATA
Which dataset do you want to use?
Ratio of training to test data: 50%
Batch size: 10
REGENERATE
FEATURES
Which properties do you want to feed in?
X_1
X_2
X_1^2
X_2^2
X_1X_2
$\sin(X_1)$
$\sin(X_2)$
0 HIDDEN LAYERS
OUTPUT
Colors shows data, neuron and weight values.
Show test data

Test loss 0.001

Training loss 0.001

L1 regularization:



Iterations 001,004

Learning rate 0.03

Activation Sigmoid

Regularization L1

Regularization rate 0.1

DATA

Which dataset do you want to use?

Ratio of training to test data: 50%

Batch size: 10

REGENERATE

FEATURES

Which properties do you want to feed in?

0 HIDDEN LAYERS

OUTPUT

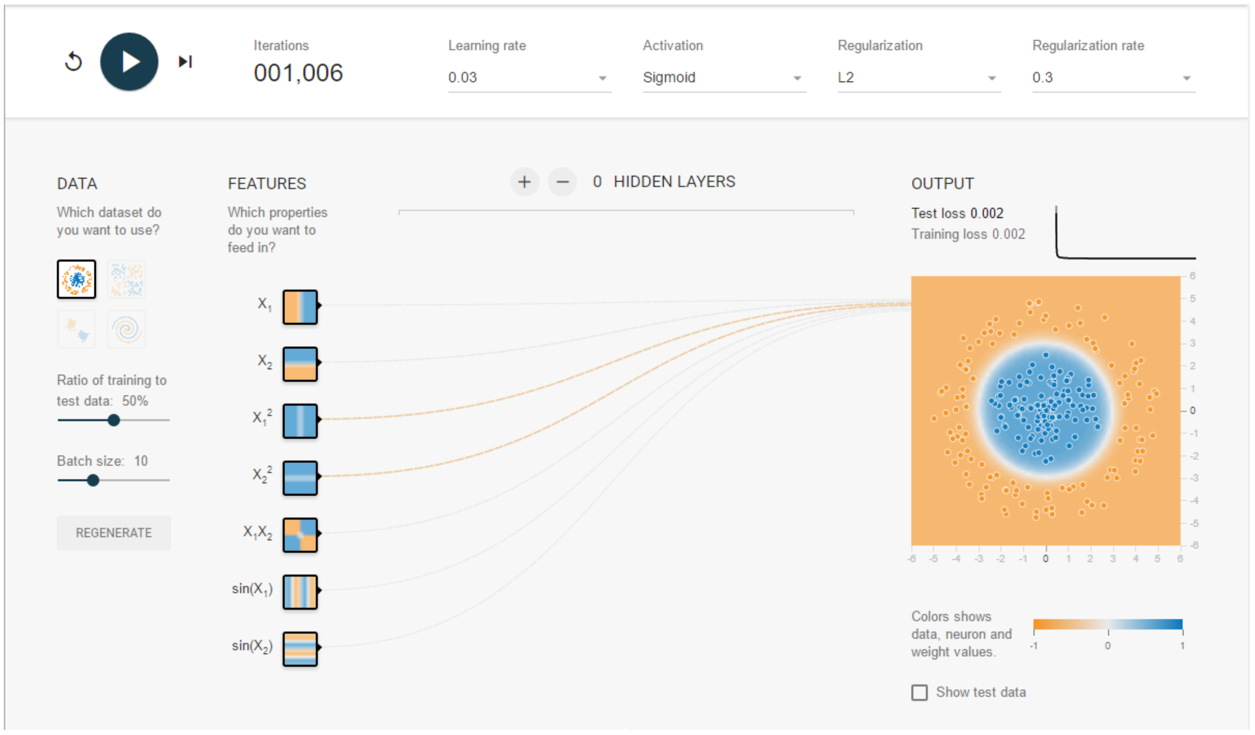
Test loss 0.002

Training loss 0.002

Colors shows data, neuron and weight values.

Show test data

L2:



Iterations 001,006

Learning rate 0.03

Activation Sigmoid

Regularization L2

Regularization rate 0.3

DATA

Which dataset do you want to use?

Ratio of training to test data: 50%

Batch size: 10

REGENERATE

FEATURES

Which properties do you want to feed in?

0 HIDDEN LAYERS

OUTPUT

Test loss 0.002

Training loss 0.002

Colors shows data, neuron and weight values.

Show test data

Circle Models: Non regularized works the best because the boundary lines neatly bisect the two different classes, and regularization is not needed, and only creates situations where points are not classified correctly.

Task B (Circle)

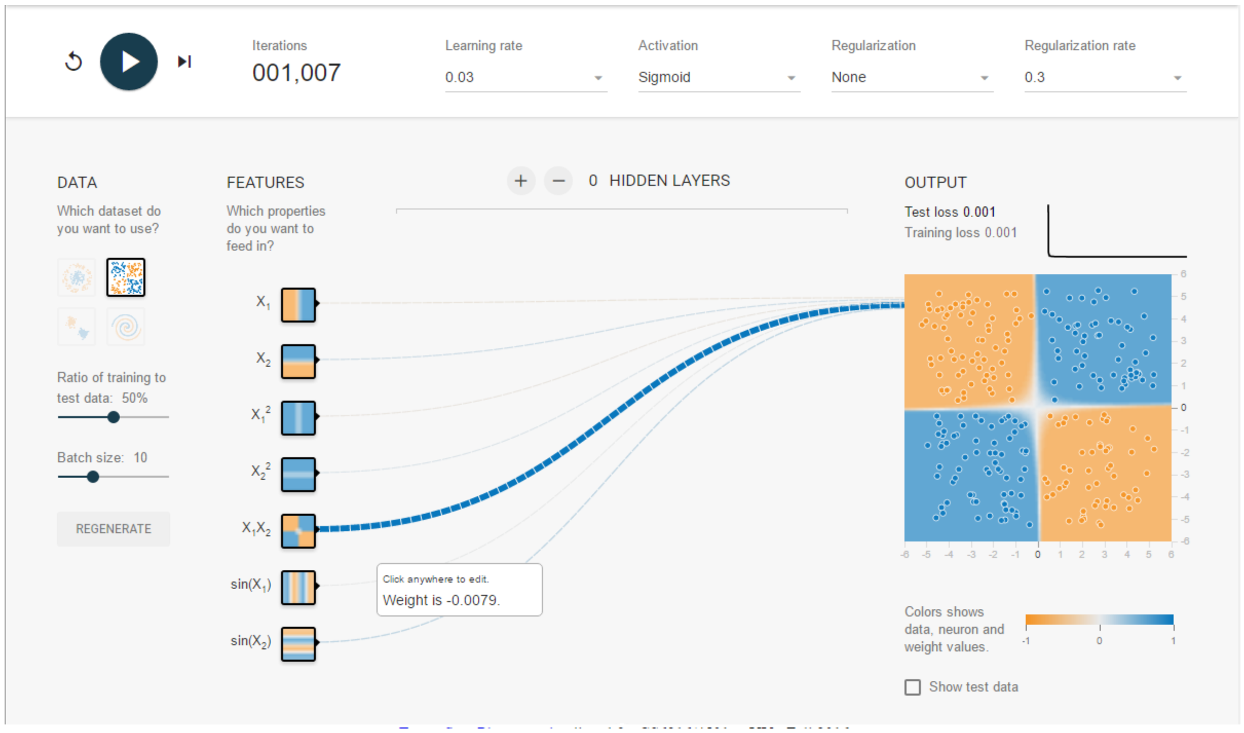
Looking at the L1 model, you can see that X1^2 and X1^2 have higher weights than the other features for the Circle Data set. These are the same as my selection in 1.1.

Data Weights
X1^2: -0.29

X2^2:-0.29

Exclusive Or

No regularization:


Iterations
001,007

Learning rate
0.03

Activation
Sigmoid

Regularization
None

Regularization rate
0.3

DATA
Which dataset do you want to use?

Ratio of training to test data: 50%

Batch size: 10

REGENERATE

FEATURES
Which properties do you want to feed in?

0 HIDDEN LAYERS

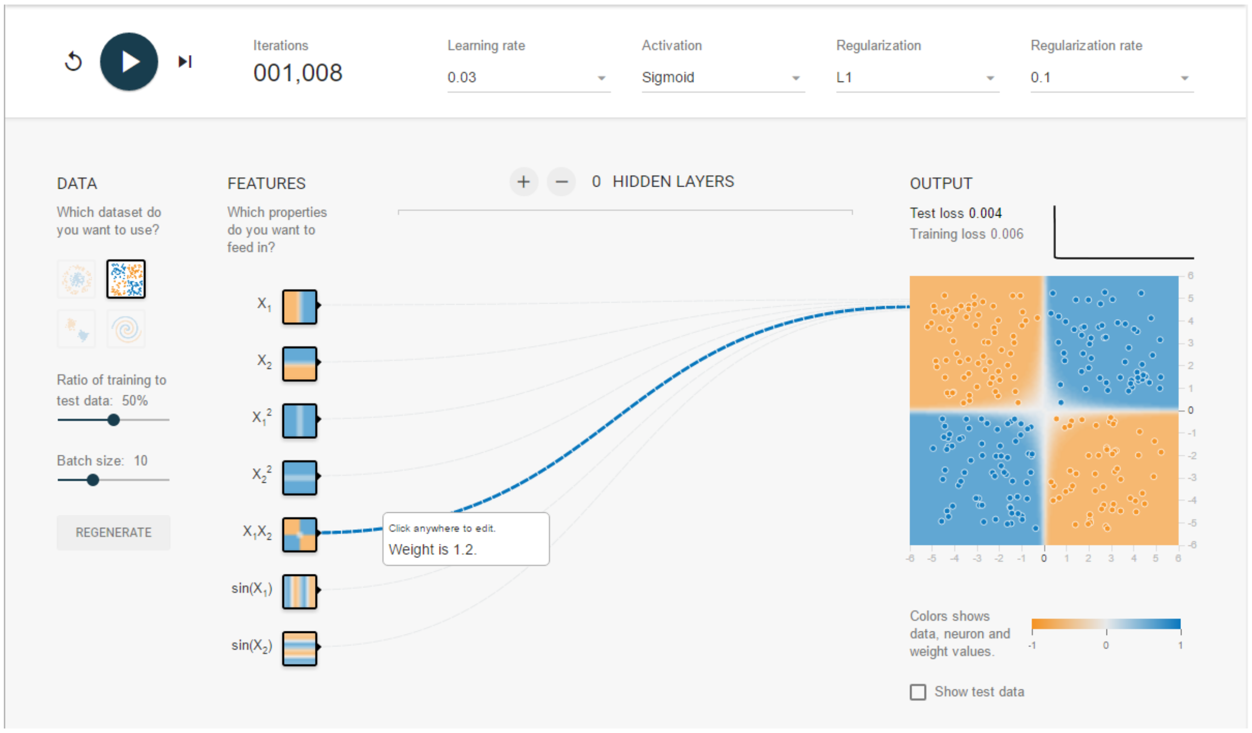
OUTPUT
Test loss 0.001
Training loss 0.001

Click anywhere to edit.
Weight is -0.0079.

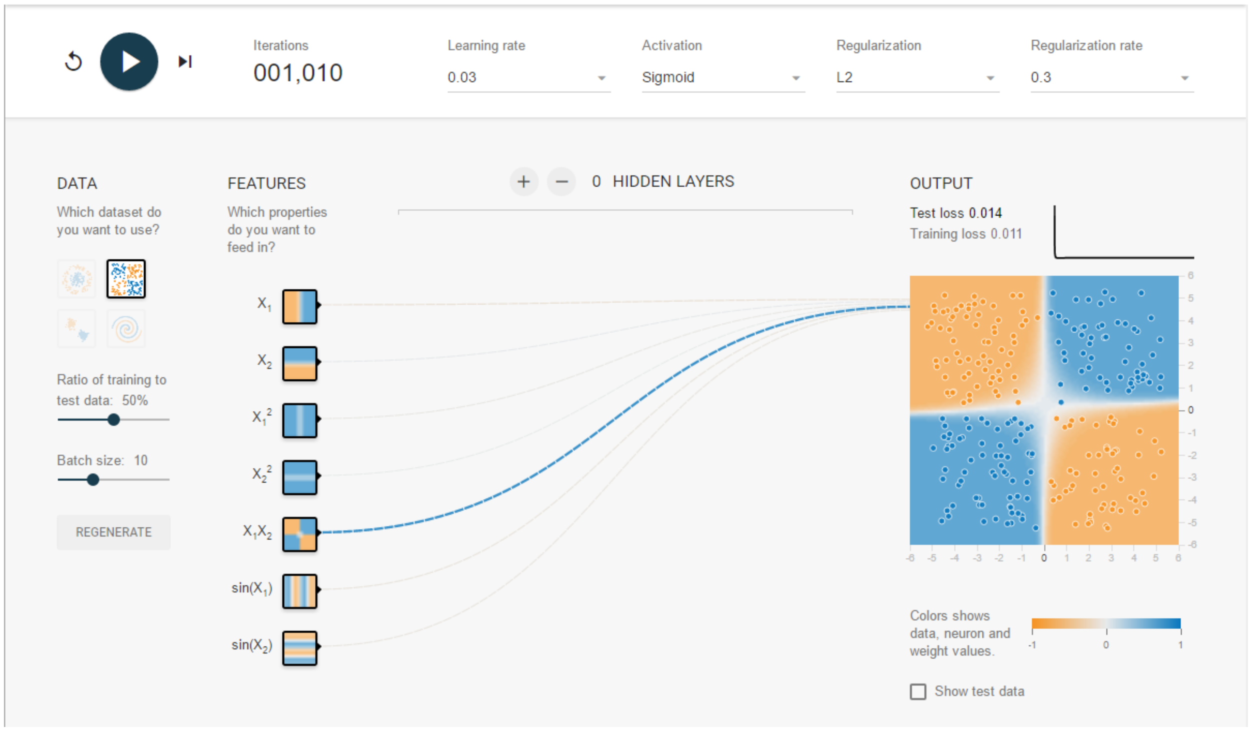
Colors shows data, neuron and weight values.

Show test data

L1:


Iterations 001,008
Learning rate 0.03
Activation Sigmoid
Regularization L1
Regularization rate 0.1
DATA
Which dataset do you want to use?
Ratio of training to test data: 50%
Batch size: 10
FEATURES
Which properties do you want to feed in?
0 HIDDEN LAYERS
Click anywhere to edit.
Weight is 1.2.
OUTPUT
Test loss 0.004
Training loss 0.006
Colors shows data, neuron and weight values.
Show test data

L2:


Iterations 001,010
Learning rate 0.03
Activation Sigmoid
Regularization L2
Regularization rate 0.3
DATA
Which dataset do you want to use?
Ratio of training to test data: 50%
Batch size: 10
FEATURES
Which properties do you want to feed in?
0 HIDDEN LAYERS
OUTPUT
Test loss 0.014
Training loss 0.011
Colors shows data, neuron and weight values.
Show test data

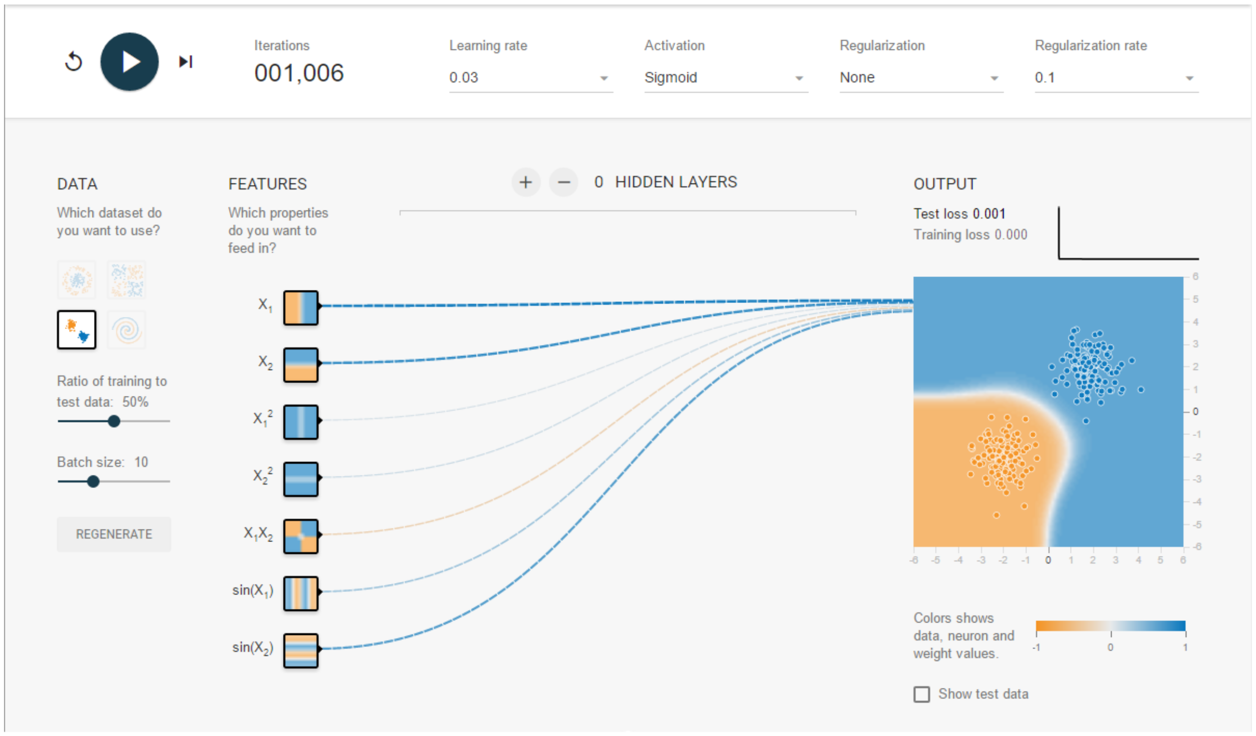
Exclusive Or models: Non regularized works the best again because the boundary lines neatly bisect the two different classes, and regularization is not needed, and only creates situations where points are not classified correctly.

Task B (Exclusive Or): As you can see, the X1X2 feature is the one weighted the highest after L1 regularization, which is the same feature I picked in 1.1

Weight (X1X2): 1.1

Gaussian:

No regularization:


Iterations 001,006

Learning rate 0.03

Activation Sigmoid

Regularization None

Regularization rate 0.1

DATA

Which dataset do you want to use?

Ratio of training to test data: 50%

Batch size: 10

REGENERATE

FEATURES

Which properties do you want to feed in?

0 HIDDEN LAYERS

OUTPUT

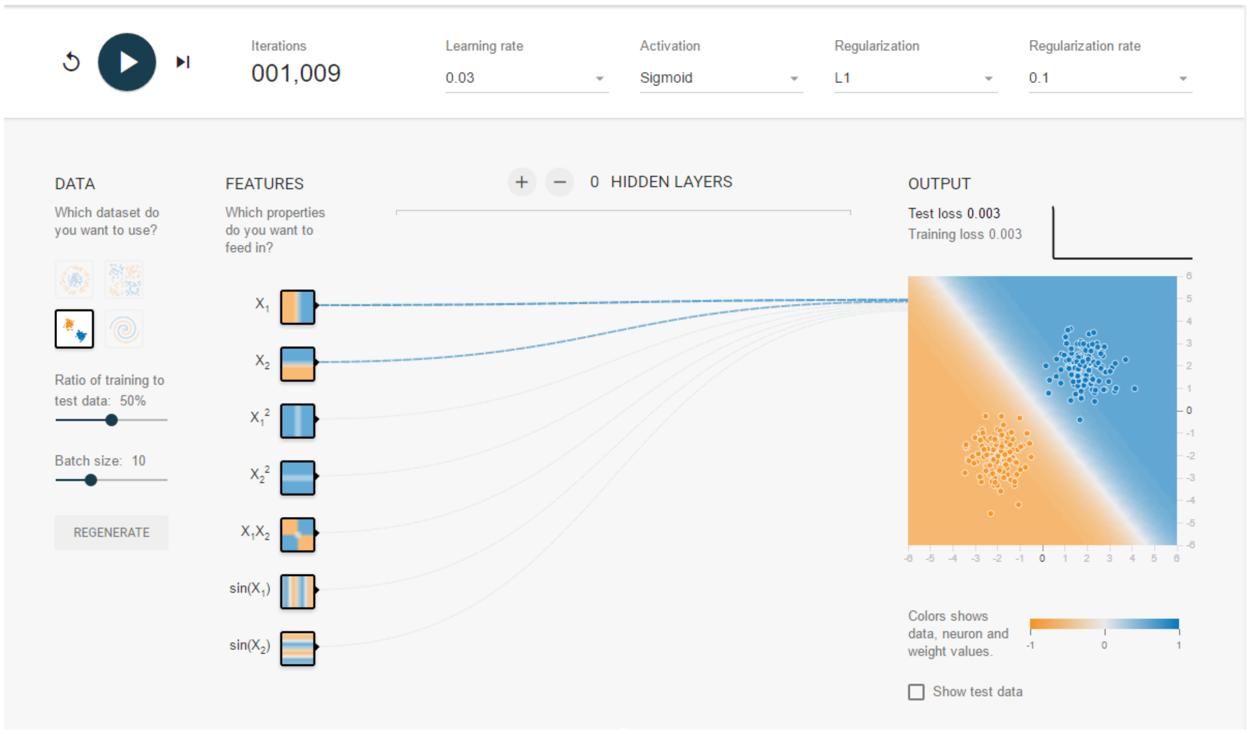
Test loss 0.001

Training loss 0.000

Colors shows data, neuron and weight values.

Show test data

L1:


Iterations 001,009
Learning rate 0.03
Activation Sigmoid
Regularization L1
Regularization rate 0.1

DATA
Which dataset do you want to use?
Ratio of training to test data: 50%
Batch size: 10
REGENERATE

FEATURES
Which properties do you want to feed in?

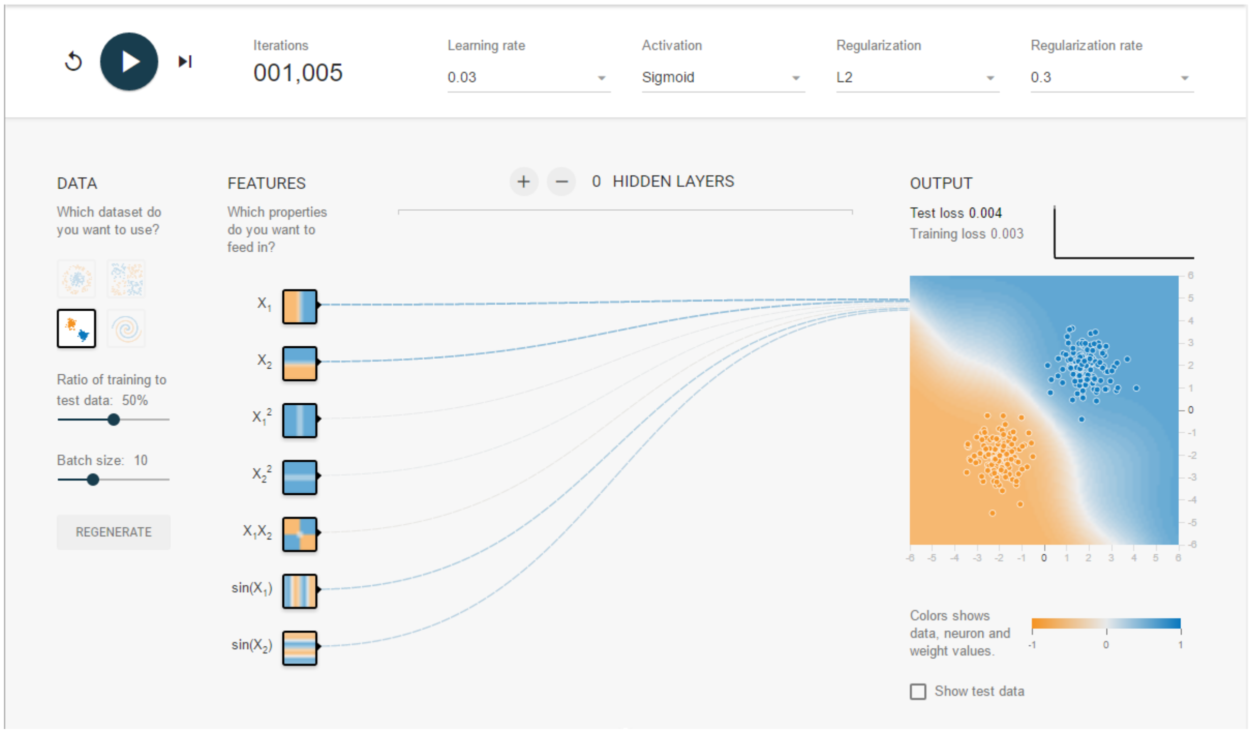
0 HIDDEN LAYERS

OUTPUT
Test loss 0.003
Training loss 0.003

Colors shows data, neuron and weight values.

Show test data

L2:


Iterations 001,005
Learning rate 0.03
Activation Sigmoid
Regularization L2
Regularization rate 0.3

DATA
Which dataset do you want to use?
Ratio of training to test data: 50%
Batch size: 10
REGENERATE

FEATURES
Which properties do you want to feed in?

0 HIDDEN LAYERS

OUTPUT
Test loss 0.004
Training loss 0.003

Colors shows data, neuron and weight values.

Show test data

Gaussian models: The boundary line was pretty straight/linear with L1 regularization, but the test loss wasn't quite as good as it was with no regularization.

Task B (Gaussian):
As we can see from the L1 Regularized model, the strongest features are X1 and X2, as I selected before.

X1 weight: 0.61

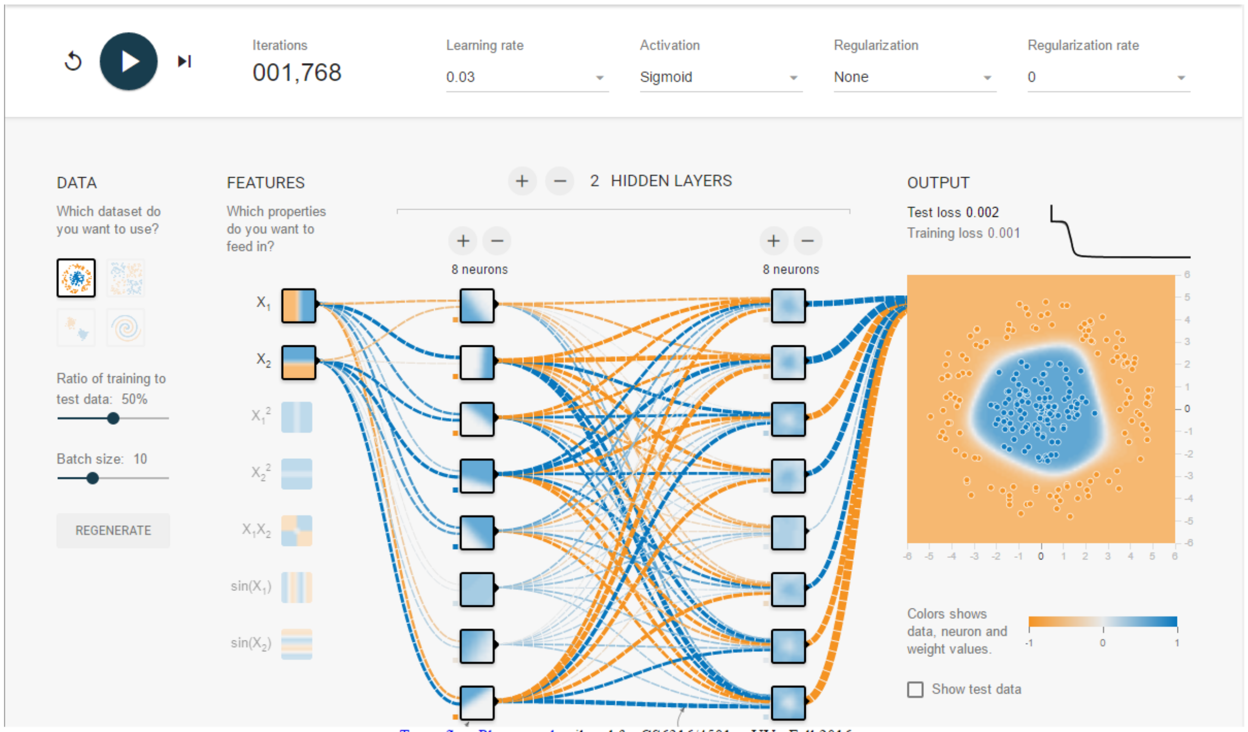
X2 weight: 0.49

1.3

Circle:

Sigmoid, 1750 iterations, no regularization

Test loss = 0.002



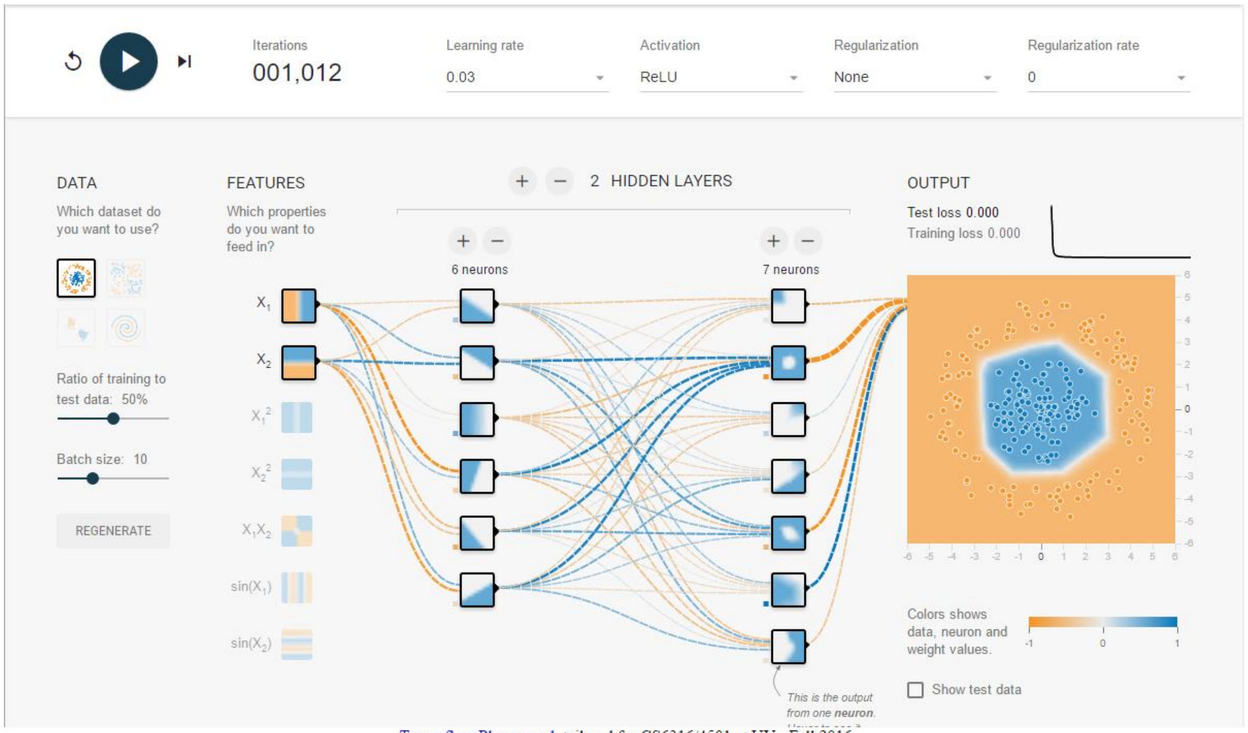
Iterations 001,768
Learning rate 0.03
Activation Sigmoid
Regularization None
Regularization rate 0

Ratio of training to test data: 50%
Batch size: 10

Test loss 0.002
Training loss 0.001

Activation: ReLU, 1000 iterations, no regularization.

Test loss = 0.000



Iterations 001,012

Learning rate 0.03

Activation ReLU

Regularization None

Regularization rate 0

Ratio of training to test data: 50%

Batch size: 10

2 HIDDEN LAYERS

6 neurons

7 neurons

Test loss 0.000

Training loss 0.000

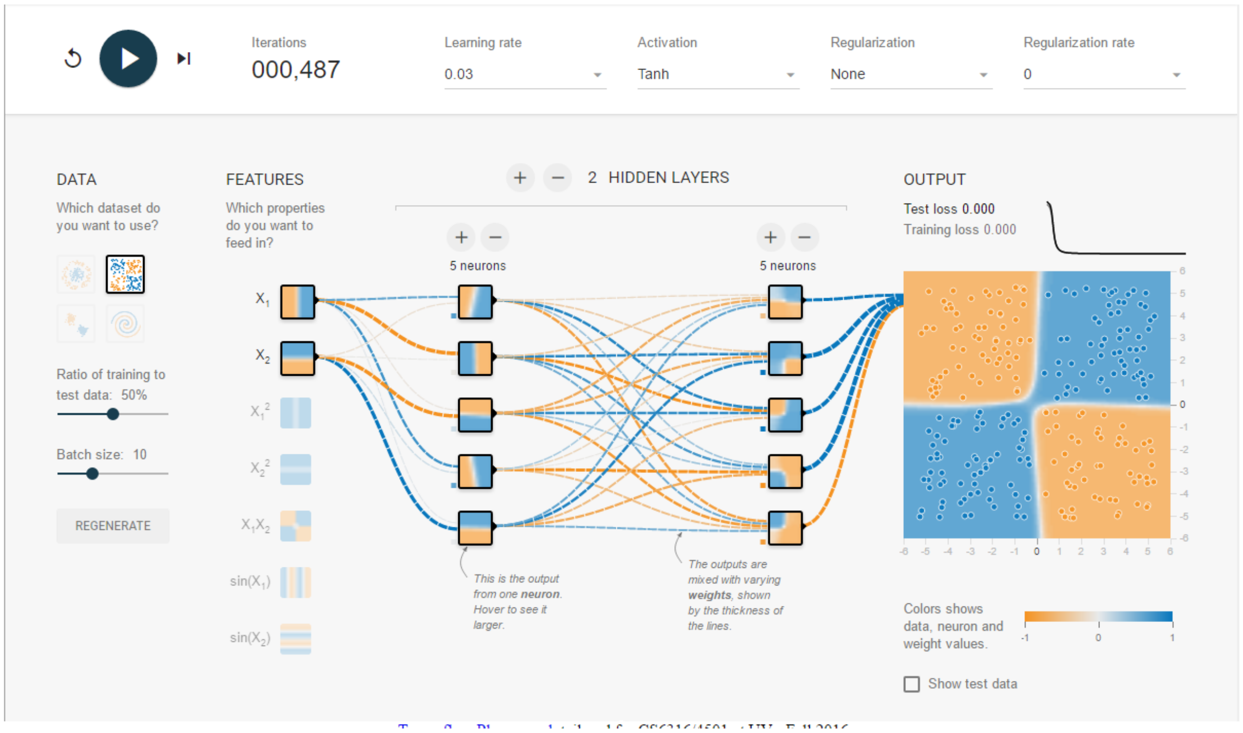
DATA
Which dataset do you want to use?
REGENERATE
FEATURES
Which properties do you want to feed in?
X_1
X_2
X_1^2
X_2^2
X_1X_2
$\sin(X_1)$
$\sin(X_2)$
OUTPUT
This is the output from one neuron.
Colors shows data, neuron and weight values.
-1
0
1
Show test data

This performs better than the hand-crafted feature solution above (i.e. it gets to a 0% test loss faster)

The changes of using the rectified linear unit activation function (ReLU) as well as the added hidden layers creates a more complicated model, allowing for adjustments needed for forming around a circular data set.

Exclusive Or:

Activation: Tanh, no regularization, 487 iterations:



Iterations 000,487

Learning rate 0.03

Activation Tanh

Regularization None

Regularization rate 0

Ratio of training to test data: 50%

Batch size: 10

Test loss 0.000

Training loss 0.000

This is the output from one **neuron**. Hover to see it larger.

The outputs are mixed with varying **weights**, shown by the thickness of the lines.

Colors shows data, neuron and weight values.

Much, much faster than the hand-crafted solution to this data set.

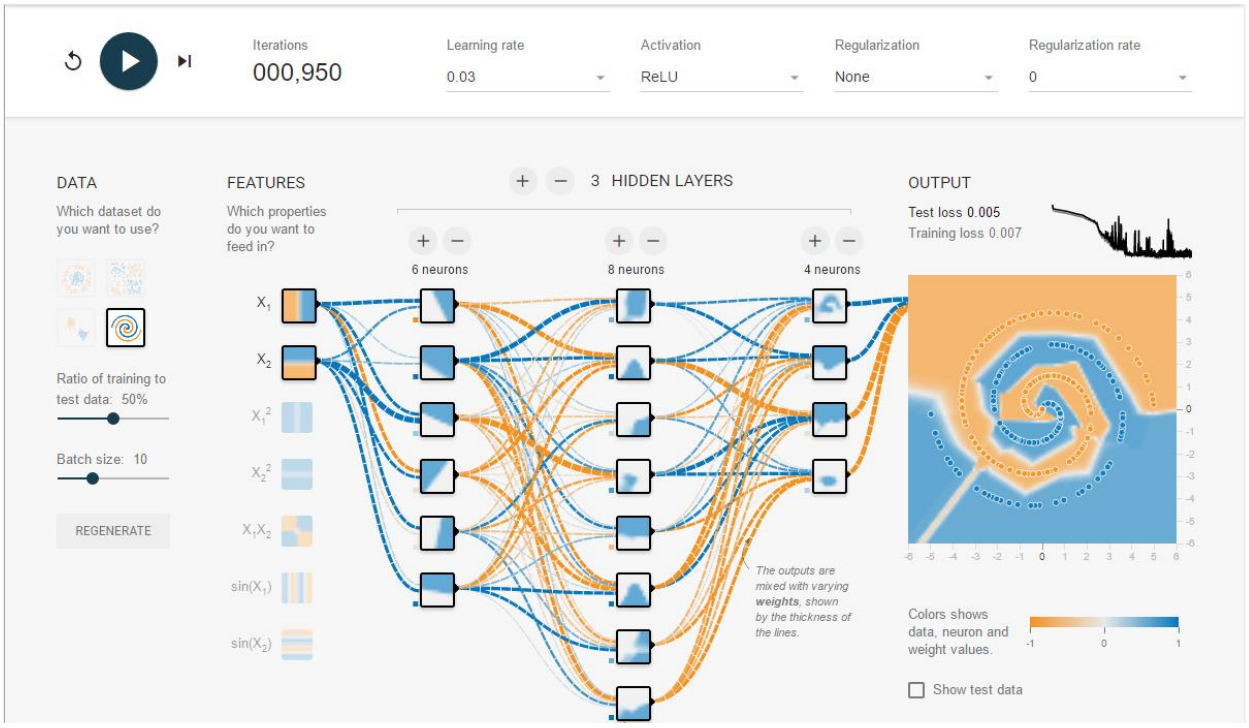
Tanh activation, which I think is the rescaled version of the sigmoid function, in addition to the hidden layers, allowed for the weights to do most of the work. Instead of taking ~3000 iterations to reach 0.000 test error, it only takes 487. This is mostly thanks to the hidden neurons, each targeting a certain decision boundary that works perfectly with this data set.

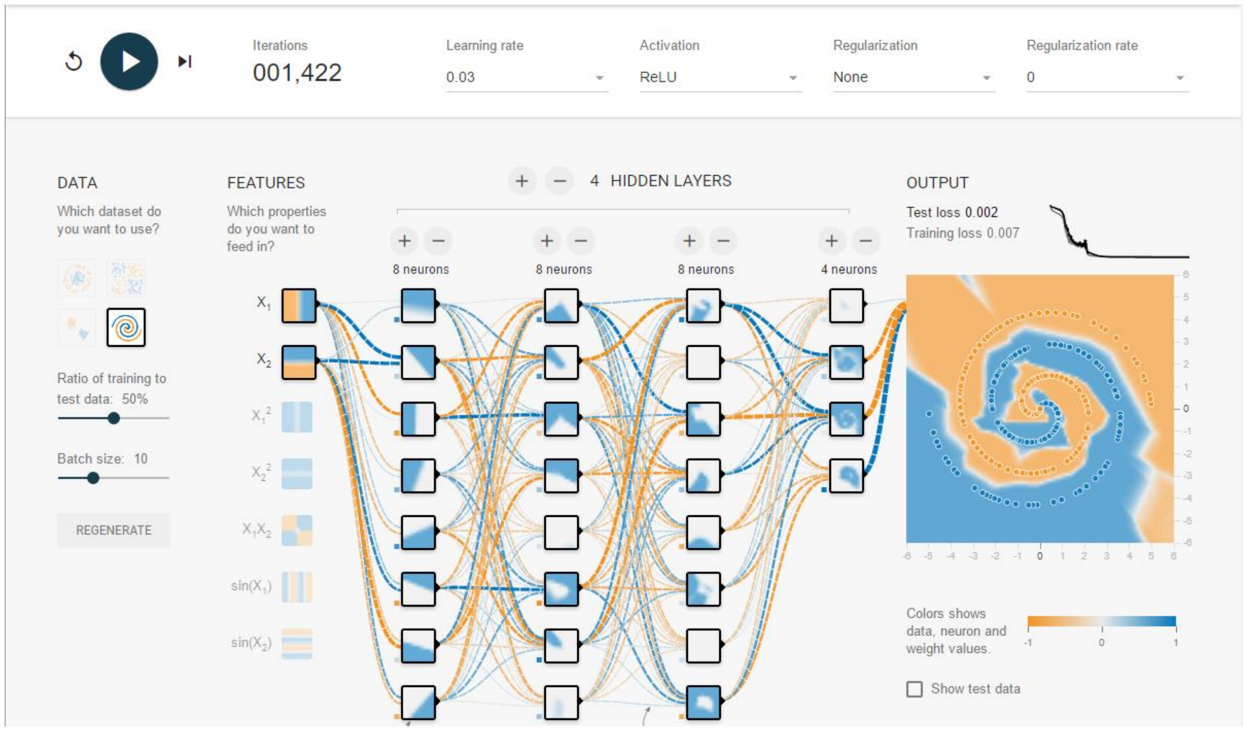
Additionally, as we saw before, regularization doesn't work very well with this particular data set, so we didn't use it here.

1.4 Spiral Challenge

Input Features:	Network Architecture	Hyperparameters	Iterations	Test loss
X1, X2	7-7-4	Tanh, none, 0	1000	0.3
X1, X2	8-8	Sigmoid, none, 0	1000	0.53

X1, X2	6-6-2	ReLU, none, 0	2000	0.024
X1, X2	8-8-6	ReLU, none, 0	1000	0.07
X1, X2	6-8-4	Tanh	1500	0.09
X1, X2	**6-8-4**	**ReLU, none, 0**	**950**	**0.005**
X1, X2	**8-8-8-4**	**ReLU, none, 0**	**1400**	**0.002**
X1, X2	8-8-8-6-4	ReLU, none, 0	1000	0.097
X1, X2	8-8-8-8-8-8	ReLU, none, 0	800	0.019
X1,X2	8-8-8-7-7-6	ReLU, none, 0	650	0.241
X1, X2	8-8-8-4	Tanh, none, 0	500	0.138
X1, X2	8-8-8-6	ReLU, L1, 0.001	1000	0.048


Iterations 000,950
Learning rate 0.03
Activation ReLU
Regularization None
Regularization rate 0
DATA
Which dataset do you want to use?
Ratio of training to test data: 50%
Batch size: 10
FEATURES
Which properties do you want to feed in?
3 HIDDEN LAYERS
6 neurons
8 neurons
4 neurons
OUTPUT
Test loss 0.005
Training loss 0.007
The outputs are mixed with varying **weights**, shown by the thickness of the lines.
Colors shows data, neuron and weight values.
Show test data


Iterations 001,422

Learning rate 0.03

Activation ReLU

Regularization None

Regularization rate 0

DATA
Which dataset do you want to use?

Ratio of training to test data: 50%

Batch size: 10

REGENERATE

FEATURES
Which properties do you want to feed in?

4 HIDDEN LAYERS

8 neurons 8 neurons 8 neurons 4 neurons

OUTPUT
Test loss 0.002
Training loss 0.007

Colors shows data, neuron and weight values.

Show test data

The 2nd screenshot above was the best run of any of my experiments, and it was the most stable. Consistently held at 0.002 test loss. As you can see from the Output graph, the boundary settles nicely around the blue points.

Overall, it seemed the rectified linear unit (ReLU) function, and the addition of multiple hidden layers, aided greatly in the creation of a stable, complicated model to account for a pretty intricate data set. As you can kind of see from the screenshot, the neurons create decision boundaries so that, when all aggregated together, it allows for the nice decision boundary around the spiral curve in the middle of the data set.

Question 2

2.1 Decision tree

Parameters:

Max_features: 250

Min_samples_split: 5

Class_weight: “balanced”

Testing error: 0.16

Max_features: The number of features considered when looking for the best split. So for 50, it will consider 50 different features at each split

Min_samples_split: The minimum number of samples required to split an internal node. In other words, there have to be 5 samples reached at a particular node in order for it to be split into child nodes

Class_weight: Since I selected balanced, it uses the values of y to automatically adjust weights inversely proportional to class frequencies.

2.2 K Nearest Neighbors

5 different values of k:

training error for KNN, k=3:

0.0133040735153

test error for KNN, k=3:

0.0553064275037

\\\\\\\\

training error for KNN, k=4:

0.0193389109861

test error for KNN, k=4:

0.0568011958146

\\\\\\\\

training error for KNN, k=7:

0.0257852146482

test error for KNN, k=7:

```
0.0582959641256
\\\\\\\\
training error for KNN, k=5:
0.0208476203539
test error for KNN, k=5:
0.0553064275037
\\\\\\\\
training error for KNN, k=6:
0.0241393498834
test error for KNN, k=6:
0.0612855007474
\\\\\\\\
['9.0000' '6.0000' '3.0000' ..., '4.0000' '0.0000' '1.0000']
```

Best train/test error: k = 3

This is the best option, because the data is distributed in such a way that just having a nearest neighbor algorithm (i.e. k=1) is not sufficient to correctly classify a point, as there is some noise. However, with k=3 there was slightly lower training error than k=5, meaning the classifying is much more successful when you can make sure there are a plurality of a certain label nearby.

The performance has a decent error rate, but not perfect, so I would imagine there is some overlap in the clusters. This would explain the need for having k > 1.

Weighing the distance would allow for closer neighbors to be weighed more heavily than farther away neighbors. This would cut down on possible noise, which is a problem with this data set, and the accuracy would go up (and the test error would go down), improving performance.

2.3 Support Vector Machine

```
goa5t@Ubuntu-goa5t:~/Desktop/HW6$ python number_recognition.py svm image_data/zip.train image_data/zip.test
test error for svm, kernel = rbf, C=3:
0.0503238664674
```

```
\\\\\\\\
test error for svm, kernel = linear:
0.0737419033383
\\\\\\\\
test error for svm, kernel = poly, deg = 2:
0.0572994519183
\\\\\\\\
test error for svm, kernel = poly, deg = 5:
0.0642750373692
\\\\\\\\
test error for svm, kernel = poly, deg = 6:
0.0847035376183
\\\\\\\\
test error for svm, kernel = poly, deg = 8:
0.160438465371
\\\\\\\\
['9.0000' '2.0000' '3.0000' ..., '4.0000' '0.0000' '1.0000']
```

So it looks like changing the C value has a bigger impact than the kernel. On initial run, "poly" performed the best, but on this run, rbf performed better with a new C value. Next run:

```
goa5t@Ubuntu-goa5t:~/Desktop/HW6$ python number_recognition.py svm image_data/zip.train image_data/zip.test
test error for svm, kernel = rbf, C=3:
0.0503238664674
\\\\\\\\
test error for svm, kernel = rbf, C=5:
0.0483308420528
```

```
\\\\\\\\
test error for svm, kernel = poly, deg = 3, C=3.0:
0.0468360737419
\\\\\\\\
test error for svm, kernel = poly, deg = 3, C=5.0:
0.0468360737419
\\\\\\\\
test error for svm, kernel = poly, deg = 4, C=3.0:
0.0468360737419
\\\\\\\\
test error for svm, kernel = poly, deg = 4, C=5.0:
0.0458395615346
\\\\\\\\
['9.0000' '6.0000' '3.0000' ..., '4.0000' '0.0000' '1.0000']
```

Increasing the C value definitely has a positive impact. In other words, the hyperplane must be smaller in order to correctly classify all the points, while it hasn't quite reached the point of overfit yet.

One more run of 6 different models:

```
goa5t@Ubuntu-goa5t:~/Desktop/HW6$ python number_recognition.py svm image_data/zip.train image_data/zip.test
test error for svm, kernel = rbf, C=7:
0.0493273542601
\\\\\\\\
test error for svm, kernel = rbf, C=5:
0.0483308420528
\\\\\\\\
test error for svm, kernel = poly, deg = 5, C=5.0:
0.0468360737419
\\\\\\\\
test error for svm, kernel = poly, deg = 5, C=7.0:

0.0463378176383

\\\\\\\\

test error for svm, kernel = poly, deg = 4, C=7.0:

0.0458395615346

\\\\\\\\

test error for svm, kernel = poly, deg = 4, C=5.0:

0.0458395615346

\\\\\\\\

['9.0000' '6.0000' '3.0000' ..., '4.0000' '0.0000' '1.0000']
```

It looks like a 4th degree poly is the best kernel, with a “large” C value.

In theory, SVM seems like it would be a difficult model to use on this data set. As we’ve discovered before, there is a decent amount of noise (i.e. outliers) which would make the tuning of the hyperplane difficult. Also, because the data sets overlap, it would require some sort of dimensional projection in order to adequately separate the points with a hyperplane.

However, it seems like the testing error is fairly low, and performs comparably to KNN, if not better. I think this might be due to there being noise, but tweaking the value of C allows for the hyperplane to account for most outlier points. SVM seems to perform well enough here.

2.4 PCA

```
PCA, features = 2

test error for KNN, k=3:

0.500747384155

PCA, features = 200

test error for KNN, k=3:

0.0533134030892
```

PCA, features = 150

test error for KNN, k=3:

0.0548081714001

The advantage of using PCA is to reduce the number of dimensions, particularly ones that don't hold relevant data. When removing 56 dimensions, the test error was not too much worse than when using KNN with full dimensions. The advantage is performance; you don't have to account for all dimensions, while still getting an accurate (or similarly accurate) model.

PCA_SVM

PCA, features = 200

test error for svm, kernel = poly, deg = 4, C=7.0:

0.0667663178874

PCA, features = 150

test error for svm, kernel = poly, deg = 4, C=7.0:

0.0572994519183

PCA, features = 50

test error for svm, kernel = poly, deg = 4, C=7.0:

0.0587942202292

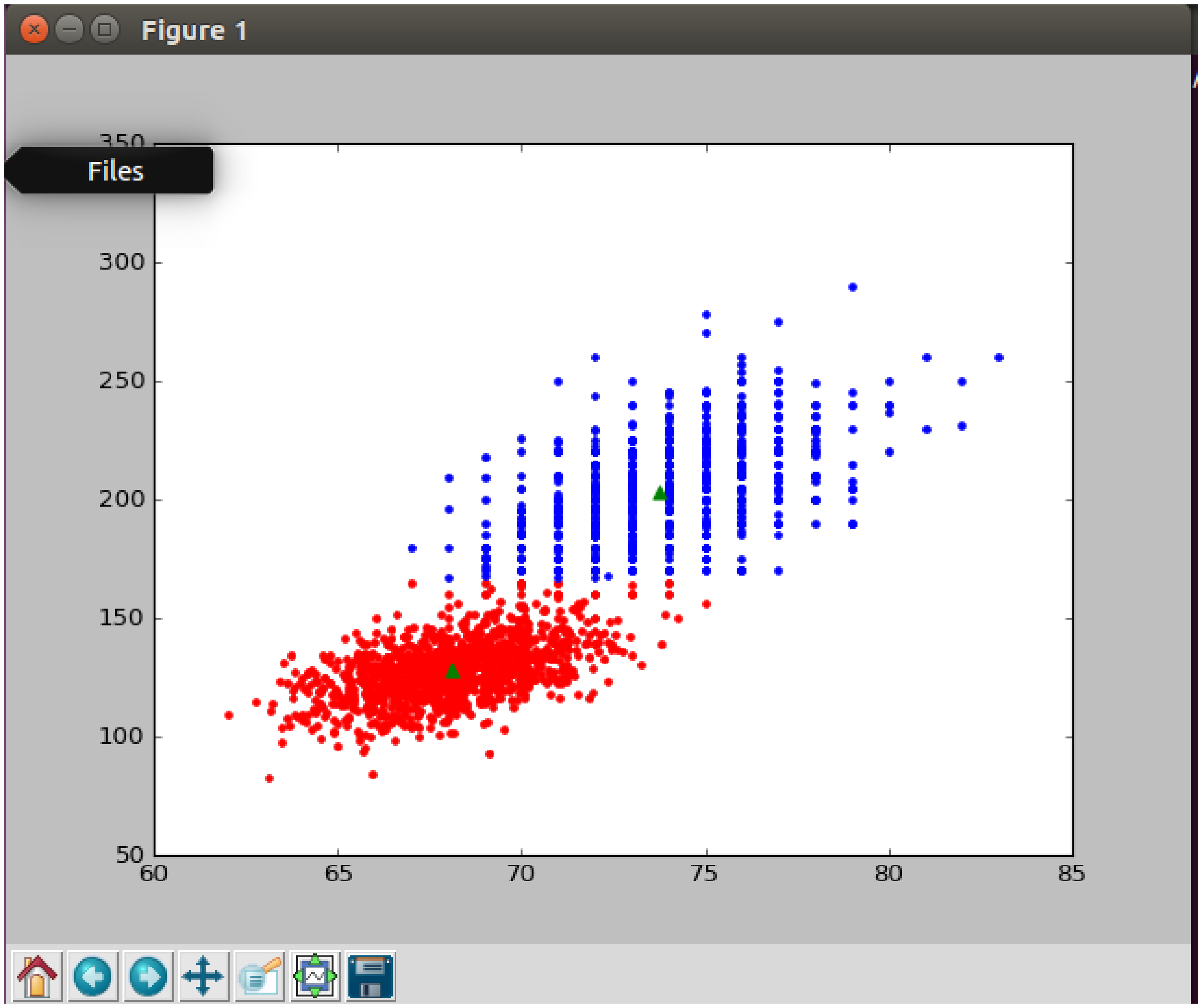
(very, very fast)

The speed at which the model is created is much, much faster than before. This is simply due to the decreased number of dimensions, meaning SVM doesn't have to have data points projected onto different dimensions in order to find an appropriate hyperplane, improving run time.

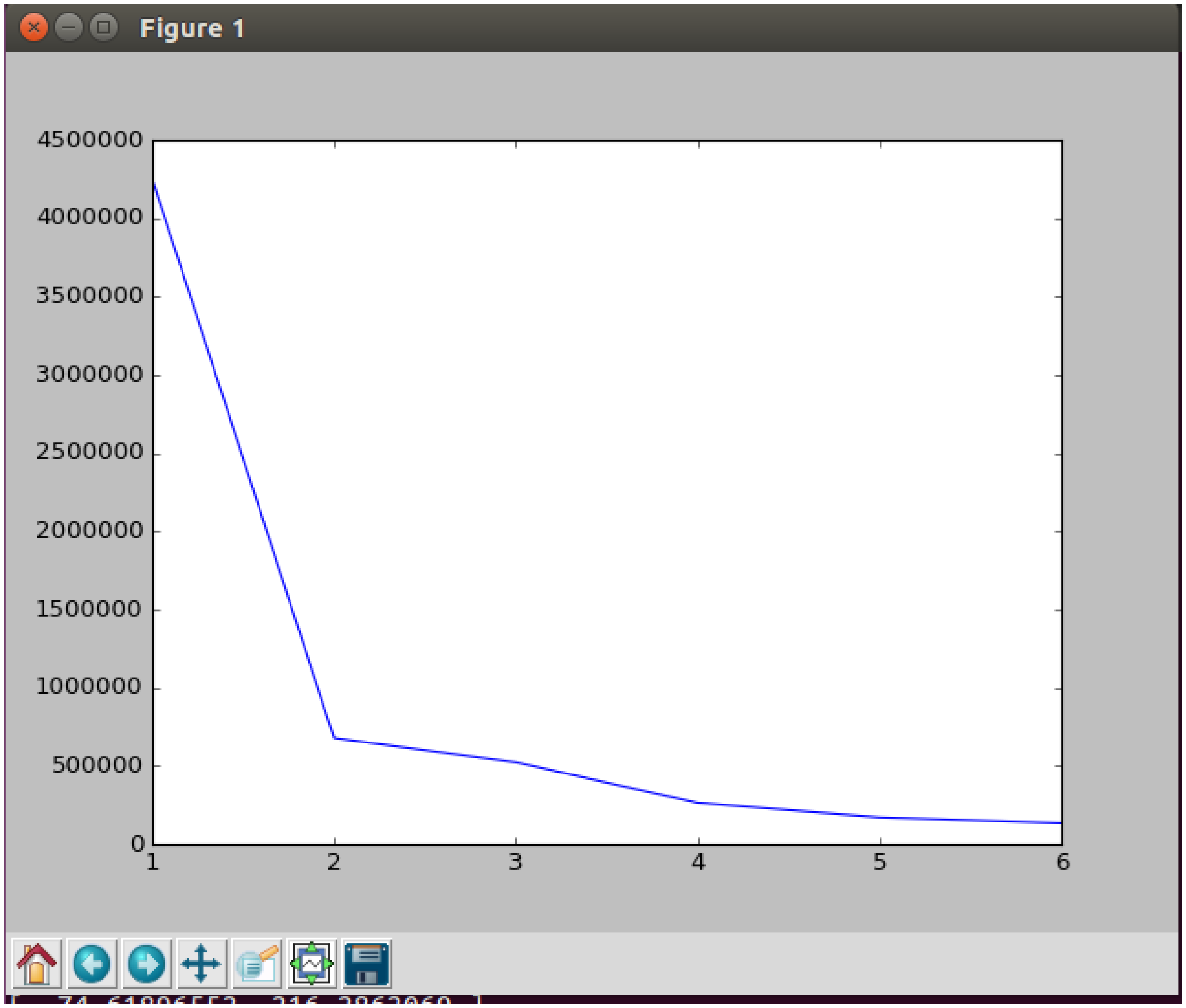
It also seems the accuracy isn't tremendously set back by using PCA. I would imagine with some testing we could find a decent test accuracy using this technique (i.e. with PCA) that would take a fraction of the time to run.

Question 3

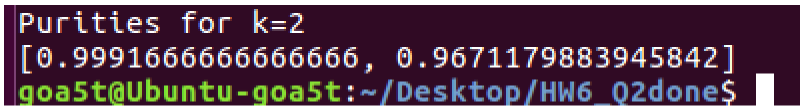
Q3:



KneeFinding graph:



Q5: Purity of k=2:



```
Purities for k=2
[0.9991666666666666, 0.9671179883945842]
goa5t@Ubuntu-goa5t:~/Desktop/HW6_Q2done$
```