Jason Teng
Programming Assignment 2 Writeup

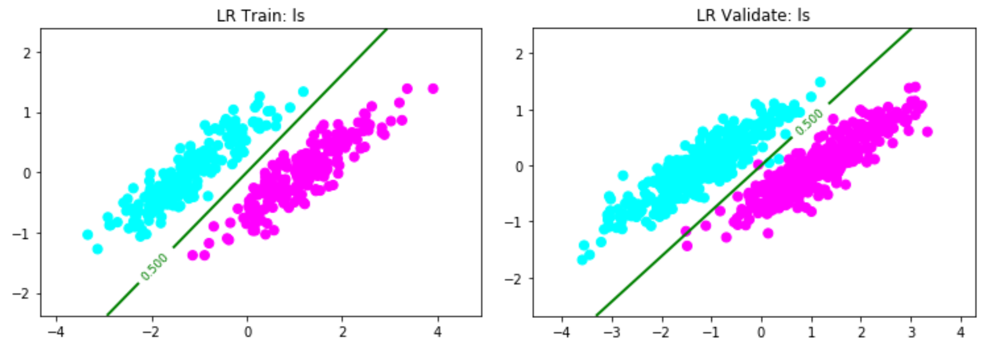
This assignment was completed with the help of Donald Hamnett, as well as using example kernels from a powerpoint given by Jiayu Zhou from MSU on SVMs and kernel methods.

1. Logistic Regression

I implemented logistic regression by adapting the gradient descent solution I had in PA1. I derived the analytic gradient of the given cost function, which turned out to be

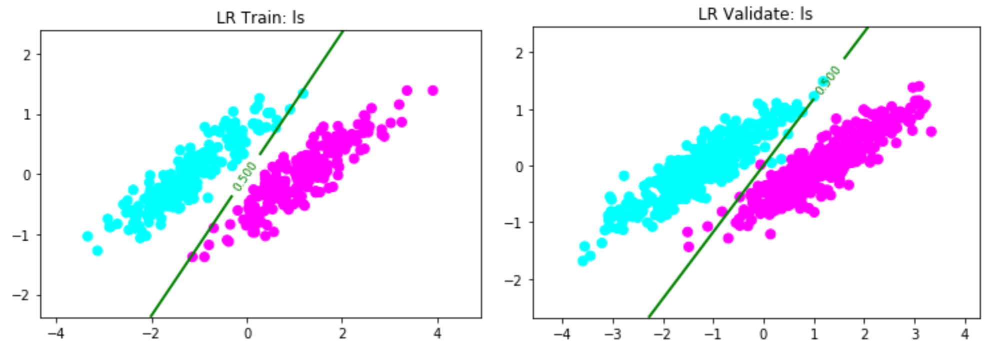
$$2l\|w\| - \sum_{j=1}^{n} y_j x_{ij} e^{y_j(-\boldsymbol{x}^T \boldsymbol{w})}$$

With this, I was able to generate logistic regression models with low error. For example, when run with a learning rate of 0.01, and a lambda of 0 for 1000 iterations on the linearly separable training data, my model had only 4 errors when tested against the validation data. The images below show this result:



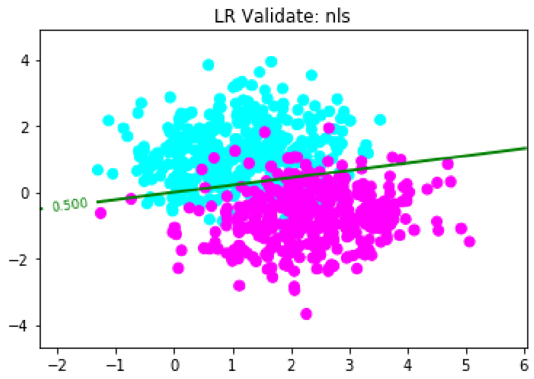
```
errors in logistic regression on ls data: 4
```

The above results were obtained when lambda was set to 0. When lambda is increased, to 0.1 for example, the results change accordingly:



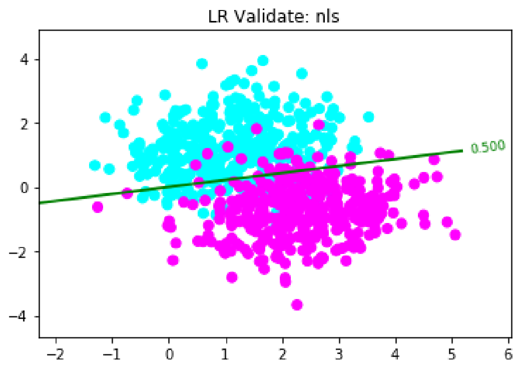
```
errors in logistic regression on nonlin data: 60
```

Below is a graph showing the results when lambda = 0.1 on the nls data:



```
errors in logistic regression on nls data: 87
```

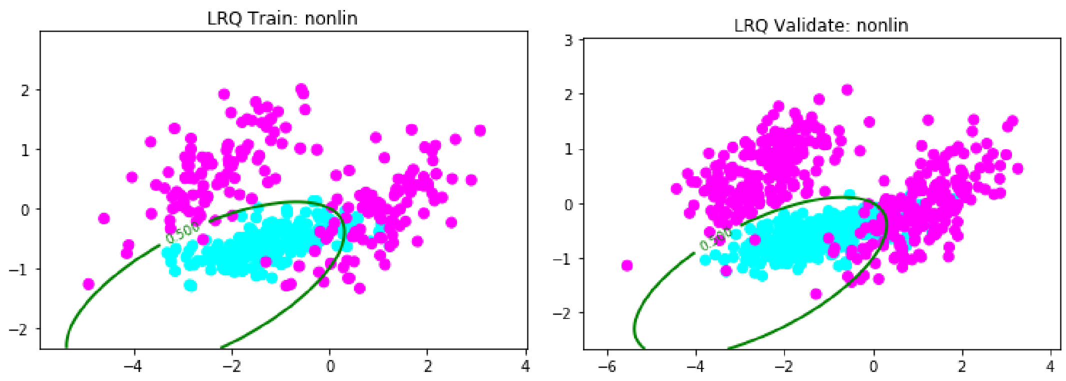
and with lambda = 0.001:



```
errors in logistic regression on nls data: 89
```

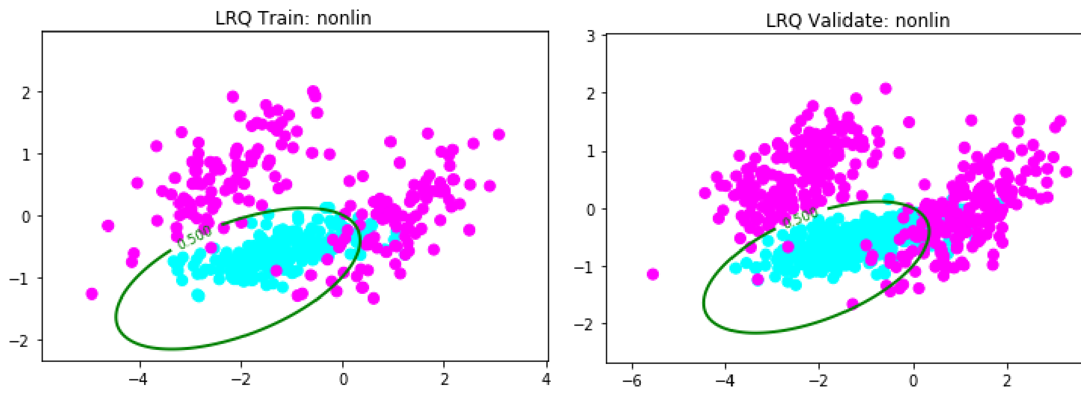
As we can see, there is little difference in the results on the non-linearly separable data when lambda is changed. In fact the accuracy decreases when lambda is reduced.
In order to allow for higher order polynomial basis functions, I simply extended the input vector, and also extended the weight vector accordingly; the regression function remained unchanged. Results are shown below:

Nonlin: lambda = 0.001, lr = 0.001



```
errors in logistic regression on nonlin data: 60
```

If lambda is increased to 0.1, the graphs change:

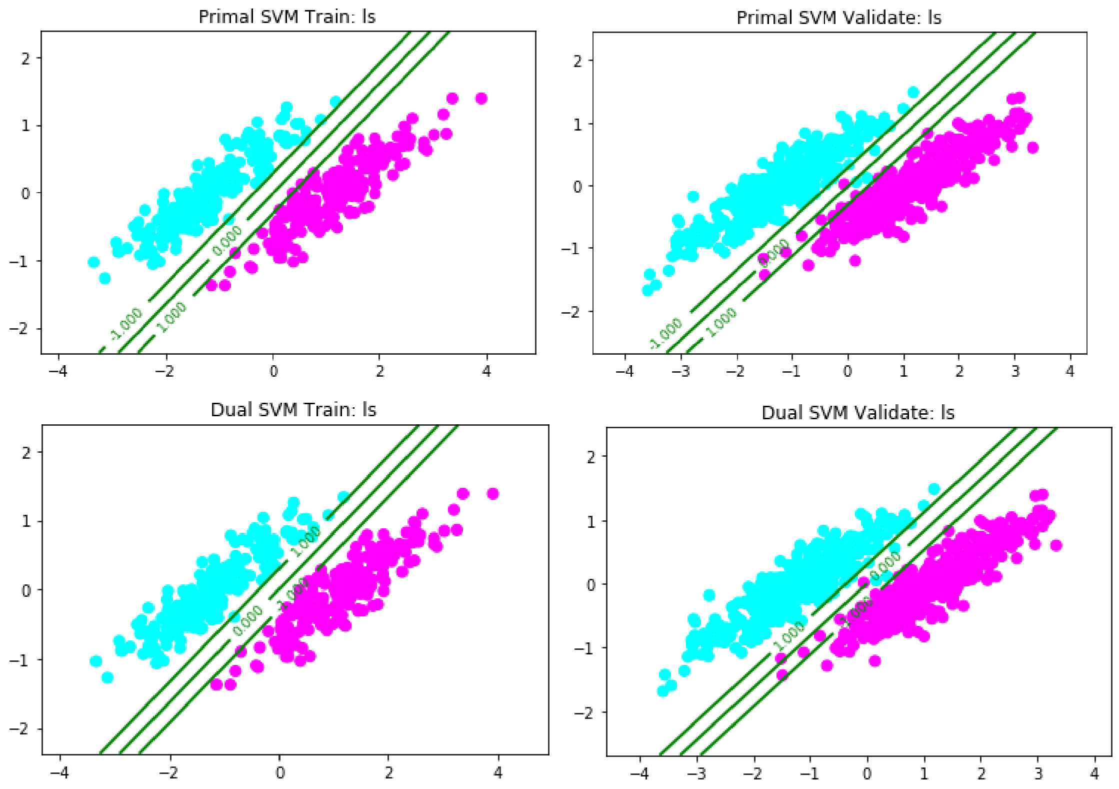


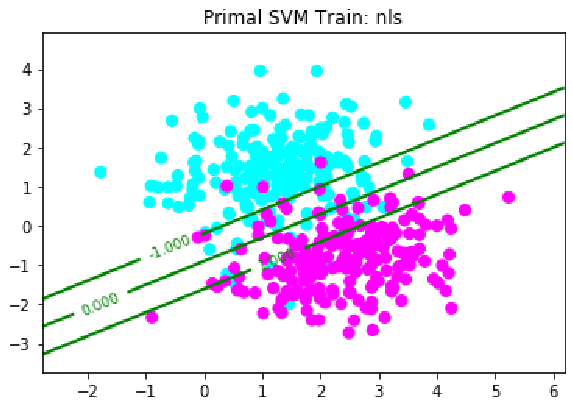
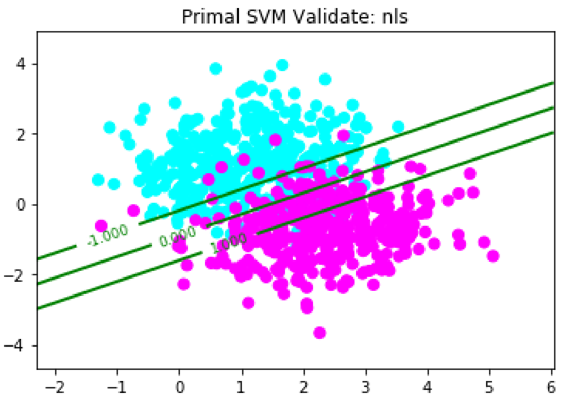
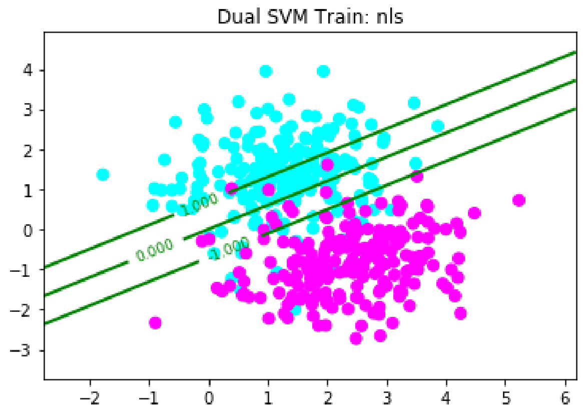
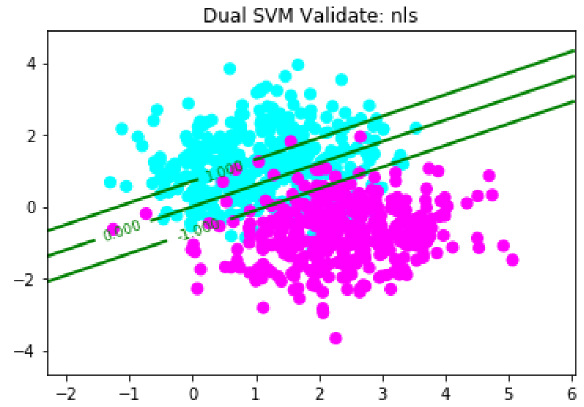
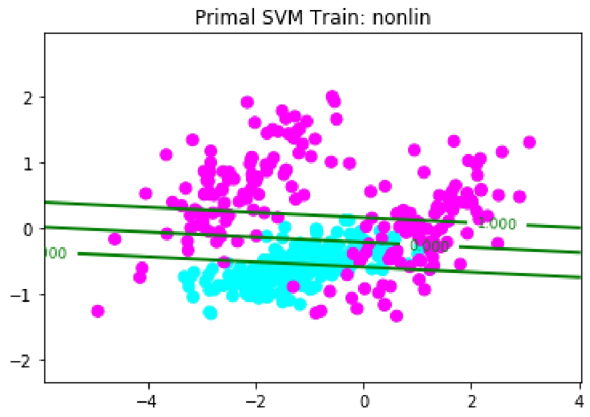
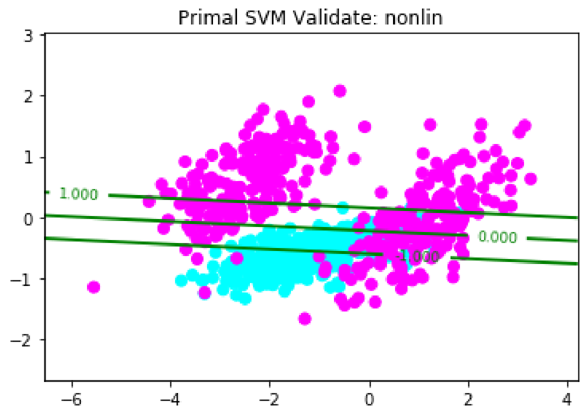
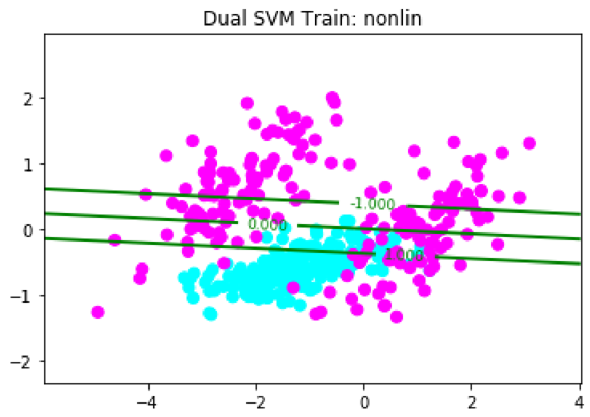
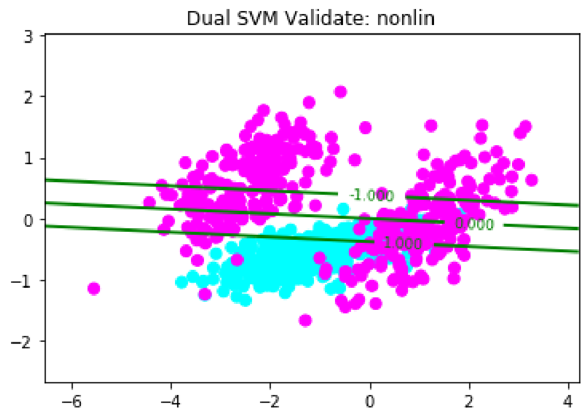
```
errors in logistic regression on nonlin data: 62
```

Here we see that increasing lambda allows the model to form a tighter circle of prediction, at the expense of slightly more errors.

2. Support Vector Machines

Below are graphs showing the performance of both primal and dual SVM implementations on all three data sets (all models generated with C = 1/(2*lambda) with lambda = 0.001):




Primal SVM Train: nls
-1.000
0.000
1.000

Primal SVM Validate: nls
-1.000
0.000
1.000

Dual SVM Train: nls
1.000
0.000
-1.000

Dual SVM Validate: nls
1.000
0.000
-1.000

Primal SVM Train: nonlin
1.000
0.000

Primal SVM Validate: nonlin
1.000
0.000
-1.000

Dual SVM Train: nonlin
-1.000
0.000
1.000

Dual SVM Validate: nonlin
-1.000
0.000
1.000

While both the primal and dual SVMs produced similar results for the linearly separable data, when the data became no longer linearly separable, the two implementations produced different models, although neither was significantly more accurate than the other.

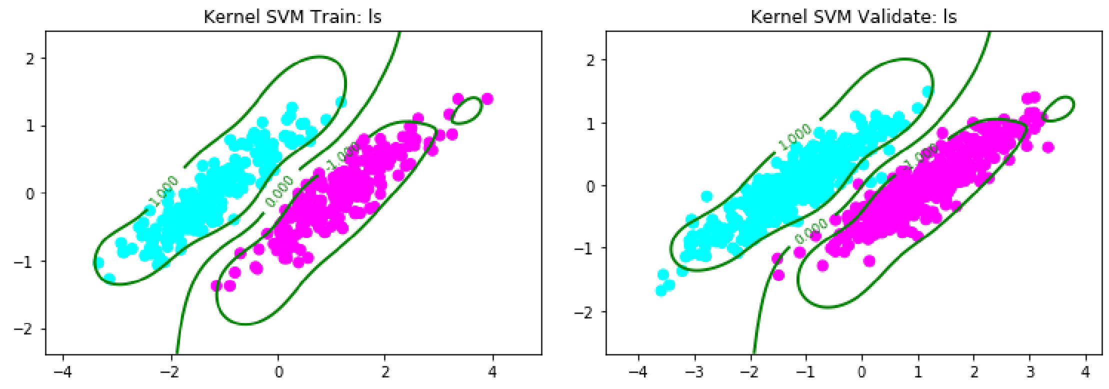
3. Kernel SVM

In order to extend my implementation to allow kernels, I simply took the dual SVM, and instead of using $x_i x_j$, I used $K(x_i, x_j)$, where K is a kernel function. When using a gaussian kernel of the form

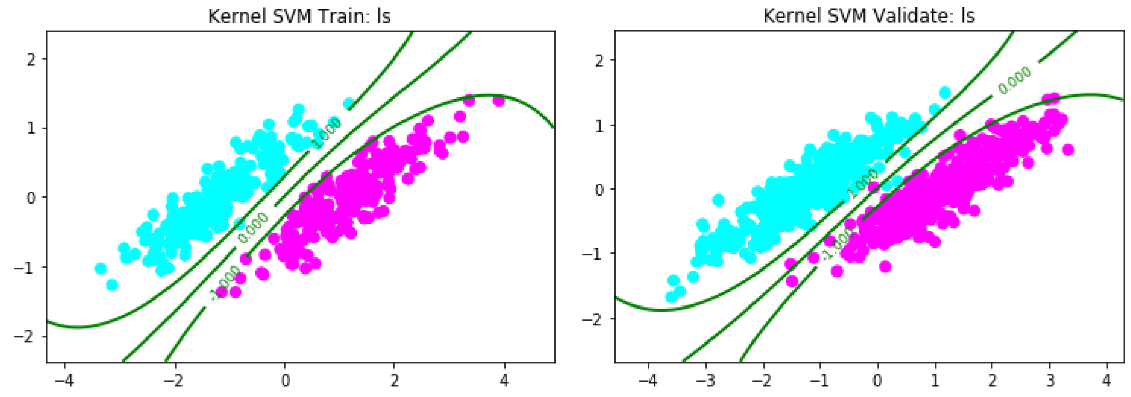
$$G(x_1, x_2) = e^{-\gamma \|x_1 - x_2\|^2}$$

I needed to experiment quite a bit with the value for γ, and the results are below:

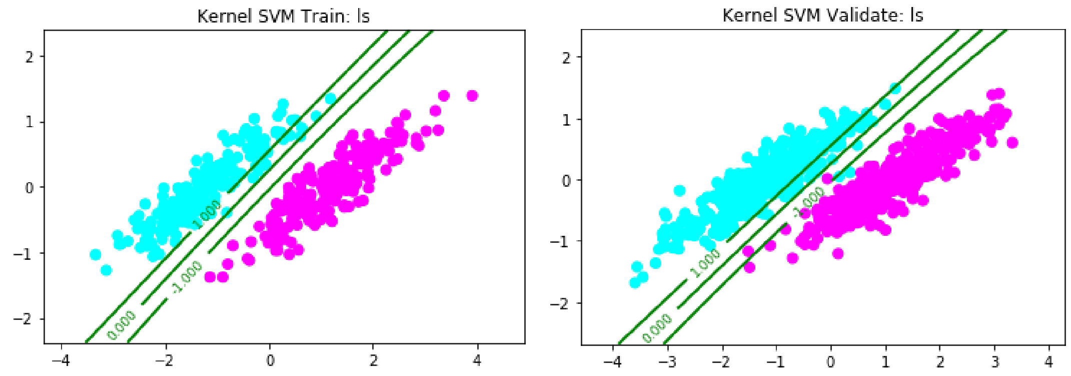
Gaussian Kernel: (γ = 1)



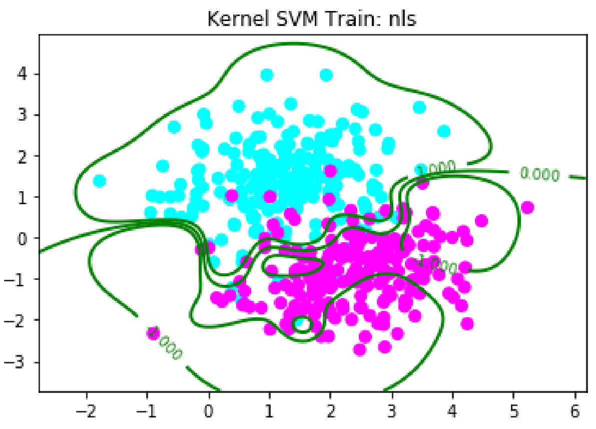
Gaussian Kernel: (γ = 0.1)

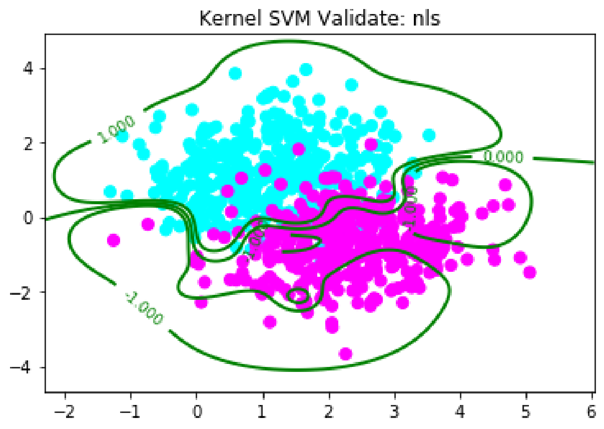


Gaussian Kernel: (γ = 0.01)

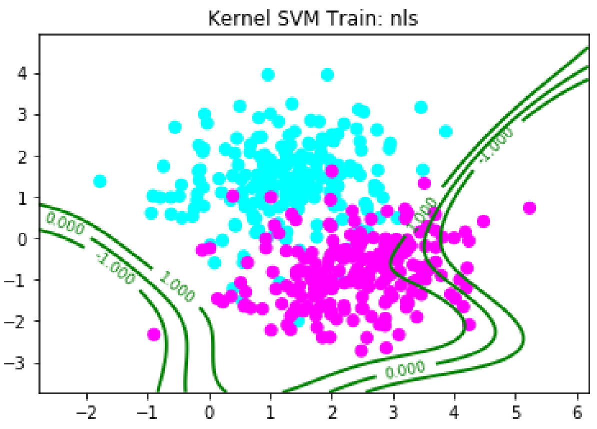


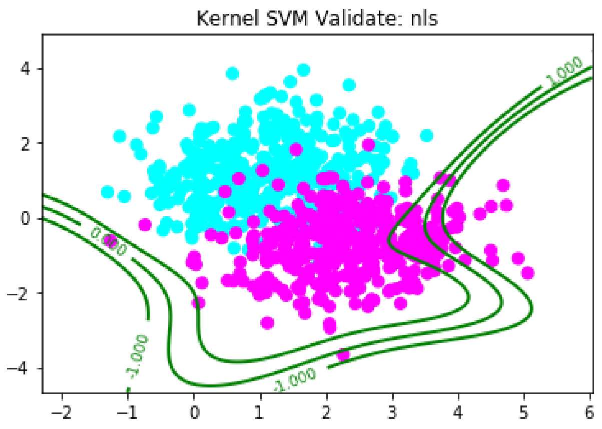
Gaussian Kernel: (γ = 1)



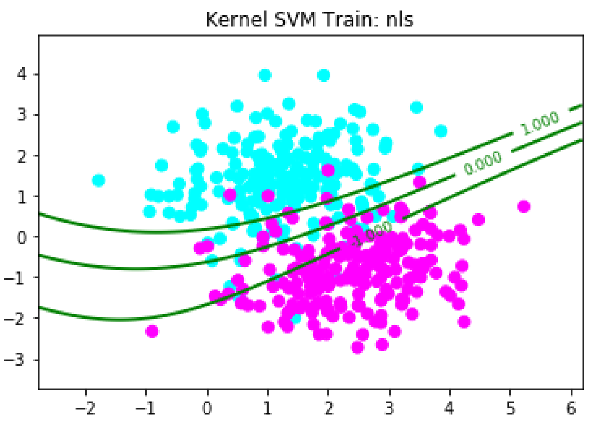


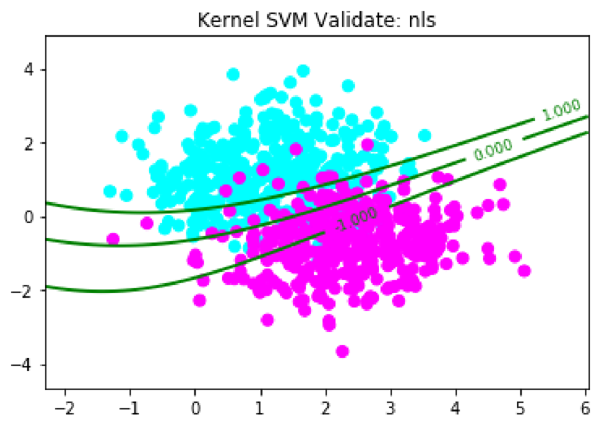
Gaussian Kernel: (γ = 0.1)



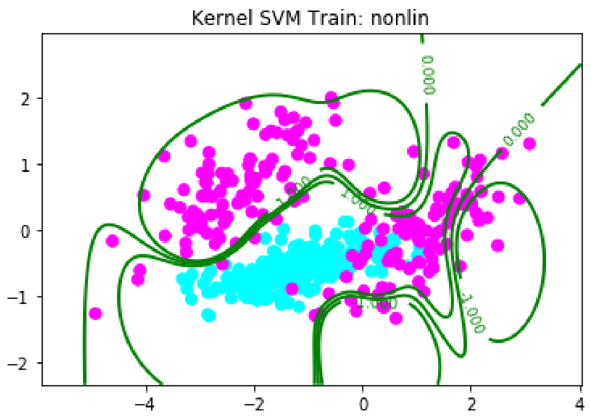


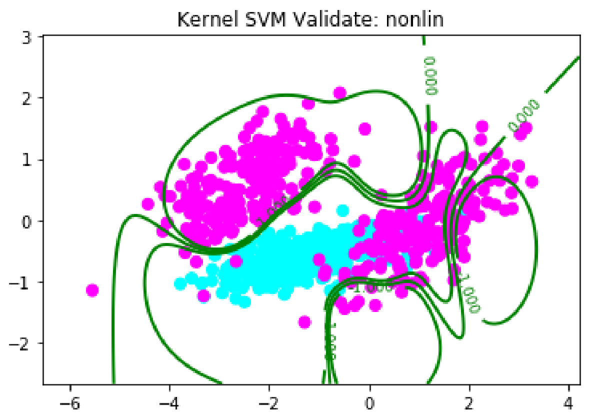
Gaussian Kernel: (γ = 0.01)



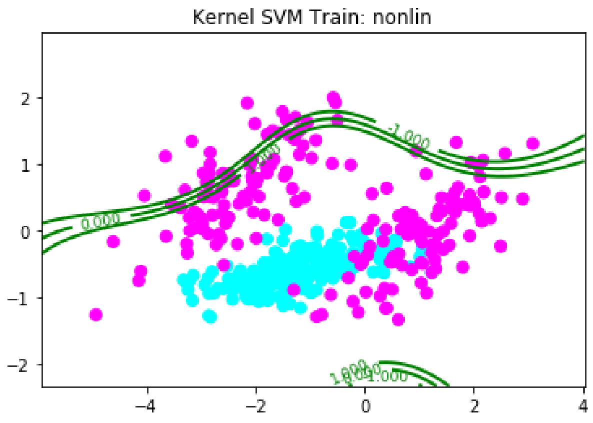


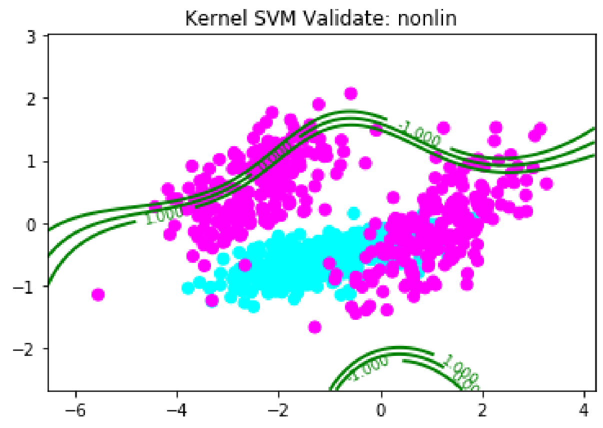
Gaussian Kernel: (γ = 1)



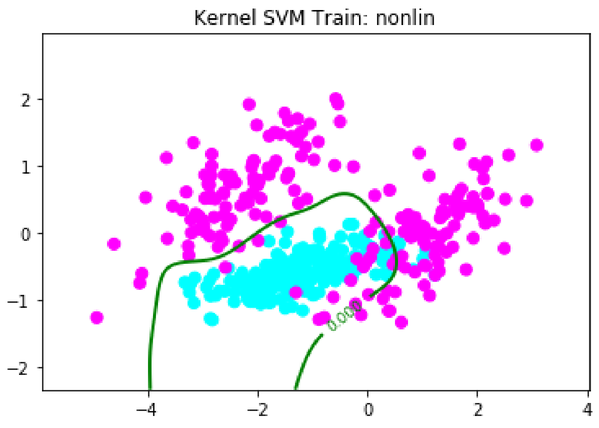


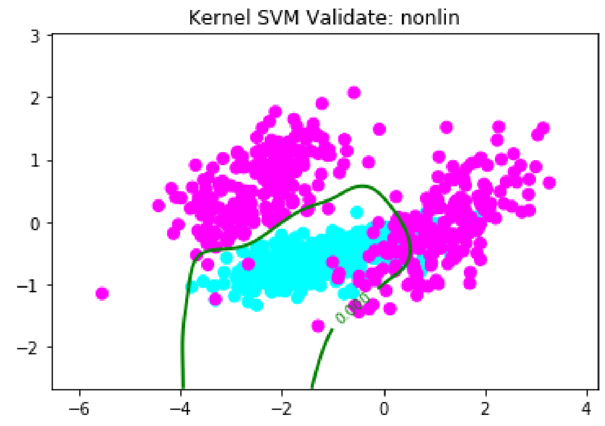
Gaussian Kernel: (γ = 0.1)





Gaussian Kernel: (γ = 5, C = 0.005)



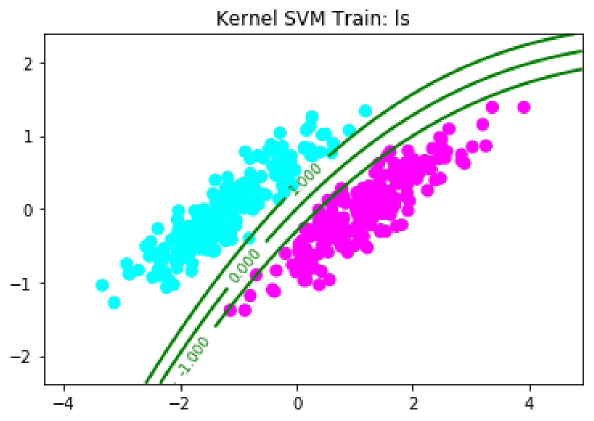


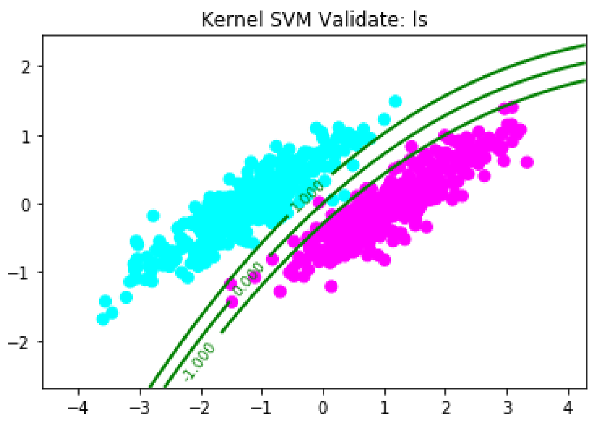
Here we see that as gamma increases, the amount of overfitting decreases. When it came to the non-lin data, it took quite a bit of modifications to find the correct parameters that would get a good fit, and ultimately I had to drastically lower C in order to get a reasonable model. In all other models (including the following models for the quadratic kernel), C was set to 500.

As for a second order polynomial kernel, I used the following:

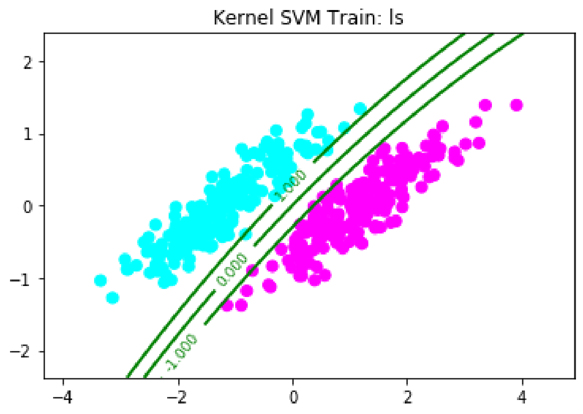
$$P(x_1, x_2) = (1 + a\boldsymbol{x}_1^T \boldsymbol{x}_2)^2$$

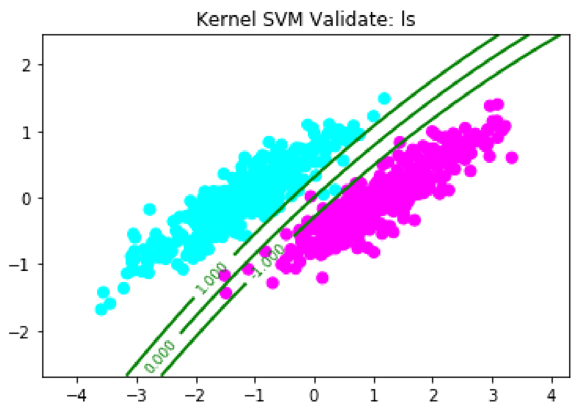
Quadratic Kernel: (a = 1)



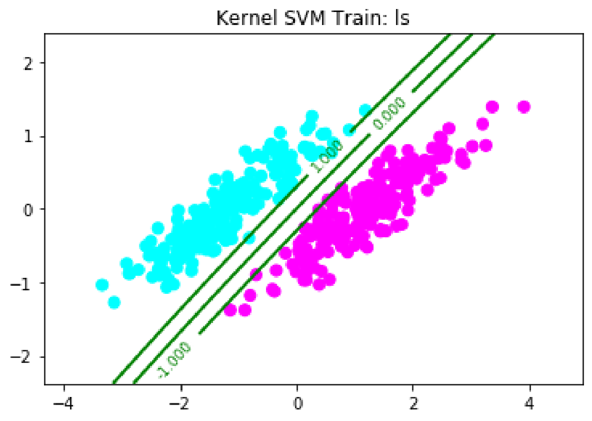


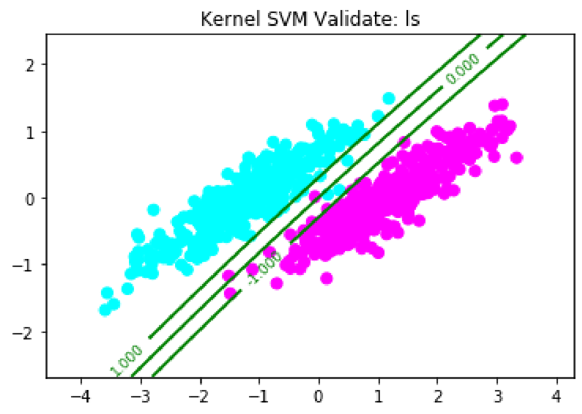
Quadratic Kernel: (a = 0.1)



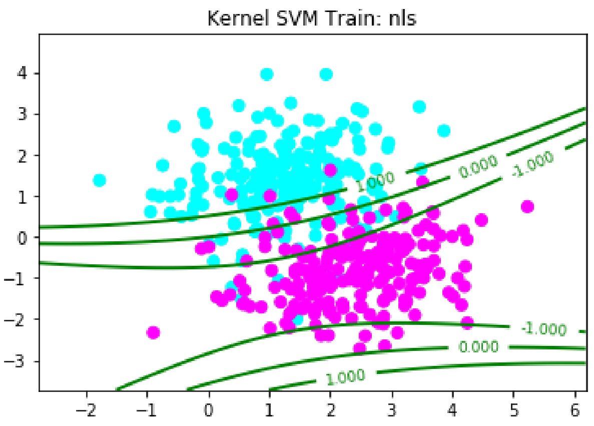


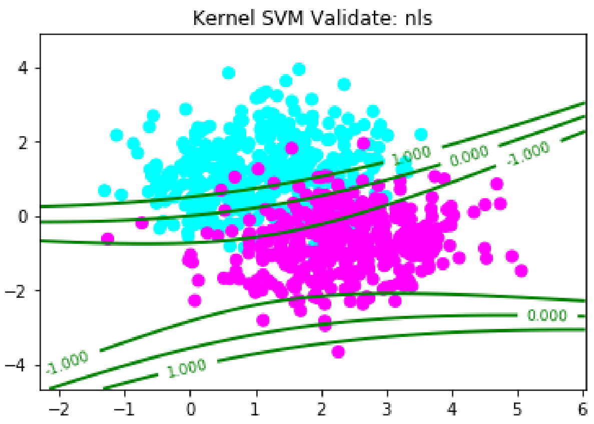
Quadratic Kernel: (a = 0.01)



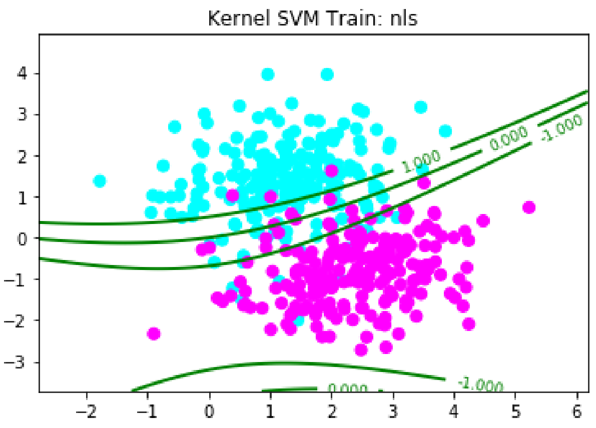


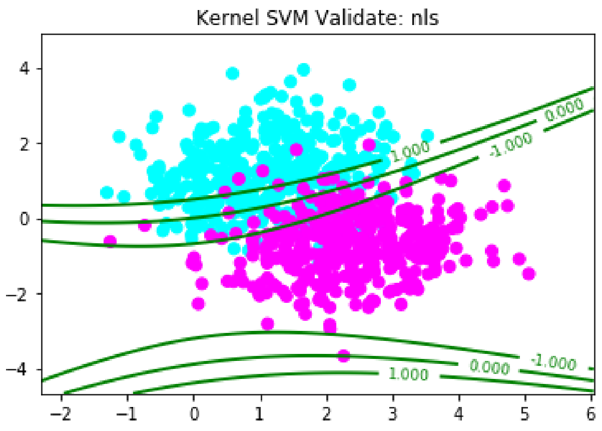
Quadratic Kernel: (a = 1)



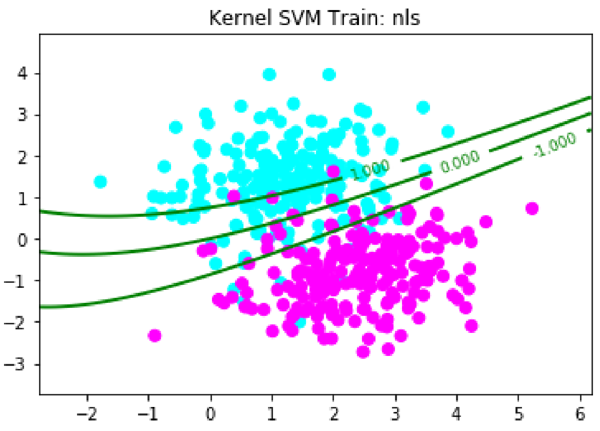


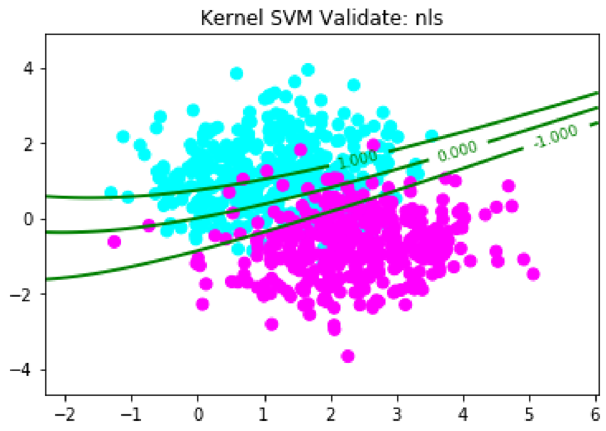
Quadratic Kernel: (a = 0.1)



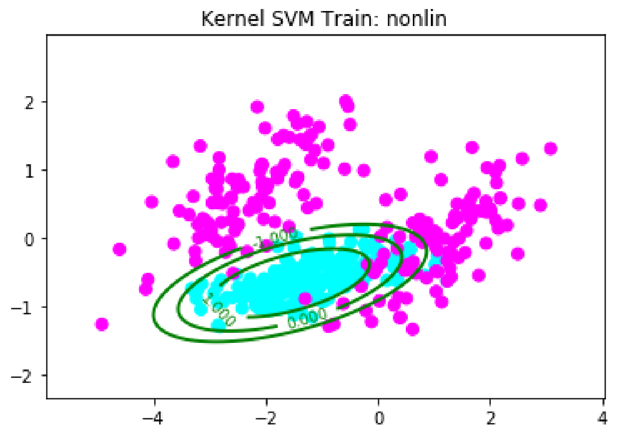


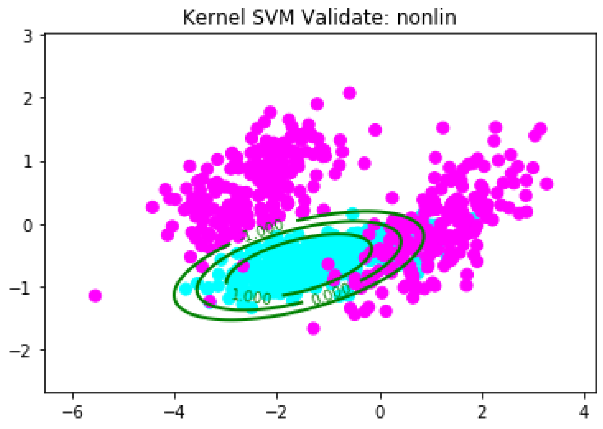
Quadratic Kernel: (a = 0.01)



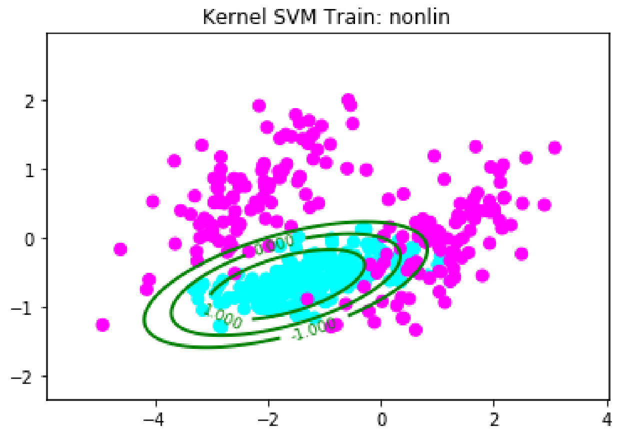


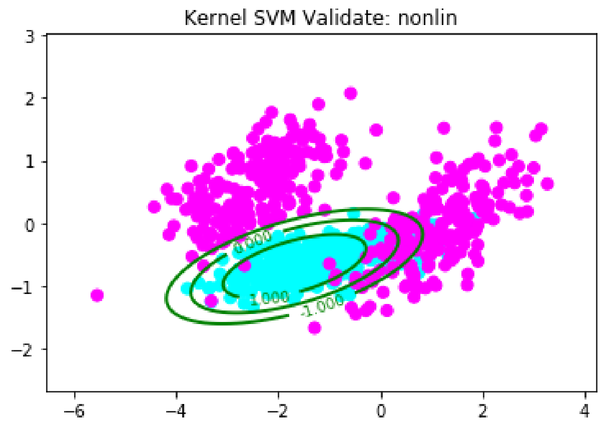
Quadratic Kernel: (a = 1)



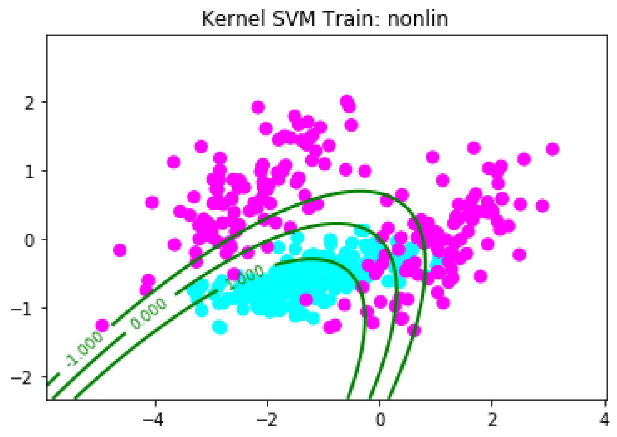


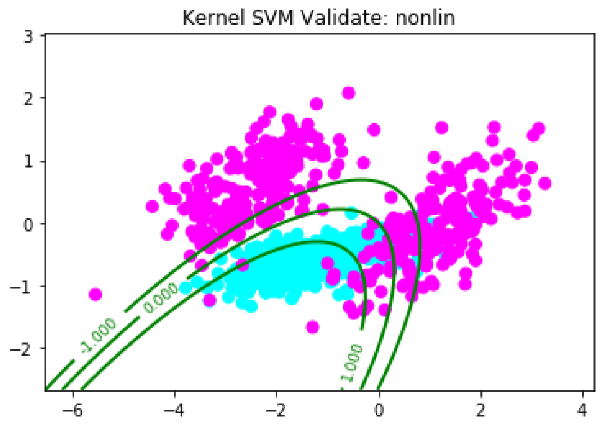
Quadratic Kernel: (a = 0.1)





Quadratic Kernel: (a = 0.01)





Here we see that decreasing the value of a, like with gamma, decreases overfitting, akin to lowering C. Overall, I found the quadratic kernel to perform better than the Gaussian kernel, both in terms of accuracy, as well as the speed at which the model was generated. Comparing with logistic regression, I found that the results were equally, if not more accurate, especially with regards to the quadratic kernel performing on the non-linear data.