**Machine Learning**

Homework 5

Report

**Name**: Sergiy Shelekh

**ID**: 0760822

**Email**: [proxitrone@gmail.com](mailto:proxitrone@gmail.com)

**Introduction**

This homework is composed of two major parts: Gaussian Process Regression and Support Vector Machine for multiclass classification. In the first part we are required to apply Gaussian Process Regression to predict the distribution of a function f and visualize the result, while in the second part we need to use libsvm package to play around with different SVM’s on 5 classes of MNIST dataset also performing a grid search for parameters and creating our own kernel. I decided to use MATLAB for this homework, since it’s very easy to work with matrices and vectors, it has a great debugger, and libsvm package is also available for MATLAB.

**Problem 1: Gaussian Process Regression**

In this problem we are given a 32x2 matrix, where the two columns represent our training data with labels. The main difference from regression we had before is that we are taking advantage of Kernels instead of explicitly defining feature space. We are using rational quadratic kernel to compute similarities between different points.

First, we compute covariance matrix for our training data, where each element is given by

And the quadratic kernel is of the form

,

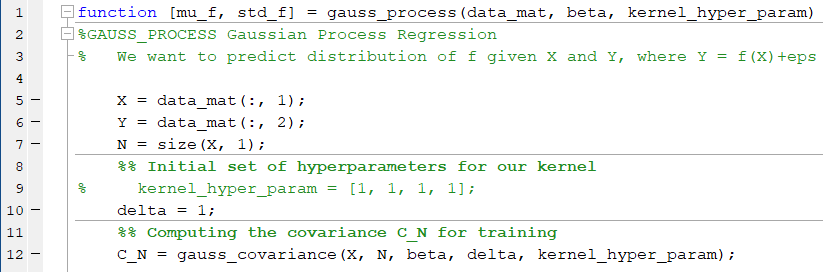
Where are the *hyperparameters* of this kernel.

Now, if we have our covariance matrix and get a new point N+1, for which we need to make a prediction, we first compute a vector **k**, which represents the difference between every training point and our new point and has elements defined by

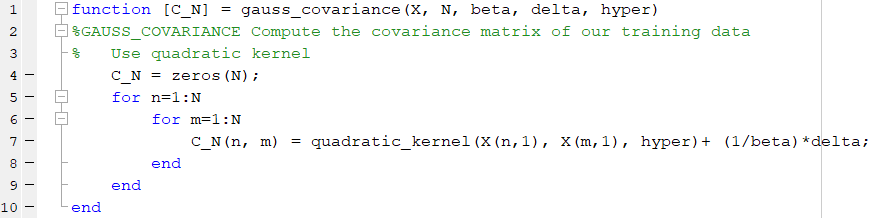
Then we get and can write the new covariance matrix as

Since everything is basically given by Gaussian distributions, our conditional probability of new datapoint w.r.t training data also a Gaussian with mean and variance given by:

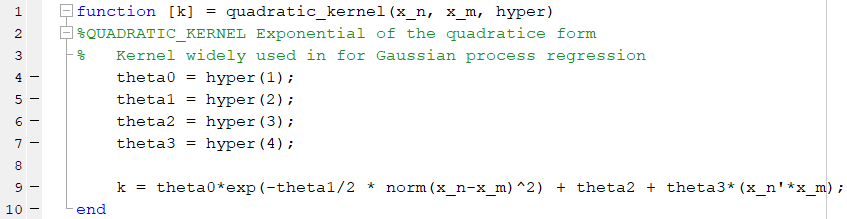
We now have everything we need to make an estimation of mean of function f, given an arbitrary set of initial hyperparameters [1, 1, 1, 1]



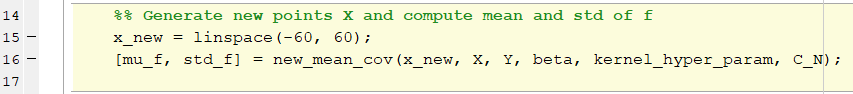
Covariance is computed element-wise, as in the definition of our kernel



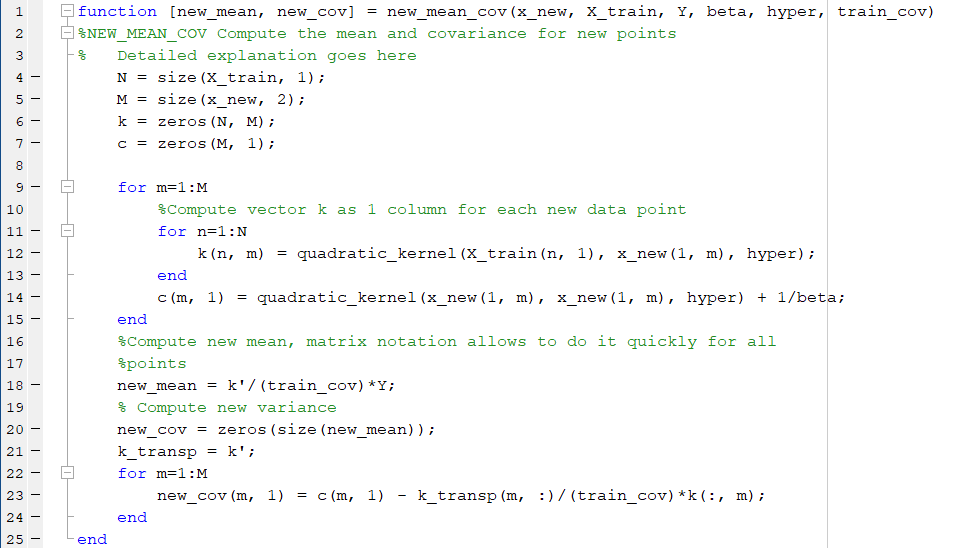
Kernel is also given by the definition above



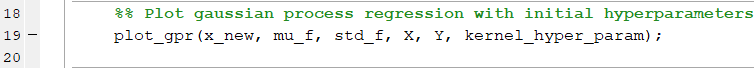
After obtaining the covariance matrix, we go on to generate new points and estimate the mean of function f



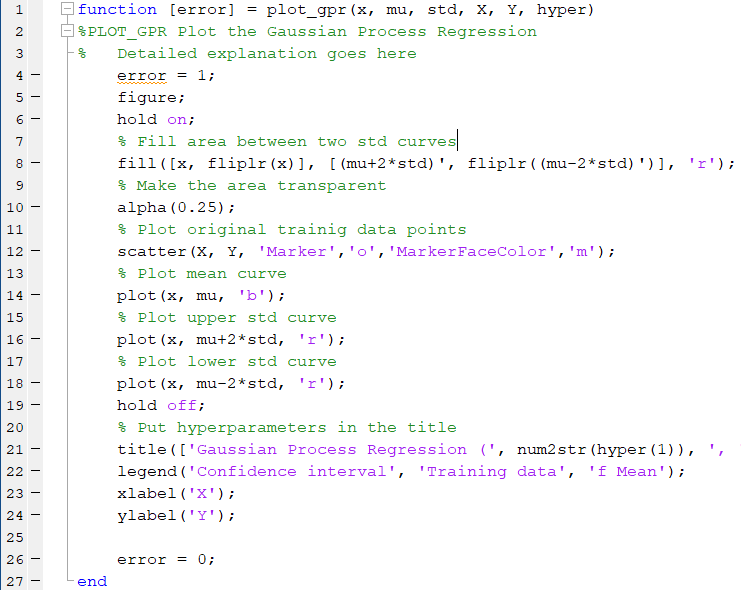
Mean and covariance are computed for all new points in this function, so we define **k** as a matrix, with each column representing 1 new datapoint

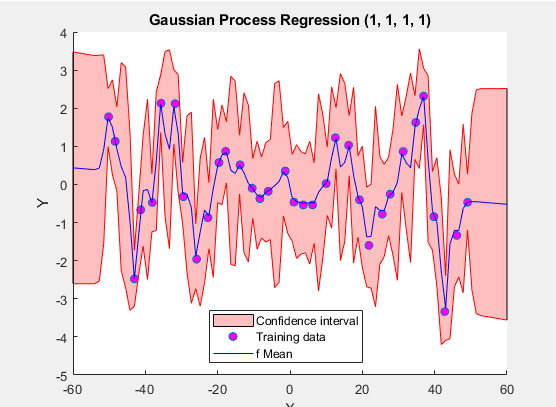


Now we have mean and covariance for each new datapoint, plot them together with our training data



We plot mean curve bounded by our 95% certainty region, which is given by the covariance values



Graph for the initial set of hyperparameters looks like 

We can clearly see, that there’s room for improvement and this brings us to learning the hyperparameters.

Basically, we are taking a log-likelihood function for our Gaussian process evaluated using standard form for multivariate Gaussian

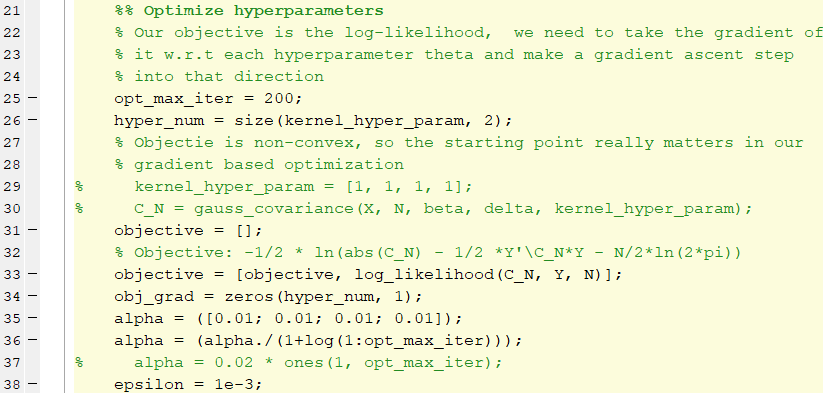
Next, we need to take the gradient of the above function w.r.t each hyperparameter

We have everything, except for , which can be found by taking derivative of our kernel w.r.t each

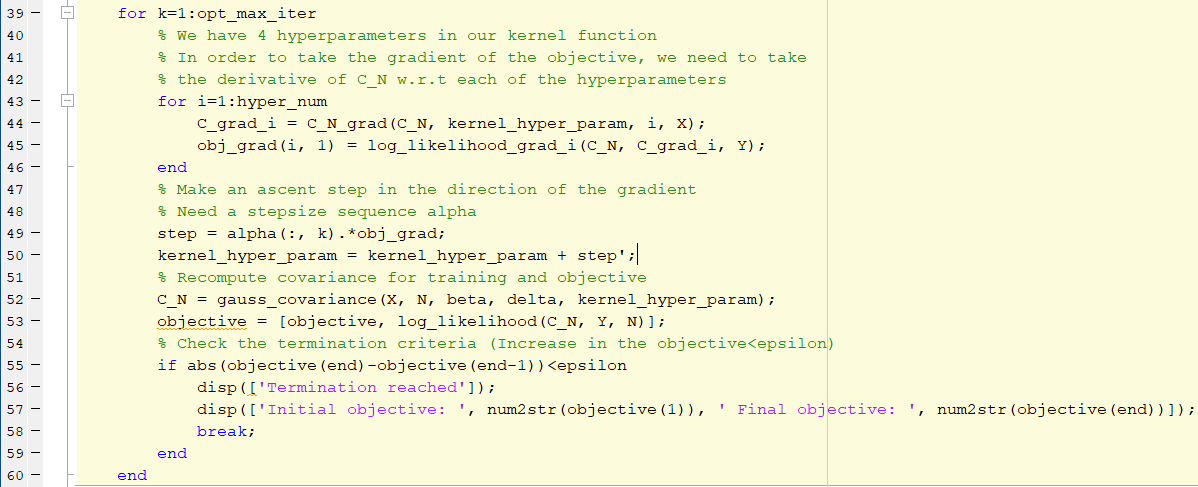
Deriving this kernel is not hard, we use a little bit of calculus and get four closed form solutions for each of the hyperparameters:

Now we can choose a step size and simply do a gradient ascent step in the direction of the gradient. New hyperparameters are given by

After a number of steps, we will reach a local optimal solution. Since our objective function (log-likelihood) is non-convex, this gradient-based optimization is very sensitive to our choice of initial hyperparameters and also the step size.

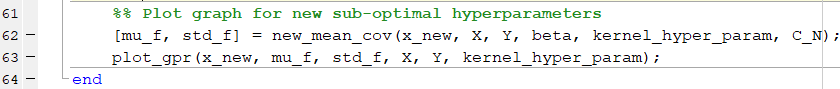


In the initialization part define a sequence of positive step sizes alpha, set initial step size (heuristic choice is 0.01, diverges for larger), compute initial objective, set termination criteria parameter to 0.001. Next, we start the optimization loop



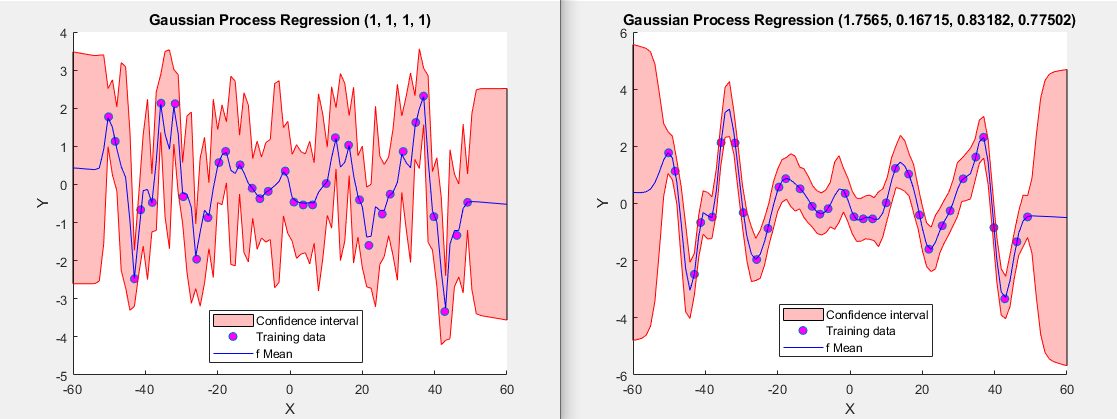
Again, we first take the gradient of our objective w.r.t each hyperparameter, according to the definition given above, then update our parameters, recompute covariance matrix and check the objective again. We terminate either on reaching the maximum number of iterations or the change in the objective is smaller than our parameter epsilon.

Finally, estimate new mean and covariance using sub-optimal hyperparameters and plot



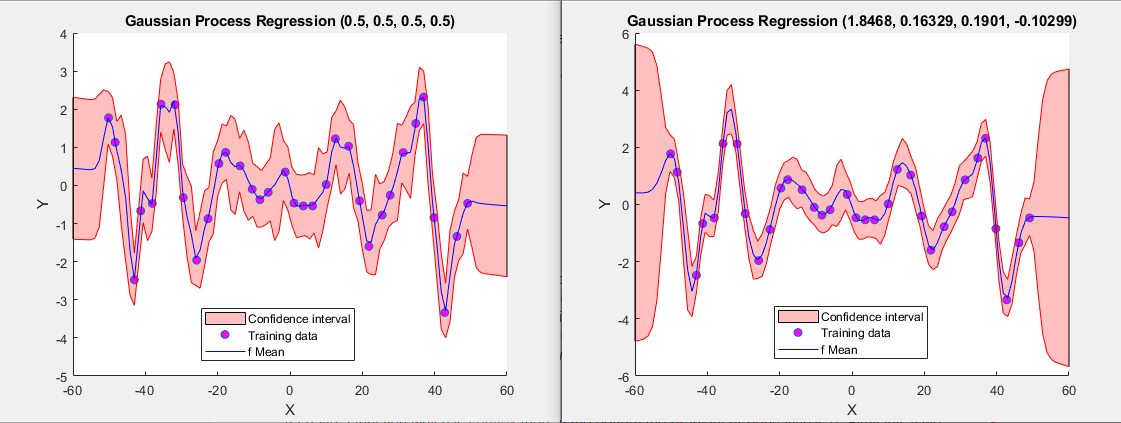
Compare results for different initial choice:

Average initial choice



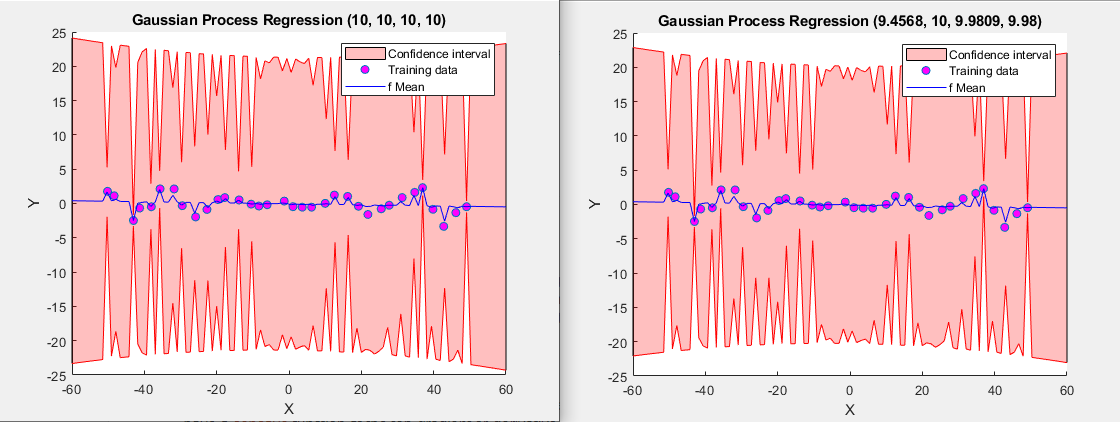
Smoothed curves, increased std at the ends, visible difference from initial

Good initial choice



Curves are smoothed again, initial point already gives plausible results

Bad initial choice



Hardly any difference

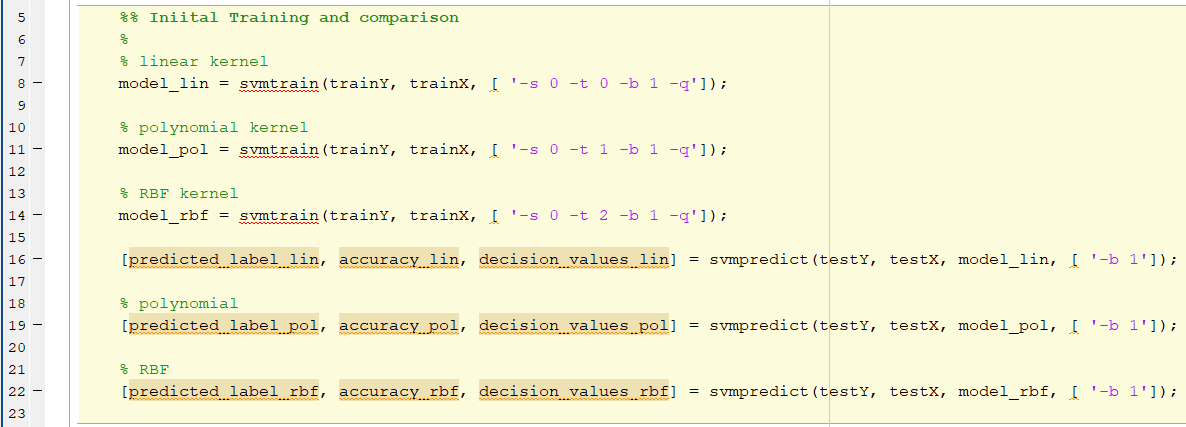
Overall, learning the hyperparameters seems quite unstable, easy to get too far from an optimal point, where we would make the covariance matrix very badly conditioned, which in turn will influence it’s inverse, leading to even larger errors. Interesting thing is that the gradient w.r.t last parameter is huge in the start (~800) compared to other parameters, which leads to the choice of a small initial step size.

**Problem 2: SVM on MNIST data**

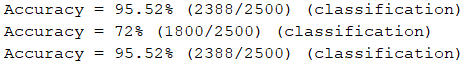
In this part, we are required to use different SVM models to perform multiclass classification (5 classes 0~4) on the MNIST dataset. LIBSVM library is very convenient, as it handles training and testing on it’s own, we just choose different parameters, such as kernel type (linear, polynomial, rbf, or arbitrary) and parameters present in each kernel. We start off with training three kernels using default parameters provided by libsvm. Kernels we are using are given by:

Default values in libsvm are ,

Using these defaults, we train and test three models:

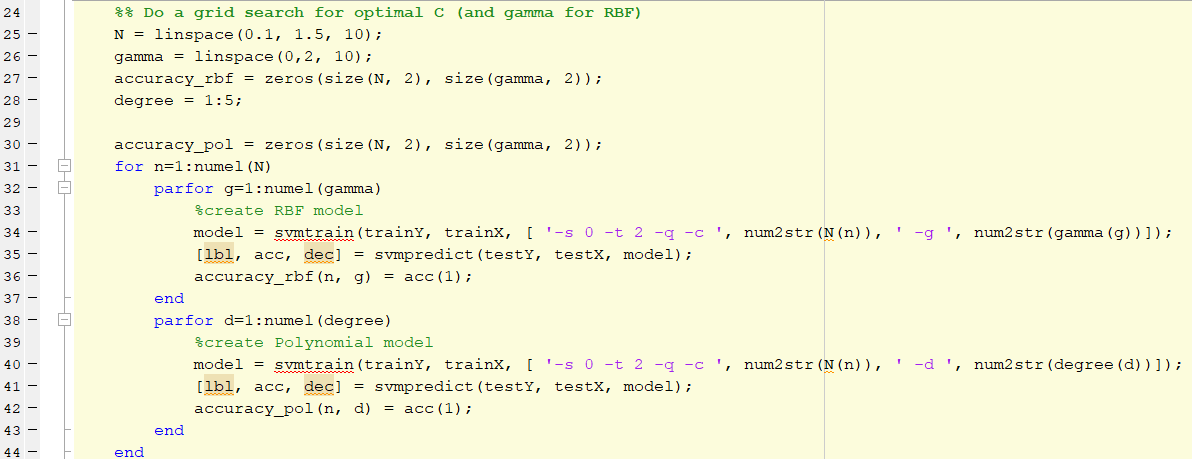


These default models already perform quite well (in order of appearance):

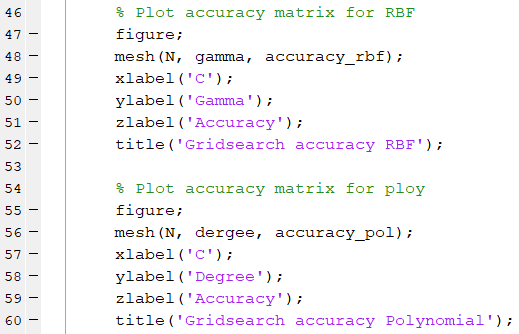


We see, that linear and RBF models get same accuracy on the test data 95.52%, while polynomial kernel model performs worse, at 72%

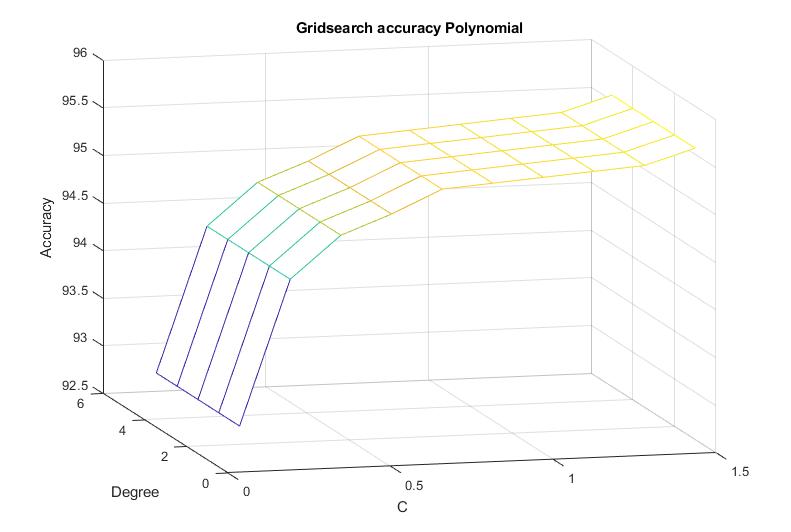
Let’s try to play around with these parameters (and also the cost C for C-SVC type) to try to find a better performing model. For that, we define ranges for our parameters and number of points in between to perform a grid search. Set C to be in range [0.1, 1.5] with 10 points, gamma in range [0, 2] with 10 points as well, and degree to be from 1 to 5. Train and test a SVM models with a set of parameters and record resulting accuracy. We don’t do a grid search for linear model

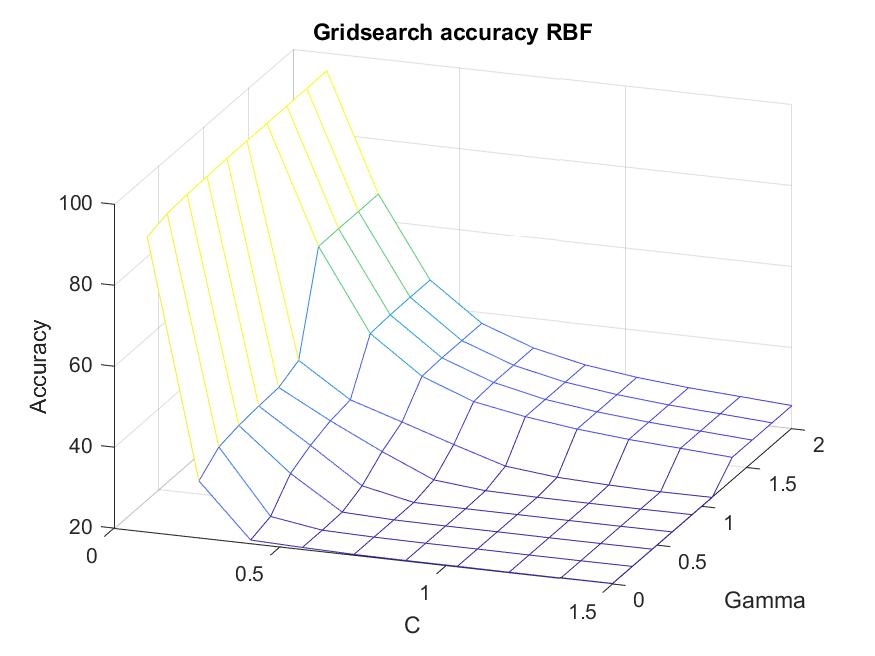


After the grid search, we plot accuracy obtained at each point



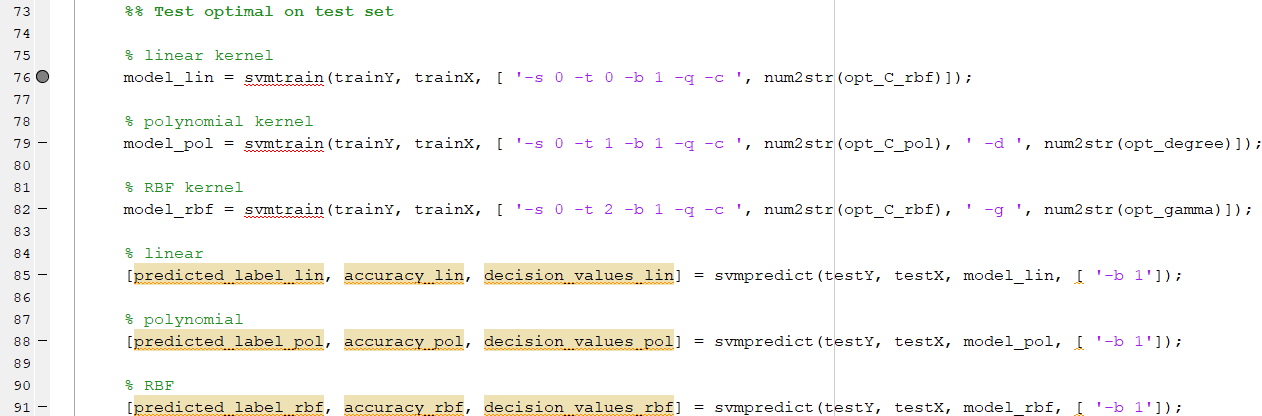
Which results in nice 3d plots:





The best parameters in this grid search are [degree = 1, C=1.5] for polynomial and [gamma=0, C=0.1 ] for RBF kernels

We then train and test three models again to verify our solution:

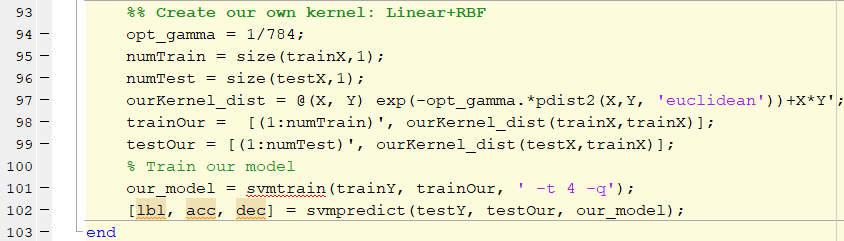


Which result in accuracies



We have found better accuracies for polynomial and rbf kernels, but not too big of improvement on RBF.

Last part of this problem is to create our own kernel and train and test SVM models with libsvm to see whether we can get any improvements on the conventional models. In order to do this, define a new function on two inputs, compute kernel matrices (our new kernel) for each pair in training set and testing set, and feed those kernel matrices as new input data into svmtrain and svmtest



Using default parameters for gamma gets us high accuracy, seems that this kernel is also efficient for our classification task, even better than default polynomial kernel.

