Machine Learning

Homework 5

Report

Name: Sergiy Shelekh
ID: 0760822
Email: 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 $Y_i = f(X_i) + \epsilon_i$ 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

$$C(\mathbf{x}_n, \mathbf{x}_m) = k(\mathbf{x}_n, \mathbf{x}_m) + \beta^{-1}\delta_{nm}$$

And the quadratic kernel is of the form

$$k(\mathbf{x}_n, \mathbf{x}_m) = \theta_0 \exp\left\{-\frac{\theta_1}{2}\|\mathbf{x}_n - \mathbf{x}_m\|^2\right\} + \theta_2 + \theta_3\mathbf{x}_n^\top\mathbf{x}_m,$$

Where $[\theta_0, \theta_1, \theta_2, \theta_3]$ are the *hyperparameters* of this kernel.

Now, if we have our covariance matrix $\mathbf{C}_N$ 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

$$k(\boldsymbol{x}_n, \boldsymbol{x}_{N+1}) \text{ for } n = 1, \dots, \mathrm{N}$$

Then we get $c = k(\boldsymbol{x}_{N+1}, \boldsymbol{x}_{N+1}) + \beta^{-1}$ and can write the new covariance matrix $\mathbf{C}_{N+1}$ as

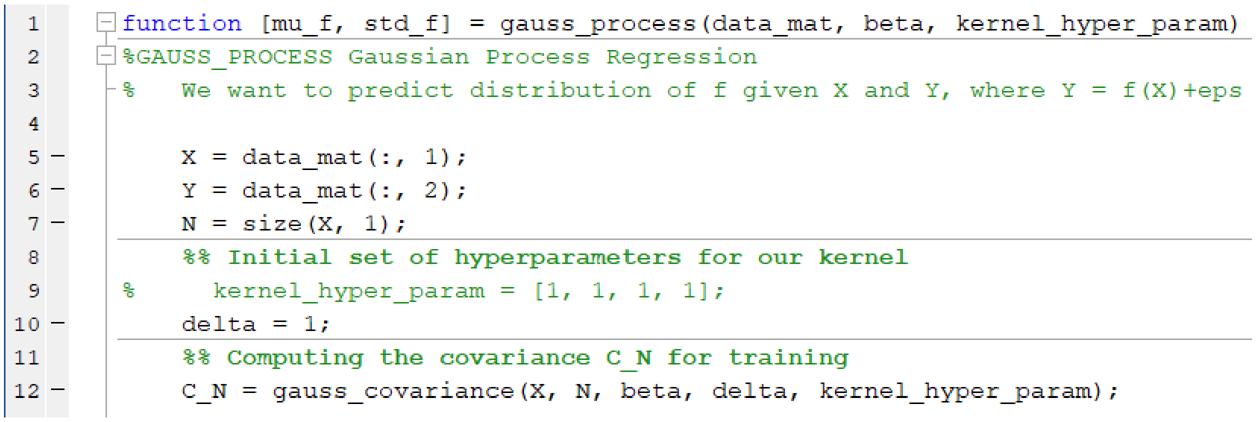
$$\mathbf{C}_{N+1} = \begin{pmatrix} \mathbf{C}_N & \mathbf{k} \\ \mathbf{k}^\top & c \end{pmatrix}$$

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:

$$m(\mathbf{x}_{\mathrm{N+1}}) = \mathbf{k}^\top\mathbf{C}_N^{-1}\mathbf{y}$$

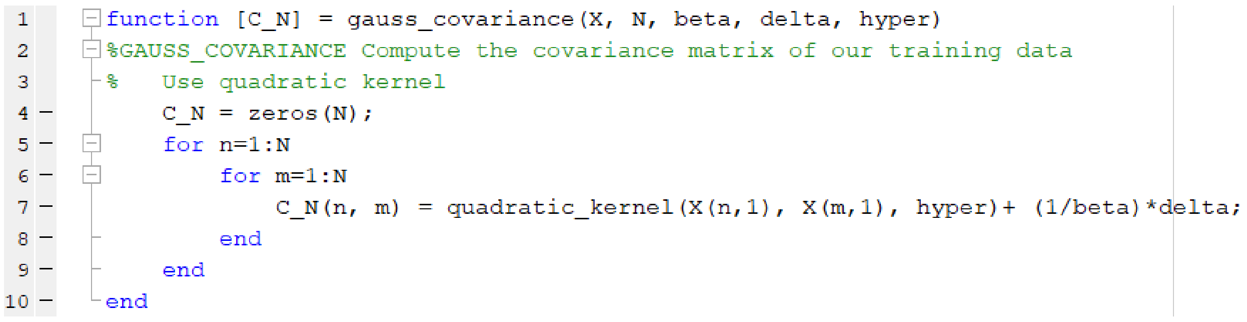
$$\sigma^2(\mathbf{x}_{\mathrm{N}+1}) = c - \mathbf{k}^{\top}\mathbf{C}_N^{-1}\mathbf{k}$$

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]



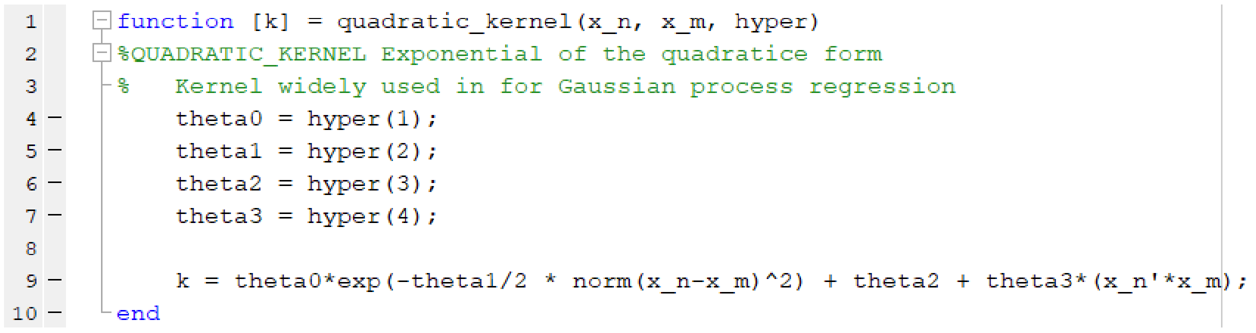
```matlab
function [mu_f, std_f] = gauss_process(data_mat, beta, kernel_hyper_param)
%GAUSS_PROCESS Gaussian Process Regression
%   We want to predict distribution of f given X and Y, where Y = f(X)+eps

    X = data_mat(:, 1);
    Y = data_mat(:, 2);
    N = size(X, 1);
    %% Initial set of hyperparameters for our kernel
%     kernel_hyper_param = [1, 1, 1, 1];
    delta = 1;
    %% Computing the covariance C_N for training
    C_N = gauss_covariance(X, N, beta, delta, kernel_hyper_param);
```

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



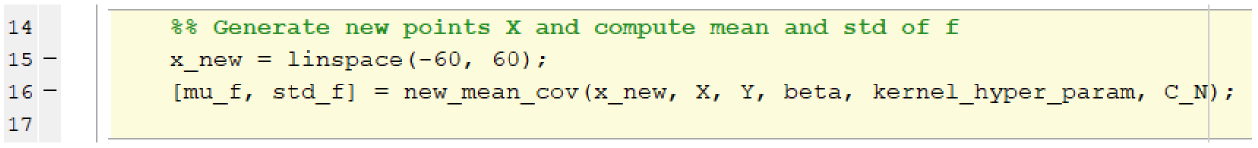
```matlab
function [C_N] = gauss_covariance(X, N, beta, delta, hyper)
%GAUSS_COVARIANCE Compute the covariance matrix of our training data
%   Use quadratic kernel
    C_N = zeros(N);
    for n=1:N
        for m=1:N
            C_N(n, m) = quadratic_kernel(X(n,1), X(m,1), hyper)+ (1/beta)*delta;
        end
    end
end
```

Kernel is also given by the definition above



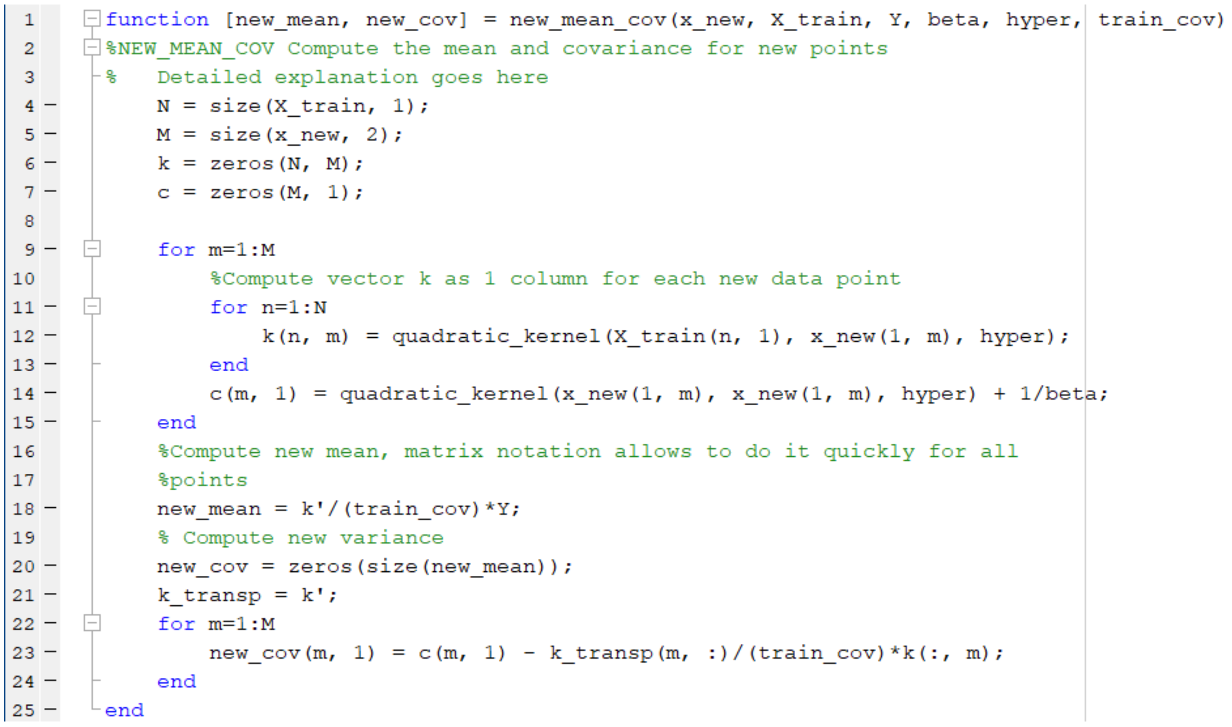
```matlab
function [k] = quadratic_kernel(x_n, x_m, hyper)
%QUADRATIC_KERNEL Exponential of the quadratice form
%   Kernel widely used in for Gaussian process regression
    theta0 = hyper(1);
    theta1 = hyper(2);
    theta2 = hyper(3);
    theta3 = hyper(4);

    k = theta0*exp(-theta1/2 * norm(x_n-x_m)^2) + theta2 + theta3*(x_n'*x_m);
end
```

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



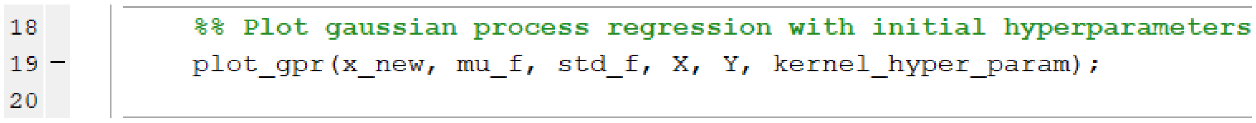
```matlab
    %% Generate new points X and compute mean and std of f
    x_new = linspace(-60, 60);
    [mu_f, std_f] = new_mean_cov(x_new, X, Y, beta, kernel_hyper_param, C_N);

```

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



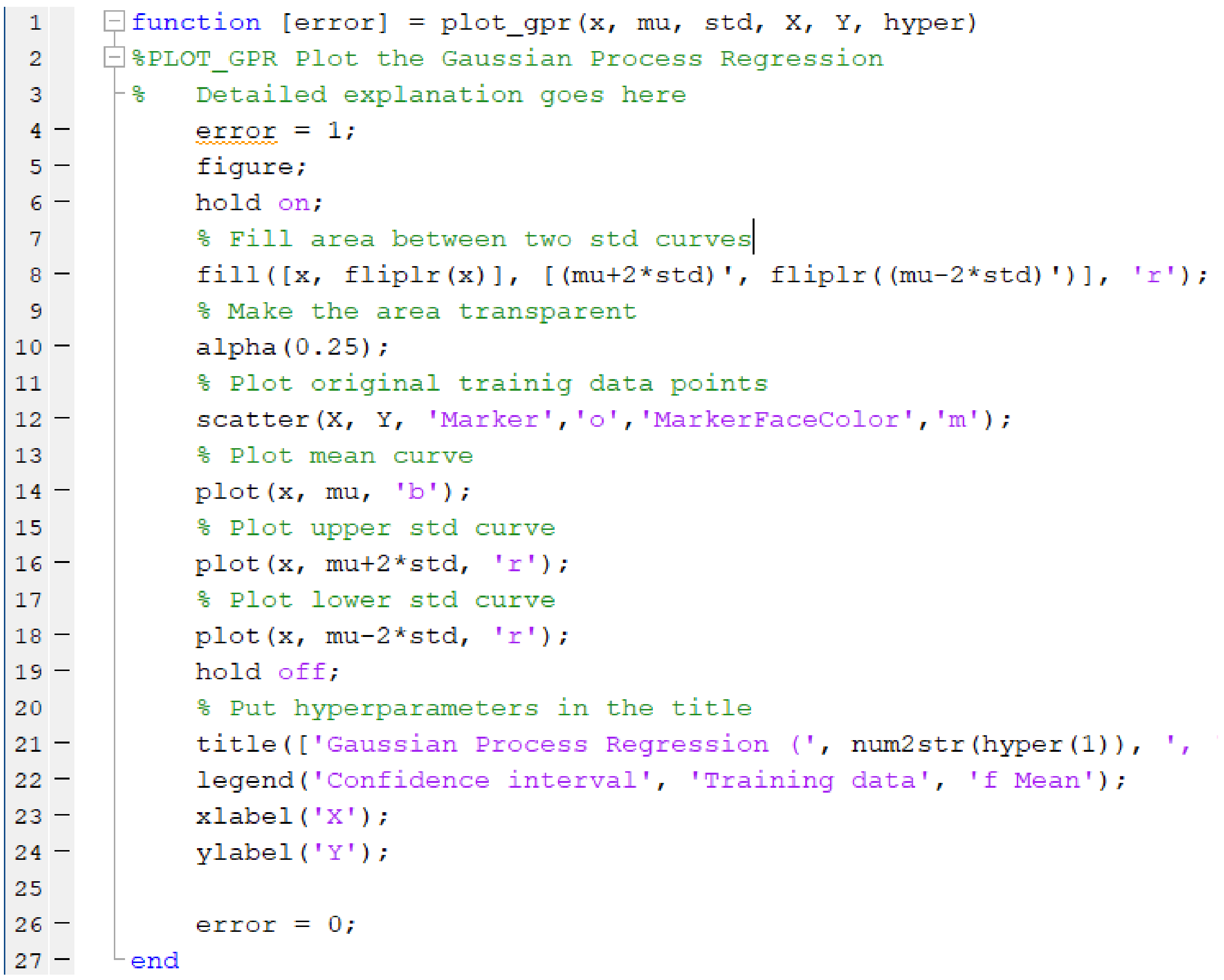
```
function [new_mean, new_cov] = new_mean_cov(x_new, X_train, Y, beta, hyper, train_cov)
%NEW_MEAN_COV Compute the mean and covariance for new points
%   Detailed explanation goes here
    N = size(X_train, 1);
    M = size(x_new, 2);
    k = zeros(N, M);
    c = zeros(M, 1);

    for m=1:M
        %Compute vector k as 1 column for each new data point
        for n=1:N
            k(n, m) = quadratic_kernel(X_train(n, 1), x_new(1, m), hyper);
        end
        c(m, 1) = quadratic_kernel(x_new(1, m), x_new(1, m), hyper) + 1/beta;
    end
    %Compute new mean, matrix notation allows to do it quickly for all
    %points
    new_mean = k'/(train_cov)*Y;
    % Compute new variance
    new_cov = zeros(size(new_mean));
    k_transp = k';
    for m=1:M
        new_cov(m, 1) = c(m, 1) - k_transp(m, :)/(train_cov)*k(:, m);
    end
end
```

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



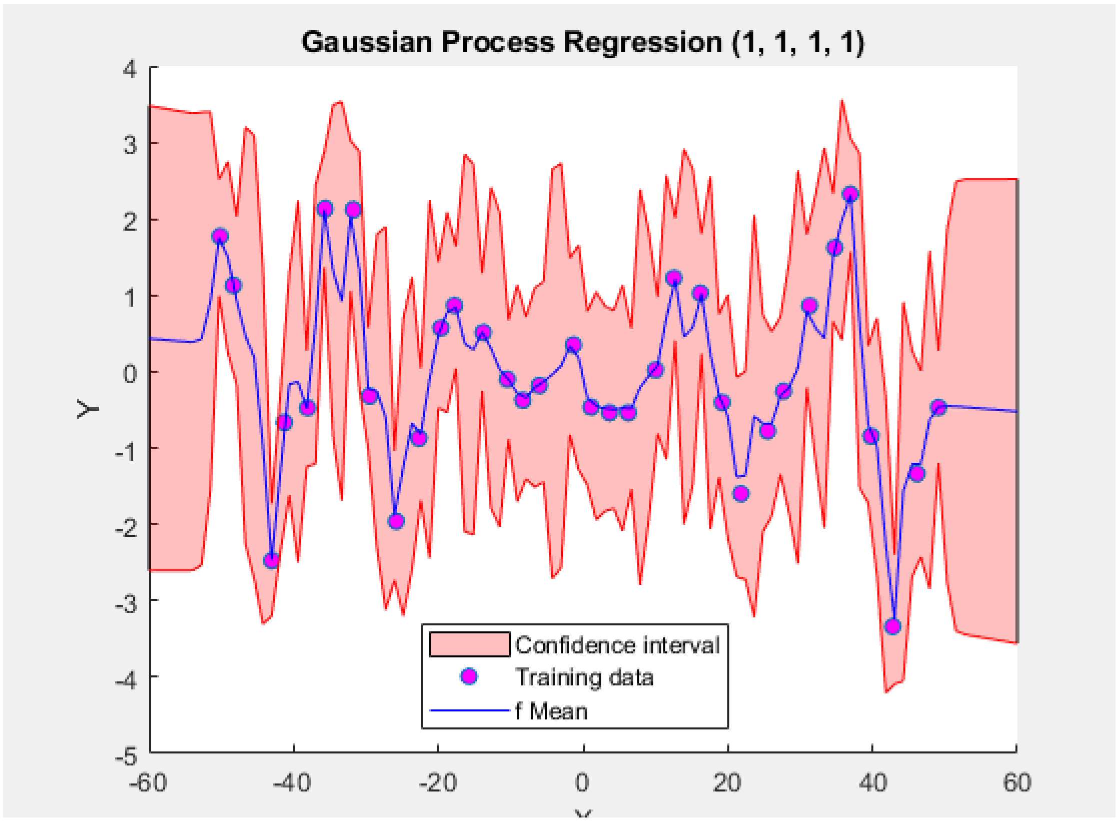
```
    %% Plot gaussian process regression with initial hyperparameters
    plot_gpr(x_new, mu_f, std_f, X, Y, kernel_hyper_param);

```

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



```matlab
function [error] = plot_gpr(x, mu, std, X, Y, hyper)
%PLOT_GPR Plot the Gaussian Process Regression
%   Detailed explanation goes here
    error = 1;
    figure;
    hold on;
    % Fill area between two std curves
    fill([x, fliplr(x)], [(mu+2*std)', fliplr((mu-2*std)')], 'r');
    % Make the area transparent
    alpha(0.25);
    % Plot original trainig data points
    scatter(X, Y, 'Marker','o','MarkerFaceColor','m');
    % Plot mean curve
    plot(x, mu, 'b');
    % Plot upper std curve
    plot(x, mu+2*std, 'r');
    % Plot lower std curve
    plot(x, mu-2*std, 'r');
    hold off;
    % Put hyperparameters in the title
    title(['Gaussian Process Regression (', num2str(hyper(1)), ',
    legend('Confidence interval', 'Training data', 'f Mean');
    xlabel('X');
    ylabel('Y');

    error = 0;
end
```

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

$$\ln \mathrm{p}(\mathbf{y}|\boldsymbol{\theta}) = -\frac{1}{2}\ln|\mathbf{C}_N| - \frac{1}{2}\mathbf{y}^\top \mathbf{C}_N^{-1}\mathbf{y} - \frac{N}{2}\ln(2\pi)$$

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

$$\frac{\partial}{\partial \theta_i}\ln \mathrm{p}(\mathbf{y}|\boldsymbol{\theta}) = -\frac{1}{2}\mathrm{Tr}\left(\mathbf{C}_N^{-1}\frac{\partial \mathbf{C}_N}{\partial \theta_i}\right) - \frac{1}{2}\mathbf{y}^\top \mathbf{C}_N^{-1}\frac{\partial \mathbf{C}_N}{\partial \theta_i}\mathbf{C}_N^{-1}\mathbf{y}$$

We have everything, except for $\frac{\partial \mathbf{C}_N}{\partial \theta_i}$, which can be found by taking derivative of our kernel w.r.t each θ_i

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

$$\frac{\partial k(\mathbf{x}_n, \mathbf{x}_m)}{\partial \theta_0} = \exp\left\{-\frac{\theta_1}{2}\|\mathbf{x}_n - \mathbf{x}_m\|^2\right\}$$

$$\frac{\partial k(\mathbf{x}_n, \mathbf{x}_m)}{\partial \theta_1} = \theta_0 \exp\left\{-\frac{\theta_1}{2}\|\mathbf{x}_n - \mathbf{x}_m\|^2\right\}\left(-\frac{\|\mathbf{x}_n - \mathbf{x}_m\|^2}{2}\right)$$

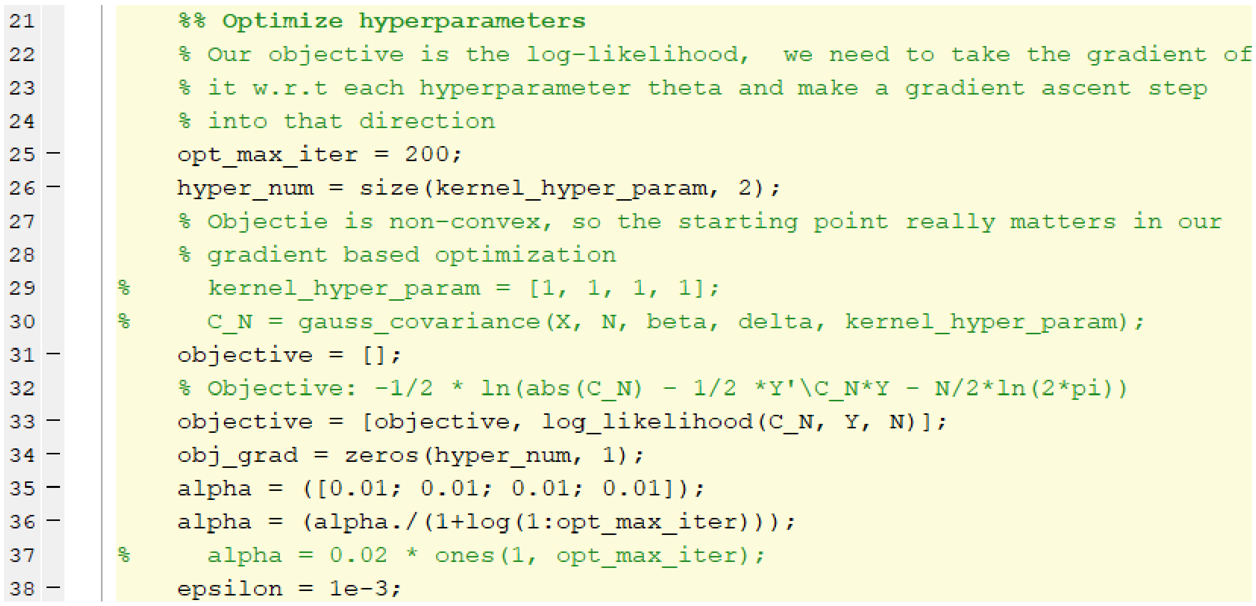
$$\frac{\partial k(\mathbf{x}_n, \mathbf{x}_m)}{\partial \theta_2} = 1$$

$$\frac{\partial k(\mathbf{x}_n, \mathbf{x}_m)}{\partial \theta_3} = \mathbf{x}_n^{\mathsf{T}}\mathbf{x}_m$$

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

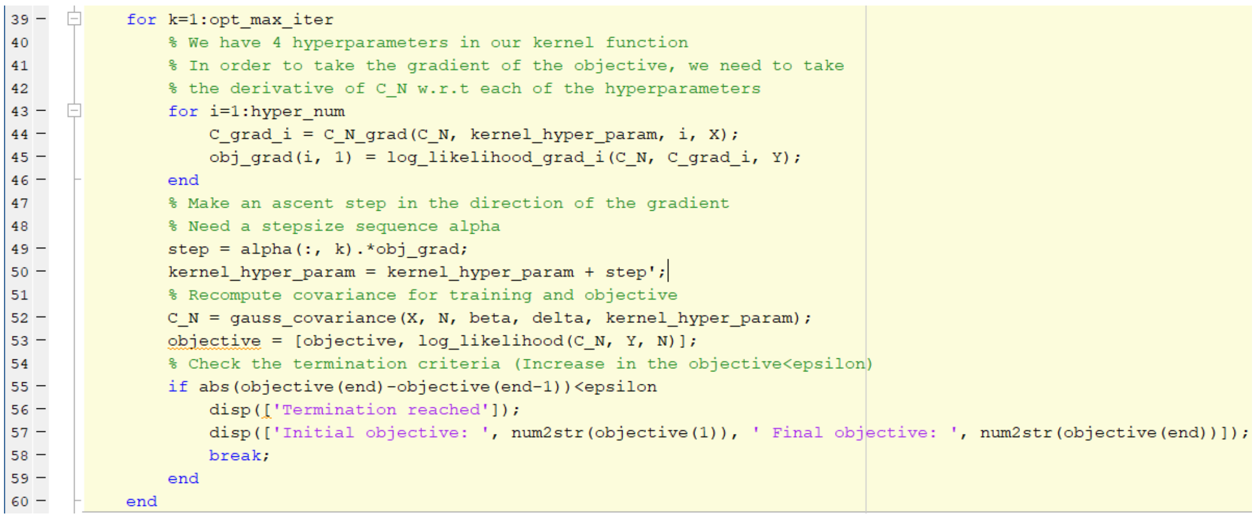
$$hyper_{new} = hyper_{old} + \alpha * hyper_{grad}$$

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.



```
    %% Optimize hyperparameters
    % Our objective is the log-likelihood,  we need to take the gradient of
    % it w.r.t each hyperparameter theta and make a gradient ascent step
    % into that direction
    opt_max_iter = 200;
    hyper_num = size(kernel_hyper_param, 2);
    % Objectie is non-convex, so the starting point really matters in our
    % gradient based optimization
%     kernel_hyper_param = [1, 1, 1, 1];
%     C_N = gauss_covariance(X, N, beta, delta, kernel_hyper_param);
    objective = [];
    % Objective: -1/2 * ln(abs(C_N) - 1/2 *Y'\C_N*Y - N/2*ln(2*pi))
    objective = [objective, log_likelihood(C_N, Y, N)];
    obj_grad = zeros(hyper_num, 1);
    alpha = ([0.01; 0.01; 0.01; 0.01]);
    alpha = (alpha./(1+log(1:opt_max_iter)));
%     alpha = 0.02 * ones(1, opt_max_iter);
    epsilon = 1e-3;
```

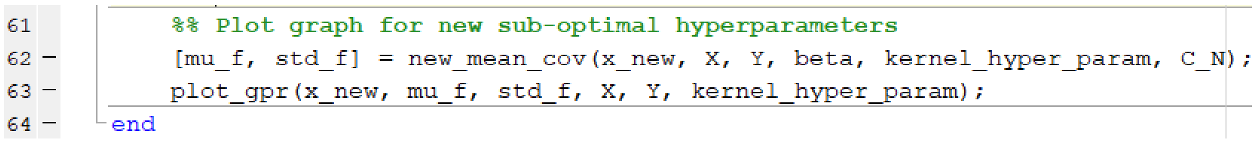
In the initialization part define a sequence of positive step sizes alpha, set initial step size (heuristic choice is 0.01, diverges for larger), compute initial objective, set termination criteria parameter to 0.001. Next, we start the optimization loop



```
    for k=1:opt_max_iter
        % We have 4 hyperparameters in our kernel function
        % In order to take the gradient of the objective, we need to take
        % the derivative of C_N w.r.t each of the hyperparameters
        for i=1:hyper_num
            C_grad_i = C_N_grad(C_N, kernel_hyper_param, i, X);
            obj_grad(i, 1) = log_likelihood_grad_i(C_N, C_grad_i, Y);
        end
        % Make an ascent step in the direction of the gradient
        % Need a stepsize sequence alpha
        step = alpha(:, k).*obj_grad;
        kernel_hyper_param = kernel_hyper_param + step';
        % Recompute covariance for training and objective
        C_N = gauss_covariance(X, N, beta, delta, kernel_hyper_param);
        objective = [objective, log_likelihood(C_N, Y, N)];
        % Check the termination criteria (Increase in the objective<epsilon)
        if abs(objective(end)-objective(end-1))<epsilon
            disp(['Termination reached']);
            disp(['Initial objective: ', num2str(objective(1)), ' Final objective: ', num2str(objective(end))]);
            break;
        end
    end
```

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.

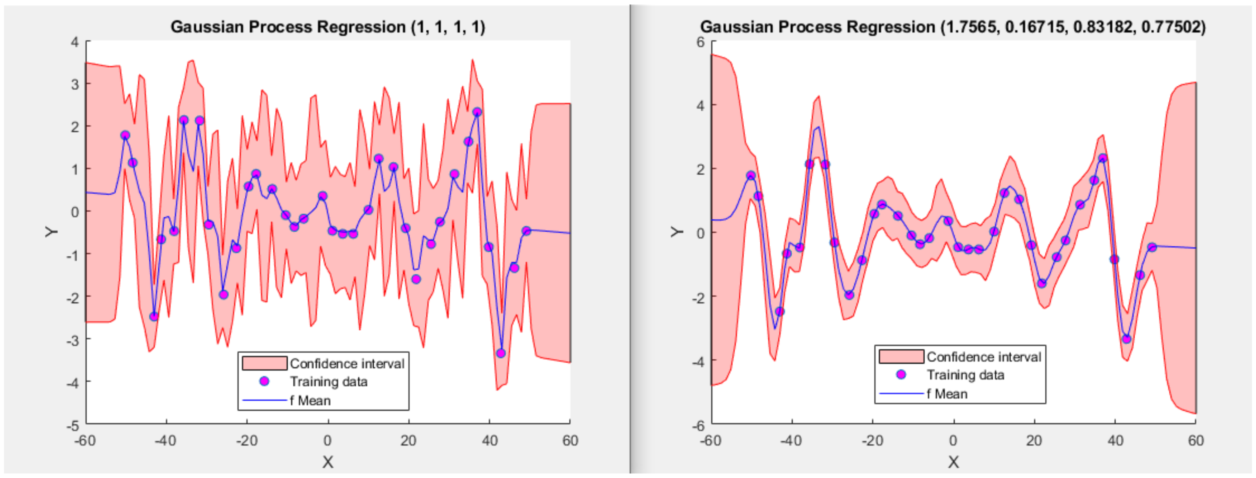
Finally, estimate new mean and covariance using sub-optimal hyperparameters and plot



```
    %% Plot graph for new sub-optimal hyperparameters
    [mu_f, std_f] = new_mean_cov(x_new, X, Y, beta, kernel_hyper_param, C_N);
    plot_gpr(x_new, mu_f, std_f, X, Y, kernel_hyper_param);
end
```

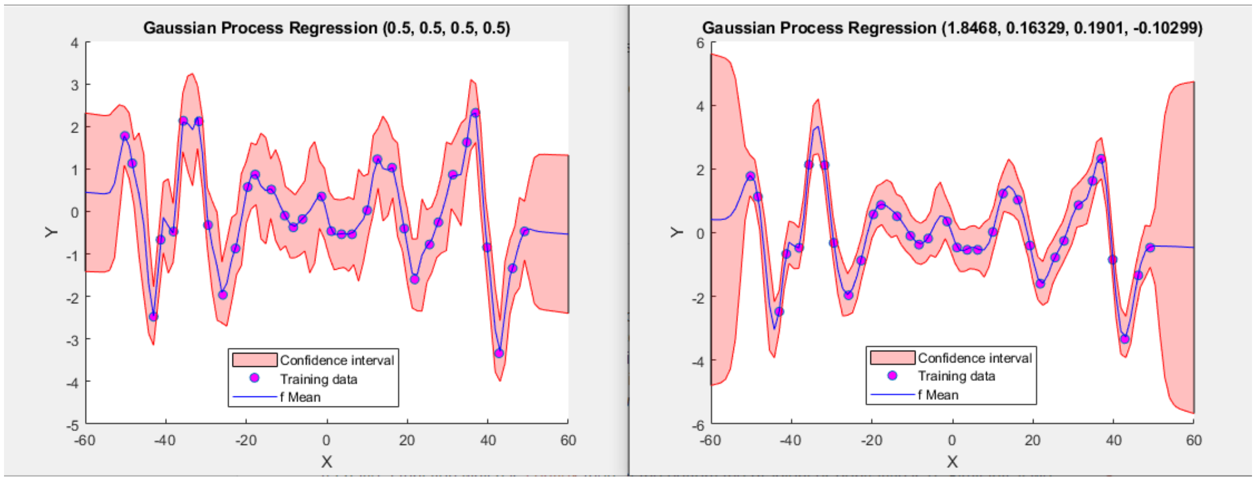
Compare results for different initial choice:

Average initial choice



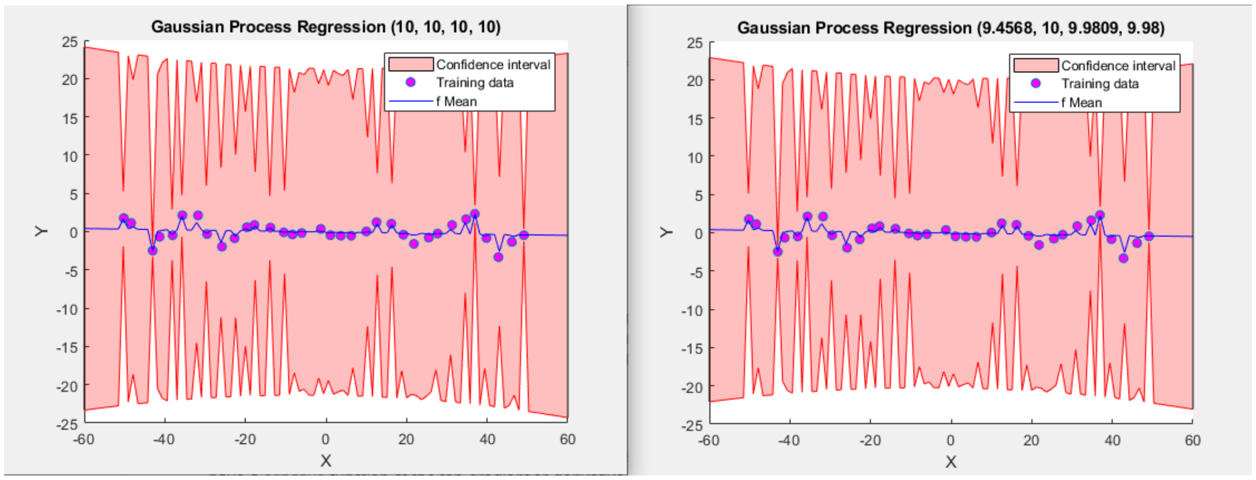
Smoothed curves, increased std at the ends, visible difference from initial

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.

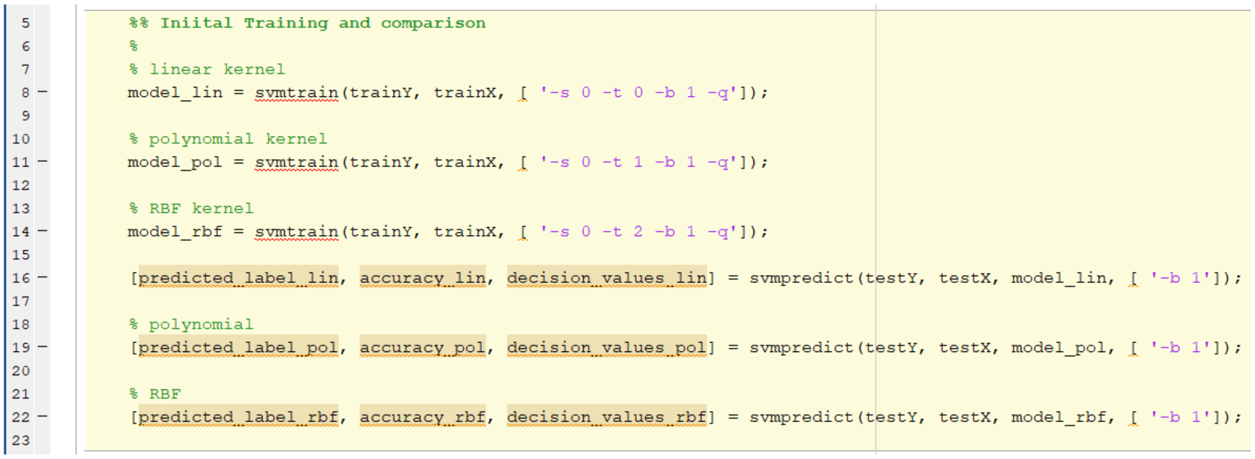
Problem 2: SVM on MNIST data

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:

$$\text{Linear: } \mathrm{K}(\mathbf{u},\ \mathbf{v}) = \mathbf{u}^T\mathbf{v}$$
$$\text{Polynomial: } \mathrm{K}(\mathbf{u},\ \mathbf{v},\ \gamma, c, d) = (\gamma\mathbf{u}^T\mathbf{v}+c)^d$$
$$\text{RBF: } \mathrm{K}(\mathbf{u},\ \mathbf{v},\ \gamma) = \exp(-\gamma\|\mathbf{u}-\mathbf{v}\|^2)$$

Default values in libsvm are $\gamma = \frac{1}{num\_features}$, c=0, $d = 3$

Using these defaults, we train and test three models:



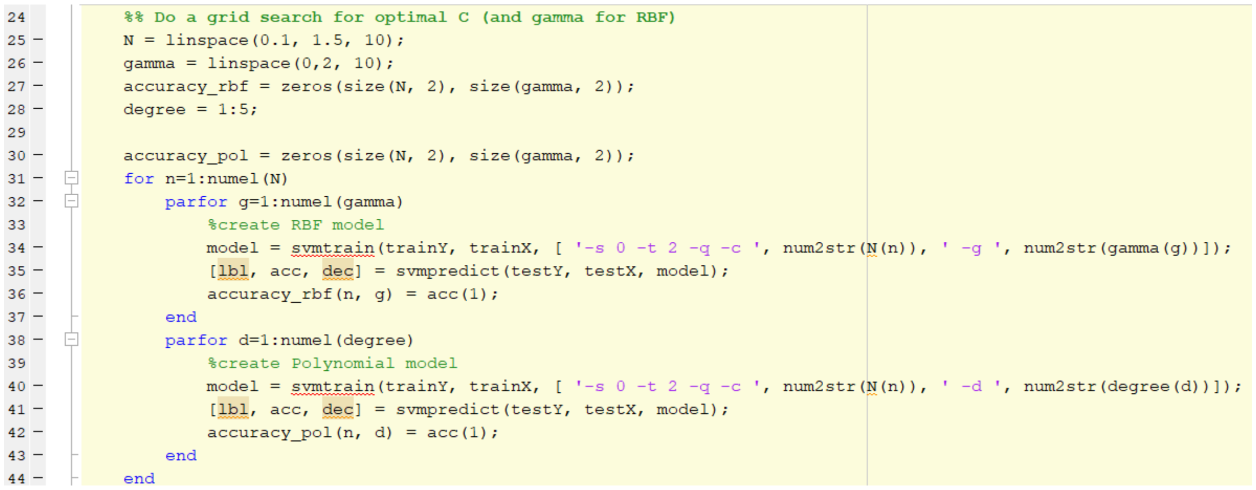
```matlab
%% Iniital Training and comparison
%
% linear kernel
model_lin = svmtrain(trainY, trainX, [ '-s 0 -t 0 -b 1 -q']);

% polynomial kernel
model_pol = svmtrain(trainY, trainX, [ '-s 0 -t 1 -b 1 -q']);

% RBF kernel
model_rbf = svmtrain(trainY, trainX, [ '-s 0 -t 2 -b 1 -q']);

[predicted_label_lin, accuracy_lin, decision_values_lin] = svmpredict(testY, testX, model_lin, [ '-b 1']);

% polynomial
[predicted_label_pol, accuracy_pol, decision_values_pol] = svmpredict(testY, testX, model_pol, [ '-b 1']);

% RBF
[predicted_label_rbf, accuracy_rbf, decision_values_rbf] = svmpredict(testY, testX, model_rbf, [ '-b 1']);

```

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

```
Accuracy = 95.52% (2388/2500) (classification)
Accuracy = 72% (1800/2500) (classification)
Accuracy = 95.52% (2388/2500) (classification)
```

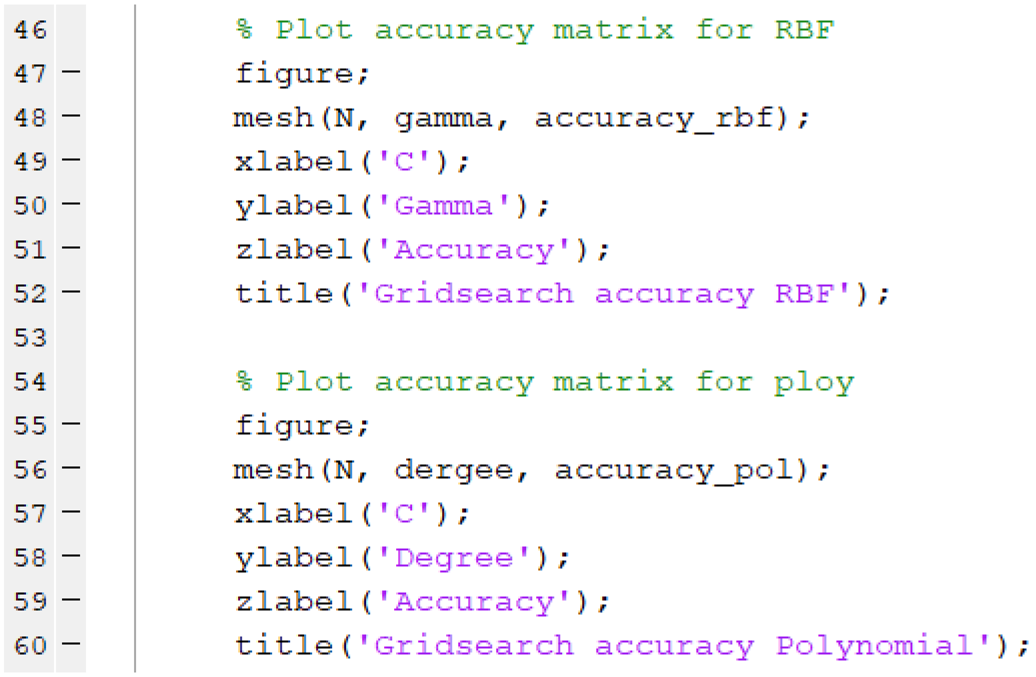
We see, that linear and RBF models get same accuracy on the test data 95.52%, while polynomial kernel model performs worse, at 72%

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



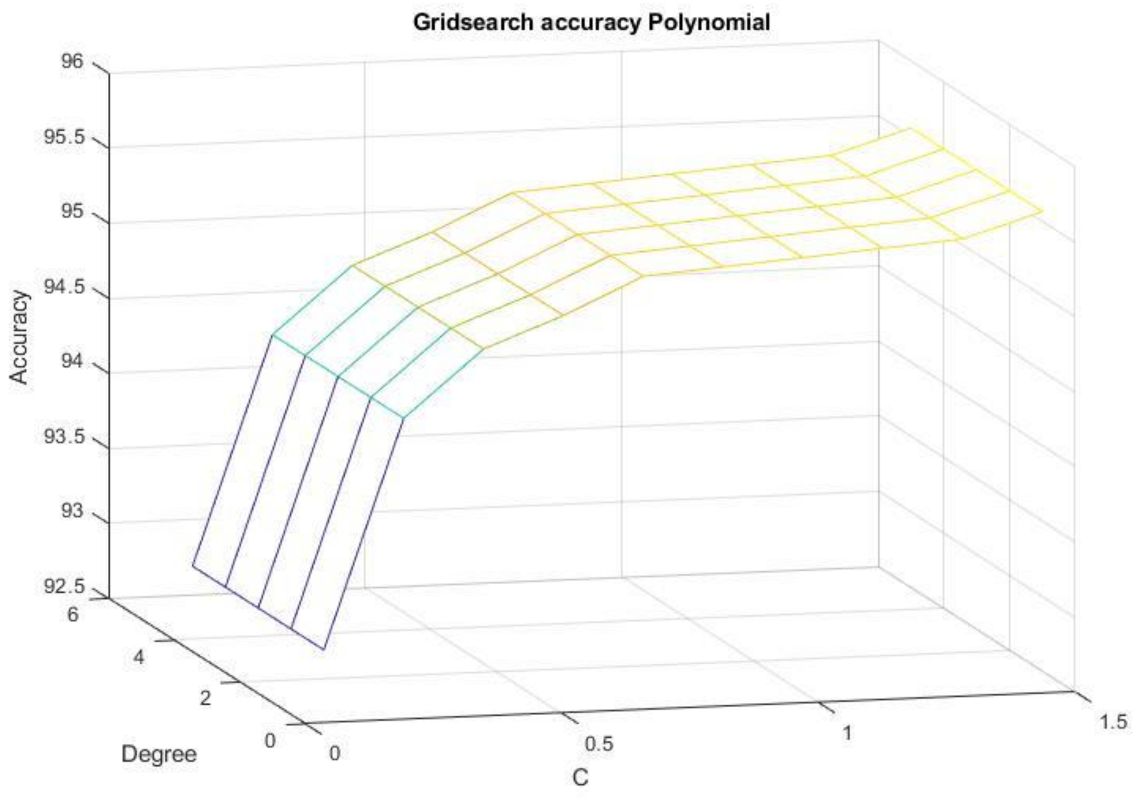
```matlab
%% Do a grid search for optimal C (and gamma for RBF)
N = linspace(0.1, 1.5, 10);
gamma = linspace(0,2, 10);
accuracy_rbf = zeros(size(N, 2), size(gamma, 2));
degree = 1:5;

accuracy_pol = zeros(size(N, 2), size(gamma, 2));
for n=1:numel(N)
    parfor g=1:numel(gamma)
        %create RBF model
        model = svmtrain(trainY, trainX, [ '-s 0 -t 2 -q -c ', num2str(N(n)), ' -g ', num2str(gamma(g))]);
        [lbl, acc, dec] = svmpredict(testY, testX, model);
        accuracy_rbf(n, g) = acc(1);
    end
    parfor d=1:numel(degree)
        %create Polynomial model
        model = svmtrain(trainY, trainX, [ '-s 0 -t 2 -q -c ', num2str(N(n)), ' -d ', num2str(degree(d))]);
        [lbl, acc, dec] = svmpredict(testY, testX, model);
        accuracy_pol(n, d) = acc(1);
    end
end
```

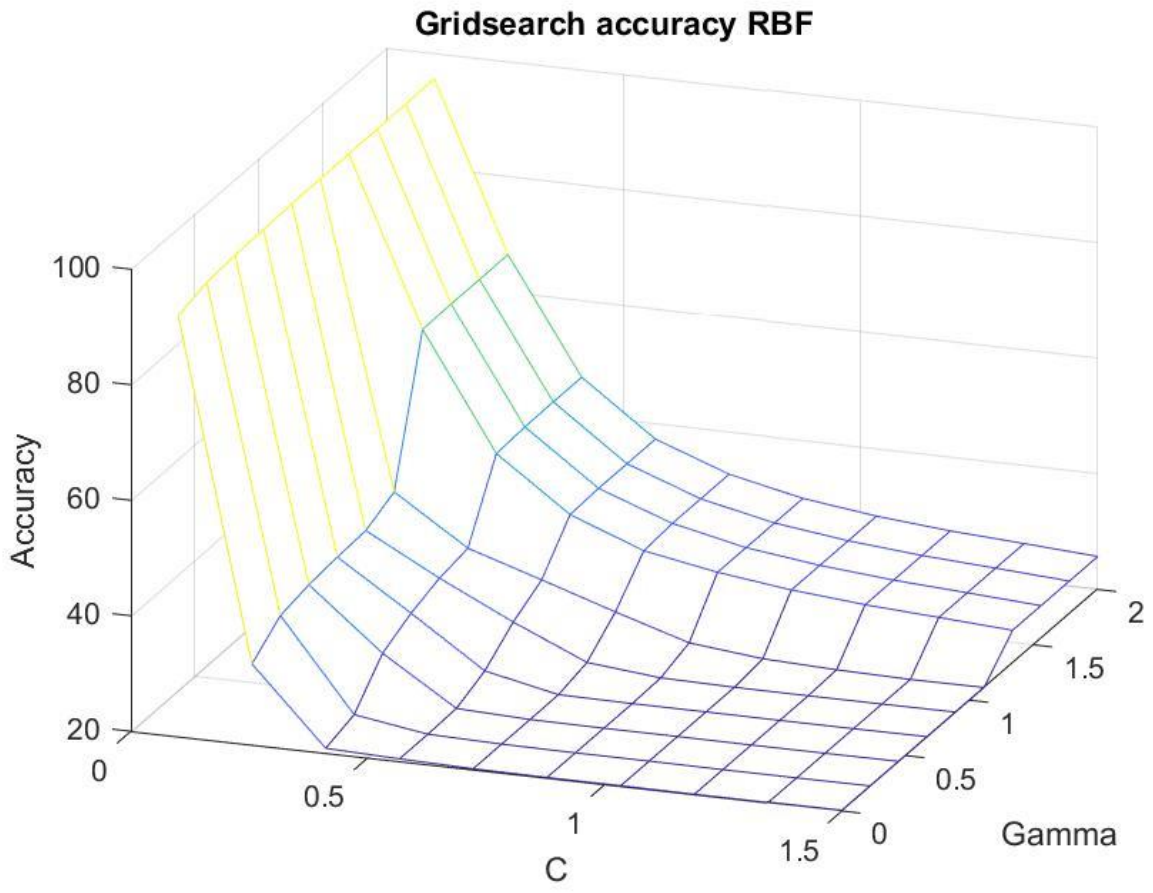
After the grid search, we plot accuracy obtained at each point



```
    % Plot accuracy matrix for RBF
    figure;
    mesh(N, gamma, accuracy_rbf);
    xlabel('C');
    ylabel('Gamma');
    zlabel('Accuracy');
    title('Gridsearch accuracy RBF');

    % Plot accuracy matrix for ploy
    figure;
    mesh(N, dergee, accuracy_pol);
    xlabel('C');
    ylabel('Degree');
    zlabel('Accuracy');
    title('Gridsearch accuracy Polynomial');
```

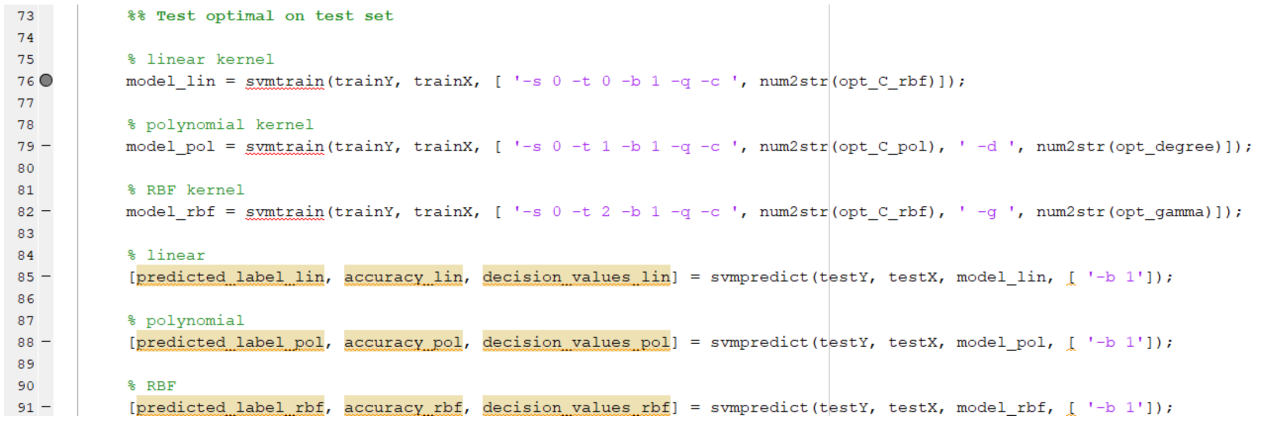
Which results in nice 3d plots:





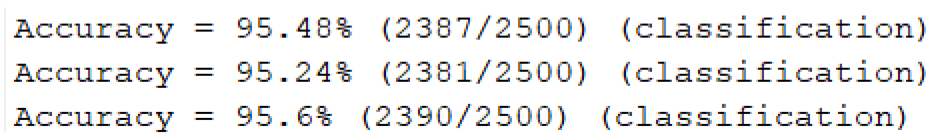
The best parameters in this grid search are [degree = 1, C=1.5] for polynomial and [gamma=0, C=0.1] for RBF kernels

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



```
%% Test optimal on test set

% linear kernel
model_lin = svmtrain(trainY, trainX, [ '-s 0 -t 0 -b 1 -q -c ', num2str(opt_C_rbf)]);

% polynomial kernel
model_pol = svmtrain(trainY, trainX, [ '-s 0 -t 1 -b 1 -q -c ', num2str(opt_C_pol), ' -d ', num2str(opt_degree)]);

% RBF kernel
model_rbf = svmtrain(trainY, trainX, [ '-s 0 -t 2 -b 1 -q -c ', num2str(opt_C_rbf), ' -g ', num2str(opt_gamma)]);

% linear
[predicted_label_lin, accuracy_lin, decision_values_lin] = svmpredict(testY, testX, model_lin, [ '-b 1']);

% polynomial
[predicted_label_pol, accuracy_pol, decision_values_pol] = svmpredict(testY, testX, model_pol, [ '-b 1']);

% RBF
[predicted_label_rbf, accuracy_rbf, decision_values_rbf] = svmpredict(testY, testX, model_rbf, [ '-b 1']);
```

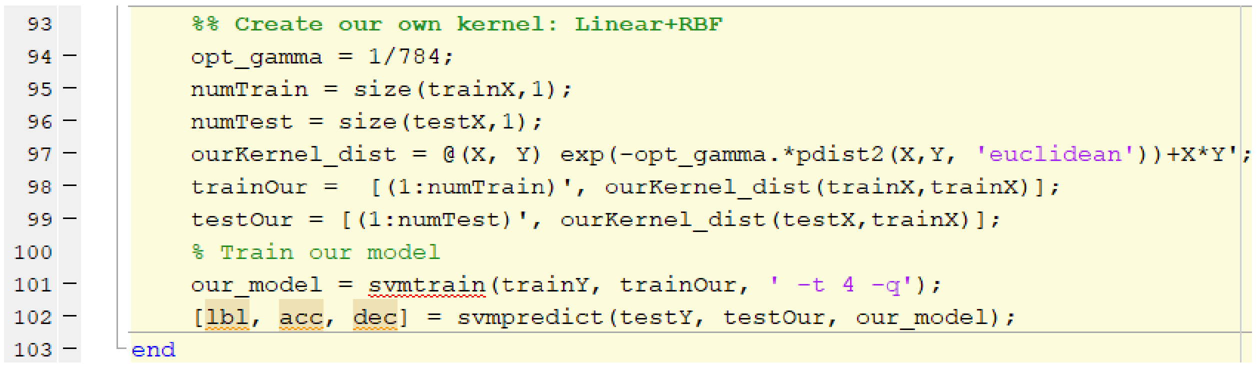
Which result in accuracies



```
Accuracy = 95.48% (2387/2500) (classification)
Accuracy = 95.24% (2381/2500) (classification)
Accuracy = 95.6% (2390/2500) (classification)
```

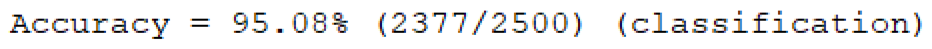
We have found better accuracies for polynomial and rbf kernels, but not too big of improvement on RBF.

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



```
    %% Create our own kernel: Linear+RBF
    opt_gamma = 1/784;
    numTrain = size(trainX,1);
    numTest = size(testX,1);
    ourKernel_dist = @(X, Y) exp(-opt_gamma.*pdist2(X,Y, 'euclidean'))+X*Y';
    trainOur =  [(1:numTrain)', ourKernel_dist(trainX,trainX)];
    testOur = [(1:numTest)', ourKernel_dist(testX,trainX)];
    % Train our model
    our_model = svmtrain(trainY, trainOur, ' -t 4 -q');
    [lbl, acc, dec] = svmpredict(testY, testOur, our_model);
end
```

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.



```
Accuracy = 95.08% (2377/2500) (classification)
```