Machine Learning

Summer Semester 2019, Homework 4

Prof. Dr. J. Peters, H. Abdulsamad, S. Stark, D.Koert

TECHNISCHE UNIVERSITÄT DARMSTADT

Total points: 44 + 10 bonus
Due date: Friday, 19 Juli 2019 (17:00)
You need to hand in the pdf in moodle **and** a printed version to the postbox from the IAS secretary office (S2 02 | E315)

Name, Surname, ID Number

Problem 4.1 Support Vector Machines [14 Points + 5 Bonus]

In this exercise, you will use the dataset `iris-pca.txt`. It is the same dataset used for Homework 3, but the data has been pre-processed with PCA and only two kinds of flower ('Setosa' and 'Virginica') have been kept, along with their two principal components. Each row contains a sample while the last attribute is the label (0 means that the sample comes from a 'Setosa' plant, 2 from 'Virginica').
You are allowed to `numpy` functions (e.g., `numpy.linalg.eig`). For quadratic programming we suggest `cvxopt`.

a) **Definition [3 Points]**

Briefly describe SVMs. What is their advantage w.r.t. other linear approaches we discussed this semester?

b) **Quadratic Programming [2 Points]**

Formalize SVMs as a constrained optimization problem.

c) **Slack Variables [2 Points]**

Explain the concept behind slack variables and reformulate the optimization problem accordingly.

d) **The Dual Problem [4 Points]**

What are the advantages of solving the dual instead of the primal?

e) **Kernel Trick [3 Points]**

Explain the kernel trick and why it is particularly convenient in SVMs.

f) **Implementation [5 Bonus Points]**

Implement and learn an SVM to classify the data in `iris-pca.txt`. Choose your kernel. Create a plot showing the data, the support vectors and the decision boundary. Show also the misclassified samples. Attach a snippet of your code.

Problem 4.2 Neural Networks [20 Points + 5 Bonus]

In this exercise, you will use the dataset `mnist_small`, divided into four files. The *mnist* dataset is widely used as benchmark for classification algorithms. It contains 28x28 images of handwritten digits (pairs <`input, output`> correspond to <`pixels, digit`>).

a) **Multi-layer Perceptron [20 Points]**

Implement a neural network and train it using backpropagation on the provided dataset. Choose your loss and activation functions and your hyperparameters (number of layers, neurons, learning rate, ...), briefly explaining your choices. You **cannot** use any library beside `numpy`! That is, you have to implement by yourself the loss and activation functions, the backpropagation algorithm and the gradient descent optimizer (if you want to use any).

Show how the misclassification error (in percentage) on the testing set evolves during the learning. An acceptable solution achieves an error of 8% or less. Attach snippets of your code.

Hint: if your network does not learn, check how the network parameters change and plot the trend of your loss function.

For the activation function I chose the sigmoid activation function. I choosed this activation function, since its derivation can be formulated using the actual function. This makes implementing gradient descent very easy. However since I choose this approach, it is non-trivial to change the activation function, since its derivative is hardcoded in the gradient descent algorithm.

For the loss function I simply try to minimize between the output of the neural network and the label. For this purpose a one-hot-encoding is used.

For the number of layers I went with a single layer. One layer should be sufficient enough to achieve the given metric. It is also easy to train and easy to implement.

The number of neurons, learning rate and epoch count are chosen after my feeling of what could potentially work.

If we now have a look at the graph showing the error on the validation set, we can see that it decreases with each epoch and reaches a minimum at 5.38 percent 1.

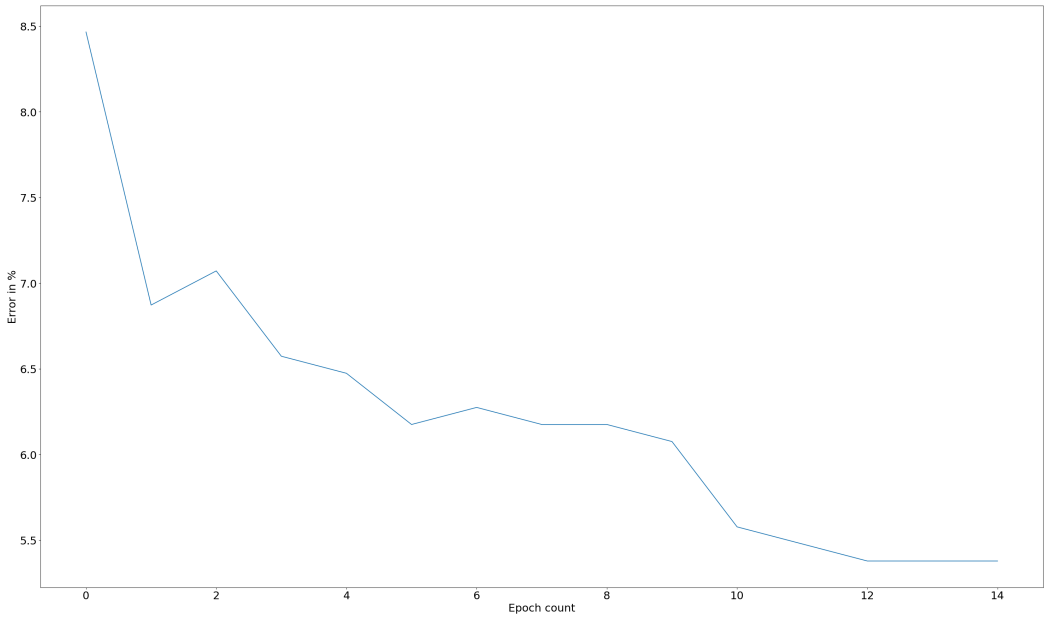

Figure 1: Classification error of the Neural Network for each epoch.

Since I have already written a Neural Network last year, the code is mostly reused from that time, with removed scipy dependencies. I learned about Neuronal Networks from Tariq Rashid, so parts of my codes will borrow from his implementation. I also had some problems loading the train/test-in data with numpy, so I put them in excel and saved them again as csv, which made them work. Since this mnist dataset is sorted we need to shuffle them. I found a nice numpy function on stackoverflow which does exactly that.

```
import numpy as np
import matplotlib.pyplot as plt

"""Neuronal Network class definition"""
class NeuralNetwork:

    # initialise the neural network
    def __init__(self, inputs, hidden, outputs, lr):
        # set number of neurons in each input, hidden and output layer and learning rate
        self.inputs = inputs
        self.hidden = hidden
        self.outputs = outputs
        self.lr = lr

        # activation function is sigmoid function
        self.activation = lambda x: 1/(1+np.exp(-x))

        # Define weight matrices from input to hidden layer wih and hidden to output layer
        self.wih = np.random.normal(0.0, pow(self.hidden, -0.5), (self.hidden, self.inputs
        self.who = np.random.normal(0.0, pow(self.outputs, -0.5), (self.outputs, self.hidd

    # train the neural network
    def train(self, inputs_list, targets_list):
        inputs = np.array(inputs_list, ndmin=2).T
        targets = np.array(targets_list, ndmin=2).T

        # forward calculations for hidden and output layer
        hidden_in = np.dot(self.wih, inputs)
        hidden_out = self.activation(hidden_in)

        final_in = np.dot(self.who, hidden_out)
        final_out = self.activation(final_in)  # final function = sigmoid = classification

        # backpropagation
        out_err = targets - final_out
        # error in hidden layer is computed by multiplying with weights
        hidden_err = np.dot(self.who.T, out_err)

        # update the weights. This is the gradient descent part for the sigmoid activation
        self.who += self.lr * np.dot((out_err * final_out * (1.0 - final_out)), hidden_out
        self.wih += self.lr * np.dot((hidden_err * hidden_out * (1.0 - hidden_out)), input

    # Forward the signal neural network
    def predict(self, input_list):
        # convert list into 2d numpy array
        inputs = np.array(input_list, ndmin=2).T

        # hidden layer calculations
        hidden_in = np.dot(self.wih, inputs)
        hidden_out = self.activation(hidden_in)
```

```
        # final output layer calculations
        final_in = np.dot(self.who, hidden_out)
        final_out = self.activation(final_in)  # final function = sigmoid = classification

        return final_out


# https://stackoverflow.com/questions/4601373/better-way-to-shuffle-two-numpy-arrays-in-uniso
def unison_shuffled_copies(a, b):
    assert len(a) == len(b)
    p = np.random.permutation(len(a))
    return a[p], b[p]


"""Load and shuffle the data"""
# load data 2
train_in = np.loadtxt("data/mnist_small_train_in2.csv", skiprows=1, delimiter=";")
train_out = np.loadtxt("data/mnist_small_train_out.csv")
# shuffle data
train_in, train_out = unison_shuffled_copies(train_in, train_out)

test_in = np.loadtxt("data/mnist_small_test_in2.csv", skiprows=1, delimiter=";")
test_out = np.loadtxt("data/mnist_small_test_out.csv")

test_in, test_out = unison_shuffled_copies(test_in, test_out)



"""Set the Hyperparameter for the Network """
input_nodes = 784  # 28x28 = 784
hidden_nodes = 200  # My Machine Learning sense tells me this is a good number
output_nodes = 10  # We have 10 labels

lr = 0.15
epochs = 15


# create instance of NN
model = NeuralNetwork(input_nodes, hidden_nodes, output_nodes, lr)


error_epoch = []
for e in range(epochs):
    """Train the Neural Network """
    for i in range(len(train_in)):
        #inputs = (train_in[i,:] / 255.0 * 0.99) + 0.01  # We would need this for the actualy
        inputs = train_in[i,:]

        targets = np.zeros(output_nodes) + 0.01  # Add 0.01 for numerical stability
        targets[int(train_out[i])] = 0.99  # One hot encoding. Take the current value from tr

        model.train(inputs, targets)  # Magic time

    """Validate the results"""
    score = []
    for i in range(len(test_in)):
```

```
            correct_label = test_out[i]

            #inputs = (train_in[i, :] / 255.0 * 0.99) + 0.01  # Again no need for in this mnis
            inputs = test_in[i,:]

            outputs = model.predict(inputs)

            label = np.argmax(outputs)

            # Keep track of the score
            if (label == correct_label):
                score.append(1)
            else:
                score.append(0)

        accuracy = (np.asarray(score)).sum() / (np.asarray(score)).size
        error = (1.0 - accuracy) * 100
        print("ERROR: ", round(error, 2), "%")

        error_epoch.append(error)

    """Plot the results"""
    plt.rcParams.update({'font.size': 36})
    plt.plot(error_epoch)
    plt.xlabel("Epoch count")
    plt.ylabel("Error in %")
    plt.show()
```

b) **Deep Learning [5 Bonus Points]**

In recent years, deep neural networks have become one of the most used tools in machine learning. Highlight the qualitative differences between classical neural networks and deep networks. Which limitations of classical NN does deep learning overcome? Give an intuition of the innovations introduced in deep learning compared to traditional NN. (Hint: Have a look at this paper. Use Google Scholar to read other scientific papers for more insights.)

The main qualitative difference between classical neural networks and deep networks is the number of layers used. Classical neural networks mostly used only a single layer. This was due to the fact that training deeper structures was hard because of vanishing gradients and considered uncesseray since the universal approximation theorem states that a single hidden layer is enough. This in combination with less training data and avaiable gpu power led to only focus on training single layer neuronal networks.

One of the main pitfalls of classical neuronal networks was not thinking about the futher consequences of the universal approximation theorem. As it turns out, even if a single layer neural network can learn every function, the theorem doesn't state the complexity needed thus resulting in massive overfitting.

When talking about learning representations in machine learning one key aspect is the hierachical organization of explanatory features. This means that concepts can can be defined in terms of other concepts which are all placed in a hierarchy. The more abstract, non-linear concepcts are at the top of the hierarchy. This allows Deep neural network to build more abstract features in the high levels on top of other features. In a sense a deep neural network is nothing more than a giant feature building machine.

Another strong point for the neural network is that it allows for the re-use of features. This property exists thanks to an exponential grow in the number of paths in the circuit that arise in deeper models.

Problem 4.3 Gaussian Processes [10 Points]

a) **GP Regression [10 Points]**

Implement a Gaussian Process to fit the target function $y = \sin(x)+\sin^2(x)$ with $x \in [0, 0.005, 0.01, 0.015, \ldots, 2\pi]$. Use a squared exponential kernel, an initial mean of 0 and assume a noise variance of 0.001. Begin with no target data points and, at each iteration, sample a new point from the target function according to the uncertainty of your GP (that is, sample the point where the uncertainty is the highest) and update it. Plot your GP (mean and two times standard deviation) after iterations 1, 2, 4, 8 and 16. In each figure, plot also the true function as ground truth and add a new marker for each new sampled point. Attach a snippet of your code.

The following pictures demonstrate our results with implementing the Gaussian Process and trying to fit the given target function. We shall notice that Gaussian Processes are a non-parametric model, meaning that they have an infinite number of parameters or that the data is our parameter. This means, that a Gaussian Process will always have a maximum uncercainity at the point where we have the least amount of data. Since the numpy linspace function spaces data points evenly out, we just need to increase the amount of points that we want to generate points which are at the maximum uncertainity of the Gaussian Process, no futher calculations required.

I want also point out that its not a good idea to naively invert the Kernel-Matrix in order to calculate the mean. Instead we need to solve two different sets of linear equation systems. In this deparment our code borrows from Nando de Freitas (Professor at UBC) who has a fantastic lecture on Youtube, talking about the numerical problems in Gaussian Processes.

Lets take a look at this beautfiul graph 2 showing all 16 iterations of the Gaussian Process. We can see that our how the standard derivation continously shrinkes when increasing the number of training points.

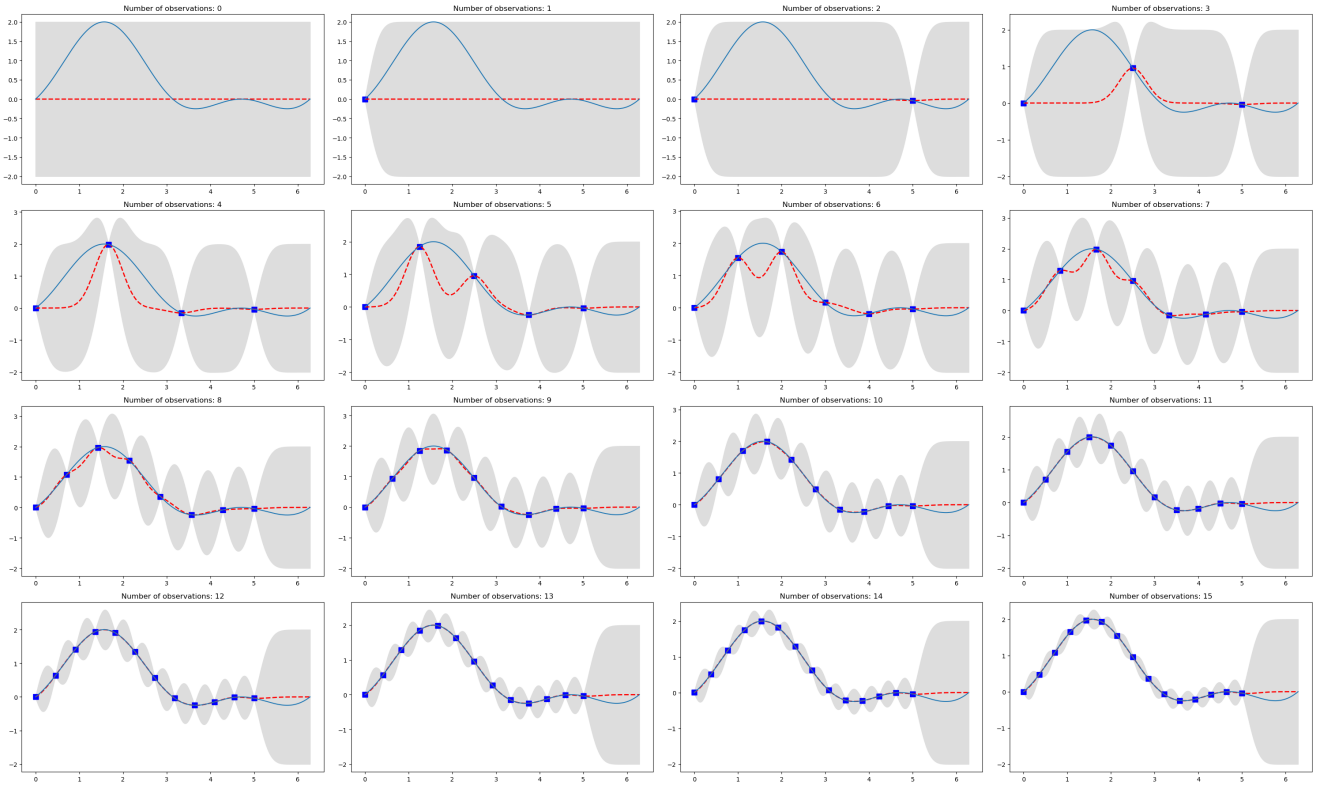

Figure 2: An overview of all 16 iterations for the Gaussian Process. It's beautfiul isn't it?

In case you're overwelmed by the information density of this graph, here are the extracted versions for first 3, second 4, fourth 5, eight 6 and sixtheen iteration 7 respectivly, showing n-1 number of observations, since we start with zero observations.

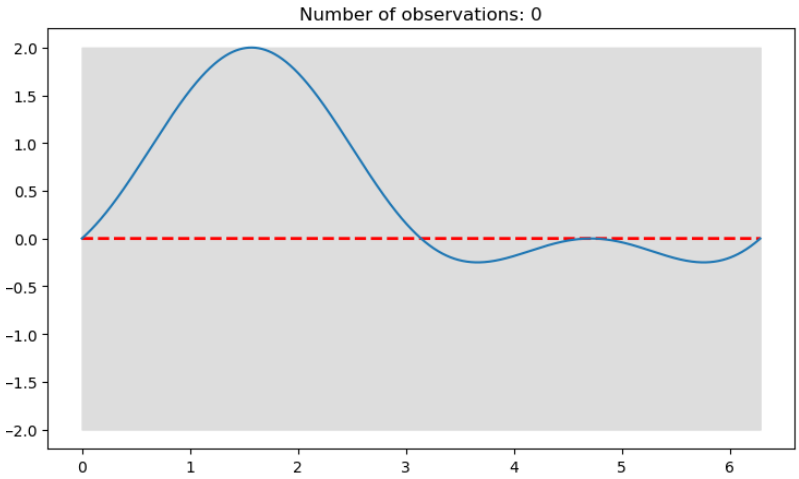

Figure 3: Gaussian Process for the first iteration with zero observations. We assume zero mean.

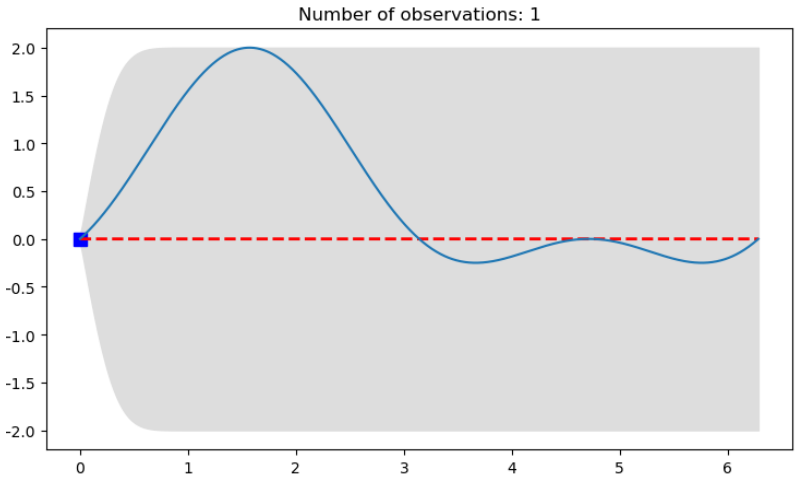

Figure 4: Gaussian Process for the second iteration with one observations.

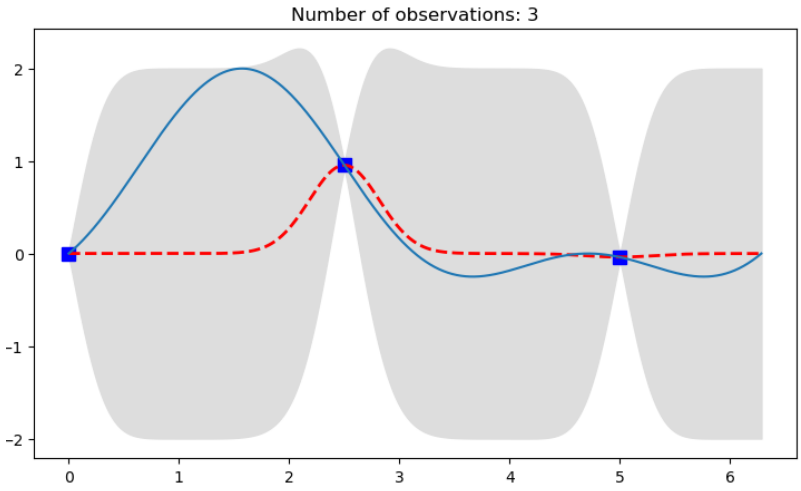

Figure 5: Gaussian Process for the fourth iteration with three observations.

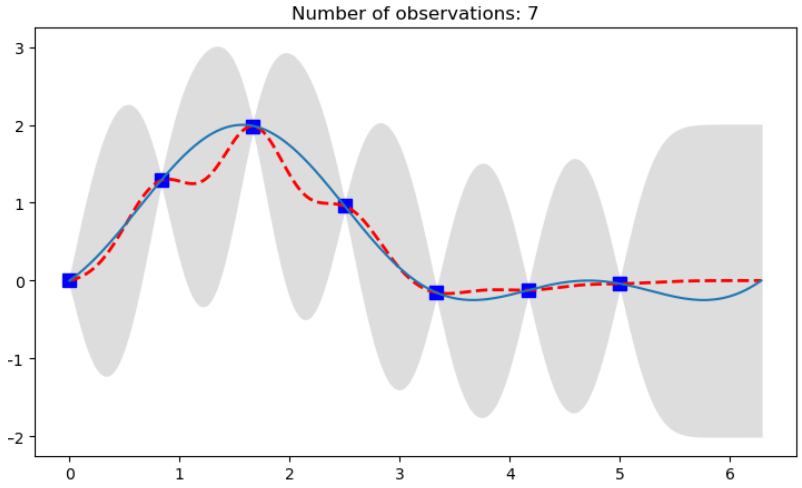

Figure 6: Gaussian Process for the eight iteration with seven observations.

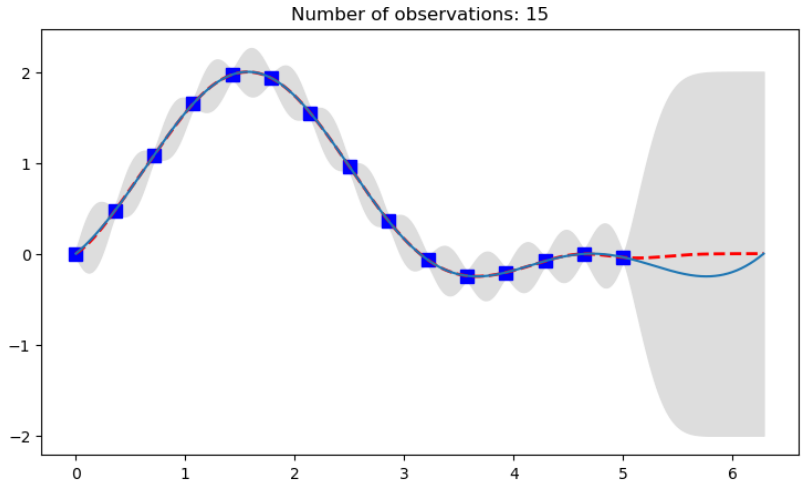

Figure 7: Gaussian Process for the sixteenth iteration with fifteen observations.

```
import numpy as np
import matplotlib.pyplot as plt

def kernel(a, b):
    l = 0.1  # Set parameter
    squared_distance = np.sum(a ** 2, 1).reshape(-1, 1) + np.sum(b ** 2, 1) - 2 * np.dot(a, b
    return np.exp(-.5 * (1 / l) * squared_distance)


# Unkown target function (used to generate data)
f = lambda x: (np.sin(x) + np.sin(x) ** 2).flatten()
s = 0.001  # Noise variance


fig = plt.figure()
for N in range(16):
    x_train = np.linspace(0, 5, N).reshape(-1, 1)  # Increase N to generate more training po
    y = f(x_train) + s * np.random.randn(N)  # Generate noisy train data
```

```
        # Calculate kernel on train data
        K = kernel(x_train, x_train)
        L = np.linalg.cholesky(K + s * np.eye(N))  # Get square-root of matrix with cholesky

        # Points at which we want to predict the function
        n = 1256  # calculate from the given intervall
        x_test = np.linspace(0, 2 * np.pi, n).reshape(-1, 1)

        # Compute kernel on train test similarity
        K_s = kernel(x_train, x_test)

        # compute the mean for the test points. Equation systems is used instead of naively in
        alpha = np.linalg.solve(L, K_s)

        mu = np.dot(alpha.T, np.linalg.solve(L, y))
        mu2 = mu.reshape(-1, 1)

        # Compute kernel on test data
        K_ss = kernel(x_test, x_test)

        # compute the variance for the  test points.
        h2 = np.diag(K_ss) - np.sum(alpha ** 2, axis=0)
        h = np.sqrt(h2)


        # Get posterior for test points
        L_ss = np.linalg.cholesky(K_ss + 1e-6 * np.eye(len(x_test)) - np.dot(alpha.T, alpha))
        f_pos = mu.reshape(-1, 1) + np.dot(L_ss, np.random.normal(size=(len(x_test), 1)))

        """ Plot the results"""
        ax = fig.add_subplot(4, 4, N+1)
        ax.plot(x_train, y, 'bs', ms=8)  # Plot obersvations
        try:
            ax.fill_between(x_test.flat, mu-2*h, mu+2*h, color="#dddddd")  # Plot sigma
        except:
            print("Couldnt fill empty space")  # Normally this shouldn't matter

        ax.plot(x_test, mu, 'r--', lw=2)  # Plot mu

        #plt.plot(x_test, f_pos)  # Plot the posterior

        plt.plot(x_test, f(x_test))  # Plot True function

        title_string = "Number of observations: " + str(N)
        plt.title(title_string)

    plt.show()
```