1. Gaussian Process
    a. Code with detailed explanation

Gaussian process is memory-based methods, that mean ours training data is valuable, and we would like to utilize the training data for prediction. So, we want to find something we may be interested, but in the origin data space sometimes it is hard to find the relation to do the regression or classification, project to another feature space will make this problem easier, we need to evaluate the similarity between feature data in the feature space, dot product is somehow related to similarity that is kernel function.

In the homework we are asked to use a rational quadratic kernel to compute similarities between different points. Rational quadratic kernel is as following:

$$k(x_a, x_b) = \sigma^2(1 + \frac{\| x_a - x_b \|^2}{2\alpha l^2})^{-\alpha}$$

- $\sigma^2$ the overall variance
- $l$ the lengthscale
- $\alpha$ the scale-mixture

By the way most of the information at the internet, the rational quadratic kernel is without the sigma square parameter.

The following screenshot is how did I implement the kernel. Basically, is same as the equation, in the distance between $x_a - x_b$ we can compute as matrix method, that can speed up the time than the iterative method.

```
def rational_quadratic_kernel(X1, X2, sigma, lengthscale, alpha):
    dist_matrix = np.sum(X1**2,1).reshape(-1,1) + np.sum(X2**2,1) - 2 * X1 @ X2.T
    kernel = sigma ** 2 * ( (1 + ( (dist_matrix) / (2*alpha*(lengthscale**2))) ) ** -alpha)
    return kernel
```

Now, we already chose the kernel for the Gaussian Process, the next step that we should do is estimate the proper hyper-parameter. And you may be confused that what is the parameter of the Gaussian Process, answer is obvious, parameters of kernel.

To estimate the proper parameter, first we defined the marginal likelihood equation, also log operation can get more benefit, so turn to negative log marginal likelihood. Second with this negative function we can use minimize function from the library scipy.optimize, like the following screenshot, the return result we be the proper hyper-parameters of this kernel.

```
res = minimize(fun=negative_log_marginal_likelihood,
        x0 = params,
        bounds=((1e-6,None),(1e-6,None),(1e-6,None)),
        args=(X, Y, beta), method='L-BFGS-B', options={}
        )

print(res.x)
```

b. Experiment settings and results

So, the big picture of Gaussian Process will like that following screenshot, let me Introduce the whole story, first we get our data capital X and Y, put the X and Y to the kernel to get the mean and covariance, the equation like the following, $x$ is the training data, $x^*$ is the testing data

$$\mu(x^*) = k(x, x^*)^T C^{-1} y$$

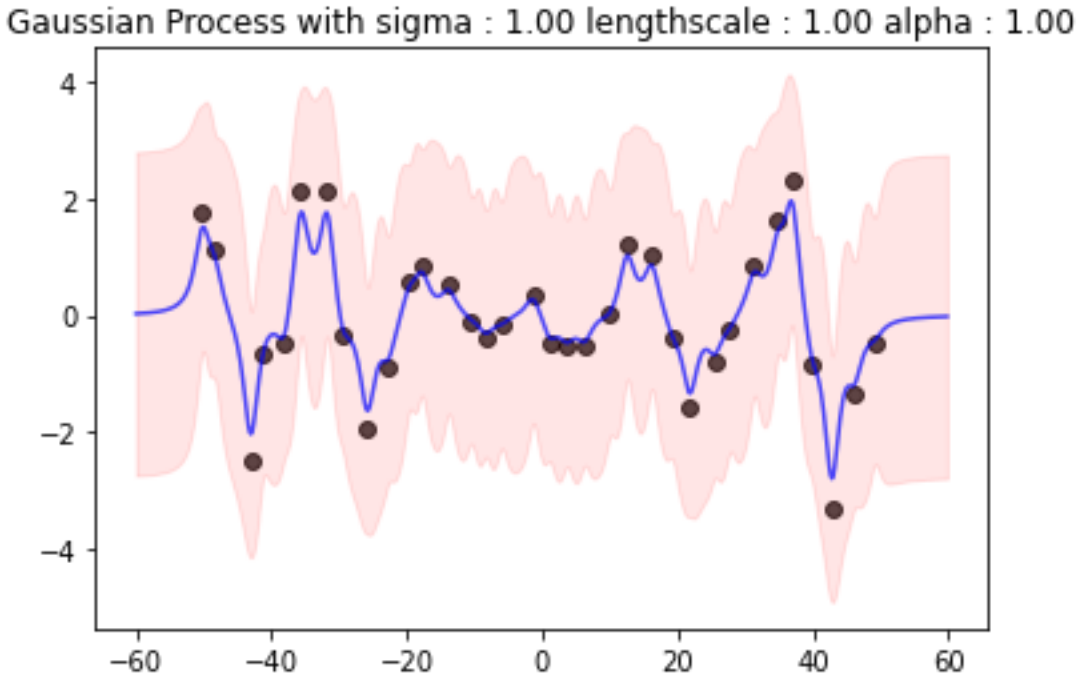
$$\sigma^2(x^*) = k^* - k(x, x^*)^T C^{-1} k^*$$

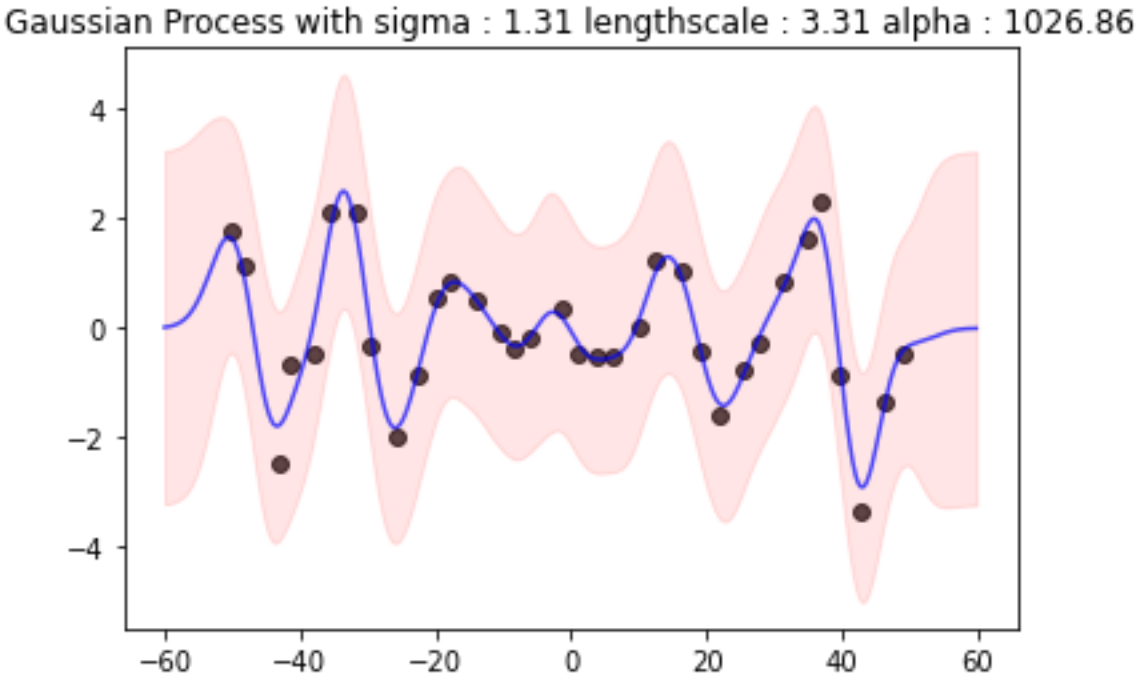
$$k^* = k(x^*, x^*) + \beta^{-1}$$

The mean and covariance can let us to predict the testing data, but if you still remember that we need to find the proper hyper-parameter of the kernel, we get the proper parameter by the minimize function with the negative log marginal likelihood, put the result to the kernel function, get the best mean and covariance, then we can compute the result and plot it.

```
def gaussian_process():
    X, Y = load_data_GP("data/input.data")
    X_s = np.arange(-60.0, 60.0, 0.2).reshape(-1,1)

    sigma = 1
    lengthscale = 1
    alpha = 1
    beta = 5
    params = [sigma, lengthscale, alpha]
    mu_s, conv_s = posterior(X_s, X, Y, params, beta)

    GP_plt(X_s, X, Y, mu_s, conv_s, params)

    res = minimize(fun=negative_log_marginal_likelihood,
            x0 = params,
            bounds=((1e-6,None),(1e-6,None),(1e-6,None)),
            args=(X, Y, beta), method='L-BFGS-B', options={}
            )

    print(res.x)

    mu_s, conv_s = posterior(X_s, X, Y, res.x, beta)

    GP_plt(X_s, X, Y, mu_s, conv_s, res.x)
```

The first image of following is the first GP_plt function at the above screenshot, and the second image is the second GP_plt function that set as the proper hyper-parameter, obvious we can see that variance is more smaller that the origin one, blue mean line and variance pink area are more smoothing and close to the black training data.



c. Observation and discussion

Gaussian Process is the memory-base methods and the kernel methods, that perhaps most powerful regressor before deep learning, the most powerful place of the kernel method is that even we don't know the mapping function for project to feature space, that is also okay. Choosing the proper kernel even unsure the feature function and find the proper hyper-parameter by minimize the negative log marginal likelihood with the kernel that we chose. With the proper parameter, sent it to the kernel get the mean and covariance. Then we can use to compute the regression and plot it.

With the effect of the rational quadratic kernel, $l$ (lengthscale) parameter controls the smoothness of the function. Higher $l$ values lead to smoother functions and therefore to coarse approximations of the training data. Lower values make function more wiggly with wide uncertainly regions between training data point. $\sigma^2$ parameter controls the vertical variation of the functions drawn from the Gaussian Process, that is the distance between the blue mean line and the upper and lower bound of pink area.

Reference: http://krasserm.github.io/2018/03/19/gaussian-processes/

2. SVM

a. A code with detailed explanations
Part 1

SVM (Support Vector machine) is also memory-based method and kernel method, and as I mentioned above using the kernel method we need to define the kernel first. In this part we use the Libsvm library to implement the SVM, as the following screenshot, first we prepare the data that we want to classification on images of hand-written digits, second define the problem just put the label and image pixel as parameter, third use the svm_train function, set the problem we defined and what kernel we want to use, for instance linear, polynomial or RBF. Last use the model after training to predict the testing data.

```
def part_1():
    X_train, Y_train = get_data_SVM("data/", "train")
    X_test, Y_test = get_data_SVM("data/", "test")

    kernel_mode = {"linear":"-t 0", "polynomial":"-t 1 -b 1", "RBF":"-t 2"}
    for kernel, params in kernel_mode.items():
        print(kernel)

        prob = svm_problem(Y_train,X_train)
        model = svm_train(prob, params)
        print(model)
        p_label, p_acc, p_val = svm_predict(Y_test, X_test, model)

        print(p_acc)
```

Part 2

To find the best kernel and its proper parameter for this classification problem, we use Grid search to find that combination, and the whole idea is easy, just like the brute force we try all the combination train and predict, find the best performance and record.

Like the following screenshot. we set the kernel_mode and cost parameter list and gamma parameter list, then use multilayer for loop to try all combination, find the best parameter result for each kernel.

Cost parameter for C-SVC:  0.001, 0.01, 0.1, 1, 10, 100

Gamma parameter for polynomial and RBF: 0.001, 0.01, 0.1, 1, 10, 100

```
def grid_search():

    X_train, Y_train = get_data_SVM("data/", "train")
    X_test, Y_test = get_data_SVM("data/", "test")

    best_kernel_acc = {"linear":0, "polynomial":0, "RBF":0}
    best_kernel_params = {"linear":f"", "polynomial":f"", "RBF":f""}

    kernel_mode = {"linear":0, "polynomial":1, "RBF":2}
    cost_list   = [0.001, 0.01, 0.1, 1, 10, 100]
    gamma_list  = [0.001, 0.01, 0.1, 1, 10, 100]

    for kernel, index in kernel_mode.items():
        print(kernel)
        for cost in cost_list:
            for gamma in gamma_list:
                params = f"-t {index} -c {cost} -g {gamma}"
                if index == 0:
                    params = f"-t {index} -c {cost}"

                prob = svm_problem(Y_train,X_train)
                model = svm_train(prob, params)

                p_label, p_acc, p_val = svm_predict(Y_test, X_test, model)

                print(p_acc,params)

                if p_acc[0] >= best_kernel_acc[kernel]:
                    best_kernel_acc[kernel] = p_acc[0]
                    if index == 0:
                        best_kernel_params[kernel] = f"kernel: {kernel} cost: {cost}"
                    else:
                        best_kernel_params[kernel] = f"kernel: {kernel} cost: {cost} gamma: {gamma}"
                if index == 0:
                    break
```

Part 3

In part 3 we are going to use custom kernel to the SVM model, we mix the linear kernel and RBF kernel, as the following screenshot, show how to implement the kernel.

```
def linear_kernel(X1,X2):
    kernel = X1 @ X2.T
    return kernel

def RBF_kernel(X1,X2,gamma):
    dist_matrix = np.sum(X1**2,1).reshape(-1,1) + np.sum(X2**2,1) - 2 * X1 @ X2.T
    kernel = np.exp(-gamma * dist_matrix)
    return kernel

def linear_RBF_kernel(X1,X2,gamma):
    kernel = np.add(linear_kernel(X1,X2), RBF_kernel(X1,X2,gamma))
    kernel = np.hstack((np.arange(1,len(X1)+1).reshape(-1,1),kernel))
    return kernel
```

After the defined the custom kernel, we can use the same mode like the part 1, but the difference is that this time the when we define the problem, we should put the Y and kernel as the parameter and turn the iskernel parameter on. Finally use the after trained model and the testing data kernel to predict the result.

```
def part_3():

    X_train, Y_train = get_data_SVM("data/", "train")
    X_test, Y_test = get_data_SVM("data/", "test")

    kernel = linear_RBF_kernel(X_train, X_train, 0.01)
    kernel_s = linear_RBF_kernel(X_test, X_train, 0.01)
    probelm = svm_problem(Y_train, kernel, isKernel=True)
    params = f"-t 4"
    model = svm_train(probelm, params)
    p_label, p_acc, p_val = svm_predict(Y_test, kernel_s, model)

    print(f"Linear + RBF kernel")
    print(p_acc)
```

b. Experiments settings and results

Part 1

With the Libsvm library we can use the -t parameter to choose the kernel type, for example 0 for linear, 1 for polynomial, 2 for radial basis function. Just like the above screenshot.

Results show on the following screenshot.

```
linear
<svm.svm_model object at 0x000001EB7287C840>
Accuracy = 95.08% (2377/2500) (classification)
(95.08, 0.1404, 0.931149802516624)
polynomial
<svm.svm_model object at 0x000001EB6C70BD40>
Model supports probability estimates, but disabled in predicton.
Accuracy = 34.68% (867/2500) (classification)
(34.68, 2.6212, 0.14887572191533946)
RBF
<svm.svm_model object at 0x000001EB68606D40>
Accuracy = 95.32% (2383/2500) (classification)
(95.32000000000001, 0.1492, 0.9271864783823697)
```

Part 2

Results are show on the following.

In the linear kernel: best accuracy is 95.96%

Parameter cost: 0.01

In the polynomial kernel: best accuracy is 97.76%

Parameters cost: 10 gamma: 0.01

In the RBF kernel: best accuracy is 98.2%

Parameters cost: 10 gamma: 0.01

Part 3

Results are show on the following screenshot.

```
Accuracy = 95.32% (2383/2500) (classification)
Linear + RBF kernel
(95.32000000000001, 0.1324, 0.9350534525357612)
```

c. Observation and discussion

As the result we can see the power of the SVM, as the mentioned in the course that SVM perhaps is the most powerful method before the deep learning, the accuracy is higher than the Gaussian and Naïve Bayes classification homework that we did before the midterm, use the kernel trick SVM can efficiently perform a non-linear classifier.

Part 1

The accuracy of the RBF kernel is the best, then is linear kernel and I and not clear that why the performance of the polynomial is not good as others.

Part 2

Grid search as brute force, spend too many times to combination and search, for example in combinatorics, the multiplication principle if we have three sets of parameters, then the number of the combination is the product of each set number.

I am wonder to know to if that exist more efficiently way or algorithm to do, instead of brute force.

Part 3

I set the parameter to the custom kernel which is mix of the linear and RBF and set it as the results parameters after doing grid search. The accuracy of the custom kernel is same as the RBF kernel, custom kernel doesn't get better performance, and I think that may because the mixture kernel is weaker or equate than the origin one, mixture kernel will get more limitation, and the power range is the subset of the origin one. Just like the neural network, fully connect layer is the powerful, and other like Convolutional neural network that we force to put some limitation on it.

Reference: https://www.csie.ntu.edu.tw/~cjlin/libsvm/