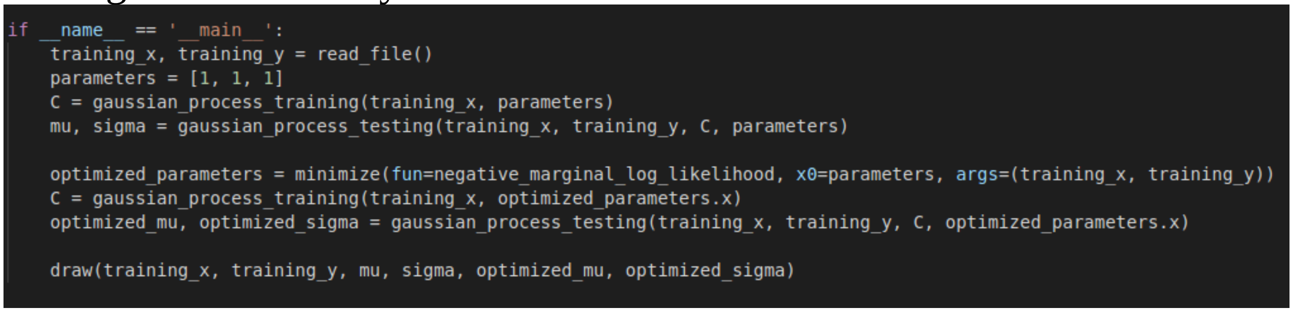
Machine Learning HW5-1

309551064 張凱翔

1. Code With Detailed Explanations

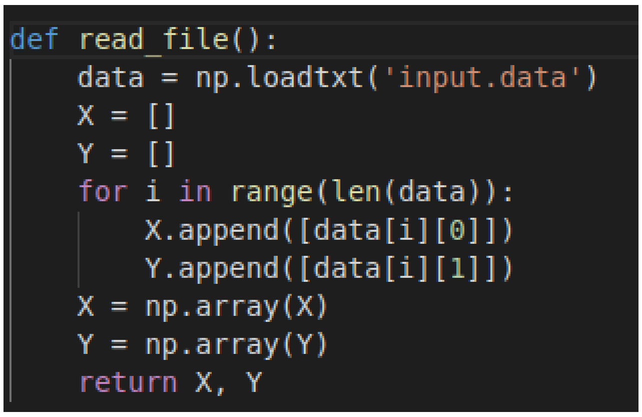
Part 1.

The figure below is my main function.



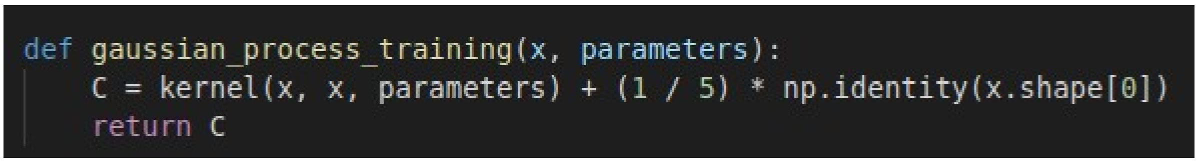
```python
if __name__ == '__main__':
    training_x, training_y = read_file()
    parameters = [1, 1, 1]
    C = gaussian_process_training(training_x, parameters)
    mu, sigma = gaussian_process_testing(training_x, training_y, C, parameters)

    optimized_parameters = minimize(fun=negative_marginal_log_likelihood, x0=parameters, args=(training_x, training_y))
    C = gaussian_process_training(training_x, optimized_parameters.x)
    optimized_mu, optimized_sigma = gaussian_process_testing(training_x, training_y, C, optimized_parameters.x)

    draw(training_x, training_y, mu, sigma, optimized_mu, optimized_sigma)
```

First, I read data from “input.data” with numpy function “loadtxt” and separate X and Y into two lists and transform to two numpy arrays.



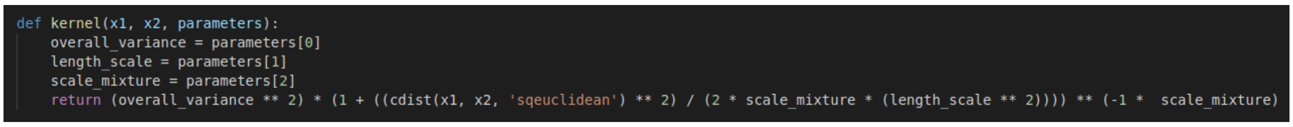
```python
def read_file():
    data = np.loadtxt('input.data')
    X = []
    Y = []
    for i in range(len(data)):
        X.append([data[i][0]])
        Y.append([data[i][1]])
    X = np.array(X)
    Y = np.array(Y)
    return X, Y
```

After reading file, I initial the parameters for computing rational quadratic kernel. Then calculating the covariance matrix of the input data that will be needed when calculating mean and variance on the prediction phase. The formula to compute covariance matrix is found on pdf and rational quadratic kernel is founded on Internet (https://peterroelants.github.io/posts/gaussian-process-kernels/#Rational-quadratic-kernel)



```python
def gaussian_process_training(x, parameters):
    C = kernel(x, x, parameters) + (1 / 5) * np.identity(x.shape[0])
    return C
```

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

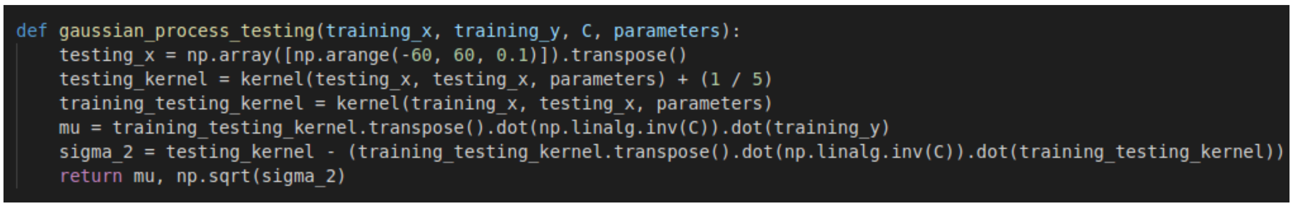


```python
def kernel(x1, x2, parameters):
    overall_variance = parameters[0]
    length_scale = parameters[1]
    scale_mixture = parameters[2]
    return (overall_variance ** 2) * (1 + ((cdist(x1, x2, 'sqeuclidean') ** 2) / (2 * scale_mixture * (length_scale ** 2)))) ** (-1 *  scale_mixture)
```

Rational quadratic kernel

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

After computing the covariance matrix of training data, it's now for the prediction phase. Initial 1200 points between -60, 60, and calculate the mean and variance according to the covariance of training data. The formula is found on the pdf.



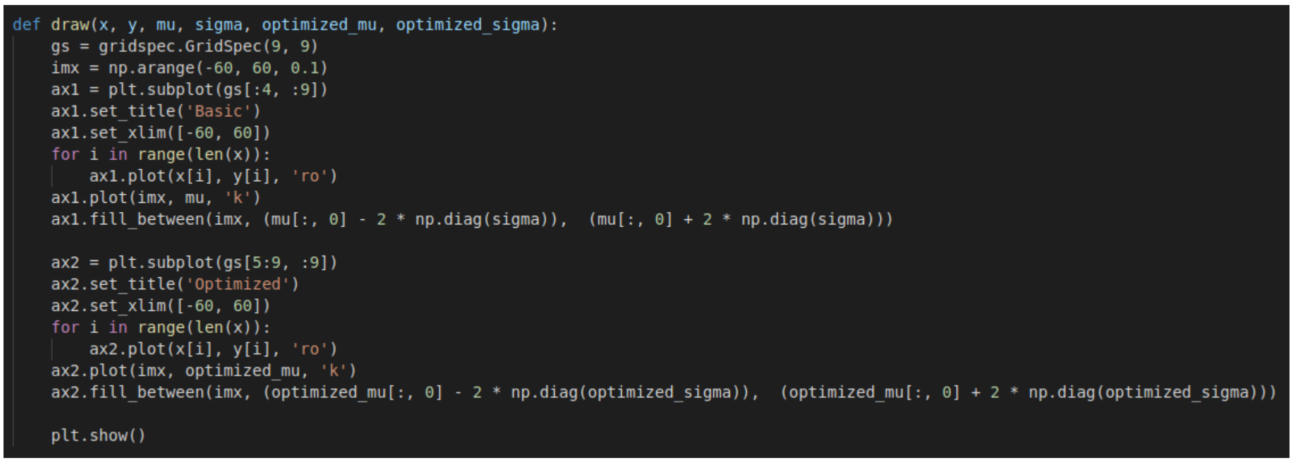
```python
def gaussian_process_testing(training_x, training_y, C, parameters):
    testing_x = np.array([np.arange(-60, 60, 0.1)]).transpose()
    testing_kernel = kernel(testing_x, testing_x, parameters) + (1 / 5)
    training_testing_kernel = kernel(training_x, testing_x, parameters)
    mu = training_testing_kernel.transpose().dot(np.linalg.inv(C)).dot(training_y)
    sigma_2 = testing_kernel - (training_testing_kernel.transpose().dot(np.linalg.inv(C)).dot(training_testing_kernel))
    return mu, np.sqrt(sigma_2)
```

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

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

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

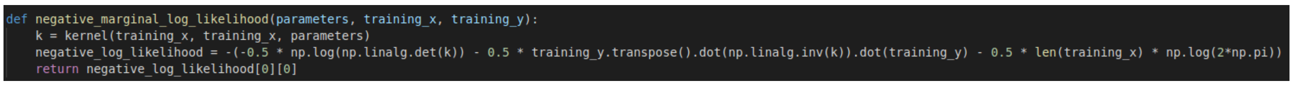
Now, I have the mean and variance, so I can draw the 95% confidence interval with "matplotlib.pyplot.fill_between". The interval is [mean - 2 * variance ^ 0.5, mean + 2 * variance ^ 0.5]. I set axis X to fall between [-60, 60], draw training data points and the line of the mean. I memorize basic mean, variance and optimized mean, variance then draw on the same figure to have a better comparison.



```python
def draw(x, y, mu, sigma, optimized_mu, optimized_sigma):
    gs = gridspec.GridSpec(9, 9)
    imx = np.arange(-60, 60, 0.1)
    ax1 = plt.subplot(gs[:4, :9])
    ax1.set_title('Basic')
    ax1.set_xlim([-60, 60])
    for i in range(len(x)):
        ax1.plot(x[i], y[i], 'ro')
    ax1.plot(imx, mu, 'k')
    ax1.fill_between(imx, (mu[:, 0] - 2 * np.diag(sigma)),  (mu[:, 0] + 2 * np.diag(sigma)))

    ax2 = plt.subplot(gs[5:9, :9])
    ax2.set_title('Optimized')
    ax2.set_xlim([-60, 60])
    for i in range(len(x)):
        ax2.plot(x[i], y[i], 'ro')
    ax2.plot(imx, optimized_mu, 'k')
    ax2.fill_between(imx, (optimized_mu[:, 0] - 2 * np.diag(optimized_sigma)),  (optimized_mu[:, 0] + 2 * np.diag(optimized_sigma)))

    plt.show()
```

Part 2.

For optimizing the parameters using in kernel computing, I use "scipy.optimize.minimize " with the default setting to minimize the negative marginal log-likelihood. The formula of marginal log-likelihood is found on the pdf. But for the "negative" in the spec, the formula should add a negative sign.



```python
def negative_marginal_log_likelihood(parameters, training_x, training_y):
    k = kernel(training_x, training_x, parameters)
    negative_log_likelihood = -(-0.5 * np.log(np.linalg.det(k)) - 0.5 * training_y.transpose().dot(np.linalg.inv(k)).dot(training_y) - 0.5 * len(training_x) * np.log(2*np.pi))
    return negative_log_likelihood[0][0]
```

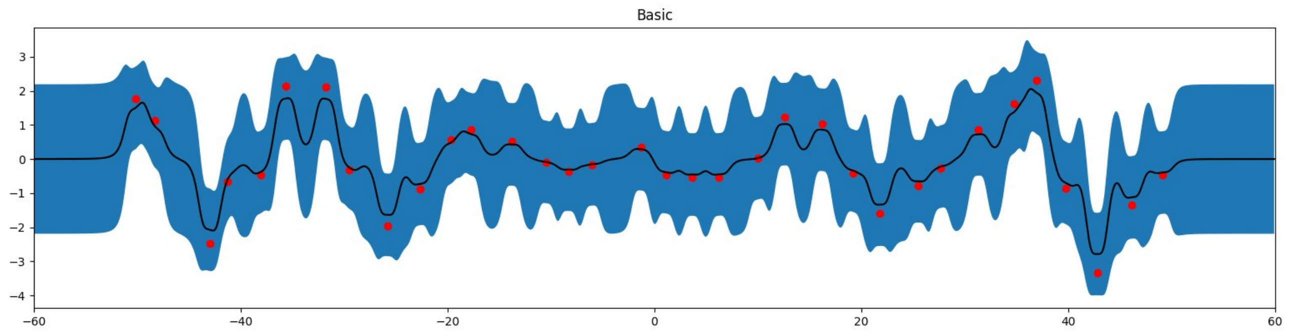
$$\ln p(\mathbf{y}|\theta) = -\frac{1}{2}\ln\ |\mathbf{C}_\theta| - \frac{1}{2}\mathbf{y}^\top\mathbf{C}_\theta^{-1}\mathbf{y} - \frac{N}{2}\ln\ (2\pi)$$

After obtaining optimized parameters, I substitute the initial parameters with optimized one and go through the same procedure as Part 1 to get the 95% confidence interval.

2. Experiments Settings and Results

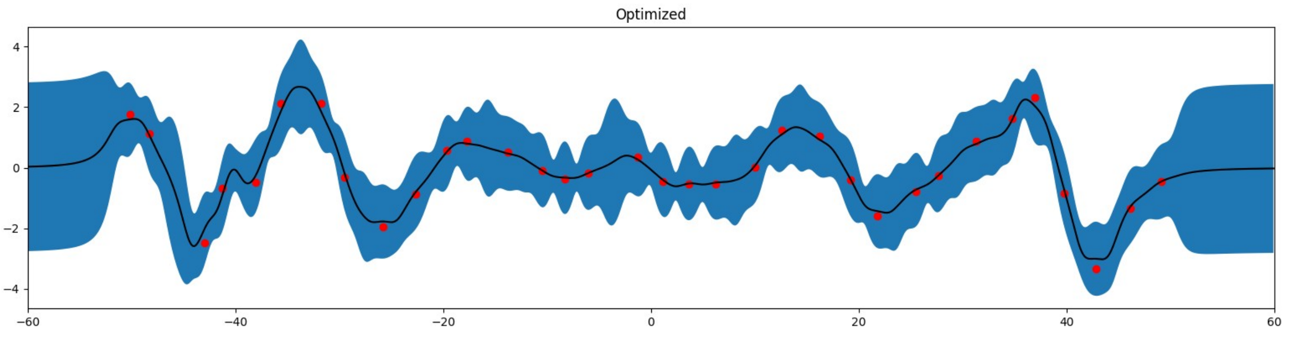
Part 1.

σ = 1, ℓ = 1, α = 1



Part 2.

σ = 1.32224603, ℓ = 3.65379347, α = 0.5538034



3. Observation and Discussion

Using the optimized parameters makes the line of mean more smoother and more fit the training data. Furthermore, the 95% confidence interval is more strict than the initial parameters.