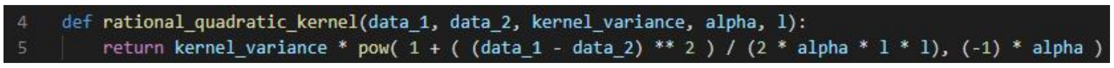
I. Gaussian Process

a. Code with detailed explanation

Part 1 (Source code: gaussian_1.py)

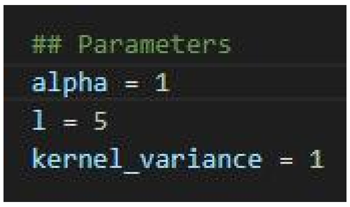
$$k(x_a, x_b) = \sigma^2 \left(1 + \frac{||x_a - x_b||^2}{2\alpha\ell^2}\right)^{-\alpha}$$

Above is the definition of the rational quadratic kernel.



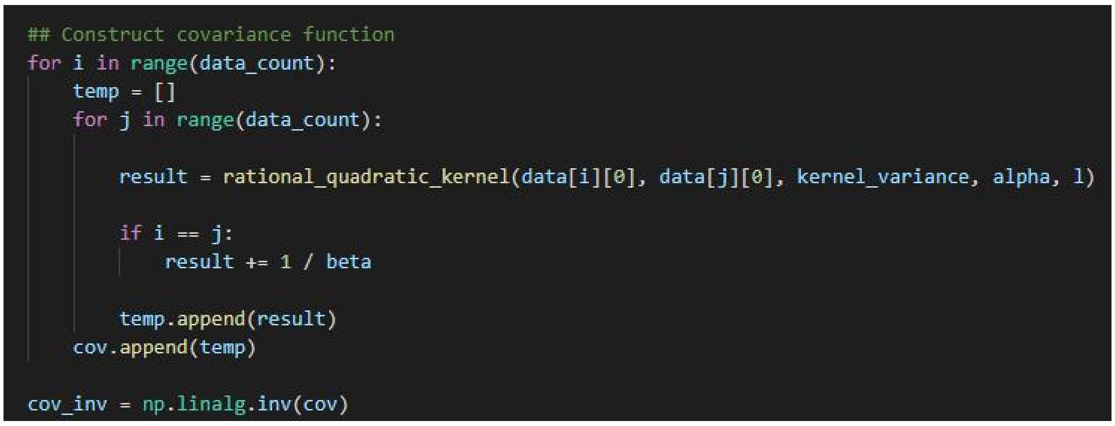
```python
def rational_quadratic_kernel(data_1, data_2, kernel_variance, alpha, l):
    return kernel_variance * pow( 1 + ( (data_1 - data_2) ** 2 ) / (2 * alpha * l * l), (-1) * alpha )
```

This function returns the rational quadratic kernel between data_1 and data_2 with respect to parameters variance (kernel_variance), scale-mixture (alpha), and lengthscale (l).



```python
## Parameters
alpha = 1
l = 5
kernel_variance = 1
```

These are the parameters of the kernel. Here I manually put their values in.



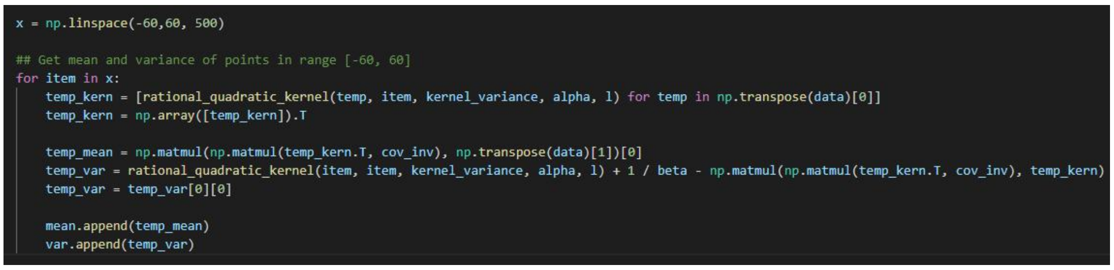
```python
## Construct covariance function
for i in range(data_count):
    temp = []
    for j in range(data_count):

        result = rational_quadratic_kernel(data[i][0], data[j][0], kernel_variance, alpha, l)

        if i == j:
            result += 1 / beta

        temp.append(result)
    cov.append(temp)

cov_inv = np.linalg.inv(cov)
```

This the construction of the covariance matrix, which the i th row and j th column are defined as

kernel(x_i, x_j)   if i ≠ j,

kernel(x_i, x_j) + $\beta^{(-1)}$ if i = j

cov_inv is the inverse of the covariance matrix.



```
x = np.linspace(-60,60, 500)

## Get mean and variance of points in range [-60, 60]
for item in x:
    temp_kern = [rational_quadratic_kernel(temp, item, kernel_variance, alpha, l) for temp in np.transpose(data)[0]]
    temp_kern = np.array([temp_kern]).T

    temp_mean = np.matmul(np.matmul(temp_kern.T, cov_inv), np.transpose(data)[1])[0]
    temp_var = rational_quadratic_kernel(item, item, kernel_variance, alpha, l) + 1 / beta - np.matmul(np.matmul(temp_kern.T, cov_inv), temp_kern)
    temp_var = temp_var[0][0]

    mean.append(temp_mean)
    var.append(temp_var)
```

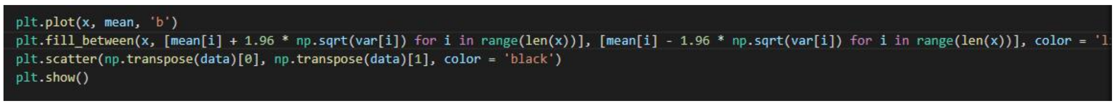
x is a list of evenly distributed points in the range [-60, 60]

Then for each point in x, I calculate the mean and variance, which can be calculated using the conditional probability as follows.

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

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

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



```
plt.plot(x, mean, 'b')
plt.fill_between(x, [mean[i] + 1.96 * np.sqrt(var[i]) for i in range(len(x))], [mean[i] - 1.96 * np.sqrt(var[i]) for i in range(len(x))], color = 'l
plt.scatter(np.transpose(data)[0], np.transpose(data)[1], color = 'black')
plt.show()
```

These lines respectively plot the mean of f, the 95% confidence interval of f, and the training data. The 95% confidence interval can be calculated using μ ± 1.96 * σ

Part 2 (Source code: gaussian_2.py)

The code for part 2 is similar to part 1, except I try to find the optimal parameter by trying to find the maximum likelihood. Therefore I'll only discuss about the parts of the code here that are different compared to part 1.

The marginal likelihood is as follows.

$$p(\mathbf{y}|\theta) = \mathcal{N}(\mathbf{y}|\mathbf{0}, \mathbf{C}_\theta)$$

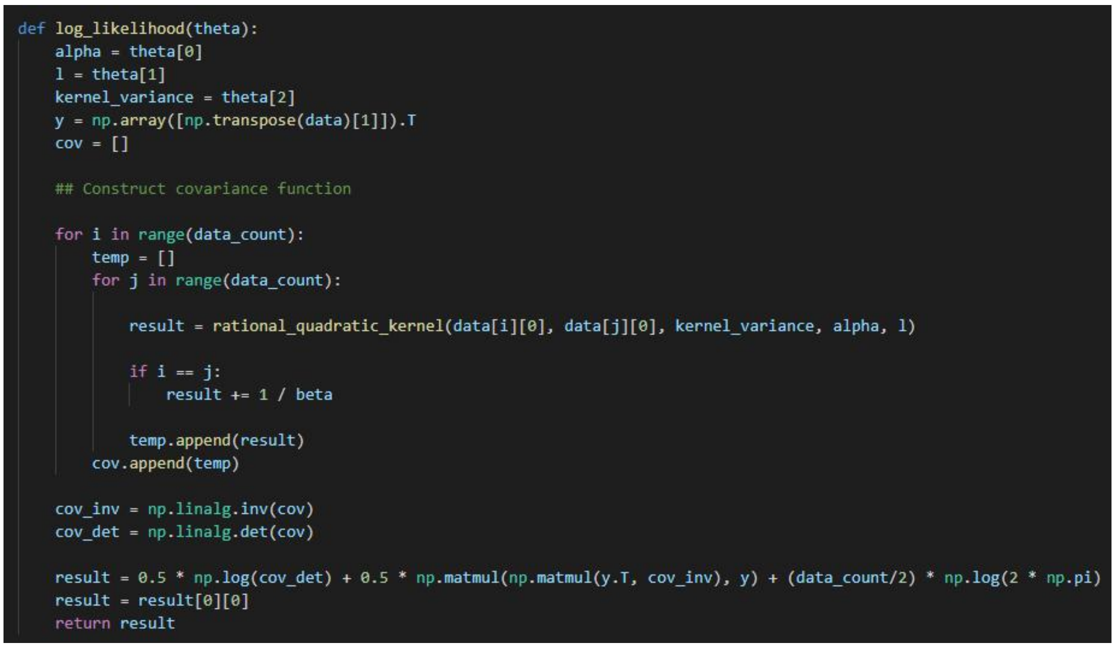
where θ is our parameters which consist of σ^2, α, and l.

Then the log likelihood is

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

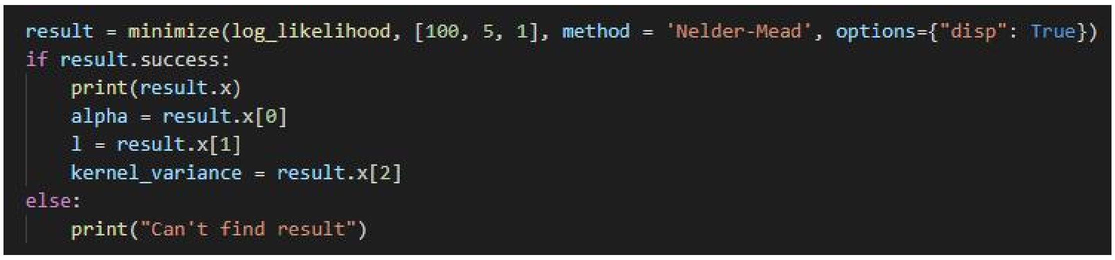
where N is the number of dimensions of our gaussian distribution.

We want to maximize these the log likelihood with respect to our parameters, which is the same as minimizing the negative log likelihood.



```python
def log_likelihood(theta):
    alpha = theta[0]
    l = theta[1]
    kernel_variance = theta[2]
    y = np.array([np.transpose(data)[1]]).T
    cov = []

    ## Construct covariance function

    for i in range(data_count):
        temp = []
        for j in range(data_count):

            result = rational_quadratic_kernel(data[i][0], data[j][0], kernel_variance, alpha, l)

            if i == j:
                result += 1 / beta

            temp.append(result)
        cov.append(temp)

    cov_inv = np.linalg.inv(cov)
    cov_det = np.linalg.det(cov)

    result = 0.5 * np.log(cov_det) + 0.5 * np.matmul(np.matmul(y.T, cov_inv), y) + (data_count/2) * np.log(2 * np.pi)
    result = result[0][0]
    return result
```

This function returns the negative log likelihood. Here N is the amount of data points (data_count).



```
result = minimize(log_likelihood, [100, 5, 1], method = 'Nelder-Mead', options={"disp": True})
if result.success:
    print(result.x)
    alpha = result.x[0]
    l = result.x[1]
    kernel_variance = result.x[2]
else:
    print("Can't find result")
```

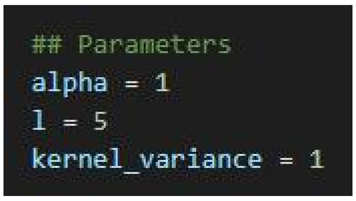
Then I optimize the three parameters using scipy.optimize.minimize.

The rest of the code then proceed normally as part one. For details, refer to part 1 of this section.

b. Experiments settings and results

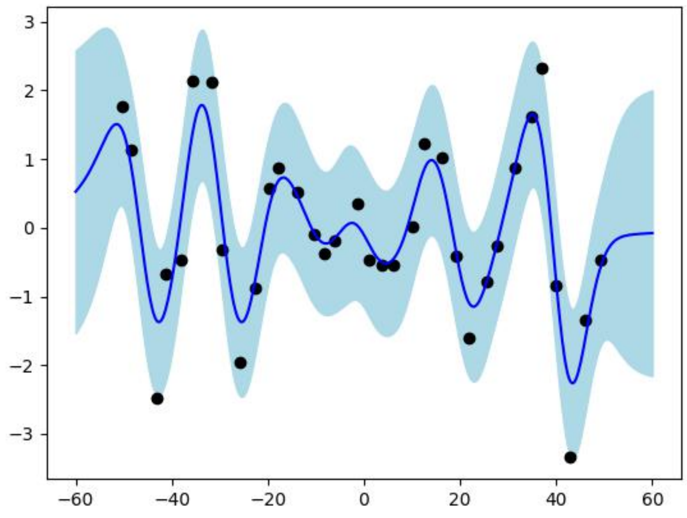
Part 1

I use the following parameters.



```
## Parameters
alpha = 1
l = 5
kernel_variance = 1
```

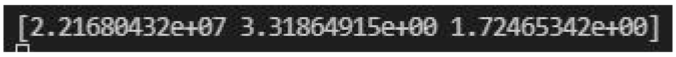
The result is as follow.



The black points are the test data, the blue line is the mean of f, and the blue area is the 95% confidence interval of f.

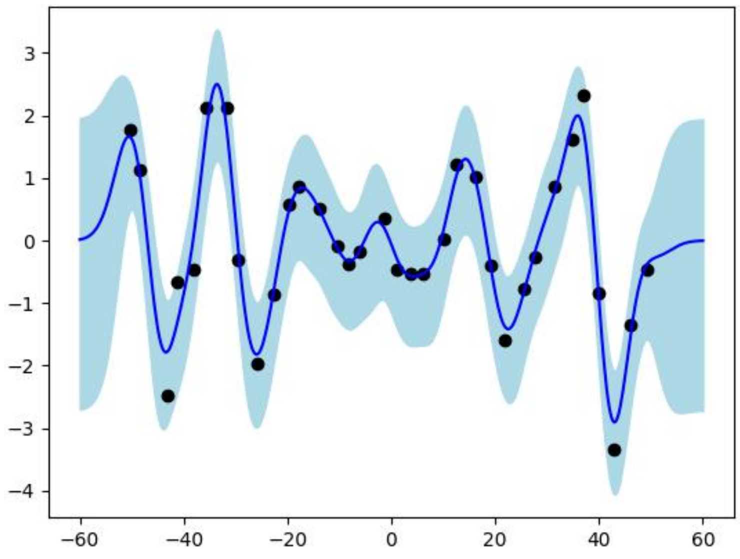
<u>Part 2</u>

Using the Nelder-Mead method for scipy.optimize.minimize and the initial guesses $\sigma^2 = 1$, $\alpha = 100$, and $l = 5$; The optimal values for α, l, σ^2 respectively are as follow.



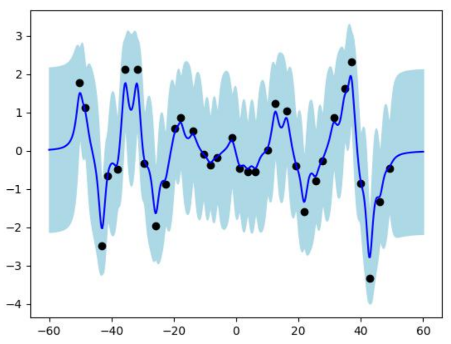
```
[2.21680432e+07 3.31864915e+00 1.72465342e+00]
```

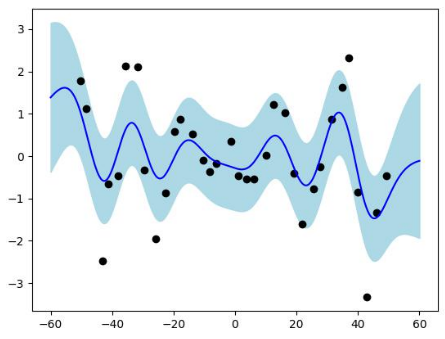
The visualization of the result is as follow.



c. Observations and discussion

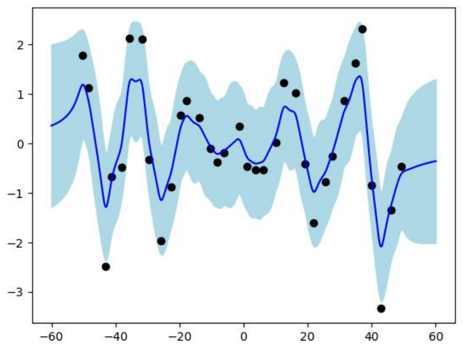
This left figure is the result when $l = 1$, while the right figure is when $l = 10$.

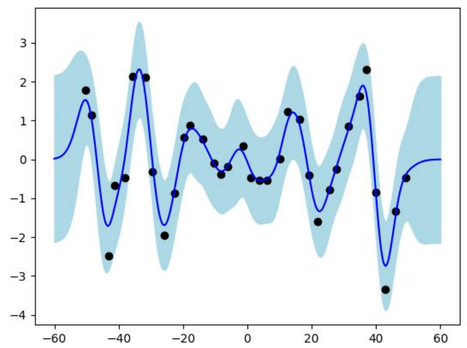




From this observation, I can conclude that lower l leads to our model being represented as polynomial with higher dimension. The mean here are also closer to our data points which lead to overfitting.
The opposite happens for higher l. The model is represented as polynomial with lower dimension, and the mean are farther than our points.

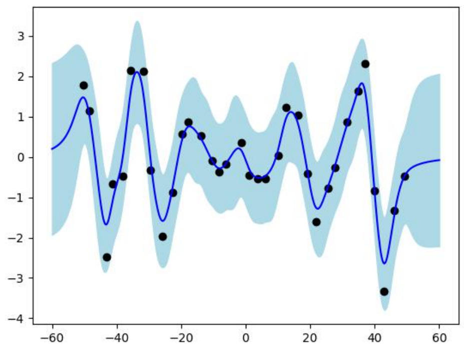
This left figure is the result when ɑ = 0.1, while the right figure is when ɑ = 10.

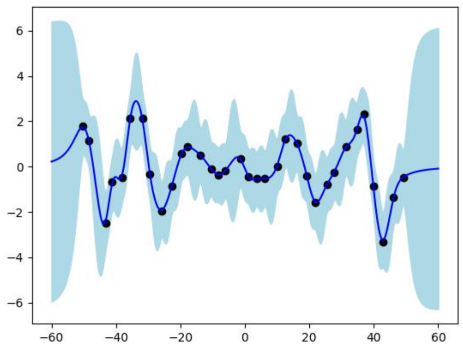




From this observation, I can conlude that the value of ɑ affects the local minima and maxima of our model. Lower ɑ makes the shape of our model follow the points more and causes our mdoel to have more local maxima and minima.

This left figure is the result when $\sigma^2 = 1$, while the right figure is when $\sigma^2 = 10$.





From this observation, I can conlude that the value of σ^2 affects the overall variance of our model. Higher σ^2 leads to higher variance at each point.

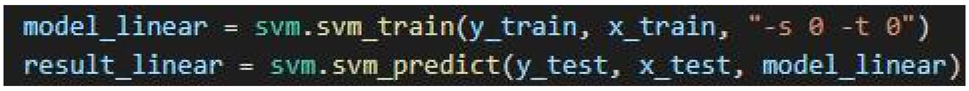
II. SVM

a. Code with detailed explanation

<u>Part 1 (Source code: svm_1.py)</u>

The linear kernel is defined as

$$x^{\mathrm{T}}x_{\mathrm{j}}$$

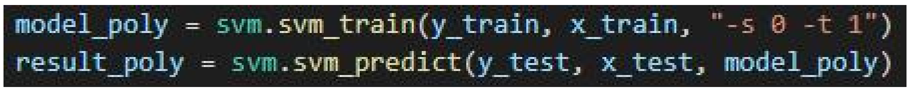


```
model_linear = svm.svm_train(y_train, x_train, "-s 0 -t 0")
result_linear = svm.svm_predict(y_test, x_test, model_linear)
```

This code train an SVM model with linear kernel, then use it to predict the labels of the test data.

The polynomial kernel is defined as

$$K(x,y) = (x^\mathsf{T} y + c)^d$$

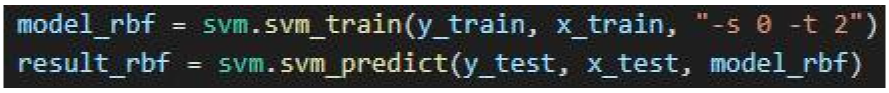


```
model_poly = svm.svm_train(y_train, x_train, "-s 0 -t 1")
result_poly = svm.svm_predict(y_test, x_test, model_poly)
```

This code train an SVM model with polynomial kernel, then use it to predict the labels of the test data.

The RBF kernel is defined as

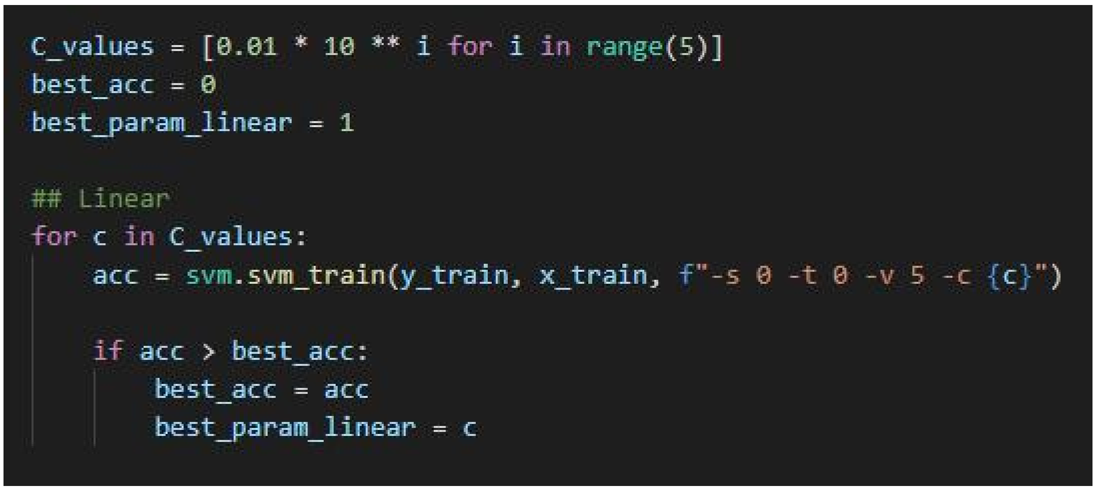
$$K(\mathbf{x},\mathbf{x'}) = \exp(-\gamma\|\mathbf{x}-\mathbf{x'}\|^2)$$



```
model_rbf = svm.svm_train(y_train, x_train, "-s 0 -t 2")
result_rbf = svm.svm_predict(y_test, x_test, model_rbf)
```

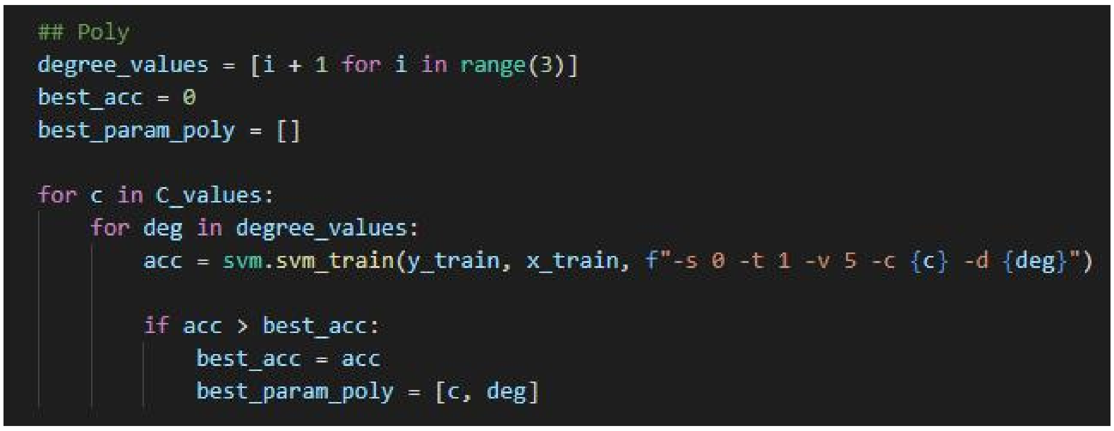
This code train an SVM model with RBF kernel, then use it to predict the labels of the test data.

Part 2 (Source code: svm_2.py)



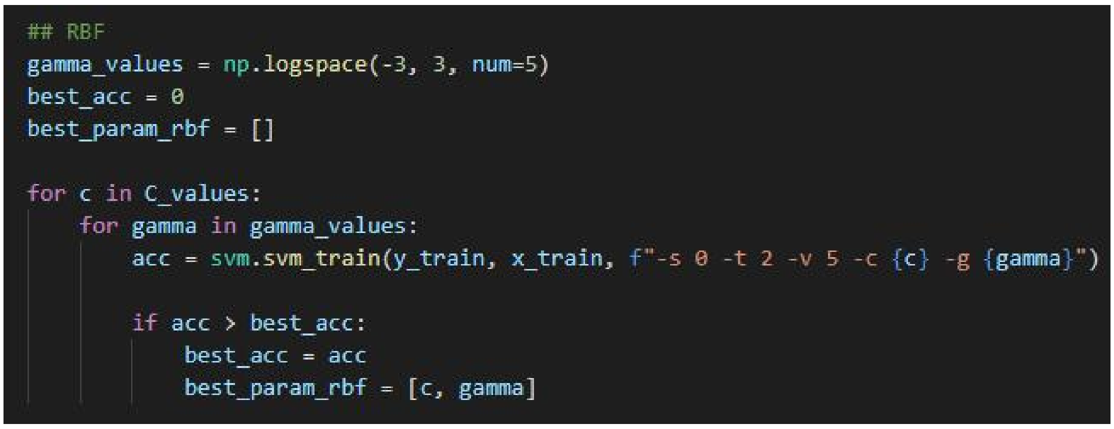
```
C_values = [0.01 * 10 ** i for i in range(5)]
best_acc = 0
best_param_linear = 1

## Linear
for c in C_values:
    acc = svm.svm_train(y_train, x_train, f"-s 0 -t 0 -v 5 -c {c}")

    if acc > best_acc:
        best_acc = acc
        best_param_linear = c
```

This code finds the best parameter C (of slack) for an SVM with linear kernel. The accuracy of the model using each predetermined parameter C is calculated using 5-fold cross validation. The C that gives the highest accuracy is chosen.



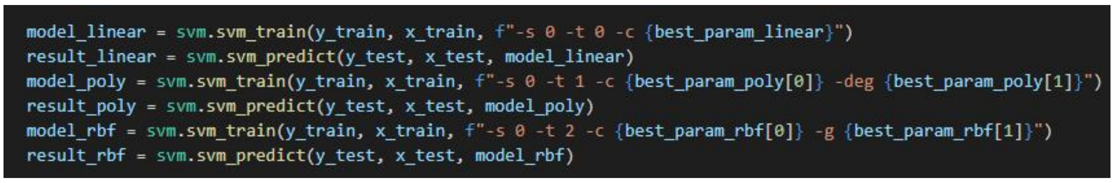
```python
## Poly
degree_values = [i + 1 for i in range(3)]
best_acc = 0
best_param_poly = []

for c in C_values:
    for deg in degree_values:
        acc = svm.svm_train(y_train, x_train, f"-s 0 -t 1 -v 5 -c {c} -d {deg}")

        if acc > best_acc:
            best_acc = acc
            best_param_poly = [c, deg]
```

Similar to above, this code finds the best parameter C and degree (of polynomial kernel) for an SVM with polynomial kernel using grid search.



```python
## RBF
gamma_values = np.logspace(-3, 3, num=5)
best_acc = 0
best_param_rbf = []

for c in C_values:
    for gamma in gamma_values:
        acc = svm.svm_train(y_train, x_train, f"-s 0 -t 2 -v 5 -c {c} -g {gamma}")

        if acc > best_acc:
            best_acc = acc
            best_param_rbf = [c, gamma]
```

Similar to above, this code finds the best parameter C and gamma (of RBF kernel) for an SVM with RBF kernel using grid search.

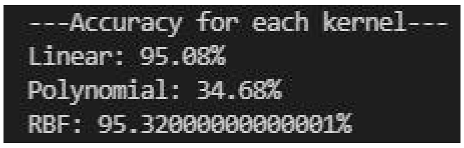


```python
model_linear = svm.svm_train(y_train, x_train, f"-s 0 -t 0 -c {best_param_linear}")
result_linear = svm.svm_predict(y_test, x_test, model_linear)
model_poly = svm.svm_train(y_train, x_train, f"-s 0 -t 1 -c {best_param_poly[0]} -deg {best_param_poly[1]}")
result_poly = svm.svm_predict(y_test, x_test, model_poly)
model_rbf = svm.svm_train(y_train, x_train, f"-s 0 -t 2 -c {best_param_rbf[0]} -g {best_param_rbf[1]}")
result_rbf = svm.svm_predict(y_test, x_test, model_rbf)
```

This code trains and tests our model using the optimal parameters for linear kernel, polynomial kernel, and RBF kernel.

b. Experiment settings and results

Part 1



```
---Accuracy for each kernel---
Linear: 95.08%
Polynomial: 34.68%
RBF: 95.32000000000001%
```

These are the accuracy of each model using default LIBSVM parameters.

Part 2

These are the best parameters for each model that I've found using grid search.

These are the accuracy of each model's classification of the test data using the best parameters.

Grid search takes a long time to run, didn't manage to capture the result.

c. Observations and discussion

Linear and RBF kernel gives the highest accuracy compared to RBF kernel.