Machine Learning Homework 5 Report

陳以瑄 (ID: 313551047)

1 Gaussian Proccess

1.1 Code

1.1.1 Task 1

1) Prepare data

Loading the training set from 'input.data' into the NumPy arrays 'X_train' and 'Y_train', and generating 1000 evenly spaced data points over the range [-60, 60] for the test set.

```
def LoadData(path):
    X = []
    Y = []
    with open(path,'r')as file:
        for data in file.readlines():
            xy = data.split()
            X.append(float(xy[0]))
            Y.append(float(xy[1]))
    X = np.array(X).reshape(-1, 1)
    Y = np.array(Y).reshape(-1, 1)
    return X,Y

X_train,Y_train = LoadData("./data/input.data")
train_num = len(X_train)

test_num = 1000
X_test = np.linspace(-60, 60, test_num).reshape(-1, 1)
```

2) Rational quadratic kernel

In this task, we use the rational quadratic kernel to compute the similarities between different data points.The formula is given as:

$$RationalQuadraticKernel: k(x_i, x_j) = \sigma^2(1 + \frac{(x_i - x_j)^2}{2\alpha l^2})^{-\alpha}$$

where σ, the signal variance, determines the overall amplitude of the kernel; l, the length scale, controls how quickly correlations between data points decay; and α, the shape parameter, adjusts the flexibility.

For the initial parameter set, I set σ=1, l=1, and α=1.

```
sigma = 1
length = 1
alpha = 1
```

```
def RQK(xi,xj,sigma, l,a):
    return (sigma**2)*(1+(xi-xj)**2/(2*a*l**2))**(-a)
```

3) Covariance matrix

The formula of covariance matrix of data is:

$$C(x_n, x_m) = k(x_n, x_m) + \beta^{-1}\delta_{n,m}$$

```
def Cov(X,beta, sigma, length, alpha):
    N = len(X)
    C = np.zeros((N,N))
    for n in range(N):
        for m in range(N):
            C[n][m] = RQK(X[n],X[m], sigma, length, alpha)
            if n==m:
                C[n][m]+=1/beta
    return C
```

4) Gaussian process

After calculating the training set covariance matrix (C), we can make predictions. For a new testing data point, we compute the kernel function values between the test point and each training data point. Using these values, we calculate the mean and variance of the predictive distribution using the following formulas:

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

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

```
def Predict(X_train,Y_train,X_test, train_num, test_num,beta, sigma, length, alpha,C):
    mean = np.zeros(test_num)
    var = np.zeros(test_num)

    for m in range (test_num):
        # Calculate k(x,x*)
        Km = np.zeros((train_num,1))
        for n in range(train_num):
            Km[n][0] = RQK(X_train[n],X_test[m], sigma, length, alpha)

        # calculate mean and var
        # mean = k(x,x*)T C-1 y
        mean[m] = mul(mul(Km.T, inv(C)),Y_train)
        # var = k(x*,x*) + B-1 - k(x,x*)T C-1 k(x,x*)
        K = RQK(X_test[n],X_test[n], sigma, length, alpha)
        var[m] = K + 1/beta -  mul(mul(Km.T, inv(C)),Km)
    return mean, var
```

5) Visualization

```
def Viz(X,Y,mean,var, test_num, sigma, l, a):
    fig = plt.figure()
    x_range = np.linspace(-60, 60, test_num)
    lower_bound = mean - 1.96 * np.sqrt(var)
    upper_bound = mean + 1.96 * np.sqrt(var)

    plt.figure(figsize=(10, 6))
    plt.scatter(X, Y, color="blue", label="Training Data Points", alpha=0.7)
    plt.plot(x_range, mean, color="red", label="Prediction Mean", linewidth=2)
    plt.fill_between(x_range, lower_bound, upper_bound, color="red", alpha=0.3,
                     label="95% Confidence Interval")

    plt.xlim(-60, 60)
    plt.title(f"sigma: {sigma:.4f}, length scale: {l:.4f}, alpha: {a:.4f})", fontsize=14)
    plt.legend(loc='upper left', bbox_to_anchor=(1, 1))
    plt.show()
```

1.1.2 Task 2

1) Negative marginal log-likelihood

The negative marginal log-likelihood is calculated using the following formulas:

$$likelihood = \frac{1}{|2\pi|^{\frac{N}{2}}} \frac{1}{|C_\theta|^{\frac{1}{2}}} e^{-\frac{1}{2} y^T C_\theta^{-1} y}$$

$$loglik = -\frac{N}{2} ln(2\pi) - \frac{1}{2} ln(C_\theta) - \frac{1}{2} y^T C_\theta^{-1} y$$

$$negloglik = \frac{N}{2} ln(2\pi) + \frac{1}{2} ln(|C_\theta|) + \frac{1}{2} y^T C_\theta^{-1} y$$

```
def NegLogLik(theta, X, Y, beta, N):
    theta = theta.reshape(len(theta), 1)
    sigma = theta[0]
    length = theta[1]
    alpha = theta[2]

    C_theta = Cov(X,beta, sigma, length, alpha)

    NLL = N* np.log(2*np.pi)/2
    NLL += np.log(np.linalg.det(C_theta))/2
    NLL += mul(mul(Y.T,inv(C_theta)),Y)/2

    return float(NLL[0])
```

2) Optimize the parameter

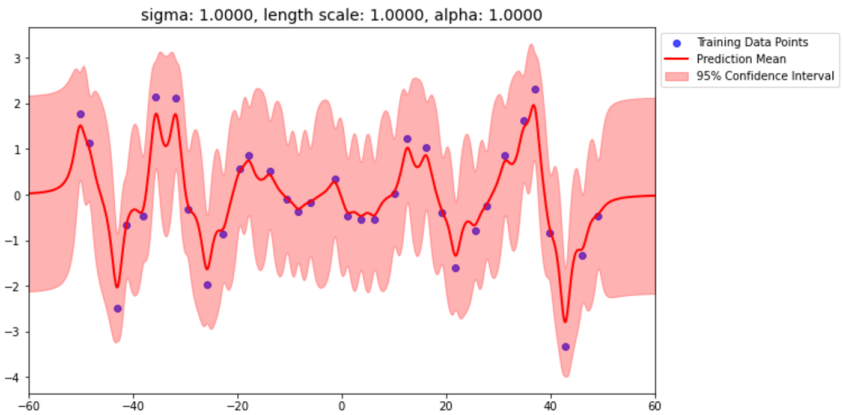
I use scipy.optimize.minimize to find the optimal parameters (σ, l, α) that minimize the negative marginal log-likelihood value. The initial guesses for the parameters are σ=1, l=1, α=1. The function NegLogLik is used as the objective function, with additional arguments.

```
from scipy.optimize import minimize
Opt = minimize(NegLogLik, [sigma,length,alpha], args = (X_train, Y_train, beta, train_num))
sigma = Opt.x[0]
length = Opt.x[1]
alpha = Opt.x[2]
```

1.2 Experiments

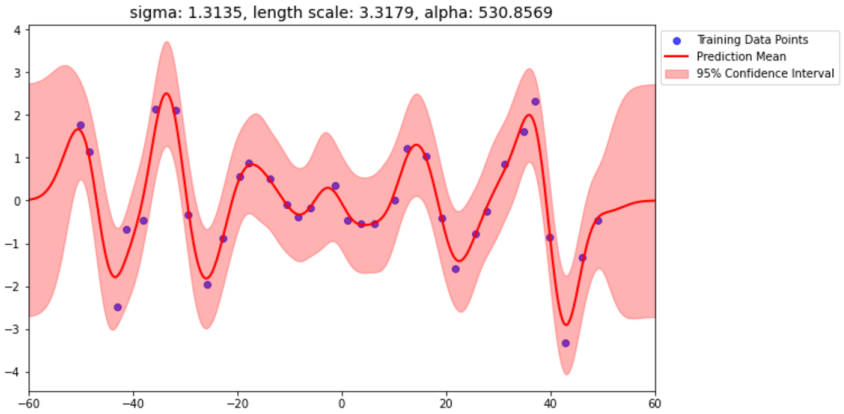
1.2.1 Task 1

The result of task 1, with σ=1, l=1, α=1:



1.2.2 Task 2

The optimal parameter I found are: σ=1.3135, l=3.3179, α=530.8569.



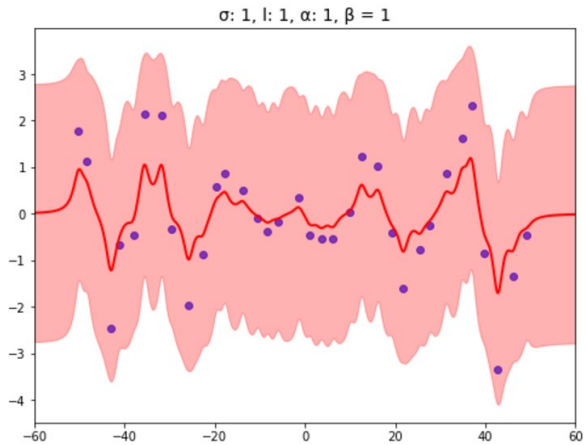
1.3 Observation and Discussion

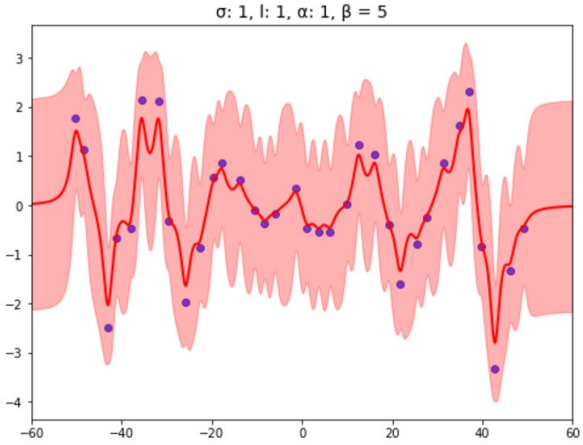
1) Comparing the result of two tasks

The initial model has wider confidence intervals, especially in regions with sparse training data, meaning that it did not fully capture the underlying data structure. The optimized model has smaller and smoother confidence intervals, with a prediction mean that fits better with the training data points.

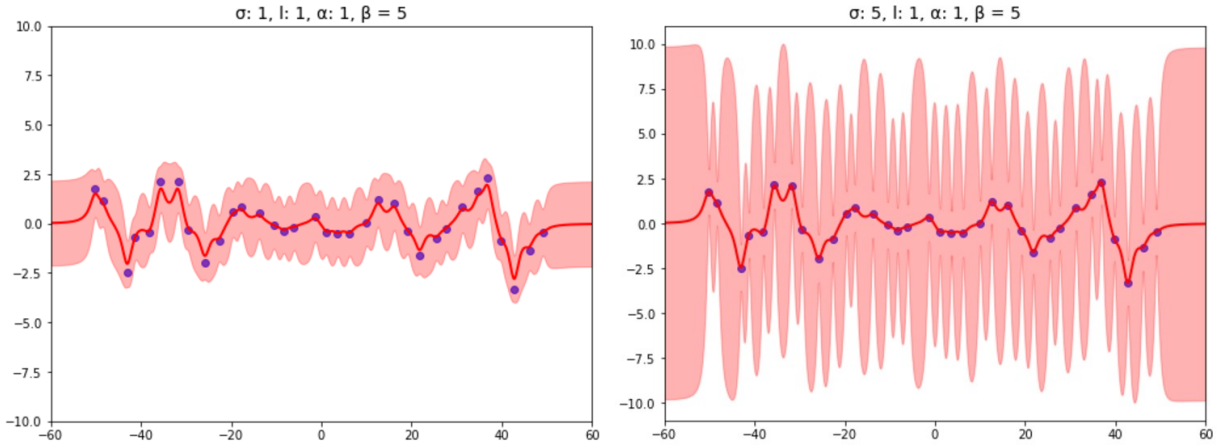
2) Impact of β:

From the figure below, we can see that when β is lower, which means the variance of the random noise is larger, the prediction mean is farther from the training data points, and the confidence interval is wider.

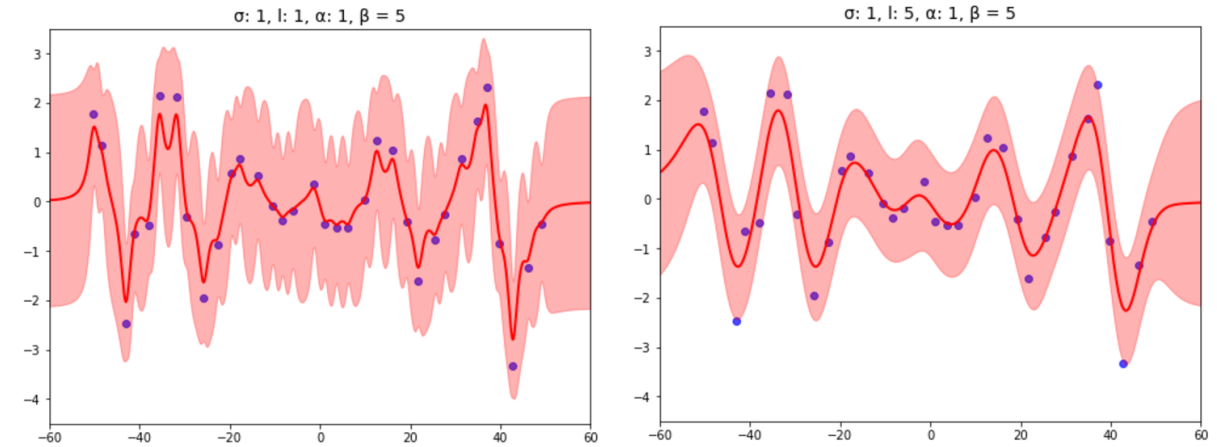




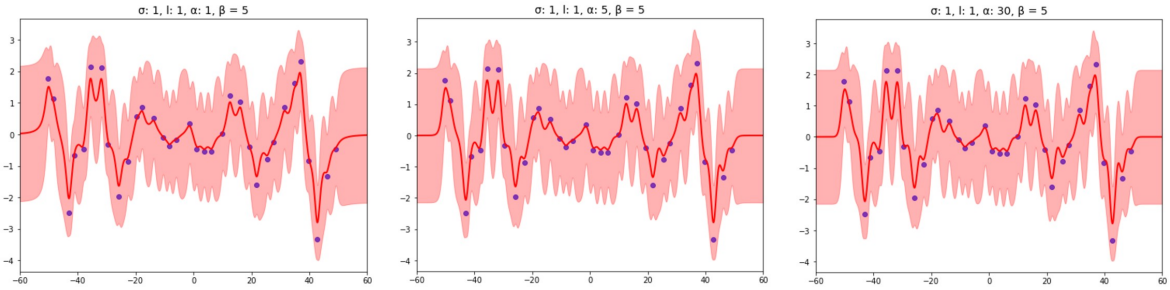
3) Impact of σ:
From the figure below, we can see that when σ is higher, the confidence interval is wider. This is because, with a higher σ , the kernel function values become larger for all data points, meaning that the influence of each training point extends further. As a result, the model becomes more uncertain about the predictions, leading to greater variability and a wider confidence interval.



4) Impact of l:
From the figure below, we can see that when the length scale is higher, the prediction mean and the confidence interval become smoother. This is because a larger length scale means that the kernel function assigns greater weights to distant data points, causing correlations between data points to decay more slowly. As a result, each training point has a broader influence on the predictions, leading to a smoother prediction mean and a more consistent, smoother confidence interval.



5) Impact of α:
From the figure below, we can see that the value of alpha seems to have no impact on the result.



2 SVM on MNIST

2.1 Code

2.1.1 Task 1

1) Prepare data
Loading the training and testing datasets from the given CSV files into the corresponding NumPy arrays.

```
def LoadData(X_path, Y_path):
    X = np.loadtxt(X_path, delimiter=',')
    Y = np.loadtxt(Y_path, delimiter=',')
    return X, Y
```

```
X_train, Y_train = LoadData("./data/X_train.csv", "./data/Y_train.csv")
X_test, Y_test = LoadData("./data/X_test.csv", "./data/Y_test.csv")
```

2) Training on different kernel and testing
First, I use libsvm's svm_train function to train the SVM on the training set. The -t flag specifies the kernel type, where 0 corresponds to linear kernel, 1 to polynomial kernel, and 2 to RBF kernel. Then, I use the svm_predict function to make predictions on the test set and obtain the accuracy.

```
# task 1 different kernrl
from libsvm.svmutil import *
kernels = {'linear': 0, 'polynomial': 1, 'RBF': 2}

for key, value in kernels.items():
    print(f"kernel: {key}", end = ", ")
    model = svm_train(Y_train, X_train, f"-t {value}")
    pred = svm_predict(Y_test, X_test, model)
```

2.1.2 Task 2

1) Grid search to find optimal parameters

Different kernels require different arguments, as shown in the table below:

Argument	Usage	Default	Grid Search	Linear	Polynomial	RBF
-c	Cost for C_SVC	1	[0.001,0.01,0.1,1,10]	v	v	v
-g	gamma of kernel	1/(28*28)	[0.001,0.1,1]		v	v
-d	degree of kernel	3	[2,3,4]		v	
-r	coefficient 0 of kernel	0	[2,4,10]		v	
-v	n-fold cross validation		fixed at 5	v	v	v

I applied both the default settings and the grid search values specific to each kernel. For each configuration, I performed 5-fold cross-validation to evaluate the performance.

```
def GridSearch(X,Y):
    D = [2, 3, 4]
    Gamma = [1e-2, 1e-1, 1]
    Coef0 = [2, 4, 10]
    C = [0.001, 0.01, 0.1, 1, 10]
    v = 5

    for key, kernel in kernels.items():
        print(f"\nkernel: {key}")
        print("Default:", end = " ")
        res = svm_train(Y, X, f"-t {kernel} -v {v}")
        if kernel==0:
            for c in C:
                arg = f"-t {kernel} -v {v} -c {c}"
                print(f"-c {c:<5.3f}", end = " ")
                res = svm_train(Y, X, arg)
        elif kernel==1:
            for c in C:
                for d in D:
                    for gamma in Gamma:
                        for coef0 in Coef0:
                            arg = f"-t {kernel} -v {v} -c {c} -d {d} -g {gamma} -r {coef0} "
                            print(f"-c {c:<5.3f} -d {d:<1d} -g {gamma:<3.2f} -r {coef0:<2d} ", end = " ")
                            res = svm_train(Y, X, arg)
        else:
            for c in C:
                for gamma in Gamma:
                    arg = f"-t {kernel} -v {v} -c {c} -g {gamma}"
                    print(f"-c {c:<5.3f} -g {gamma:<3.2f}", end = " ")
                    res = svm_train(Y, X, arg)
```

2) Performance of optimal parameters on testing set

I used the optimal parameters obtained from the grid search to train the best model for each kernel type. After training, I made predictions on the test set to evaluate the model's performance.

```
kernels = {'linear': 0, 'polynomial': 1, 'RBF': 2}
best_opt = {'linear': "-c 0.01",
            'polynomial': "-c 0.1 -d 2 -g 1 -r 10",
            'RBF': "-c 10 -g 0.01"}

for key, value in kernels.items():
    print(f"kernel: {key}", end = ", ")
    print(f"best opt: -t {value} {best_opt[key]}")
    model = svm_train(Y_train, X_train, f"-t {value} "+best_opt[key])
    pred = svm_predict(Y_test, X_test, model)
```

2.1.3 Task 3

1) Kernels

The formula of linear kernels and RBF kernel is:

Kernel	K_{ij}	Matrix K
Linear	$K_{ij} = x_i \cdot x_j$	$K = XX^{'T}$
RBF	$K_{ij} = \exp^{-\gamma\|x_i - x_j\|^2}$	$K = e^{-\gamma D}, where D = \|x_i\|^2 + \|x_j\|^2 - 2(x_i \cdot x_j)$

```
def Linear_K(X1,X2):
    return mul(X1,X2.T)
```

```
def RBF_K(X1,X2, gamma):
    square_X1 = np.sum(X1 ** 2, axis=1).reshape(-1, 1)
    square_X2 = np.sum(X2 ** 2, axis=1)
    d = square_X1 - 2 * mul(X1, X2.T) + square_X2
    k = np.exp(-gamma* d)
    return k
```

2) Training on precompute kernel

I chose the optimal gamma value of 0.01 found in Task 2 and calculated the kernel values for both the training and testing sets by combining Linear and RBF kernels. Since precomputed kernels require the first column of the training matrix to be the ID of each instance, I used np.column_stack to append sequential indices as the first column to both K_train and K_test. Then, I used the precomputed kernel for training the SVM model (flag -t 4) and applied the model to the testing set to evaluate its performance.

```
gamma = 0.01
K_train = Linear_K(X_train, X_train) + RBF_K(X_train, X_train, gamma)
K_test = Linear_K(X_train,X_test).T + RBF_K(X_train,X_test, gamma).T

# add index for every Ki
train_idx = np.arange(1,K_train.shape[0]+1)
K_train = np.column_stack((train_idx, K_train))

test_idx = np.arange(1,K_test.shape[0]+1)
K_test = np.column_stack((test_idx, K_test))
```

```
# -t 4: precomputed kernel
model = svm_train(Y_train, K_train, "-t 4")
pred = svm_predict(Y_test, K_test, model)
```

2.2 Experiments

2.2.1 Task 1

Kernel	Accuracy
Linear	95.08%
Polynomial	34.68%
RBF	95.32%

2.2.2 Task 2

Kernel	Best parameters	Validation Accuracy	Testing Accuracy
Linear	-c 0.01	97.12%	95.96%
Polynomial	-c 0.1 -d 2 -g 1 -r 10	98.22%	97.84%
RBF	-c 10 -g 0.01	98.14%	98.20%

2.2.3 Task 3

Kernel	Parameter	Accuracy
Linear + RBF	gamma = 0.01	95.32%

2.3 Observation and Discussion

1) The degree in polynomial kernel
The optimal degree for the polynomial kernel is 2, which significantly improves performance compared to the default degree of 3 (97.84% $>$ 34.68%). This indicates that a higher degree makes the model overly complex, leading to overfitting. Reducing the degree to 2 simplifies the model and enhances its ability to generalize.

2) Best Kernel
The RBF kernel achieves the best results, suggesting that the dataset exhibits complex, non-linear relationships that are effectively captured by the RBF kernel.

3) Combining Kernels
Combining the linear and RBF kernels does not offer any significant improvement over using the RBF kernel alone with optimal parameters. This suggests that the RBF kernel already captures the critical patterns in the data, making the linear component redundant.