ML-HW5

資工所博士班
周芝妤 0886004

I. Gaussian Process

■ a. code with detailed explanations (20%)

Part 1: Apply Gaussian Process Regression to predict the distribution of f and visualize the result.

✓ Define the kernel function:

Use rational quadratic kernel here to find similarity between data points, the equation is shown as bellow:

$$k(x_i, x_j) = \left(1 + \frac{d(x_i, x_j)^2}{2\alpha l^2}\right)^{-\alpha}$$

Code:

```
def kernel(x1, x2, alpha, length):
    '''
    Use rational quadratic kernel: k(x1,x2)=(1+(x1-x2)^2/2αℓ^2)^(-α)
    input data x1(n), x2(m) array
    return (n,m) array
    '''
    distance = np.power(x1.reshape(-1,1)-x2.reshape(1,-1),2.0)
    K = np.power(1 + distance/2*alpha*length**2, -alpha)

    return K
```

✓ Define predict function first and use it to get the mean and variance of sampling data points in np.linspace(-60, 60, num=1000), then plot the figure (shown in part b).

Code:

```
def predict(line,x,y,K,beta,alpha=1,length=1):
    '''
    vectorize calculate k_x_xstar
    line: sampling in linspace(-60,60)
    X:  (n) ndarray
    y: (n) ndarray
    K: (n,n) ndarray
    beta:
    return: (len(x_line),1) ndarray, (len(x_line),len(x_line)) ndarray
    '''
    k_x_xstar=kernel(x,line,alpha=1,length=1)
    k_xstar_xstar=kernel(line,line,alpha=1,length=1)
    means=k_x_xstar.T @ np.linalg.inv(K) @ y.reshape(-1,1)
    variance=k_xstar_xstar+(1/beta)*np.identity(len(k_xstar_xstar))-k_x_xstar.T @ np.linalg.inv(K) @ k_x_xstar

    return means,variance
```

```
# main

def GP(x, y, theta, beta, alpha=1, length=1):
    # Covariance matrix = kernel + whilte noise
    C = kernel(x, x, 1, 1) + 1/beta*theta

    # Get means and variance from range[-60,60]
    line = np.linspace(-60, 60, num=1000)
    M, V = predict(line, x, y, C, beta, alpha, length)
    M_pre = M.reshape(-1)
    V_pre = np.sqrt(np.diag(V))

    # Plot it out
    plt.plot(x, y, 'bo')
    plt.plot(line, M_pre, 'r-')
    plt.fill_between(line, M_pre+V_pre*2, M_pre-V_pre*2, facecolor='salmon')
    plt.xlim(-60, 60)
    plt.show()
```

Part 2: Optimize the kernel parameters by minimizing negative marginal log-likelihood.

✓ Define the objective function:

Follow the equation bellow to write the code.

$$\log p(\mathbf{y}|\mathbf{X}) = \log \mathcal{N}(\mathbf{y}|\mathbf{0}, \mathbf{K}_y) = -\frac{1}{2}\mathbf{y}^T\mathbf{K}_y^{-1}\mathbf{y} - \frac{1}{2}\log|\mathbf{K}_y| - \frac{N}{2}\log(2\pi)$$

Code:

```
# Define objective function

def objective(x,y,beta):
    '''
    x:  (n) array
    y:  (n) array
    return: objective function
    '''
    def objective_return(the):
        C=kernel(x,x,alpha=the[0], length=the[1]) +(1/beta)*np.identity(len(x))
        L=np.linalg.cholesky(C)
        log_likelihood = 0.5*y.reshape(1,-1) @ np.linalg.inv(C) @ y.reshape(-1,1) +
                                np.sum(np.log(np.diag(L))) + 0.5*len(x)*np.log(2*np.pi)
        return log_likelihood
    return objective_return
```

✓ Use scipy.optimize to minimize objective function.

Use [1e-3,1e-2,1e-1,0,1e1,1e2,1e3] as initial value to run minimize_objective and try to find the optimal alpha and length_scale to minimize log p(y|X).

Code:

```
# main
def minimize_objective(x, y, beta,theta ,obj_value, init_value):
    for init in init_value:
        for init_scale in init_value:
            mini_obj = minimize(objective(x, y, beta), x0=[init, init_scale], bounds=((1e-5, 1e5), (1e-5, 1e5)))
            if mini_obj.fun < obj_value:
                obj_value = mini_obj.fun
                alpha_optimize, length_scale =mini_obj.x

    # Covariance matrix = kernel + whilte noise
    C = kernel(x, x,alpha_optimize,length_scale) + 1/beta*theta

    # Get means and variance from range[-60,60]
    line = np.linspace(-60, 60, num=1000)
    M, V = predict(line, x, y, C, beta,alpha_optimize,length_scale)
    M_pre = M.reshape(-1)
    V_pre = np.sqrt(np.diag(V))

    # Plot it out
    plt.plot(x, y, 'bo')
    plt.plot(line, M_pre, 'r-')
    plt.fill_between(line, M_pre+V_pre*2, M_pre-V_pre*2, facecolor='salmon')
    plt.xlim(-60, 60)
    plt.show()
```

```
beta =5
x ,y =load_data()
obj_value=1e9
init_value=[1e-3,1e-2,1e-1,0,1e1,1e2,1e3]
theta = np.eye(len(x))
minimize_objective(x, y, beta,theta ,obj_value, init_value)
```

■ b. experiments settings and results (20%)

Part 1:

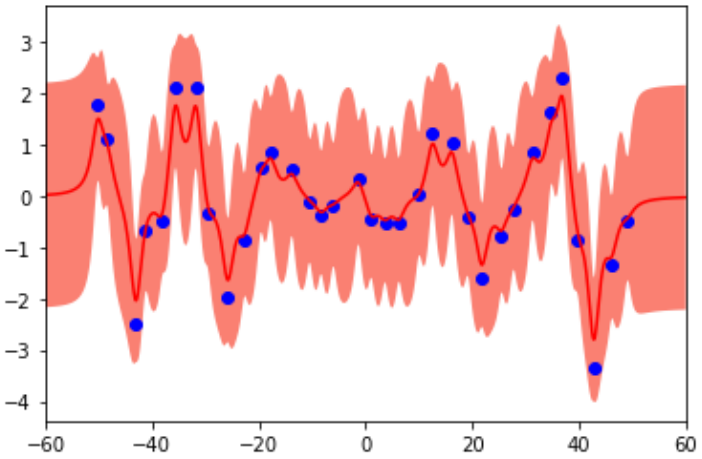
Parameters:

beta=5

alpha=1

length=1

theta= np.eye(len(x))



Part 2:

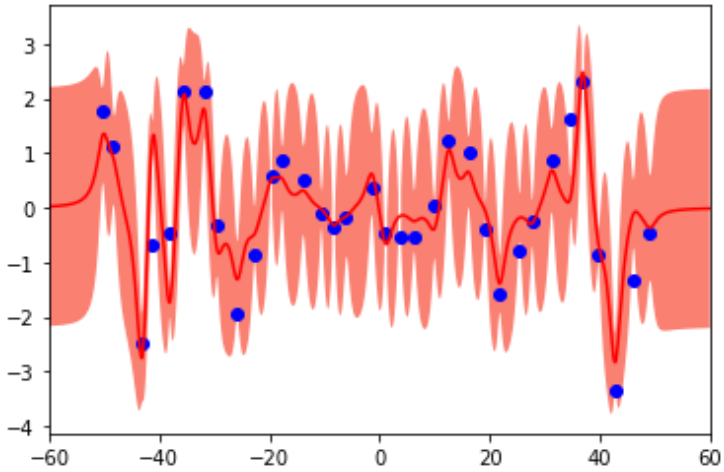
Parameters:

beta =5

obj_value=1e9

init_value=[1e-3,1e-2,1e-1,0,1e1,1e2,1e3]

theta = np.eye(len(x))



■ c. observations and discussion (10%)

By visualizing the data and 95% confidence interval, I observed that the 95% confidence interval became clearer and more unsmooth after minimizing negative marginal log-likelihood. I also observed at the front part and last part, the range of 95% confidence interval is quite large, cause by the lack of data, and it matched our expectation (If there is no data, it will be harder to predict the outcome correctly.)

II. SVM on MNIST dataset

■ a. code with detailed explanations (20%)

Part 1: Use different kernel functions (linear, polynomial, and RBF kernels).

✓ Try different kernels:

Import libsvm.svmutil libarary, reference : https://www.csie.ntu.edu.tw/~cjlin/libsvm/

Use **t kernel_type** : to set type of kernel function (default 2)

0 -- linear: u'*v

1 -- polynomial: (gamma*u'*v + coef0)^degree

2 -- radial basis function: exp(-gamma*|u-v|^2)

In this assignment we only use '-t 0', '-t 1', '-t 2' as linear, polynomial, and RBF.

The parameters here I just use the default settings.

Code:

```
from libsvm.svmutil import *

# linear
print('Linear kernel:')
options = '-t 0'
linear = svm_train(y_train, x_train, options)
l_label, l_acc, l_vals = svm_predict(y_test, x_test, linear)
#print('The accuracy of linear kernel:', l_acc[0])

# polynomial
print('Polynomial kernel:')
options = '-t 1'
poly = svm_train(y_train, x_train, options)
p_label, p_acc, p_vals = svm_predict(y_test, x_test, poly)
#print('The accuracy of polynomial kernel:', p_acc[0])

# RBF
print('RBF kernel:')
options = '-t 2'
rbf = svm_train(y_train, x_train, options)
r_label, r_acc, r_vals = svm_predict(y_test, x_test, rbf)
#print('The accuracy of RBF kernel:', r_acc[0])
```

```
Linear kernel:
Accuracy = 95.08% (2377/2500) (classification)
Polynomial kernel:
Accuracy = 34.68% (867/2500) (classification)
RBF kernel:
Accuracy = 95.32% (2383/2500) (classification)
```

Result showed as above.

Accuracy of linear: 95.08%

Accuracy of polynomial: 34.68%

Accuracy of RBF kernels: 95.32%

Part 2: Grid search.

✓ Define grid search function:

Linear & polynomial kernel:

Try every parameter C and record all accuracy data.

My setting C is [0.0001, 0.001, 0.01, 0.1, 1, 10, 100, 1000].

Code:

```
C =[0.0001, 0.001, 0.01, 0.1, 1, 10, 100, 1000]

# grid sreach function for linear

def grid_search_linear(C ,x_train, y_train):
    grid = np.zeros(len(C))
    for i in range(len(C)):
        options = '-t 0 -v 3 -c {}'.format(C[i])
        acc = svm_train(y_train, x_train, options)
        grid[i] = acc
    return grid

# grid sreach function for polynomial

def grid_search_polynomial(C,x_train, y_train):
    grid = np.zeros(len(C))
    for i in range(len(C)):
        options = '-t 1 -v 3 -c {}'.format(C[i])
        acc = svm_train(y_train, x_train, options)
        grid[i] = acc
    return grid
```

```
linear_grid = grid_search_linear(C ,x_train, y_train)
poly_grid = grid_search_polynomial(C,x_train, y_train)
```

Then run the code and plot both kernels results.

RBF kernel:

Try every parameter pairs (gamma, C) and record the accuracy in every pair.

My setting of gamma and C are both [0.0001, 0.001, 0.01, 0.1, 1, 10, 100, 1000].

Code:

```
# grid sreach function

def grid_search(C, gamma, x_train, y_train):
    grid = np.zeros((len(C), len(gamma)))
    for i in range(len(C)):
        for j in range(len(gamma)):
            options = '-t 2 -v 3 -c {} -g {}'.format(C[i], gamma[j])
            acc = svm_train(y_train, x_train, options)
            grid[i, j] = acc
    return grid
```

```
# set C and gamma

gamma= C =[0.0001, 0.001, 0.01, 0.1, 1, 10, 100, 1000]
```

```
# main

final_list = grid_search(C, gamma, x_train, y_train, x_test, y_test)
```

Part 3: Linear kernel + RBF kernel

✓ Use scipy.spatial.distance to calculate the (Euclidean distance)^2 between two data points.

Here I used linear kernel combine with RBF kernel and create new kernel, and I used 0.01 as gamma. (cause 0.01 is the best gamma we get in grid search.)

Code:

```
# Use scipy.spatial.distance
# Set new kernel: liner + rbf kernel

from scipy.spatial.distance import cdist

def pre_kernel(x_data, x_data2, gamma):
    kernel_linear = x_data@x_data2.T
    kernel_rbf = np.exp(-gamma*cdist(x_data, x_data2, 'sqeuclidean'))
    kernel = kernel_linear + kernel_rbf
    kernel = np.hstack((np.arange(1, len(x_data)+1).reshape(-1, 1), kernel))
    return kernel
```

```
# main
# Use gamma = 0.01
kernel_train = pre_kernel(x_train, x_train, 0.01)
problem = svm_problem(y_train, kernel_train, isKernel=True)
options = svm_parameter('-t 4')
model = svm_train(problem, options)
kernel_test = pre_kernel(x_test, x_train, 0.01)
label, acc, vals = svm_predict(y_test, kernel_test, model)
print('linear kernel + RBF kernel accuracy: {:.2f}%'.format(acc[0]))
```

```
Accuracy = 95.32% (2383/2500) (classification)
linear kernel + RBF kernel accuracy: 95.32%
```

Result:

Accuracy = 95.32% (2383/2500) (classification)

linear kernel + RBF kernel accuracy: 95.32%

■ b. experiments settings and results (20%)

Part 1:

The parameters here I just use the default settings.

Result for first part:

Linear kernel:

Accuracy = 95.08% (2377/2500) (classification)

Polynomial kernel:

Accuracy = 34.68% (867/2500) (classification)

RBF kernel:

Accuracy = 95.32% (2383/2500) (classification)

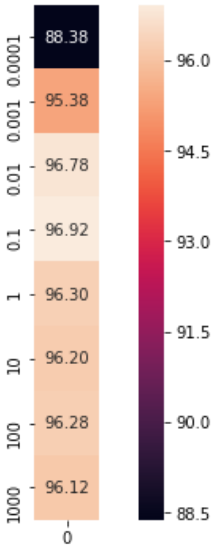
Part 2:

Linear & polynomial kernel:

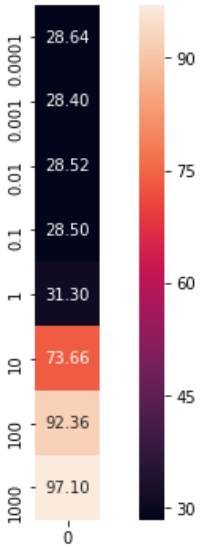
Try every parameter C and record all accuracy data.

My heatmap shown as bellow:

Both setting of C is [0.0001, 0.001, 0.01, 0.1, 1, 10, 100, 1000].



(linear)



(polynomial)

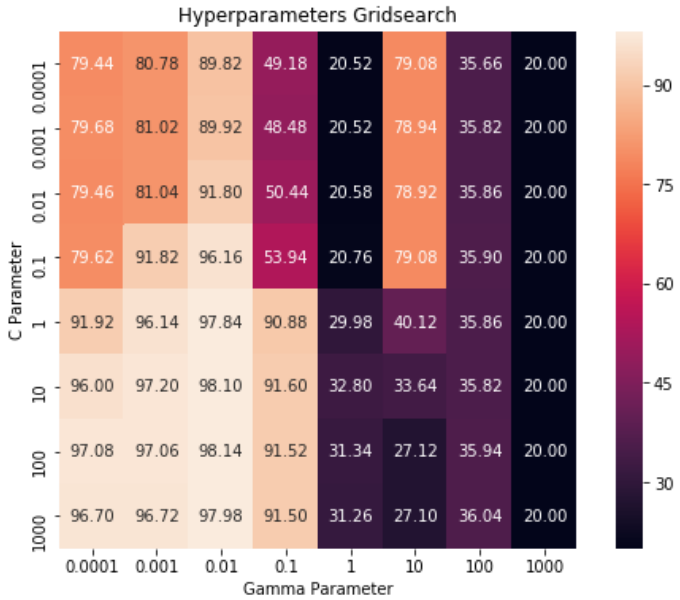
We can find the best performance of **linear kernel** is **96.92%** with **C = 0.1**.

We can find the best performance of **polynomial kernel** is **97.1%** with **C = 1000**.

RBF kernel:

My heatmap shown as bellow:

My setting of gamma and C are both [0.0001, 0.001, 0.01, 0.1, 1, 10, 100, 1000].



We can find the best performance **98.14%** with **gamma = 0.01, C = 100.**

Part 3:

I used 0.01 as gamma, cause 0.01 is the best gamma we get in grid search.

Result:

My linear kernel + RBF get 95.32% accuracy.

Accuracy = 95.32% (2383/2500) (classification)

■ c. observations and discussion (10%)

It's interesting that at first part, the polynomial kernel (Accuracy = 34.68%) get the lowest accuracy, so I organize the pros and cons of these linear and polynomial kernels and hope to figure it out.

Linear kernel: Simple dot product of data.

Polynomial kernel: $K(x, y) = (x^\top y + c)^d$

	Linear kernel	Polynomial kernel
Pros	It's simple and safe, and easier to design quadratic programming solver. Easy to figure out how does the SVM separate data. Always use linear first!	Less restricted than linear. We can set degree "d" by ourselves.
Cons	It can only use in linear divisible data.	Hard to calculate if we get large "d". More parameters to choose.

The polynomial kernel gets better performance with smaller degree "d".

So…maybe in this task, degree is too large? And linear is more efficient sometimes.