ML_HW5 Report

I. Gaussian Process

1.

- Input data

```
X = []
Y = []
f = open("input.data")
line = f.readline()
while line:
    x,y = line.split(" ")
    X.append(float(x))
    Y.append(float(y))
    line = f.readline()
f.close()
```

- Define kernel function

rational quadratic kernel : $k_{\mathrm{RQ}}(x, x') = \sigma^2 \left(1 + \frac{(x - x')^2}{2\alpha\ell^2}\right)^{-\alpha}$

```
def kernel_function(x, y, sigma, a, l):
    kernel = (sigma**2) * ((1+ ((x-y)**2 / (2 * a * l**2)) )**(-a))
    return kernel
```

- Define a function can generate K(x, x), K(x, x*), and K(x*, x*)

```
def compute_cov_matrices(x, x_star, sigma, a , l):

    K = []
    K_star = []
    K_star2 = []
    n = x.shape[0]
    n_star = x_star.shape[0]

    for i in x:
        for j in x:
            K.append(kernel_function(i, j, sigma, a , l))

    K = np.array(K).reshape(n, n)

    for i in x:
        for j in x_star:
            K_star.append(kernel_function(i, j, sigma, a , l))

    K_star = np.array(K_star).reshape(n, n_star)

    for i in x_star:
        for j in x_star:
            K_star2.append(kernel_function(i, j, sigma, a , l))

    K_star2 = np.array(K_star2).reshape(n_star, n_star)

    return K, K_star, K_star2
```

- Define a function to generate the predicted mean and variance.

Use the formula $$\begin{aligned} \mu(\mathbf{x}^*) &= k(\mathbf{x}, \mathbf{x}^*)^\top \mathbf{C}^{-1} \mathbf{y} \\ \sigma^2(\mathbf{x}^*) &= k^* - k(\mathbf{x}, \mathbf{x}^*)^\top \mathbf{C}^{-1} k(\mathbf{x}, \mathbf{x}^*) \\ k^* &= k(\mathbf{x}^*, \mathbf{x}^*) + \beta^{-1} \end{aligned}$$ in the slides.

var = 1/5, given in the reference.

```
def predict(sigma, a, l):

    K, K_star, K_star2 = compute_cov_matrices(np.array(X), x_star, sigma, a, l)
    C = K + var*np.identity(n)
    k_predict  = K_star2 + var*np.identity(K_star2.shape[0])
    var_predict = k_predict - K_star.T @ np.linalg.inv(C) @ K_star
    mean_predict = K_star.T @ np.linalg.inv(C) @ Y
    prediction = np.random.multivariate_normal(mean_predict, var_predict)

    return  mean_predict, var_predict, prediction
```

- Define a function that can visualize the training data point, mean of f and 95% confidence.

```
from matplotlib import cm

def plot_gp(mu, cov, X, X_train=None, Y_train=None, samples = None):
    X = X.ravel()
    mu = mu.ravel()
    uncertainty = 1.96 * np.sqrt(np.diag(cov))

    plt.fill_between(X, mu + uncertainty, mu - uncertainty, alpha=0.1)
    plt.plot(X, mu, label='Mean')
    #plt.plot(X, samples, lw=1, ls='--')
    plt.plot(X_train, Y_train, 'ro')
    plt.show()
```

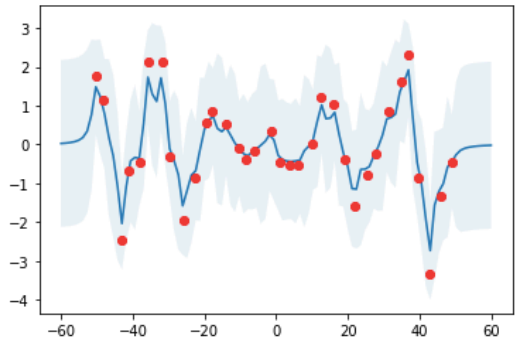
- First, set every parameters in kernel function equal to 1. And visualize the result.

Take 100 points in range(-60,60) to be the testing data.

The red points is training data points, and the blue line is the mean of f.

The light blue shadow is the 95% confidence interval of f.

```
x_star = np.linspace(-60,60,100)
mean_predict, var_predict, prediction = predict(1,1,1)
plot_gp(mean_predict, var_predict, x_star, X_train=np.array(X), Y_train=np.array(Y), samples=prediction)
```



2.

- Define a function to return the negative log likelihood.

  The likelihood is based this formula in the slides. The function just return the value equal to log likelihood multiples -1.

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

```
def likelihood(params):
    sigma, a, l = params[0], params[1], params[2]
    K = [kernel_function(i, j, sigma, a, l) for (i, j) in itertools.product(np.array(X), np.array(X))]
    K = np.array(K).reshape(n, n)
    C = K + var*np.eye(n)
    L = (0.5 * np.log(np.linalg.det(C)) + 0.5 * (np.array(Y).T @ np.linalg.inv(C) @ np.array(Y)) + (n/2)*np.log(2*(np.pi)))

    return L
```

- Optimize parameters by using the scipy.optimize.minimize to find the minimum negative log likelihood.

  There are lots of method can choose, I tried every and found that some of them can't converge, some of them are not fit. Finally I choose "L-BFGS-B".

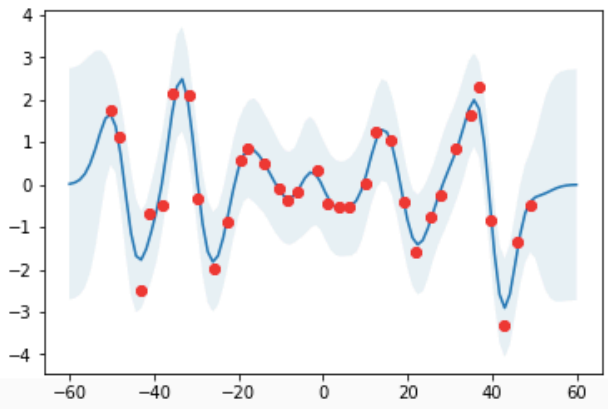
  By Using "L-BFGS-B" method, [sigma = 1.313, a = 305.775, l = 3.317] are the optimized parameters.

```
guess = np.array([1,1,1])
results = minimize(likelihood, guess, method = 'L-BFGS-B', options={"disp": True})
params = results.x
print(params)
```

```
[ 1.3136154  305.7764122    3.31765194]
```

- Draw out the result

```
mean_predict, var_predict, prediction = predict(params[0],params[1],params[2])
plot_gp(mean_predict, var_predict, x_star, X_train=np.array(X), Y_train=np.array(Y), samples=prediction)
```



II. SVM

1.

- Define read_data function, and read data

```
def read_data(filename):

    data = []
    with open(filename) as csvfile:
        rows = csv.reader(csvfile)
        for row in rows:
            data.append(row)

    return np.array(data,dtype='double')
```

```
X_train = read_data('X_train.csv')
X_test = read_data('X_test.csv')
Y_train = read_data('Y_train.csv').reshape(5000)
Y_test = read_data('Y_test.csv').reshape(2500)
```

- Import libsvm

```
import sys
sys.path.append('./libsvm-3.24/python')

import svm
from svmutil import *
```

- Set parameters, train and predict to get result.

  Parameters : linear kernel : '-t 0', poly kernel : '-t 1', rbf kernel : '-t 2'

  Use the default parameters to train can get the performance as below:

```
prob = svm_problem(Y_train, X_train)
param_linear = svm_parameter('-t 0')
param_poly = svm_parameter('-t 1')
param_rbf = svm_parameter('-t 2')
```

```
m_linear = svm_train(prob, param_linear)
res_linear = svm_predict(Y_test, X_test, m_linear)

m_poly = svm_train(prob, param_poly)
res_poly = svm_predict(Y_test, X_test, m_poly)

m_rbf = svm_train(prob, param_rbf)
res_rbf = svm_predict(Y_test, X_test, m_rbf)
```

```
print("linear:")
print(res_linear[1])
print("poly:")
print(res_poly[1])
print("rbf:")
print(res_rbf[1])
```

result (ACC, MSE, SCC):

```
linear:
(95.08, 0.1404, 0.931149802516624)
poly:
(34.68, 2.6212, 0.14887572191533946)
rbf:
(95.32000000000001, 0.1492, 0.9271864783823697)
```

comparison (order in accuracy) :

	ACC(Accuracy)	MSE(mean squared error)	SCC(squared correlation coefficient)
Rbf	**95.32%**	0.1492	0.927
Linear	95.08%	0.1404	0.931
Polynomial	34.68%	2.6212	0.149

- The performance of the polynomial kernel is much worse than others. I guess the reason may be the default parameters are not good enough. I think it's not fair to do the comparison between each others. So, I did a simple grid search to find the better parameters then do the comparison again.

  The search range I set:

  degree:[1, 3, 5], since the default is 3, I pick up a bigger and a smeller one.

  coef0 : [-1, 0, 1], since the default is 1, I pick up a bigger and a smeller one.

  gamma : [1/784, 3/784, 5/784] , since the default is 1/num_features, I pick up bigger ones.

```
DEGREE = [1,3,5]
COEF = [-1,0,1]
GAMMA = [1.0/784.0, 3.0/784.0, 5.0/784.0]

param_linear = svm_parameter('-t 0')
m_linear = svm_train(prob, param_linear)
res_linear = svm_predict(Y_test, X_test, m_linear)

max_poly = None
max_poly_i = None
max_accu = 0

for d in DEGREE:
    for coef in COEF:
        for g in GAMMA:
            param_poly = svm_parameter('-t 1 -d '+ str(d)+ ' -r '+ str(coef)+ ' -g ' + str(g))
            m_poly = svm_train(prob, param_poly)
            res_poly = svm_predict(Y_test, X_test, m_poly)
            if max_accu< res_poly[1][0]:
                max_accu = res_poly[1][0]
                max_poly = res_poly
                max_poly_i = d,coef,g

max_accu = 0
max_rbf = None
max_rbf_i  = 0

for g in GAMMA:
    param_rbf = svm_parameter('-t 2 -g ' + str(g))
    m_rbf = svm_train(prob, param_rbf)
    res_rbf = svm_predict(Y_test, X_test, m_rbf)
    if max_accu< res_rbf[1][0]:
        max_accu = res_rbf[1][0]
        max_rbf = res_rbf
        max_rbf_i = g
```

```
print("linear:")
print(res_linear[1])
print("poly:")
print(max_poly[1])
print("degree:{}, coef0:{}, gamma:{}".format(max_poly_i[0],max_poly_i[1],max_poly_i[2]))
print("rbf:")
print(max_rbf[1])
print("gamma:{}".format(max_rbf_i))
```

result (ACC, MSE, SCC):

```
linear:
(95.08, 0.1404, 0.931149802516624)
poly:
(97.92, 0.056, 0.9721745192307693)
degree:5, coef0:1, gamma:0.00637755102041
rbf:
(96.88, 0.1016, 0.9498554017930114)
gamma:0.00637755102041
```

comparison (order in accuracy):

	Parameters	ACC(Accuracy)	MSE(mean squared error)	SCC(squared correlation coefficient)
Polynomial	degree = 5 coef0 = 1 gamma = 5/784	**97.92%**	0.056	0.972
Rbf	gamma = 5/784	96.88%	0.1016	0.950
Linear	NO	95.08%	0.1404	0.931

2.

- Use libsvm/tools/grid.py to do the grid search with cross validation to optimize the parameters in the rbf kernel.

  The input data format needs to be like <label> <index1>:<value1> <index2>:<value2> … . So, first, I rewrite the data into file ”data”. And then call “find_parameters”, it’s the function in the grid.py. The search range I set both of c (cost) and g(gamma) are $[2^{-5}, 2^{-3}, 2^{-1}, 2^{1}, 2^{3}, 2^{5}]$. Since the default search range of c is $[2^{-5}, 2^{-3},\ldots, 2^{13}, 2^{15}]$, g is $[2^{3}, 2^{1},\ldots, 2^{-13}, 2^{-15}]$, and it costs a lots of time to run. I chose $[2^{-5}, 2^{-3}, 2^{-1}, 2^{1}, 2^{3}, 2^{5}]$, number of the value smaller than 1 is equal to the value bigger than 1, also, it run much faster. And, use 3-fold cross validation to train. The result will output to the “output” file.

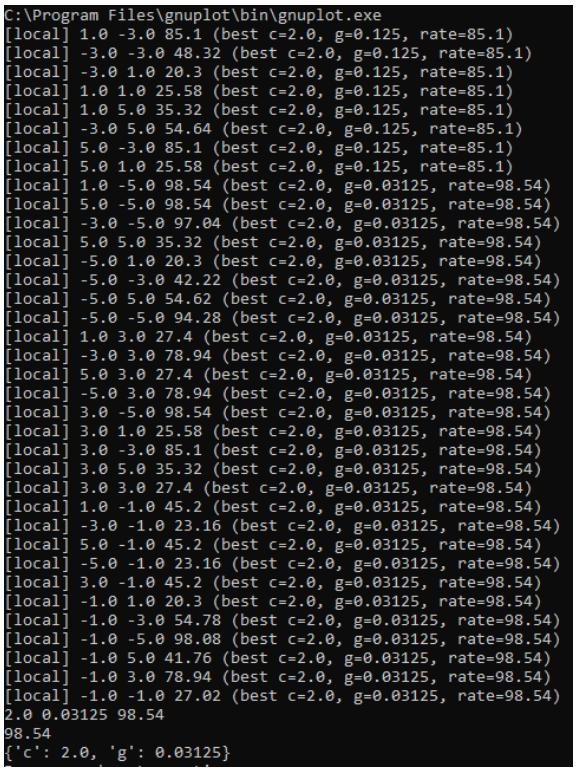
```
sys.path.append('./libsvm-3.24/tools')
from grid import *
```

```
with open('./libsvm-3.24/tools/data','w') as f:
    for i in range(X_train.shape[0]):
        s = str(int(Y_train[i])) + ' '
        for j in range(X_train.shape[1]):
            s = s  + '{}:{} '.format(j+1,X_train[i][j])
        f.write(s + '\n')
f.close()
```

```
rate, param = find_parameters('./libsvm-3.24/tools/data', '-log2c -5,5,2 -log2g -5,5,2 -v 3 -out output')
print(rate)
print(param)
```

result:

The best accuracy appears when c = 2, g = 0.03125, accuracy = 98.54%



3.

- To use user-define kernel in the libsvm, we need calculate K(x,x) first, and add the index(1~n) in the first column to fit the libsvm form.

  scipy.spatial.distance.cdist be used to calculate the pair distance in two matrix.

  Define the “get_kernel” function to generate the combined kernel. The combined kernel is equal to Kernel(linear) + a * Kernel(rbf), a is the weight.

```
from scipy.spatial.distance import cdist

def get_kernel(X, Y, sigma, a):

    K_combine = np.zeros((X.shape[0], Y.shape[0] + 1))

    D = cdist(X,Y, 'euclidean')
    K_rbf = np.exp(-sigma * D**2)
    K_linear = np.dot(X,Y.T)

    K_combine[: , 1:] = K_linear + a*K_rbf
    K_combine[: , 0] = [i+1 for i in range(X.shape[0])]

    return K_combine
```

- Use “get_kernel” to get the Kernel matrix for train and test.

  x is training data, x^* is testing data:

  For train : K(x, x) , for test : K(x^*, x)

  And, set the training parameters isKernel = True, t = 4 (precomputed kernel)

  First, let sigma in ref kernel is 1/784 (same as the default of libsvm), and the weight a =1.

  We can get the result below:

```
sigma = 1.0/784.0
a = 1
K_combine_train = get_kernel(X_train, X_train, sigma, a)
K_combine_test = get_kernel(X_test, X_train, sigma, a)
```

```
prob = svm_problem(Y_train, K_combine_train, isKernel=True)
m = svm_train(prob, '-t 4')
```

```
res = svm_predict(Y_test, K_combine_test, m)
print (res[1])
```

result (ACC, MSE, SCC):

```
(95.08, 0.1404, 0.931149802516624)
```

comparison with others in the condition of default parameters (order in accuracy):

	ACC(Accuracy)	MSE(mean squared error)	SCC(squared correlation coefficient)
Rbf	**95.32%**	0.1492	0.927
Linear	95.08%	0.1404	0.931
Linear + Rbf	95.08%	0.1404	0.931
Polynomial	34.68%	2.6212	0.149

- Also I did a simple grid search to find the better parameters

  First, use a(weight) = 1 find the best accuracy by searching gamma under the range of [1/784,3/784,5/784].

```
max_accu = 0
max_combine = None
max_combine_g  = 0

for g in GAMMA:
    K_combine_train = get_kernel(X_train, X_train, g, a)
    K_combine_test = get_kernel(X_test, X_train, g, a)


    prob = svm_problem(Y_train, K_combine_train, isKernel=True)
    m = svm_train(prob, '-t 4')

    res = svm_predict(Y_test, K_combine_test, m)
    if max_accu< res[1][0]:
        max_accu = res[1][0]
        max_combine = res
        max_combine_g = g

print(max_combine[1])
print("gamma:{}".format(max_combine_g))
```

result (ACC, MSE, SCC):

```
(95.19999999999999, 0.138, 0.9323215333078685)
gamma:0.00637755102041
```

Then, fix the gamma, search a. The search range of a is [0.1,1,10]

```
max_accu = 0
max_combine = None
max_combine_a  = 0
weight = [0.1,1,10]

for a in weight:
    K_combine_train = get_kernel(X_train, X_train, max_combine_g, a)
    K_combine_test = get_kernel(X_test, X_train, max_combine_g, a)


    prob = svm_problem(Y_train, K_combine_train, isKernel=True)
    m = svm_train(prob, '-t 4')

    res = svm_predict(Y_test, K_combine_test, m)
    if max_accu< res[1][0]:
        max_accu = res[1][0]
        max_combine = res
        max_combine_a = a

print(max_combine[1])
print("weight:{}".format(max_combine_a))
```

result (ACC, MSE, SCC):

```
(96.0, 0.1128, 0.9444245741402817)
weight:10
```

I am wondering that linear + rbf really has the worse performance than rbf ? I tried to do another grid search in the search range a : [200,300,400], since the weight tends to be bigger. I get the better result when a = 300, accuracy up to 97.68%.

comparison with others in the condition of better parameters (order in accuracy):

	Parameters	ACC(Accuracy)	MSE(mean squared error)	SCC(squared correlation coefficient)
Polynomial	degree = 5 coef0 = 1 gamma = 5/784	**97.92%**	0.056	0.972
Linear + Rbf	weight = 300 gamma = 5/784	97.68%	0.0632	0.698
Rbf	gamma = 5/784	96.88%	0.1016	0.950
Linear	NO	95.08%	0.1404	0.931

- Discussion:

  Since the grid search range is set by myself, those maybe fail to include the optimized one. So, I did all simple grid search under the same search range, to make the comparison seems fairer under the suppose that we don't have any prior knowledge. However, the comparison I made in this report may not reflect the real performance of those kernels. There definitely have better performance train by other parameters in each kernel.