ML HW05 Report

I. Gaussian Process

a. Code

Part 1. Apply Gaussian Process Regression

```
In [2]: def Load_data():
            X = []
            Y = []
            with open("input.data",'r') as file:
                for line in file:
                    x,y = line.split(' ')
                    X.append(float(x))
                    Y.append(float(y))
            return np.array(X).reshape(-1,1) , np.array(Y).reshape(-1,1)
```

First, I load the data with this function and reshape NumPy array to n*1 where the n is the number of x in input data.

```
In [3]: def Rational_quadratic_kernel(sigma,alpha,l,xi,xj):
            dist = np.sum(xi ** 2,axis=1).reshape(-1,1) + np.sum(xj ** 2,axis=1) - 2*xi.dot(np.transpose(xj))
            kernel = (sigma ** 2) * (1 + dist / (2 * alpha * (l ** 2))) ** (-1 * alpha)
            return kernel
```

Use rational quadratic kernel as a kernel to compute the similarity between xi and xj, where sigma, alpha, l are the parameter of kernel. We can search rational quadratic kernel format on Wiki.

$$rational\ quadratic\ kernel = \sigma^2(1 + \frac{||x_i - x_j||^2}{2\alpha l^2})^{-\alpha}$$

```
In [6]: if __name__ == '__main__':
            X,Y = Load_data()
            X_perdict = np.linspace(-60.0, 60.0, 1000).reshape(-1, 1)
            beta = 5
            sigma = 1
            alpha = 1
            l = 1
            Gaussian_process(X, Y, X_perdict, beta, sigma, alpha, l)
```

X and Y are training data, beta is the reciprocal of variance of the noise function (error function). X_predict is the testing data. Because Spec said that we want to predict the distribution from X= -60 to X=+60, we space the number over the interval. I simply set the kernel parameter to 1

```
In [5]: def Gaussian_process(X,Y,X_star,beta,sigma,alpha,l):
            kernel = Rational_quadratic_kernel(sigma,alpha,l,X,X)
            kernel_1star = Rational_quadratic_kernel(sigma,alpha,l,X,X_star)
            kernel_2star = Rational_quadratic_kernel(sigma,alpha,l,X_star,X_star)
            C = kernel + np.identity(len(X))*(1/beta)
            inver_C = np.linalg.inv(C)

            mean_star = np.transpose(kernel_1star).dot(inver_C).dot(Y)
            k_star = kernel_2star + (1/beta)
            var_star = k_star - np.transpose(kernel_1star).dot(inver_C).dot(kernel_1star)
            #print(mean_star.shape,k_star.shape,var_star.shape)
```

Then I can implement Gaussian Process. I solve the covariance matrix C first, it's just like the format in the slide (picture below). k is kernel function, beta is the reciprocal of variance, and delta is Kronecker delta.

$$C(X_i, X_j) = k(x_i, x_j) + \beta^{-1}\delta_{ij}$$
$$\delta_{ij} = \begin{cases} 1, & i = j, \\ 0, & otherwise. \end{cases}$$

For the prediction, we can compute the mean and variance with the formula below. x is training data and x* is testing data.

$$\mu(x^*) = k(x, x^*)^T C^{-1} y$$
$$\sigma^2(x^*) = k^* - k(x, x^*)^T C^{-1} k(x, x^*)$$
$$k^* = k(x^*, x^*) + \beta^{-1}$$

```
#visualization
plt.plot(X_star, mean_star)
plt.scatter(X, Y, s=10)

interval = 1.95 * np.sqrt(np.diag(var_star))
X_star = X_star.reshape(-1)
mean_star = mean_star.reshape(-1)

plt.plot(X_star, mean_star + interval,color = "green")
plt.plot(X_star, mean_star - interval,color = "green")
plt.fill_between(X_star, mean_star + interval, mean_star - interval,alpha=0.3)

#plt.title(f'sigma: {sigma:.5f}, alpha: {alpha:.5f}, length scale: {l:.5f}')
plt.xlim(-60, 60)
plt.show()
```

After computing the mean and variance, we can plot the result.

Part 2. Optimize the kernel parameters

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

We want to optimize the kernel parameters by minimizing negative marginal log-likelihood. Here is the format above.

```
In [4]: def Negative_marginal_likelihood(theta):
            theta.ravel()
            C = Rational_quadratic_kernel(theta[0],theta[1],theta[2],X,X) + np.identity(len(X))*(1/beta)
            res = 0.5*np.log(np.linalg.det(C)) + 0.5*np.transpose(Y).dot(np.linalg.inv(C)).dot(Y) + len(X)/2*np.log(2*math.pi)
            return res.ravel()
```

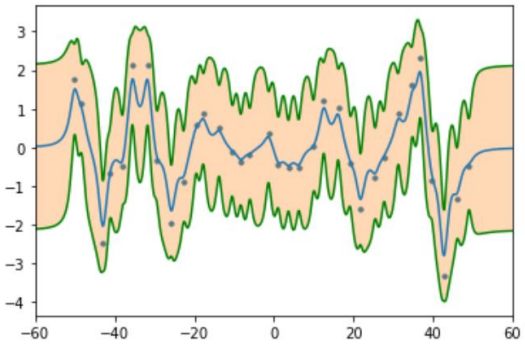
theta is kernel parameter

```
theta = [1,1,1]
res = minimize(Negative_marginal_likelihood, theta)
sigma_ = res.x[0]
alpha_ = res.x[1]
l_ = res.x[2]
Gaussian_process(X, Y, X_perdict, beta, sigma_, alpha_, l_)
```

And we use `scipy.optimize` to minimize the marginal log-likelihood. We set all the kernel parameters to 1 for the initial guess.

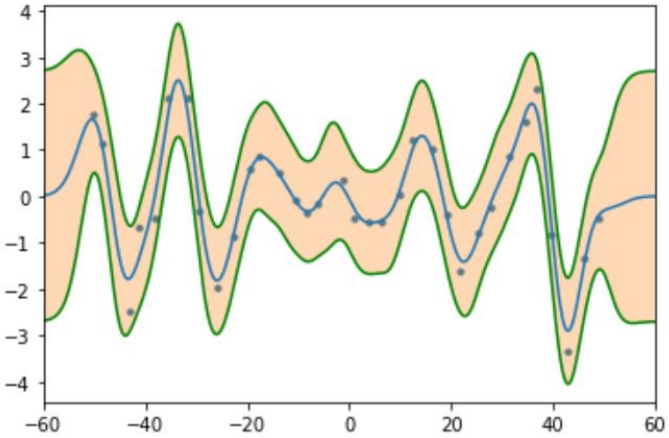
b. Result

Part 1. Apply Gaussian Process Regression.



Part 2. Optimize the kernel parameters

Result



c. Discussion & observations

1. result 2 is obviously better than result 1
2. When l in kernel parameter is high → underfitting
   l in kernel parameter is low → overfitting
3. It may show bad result if we have bad parameters. The worst case of overfitting is multiple impulse function. It may fit the training data, bit it is not the general function we are trying to get.

II. SVM

a. Code

Part 1. Different kernel functions to SVM classifier

In [2]:
```
def read_file(filename):
    with open(filename,'r') as file:
        lines = file.readlines()
    lines = np.array( [line.strip().split(',') for line in lines] , dtype = 'float64')
    return lines
```

In [3]:
```
def read_MINST():
    X_train = read_file("X_train.csv")   #5000 * 784
    X_test = read_file("X_test.csv")     #2500 * 784
    Y_train = read_file("Y_train.csv")   #5000 * 1
    Y_test = read_file("Y_test.csv")     #2500 * 1

    Y_train = Y_train.reshape(1,-1)
    Y_test = Y_test.reshape(1,-1)

    return X_train, X_test, Y_train, Y_test
```

Use read_MINST to read .csv and return X_train, X_test, Y_train, Y_test as NumPy array.

In [17]:
```
if __name__ == '__main__':

    X_train, X_test, Y_train, Y_test = read_MINST()

    #Part 1
    SVM("Linear","-s 0 -t 0")  #-c
    SVM("Polynomail","-s 0 -t 1") # -c -r -g -d
    SVM("RBF","-s 0 -t 2") #-c -g
```

param record the parameter of SVM training. -s: SVM type and 0 is default as C-SVC.
-t: 0,1,2 is used to choose kernel function. 0,1,2 is correspond to Linear, Polynomial, RBF.

In [4]:
```
def SVM(kernel,param):
    print(kernel)
    param += " -q"
    param_ = svm_parameter(param)

    start = time.time()
    prob = svm_problem(Y_train[0],X_train)   #prob  = svm_problem(y, x)
    model = svm_train(prob,param_)        #model = svm_train(prob, param)
    p_label, p_acc, p_val = svm_predict(Y_test[0], X_test, model)  #p_label, p_acc, p_val = svm_predict(yt, xt, model)
    print("%0.2f sec" %(time.time()-start))
    print("p_acc",p_acc)
    print("\n")
    return p_acc
```

In this code, -q means not to print while training. We can use the function which is imported by libsvm.svmutil. svm_predict return three term, which are predicted label, acc, val. In the accuracy include three value: accuracy, mean square error, and squared correlation coefficient.

Part 2. Grid search

In [5]:
```
def GridSearch_svm(kernel_type,param,fold):
    print(kernel_type,"\n")
    if fold !=0:
        param += " -v "+str(int(fold))
    param += " -q"
    param_ = svm_parameter(param)
    prob = svm_problem(Y_train[0],X_train)
    acc = svm_train(prob,param_)

    return acc
```

```
#Part 2
GridSearch("Linear")
GridSearch("Polynomail")
GridSearch("RBF")
```

In [6]:
```
def GridSearch(kernel_type):
    c = [1e-2,1e-1,1,10,100]   #cost
    #d = range(1,10)           #polynomial_kernel degree
    g = [1e-2,1e-1,1,10,100]   #polynomial_kernel and kbf gmma
    #r = range(-10,10,1)       #polynomial_kernel cof0
    n = 3

    max_acc = 0
    max_param = 0
    if kernel_type == 'Linear': #-c
        for ci in c:
            param = " -c " + str(float(ci))
            acc = GridSearch_svm(kernel_type+param, "-t 0"+param, n)
            if acc > max_acc:
                max_acc = acc
                max_param = param

    elif kernel_type == 'Polynomail': # -c -r -g -d
        for ci in c:
            param = " -c " + str(float(ci))
            for ri in range(0,5,1):
                param = " -r " + str(int(ri))
                for gi in g:
                    param = " -g " + str(float(gi))
                    for di in range(2,5):
                        param = " -d " + str(int(di))
                        acc = GridSearch_svm(kernel_type+param, "-t 1"+param, n)
                        if acc > max_acc:
                            max_acc = acc
                            max_param = param


    elif kernel_type == 'RBF': #-c -g
        for ci in c:
            param = " -c " + str(float(ci))
            for gi in g:
                    param = " -g " + str(float(gi))
                    acc = GridSearch_svm(kernel_type+param, "-t 1"+param, n)
                    if acc > max_acc:
                        max_acc = acc
                        max_param = param
    print("#############")
    print("kerenl_type",kernel_type)
    print("Max acc",max_acc)
    print("Max param",max_param)
    print("#############")
```

Within GridSearch(), I defined some parameters those will use in different kernel_type. After GridSearch() went through all parameter, we will get the best parameter store in max_param.

Some new parameters:

-c: cost, set C in C_SVC

-d: degree, set degree in kernel function

-g: gamma, default is 1/num_features (num_features is 784 in this case)

-r: coef0

-v: n, n-fold cross validation, without -v means not using cross validation

Part 3. Linear kernel + RBF kernel

```
#Part 3
g = [1/784, 1, 1e-1, 1e-2, 1e-3, 1e-4]
c = [1e-4, 1e-3, 1e-2, 0.1, 1, 10, 100, 1000, 10000]
row, col = X_train.shape
max_acc = 0.0
g_best = 0
linear_k = Linear_kernel(X_train, X_train)
for gi in range(len(g)):
    rbf_k = RBF_kernel(X_train, X_train, -g[gi])
    user_k = linear_k + rbf_k
    user_k = np.hstack((np.arange(1, row+1).reshape(-1, 1), user_k))
    prob = svm_problem(Y_train[0], user_k, isKernel=True)
    for ci in range(len(c)):
        param_str = "-t 4 -c " + str(c[ci]) + " -v 3 -q"
        param_rec = "-t 4 -c " + str(c[ci]) + " -q"
        print("-g", g[gi], param_str)
        param = svm_parameter(param_str)
        val_acc = svm_train(prob, param)

        if val_acc > max_acc:
            max_acc = val_acc
            max_param = param_rec
            g_best = gi
print("============================")
print("Best Parameters:", " -g", g[g_best], max_param)
print("Max accuracy:", max_acc)

linear_k = Linear_kernel(X_train,X_train)
rbf_k = RBF_kernel(X_train, X_train, -100)
user_k = linear_k + rbf_k
user_k = np.hstack((np.arange(1, row+1).reshape(-1, 1), user_k))
prob = svm_problem(Y_train[0], user_k, isKernel=True)
param = svm_parameter("-g 100 -t 4 -c 0.1")
model = svm_train(prob, param)

#x_test = read_dataset("X_test.csv")  # (2500,784)
row, col = X_test.shape
linear_k = Linear_kernel(X_test, X_test)
rbf_k = RBF_kernel(X_test, X_test, -100)
user_k = linear_k + rbf_k
user_k = np.hstack((np.arange(1, row+1).reshape(-1, 1), user_k))
p_label, p_acc, p_val = svm_predict(Y_test[0], user_k, model)
print("p_acc",p_acc)
print("\n")
```

libsvm.svmutil supports user-defined kernel, we need to precomputed the data and set isKernel = True. We also need to do GridSearch to get the best parameter. Use the linear + RBF as a kernel and use best parameter and train it again.

b. Result

Part 1. Apply different kernel functions to SVM classifier

Kernel	Testing Acc(%)	Default parameters
Linear	95.08	
Polynomial	34.68	-g 1/784 -r 0 -d 3
RBF	95.32	-g 1/784

Part 2. Grid search

Kernel	Cross validation Acc(%)	Testing Acc(%)	parameters
Linear	96.86	95.96	-c 0.01
Polynomial	98.24	97.72	-c 0.1 -r 2 -g 1.0 -d 2
RBF	97.61	97.12	-c 0.01 -g 10.0

Part 3. Linear kernel + RBF kernel

Kernel	Cross validation Acc(%)	Testing Acc(%)	parameters
Linear + RBF	97.06	34.2	-g 0.01 -t 4 -c 0.01

c. Discussion & observations

1. The highest accuracy is SVM with RBF kernel and the lowest one is SVM with polynomial kernel function.
2. Testing result(%) may be worse than cross validation result, because the best parameter with training data may not be the best in testing data.
3. Linear kernel doesn't need any parameter, so it takes minimum times to compute.
4. Polynomial kernel takes maximum times while doing compute since it need to fine-tuned lots of parameters.
5. RBF kernel is a well-used kernel for its great ability in classification. It take less time than linear, but RBF get better testing accuracy in result.
6. Parameter c is control the tolerance to the model, if c value is higher, it means that the less tolerance the model has a error. The result may too fit to the training data.