I. Gaussian Process

> ### a. code with detailed explanations (20%)
>
> > There are some pre-defined function in HW05_GaussianProcess_util.py before implement Gaussian process's partI and part II.
> >
> > First, the function of load_file, it read data from file "input.data" and store them in the variable trainX and trainY.
> >
> > ```
> > def load_file():
> > fin = open('./data/input.data', 'r')
> > trainX=np.zeros(0)
> > trainY=np.zeros(0)
> > for line in fin.readlines():
> > x,y=np.float_(line.split(' '))
> > trainX=np.append(trainX,x)
> > trainY=np.append(trainY,y)
> > #plt.scatter(trainX,trainY)
> > return trainX,trainY
> > ```
> >
> > Second, the function of kernel, it's the rational quadratic kernel whose formula is $k(x_1,x_2)=(1+\frac{(x_1-x_2)^2}{2\alpha l^2})^{-\alpha}$, where α and l are hyperparameters and their default value are both 1. Note that here x_1 and x_2 are reshaped in order to find all the "pairwise" datapoint's similarity.
> >
> > ```
> > def kernel(x1,x2,alpha=1,lengthScale=1):
> > #rational quadratic kernel, K(x, x') = (1 + (x-x')^2 / (2*alpha *
> > length_scale^2))^-alpha,  refer to
> > https://www.cs.toronto.edu/~duvenaud/cookbook/
> > #compute "every" data point pairs' similarity by kernel
> > x1=x1.reshape(-1,1)
> > x2=x2.reshape(1,-1)
> > return np.power(1 + np.power((x1-x2),2)/(2*alpha*lengthScale**2),-alpha)
> > ```
> >
> > Third, the function of drawing, it will present the result of Gaussian process.
> >
> > ```
> > def drawing(trainX,trainY,testX,testY_mean,testY_sd):
> > #color list : https://zhuanlan.zhihu.com/p/65220518
> > plt.plot(trainX,trainY,'ko') # Show all training data points.
> > plt.plot(testX,testY_mean,'k-')
> > plt.fill_between(testX,testY_mean+2*testY_sd,testY_mean-
> > 2*testY_sd,facecolor='papayawhip') #draw 95% confident interval => 2*standard
> > deviation ;vs.  68%=>1*standard deviation , 99.7%=>3*standard deviation
> > plt.xlim(-60,60)
> > plt.show()
> > ```

Part1 (10%)

Part1's code applies Gaussian process with rational quadratic kernel by using the formula in PPT p.48.

x represents the train datapoints, x^* represents the test datapoint.

Test label's mean, testY_mean is defined by $\mu(x^*) = k(x, x^*)^T C^{-1} y$.

Test label's variance, testY_variance is defined by
$\sigma^2(x^*) = (k(x^*, x^*) + \beta^{-1}) - k(x, x^*)^T C^{-1} k(x, x^*)$.

Covariance matrix C is defined by using the formula in PPT p.45,
$C(x_n, x_m) = k(x_n, x_m) + \beta^{-1}\delta_{nm}$.

Here the kernel uses the default parameters, $\alpha = 1, l = 1$.

```python
import numpy as np
from HW05_GaussianProcess_util import *
if __name__=='__main__':
#load file and initialize parameters
trainX,trainY=load_file()
beta=5

#prediction refer to PPT p.48
C = kernel(trainX,trainX) + 1/beta * np.identity(len(trainX)) #we can compute
the similarity between random variables s by covariance because the covariance
is actually the kernel, refer to PPT p.45
testX=np.linspace(-60,60,num=100) #create test data in [-60,60]
testY_mean = kernel(trainX,testX).T @ np.linalg.inv(C) @ trainY.reshape(-1,1)
#use gaussian regression to predict the mean of test y
testY_mean = testY_mean.reshape(-1) #for plot readibly
testY_variance = kernel(testX,testX)+1/beta*np.identity(len(testX)) -
kernel(trainX,testX).T @ np.linalg.inv(C) @ kernel(trainX,testX) #use gaussian
regression to predict the variance of test y
testY_sd = np.sqrt(np.diag(testY_variance)) #only need variance(x1,x1), not
variance(x1,x2), and we convert it from variance to standard deviation

#visualization
drawing(trainX,trainY,testX,testY_mean,testY_sd)
```

Part2 (10%)

Part2's code optimizes the kernel parameters by minimizing negative marginal log-likelihood which is in PPT p.52.

$$\begin{aligned}
\ln p(y|\theta) &= -\frac{1}{2}ln|C_\theta| - \frac{1}{2}y^T C_\theta^{-1} y - \frac{N}{2}ln(2\pi) \\
\Rightarrow \theta^* &= \text{argmax}\{\ln p(y|\theta)\} \\
&= \text{argmax}\{-\frac{1}{2}ln|C_\theta| - \frac{1}{2}y^T C_\theta^{-1} y - \frac{N}{2}ln(2\pi)\} \\
&= \text{argmin}\{\frac{1}{2}ln|C_\theta| + \frac{1}{2}y^T C_\theta^{-1} y + \frac{N}{2}ln(2\pi)\}
\end{aligned}$$

We find the optimal parameter by using scipy.optimize.minimize and set the initial parameters' value from 1e-10 to 1e+10.

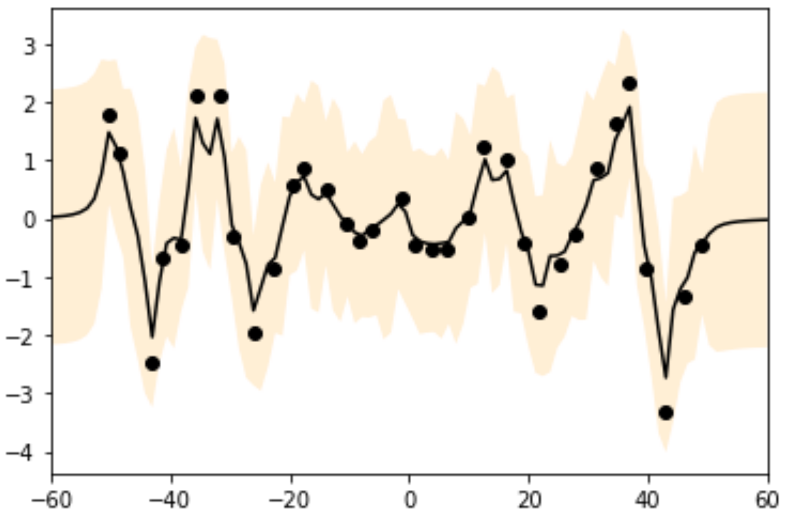
After finding the optimal parameter α, l by the formula, we fit the them into the same process as partI to do the prediction.

```
import numpy as np
from scipy.optimize import minimize
from HW05_GaussianProcess_util import *

#objective function refer to PPT p.52
def objective(trainX,trainY,beta):
def negative_log_likelihood(theta): #theta represents the kernel's
hyperparameters, here the hyperparametersare are alpha and length_scale
   C_theta=kernel(trainX,trainX,theta[0],theta[1]) + 1/beta *
np.identity(len(trainX))
   return ((1/2)*np.log(np.linalg.det(C_theta)) +
(1/2)*trainY.reshape(1,-1)@np.linalg.inv(C_theta)@trainY.reshape(-1,1) +
(1/2)*len(trainX)*np.log(2*np.pi)).item()
return negative_log_likelihood #return a function , refer to
https://codertw.com/程式語言/359722/

if __name__=='__main__':
#load file and initialize parameters
trainX,trainY=load_file()
beta=5

#maximize log likelihood is same as minimize negative log likelihood
opt_min_log_likelihood_value = 1e10
init_thetas=[10**x for x in range(-10,10)] #search for the optimal parameter by
setting the initial value pairs from 1e-10 to 1e+10
for init_theta1 in init_thetas:
   for init_theta2 in init_thetas:
       cur_min_log_likelihood=minimize(objective(trainX,trainY,beta),x0=
[init_theta1,init_theta2],bounds=((1e-5,1e5),(1e-5,1e5))) #bound to avoid
dividing by 0 in kernel function
       if cur_min_log_likelihood.fun < opt_min_log_likelihood_value:
           opt_min_log_likelihood_value = cur_min_log_likelihood.fun
           opt_theta1 , opt_theta2 = cur_min_log_likelihood.x
           print(opt_theta1 , opt_theta2)
print("optimal alpha:{}, optimal length scale:
{}".format(opt_theta1,opt_theta2))

#prediction refer to PPT p.48, now with the optimal parameter theta => optimal
alpha and optimal length scale, the others are same as
HW05_GaussianProcess_part1
C = kernel(trainX,trainX,opt_theta1,opt_theta2) + 1/beta *
np.identity(len(trainX)) #we can compute the similarity between random
variables s by covariance because the covariance is actually the kernel, refer
to PPT p.45
testX=np.linspace(-60,60,num=100) #create test data in [-60,60]
testY_mean = kernel(trainX,testX,opt_theta1,opt_theta2).T @ np.linalg.inv(C) @
trainY.reshape(-1,1) #use gaussian regression to predict the mean of test y
testY_mean = testY_mean.reshape(-1) #for plot readibly
testY_variance =
kernel(testX,testX,opt_theta1,opt_theta2)+1/beta*np.identity(len(testX)) -
kernel(trainX,testX,opt_theta1,opt_theta2).T @ np.linalg.inv(C) @
kernel(trainX,testX,opt_theta1,opt_theta2) #use gaussian regression to predict
the variance of test y
testY_sd = np.sqrt(np.diag(testY_variance)) #only need variance(x1,x1), not
variance(x1,x2), and we convert it from variance to standard deviation

#visualization
drawing(trainX,trainY,testX,testY_mean,testY_sd)
```

b. experiments settings and results (20%)

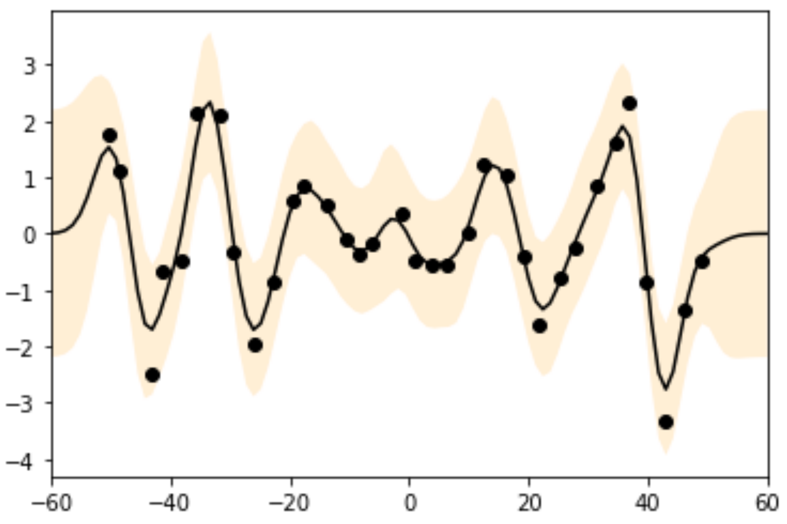
Part1 (10%)

default alpha = 1, length scale = 1



Part2 (10%)

```
optimal alpha:99999.99999149832,
optimal length scale:2.967833657122355
```



c. observations and discussion (10%)

1. When scaling up the length scale, the prediction becomes more smooth.
2. A little change in length scale can change the result more than a little change in alpha.
3. The result with optimal parameters has a smaller variance than the result with default parameters.

II. SVM

a. code with detailed explanations (20%)

Part1 (5%)

Use libsvm.svmutil.svm_train with different kernel with default parameters to do the prediction.

linear kernel : $k(x_1, x_2) = x_1^T \cdot x_2$

polynomial kernel : $k(x_1, x_2) = (\gamma x_1^T \cdot x_2 + c)^P$, P is the polynomial degree, γ and c are hyperparameters.

RBF kernel : $k(x_1, x_2) = \exp(-\gamma||x_1 - x_2||^2)$, γ is hyperparameter.

```
#!pip install libsvm #https://projets-lium.univ-
lemans.fr/sidekit/_modules/libsvm/svmutil.html
from libsvm.svmutil import *
import numpy as np

#load files
trainX = np.genfromtxt('./data/X_train.csv', delimiter=',')
trainY = np.genfromtxt('./data/Y_train.csv', delimiter=',')
testX = np.genfromtxt('./data/X_test.csv', delimiter=',')
testY = np.genfromtxt('./data/Y_test.csv', delimiter=',')
```

```
#use different kernels and see their performance
kernel = ['linear','polynomial','RBF']
for kernel_type in range(3): #-t kernel_type, kernel_type=0/1/2 means
linear/polynomial/RBF kernel respectively
 model = svm_train(trainY,trainX,'-q -t {}'.format(kernel_type)) #-q means
quiet mode (no outputs)
 p_label,p_acc,p_vals = svm_predict(testY,testX,model,'-q')
 #p_labels means a list of predicted labels
 #p_acc: a tuple including  accuracy (for classification), mean-squared error,
and squared correlation coefficient (for regression).
 #p_vals: a list of decision values or probability estimates
            #(if \'-b 1\' is specified). If k is the number of classes,
            #for decision values, each element includes results of predicting
k(k-1)/2 binary-class SVMs.
            #for probabilities, each element contains k values indicating the
probability that the testing instance is in each class.
 print('{} kernel\'s accuracy: {:.2f}'.format(kernel[kernel_type],p_acc[0]))

#result
#linear kernel's accuracy: 95.08
#polynomial kernel's accuracy: 34.68
#RBF kernel's accuracy: 95.32
```

Part2 (10%)

Find parameters of the best performing model by using the grid search.

The hyperparameter C and γ are from 2^{-5} to 2^5.

```
import numpy as np
from libsvm.svmutil import * #https://projets-lium.univ-
lemans.fr/sidekit/_modules/libsvm/svmutil.html

#load files and initialize parameters
trainX = np.genfromtxt('./data/X_train.csv', delimiter=',')
trainY = np.genfromtxt('./data/Y_train.csv', delimiter=',')
testX = np.genfromtxt('./data/X_test.csv', delimiter=',')
testY = np.genfromtxt('./data/Y_test.csv', delimiter=',')

#grid search
hyperParameters = [2**x for x in range(-5,5)]
accuracyMatrix=np.zeros((len(hyperParameters),len(hyperParameters)))
for IndexC in range(len(hyperParameters)):
 for IndexGamma in range(len(hyperParameters)):
     accuracyMatrix[IndexC,IndexGamma]=svm_train(trainY,trainX,'-q -s 0 -t 2 -v
3 -c {} -g {}'.format(hyperParameters[IndexC],hyperParameters[IndexGamma]))
     #-q : quiet mode (no outputs)
     #-s svm_type : set type of SVM (default 0 => C-SVC)
     #-t kernel_type : set type of kernel function (default 2 => RBF)
     #-v n: n-fold cross validation mode
     #-c cost : set the parameter C of C-SVC, epsilon-SVR, and nu-SVR (default 
1)
     #-g gamma : set gamma in kernel function (default 1/num_features)

#extract the optimal hyperparameters
OptIndexC,OptIndexGamma = 
np.unravel_index(np.argmax(accuracyMatrix,axis=None),accuracyMatrix.shape)
OptC,OptGamma = hyperParameters[OptIndexC],hyperParameters[OptIndexGamma]
print("accuracy matrix: ")
np.set_printoptions(precision=2, suppress=True)
print(accuracyMatrix)
print("optimal C: {},\noptimal Gamma: {},\noptimal accuracy: 
{}".format(OptC,OptGamma,accuracyMatrix[OptIndexC,OptIndexGamma]))
```

Part3 (5%)

Kernel + kernel is still a kernel.

Part3 use user-defined kernel to do the SVM.

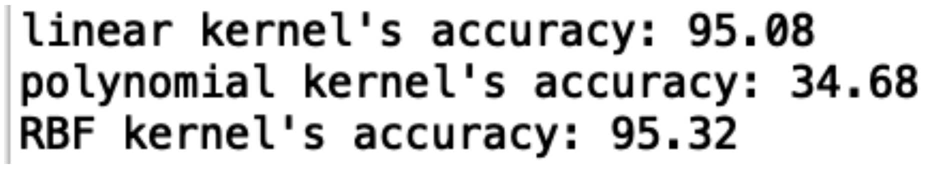
The formulas of linear kernel and RBF kernel have been discussed in SVM part2, where the RBF's gamma is set as the optimal value from SVM part2, 0.03125.

```
import numpy as np
from libsvm.svmutil import *
from scipy.spatial.distance import cdist
#https://projets-lium.univ-lemans.fr/sidekit/_modules/libsvm/svmutil.html
#https://stackoverflow.com/questions/7715138/using-precomputed-kernels-with-
libsvm

def precomputed_kernel(x1,x2,gamma):
 #compute kernel matrices between every pairs of (x1,x2) and include sample 
serial number as first column
 #shape of x1 : (Lx1,784)
 #shape of x2 : (Lx2,784)
 LinearKernel = x1 @ x2.T #=> (Lx1,Lx2)
 RBFKernel = np.exp(-gamma * cdist(x1, x2, 'sqeuclidean')) #=> (Lx1,Lx2); 
sqeuclidean refers to 
https://docs.scipy.org/doc/scipy/reference/generated/scipy.spatial.distance.sqe
uclidean.html
 LinearRBFKernel = LinearKernel + RBFKernel #=> (Lx1,Lx2); combine kernel with 
linear kernel and RBF kernel
 LinearRBFKernel = 
np.hstack((np.arange(1,len(x1)+1).reshape(-1,1),LinearRBFKernel)) #the training 
file the first column must be the "ID" of xi. In testing, ? can be any value. 
refers to https://github.com/cjlin1/libsvm/blob/master/README
 return LinearRBFKernel
```

```
if __name__=='__main__':
 #load files and initialize parameters
 trainX = np.genfromtxt('./data/X_train.csv', delimiter=',')
 trainY = np.genfromtxt('./data/Y_train.csv', delimiter=',')
 testX = np.genfromtxt('./data/X_test.csv', delimiter=',')
 testY = np.genfromtxt('./data/Y_test.csv', delimiter=',')
 gamma=0.03125 #is the optimal gamma from HW05_SVM_part2

 #defined our own kernels for every pairs of (trainX,trainX) and (testX,trainX)
 kernel_train_train = precomputed_kernel(trainX,trainX,gamma)
 kernel_train_test = precomputed_kernel(testX, trainX, gamma)

 #use user-defined kernel to do SVM
 #training
 prob = svm_problem(trainY,kernel_train_train,isKernel=True) #isKernel=True
must be set for precomputed kernel. refers to
https://github.com/cjlin1/libsvm/blob/master/python/README
 param = svm_parameter('-q -t 4')
 #-q : quiet mode (no outputs)
 #-t kernel_type : set type of kernel function (default 2), kernel_type==4
means precomputed kernel (kernel values in training_set_file)
 model = svm_train(prob,param)

 #prediction
 p_label,p_acc,p_vals = svm_predict(testY,kernel_train_test,model,'-q') #"-q" :
quiet mode (no outputs).
 print('SVM using linear kernel + RBF kernel together\'s accuracy:
{}'.format(p_acc[0]))
```

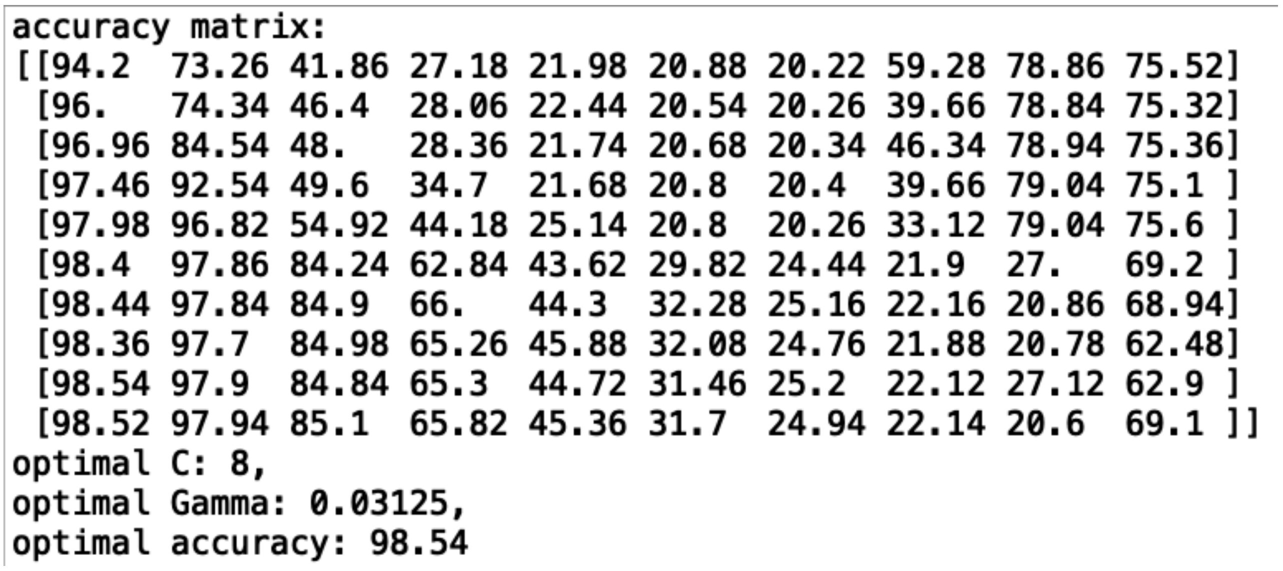
b. experiments settings and results (20%)

Part1 (5%)



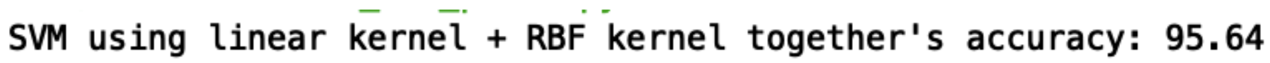
```
linear kernel's accuracy: 95.08
polynomial kernel's accuracy: 34.68
RBF kernel's accuracy: 95.32
```

Part2 (10%)



```
accuracy matrix:
[[94.2  73.26 41.86 27.18 21.98 20.88 20.22 59.28 78.86 75.52]
 [96.   74.34 46.4  28.06 22.44 20.54 20.26 39.66 78.84 75.32]
 [96.96 84.54 48.   28.36 21.74 20.68 20.34 46.34 78.94 75.36]
 [97.46 92.54 49.6  34.7  21.68 20.8  20.4  39.66 79.04 75.1 ]
 [97.98 96.82 54.92 44.18 25.14 20.8  20.26 33.12 79.04 75.6 ]
 [98.4  97.86 84.24 62.84 43.62 29.82 24.44 21.9  27.   69.2 ]
 [98.44 97.84 84.9  66.   44.3  32.28 25.16 22.16 20.86 68.94]
 [98.36 97.7  84.98 65.26 45.88 32.08 24.76 21.88 20.78 62.48]
 [98.54 97.9  84.84 65.3  44.72 31.46 25.2  22.12 27.12 62.9 ]
 [98.52 97.94 85.1  65.82 45.36 31.7  24.94 22.14 20.6  69.1 ]]
optimal C: 8,
optimal Gamma: 0.03125,
optimal accuracy: 98.54
```

Part3 (5%)



```
SVM using linear kernel + RBF kernel together's accuracy: 95.64
```

c. observations and discussion (10%)

1. Based on the result from SVM part1, the polynomial kernel has the worst performance.
2. Compare SVM part 3 with SVM part1, linear kernel + RBF kernel's performance(95.64%) is better than linear kernel's accuracy(95.08%) and RBF kernel's accuracy(95.32%). Hence, it's good the combined the two kernels.