Machine Learning Homework 5

Student ID: 0760814

Name: 黃日明

1. Gaussian Process

At first, we need to read the input data from “input.data”.

```python
def read_file():
    x = []
    y = []
    with open('./input.data') as file:
        for line in file:
            x.append(float(line.split()[0]))
            y.append(float(line.split()[1]))
    return np.array(x).reshape(-1, 1), np.array(y).reshape(-1, 1)
```

After that, I take the return values from read_file() and set default values. Here noise is set to ⅕ as in the document and X is an array which is used to plot the graph later.

```python
x_train, y_train = read_file()
noise = 1 / 5
X = np.arange(-60, 60, 1).reshape(-1, 1)
```

1.1. Apply Gaussian Process Regression to predict the distribution of f and visualize the result

In this homework, it required to use Rational Quadratic Kernel which is defined as follows:

$$k_{RQ}(x, x') = \sigma^2\left(1 + \frac{(x-x')^2}{2\alpha l^2}\right)^{-\alpha}$$

with:

- σ is the overall variance
- l the length scale
- α the scale-mixture

The formula of Rational Quadratic Kernel is easily transferred to code with the help of scipy.spatial.distance.cdist.

```python
def rational_quadratic_kernel(x, x_s, l=1.0, sigma=1.0, alpha=1.0):
    return (sigma ** 2) * (1 + cdist(x, x_s, 'sqeuclidean') / (2 * alpha * l ** 2)) **
(-alpha)
```

By the definition of the Gaussian Process, the joint distribution of observed data y and prediction f_* is

$$\binom{y}{f_*} \sim N\left(0, \begin{pmatrix} K_y & K_* \\ K_*^y & K_{**} \end{pmatrix}\right)$$

With:

- $K \ = k(x , x) \ + \ \beta^{-1}$
- $K_{**} = k(x^*, x^*) \ + \ \beta^{-1}$
- $K_* = k(x , x^*)$
- $K_*^y = k(x , x^*)^T$
- $k(x , y)$ is the Rational Quadratic Kernel

Then it is possible to calculate new mean and covariance as follows:

$$\mu_* = K_*^T K_y^{-1} y$$
$$\Sigma_* = K_{**} - K_*^T K_y^{-1} K_*$$

All formulation above then transfer into code to predict the distribution of f.

```python
def posterior_predictive(x_s, x_train, y_train, l=1.0, sigma_f=1.0, alpha=1.0, noise=1e-8):
    K = rational_quadratic_kernel(x_train, x_train, l, sigma_f, alpha) + noise * 
np.eye(len(x_train))
    K_s = rational_quadratic_kernel(x_train, x_s, l, sigma_f, alpha)
    K_ss = rational_quadratic_kernel(x_s, x_s, l, sigma_f, alpha) + noise * np.eye(len(x_s))
    K_inv = inv(K)
    mu_s = K_s.T.dot(K_inv).dot(y_train)
    cov_s = K_ss - K_s.T.dot(K_inv).dot(K_s)

    return mu_s, cov_s
```

Then call the posterior_predictive function to get mu_s and cov_s and feed them into plot_gp to illustrate the result

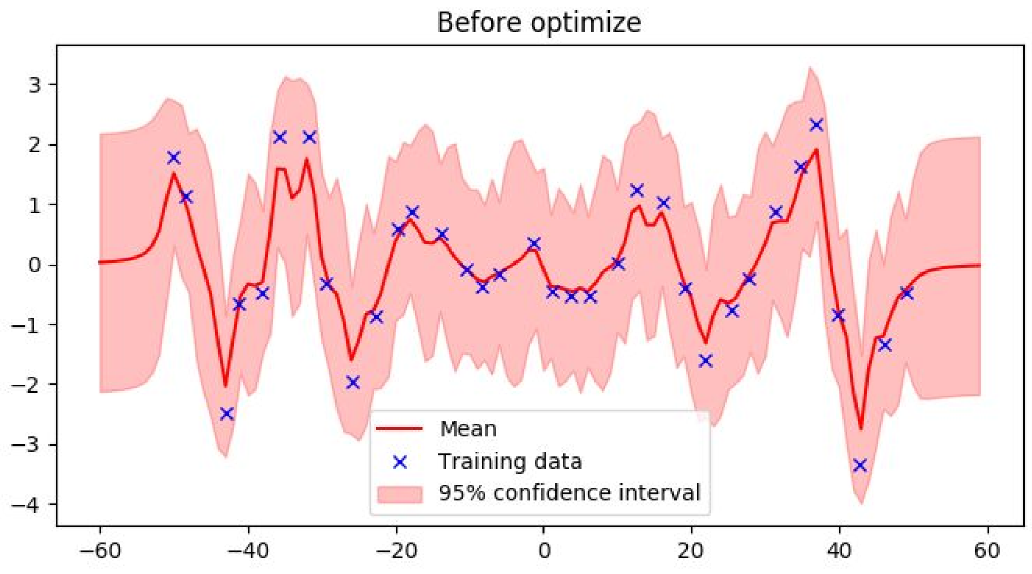
```python
mu_s, cov_s = posterior_predictive(X, x_train, y_train, noise=noise)
plot_gp(mu_s, cov_s, X, x_train=x_train, y_train=y_train, fig_name="Before optimize")
```

The function plot_gp is implemented as below:

```python
def plot_gp(mu, cov, x, x_train, y_train, fig_name):
    x = x.ravel()
    mu = mu.ravel()
    uncertainty = 1.96 * np.sqrt(np.diag(cov))
    plt.box(False)
    plt.figure(figsize=(8, 4))
    plt.fill_between(x, mu + uncertainty, mu - uncertainty, alpha=0.25, color='r', 
label='95% confidence interval')
```

```
    plt.plot(x, mu, label='Mean', color='r')

    plt.plot(x_train, y_train, 'bx', label='Training data')
    plt.legend()
    plt.title(fig_name)
    plt.savefig(fig_name + ".png")
```



After plot_gb has been executed, the visualization is displayed as above with red line is the mean of f in range of [-60, 60], blue x is training data and the blur red marks 95% confidence inteval of f.

1.2. Optimize the kernel parameters by minimizing negative marginal log-likelihood, and visualize the result again

The optimal kernel parameters can be estimated by minimizing the negative marginal log-likelihood which is given by

$$\frac{1}{2}y^T K_y^{-1} y + \frac{1}{2}log|K_y| - \frac{N}{2}log(2\pi)$$

This formula is implemented and using with scipy.optimize.minimize as follows:

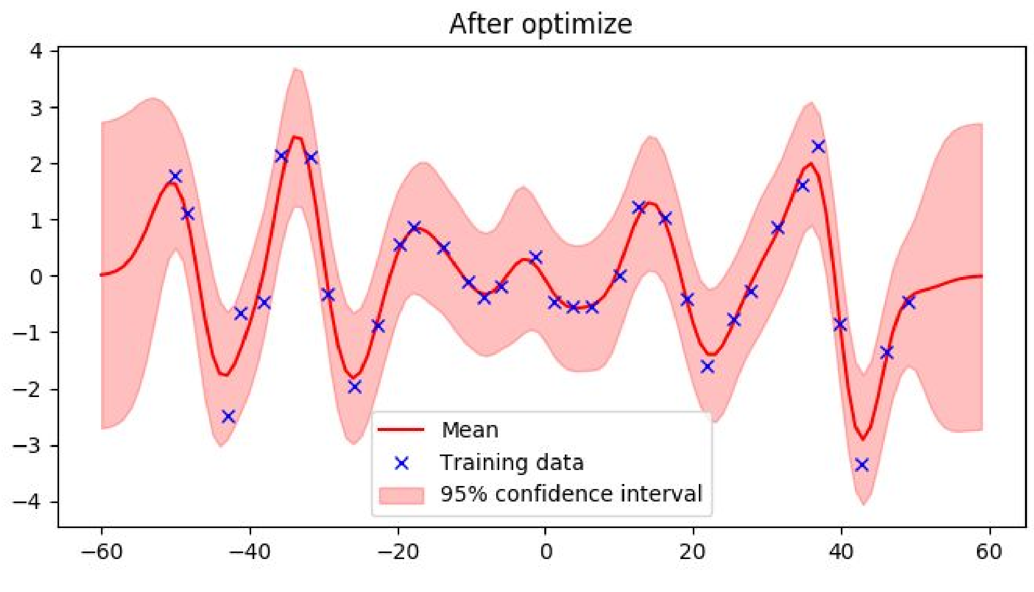
```
def nll_fn(X, x_train, y_train, noise):
   l = X[0]
   sigma = X[1]
   alpha = X[2]
   K = rational_quadratic_kernel(x_train, x_train, l, sigma, alpha) + noise *
np.eye(len(x_train))
   return 0.5 * np.log(det(K)) + 0.5 * y_train.T.dot(inv(K).dot(y_train)) + 0.5 *
len(x_train) * np.log(2 * np.pi)

res = minimize(nll_fn, x0=np.array([1, 1, 1]), args=(x_train, y_train, noise),
bounds=((1e-5, None), (1e-5, None), (1e-5, None)), method='L-BFGS-B')
```

The minimize function take in nll_fn with arguments are x_train, y_train, noise which was already defined before. Because it optimizes 3 values l, σ and α in Rational Quadratic Kernel then we need to input 3 initial values. In this case are [1, 1, 1] to simplify.

```
l_opt, sigma_f_opt, alpha_opt = res.x
mu_s, cov_s = posterior_predictive(X, x_train, y_train, l=l_opt, sigma_f=sigma_f_opt,
alpha=alpha_opt, noise=noise)
plot_gp(mu_s, cov_s, X, x_train=x_train, y_train=y_train, fig_name="After optimize")
```

The optimized values is put back in to posterior_predictive function to retrieve mu_s and cov_s then the result is visualized as below



2. SVM on MNIST dataset

At first, let read training data and testing data.

```
with open('./X_train.csv') as x_train_file:
   csv_reader = csv.reader(x_train_file, delimiter=',')
   x_train = scipy.asarray([row for row in csv_reader], dtype=float)

with open('./Y_train.csv') as x_train_file:
   csv_reader = csv.reader(x_train_file, delimiter=',')
   y_train = scipy.asarray([row for row in csv_reader], dtype=float).squeeze()

with open('./X_test.csv') as x_train_file:
   csv_reader = csv.reader(x_train_file, delimiter=',')
   x_test = scipy.asarray([row for row in csv_reader], dtype=float)

with open('./Y_test.csv') as x_train_file:
   csv_reader = csv.reader(x_train_file, delimiter=',')
   y_test = scipy.asarray([row for row in csv_reader], dtype=float).squeeze()
```

2.1. Use different kernel functions

For this part, I only use default parameters for linear, polynomial and RBF kernels. The settings array represents each kernel's parameters. The only difference between these settings is that it change the ***-t*** flag to select linear, polynomial or RBF kernel.

```
settings = ['-s 0 -t 1 -b 1 -q', '-s 0 -t 1 -b 1 -q', '-s 0 -t 1 -b 1 -q']
prob = svm_problem(y_train, x_train)
for setting in settings:
   print('Training setting {}'.format(setting))
   param = svm_parameter(setting)
   model = svm_train(prob, param)
   p_label, p_acc, p_val = svm_predict(y_test, x_test, model)
   ACC, MSE, SCC = evaluations(y_test, p_label)
   print("ACC = {} MSE = {} SCC ={}".format(ACC, MSE, SCC))
   print('=====================================================')
```

After running the above code, here is the result of each kernel. The highest accuracy is 77.72% of Polynomial kernel.

	Linear	Polynomial	RBF
Accuracy	73.72%	**77.72%**	72.72%
Mean squared error	1.2296	0.9744	1.2224
Squared correlation coefficient	0.5088	0.5937	0.5137

2.2. C-SVC and GridSearch

In this part, I use C-SVC and implement a grid search to find the best parameters. Below is the default values to use in grid search.

```
s = 0
kernel_type = [0, 1, 2]
cost = np.logspace(-3, 5, base=2, num=5)
gamma = np.logspace(-7, 3, base=2, num=6)
coef0 = [0, 1, 10]
degree = [2, 3, 4]
k_fold = 5
```

The grid search algorithm is implemented as follows. At first, it will loop through each predefined values to create a setting to train. Next, a model is trained based on the defined setting and validate with 5-fold cross validation. After that, the best setting is saved based on the highest accuracy of models.

```
for t in kernel_type:
   for c in cost:
       if t == 0:
           setting = '-s {} -t {} -c {} -v {} -b 1 -q'.format(s, t, c, k_fold)
           print('Training setting {}'.format(setting))
           param = svm_parameter(setting)
           acc = svm_train(prob, param)
           if best_accuracy < acc:
               best_accuracy = acc
               best_setting = setting
           print('=====================================================')
       elif t == 1:
           for g in gamma:
               for d in degree:
                   for r in coef0:
                       setting = '-s {} -t {} -d {} -r {} -c {} -g {} -v {} -b 1 -q'\
                                 .format(s, t, d, r, c, g, k_fold)
                       print('Training setting {}'.format(setting))
                       param = svm_parameter(setting)
                       acc = svm_train(prob, param)
                       if best_accuracy < acc:
                           best_accuracy = acc
                           best_setting = setting
                       print('=====================================================')
       elif t == 2:
           for g in gamma:
               setting = '-s {} -t {} -c {} -g {} -v {} -b 1 -q'.format(s, t, c, g, k_fold)
               print('Training setting {}'.format(setting))
               param = svm_parameter(setting)
               acc = svm_train(prob, param)
               if best_accuracy < acc:
                   best_accuracy = acc
                   best_setting = setting
               print('=====================================================')
```

- Linear kernel
    - Result of finding parameter c, performance on cross-validation.
    - With c = 0.125 gives the best performance (**96.94%**) on cross-validation.
    - The accuracies of models trained with linear kernel fluctuate slightly with different C values.

C	Accuracy
0.125	**96.94%**
0.5	96.78%
2	96.5%
8	96.34%
32	96.46%

- Polynomial kernel
    - There are 4 parameters in polynomial kernel degree (d), coef0 (r), cost (c), gamma (g). Their range values are [2, 3, 4], [0, 1, 10], [0.125, 0.5, 2, 8, 32] and [0.0078125, 0.03125, 0.125, 0.5, 2, 8] corresponding to degree, coef0, cost and gamma.
    - The best result is **98.42%** with setting “-s 0 -t 1 -d 2 -r 1 -c 8.0 -g 2.0 -v 5 -b 1 -q”
    - The table below shows the results of grid search with *d = 2* and *r = 1*

C \ Gamma	0.0078125	0.03125	0.125	0.5	2	8
0.125	96.76%	97.86%	98.2%	98.1%	98.2%	98.32%
0.5	97.36%	97.98%	98.12%	98.16%	98%	98.22%
2	97.64%	97.98%	98.12%	97.98%	98.12%	98.14%
8	97.76%	97.84%	98%	98.18%	**98.42%**	98.2%
32	97.56%	98.08%	98.28%	98.04%	98.3%	97.96%

- RBF kernel
    - Parameters of this kernel are *c* and *gamma*. The values of parameters in grid search are [0.125, 0.5, 2, 8, 32] and [0.0078125, 0.03125, 0.125, 0.5, 2, 8] for c and gamma, respectively.

- The highest accuracy of RBF kernel is **98.74%** corresponding to *c = 32* and *gamma = 0.03125*
- While c and gamma keep increasing the accuracy drops significantly. In the end, with the maximum gamma = 8 in grid search, all models perform badly with 20% of accuracy.

Gamma C	0.0078125	0.03125	0.125	0.5	2	8
0.125	96.42%	97.24%	48.86%	20.08%	20.00%	20.00%
0.5	97.64%	98.20%	75.96%	32.06%	20.00%	20.00%
2	97.88%	98.64%	96.38%	35.30%	24.30%	20.00%
8	98.16%	98.56%	96.18%	33.94%	24.92%	20.00%
32	98.12%	**98.74%**	96.12%	33.46%	24.92%	20.00%

In summary, the best performance in accuracy among all models is 98.74% which belongs to RBF kernel.

2.3. Use linear kernel and RBF kernel together

At first, let implement custom_kernel function and transform the x_train and x_test data using our new kernel. As required, the custom kernel is a combination of linear kernel and RBF kernel then the custom kernel is in a form of:

$$K(x_i,\ x_j) = X_i^T X_j + exp(-\gamma \left\| X_i - X_j \right\|^2)$$

The code below is the implementation of custom kernel. In this code, I used scipy.spatial.distance.squareform, scipy.spatial.distance.cdist, scipy.spatial.distance.pdist to implement the kernel. In addition, the parameters c and gamma are equal to c and gamma of the best result which are 32 and 0.03125, respectively.

```python
def custom_kernel(x_train, x_test):
   gamma = 0.03125
   train_linear_kernel = np.matmul(x_train, np.transpose(x_train))
   train_rbf_kernel = squareform(np.exp(-gamma * pdist(x_train, 'sqeuclidean')))
   x_train_kernel = np.hstack((np.arange(1, 5001).reshape((5000, 1)),
                               np.add(train_linear_kernel, train_rbf_kernel)))

   test_linear_kernel = np.matmul(x_test, np.transpose(x_train))
   test_rbf_kernel = np.exp(-gamma * cdist(x_test, x_train, 'sqeuclidean'))
   x_test_kernel = np.hstack((np.arange(1, 2501).reshape((2500, 1)),
                              np.add(test_linear_kernel, test_rbf_kernel)))

   return x_train_kernel, x_test_kernel
```

To use custom kernel we need to set parameter *isKernel=True* while using svm_problem function then set svm_parameter with t = 4 and c = 32. After that, it is able to train and evaluate model as normal.

```
def svm(x_train_kernel, y_train, x_test_kernel, y_test):
   prob = svm_problem(y_train, x_train_kernel, isKernel=True)
   param = svm_parameter('-q -t 4 -c 32')
   model = svm_train(prob, param)
   svm_predict(y_test, x_test_kernel, model)
```

After the evaluation, the custom kernel's performance is not as good as other kernels. In comparison, custom kernel only achieve 95.08% in accuracy while the best performance of Linear, Polynomial and RBF kernels are 96.94% 98.42% and 98.74%, respectively.

	Linear	Polynomial	RBF	Linear + RBF
Accuracy	96.94%	98.42%	98.74%	95.08%