ML Lab5 Report

I. Gaussian Process

a. code with detailed explanations

Part1:

```
X,y = load_data()
beta = 5
K = kernel(X,X,alpha=1,length_scale=1)
K = K + np.identity(len(X), dtype=np.float64) * (1 / beta)
K_inv = np.linalg.pinv(K)
#print(np.identity(len(X)))
print(K)
```

✓ 0.3s

```
def kernel(X1,X2,sigma=1, alpha=1,length_scale=1):
    '''
    using rational quadratic kernel function: k(x_i, x_j) = (1 + (x_i-x_j)^2 / (2*alpha *
    :param X1: (n) ndarray
    :param X2: (m) ndarray
    return: (n,m)  ndarray
    '''
    square_error = np.power(X1.reshape(-1,1)-X2.reshape(1,-1),2.0)
    # print(square_error)
    kernel = sigma**2 * np.power(1+square_error/(2*alpha*length_scale**2),-alpha)

    return kernel
```

Rational quadratic kernel :

$$k(x_a, x_b) = \sigma^2 \left(1 + \frac{||x_a - x_b||^2}{2\alpha\ell^2}\right)^{-\alpha}$$

Covariance matrix:

$$\mathbf{C}(\mathbf{x}_n, \mathbf{x}_m) = k(\mathbf{x}_n, \mathbf{x}_m) + \beta^{-1}\delta_{nm}$$

In this part, first we load data and build our rational quadratic kernel. We just initialize parameters all with 1.

We use beta=5 in this implementation. Then we add some noise(1/beta) to the kernel to build covariance matrix based on the formula above.

```
x_line=np.linspace(-60,60,num=500)
mean_predict,variance_predict=predict(x_line,X,y,K,beta,alpha=1,length_scale=1)
mean_predict=mean_predict.reshape(-1)
sigma_predict=np.sqrt(np.diag(variance_predict))
```

✓ 0.6s

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

    return means,vars
```

Mean and variance prediction formula:

$$\mu(\mathbf{x}^*) = k(\mathbf{x}, \mathbf{x}^*)^\top \mathbf{C}^{-1} \mathbf{y}$$

$$\sigma^2(\mathbf{x}^*) = k^* - k(\mathbf{x}, \mathbf{x}^*)^\top \mathbf{C}^{-1} k(\mathbf{x}, \mathbf{x}^*)$$

$$k^* = k(\mathbf{x}^*, \mathbf{x}^*) + \beta^{-1}$$

We start to make predictions in this part. First we need to calculate the gram matrix of k(x*, x) and k(x*, x*). After we have these values, then we start to predict the mean

and variance of our testing data based on the formula above. Notice that we should extract the diagonal element of the N by N variance matrix to get the variance of each point in testing data.

```
#plot
plt.plot(X,y,'bo')
plt.plot(x_line,mean_predict,'k-')
plt.fill_between(
    x_line,mean_predict+1.96*sigma_predict,mean_predict-1.96*sigma_predict,facecolor='salmon')
plt.xlim(-60,60)
plt.show()
```

Finally, we visualize our result. 95% confidence interval is about 1.96*sigma. The result will be showed in next section.

Part2:

In this part, we need to use objective function to optimize our hyperparameter in kernel.

Marginal log likelihood:

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

```
def NegativeLogLikelihood(theta, X, Y, beta):
    theta = theta.ravel()
    K = kernel(X, X, theta[0], theta[1], theta[2])
    K += np.identity(len(X), dtype=np.float64) * (1 / beta)
    nll = np.sum(np.log(np.diagonal(cholesky(K))))
    nll += 0.5 * Y.T @ inv(K) @ Y
    nll += 0.5 * len(X) * np.log(2 * np.pi)
    return nll
```

First we define our Negative Loglikelihood function. Notice that we get the determinant by calculate the product of all diagonal element of L(Cholesky(K)).

```
sigma = 1
alpha = 1
length_scale = 1
```
✓ 0.3s Python

```
opt = minimize(NegativeLogLikelihood, [sigma, alpha, length_scale],
                    bounds=((1e-8, 1e6), (1e-8, 1e6), (1e-8, 1e6)),
                    args=(X, y, beta))
sigma_opt = opt.x[0]
alpha_opt = opt.x[1]
length_scale_opt = opt.x[2]
```
✓ 0.2s Python

Then we use scipy.optimize.minimize to minimize our objective function. After minimize, we can get the optimize value of sigma, alpha and length_scale.

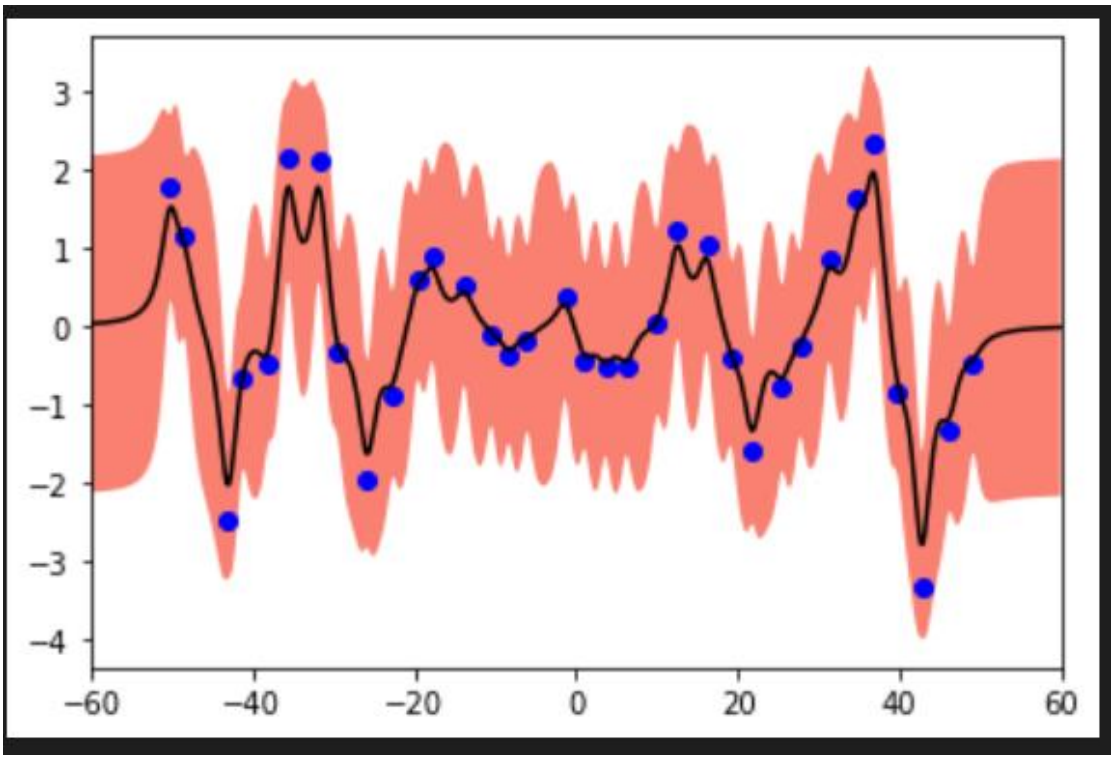
```
# do Gaussian process again
K = kernel(X, X, alpha=alpha_opt, length_scale=length_scale_opt)
K = K + np.identity(len(X), dtype=np.float64) * (1 / beta)
K_inv = np.linalg.pinv(K)
x_line = np.linspace(-60,60,num=500)
mean_predict, variance_predict = \
    predict(x_line, X, y, K, beta, alpha=alpha_opt, length_scale=length_scale_opt)
mean_predict = mean_predict.reshape(-1)
sigma_predict = np.sqrt(np.diag(variance_predict))
#plot
plt.plot(X,y,'bo')
plt.plot(x_line,mean_predict,'k-')
plt.fill_between(x_line,mean_predict+1.96*sigma_predict,\
    mean_predict-1.96*sigma_predict,facecolor='salmon')
plt.xlim(-60,60)
plt.show()
```
✓ 0.1s Python

We can use the optimize values we got and do the same things like part 1. The result will be showed in next section.

b. experiments settings and results

Part1

```
sigma = 1
alpha = 1
length_scale = 1
```
✓ 0.3s



First we initialize the kernel parameters and get the

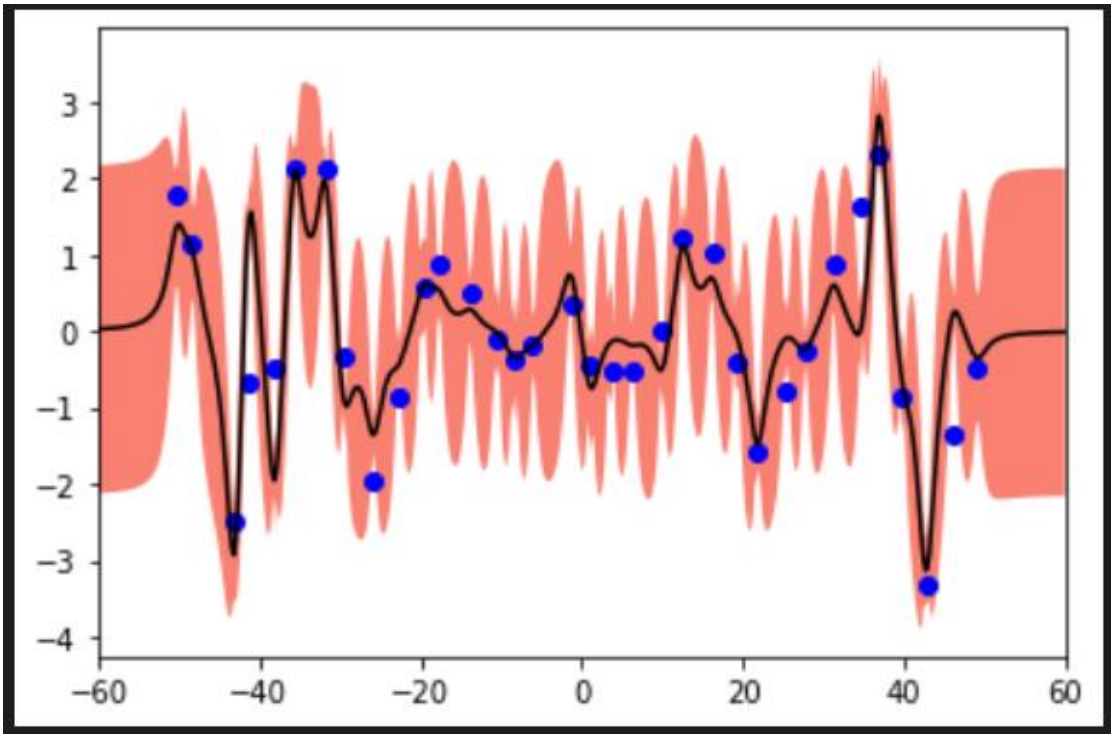
result above.

Part2

```
sigma_opt : 1.313590767915769
alpha_opt :  419.094093326531 55
length_scale_opt : 3.3180125342988815
```

In this part, we use our optimize parameters in kernel

and visualize the prediction result.



We can clearly see that the range of 95% confidence interval after parameter optimization is relatively small.

c. Observations and discussion

I found that under some situation variance calculated can be negative value. My current approach is to take absolute value of them.

II. SVM

a. code with detailed explanations

- Part1

```
def openCSV(filename):
    with open(filename, 'r') as file:
        content = list(csv.reader(file))
        content = np.array(content)
    return content
```

```
X_train = openCSV('data/X_train.csv').astype(np.float64)
Y_train = list(openCSV('data/Y_train.csv').astype(np.int32).ravel())
X_test = openCSV('data/X_test.csv').astype(np.float64)
Y_test = list(openCSV('data/Y_test.csv').astype(np.int32).ravel())
```

First we need to load our Mnist data to program.

$$\text{Linear kernel: } k(x,y) = < x, y >$$

$$\text{Polynomial kernel: } k(x,y) = (< x, y > + c)^d$$

$$\text{Gaussian Radial Basis Function kernel (RBF): }$$

$$k(x,y) = e^{-\sigma \|x-y\|^2}$$

```
# linear kernel
m =  svm_train(Y_train, X_train, f'-t 0 -d 2 -q')
res = svm_predict(Y_test, X_test, m)
# polynomial kernel
m =  svm_train(Y_train, X_train, f'-t 1 -d 2 -q')
res = svm_predict(Y_test, X_test, m)
# RBF kernel
m =  svm_train(Y_train, X_train, f'-t 2 -d 2 -q')
res = svm_predict(Y_test, X_test, m)
```

We then train our model with 3 different kernels above and compare their results. Output is in (b) section.

◯ Part2

```
def gridSearch(X, Y):
    costs = [0.001, 0.01, 0.1, 1, 10]
    gammas = [0.001, 0.01, 0.1, 1]
    cnt = 0
    optimal_cv_acc = 0
    optimal_opt = ''
    for k in kernel:
        for c in costs:
            if k == 'linear':
                opt = f'-t 0 -d 2 -c {c} -q -v 5'
                optimal_cv_acc, optimal_opt = compareAccuracy(Y, X, opt, optimal_cv_acc,
                    optimal_opt)
                cnt += 1
```

```
            elif k == 'polynomial':
                for gamma in gammas:
                    opt = f'-t 1 -d 2 -c {c} -q -v 5 -g {gamma}'
                    optimal_cv_acc, optimal_opt = compareAccuracy(Y, X, opt, optimal_cv_acc,
                        optimal_opt)
                    cnt += 1
            elif k == 'RBF':
                for gamma in gammas:
                    opt = f'-t 2 -d 2 -c {c} -q -v 5 -g {gamma}'
                    optimal_cv_acc, optimal_opt = compareAccuracy(Y, X, opt, optimal_cv_acc,
                        optimal_opt)
                    cnt += 1
```

```
print(f'Total combinations: {cnt}')
print(f'Optimal cross validation accuracy: {optimal_cv_acc}')
print(f'Optimal option: {optimal_opt}')
```

First we need to define our gridSearch function. We iterate over 3 different kernels, 4 different gammas, and 5 different costs value, so we have totally 45 combinations to try.

```
    opt = f"-t 1 -d 2 -c 10 -q -g 1"
    m =  svm_train(Y_train, X_train, opt)
    res = svm_predict(Y_test, X_test, m)
```

Then we use the best combination we found to train a model and validate on testing data.

◯ Part3

```python
def linearKernel(X1, X2):
    kernel = X1 @ X2.T
    return kernel
def RBFKernel(X1, X2, gamma):
    dist = np.sum(X1 ** 2, axis=1).reshape(-1, 1) + np.sum(X2 ** 2, axis=1) - 2 * X1 @ X2.T
    kernel = np.exp((-1 * gamma * dist))
    return kernel
```

First we use the formula showed above to define 2 different kernels.

```python
linear_kernel = linearKernel(X_train, X_train)
RBF_kernel = RBFKernel(X_train, X_train, 1 / 784)
linear_kernel_s = linearKernel(X_train, X_test).T
RBF_kernel_s = RBFKernel(X_train, X_test, 1 / 784).T

X_kernel = np.hstack((np.arange(1, 5001).reshape((-1, 1)), linear_kernel + RBF_kernel))
X_kernel_s = np.hstack((np.arange(1, 2501).reshape((-1, 1)), linear_kernel_s + RBF_kernel_s))
```

We first need to calculate the k(x, x) and k(x, x*) and add index to it. We choose 1/num_features as gamma value based on experience.

```python
opt = '-t 4 -q'
m = svm_train(Y_train, X_kernel, opt)
svm_predict(Y_test, X_kernel_s, m)
```

t=4 means precomputed kernel. We use this opt to train and validate.

b. experiments settings and results

◯ Part1

```
Accuracy = 95.08% (2377/2500) (classification)
Accuracy = 88.24% (2206/2500) (classification)
Accuracy = 95.32% (2383/2500) (classification)
```

- Linear Kernel: 95.08% (2377/2500)

- Polynomial Kernel: 88.24% (2206/2500)

- Radial Basis Function Kernel: 95.32% (2383/2500)

RBF kernel seems the best for this case.

◯ Part2

Output is like below.

```
-t 0 -d 2 -c 0.001 -q -v 5
Cross Validation Accuracy = 95.48%
-t 0 -d 2 -c 0.01 -q -v 5
Cross Validation Accuracy = 97.04%
-t 0 -d 2 -c 0.1 -q -v 5
Cross Validation Accuracy = 96.82%
-t 0 -d 2 -c 1 -q -v 5
Cross Validation Accuracy = 96.26%
-t 0 -d 2 -c 10 -q -v 5
Cross Validation Accuracy = 95.9%
-t 1 -d 2 -c 0.001 -q -v 5 -g 0.001
Cross Validation Accuracy = 45.4%
-t 1 -d 2 -c 0.001 -q -v 5 -g 0.01
Cross Validation Accuracy = 45.36%
-t 1 -d 2 -c 0.001 -q -v 5 -g 0.1
Cross Validation Accuracy = 94.7%
```

```
Cross Validation Accuracy = 98.04%
-t 1 -d 2 -c 0.01 -q -v 5 -g 0.001
Cross Validation Accuracy = 45.64%
-t 1 -d 2 -c 0.01 -q -v 5 -g 0.01
Cross Validation Accuracy = 80.5%
-t 1 -d 2 -c 0.01 -q -v 5 -g 0.1
Cross Validation Accuracy = 97.6%
-t 1 -d 2 -c 0.01 -q -v 5 -g 1
show more (open the raw output data in a text editor) ..
-t 2 -d 2 -c 10 -q -v 5 -g 1
Cross Validation Accuracy = 31.52%
Total combinations: 45
Optimal cross validation accuracy: 98.2
Optimal option: -t 1 -d 2 -c 10 -q -v 5 -g 1
```

After we iterate all these combinations, we can get the best combinations of parameters may be Poly kernel, c=10, gamma=1.

```
Accuracy = 97.68% (2442/2500) (classification)
```

We can get 97.68% accuracy. It does surpass the result in part 1.

◯ Part3

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

This result is good enough but didn't surpass the result in part 2.

c. observations and discussion

Part 1:

The highest accuracy is RBF kernel and lowest is Poly kernel.

Part 2:

An interesting founding is in part 1, Poly kernel has worst result, but after our optimization with gridsearch, The best combination of parameters contains Poly kernel. All in all, as long as there is a good combination of parameters, every kernel should be able to achieve good results.

Part 3:

The result of combination of linear kernel and RBF kernel is same as linear kernel in part 1. It's because maximum of RBF kernel is 1, but linear kernel can be very large. Because of the linear kernel dominate addition result, the result of this part is same as part 1. To prove my idea, I try to first normalize linear kernel to (0, 1) then add RBF kernel. I got 95.88% of accuracy which really is different with previous results.