Machine Learning
HW5-Gaussian Process & SVM

Yu-Ju Tsai, 311652010

May 12, 2023

1 Gaussian Proces

1.1 Code with detailed explanations

First I load dataset

```
def load_data(path):
    X = []
    Y = []
    with open(path, 'r') as f:
        for line in f.readlines():
            x, y = line.split(' ')
            X.append(float(x))
            Y.append(float(y))
    X = np.asarray(X)
    Y = np.asarray(Y)
    return X, Y
```

Figure 1: Load Data

- Kernel:
  I defined rational quadratic function according to the definition.
  $k(x_i, x_j) = \sigma(1 + \frac{(x_i - x_j)^2}{2\alpha l^2})^{-\alpha}$,
  and the initial value of $\alpha = 1$, $\sigma = 1$, length scale $= 1$, $\beta = 5$

```
def kernel(X1, X2, sigma, alpha, length_scale):

    # rational quadratic kernel function: k(x_i, x_j) = sigma*(1 + (x_i-x_j)^2 / (2*alpha * length_scale^2)) ^ -alpha
    kernel = sigma * np.power(1 + np.power(X1.reshape(-1,1) - X2.reshape(1,-1), 2) / (2 * alpha * length_scale ** 2), -alpha)
    return kernel
```

Figure 2: rational quadratic function

- Gaussian Process:
  Based on the lecture, C represents a set of kernels that capture the feature distances between all training data points and euth some noise, and k_x_xstar measures the similarity between training and testing data points. Additionally, k_xstar_xstar represents the kernel distance between all testing data points . Beta denotes the level of variation in the noise, With the aforementioned components, we can now calculate the mean and variance.
  $\mu = k(x, x^*)^T C^{-1} y$
  $\sigma = k(x^*, x^*) + \frac{1}{\beta} - k(x, x^*) C^{-1} k(x, x^*)$

```python
def predict(x_line, X, y, C, beta, sigma, alpha, length_scale):

    # K(X, X*): 34 X 500
    k_x_xstar = kernel(X, x_line, sigma, alpha, length_scale)

    # K(X*, X*): 500 X 500
    k_xstar_xstar = kernel(x_line, x_line, sigma, alpha, length_scale)

    # mean = k(x,x*)^T C^(-1) y
    means = k_x_xstar.T @ np.linalg.inv(C) @ y.reshape(-1,1) # 500 x 1

    # var = k(x*,x*) + 1/beta - k(x,x*) C^(-1) k(x,x*)
    var = k_xstar_xstar + (1/beta) * np.identity(len(k_xstar_xstar)) - k_x_xstar.T @ np.linalg.inv(C) @ k_x_xstar # 500 x 500

    return means, var
```

Figure 3: Gaussian Process

Subsequently, I compute the mean and variance using the following equations.

- Optimization:
  I use function minimize that in scipy.optimize for optimization, and the objective function is f(σ, α, l) $= -\frac{1}{2}y^T\alpha - \Sigma_i log(L_{ii}) - \frac{N}{2}log(2\pi)$, and every parameters is bounded by (1e-5, 1e5).

```python
def objective_function(theta, X, y, beta):

    theta = theta.ravel()
    K = kernel(X, X, sigma=theta[0], alpha=theta[1], length_scale=theta[2]) + (1/beta) * np.identity(len(X))
    L = np.linalg.cholesky(K) # K = LL^T
    nll = 0.5 * y.reshape(1,-1) @ np.linalg.inv(K) @ y.reshape(-1,1) + np.sum(np.log(np.diag(L))) + 0.5 * len(X) * np.log(2*np.pi)
    return nll.item()
```

Figure 4: Optimization

- Main function:

```python
############### part 1 ################

# initial value
X, y = load_data(path='input.data')    # X: 34, y: 34
beta = 5
sigma = 1
alpha = 1
length_scale = 1

# convariance C(Xn, Xm)
C = kernel(X, X, sigma, alpha, length_scale) + 1 / beta * np.identity(len(X))  # C(Xn, Xm): 34 x 34

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

# plot
plt.plot(X, y, 'bo')
plt.plot(x_line, mean_predict,'k-')

# 95% confidence interval: +- 2*variance_predict
plt.fill_between(x_line, mean_predict + 2 * variance_predict, mean_predict - 2 * variance_predict, facecolor='salmon')
plt.xlim(-60, 60)
plt.savefig("GP.jpg")
# plt.show()
```

Figure 5: Part 1

```
############### part 2 ################

opt = minimize(objective_function, x0=[sigma, alpha, length_scale], bounds=((1e-5, 1e5), (1e-5, 1e5), (1e-5, 1e5)), args=(X, y, beta))
sigma_opt = opt.x[0]
alpha_opt = opt.x[1]
length_scale_opt = opt.x[2]
print(f"sigma_opt: {sigma_opt}")
print(f"alpha_opt: {alpha_opt}")
print(f"length_scale_opt: {length_scale_opt}")

# convariance C(Xn, Xm)
C = kernel(X, X, sigma=sigma_opt, alpha=alpha_opt, length_scale = length_scale_opt) + 1 / beta * np.identity(len(X))  # C(Xn, Xm): 34 x

x_line = np.linspace(-60, 60, num=500)
mean_predict, variance_predict = predict(x_line, X, y, C, beta, sigma=sigma_opt, alpha=alpha_opt, length_scale=length_scale_opt)
mean_predict = mean_predict.reshape(-1)
variance_predict = np.sqrt(np.diag(variance_predict))

# plot
fig = plt.figure()
plt.plot(X, y, 'bo')
plt.plot(x_line, mean_predict,'k-')

# 95% confidence interval: +- 2*variance_predict
plt.fill_between(x_line, mean_predict + 2 * variance_predict, mean_predict - 2 * variance_predict, facecolor='salmon')
plt.xlim(-60, 60)
plt.savefig("opt_GP.jpg")
# plt.show()
```

Figure 6: Part 2

1.2 Experiments settings and results

1. Random picked parameters
   $\alpha = 1$, length scale $= 1$, $\sigma = 1$

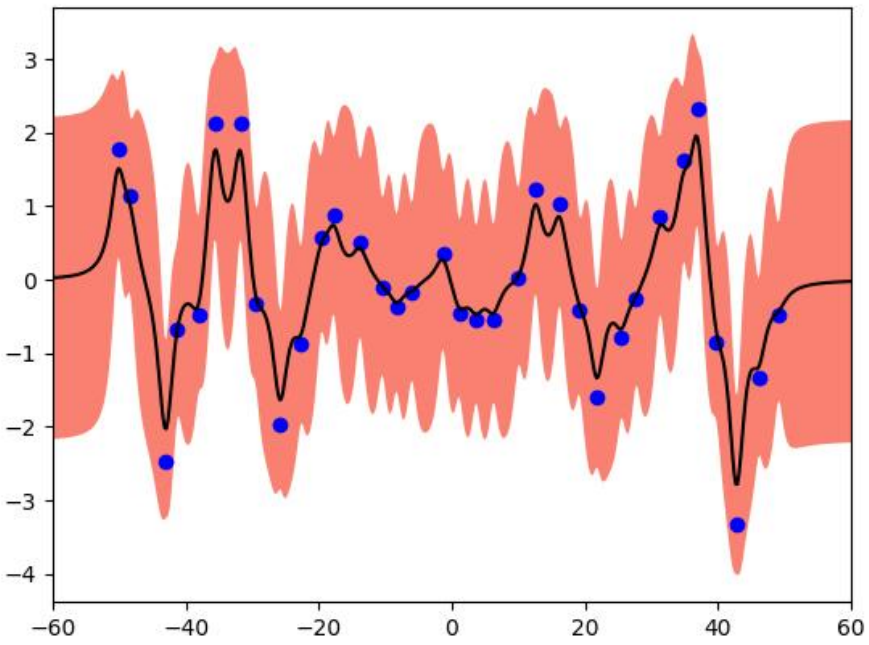

Figure 7: Original

2. Optimized parameters by minimizing negative marginal log-likelihood
   $\alpha = 447.36$, length scale $= 1.72$, $\sigma = 3.317$

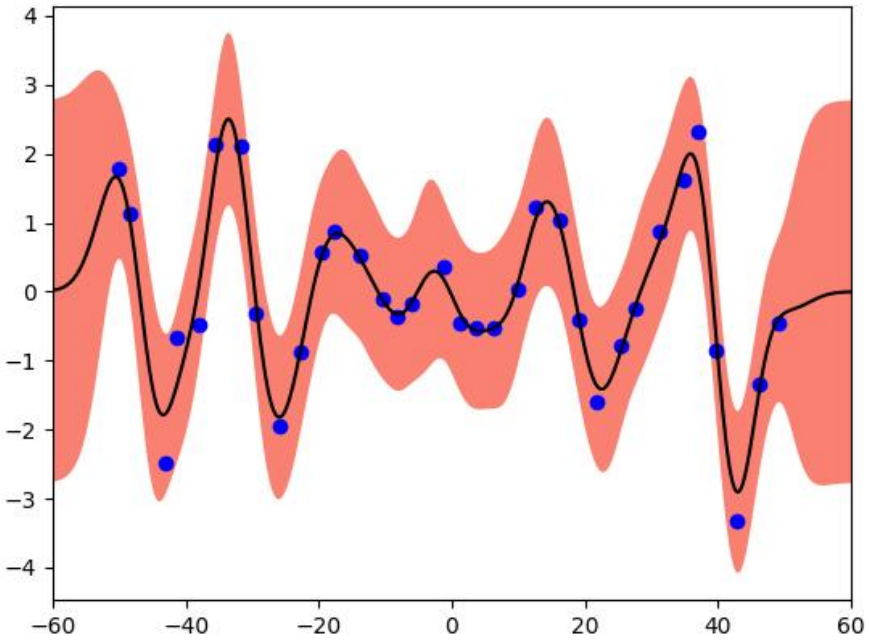

Figure 8: Optimization

1.3 Observations and discussion

1. Result 2 clearly outperforms result 1, as shown in Figure 7 8.

2. From the results, it is evident that Gaussian process exhibits higher confidence in predicting data points within the training set. It can still make reasonable estimations among the training data points. However, when it comes to regions without any training data, Gaussian process struggles to make accurate predictions. As a result, the 95% confidence region appears much larger in those areas.

3. Poor parameter choices can lead to highly unfavorable outcomes. The worst-case scenario of overfitting involves fitting multiple impulse functions to the training data, as shown in Figure 7. While this may result in a good fit to the training set, it becomes evident that such a function does not represent the desired general behavior.

2 SVM

2.1 Code with detailed explanations

First I read MNIST as my dataset

```
def load(path, type):
    re = []
    if type == 'image':
        with open(path) as csvfile:
            rows = csv.reader(csvfile)
            for row in rows:
                re.append([float(v) for v in row])
        re = np.asarray(re, dtype='float')

    if type == 'label':
        with open(path) as csvfile:
            rows = csv.reader(csvfile)
            for row in rows:
                re.append(int(row[0]))
        re = np.asarray(re, dtype=int)
    return re
```

Figure 9: Load Data

- Part 1:
  -q: quiet mode (no outputs)
  -t kernel type: 0 means linear, 1 means polynomial , 2 means radial basis function, and 4 means precomputed kernel.
  svm predict return three values (label:a list of predicted labels, acc: a tuple including accuracy , mean-squared error, and squared correlation coefficient , vals: a list of decision values or probability estimates), and I use different kernel functions to do the SVM.

```
############### Part 1 ###############

X_train = load('X_train.csv', 'image')  # (5000,784)
Y_train = load('Y_train.csv', 'label')  # (5000, )
X_test = load('X_test.csv', 'image')    # (2500,784)
Y_test = load('Y_test.csv', 'label')    # (2500, )
kernel_param = ['-t 0', '-t 1', '-t 2'] # linear, polynomial, RBF
accuracy = []

for param in kernel_param:
    model = svm_train(Y_train, X_train, '-q ' + param)
    p_label, p_acc, p_vals = svm_predict(Y_test, X_test, model, '-q')
    accuracy.append(p_acc[0]) # p_acc: (accuracy, mse, scc)

print(f"linear kernel accuracy: {accuracy[0]: .2f}%")
print(f"polynomial kernel accuracy: {accuracy[1]: .2f}%")
print(f"radial basis function kernel accuracy: {accuracy[2]: .2f}%")
```

Figure 10: Part 1

- Part 2:
  I have created a custom GridSearch function to handle the entire process of grid search, as shown in Figure 11. Within the GridSearch() function, I first define a set of parameters. The grid search process iterates through all these parameters to find the optimal combination. For each kernel type, my code simply iterates through all the defined parameters and performs cross-validation. It also keeps track of the best accuracy and corresponding parameters while iterating through all the parameter combinations.

  Some parameters:

  - c cost: set the parameter C of C-SVC
  - g gamma: set gamma in kernel function (default 1/num features)
  - v n: n-fold cross validation mode

  If '-v' is specified in 'options' either accuracy (ACC) or mean-squared error (MSE) is returned, and I select the best accuracy as the parameter for my prediction, By removing the -v 3 option, the svm_train function will return the trained model instead of the cross-validation accuracy, and use the model to predict the test image label.

  In the following image Figure 11, I have provided the approach for the linear kernel, the approach for the Polynomial kernel is the same as linear kernel. Please note that if the kernel is RBF, an additional parameter γ will be included.

```python
def GridSearch(kernel_type, X_train, y_train, X_test, y_test):
    costs = [1e-4, 1e-3, 1e-2, 0.1, 1, 10, 100, 1000, 10000]
    gammas = [1/784, 1, 1e-1, 1e-2, 1e-3, 1e-4]
    max_acc = 0.0

    if kernel_type == 'Linear':
        for cost in costs:
            param = "-t 0 -c {} -v 3 -q".format(cost)
            ACC = svm_train(y_train, X_train, param)
            if ACC > max_acc:
                max_acc = ACC
                max_param = param

        param_list = max_param.split()
        if "-v" in param_list:
            idx = param_list.index("-v")
            param_list.pop(idx)
            param_list.pop(idx)
        max_param = " ".join(param_list)
        model = svm_train(y_train, X_train, max_param)
        _, accuracy, _ = svm_predict(y_test, X_test, model)
```

Figure 11: Grid Search

and run the Grid Search for three cases, see Figure 12

```python
############### Part 2 ###############

GridSearch('Linear', X_train, Y_train, X_test, Y_test)
GridSearch('Polynomial', X_train, Y_train, X_test, Y_test)
GridSearch('RBF', X_train, Y_train, X_test, Y_test)
```

Figure 12: Part 2

- Part 3:
  Self defined kernel function, Linear kernel and RBF kernel.

```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
```

Figure 13: kernel

and I define Linear + RBF kernel as figure 14

```python
def mixture_kernel(X1, X2, gamma):
    kernel_linear = linearKernel(X1, X2)
    kernel_RBF = RBFKernel(X1, X2, gamma)
    kernel = kernel_linear + kernel_RBF
    kernel = np.hstack((np.arange(1, len(X1) + 1).reshape(-1,1), kernel))
    return kernel
```

Figure 14: linear + RBF kernel

Libsvm provides support for user-defined kernels. Instead of directly inputting the training data, we need to input pre-calculated data in the kernel space and set the parameter isKernel to True, see figure 15.

```python
############### Part 3 ###############

kernel_train = mixture_kernel(X_train, X_train, 1/784)
prob = svm_problem(Y_train, kernel_train, isKernel = True)
param = svm_parameter('-q -t 4')
model = svm_train(prob, param)

kernel_test = mixture_kernel(X_test, X_train, 1/784)
p_label, p_acc, p_vals = svm_predict(Y_test, kernel_test, model, '-q')
print(f"linear kernel + RBF kernel accuracy: {p_acc[0]: .2f}%")
```

Figure 15: Part 3

2.2 Experiments settings and results

The results in Figure 16 17, and 18 represent the outcomes obtained using three different kernel types, each with varying values of costs and gamma.

- Linear kernel

Costs	Cross Validation Accuracy (%)
1e-4	88.54
1e-3	95.3
1e-2	96.88
0.1	96.9
1	96.16
10	96.38
100	96.06
1000	95.94
10000	96.24

Figure 16: Linear kernel result

- Polynoimal kernel

Costs	Cross Validation Accuracy (%)
1e-4	29.04
1e-3	28.52
1e-2	28.7
0.1	28.38
1	31.12
10	73.6
100	92.28
1000	97.08
10000	**97.5**

Figure 17: Polynoimal kernel result

- RBF kernel

Costs\gamma	1/784	1	0.1	1e-2	1e-3	1e-4
1e-4	81.26	20.56	49.24	89.76	81.06	79.88
1e-3	81.58	20.62	49.96	89.92	81.02	79.92
1e-2	81.46	20.52	49.02	91.82	81.16	79.7
0.1	92.7	20.58	52.82	96.16	91.82	79.74
1	96.16	29.76	91.06	97.7	96.12	91.88
10	97.22	31.66	91.5	**98.24**	97.16	95.9
100	97.24	32.26	91.4	98.14	96.86	96.74
1000	97.02	31.1	91.72	98.12	97.06	96.56
10000	96.82	31.9	91.66	98.16	96.84	96.18

Figure 18: RBF kernel result

Part 1:			
Kernel		Testing Accuracy (%)	Parameters
Linear		95.08	-q -t 0
Polynomial		34.68	-q -t 1
RBF		95.32	-q -t 2
Part 2:			
Kernel	Cross-validation Accuracy (%)	Testing Accuracy (%)	Parameters
Linear	96.9	95.8	-q -t 0 -v 3 -c 0.1
Polynomial	97.5	97.4	-q -t 1 -v 3 -c 10000
RBF	98.24	98.2	-q -t 2 -v 3 -c 10 -g 0.01
Part 3:			
Kernel		Testing Accuracy (%)	Parameters
Linear+RBF		95.08	-q -t 4

Table 1: Consolidate the accuracy of Part 1, Part 2, and Part 3

2.3 Observations and discussion

1. RBF kernel achieves the highest accuracy.
2. From Part 1 and Part 2, we can see the importance of finding good parameters.
3. The polynomial kernel consumes the most time during the search process because it requires fine-tuning of multiple parameters.
4. C and gamma are two most important parameters to fine-tuned. C is the cost. The higher c value is, the less tolerance the model has to error. This results in too fit to training data. As to gamma, it determine the space RBF can project to. If gamma is smaller, it can support more vectors.
5. Both the Linear + RBF kernel in part 3 and the Linear kernel, RBF kernel in part 1 achieve similar levels of accuracy.