HW5 report

309552026

鄭偉丞

Question 1:

a. Code explanations:

Part 1:

```
def Gaussian(points, beta, sigma, alpha, l):
    Xs = points.T[0].reshape(-1, 1)
    C = kernel(Xs, Xs.T, sigma, alpha, l)
    C += np.eye(len(Xs)) / beta
    return C

def kernel(x1, x2, sigma, alpha, l):
    return sigma * (1 + np.power(x1 - x2, 2) / (2 * alpha * l ** 2)) ** (-alpha)
```

The function Gaussian generates the covariance matrix for training data, just as mentioned in class { $C(x_n, x_m) = k(x_n, x_m) + \beta^{-1}\delta_{nm}$ }, for $\beta = 5$. Function kernel is rational quadratic kernel $k(x_1, x_2) = \sigma^2 \left(1 + e^{\frac{\|x_1 - x_2\|^2}{2\alpha l^2}}\right)^{-\alpha}$. The parameters represent, $\text{sigma} = \sigma^2, \text{alpha} = \alpha, l = l$ respectively. In part 1, without optimizing, $\text{sigma} = \text{alpha} = l = 1$ as below.

```
if __name__ == "__main__":
    points = load("./data/input.data")

    beta = 5
    sigma = 1
    alpha = 1
    l = 1

    C = Gaussian(points, beta, sigma, alpha, l)
```

```
def drawPlot(axs, title, points, C, beta, sigma, alpha, l):
    axs.set_title(title)
    axs.set_xlim(-60, 60)

    Xs = np.linspace(-60, 60, 1000)
    Ys_mean = np.zeros(1000)
    Ys_var = np.zeros(1000)

    C_inv = np.linalg.inv(C)
    for i in range(1000):
        kT = kernel(points.T[0], Xs[i], sigma, alpha, l)
        Ys_mean[i] = kT @ C_inv @ points.T[1].T
        Ys_var[i] = np.abs((kernel(Xs[i], Xs[i], sigma, alpha, l) + 1/beta) - kT @ C_inv @ kT.T) ** 0.5

    axs.plot(points.T[0], points.T[1], 'bo', markersize=5)
    axs.plot(Xs, Ys_mean, color='black')
    axs.fill_between(Xs, Ys_mean + 2 * Ys_var, Ys_mean - 2 * Ys_var, color='pink')
    pyplot.draw()
```

The predict part is defined in function drawPlot, especially the highlight part. We

need to predict means and variance of y for all points within x=[-60,60]. The prediction format is $\begin{cases} \mu(x^*) = k(x,x^*)^T C^{-1} y \\ \sigma^2(x^2) = k^* - k(x,x^*)^T C^{-1} k(x,x^*) \\ k^* = k(x^*,x^*) + \beta^{-1} \end{cases}$, for $\begin{cases} C_{inv} = C^{-1} \\ kT = k(x,x^*)^T \rightarrow k = kT.T \end{cases}$. Since we need to mark the 95% confidence interval, which within 2 standard deviation from mean, there's a ^0.5 at the end of var to get the standard deviation for each x.

Part 2:

In this part, the functions for Gaussian process, kernel and prediction are the same as part 1.

```
opt = minimize(object_function, [sigma, alpha, l],
               bounds=((1e-8, 1e6), (1e-8, 1e6), (1e-8, 1e6)),
               args=(points, beta))
sigma_opt = opt.x[0]
alpha_opt = opt.x[1]
l_opt = opt.x[2]

C_opt = Gaussian(points, beta, sigma_opt, alpha_opt, l_opt)
```

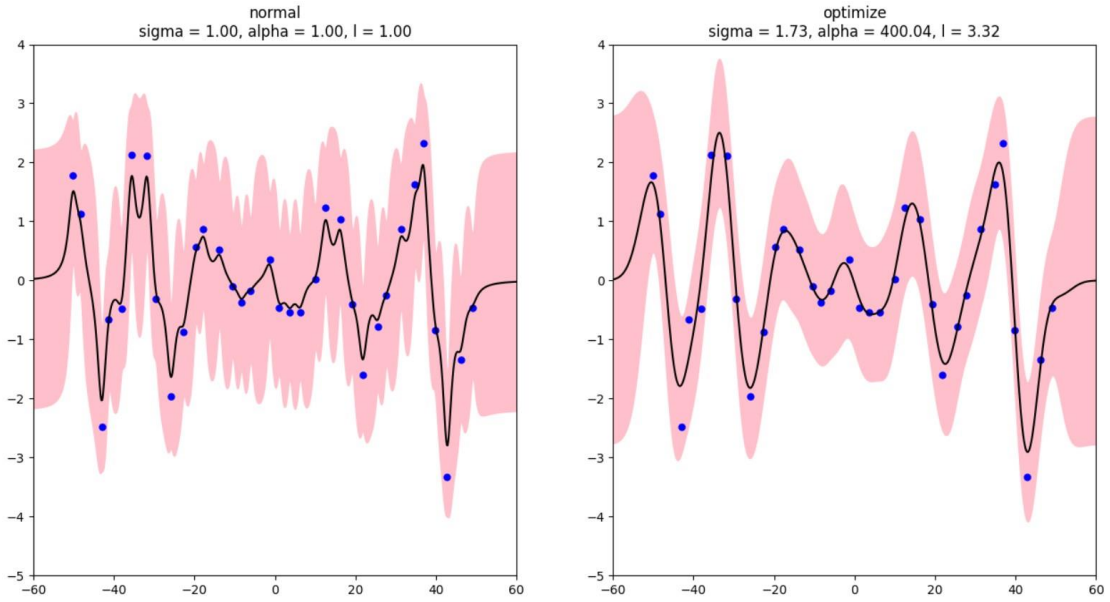
For optimizing the parameters (sigma, alpha and l), I use scipy.optimize.minimize to find the parameters minimize the object_function as below. And set bounds = $(10^{-8}, 10^6)$ for each parameter.

```
def object_function(theta, points, beta):
    theta = theta.ravel()
    C = Gaussian(points, beta, theta[0], theta[1], theta[2])
    Ys = points.T[1].reshape(-1, 1)
    target = 0.5 * np.log(2*np.pi) * len(points)
    target += 0.5 * Ys.T @ np.linalg.inv(C) @ Ys
    target += 0.5 * np.log(np.linalg.det(C))
    return target.ravel()
```

The object function is negative marginal log-likelihood function $\frac{\mathrm{N}}{2}\ln(2\pi) + \frac{1}{2}y^T C^{-1} y + \frac{1}{2}\ln|\mathrm{C}|$, which C is the covariance matrix from Gaussian process. After optimization, do Gaussian again with optimized parameters to get optimized covariance matrix.

Finally, draw plot and test with optimized parameters.

b. Result:



c. Observation:

With optimization, the curve and the 95% confidence interval become smoother and fit the data better, avoid the curve from overfitting. The variance at some place is smaller, but some place, especially the boundary, get a little bit larger than before. It may because there's no training data.

Question 2:

Part 1:

a. Code explanations:

```
linear = '0'
polynomial = '1'
RBF = '2'
```

```
# training
train = svm_problem(Y_train, X_train)
linear_param = svm_parameter('-q -t ' + linear)
polynomail_param = svm_parameter('-q -t ' + polynomial)
RBF_param = svm_parameter('-q -t ' + RBF)
linear_model = svm_train(train, linear_param)
polynomial_model = svm_train(train, polynomail_param)
RBF_model = svm_train(train, RBF_param)
```

1. Use svm_problem to generate the training data with label.
2. Use svm_parameter to set arguments for training, -q means don't show message during training, -t means set kernel in svm, the value of the 3 kernels are as above (0, 1, 2 for linear, polynomial, RBF respectively). Here uses default SVM

(argument -c 0, which is C-SVC). Since here are 4 classes, the normal one class SVM cannot classify them.

3. Do training by svm_train for each kernel and get the models.

```
# testing
print("linear kernel:")
svm_predict(Y_test, X_test, linear_model)
print("\npolynomial kernel:")
svm_predict(Y_test, X_test, polynomial_model)
print("\nRBF kernel:")
svm_predict(Y_test, X_test, RBF_model)
```

Set different model as parameter in svm_predict to do testing. This function will print the accuracy.

b. Result:

```
linear kernel:
Accuracy = 95.08% (2377/2500) (classification)

polynomial kernel:
Accuracy = 34.68% (867/2500) (classification)

RBF kernel:
Accuracy = 95.32% (2383/2500) (classification)
```

c. Observation:

Linear and RBF kernel have high accuracy. Polynomial kernel with default parameters doesn't perform well.

Part 2:

a. Code explanations:

```
# set parameters
linear = 0
polynomial = 1
RBF = 2
log2c = log2g = [i-10 for i in range(0, 21, 2)]
```

```
# create grid search result
train = svm_problem(Y_train, X_train)

best, best_log2c, _, acc, p_acc = svm_grid_search(log2c, [0], train, linear, Y_test, X_test)
resultFile.write("============ linear kernel ============\n")
resultFile.write(f"best:       {best}\n")
resultFile.write(f"best_log2c: {best_log2c}\n")
resultFile.write(f"p_acc:      {p_acc}%\n")
resultFile.write(f"acc:\n{acc}\n")
resultFile.write("=======================================\n\n")

drawTable((6, 3), "linear", linear, acc, log2c, log2g)
```

In main, set parameters first, log2c and log2g represent the power of C and γ to do grid search with value [-10, -8, -6, -4, -2, 0, 2, 4, 6, 8, 10]. Then set training data and do grid search for different type of kernel. Write the results into file and show the grid table.

```
def svm_grid_search(log2c, log2g, train, kernel, Y_test, X_test):
    best = 0.0
    best_log2c = 0
    best_log2g = 0

    # search best params
    acc = np.zeros((len(log2g), len(log2c)), dtype=float)
    for i in range(len(log2g)):
        for j in range(len(log2c)):
            param = f"-q -t {kernel} -v 3 -c {2**log2c[j]}"
            if (kernel != 0):
                param += f" -g {2**log2g[i]}"
            model = svm_train(train, param)
            acc[i][j] = round(model, 2)
            if (best < model):
                best = model
                best_log2g = log2g[i]
                best_log2c = log2c[j]
    # test
    param = f"-q -t {kernel} -c {2**best_log2c}"
    if (kernel != 0):
        param += f" -g {2**best_log2g}"
    model = svm_train(train, param)
    _, p_acc, _ = svm_predict(Y_test, X_test, model)
    return best, best_log2c, best_log2g, acc, p_acc[0]
```

The grid search performs cross-validation with amount of folds = 3 on different parameter pair (C, γ). Record accuracy for each pair and the parameter pair with the best accuracy. Arguments -q, -t has same meaning mentioned in Part 1. -c and -g mean the parameters (C, γ). Linear kernel doesn't have parameter γ, so only set -g when kernel = 1 or 2 (polynomial or RBF). -v means the amount of folds in cross-validation. Normally it will set 10, but it takes too long to get result and -v 3 is good enough to see the difference between different (C, γ).

After training, use the best pair (C, γ) to do testing and return all values we need.

b. Result:

linear

log2c=-10	log2c=-8	log2c=-6	log2c=-4	log2c=-2	log2c=0	log2c=2	log2c=4	log2c=6	log2c=8	log2c=10
95.14	96.34	96.9	96.86	96.44	96.38	96.06	95.94	96.24	96.3	96.48

```
============ linear kernel ============
best:       96.89999999999999
best_log2c: -6
p_acc:      95.92%
acc:
[[95.14 96.34 96.9  96.86 96.44 96.38 96.06 95.94 96.24 96.3  96.48]]
=======================================
```

polynomial

	log2c=-10	log2c=-8	log2c=-6	log2c=-4	log2c=-2	log2c=0	log2c=2	log2c=4	log2c=6	log2c=8	log2c=10
log2g=-10	28.7	28.38	28.56	28.36	28.68	28.38	35.66	67.26	85.16	92.66	96.14
log2g=-8	28.64	28.6	28.48	35.74	67.42	85.18	92.68	96.14	97.52	97.54	97.66
log2g=-6	35.78	67.4	85.02	92.7	95.88	97.24	97.46	97.46	97.44	97.4	97.34
log2g=-4	92.88	96.16	97.14	97.72	97.32	97.3	97.44	97.68	97.36	97.38	97.56
log2g=-2	97.5	97.34	97.42	97.44	97.3	97.54	97.4	97.4	97.4	97.68	97.36
log2g=0	97.28	97.3	97.52	97.34	97.46	97.42	97.46	97.52	97.42	97.18	97.34
log2g=2	97.38	97.4	97.42	97.4	97.6	97.28	97.32	97.46	97.7	97.32	97.44
log2g=4	97.28	97.5	97.44	97.52	97.36	97.4	97.64	97.3	97.36	97.32	97.36
log2g=6	97.42	97.34	97.4	97.28	97.48	97.3	97.7	97.3	97.6	97.28	97.56
log2g=8	97.52	97.4	97.8	97.7	97.34	97.34	97.48	97.52	97.4	97.38	97.6
log2g=10	97.34	97.34	97.44	97.44	97.64	97.34	97.46	97.58	97.34	97.6	97.6

```
========== polynomial kernel ==========
best:       97.8
best_log2c: -6
best_log2g: 8
p_acc:      97.48%
acc:
[[28.7  28.38 28.56 28.36 28.68 28.38 35.66 67.26 85.16 92.66 96.14]
 [28.64 28.6  28.48 35.74 67.42 85.18 92.68 96.14 97.52 97.54 97.66]
 [35.78 67.4  85.02 92.7  95.88 97.24 97.46 97.46 97.44 97.4  97.34]
 [92.88 96.16 97.14 97.72 97.32 97.3  97.44 97.68 97.36 97.38 97.56]
 [97.5  97.34 97.42 97.44 97.3  97.54 97.4  97.4  97.4  97.68 97.36]
 [97.28 97.3  97.52 97.34 97.46 97.42 97.46 97.52 97.42 97.18 97.34]
 [97.38 97.4  97.42 97.4  97.6  97.28 97.32 97.46 97.7  97.32 97.44]
 [97.28 97.5  97.44 97.52 97.36 97.4  97.64 97.3  97.36 97.32 97.36]
 [97.42 97.34 97.4  97.28 97.48 97.3  97.7  97.3  97.6  97.28 97.56]
 [97.52 97.4  97.8  97.7  97.34 97.34 97.48 97.52 97.4  97.38 97.6 ]
 [97.34 97.34 97.44 97.44 97.64 97.34 97.46 97.58 97.34 97.6  97.6 ]]
=======================================
```

RBF

	log2c=-10	log2c=-8	log2c=-6	log2c=-4	log2c=-2	log2c=0	log2c=2	log2c=4	log2c=6	log2c=8	log2c=10
log2g=-10	81.06	81.1	80.82	89.46	94.2	95.86	97.02	97.18	97.08	96.86	97.0
log2g=-8	84.92	84.7	89.26	94.08	96.04	97.08	97.6	97.78	97.74	97.82	97.42
log2g=-6	92.36	92.36	94.08	95.98	97.42	98.08	98.4	98.38	98.22	98.48	98.34
log2g=-4	70.8	71.0	70.86	74.78	92.86	97.66	97.86	97.88	97.66	97.8	97.9
log2g=-2	28.02	26.92	28.02	27.02	33.74	62.32	65.16	65.14	64.82	64.92	64.5
log2g=0	20.72	20.9	20.62	20.88	20.56	29.76	30.96	32.5	30.92	33.84	30.58
log2g=2	39.84	39.66	45.96	39.62	39.58	21.84	22.16	22.06	21.8	22.54	21.62
log2g=4	75.04	75.26	75.44	75.64	75.36	68.94	62.7	62.92	68.92	62.64	62.64
log2g=6	38.8	39.0	38.82	38.66	38.84	20.32	20.28	20.42	20.28	20.54	20.5
log2g=8	22.78	23.0	22.72	22.94	22.98	23.06	22.98	22.98	23.1	22.84	22.62
log2g=10	20.0	20.0	20.0	20.0	20.0	20.0	20.0	20.0	20.0	20.0	20.0

```
============ RBF  kernel =============
best:       98.48
best_log2c: 8
best_log2g: -6
p_acc:      98.44000000000001%
acc:
[[81.06 81.1  80.82 89.46 94.2  95.86 97.02 97.18 97.08 96.86 97.  ]
 [84.92 84.7  89.26 94.08 96.04 97.08 97.6  97.78 97.74 97.82 97.42]
 [92.36 92.36 94.08 95.98 97.42 98.08 98.4  98.38 98.22 98.48 98.34]
 [70.8  71.   70.86 74.78 92.86 97.66 97.86 97.88 97.66 97.8  97.9 ]
 [28.02 26.92 28.02 27.02 33.74 62.32 65.16 65.14 64.82 64.92 64.5 ]
 [20.72 20.9  20.62 20.88 20.56 29.76 30.96 32.5  30.92 33.84 30.58]
 [39.84 39.66 45.96 39.62 39.58 21.84 22.16 22.06 21.8  22.54 21.62]
 [75.04 75.26 75.44 75.64 75.36 68.94 62.7  62.92 68.92 62.64 62.64]
 [38.8  39.   38.82 38.66 38.84 20.32 20.28 20.42 20.28 20.54 20.5 ]
 [22.78 23.   22.72 22.94 22.98 23.06 22.98 22.98 23.1  22.84 22.62]
 [20.   20.   20.   20.   20.   20.   20.   20.   20.   20.   20.  ]]
=======================================
```

c. Observation:

- The linear kernel has high performance no matter which c.
- The polynomial kernel has high performance on large g and c.
- The RBF kernel has high performance on large c and small g, but the performance will decrease when g is too small. The interesting thing is, although the RBF kernel doesn't perform well on g = 2^4, it still better than its neighbor $2^0 \sim 2^2$ *and* $2^6 \sim 2^{10}$.
- No matter which kernel, it will have high performance on suitable parameters.

Part 3:

a. Code explanations:

```
def new_kernel(x1, x2, gamma):
    data = np.zeros((len(x1), len(x2) + 1))
    linear = x1 @ x2.T
    RBF = np.exp(- gamma * cdist(x1, x2, 'sqeuclidean'))
    data[:, 1:] = linear + RBF
    data[:, :1] = np.arange(len(x1))[:, np.newaxis]+1
    return data
```

The function new_kernel combine the linear kernel and RBF kernel. Then transform the data into the input form svm_train need <label, <index: data>>.

```
# do grid search to see the difference between different parameters
best = 0.0
best_log2c = 0
best_log2g = 0

# search best params
acc = np.zeros((len(log2g), len(log2c)), dtype=float)
for i in range(len(log2g)):
    for j in range(len(log2c)):
        param = f"-q -t 4 -v 3 -c {2**log2c[j]} -g {2**log2g[i]}"
        data = new_kernel(np.array(X_train), np.array(X_train), 2**log2g[i])
        model = svm_train(Y_train, [list(row) for row in data], param)
        acc[i][j] = round(model, 2)
        if (best < model):
            best = model
            best_log2g = log2g[i]
            best_log2c = log2c[j]
# test
param = f"-q -t 4 -c {2**best_log2c} -g {2**best_log2g}"
best_train_data = new_kernel(np.array(X_train), np.array(X_train), 2**best_log2g)
model = svm_train(Y_train, [list(row) for row in best_train_data], param)
test_data = new_kernel(np.array(X_test), np.array(X_train), 2**best_log2g)
_, p_acc, _ = svm_predict(Y_test, [list(row) for row in test_data], model)
```

This part in main do grid search the same as part 2 to find the difference between different parameters. -t here is 4 means user-defined kernel.

b. Result:

linear + RBF

	log2c=-10	log2c=-8	log2c=-6	log2c=-4	log2c=-2	log2c=0	log2c=2	log2c=4	log2c=6	log2c=8	log2c=10
log2g=-10	95.14	96.34	96.9	96.88	96.44	96.38	96.06	95.94	96.24	96.3	96.48
log2g=-8	95.2	96.36	97.0	96.72	96.3	96.22	96.38	96.32	96.18	96.22	96.42
log2g=-6	95.34	96.52	96.82	96.84	96.36	96.4	96.44	96.34	96.7	96.3	96.26
log2g=-4	95.32	96.46	96.92	97.06	96.38	96.4	96.32	96.2	96.42	96.06	96.4
log2g=-2	95.42	96.24	96.9	96.92	96.54	96.24	96.38	96.5	96.6	96.36	96.42
log2g=0	95.34	96.2	96.86	96.94	96.34	96.48	96.54	96.36	96.28	96.46	96.16
log2g=2	95.16	96.4	97.0	96.66	96.48	96.2	96.52	96.2	96.76	96.32	96.44
log2g=4	95.16	96.4	96.92	96.7	96.44	96.02	96.4	96.22	96.1	96.42	96.4
log2g=6	95.44	96.42	96.9	97.08	96.48	96.38	95.98	96.64	96.2	96.12	96.62
log2g=8	95.22	96.38	97.12	96.86	96.46	96.18	96.54	96.64	96.5	96.32	96.38
log2g=10	95.3	96.34	97.04	96.76	96.52	96.02	96.52	96.52	96.56	96.38	96.4

```
Accuracy = 95.96% (2399/2500) (classification)
best:       97.11999999999999
best_log2c: -6
best_log2g: 8
p_acc:      95.96000000000001%
acc:
[[95.14 96.34 96.9  96.88 96.44 96.38 96.06 95.94 96.24 96.3  96.48]
 [95.2  96.36 97.   96.72 96.3  96.22 96.38 96.32 96.18 96.22 96.42]
 [95.34 96.52 96.82 96.84 96.36 96.4  96.44 96.34 96.7  96.3  96.26]
 [95.32 96.46 96.92 97.06 96.38 96.4  96.32 96.2  96.42 96.06 96.4 ]
 [95.42 96.24 96.9  96.92 96.54 96.24 96.38 96.5  96.6  96.36 96.42]
 [95.34 96.2  96.86 96.94 96.34 96.48 96.54 96.36 96.28 96.46 96.16]
 [95.16 96.4  97.   96.66 96.48 96.2  96.52 96.2  96.76 96.32 96.44]
 [95.16 96.4  96.92 96.7  96.44 96.02 96.4  96.22 96.1  96.42 96.4 ]
 [95.44 96.42 96.9  97.08 96.48 96.38 95.98 96.64 96.2  96.12 96.62]
 [95.22 96.38 97.12 96.86 96.46 96.18 96.54 96.64 96.5  96.32 96.38]
 [95.3  96.34 97.04 96.76 96.52 96.02 96.52 96.52 96.56 96.38 96.4 ]]
```

c. <u>Observation:</u>

The linear + RBF kernel has high performance and stable no matter which c and g, just like linear kernel. According to the outcomes, compare with RBF kernel, user-defined kernel can combine advantage of several kernels, it seems better than single kernel. For example, it's stabler than RBF kernel.

Same as the observation in part 2, even user-defined kernel has high performance when you choose suitable parameters.