ML_HW5_report

309554027 鍾弈言

I. Gaussian Process

A. code with detailed explanations (20%)

1. Part1: Apply Gaussian Process Regression to predict the distribution of f and visualize the result.

a) Running command:
$ python3 main.py hw5-1

b) read input data(in main.py): I use np.genfromtxt to read the input data as np.ndarray and pass input data into my implement function gaussianProcessRegression(), which is defined in ml/Gaussian_process.py.

```
18      if (sys.argv[1] == 'hw5-1'):
19          if (len(sys.argv) != 2):
20              usage()
21          # read data
22          input_data = np.genfromtxt('./data/input.data', delimiter=' ')
23          ml.gaussianProcessRegression(input_data)
```

c) Rational Quadratic Kernel(Gaussian_process.py)
Rational Quadratic Kernel is parameterized by a length scale parameter $l > 0$ and a scale mixture parameter $a > 0$. The kernel is given by:

$$k(x_i, x_j) = \left(1 + \frac{d(x_i, x_j)^2}{2\alpha l^2}\right)^{-\alpha}$$

, where α is the scale mixture parameter, l is the length scale of the kernel and $d(\cdot, \cdot)$ is the Euclidean distance.
I implement it in the rationalQuadraticKernel() function and set default length = 1 and alpha = 1.

```
kernel_function = (1 + cdist(Xi, Xj, 'sqeuclidean') /
                   (2 * alpha * (length ** 2))) ** (-alpha)
return kernel_function
```

d) computeGaussianProcess(): compute Gaussian Process and do prediction, it return $\mu(x^*)$ and $\sigma^2(x^*)$.

First, we want to compute the marginal likelihood $P(y) = N(y|0, C)$, where the covariance matrix C has elements

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

Thus,we compute covariance matrix C as follows

```
# Gaussian process regression
# k(x, x)
k_x_x = rationalQuadraticKernel(X, X, length, alpha)
# covariance matrix
C = k_x_x + (1 / beta) * np.identity(X.shape[0])
```

Next, we want to predict the distribution of new f^* , we just need to cut the covariance matrix C on f* to see the conditional distribution thus achieve prediction.

conditional distribution $p(y^*|y)$ is a Gaussian distribution with:

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

following the above formula, the code is:

```
# Prediction
# k(x, x*)
k_x_xStar = rationalQuadraticKernel(X, X_star, length, alpha)
# k(x*, x*)
k_xStar_xStar = rationalQuadraticKernel(X_star, X_star, length, alpha)
# k*
k_star = k_xStar_xStar + 1 / beta
# μ(x*)
predict_mean_xStar = np.matmul(
    np.transpose(k_x_xStar), np.matmul(
        np.linalg.inv(C), Y))
# σ^2(x)
predict_variance_xStar = k_star - \
    np.matmul(np.transpose(k_x_xStar), np.matmul(np.linalg.inv(C), k_x_xStar))
```

e) Visualize: Finally, i use matplotlib.pyplot to visualize the result of Gaussian Process Regression. The 95% confidence interval indicates that the Z score is 1.96. And the Z score formula is: $z = \frac{x-\mu}{\sigma}$, thus we get the confidence interval is

```
# 95% confidence interval = 1.96x variance
confidence_interval_95 = 1.96 * var
```

Then, we can draw the figure by following code

```
    drawRegression(X_star[:, 0], predict_mean[:, 0], confidence_interval_95)
def drawRegression(x, y, boundary):
    plt.plot(x, y, linewidth=1, color='blue')
    plt.plot(x, y + boundary, linewidth=1, color='red')
    plt.plot(x, y - boundary, linewidth=1, color='red')
    plt.fill_between(x, y + boundary, y - boundary, color='coral', alpha=0.5)
    plt.xlim(-60.0, 60.0)
    plt.ylim(-3.0, 3.0)
```

2. Part2: Optimize the kernel parameters by minimizing negative marginal log-likelihood, and visualize the result again.
    a) Optimize the kernel parameters: According to our kernel function, we have two parameters need to be optimized(length and alpha).In the beginning, I guess theta is(length = 1, alpha = 1), and I use scipy.optimize.minimize to optimize the

parameters(implement at the function optimizeKernelParameter()). I pass my compute negative marginal log likelihood function and theta to minimize and get optimized parameters.

```
opt_result = minimize(
    computeNegativeMarginalLogLikelihood,
    theta,
    args=(
        X,
        Y,
        beta))
```

b) compute negative marginal log likelihood: First, we use rationalQuadraticKernel() to get covariance matrix C_θ. The marginal likelihood function is $p(y|\theta) = N(y|0, C_\theta)$. Thus, the negative marginal log likelihood is

$$-\ ln\, p(y|\theta) = \frac{1}{2} ln\, |C_\theta| + \frac{1}{2} y^T C_\theta^{-1} y + \frac{N}{2} ln\, (2\pi)$$

According to above formula, the code implement is

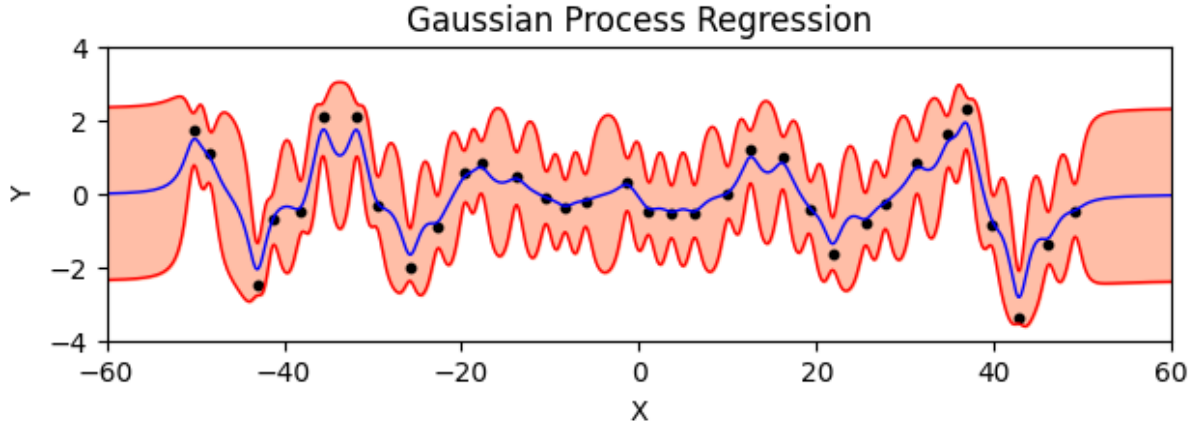
```
nega_log_likelihood = 0.5 * np.log(np.linalg.det(C_theta)) + 0.5 * np.matmul(
    np.transpose(Y), np.matmul(np.linalg.inv(C_theta), Y)) + 0.5 * N * np.log(2 * math.pi)
```

c) Visualize: same as part1.

B. experiments settings and results

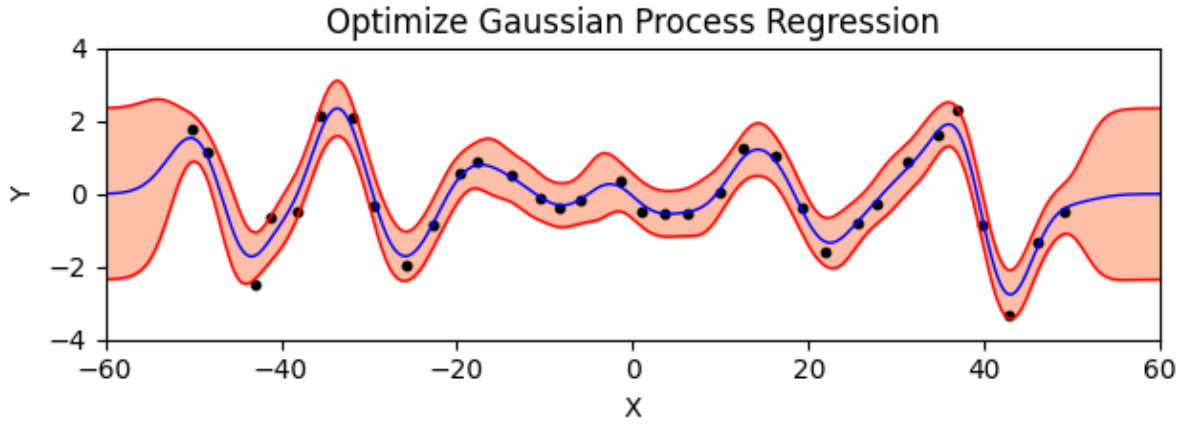
1. Part1:

$l = 1,\ \alpha = 1$



2. Part2:

$l = 2.9679647,\ \alpha = 473.616351$



C. observations and discussion

1. The optimized version is better than the original version.
2. The bigger length^2 is, the less wiggly your random functions are. This is because larger length will effectively be blurring together points in a larger window. Thus, you can see the wiggly in the optimized version($l = 2.9679647$) is less than the original version ($l = 1$) .
3. The 95% confidence interval of the optimized version is smaller than the original version.
4. The interval with data points has a smaller confidence interval than the interval without data points. This means the interval with data points is easier to predict than the interval without data points.
5. Using default parameters($l = 1, \alpha = 1$) is about 2.3 times faster than using optimized parameters ($l = 2.9679647,\ \alpha = 473.616351$).

II. SVM

A. code with detailed explanations (20%)

1. Part1: Use different kernel functions (linear, polynomial, and RBF kernels) and compare their performance.
   a) Running command:
      $ python3 main.py hw5-2
   b) Read data in main.py: I directly read data in the ./data/*.csv and pass data into my implement

function ml.SVM(), which is defined in ml/SVM.py.

```
25      elif (sys.argv[1] == 'hw5-2'):
26          if (len(sys.argv) != 2):
27              usage()
28          # read data
29          X_train = np.genfromtxt('./data/X_train.csv', delimiter=',')
30          Y_train = np.genfromtxt('./data/Y_train.csv', delimiter=',')
31          X_test = np.genfromtxt('./data/X_test.csv', delimiter=',')
32          Y_test = np.genfromtxt('./data/Y_test.csv', delimiter=',')
33          ml.SVM(X_train, Y_train, X_test, Y_test)
```

c) At SVM.py: Since I need to compare the performance of three kernel functions, I call my own function computeSVMAndTime() to compute SVM and measure the execution time.

```
282     print('===== Compare the performance with linear, polynomial, RBF kernel function =====')
283     # linear = 0
284     computeSVMAndTime(0, X_train, Y_train, X_test, Y_test)
285     # polynomial = 1
286     computeSVMAndTime(1, X_train, Y_train, X_test, Y_test)
287     # radial basis function(RBF) = 2
288     computeSVMAndTime(2, X_train, Y_train, X_test, Y_test)
```

d) computeSVMAndTime(): I use the option -t provided in the libsvm package to indicate the kernel function I want. The number 0 represents linear, 1 represents polynomial and 2 represents RBF. I pass the kernel_type by the first parameter and call the svm_train for training the model. Finally, I call svm_predict to use my model to predict the test data. I compute the training and prediction time and print the total execution time in the end. (The code is pasted on the next page.)

```python
def computeSVMAndTime(
        kernel_type: int,
        X_train: np.ndarray,
        Y_train: np.ndarray,
        X_test: np.ndarray,
        Y_test: np.ndarray):
    """
    compute SVM use input kernel_type(kernel function) and compute total cost of time.

    parameters
    kernel_type: type of kernel function, 0 = linear, 1 = polynomial, 2 = radial basis function(RBF)

    <source> https://www.itread01.com/content/1549816773.html
    """
    print('kernel_type: ', end='')
    if kernel_type == 0:
        print('linear')
    elif kernel_type == 1:
        print('polynomial')
    elif kernel_type == 2:
        print('RBF')
    else:
        print('Error kernel type!')
        exit(-1)

    start_time = time.time()
    # -t: choose kernel function, -q: quite mode(no outputs)
    parameters = svm_parameter('-t ' + str(kernel_type) + ' -q')
    problem = svm_problem(Y_train, X_train)
    model = svm_train(problem, parameters)
    p_label, p_acc, p_val = svm_predict(Y_test, X_test, model)
    end_time = time.time()
    print(f'total cost {end_time - start_time:.2f} sec.\n')
```

2. Part2: Please do the grid search for finding parameters of the best performing model.
    a) I tried to find the best parameters for all kernel functions. Thus, I call the function gridSearch() three times by passing different kernel function types.

```python
print('\n===== Use grid search method to find the best performing parameters in each kernel function =====')
# linear = 0
gridSearch(0, X_train, Y_train)
# polynomial = 1
gridSearch(1, X_train, Y_train)
# radial basis function(RBF) = 2
gridSearch(2, X_train, Y_train)
```

    b) The formula of the three kernel functions: Since we want to find out the best parameters of each kernel function, we need to figure out the formula of each kernel function first.

$$linear\ kernel\colon\ k(u,\ v) =< u,\ v >$$

$$polynomial\ kernel\colon\ k(u,\ v) = (\gamma\ *< u,\ v >+\ coef0)^{d}$$

$$RBF\ kernel\colon\ k(u,\ v) = e^{-\gamma||u-v||^{2}}$$

Because in C-SCV, all kernel functions have a parameter C. Therefore, the parameters that need to be trained for each kernel function are summarized in the below table.

kernel function	linear	polynomial	RBF
need parameters	c	$c,\ \gamma,\ coef0,\ d$	$c,\ \gamma$

The below table is the parameters I am trying to tune.

parameters	c	γ	$coef0$	d
tuning value	$0.1,\ 1,\ 10$	$\frac{1}{784},\ 0.1,\ 0$	$0.1,\ 1,\ 10$	$0,\ 1,\ 2,\ 3$

```
cost = [np.power(10.0, i) for i in range(-1, 2)]
gamma = [1.0 / 784] + [np.power(10.0, i) for i in range(-1, 1)]
coef0 = [np.power(10.0, i) for i in range(-1, 2)]
degree = [i for i in range(0, 4)]
```

Since we need to do cross-validation, we should decide the number of n fold we use. There, I set n=3 in my implementation.

```
# set n fold
num_n = 3
```

Then I tried all possible combinations of parameters and found out the best accuracy and best parameters in each kernel function. Finally, I use the best parameters to train the model and make predictions.
Paste the complete code as follows:

gridSearch(): which implements the grid search.

```
def gridSearch(
        kernel_type: int,
        X_train: np.ndarray,
        Y_train: np.ndarray,
        X_test: np.ndarray,
        Y_test: np.ndarray):
    """
    Use grid search method to find the best performing parameters in each kernel function.

    If '-v' is specified in 'options' (i.e., cross validation)either accuracy (ACC) or mean-squared error (MSE) is returned.
        -v n: n-fold cross validation mode

    -t kernel_type : set type of kernel function (default 2)
    0 -- linear: u'*v, the dot product of u and v
        -c cost : set the parameter C of C-SVC, epsilon-SVR, and nu-SVR (default 1)
    1 -- polynomial: (gamma*u'*v + coef0)^degree
        -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)
        -r coef0 : set coef0 in kernel function (default 0), coef0 is the intercept of polynomial.
        -d degree : set degree in kernel function (default 3)
    2 -- radial basis function: exp(-gamma*|u-v|^2)
        -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)
    p.s. the means of cost and gamma
        The larger the gamma value, the smaller the range of influence of data point.
        The larger the cost(C) value, the smaller the tolerance of outlier.
    """
    # set n fold
    num_n = 3

    best_accuracy = 0
    best_parameter = []
```

Linear:

```
    if kernel_type == 0:
        print('kernel function: linear')
        # set parameters
        cost = [np.power(10.0, i) for i in range(-1, 2)]

        # set stdout to devnull
        save_stdout = sys.stdout
        sys.stdout = open(os.devnull, 'w')

        for c in cost:
            parameters = svm_parameter(
                '-t ' +
                str(kernel_type) +
                ' -c ' +
                str(c) +
                ' -v ' +
                str(num_n) +
                ' -q')
            problem = svm_problem(Y_train, X_train)
            accuracy = svm_train(problem, parameters)
            if accuracy > best_accuracy:
                best_accuracy = accuracy
                best_parameter = {'cost': c}

        # reset stdout
        sys.stdout = save_stdout
        # Prediction
        opt_parameters = svm_parameter(
            '-t ' + str(kernel_type) + ' -c ' + str(best_parameter['cost']) + ' -q')
        problem = svm_problem(Y_train, X_train)
        model = svm_train(problem, opt_parameters)
        p_label, p_acc, p_val = svm_predict(Y_test, X_test, model)
```

plynomial:

```
    elif kernel_type == 1:
        print('kernel function: polynomial')
        # set parameters
        cost = [np.power(10.0, i) for i in range(-1, 2)]
        gamma = [1.0 / 784] + [np.power(10.0, i) for i in range(-1, 1)]
        coef0 = [np.power(10.0, i) for i in range(-1, 2)]
        degree = [i for i in range(0, 4)]

        # set stdout to devnull
        save_stdout = sys.stdout
        sys.stdout = open(os.devnull, 'w')

        for c in cost:
            for g in gamma:
                for r in coef0:
                    for d in degree:
                        parameters = svm_parameter(
                            '-t ' +
                            str(kernel_type) +
                            ' -c ' +
                            str(c) +
                            ' -g ' +
                            str(g) +
                            ' -r ' +
                            str(r) +
                            ' -d ' +
                            str(d) +
                            ' -v ' +
                            str(num_n) +
                            ' -q')
                        problem = svm_problem(Y_train, X_train)
                        accuracy = svm_train(problem, parameters)
                        if accuracy > best_accuracy:
                            best_accuracy = accuracy
                            best_parameter = {
                                'cost': c, 'gamma': g, 'coef0': r, 'degree': d}
```

```
        # reset stdout
        sys.stdout = save_stdout
        # Prediction
        opt_parameters = svm_parameter('-t ' +
                                       str(kernel_type) +
                                       ' -c ' +
                                       str(best_parameter['cost']) +
                                       ' -g ' +
                                       str(best_parameter['gamma']) +
                                       ' -r ' +
                                       str(best_parameter['coef0']) +
                                       ' -d ' +
                                       str(best_parameter['degree']) +
                                       ' -q')
        problem = svm_problem(Y_train, X_train)
        model = svm_train(problem, opt_parameters)
        p_label, p_acc, p_val = svm_predict(Y_test, X_test, model)
```

RBF:

```
    elif kernel_type == 2:
        print('kernel function: RBF')
        # set parameters
        cost = [np.power(10.0, i) for i in range(-1, 2)]
        gamma = [1.0 / 784] + [np.power(10.0, i) for i in range(-1, 1)]

        # set stdout to devnull
        save_stdout = sys.stdout
        sys.stdout = open(os.devnull, 'w')

        for c in cost:
            for g in gamma:
                parameters = svm_parameter(
                    '-t ' +
                    str(kernel_type) +
                    ' -c ' +
                    str(c) +
                    ' -g ' +
                    str(g) +
                    ' -v ' +
                    str(num_n) +
                    ' -q')
                problem = svm_problem(Y_train, X_train)
                accuracy = svm_train(problem, parameters)
                if accuracy > best_accuracy:
                    best_accuracy = accuracy
                    best_parameter = {'cost': c, 'gamma': g}
        # reset stdout
        sys.stdout = save_stdout
        # Prediction
        opt_parameters = svm_parameter('-t ' +
                                       str(kernel_type) +
                                       ' -c ' +
                                       str(best_parameter['cost']) +
                                       ' -g ' +
                                       str(best_parameter['gamma']) +
                                       ' -q')
        problem = svm_problem(Y_train, X_train)
        model = svm_train(problem, opt_parameters)
        p_label, p_acc, p_val = svm_predict(Y_test, X_test, model)

    else:
        print('Invalid kernel_type[0 ~ 2], exit')
        exit(-1)

    # print best parameter and accuracy
    printGridSearchResult(best_accuracy, best_parameter)
```

Finally, I call the function printGridSearchResult() to show the result.

```
def printGridSearchResult(best_accuracy: float, best_parameter: dict):
    print(f'Best performing accuracy: {best_accuracy:.2f}%')
    print(f'Best parameters: {best_parameter}\n')
```

3. Part3: Use linear kernel + RBF kernel together (therefore a new kernel function) and compare its performance with respect to others.
    a) User defined kernel in libsvm: First of all, we need to figure out how to use user-defined kernel function in libsvm. The option -t can set kernel function type and libsvm has provided some regular kernel functions(which used at previous implementation). Setting the option -t to value 4 means we use a precomputed kernel. If we set the option -t to value 4, then the training data of X(X_train) must rewrite to the following format:

index of X	return value of kernel function
1	k(1)
2	k(2)
...	...
n	k(n)

When passing our new data to svm_problem, we need to set the parameter ‘isKernel=True’ means that we use the kernel function that is defined by myself. The code show as below:

```
combination = np.hstack(
    (np.arange(1, train_rows + 1).reshape(-1, 1), linear + rbf))
parameters = svm_parameter(f'-t 4 -c {c} -v {num_n} -q')
problem = svm_problem(Y_train, combination, isKernel=True)
accuracy = svm_train(problem, parameters)
```

b) Code implement:
First, I defined my own function to compute the linear and RBF kernel function.

```python
def computeLinearKernel(u: np.ndarray, v: np.ndarray):
    """
    Linear kernel function, compute dot product <u, v> and return the linear distance between them.

    parameters
    u: the point u
    v: the point v
    """
    return np.dot(u, np.transpose(v))


def computeRBF(u: np.ndarray, v: np.ndarray, gamma=1 / 784):
    """
    radial basis function: exp(-gamma*|u-v|^2)
        -g gamma : set gamma in kernel function (default 1/num_features)
    """
    return np.exp(-gamma * cdist(u, v, 'sqeuclidean'))
```

Second, in the main implement function combineLinearAndRBF(), I tried to find out the best parameters($c\ and\ \gamma$). Therefore, I use a grid search method with n=3 fold to find out the best accuracy model.

```
199 def combineLinearAndRBF(
200         X_train: np.ndarray,
201         Y_train: np.ndarray,
202         X_test: np.ndarray,
203         Y_test: np.ndarray):
204     # compute linear kernel
205     linear = computeLinearKernel(X_train, X_train)
206 
207     # set n fold
208     num_n = 3
209 
210     # find best parameters for RBF kernel
211     # set parameters
212     cost = [np.power(10.0, i) for i in range(-1, 2)]
213     gamma = [1.0 / 784] + [np.power(10.0, i) for i in range(-1, 1)]
214     best_accuracy = 0
215     best_c = 0
216     best_gamma = 0
217     train_rows = X_train.shape[0]
218 
219     # set stdout to devnull
220     save_stdout = sys.stdout
221     sys.stdout = open(os.devnull, 'w')
222 
223     for c in cost:
224         for g in gamma:
225             rbf = computeRBF(X_train, X_train, g)
226             combination = np.hstack(
227                 (np.arange(1, train_rows + 1).reshape(-1, 1), linear + rbf))
228             parameters = svm_parameter(f'-t 4 -c {c} -v {num_n} -q')
229             problem = svm_problem(Y_train, combination, isKernel=True)
230             accuracy = svm_train(problem, parameters)
231             if accuracy > best_accuracy:
232                 best_accuracy = accuracy
233                 best_c = c
234                 best_gamma = g
```

Finally, I put the best parameters I searched before in the parameters to train the model. Then I transform the X_test data form and predict the X_test data. Measuring the execution time starts from the training model and ends after finishing prediction.

```
                best_gamma = g

    start_time = time.time()

    # use the best parameter to training model
    rbf = computeRBF(X_train, X_train, best_gamma)
    combination = np.hstack(
        (np.arange(1, train_rows + 1).reshape(-1, 1), linear + rbf))
    parameters = svm_parameter(f'-t 4 -c {best_c} -q')
    problem = svm_problem(Y_train, combination, isKernel=True)
    model = svm_train(problem, parameters)

    sys.stdout = save_stdout

    # print grid search result
    print(f'Combine linear and RBF kernel function')
    print(f'Best accuracy: {best_accuracy}%')
    print(f'Best parameters: cost = {best_c}, gamma = {best_gamma}')

    # transform X_test
    test_rows = X_test.shape[0]
    linear = computeLinearKernel(X_test, X_test)
    rbf = computeRBF(X_test, X_test, best_gamma)
    new_X_test = np.hstack(
        (np.arange(1, test_rows + 1).reshape(-1, 1), linear + rbf))

    # predict
    p_label, p_acc, p_val = svm_predict(Y_test, new_X_test, model)
    print(f'Predict accuracy: {p_acc}%')

    end_time = time.time()
    print(f'total cost {end_time - start_time:.2f} sec.\n')
```

B. experiments settings and results (20%)

1. Part1:

```
===== Compare the performance with linear, polynomial, RBF kernel function =====
kernel_type: linear
Accuracy = 95.08% (2377/2500) (classification)
train cost 1.52660 sec.
predict cost 1.04186 sec.
total cost 2.57 sec.

kernel_type: polynomial
Accuracy = 34.68% (867/2500) (classification)
train cost 18.87900 sec.
predict cost 6.21107 sec.
total cost 25.09 sec.

kernel_type: RBF
Accuracy = 95.32% (2383/2500) (classification)
train cost 3.42083 sec.
predict cost 2.21455 sec.
total cost 5.64 sec.
```

2. Part2:

```
===== Use grid search method to find the best performing parameters in each kernel function
kernel function: linear
Accuracy = 95.8% (2395/2500) (classification)
Best performing accuracy: 96.66%
Best parameters: {'cost': 0.1}

kernel function: polynomial
Accuracy = 97.84% (2446/2500) (classification)
Best performing accuracy: 98.16%
Best parameters: {'cost': 0.1, 'gamma': 1.0, 'coef0': 10.0, 'degree': 2}

kernel function: RBF
Accuracy = 96.28% (2407/2500) (classification)
Best performing accuracy: 97.24%
Best parameters: {'cost': 10.0, 'gamma': 0.0012755102040816326}
```

3. Part3: (compare by part1.)

```
===== Combine Linear and RBF kernel function and compare the performance with others
Combine linear and RBF kernel function
Best accuracy: 96.82%
Best parameters: cost = 0.1, gamma = 0.1
Accuracy = 24.56% (614/2500) (classification)
train cost 19.77645 sec.
transform X_test cost 2.59439 sec.
predict cost 2.18856 sec.
total cost 24.56 sec.
```

C. observations and discussion (10%)

1. We can summarize the performance of each kernel function in the following table.

kernel function	linear	polynomial	RBF	linear+RBF
train time	1.52660	18.87900	3.42083	19.77645
transform time	X	X	X	2.59439
predict time	1.04186	6.21107	2.21455	2.18856
performance(sec)	2.57	25.09	5.64	24.56
predict accuracy (part1)	95.08%	34.68%	95.32%	X
predict accuracy (part2)	95.8%	97.84%	96.28%	X
best parameter (part2)	cost=0.1	cost=0.1 gamma=1 coef0=10 degree=2	cost=10 gamma $=\frac{1}{784}$	cost=0.1 gamma=0.1 (part3)
predict accuracy (part3)	X	X	X	24.56%

2. We can observe that the training took more time than predicted.
3. The observation is that the more parameters the kernel function uses, the more time cost in training the model and prediction.
   (poly (4 param)>RBF (2 param)>linear(1 param))
4. The observation is that user defined kernel functions cost more times than original kernel functions in training stage.(19.16916 > 1.53887 + 3.41844)
5. Since the polynomial kernel function has four parameters, when executing the grid methods it costs more time than other kernel functions.
6. According to the result of part1 and part2, we can find that the RBF kernel function has higher predicted accuracy than others. This is because RBF can project

data into an infinite dimension feature space, and it can find a hyperplane to better separate the data.

7. After using the grid search method, the accuracy of the polynomial kernel function increases the most (34.68% -> 97.84%).
8. The parameter cost C in C-SVM is a regularization parameter. This means how well the model tolerates outliers. The larger C we choose, the less outliers we can tolerate. Relatively, the larger C means the correctness of classify is higher.

Running environment:

```
python = "^3.8"
numpy = "^1.21"
scipy = "^1.6.1"
matplotlib = "^3.4"
pyqt5 = "^5.15"
numba = "^0.50"
libsvm-official = "^3.25"
```

Use poetry to build the environment and run the code:

$ poetry install --no-root
$ poetry shell
$ python3 main.py hw5-1 // hw5 part1
$ python3 main.py hw5-2 // hw5 part2