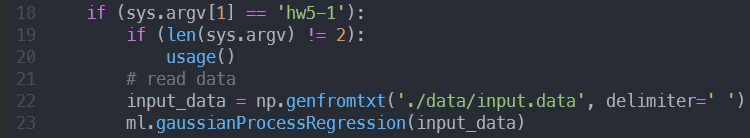
ML\_HW5\_report

309554027 鍾弈言

1. Gaussian Process
   1. code with detailed explanations (20%)
      1. Part1: Apply Gaussian Process Regression to predict the distribution of and visualize the result.
         1. Running command:

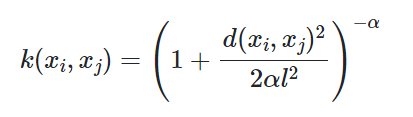
$ python3 main.py hw5-1

* + - 1. 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.

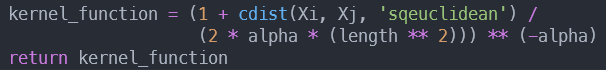


* + - 1. Rational Quadratic Kernel(Gaussian\_process.py)

Rational Quadratic Kernel is parameterized by a length scale parameter and a scale mixture parameter . The kernel is given by:



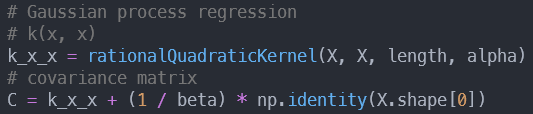
, where is the scale mixture parameter, is the length scale of the kernel and is the Euclidean distance.

I implement it in the rationalQuadraticKernel() function and set default length = 1 and alpha = 1.

* + - 1. computeGaussianProcess(): compute Gaussian Process and do prediction, it return and .

First, we want to compute the marginal likelihood , where the covariance matrix C has elements

Thus,we compute covariance matrix C as follows



Next, we want to predict the distribution of new

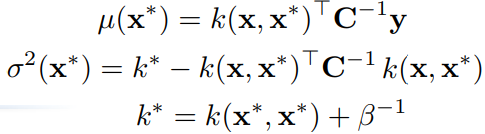
, we just need to cut the covariance matrix C on

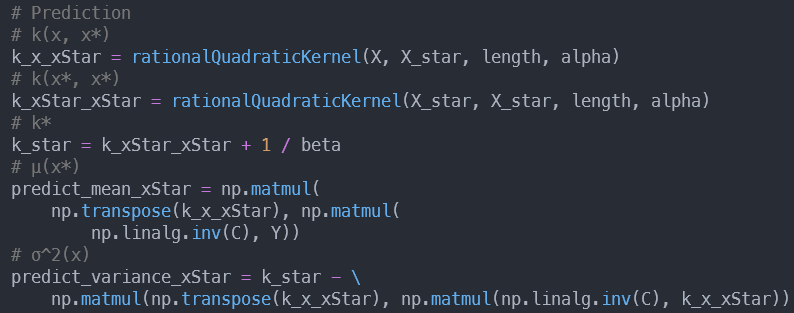
f\* to see the conditional distribution thus achieve

prediction.

conditional distribution is a Gaussian

distribution with:

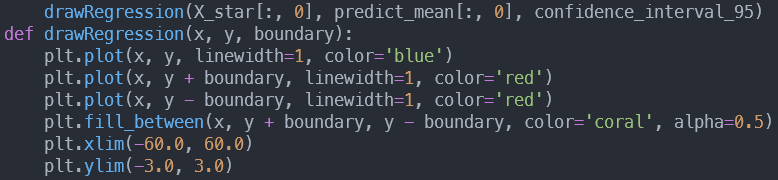
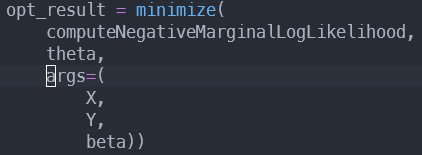


following the above formula, the code is:

* + - 1. 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: , thus we get the confidence interval is

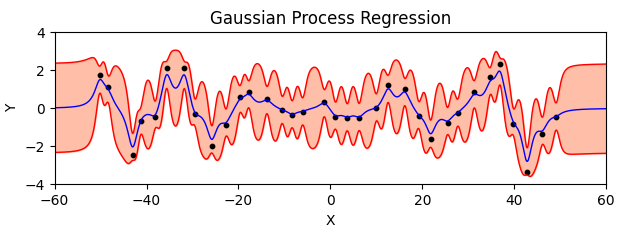


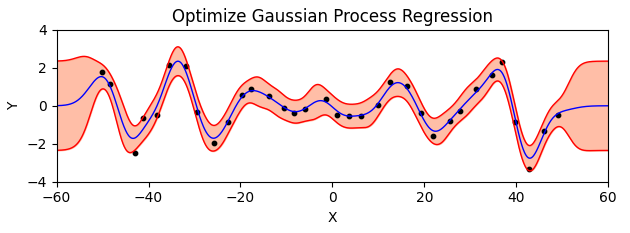
Then, we can draw the figure by following code

* + 1. Part2: Optimize the kernel parameters by minimizing negative marginal log-likelihood, and visualize the result again.
       1. 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.
       2. compute negative marginal log likelihood: First, we use rationalQuadraticKernel() to get covariance matrix . The marginal likelihood function is . Thus, the negative marginal log likelihood is 

According to above formula, the code implement is

* + - 1. Visualize: same as part1.
  1. experiments settings and results
     1. Part1:



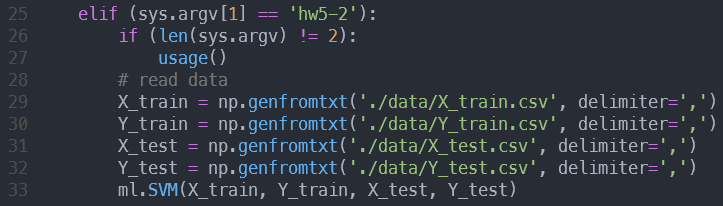
* + 1. Part2:
  1. observations and discussion
     1. The optimized version is better than the original version.
     2. The bigger length^2 is, the less wiggly your random functions are. This is because larger length will effectively be blurring together points in a larger window. Thus, you can see the wiggly in the optimized version() is less than the original version () .
     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() is about 2.3 times faster than using optimized parameters

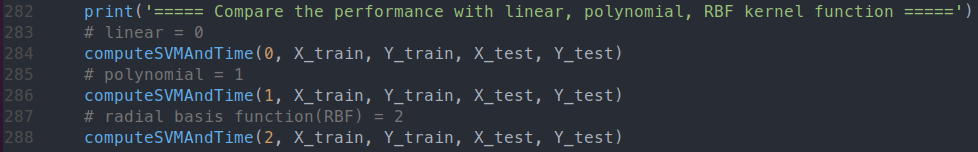
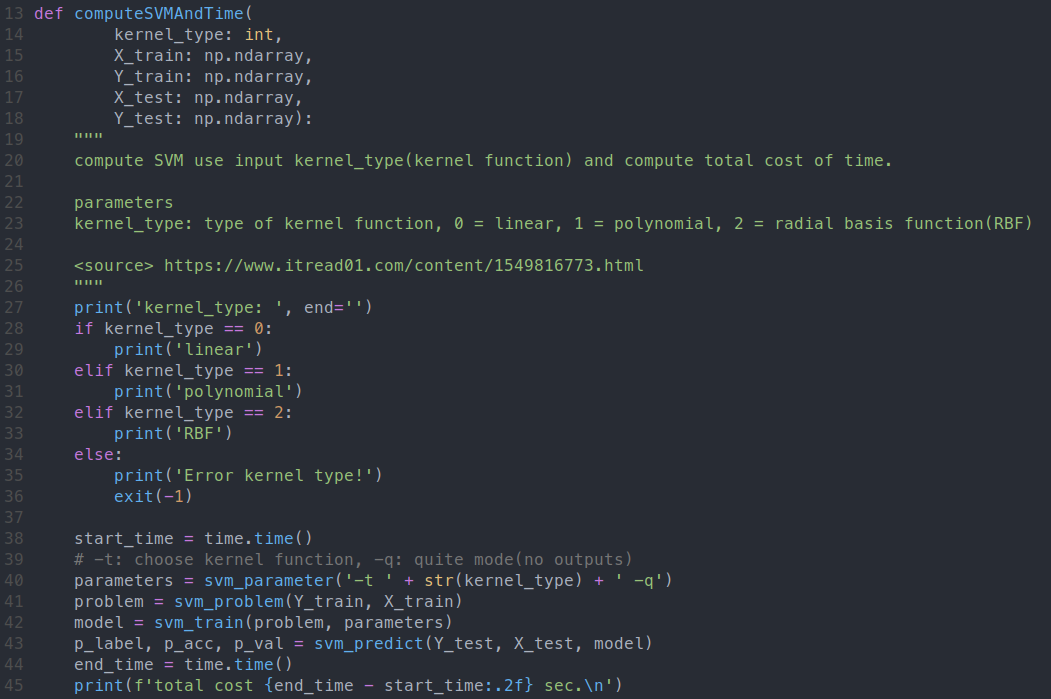
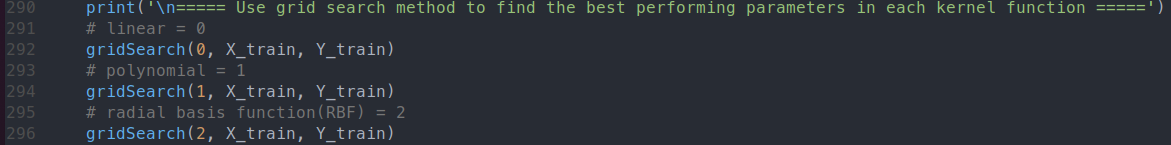
().

1. SVM
   1. code with detailed explanations (20%)
      1. Part1: Use different kernel functions (linear, polynomial, and RBF kernels) and compare their performance.
         1. Running command:

$ python3 main.py hw5-2

* + - 1. 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.



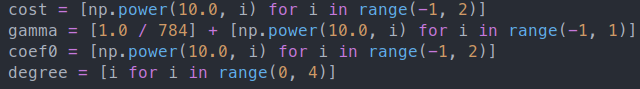
* + - 1. At SVM.py: Since I need to compare the performance of three kernel functions, I call my own function computeSVMAndTime() to compute SVM and measure the execution time.
      2. computeSVMAndTime(): I use the option -t provided in the libsvm package to indicate the kernel function I want. The 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.)
    1. Part2: Please do the grid search for finding parameters of the best performing model.
       1. 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.
       2. 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.

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** |  |  |  |

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

| **parameters** |  |  |  |  |
| --- | --- | --- | --- | --- |
| **tuning value** |  |  |  |  |



Since we need to do cross-validation, we should

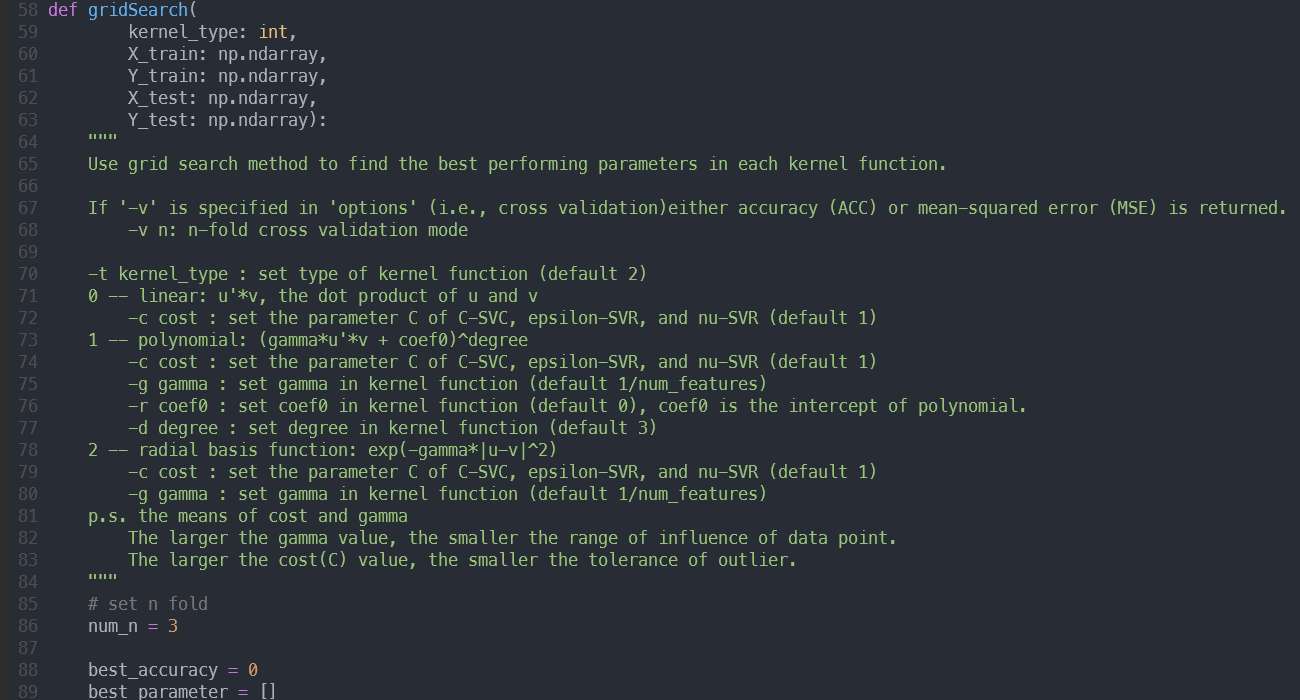
decide the number of n fold we use. There, I set

n=3 in my implementation.

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.

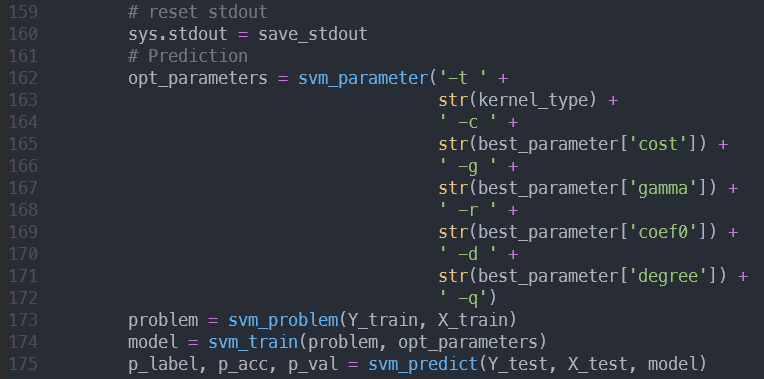


Linear:

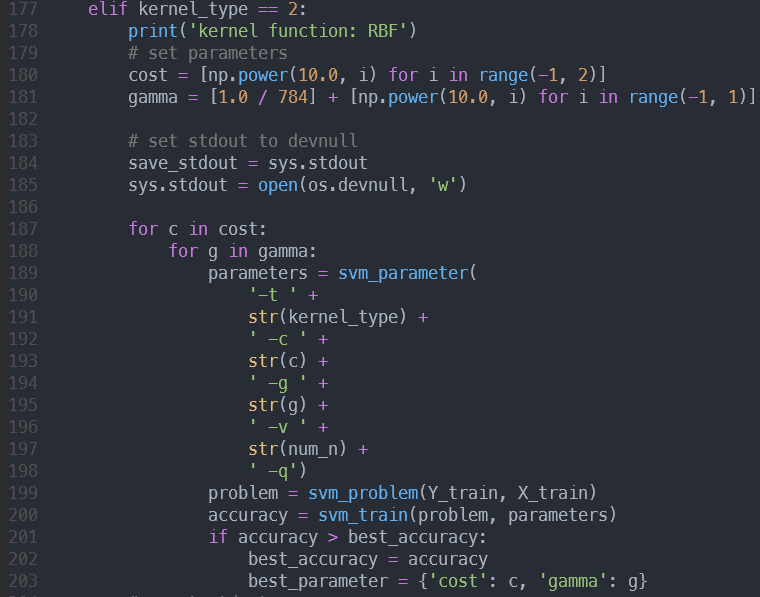


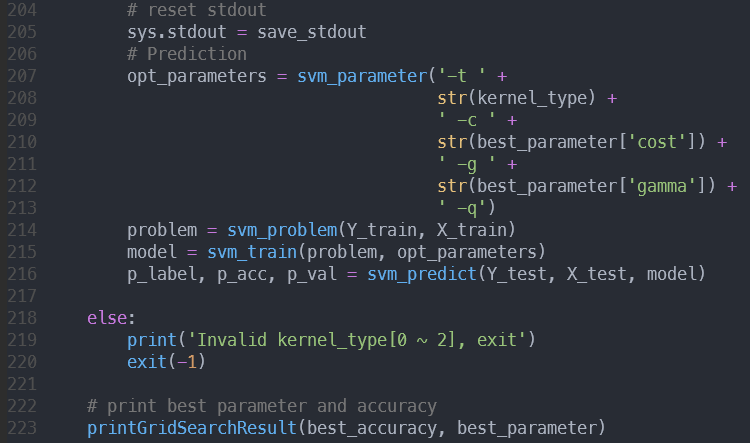
plynomial:



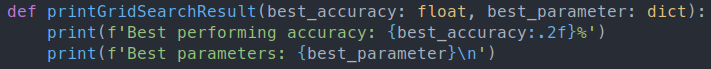


RBF:

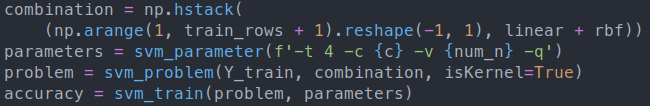




Finally, I call the function printGridSearchResult() to show the result.

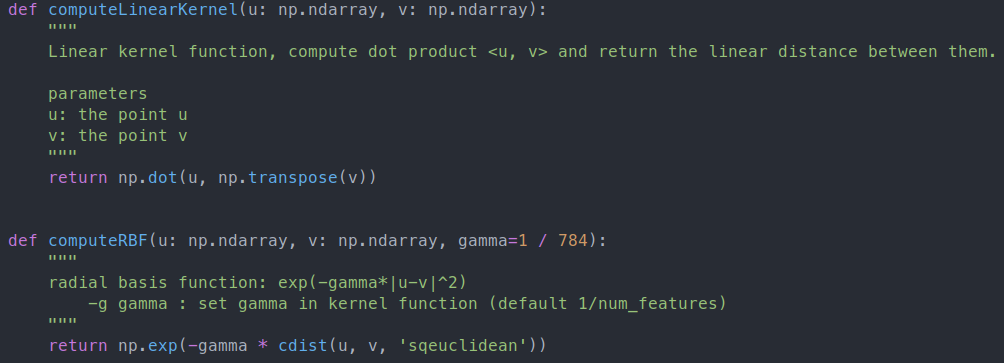
* + 1. Part3: Use linear kernel + RBF kernel together (therefore a new kernel function) and compare its performance with respect to others.
       1. User defined kernel in libsvm: First of all, we need to figure out how to use user-defined kernel function in libsvm. The option -t can set kernel function 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) |

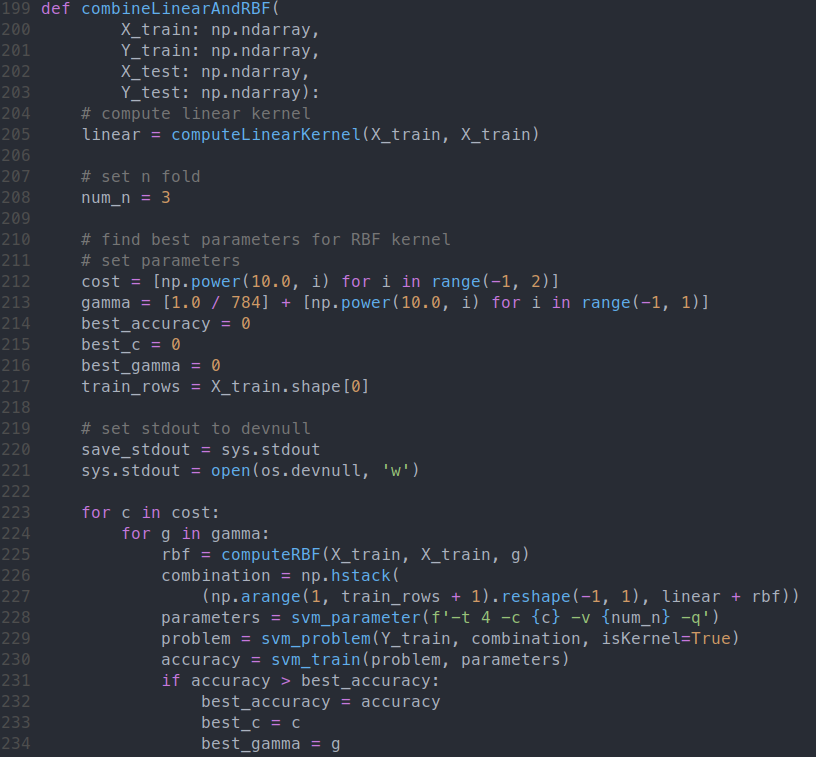
When passing our new data to svm\_problem, we need to set the parameter ‘isKernel=True’ means that we use the kernel function that is defined by myself. The code show as below:

* + - 1. Code implement:

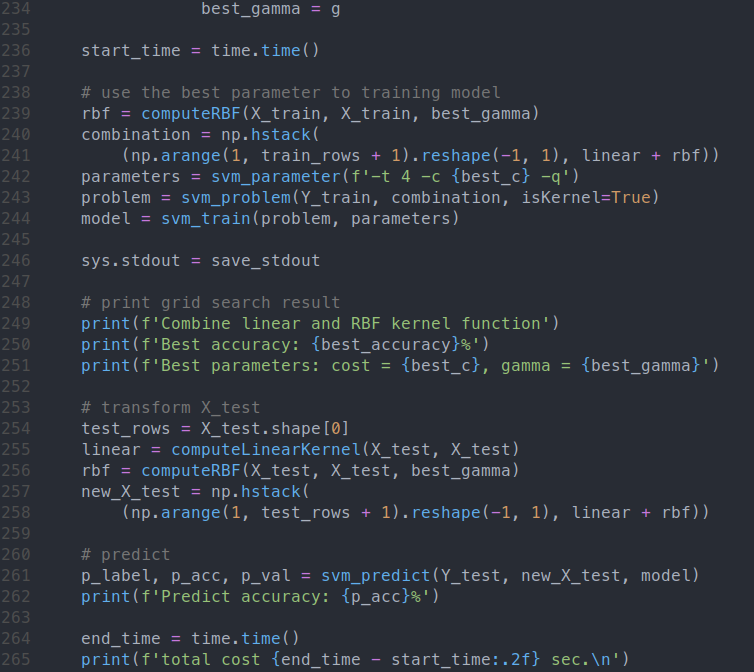
First, I defined my own function to compute the linear and RBF kernel function.

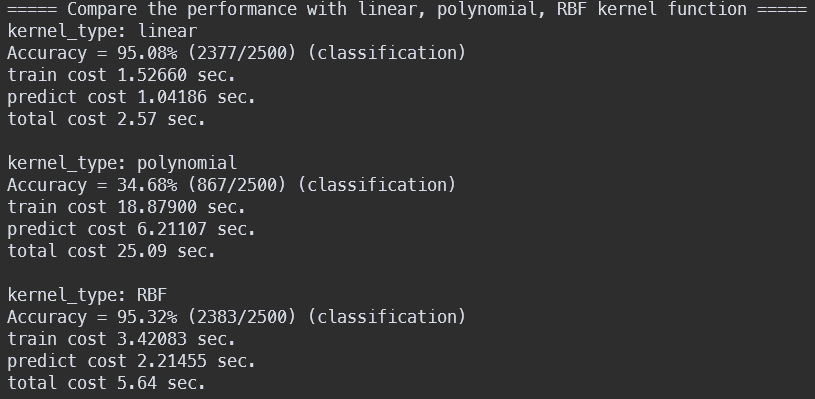
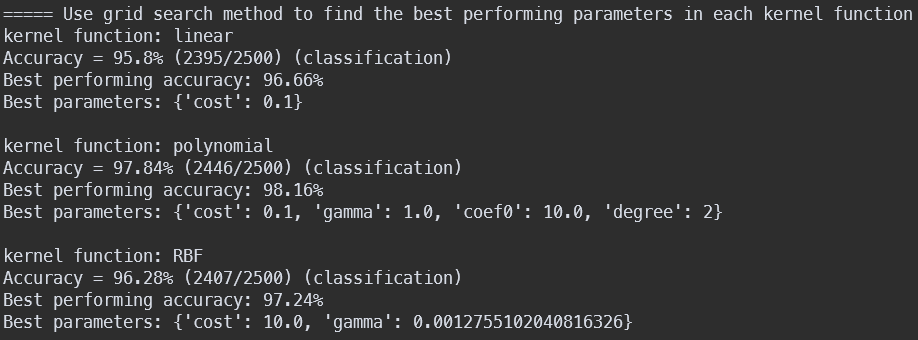
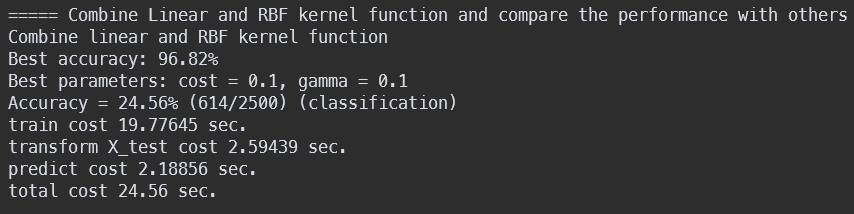


Second, in the main implement function combineLinearAndRBF(), I tried to find out the best parameters(). Therefore, I use a grid search method with n=3 fold to find out the best accuracy model.



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.



* 1. experiments settings and results (20%)
     1. Part1:
     2. Part2:
     3. Part3: (compare by part1.)
  2. 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  = | cost=0.1  gamma=0.1  (part3) |
| **predict accuracy**  **(part3)** | X | X | X | 24.56% |

* + 1. We can observe that the training took more time than predicted.
    2. The observation is that the more parameters the kernel function uses, the more time cost in training the model and prediction.

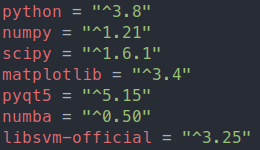
(poly (4 param)>RBF (2 param)>linear(1 param))

* + 1. The observation is that user defined kernel functions cost more times than original kernel functions in training stage.(19.16916 > 1.53887 + 3.41844)
    2. Since the polynomial kernel function has four parameters, when executing the grid methods it costs more time than other kernel functions.
    3. 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.
    4. After using the grid search method, the accuracy of the polynomial kernel function increases the most

(34.68% -> 97.84%).

* + 1. 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:



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