a. code with detailed explanations

- Part1

Import package and load data:

```
from libsvm.svmutil import *
import numpy as np
from numpy import genfromtxt
import matplotlib.pyplot as plt
import seaborn as sns
```

```
X_test = genfromtxt('ML_HW05-2/X_test.csv', delimiter=',')
X_train = genfromtxt('ML_HW05-2/X_train.csv', delimiter=',')
Y_test = genfromtxt('ML_HW05-2/Y_test.csv', delimiter=',')
Y_train = genfromtxt('ML_HW05-2/Y_train.csv', delimiter=',')
```

Import package used in HW5-2, including libsvm, numpy matplotlib…. Note that seaborn is used to create heatmap, we will used it to visualize our result.

SVM implementation:

```
def different_kernel_svm(kernel_type):
    prob  = svm_problem(Y_train, X_train)
    if kernel_type=="linear":
        param = svm_parameter('-t 0')
    elif kernel_type=="polynomial":
        param = svm_parameter('-t 1')
    else:
        param = svm_parameter('-t 2')

    model = svm_train(prob, param)
    p_label, p_acc, p_val = svm_predict(Y_test, X_test, model)

    confusion_matrix = show_confusion_matrix(Y_test, p_label)
```

```
different_kernel_svm("linear")
```

```
different_kernel_svm("polynomial")
```

```
different_kernel_svm("RBF")
```

According to libsvm, we can use different kernel function to run our model by set the parameters ‘-t’ (when t==0, kernel type is linear; when t==1, kernel type is polynomial; when t==2, kernel type is RBF). We run the function ‘different_kernel_svm()’ with different kernel type three times, and visualize their result through confusion matrix.

Visualize function

```
def show_confusion_matrix(real, pre):

    confusion_matrix = np.zeros((5, 5), dtype=int)

    for i in range(len(real)):

        confusion_matrix[int(real[i])-1][int(pre[i])-1] += 1

    sns.heatmap(confusion_matrix, cmap='Greens', annot=True, fmt='d')
    plt.xlabel("Predict")
    plt.ylabel("Real")
    x = [1,2,3,4,5]
    xi = [0.5,1.5,2.5,3.5,4.5]
    plt.xticks(xi, x)

    y = [1,2,3,4,5]
    yi = [0.5,1.5,2.5,3.5,4.5]
    plt.yticks(yi, y)
```

We use function above to create a heatmap in order to show our confusion matrix of predict label and true label. The heatmap will be shown later.

- ## Part2

Grid search

In part2, we use grid search to optimize two parameters: cost of C-SVC and gamma use in our RBF kernel.

```
def grid_search(all_cost, all_gamma):

    result_matrix = np.zeros((len(all_cost), len(all_gamma)))
    for i in range(len(all_cost)):
        for j in range(len(all_gamma)):
            result_matrix[i][j] = C_SVC(all_cost[i], all_gamma[j])

    return result_matrix
```

To implement grid search, we will input each cost and gamma that we want to estimate. For example:

```
all_cost = [1, 5, 10, 25, 50]
all_gamma = [0.0001, 0.001,0.01,0.1,1]
grid_search_result1 = grid_search(all_cost, all_gamma)
```

The ‘C_SVC’ function is implemented below, it will return the accuracy of specific cost and gamma.

```
def C_SVC(cost, gamma):
    prob  = svm_problem(Y_train, X_train)
    param = svm_parameter('-t 2 -c ' + str(cost) + ' -g ' + str(gamma))
    model = svm_train(prob, param)
    p_label, p_acc, p_val = svm_predict(Y_test, X_test, model)
    return p_acc[0]
```

Visualize function

```
def draw_result_matrix(all_cost, all_gamma, grid_search_result):

    sns.heatmap(grid_search_result, cmap='Blues', annot=True, fmt='.2f')

    plt.xlabel("Gamma")
    plt.ylabel("Cost")

    xi = []
    yi = []
    for i in range(len(all_gamma)):
        xi.append(i+0.5)
    plt.xticks(xi, all_gamma)

    for i in range(len(all_cost)):
        yi.append(i+0.5)
    plt.yticks(yi, all_cost)

```

Just like part1, we use above function to draw a heatmap and show our grid search result. Each grid in the heatmap represents the accuracy of specific gamma and cost.

- Part3

```
def linear_kernel(a, b):
    a = np.array(a)
    b = np.array(b)
    return a.T @ b

def RBF_kernel(a, b):
    gamma = 1/784
    a_b = np.array([a[i] - b[i] for i in range(len(a))])
    return np.exp(-1*gamma*(a_b.T @ a_b))
```

Above code is simply the implement of linear kernel and RBF

kernel.

```
def build_K(data1, data2):
    N1 = len(data1)
    N2 = len(data2)

    K = []
    for i in range(N1):
        tmp = []
        tmp.append(i+1)
        for j in range(N2):
            tmp.append(linear_kernel(data1[i], data2[j]) + RBF_kernel(data1[i], data2[j]))
        K.append(tmp)
        if i%500 ==0:
            print(i)

    return K
```

To use user-defined kernel, we will have to compute our own kernel matrix. Above code is the function to build the kernel matrix. Note that in each row, we have to put the index in the front.

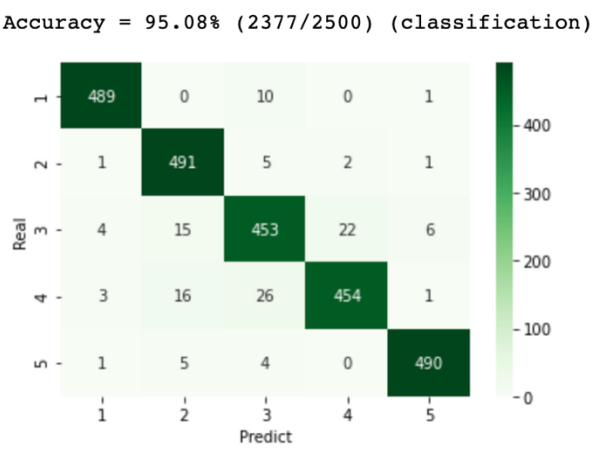
Finally, we could run our model:

```
K_train = build_K(X_train, X_train)
K_test = build_K(X_test, X_train)
model = svm_train(Y_train, K_train, '-t 4')
p_label, p_acc, p_val = svm_predict(Y_test, K_test, model)
```

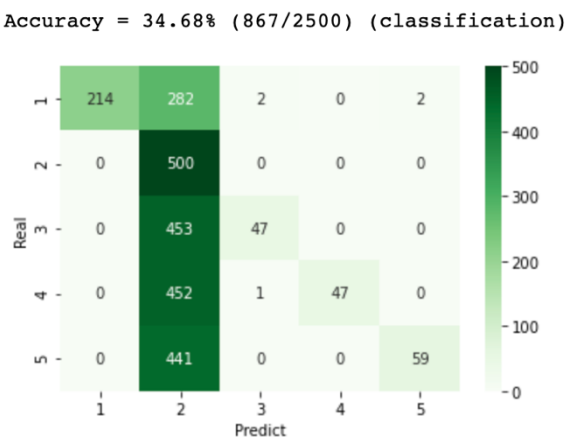
b. experiments setting and results

- Part1

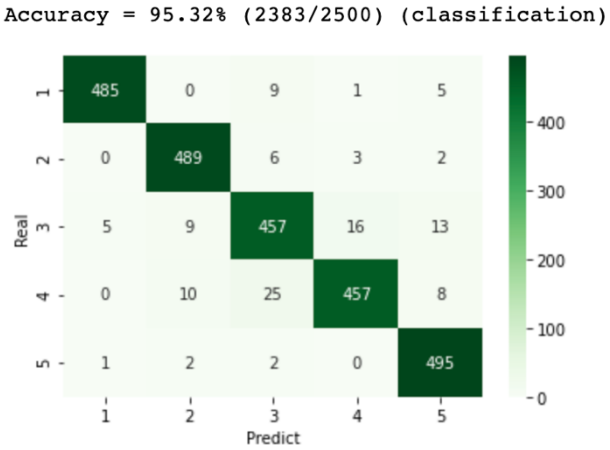
Linear



polynomial



RBF



According to heatmaps above, the performance of three different kernel type is RBF (Accuracy = 95.32) > Linear (Accuracy = 95.08) >>> polynomial (Accuracy = 34.68). Note that in polynomial kernel type prediction, the model has a high chance to predict the label is 2.

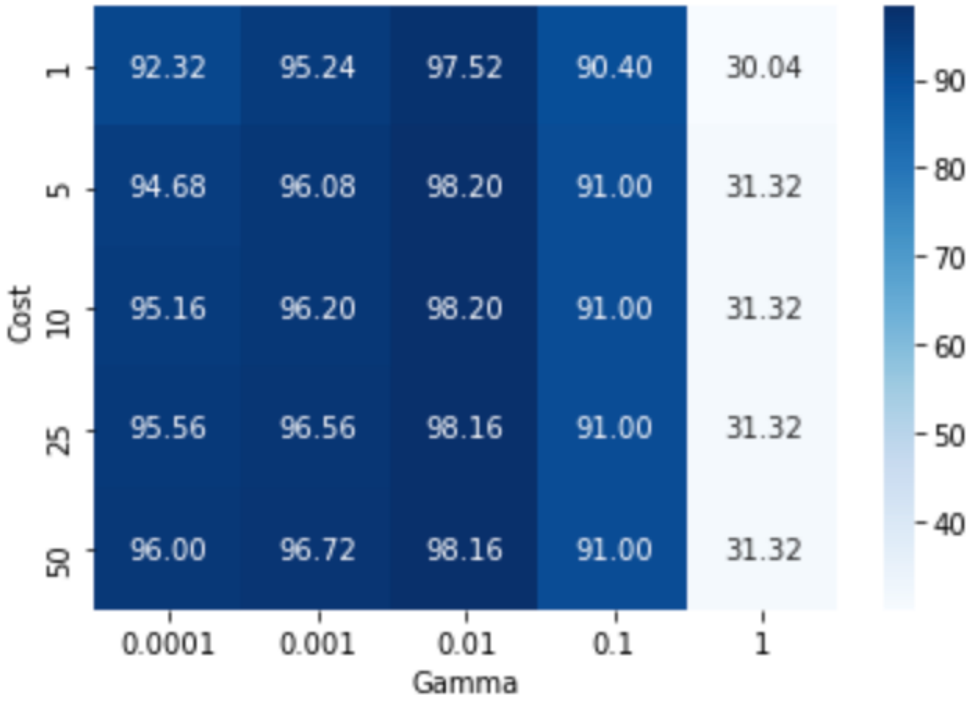
- Part2

We use “two layers” grid search. That’s mean we will run grid search algorithm two times, first time is global search, and the second time is local search base on first time result.

In first time, we set parameters(cost and gamma) like:

```
all_cost = [1, 5, 10, 25, 50]
all_gamma = [0.0001, 0.001,0.01,0.1,1]
grid_search_result1 = grid_search(all_cost, all_gamma)
```

Results:



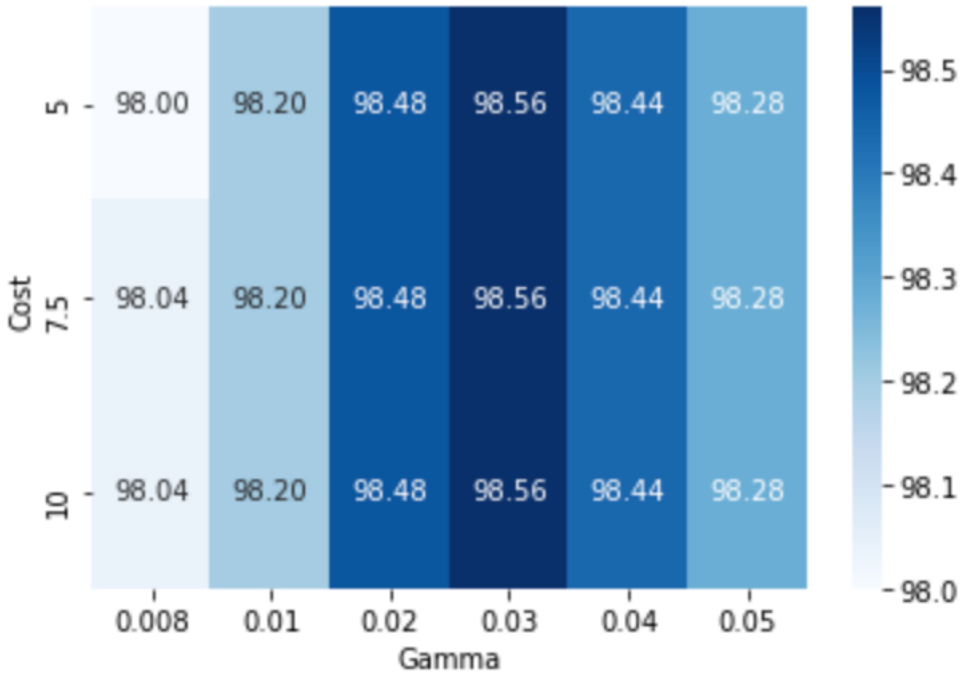
The number in the heatmap in each grid represent the accuracy of the corresponding Gamma(x-axis) and Cost(y-axis).

Obviously, when Gamma is about 0.01, and the Cost is in the range 5-10, the performance is better (because the color in the heatmap is much deeper).

Therefore, in next step, we reset the Cost and Gamma, and run our grid search function again:

```
all_cost = [5, 7.5, 10]
all_gamma = [0.008, 0.01, 0.02, 0.03, 0.04, 0.05]
grid_search_result2 = grid_search(all_cost, all_gamma)
```

Result:



According to above result, when Gamma = 0.03, Cost = 5, 7.5, or 10, we can get our best result which accuracy = 98.56.

c. observations and discussion

In Part1, we noticed that when we use polynomial kernel type, the prediction of the model is very uneven: the model has a high chance to predict the label is 2. I think the reason is that the training data isn't good enough, each kind of data is too identical and make our model would easily confuse during training.

In order to figure out which kind of data is the "worst data", which mean their features is most implicit and similar to other data, we ignore specific label of data during training each iteration, and compare model performance.

See the code below:

```
def ignore_some_data(n):
    new_train_Y = []
    new_train_X = []
    for i in range(len(Y_train)):
        if i <n*1000-1000 or i > n*1000-1:
            new_train_Y.append(Y_train[i])
            new_train_X.append(X_train[i])
    new_train_Y = np.array(new_train_Y)
    new_train_X = np.array(new_train_X)

    prob  = svm_problem(new_train_Y, new_train_X)
    param = svm_parameter('-t 0')
    model = svm_train(prob, param)
    p_label, p_acc, p_val = svm_predict(Y_test, X_test, model)
    return p_acc[0]
```

```
result_ = []
for i in range(1, 6):
    result_.append(ignore_some_data(i))
```

For example, if we run "ignore_some_data(2)", that means the data which label is 2 will all be ignore during training.

The result is shown below:

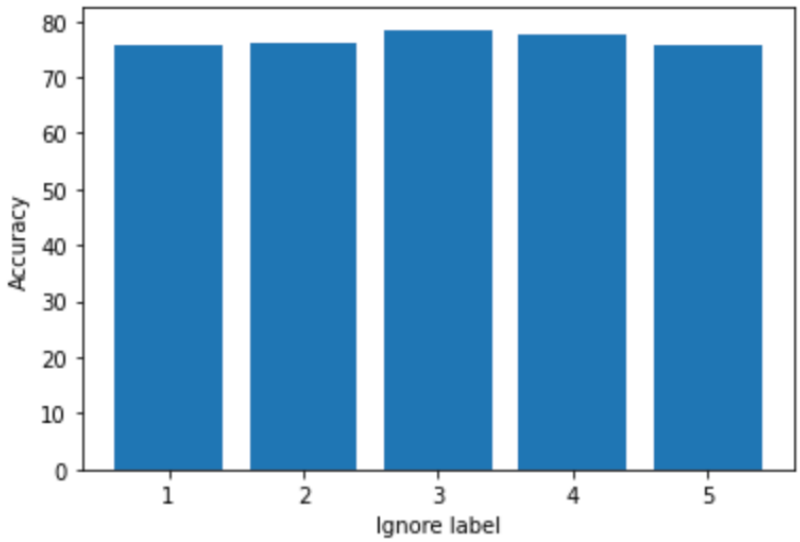
```
Ignore 1:
Accuracy = 75.76% (1894/2500) (classification)

Ignore 2:
Accuracy = 76.08% (1902/2500) (classification)

Ignore 3:
Accuracy = 78.48% (1962/2500) (classification)

Ignore 4:
Accuracy = 77.8% (1945/2500) (classification)

Ignore 5:
Accuracy = 75.64% (1891/2500) (classification)
```



According to the result, we can't easily decide which kind of data is the worst data, because the accuracy of each results is very close. However, compare to our result in Part1 (when the kernel type is polynomial, the accuracy is 34.68%), the accuracy is still extremely larger.