Machine Learning Homework 5

Gaussian Process & SVM

ChiaLiang Kuo 310552027

I. Gaussian Process

In this section, we are going to implement the Gaussian process and visualize the result. Given a matrix A34x2 data entries, each row corresponds to a 2D data point (xi, yi).

Code

main function

```
def main():
    # read raw data
    input_data = open('data/input.data', 'r')
    lines = input_data.readlines()
    data_list = np.array([[float(data.split(' ')[0]), float(data.split(' ')[1])]\
                          for data in lines])

    # config arguments
    x_samples = np.linspace(-60, 60, 1000)
    X = data_list[:, 0].reshape((data_list.shape[0], 1))
    Y = data_list[:, 1].reshape((data_list.shape[0], 1))
    beta = 5

    option = int(input('please input task id (1 or 2):'))

    if option == 1:
        # code for task 1.1
        task1(x_samples, X, Y, beta)

    elif option == 2:
        # code for task 1.2
        task2(x_samples, X, Y, beta)
```

Task 1.1 : Apply Gaussian Process Regression to predict the distribution of f

This section, we will use a rational quadratic kernel to compute similarities between different points, the formula is listed below :

- $k(x_a, x_b) = \sigma^2(1 + \frac{\|x_a - x_b\|^2}{2\alpha l^2})^{-\alpha}$

task1 function :

- `x_samples` : uniformly sampled 1000 points in [-60, 60]
- `X` : images list (N x 784)
- `Y` : label list (N x 1)
- `beta` : the variance of the normal distribution of the noise signal (this implementation will set beta to 5)

First, `VAR`, `L`, `ALPHA` were set to 1, which were the parameters of the rational quadractic kernel. Then, input `x_sample`, `X`, `Y`, `beta` and the parameters for kernel to the function `gaussian_process_regression` for computing the 95% confidence interval of f. Finally, the result figure would be shown and saved to `before_optimization.png` (see **Fig 1**). The implementation of `gaussian_process_regression` could be seen in the next code section.

```
def task1(x_samples, X, Y, beta):
    # hyper parameters
    kernel_args=(VAR, L, ALPHA)

    Y_up_95, Y_mean, Y_down_95 = \
        gaussian_process_regression(x_samples, X, Y, beta, kernel_args)

    plt.figure(figsize=(10, 6))
    plt.title(f'Before optimization (amplitude^2 = {VAR}, \
        lengthscale = {L}, alpha = {ALPHA})')
    plt.plot(x_samples, Y_up_95, c='#15dc15')
    plt.plot(x_samples, Y_mean, c='#8080fe')
    plt.plot(x_samples, Y_down_95, c='#15dc15')
    plt.fill_between(x_samples, Y_up_95, Y_down_95, color='#e0fce0')
    plt.scatter(X, Y, c='b', s=10)
    plt.xlabel('x')
    plt.ylabel('y')
    plt.savefig('before_optimization.png')
    plt.show()
```

gaussian_process_regression function :

- `x_samples` : uniformly sampled 1000 points in [-60, 60]
- `X` : images list (N x 784)
- `Y` : label list (N x 1)
- `beta` : the variance of the normal distribution of the noise signal (this implementation will set beta to 5)
- `kernel_args` : contain three parameters of rational quadratic kernel

The formulas of mean and variance of f are shown below.

- $\mu(x^*) = k(x, x^*)^T C^{-1} y$ where $C(x_n, x_m) = k(x_n, x_m) + \beta^{-1}\delta_{nm}$
- $\sigma^2(x^*) = k^* - k(x, x^*)^T C^{-1} k(x, x^*)$
  - where $k^* = k(x^*, x^*) + \beta^{-1}$

In order to compute the mean and variance of f, we need covariance matrix C and label y. After computing the C matrix, we could next compute the mean and variance of f for each new x by `k(x, x*)`, `k*`, `C`, `y` given. The 95% confidence interval would be in [-1.96 std, 1.96 std], so this function returned 3 lists : one for y_mean and two for both upper bound and lower bound of the confidence interval. The implementation of `quadratic_kernel_mat` was shown in next code section.

```
def gaussian_process_regression(x_samples, X, Y, beta, kernel_args):
    # get covariance matrix
    # c(n, m) = k(x_n, x_m) + 1/beta * delta(n, m)
    C = quadratic_kernel_mat(X, kernel_args) \
        + (1/beta) * np.identity(X.shape[0])
    assert np.linalg.det(C) != 0, 'C is not invertible'

    Y_up_95,Y_mean, Y_down_95 = [], [], []
    for x_sample in x_samples:

        # k(x, x*)
        k_x_xstar = np.array(\
             [rational_quadratic_kernel(x_sample, x, kernel_args) for x in X])
```

```
        # k* = k(x*, x*) + 1/beta
        k_star = rational_quadratic_kernel(x_sample, x_sample, kernel_args) \
            + 1/beta

        # compute mu(x_star), std(x_star)
        mean = ((k_x_xstar.T @ np.linalg.inv(C)) @ Y)[0,0]
        std = (k_star - (k_x_xstar.T @ np.linalg.inv(C)) @ k_x_xstar)[0,0]

        Y_up_95.append(mean + 1.96 * (std**0.5))
        Y_mean.append(mean)
        Y_down_95.append(mean - 1.96 * (std**0.5))

    return Y_up_95, Y_mean, Y_down_95
```

rational quadratic kernel function:

There are two different version for computing the kernel :

- `rational_quadratic_kernel` took single (x, y) pair as input
- `rational_quadratic_kernel_mat` took the whole X and compute the gram matrix

These two functions implemented rational quadratic kernel, and the formula of this kernel was :

- $k(x_a, x_b) = \sigma^2(1 + \frac{\|x_a - x_b\|^2}{2\alpha l^2})^{-\alpha}$

```
def quadratic_kernel(x_n, x_m, kernel_args):
    (var, l, alpha) = kernel_args

    # rational quadratic kernel
    return var * \
        (1 + ((x_n - x_m)**2 / (2 * alpha * l**2)) ** (-alpha)

def quadratic_kernel_mat(X, kernel_args):
    (var, l, alpha) = kernel_args
    X_repeat_h = np.repeat(X, X.shape[0], axis=1) # (n, 1) ->  (n, n)
    X_repeat_v = np.repeat(X.T, X.shape[0], axis=0) # (1, n) -> (n, n)

    # rational quadratic kernel matrix
    return var * (1 + ((X_repeat_h - X_repeat_v)**2 \
                       (2 * alpha * l**2))) ** (-alpha)
```

Task 1.2 : Optimize the Kernel Parameters by minimizing negative marginal log-likelihood

- The marginal likelihood function :

$$p(y) = \int p(y|\mathrm{f})p(\mathrm{f})\mathrm{df} = N(y|0, \mathrm{C}), \text{ where } \mathrm{C}(x_n, x_m) = k(x_n, x_m) + \beta^{-1}\delta_{nm}$$

- Next, compute the log of the function above :

$$ln(p(y)) = ln(det(2\pi C)^{-\frac{1}{2}} \times e^{-\frac{1}{2}(y^T C^{-1} y)})$$

$$= -\frac{1}{2}ln(det(C)) - \frac{n}{2}ln(2\pi) - \frac{1}{2}(y^T C^{-1} y)$$

task2 function

- `x_samples` : uniformly sampled 1000 points in [-60, 60]
- `X` : images list (34 x 1)
- `Y` : label list (34 x 1)
- `beta` : the variance of the normal distribution of the noise signal (this implementation will set beta to 5)

To complete task2, I applied `scipy.optimize.minimize` to minimize the negative marginal log-likelihood function. Because an initial guess for three parameters was needed for this method, I set the initial three parameters the same as **Task 1**.

```
def task2(x_samples, X, Y, beta):

    # [VAR, L, ALPHA] = [1, 1, 1]
    # run optimization by initial guess given
    result = optimize.minimize(negative_marginal_log_likelihood, \
                               [VAR, L, ALPHA], args=(-1, X, Y, beta))

    [var, l, alpha] = result.x
    kernel_args=(var, l, alpha)
    Y_up_95, Y_mean, Y_down_95 = \
        gaussian_process_regression(x_samples, X, Y, beta, kernel_args)
    after_optimization = \
        negative_marginal_log_likelihood([var, l, alpha], 1, X, Y, beta)

    plt.figure(figsize=(10, 6))
    plt.title(f'After optimization (amplitude^2 = {np.round(100 * var) / 100}, \
        lengthscale = {np.round(100 * l) / 100}, \
        alpha = {np.round(100 * alpha) / 100})')
    plt.plot(x_samples, Y_up_95, c='#15dc15')
    plt.plot(x_samples, Y_mean, c='#8080fe')
    plt.plot(x_samples, Y_down_95, c='#15dc15')
    plt.fill_between(x_samples, Y_up_95, Y_down_95, color='#e0fce0')
    plt.scatter(X, Y, c='b', s=10)
    plt.xlabel('x')
    plt.ylabel('y')
    plt.savefig('after_optimization.png')
    plt.show()
```

Experiments

Task1

Fig 1 shown the regression result with 3 parameters for computing kernel (VAR, L, ALPHA) set to 1

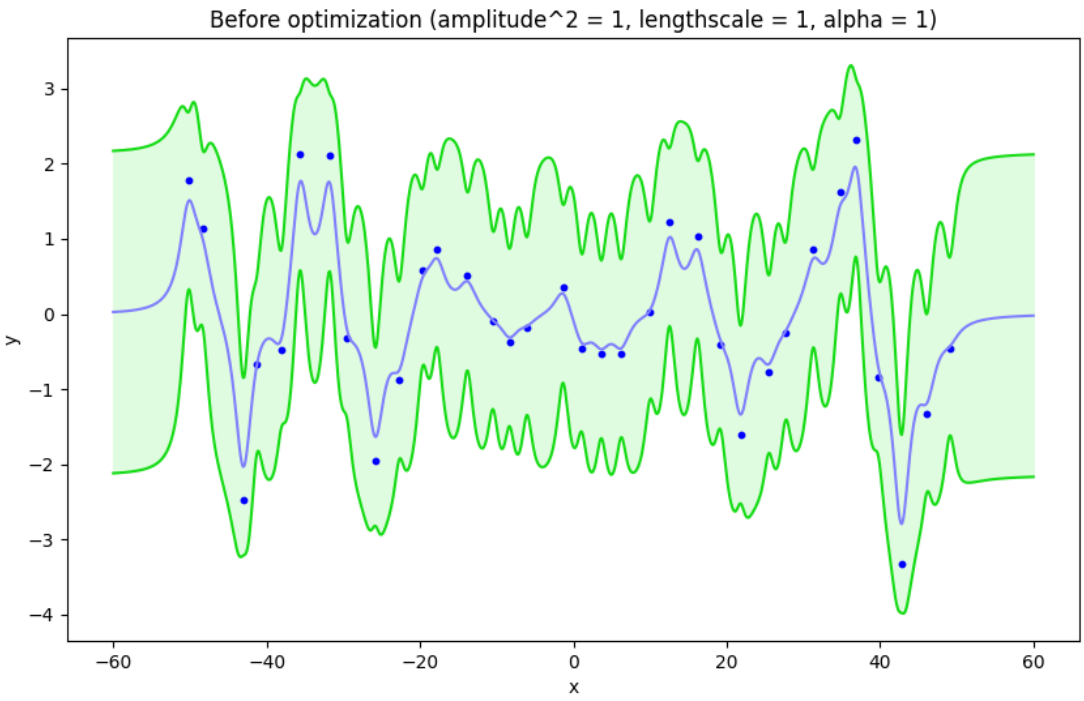

Fig 1. the regression result with 3 parameters set to 1:

The parameters of rational quadratic kernel and the log likelihood of `p(y)`

```
var   : 1.000
l     : 1.000
alpha : 1.000
log likelihood : -78.088
```

Task2

Before optimization, 3 parameters for initial guess of the kernel function (VAR, L, ALPHA) were set to 1. After optimization process, the result figure was shown in **Fig 2**:

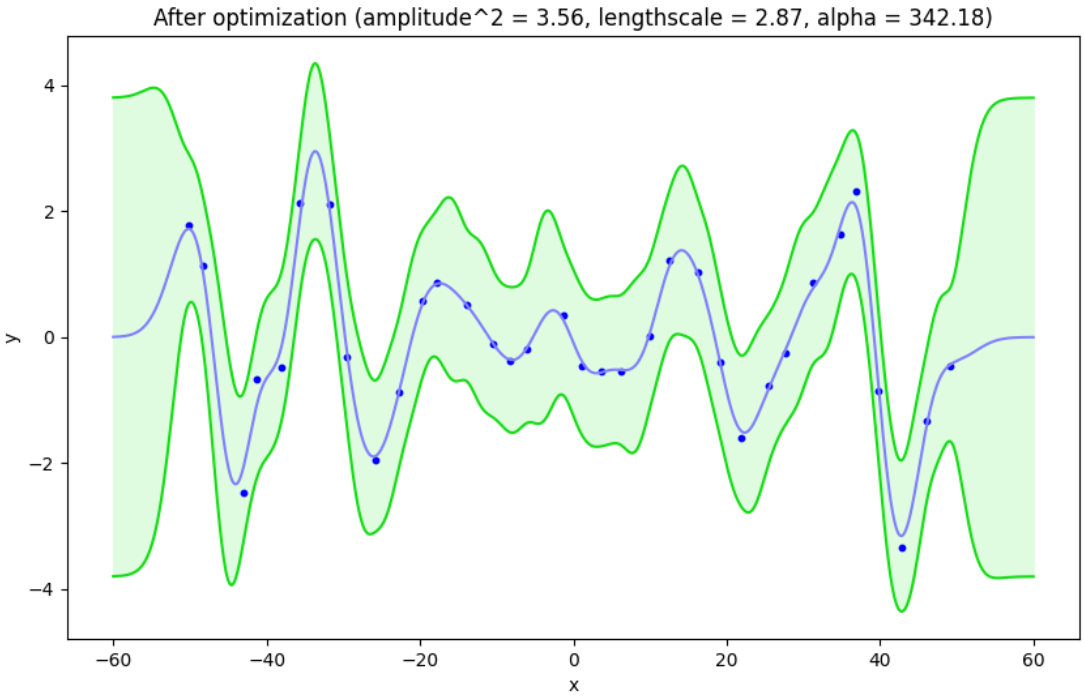

Fig 2. the regression result with 3 parameters optimized

The parameters of rational quadratic kernel and the log likelihood of `p(y)` after optimization:

```
var   : 3.560
l     : 2.872
alpha : 342.185
log likelihood : -63.048
```

Observations and Discussion

From **Fig 1** and **Fig 2**, we could know that after the optimization process, the curves of the regression result will be more smooth than that not been optimized. Besides, the log likehood of the result become much larger, and the 95% confidence interval was also narrower. I also tried different initial parameters of the kernel function, and the optimized result are all the same.

II. SVM on MNIST

In this section, we are going to use SVM models to tackle classification on images of hand-written digits (digit class only ranges from digit 0 to 4, as **Fig 3** shown below). Each image in this dataset is a 28x28 gray-scale image, and the training and testing set are all .csv files that contain 5000 and 2500 images respectively.

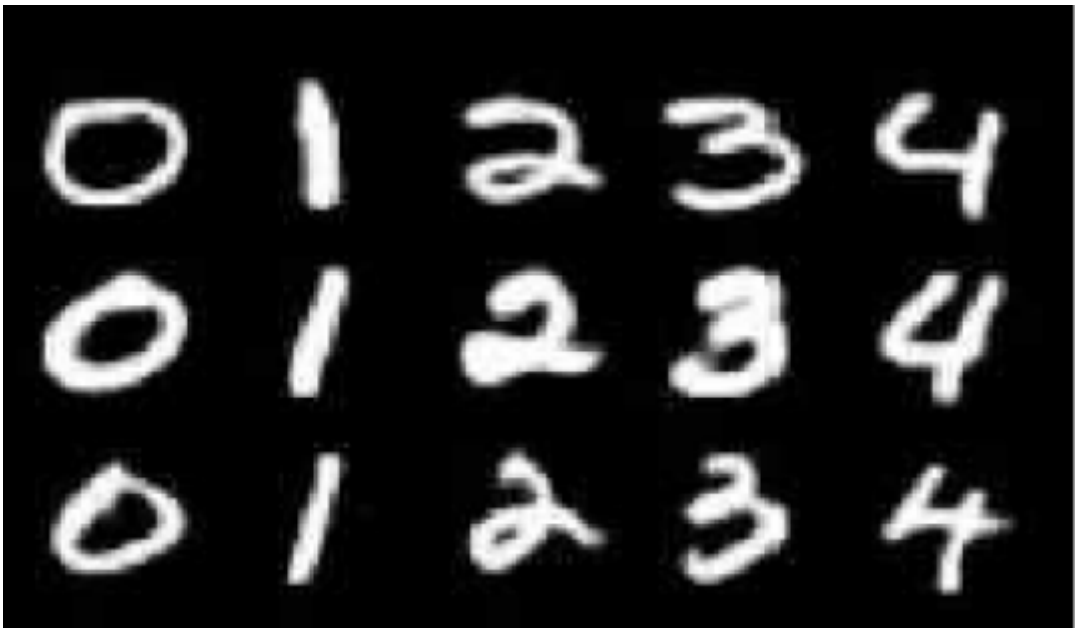

Fig 3. the subset of the MNIST dataset (only from digit 0 to 4)

Code

Task 2.1 : Use different kernel functions (linear, polynomial, and RBF kernels) and compare their performance.

The formulas of three different kernels are shown below. In this section, I set the hyperparameters as default, and set the parameter `c` (cost) to 10.

- linear kernel

$$k(u, v) = u^T \cdot v$$

- polynomial kernel (`gamma` = 1/num_features, `coef0`= 0, `degree` = 3)

$$k(u, v) = (\gamma \times u^T \cdot v + C_0)^d$$

- RBS (`gamma` = 1/num_features)

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

task1 function

- `X_train` : training images (5000 x 784)
- `Y_train` : training lebels (5000 x 1)
- `X_test` : testing images (2500 x 784)
- `Y_test` : testing labels (2500 x 1)

To compare the results of the three functions, I applied 11 different cost parameter `c` from 0.00001 to 100000 (latter `c` is 10 times larger than former `c`) in each iteration which run three SVM processes by applying `linear`, `polynomial`, `RBF` kernels with the same cost parameter `c` respectively. (I ignored some code for plotting the results stored in `acc_map`.)

```
def task1(X_train, Y_train, X_test, Y_test):
    s_dict = {'C-SVC': 0}
    t_dict = {'linear': 0, 'polynomial':1, 'RBF':2}

    f_out = open('task2.1_out.txt', 'w')
```

```
    c_default = 1
    c_list = [c_default / 100000, c_default / 10000, c_default / 1000, c_default /
100, c_default / 10, c_default, c_default * 10, c_default * 100, c_default * 1000,
c_default * 10000, c_default / 100000]
    acc_map = np.zeros((3, len(c_list)))

    for i, c in enumerate(c_list):
        # -s 0 (C-SVC) -t 0 (linear)
        kernel_type = 'linear'
        param = '-s {} -t {} -c {}'.format(s_dict['C-SVC'], t_dict[kernel_type], c)
        m = svm_train(Y_train, X_train, param)
        p_label, p_acc, p_val = svm_predict(Y_test, X_test, m)
        ACC, MSE, SCC = evaluations(Y_test, p_label)
        acc_map[0, i] = ACC
        f_out.write('[kernel type = {}, c = {}, ACC={}, MSE={}, SCC=
{}]\n'.format(kernel_type, c, ACC, MSE, SCC))

        # -s 0 (C-SVC) -t 1 (polynomial)
        kernel_type = 'polynomial'
        param = '-s {} -t {} -c {}'.format(s_dict['C-SVC'], t_dict[kernel_type], c)
        m = svm_train(Y_train, X_train, param)
        p_label, p_acc, p_val = svm_predict(Y_test, X_test, m)
        ACC, MSE, SCC = evaluations(Y_test, p_label)
        f_out.write('[kernel type = {}, c = {}, ACC={}, MSE={}, SCC=
{}]\n'.format(kernel_type, c, ACC, MSE, SCC))
        acc_map[1, i] = ACC

        # -s 0 (C-SVC) -t 2 (RBF)
        kernel_type = 'RBF'
        param = '-s {} -t {} -c {}'.format(s_dict['C-SVC'], t_dict[kernel_type], c)
        m = svm_train(Y_train, X_train, param)
        p_label, p_acc, p_val = svm_predict(Y_test, X_test, m)
        ACC, MSE, SCC = evaluations(Y_test, p_label)
        f_out.write('[kernel type = {}, c = {}, ACC={}, MSE={}, SCC=
{}]\n'.format(kernel_type, c, ACC, MSE, SCC))
        acc_map[2, i] = ACC

    # code for plotting results of acc_map
    ...
```

task2 function

- `X_train` : training images (5000 x 784)
- `Y_train` : training lebels (5000 x 1)
- `X_test` : testing images (2500 x 784)
- `Y_test` : testing labels (2500 x 1)

In this section, we will do the grid search for finding parameters of the best performing model.

- In polynomial kernel, there are 4 different parameters we can adjust (`c` for cost, `gamma` for the coefficient of the dot product term, `coef0` for the constant term, `degree` for the power term)
- In RBF kernel, there are 2 different parameters we can adjust (`c` for cost, `gamma` for the coefficient of the dot product term)

The effect of `c` and `gamma` is described in many online sources, I refer the online documentation of scikit-learn **[1]** :

- **`gamma`** : defines how far the influence of a single training example reaches, with low values meaning 'far' and high values meaning 'close'. The gamma parameters can be seen as the inverse of the radius of influence of samples selected by the model as support vectors.
- **`c`** : parameter trades off correct classification of training examples against maximization of the decision function's margin. For larger values of `c`, a smaller margin will be accepted if the decision function is better at classifying all training points correctly A lower `c` will encourage a larger margin, therefore a simpler decision function, at the cost of training accuracy. In other words `c` behaves as a regularization parameter in the SVM.

To do the grid research, I set different values for the correspondent parameters and ran SVM process for every pair of them. Finally, I would plot the result as several heatmaps (see experiment section). (I ignored some code for plotting the results stored in `polynomial_acc_map` and `RBF_acc_map`.)

polynomial :

- degree : `[2, 3, 4]`
- gamma : from `1.25e-8` to `125` (the latter value is 10 times larger than the former value)
- coef0 : `[-1000, -100, -10, -1, -0.1, 0, 0.1, 1, 10, 100, 1000]`
- c : `[0.1, 1, 10]`

RBF :

- gamma : from `1.25e-8` to `125` (the latter value is 10 times larger than the former value)
- c : from `0.00001` to `100000` (the latter value is 10 times larger than the former value)

```
def task2(X_train, Y_train, X_test, Y_test):
    # for polynomial (gamma*u'*v + coef0)^degree
    gamma_default = 0.00125
    coef_default = 0
    degree_default = 3
    c_default = 1 # will set to 0.1, 1, 10 and run this function 3 times

    f_out = open('task2.2_out.txt', 'w')

    degree_list = [degree_default - 1, degree_default, degree_default + 1]
    gamma_list = [gamma_default / 100000, gamma_default / 10000, gamma_default /
1000, gamma_default / 100, gamma_default / 10, gamma_default, gamma_default * 10,
gamma_default * 100, gamma_default * 1000, gamma_default * 10000, gamma_default *
100000]
    coef_list = [coef_default - 1000, coef_default - 100, coef_default - 10,
coef_default - 1, coef_default - 0.1, coef_default, coef_default + 0.1, coef_default
+ 1, coef_default + 10, coef_default + 100, coef_default + 100]
    polynomial_acc_map = np.zeros((len(degree_list), len(gamma_list),
len(coef_list)))

    # -s 0 (C-SVC) -t 1 (polynomial) -c 10
    for i, degree in enumerate(degree_list):
        for j, gamma in enumerate(gamma_list):
            for k, coef in enumerate(coef_list):
                print(f'[degree = {degree}, coef = {coef},gamma = {gamma}]\n')
                param = '-s 0 -t 1 -c 10 -g {} -r {} -d {}'.format(gamma, coef,
degree)
                m = svm_train(Y_train, X_train, param)
                p_label, p_acc, p_val = svm_predict(Y_test, X_test, m)
                ACC, MSE, SCC = evaluations(Y_test, p_label)
                f_out.write(f'[param : {param}, ACC = {ACC}, MSE = {MSE}, SCC =
{SCC}]\n')
                polynomial_acc_map[i, j, k] = np.round(100 * ACC) / 100

    f_out.write('\nPolynomial Based Kernel Analysis\n')
    for i in range(polynomial_acc_map.shape[0]):
        f_out.write(f'\ndegree = {degree_list[i]}\n')
        for j in range(polynomial_acc_map.shape[1]):
            for k in range(polynomial_acc_map.shape[2]):
                f_out.write(f'{polynomial_acc_map[i, j, k]} ')
            f_out.write('\n')
    f_out.write('\n')

    # for RBF exp(-gamma*|u-v|^2)
    gamma_list = [gamma_default / 100000, gamma_default / 10000, gamma_default /
1000, gamma_default / 100, gamma_default / 10, gamma_default, gamma_default * 10,
gamma_default * 100, gamma_default * 1000, gamma_default * 10000, gamma_default *
```

```
100000]
    c_list = [c_default / 100000, c_default / 10000, c_default / 1000, c_default /
100, c_default / 10, c_default, c_default * 10, c_default * 100, c_default * 1000,
c_default * 10000, c_default / 100000]
    RBF_acc_map = np.zeros((len(gamma_list), len(c_list)))

    # -s 0 (C-SVC) -t 2 (RBF)
    for i, gamma in enumerate(gamma_list):
        for j, c in enumerate(c_list):
            print(f'[gamma = {gamma}, c = {c}]\n')
            param = '-s 0 -t 2 -c {} -g {}'.format(c, gamma)
            m = svm_train(Y_train, X_train, param)
            p_label, p_acc, p_val = svm_predict(Y_test, X_test, m)
            ACC, MSE, SCC = evaluations(Y_test, p_label)
            f_out.write(f'[param : {param}, ACC = {ACC}, MSE = {MSE}, SCC =
{SCC}]\n')
            RBF_acc_map[i, j] = np.round(100 * ACC) / 100

    f_out.write('\nRBF Based Kernel Analysis\n')
    for i in range(RBF_acc_map.shape[0]):
        for j in range(RBF_acc_map.shape[1]):
            f_out.write(f'{RBF_acc_map[i, j]} ')
        f_out.write('\n')
    f_out.write('\n')

    # plot result to heatmap
    plot_res(polynomial_acc_map, 'polynomial', degree_list, gamma_list, coef_list)
    plot_res(RBF_acc_map, 'RBF', gamma_list, c_list, None)

    print('process completed.')
```

Task 2.3

This section we will use linear kernel + RBF kernel together (form a new kernel function) and compare the performance with respect to others (linear, RBF).

kernel function

- `kernel_linear` : `gamma` is dummy value, it is actually not used
- `kernel_RBF` : dist is `scipy.spatial.distance` and will return the RBF result by `x`, `y`, `gamma`.
- `kernel_linear_plus_RBF` : plus `kernel_linear` and `kernel_RBF` together.

This function computed the gram matrix of the input `x` and `y.

```
def kernel_linear(x, y, gamma):
    return x @ y.T

def kernel_RBF(x, y, gamma):
    return np.exp(-gamma * dist.cdist(x, y))

def kernel_linear_plus_RBF(x, y, gamma):
    return kernel_linear(x, y, gamma) + kernel_RBF(x, y, gamma)
```

kernelize function

- `X_train` : training images (5000 x 784)
- `X_test` : testing images (2500 x 784)
- `func` : kernel function
- `gamma` : the parameter of kernel function

This function pre-computed the gram matrix of (`X_train`, `X_train`) and (`X_test`, `X_train`) and stored the result as the `svm_train` intput format described in the comment below.

```
def kernelize(X_train, X_test, func, gamma):
    '''
    link : https://github.com/cjlin1/libsvm, keyword : precomputed kernel
    Assume the original training data has three four-feature
    instances and testing data has one instance:

    15  1:1 2:1 3:1 4:1
    45      2:3     4:3
    25          3:1

    15  1:1     3:1

    If the linear kernel is used, we have the following new
    training/testing sets:8

    15  0:1 1:4 2:6  3:1
    45  0:2 1:6 2:18 3:0
    25  0:3 1:1 2:0  3:1

    15  0:? 1:2 2:0  3:1

    ? can be any value.
    '''

    print('computing kernel for training')
    training_kernel = func(X_train, X_train, gamma)
    testing_kernel = func(X_test, X_train, gamma)

    training_kernelized = []
    for i in range(training_kernel.shape[0]):
        training_dict = {}
        training_dict[0] = i+1
        for j in range(training_kernel.shape[1]):
            training_dict[j+1] = training_kernel[i, j]
        training_kernelized.append(training_dict)

    testing_kernelized = []
    for i in range(testing_kernel.shape[0]):
        testing_dict = {}
        testing_dict[0] = i+1
        for j in range(testing_kernel.shape[1]):
            testing_dict[j+1] = testing_kernel[i, j]
        testing_kernelized.append(testing_dict)

    return training_kernelized, testing_kernelized
```

task3 function

- `X_train` : training images (5000 x 784)
- `Y_train` : training lebels (5000 x 1)
- `X_test` : testing images (2500 x 784)
- `Y_test` : testing labels (2500 x 1)

This function ran the grid research that take different `gamma` values and compared **linear + RBF kernel** , **linear kernel** , and **RBF kernel** with the same `gamma` in each iteration. Finally, the accuracy results would be shown in **Fig 14** (see experiment section).

- gamma : from `1.25e-8` to `125` (the latter value is 10 times larger than the former value)

```
def task3(X_train, Y_train, X_test, Y_test):

    gamma_default = 0.00125
    gamma_list = [gamma_default / 100000, gamma_default / 10000, gamma_default /
1000, gamma_default / 100, gamma_default / 10, gamma_default, gamma_default * 10,
gamma_default * 100, gamma_default * 1000, gamma_default * 10000, gamma_default *
100000]
    func_list = [kernel_linear, kernel_RBF, kernel_linear_plus_RBF]
    acc_map = np.zeros((len(func_list), len(gamma_list)))

    # precompute kernel

    f_out = open('task2.3_out.txt', 'w')

    # run for all gamma
    for i, func in enumerate(func_list):
        for j, gamma in enumerate(gamma_list):
            X_train_kenelized, X_test_kernelized = kernelize(X_train, X_test, func,
gamma)
            param = '-s 0 -t 4 -c 1 -g {}'.format(gamma)
            m = svm_train(Y_train, X_train_kenelized, param)
            p_label, p_acc, p_val = svm_predict(Y_test, X_test_kernelized, m)
            ACC, MSE, SCC = evaluations(Y_test, p_label)
            f_out.write(f'[kernel : {func}, gamma : {gamma}, ACC = {ACC}, MSE =
{MSE}, SCC = {SCC}]\n')
            acc_map[i, j] = np.round(100 * ACC) / 100

    # code for plotting the result using acc_map
    ...
```

Experiment

Task 2.1

As described in **Code** section, I set 11 different cost values `c` and ran SVM for three kernel functions (linear, polynomial, RBF repectively) in each iteration, and the results were shown in **Fig 4**. The bar graph cantained all the testing accuracies for each SVM process, and the highest one was that apply polynomial kernel with cost value `c` set to 1 (ACC = 97.6%), and the lowest one was that apply polynomial kernel with cost value set higher than 1000 or lower than 1 (ACC = 28.9%).

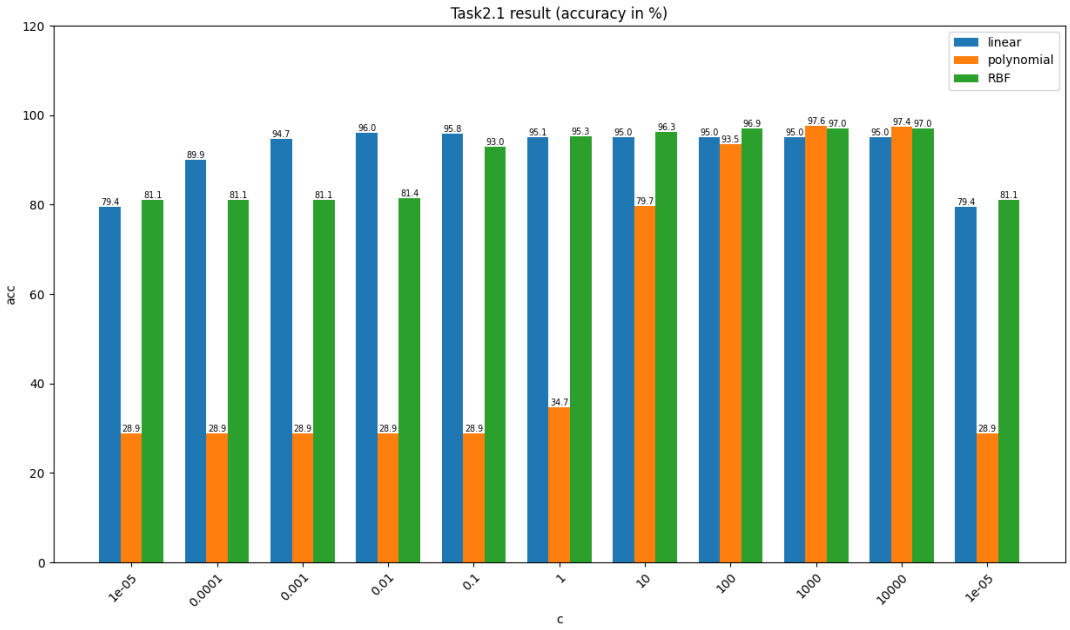

Fig 4. the testing accuracy of three different kernels with 11 cost values

Task 2.2

In this section we would do the grid research on the different parameters of the kernel functions. Because there was no parameter in linear kernel, I only did the research for polynomial and RBF based kernel. **Fig 5-13** were for polynomial kernel with `c` set to 0.1, 1, 10 and `degree` parameter set to 2, 3, 4 respectively, while **Fig 14** was for RBF kernel. As discribed in **Code** section, I applied 11 `coef` and `gamma` parameters in polynomial kernel, and applied 11 `gamma` and `c` parameters in RBF kernel.

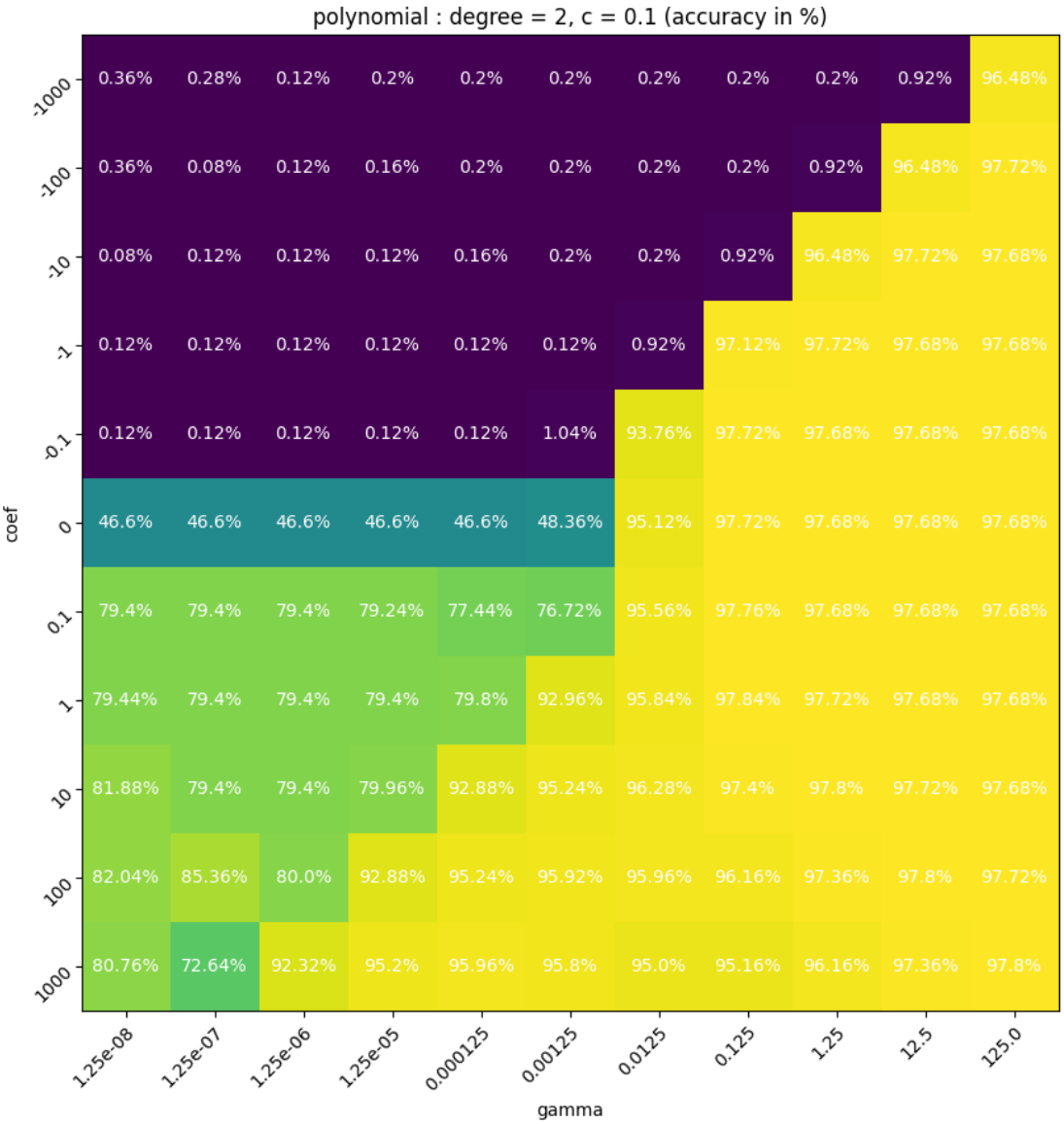

Fig 5. the testing accuracy of polynomial kernel with `degree` = 2 and `c`=0.1

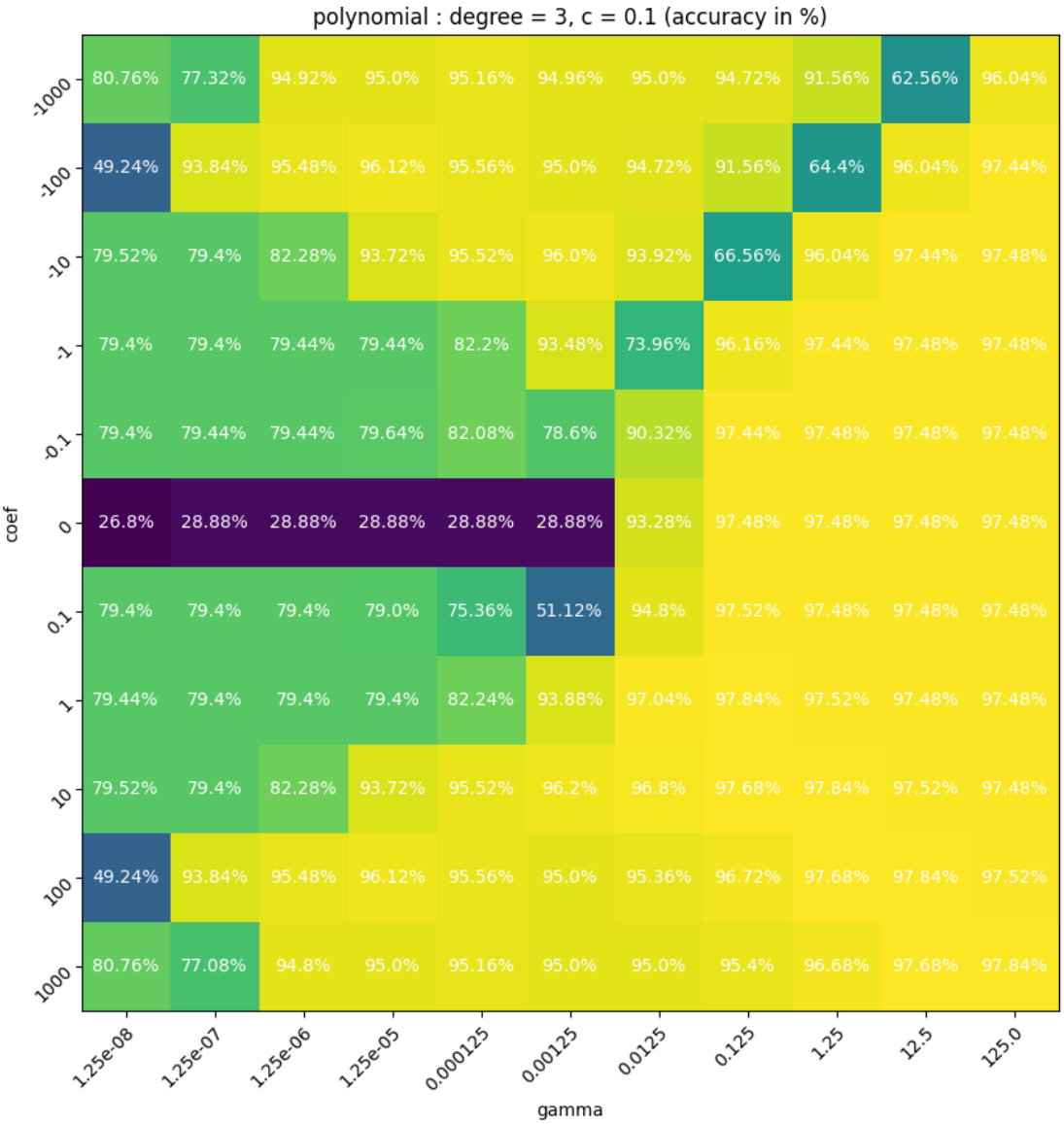

Fig 6. the testing accuracy of polynomial kernel with `degree` = 3 and `c`=0.1

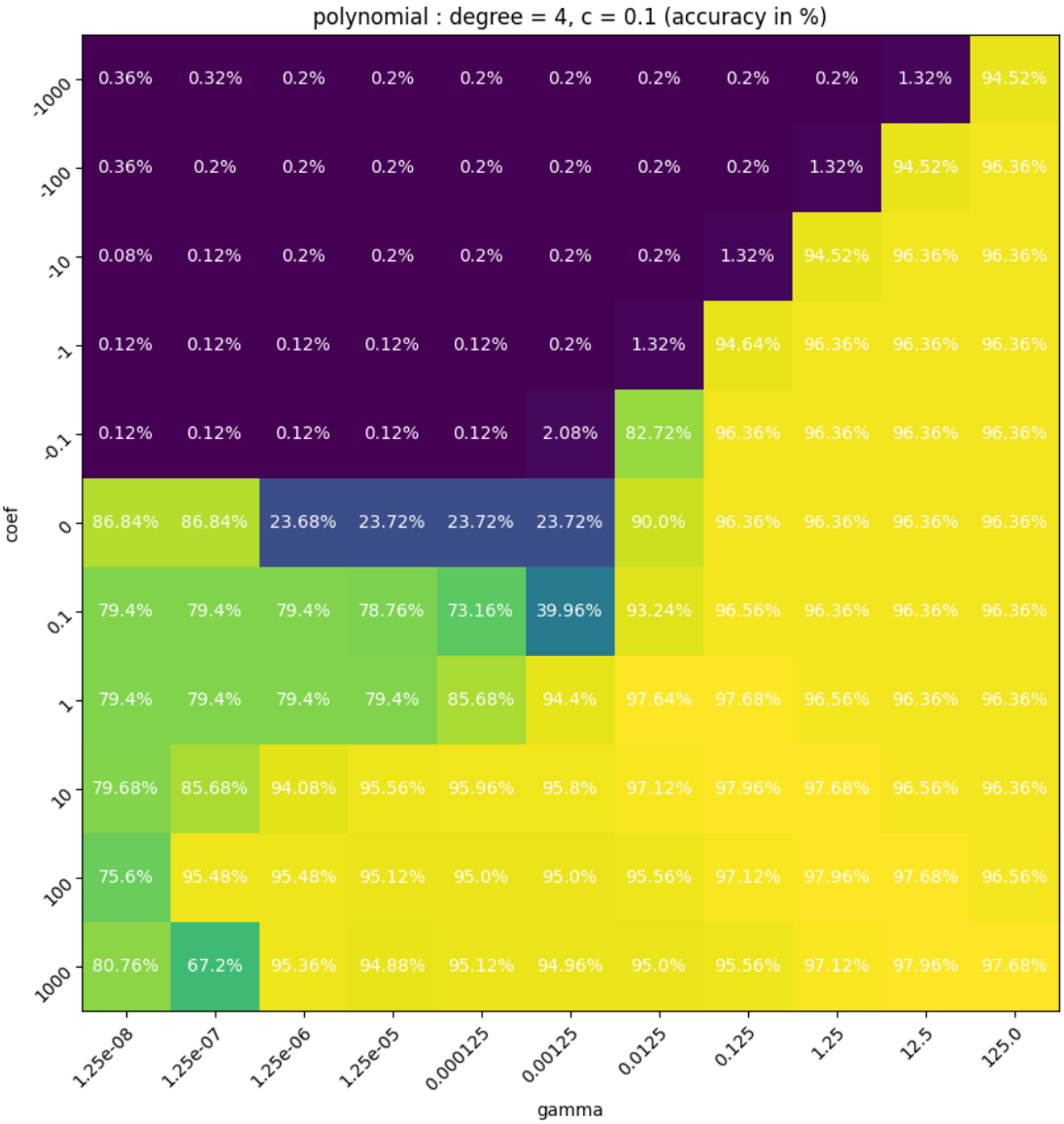

Fig 7. the testing accuracy of polynomial kernel with `degree` = 4 and `c`=0.1

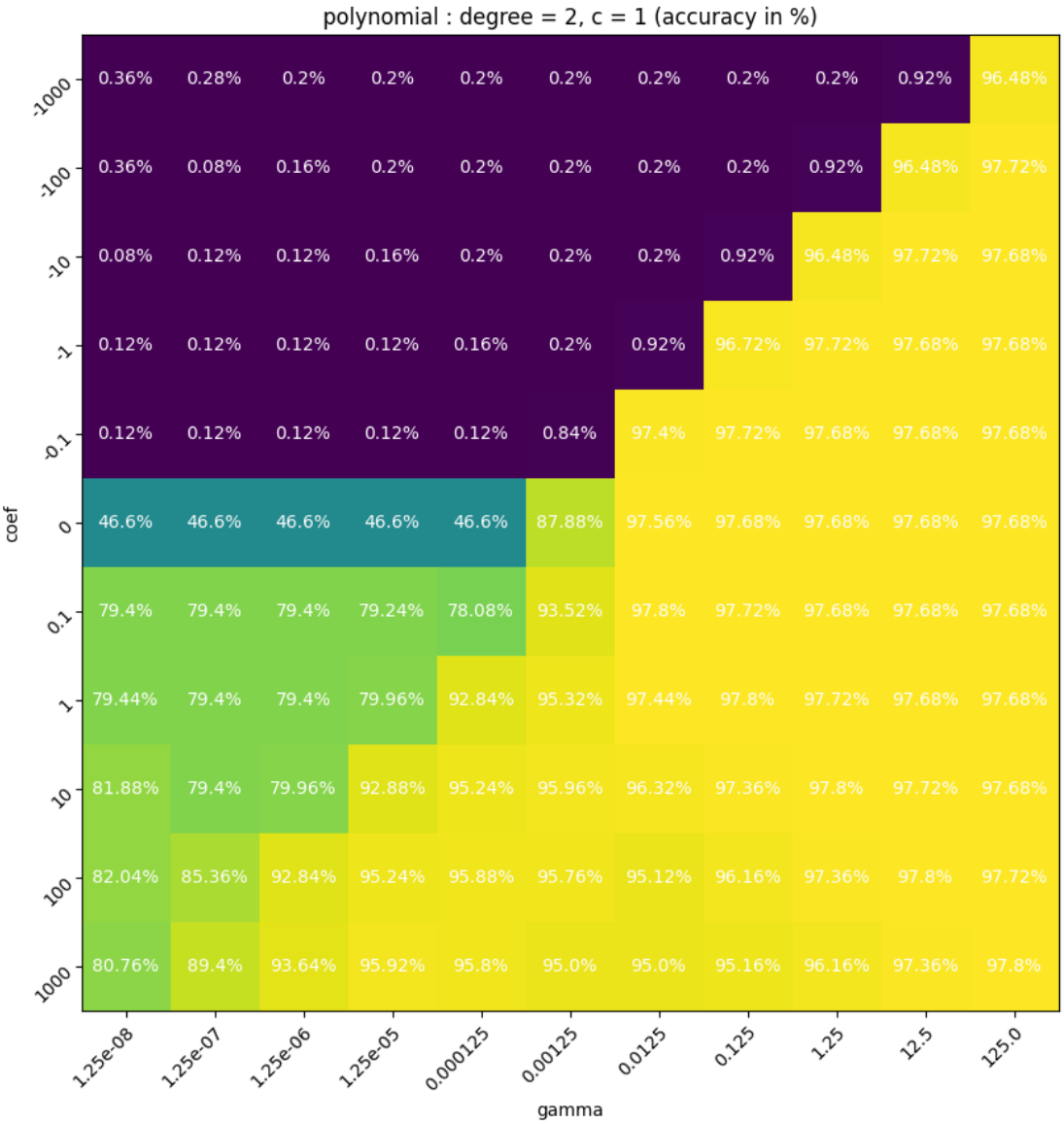

Fig 8. the testing accuracy of polynomial kernel with `degree` = 2 and `c`=1

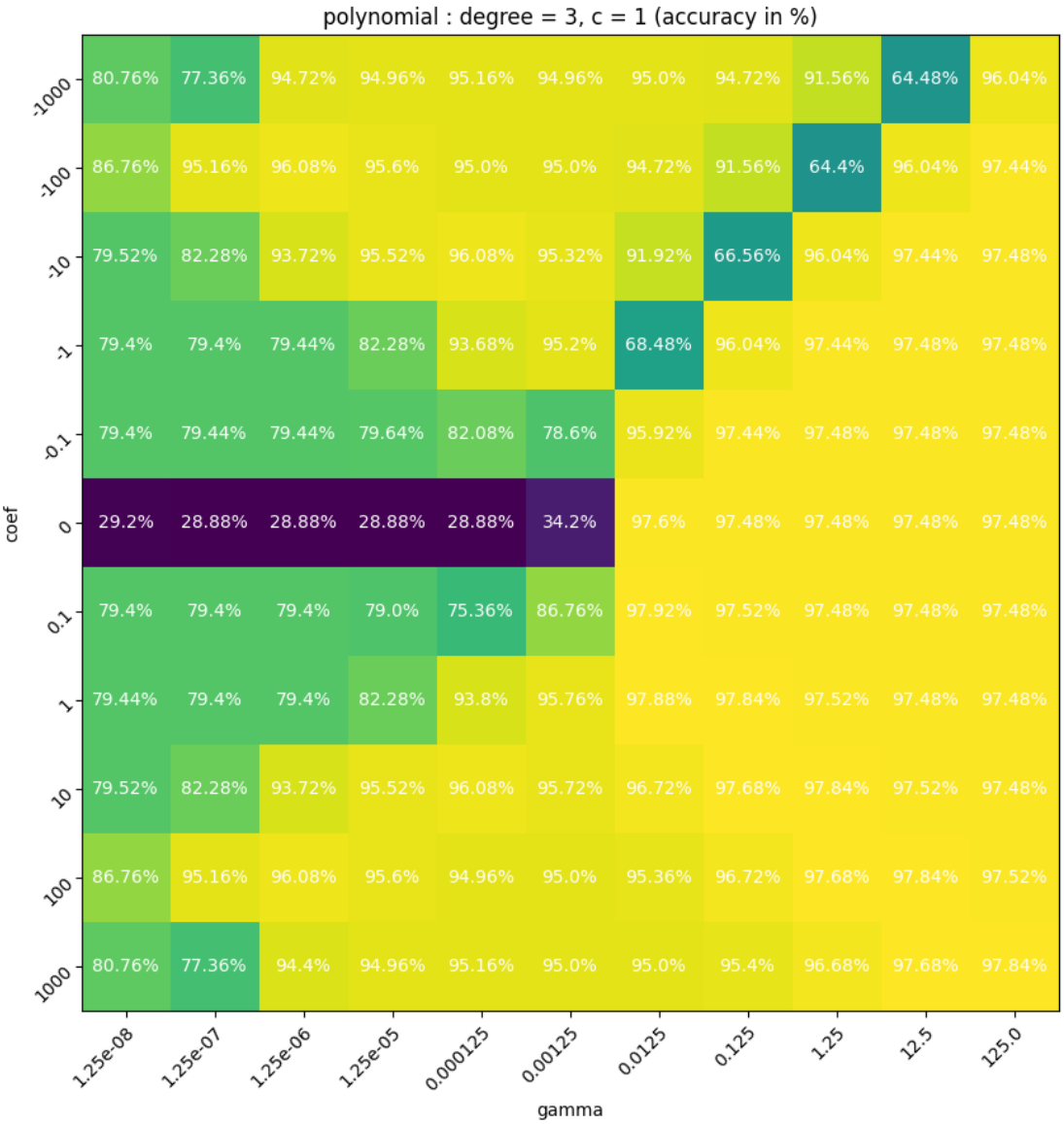

Fig 9. the testing accuracy of polynomial kernel with `degree` = 3 and `c`=1

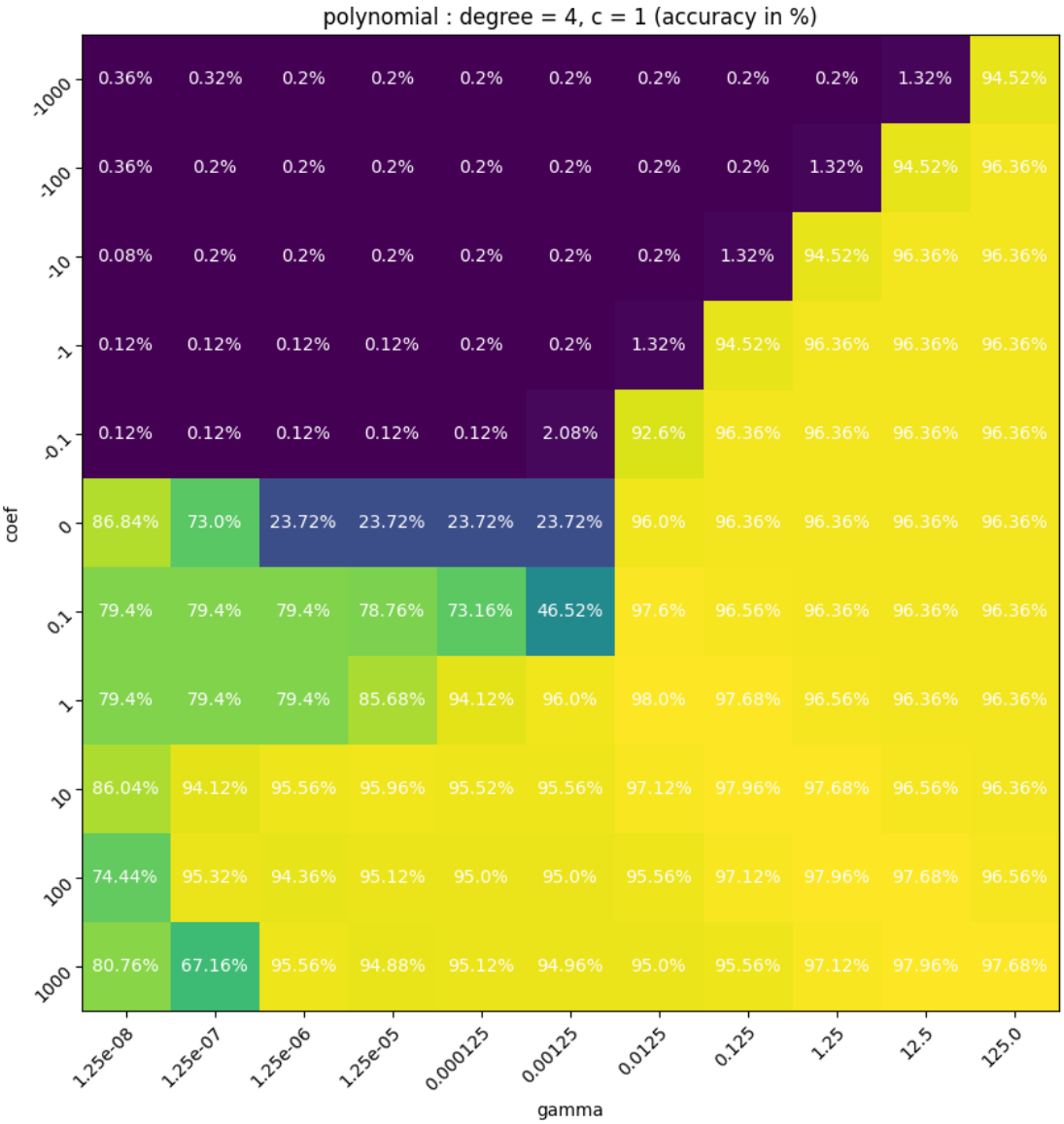

Fig 10. the testing accuracy of polynomial kernel with `degree` = 4 and `c`=1

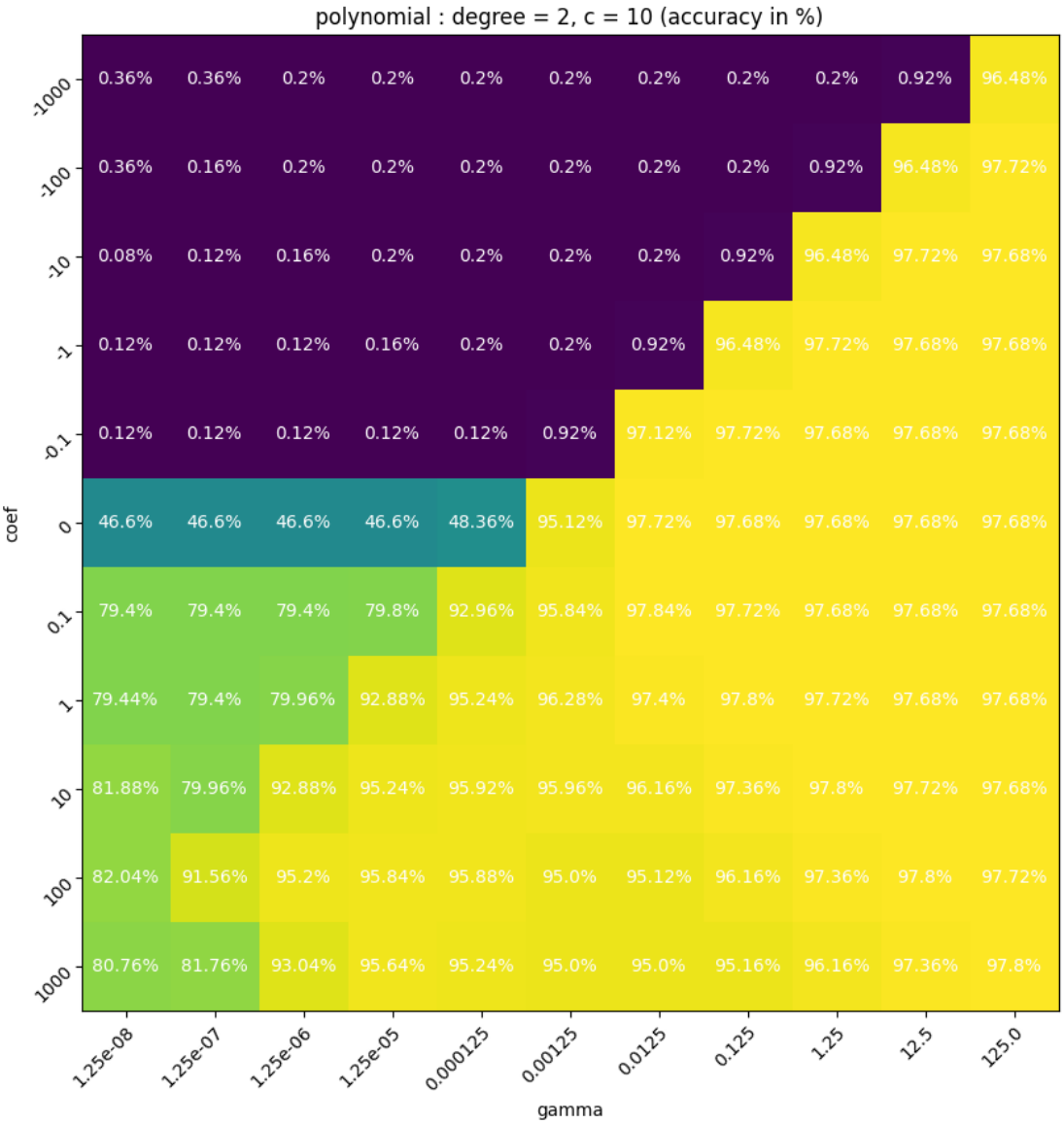

Fig 11. the testing accuracy of polynomial kernel with `degree` = 2 and `c`=10

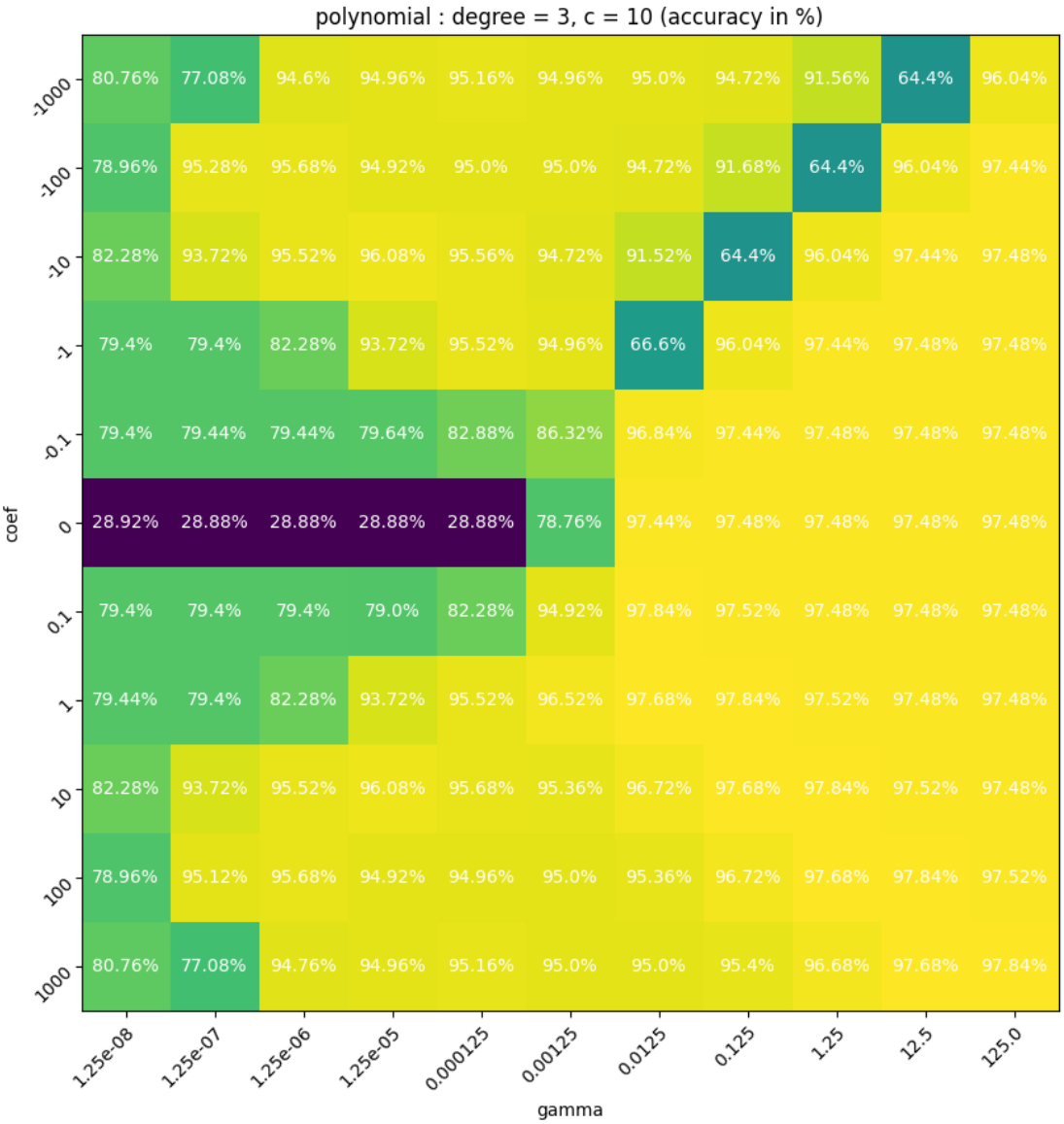

Fig 12. the testing accuracy of polynomial kernel with `degree` = 3 and `c`=10

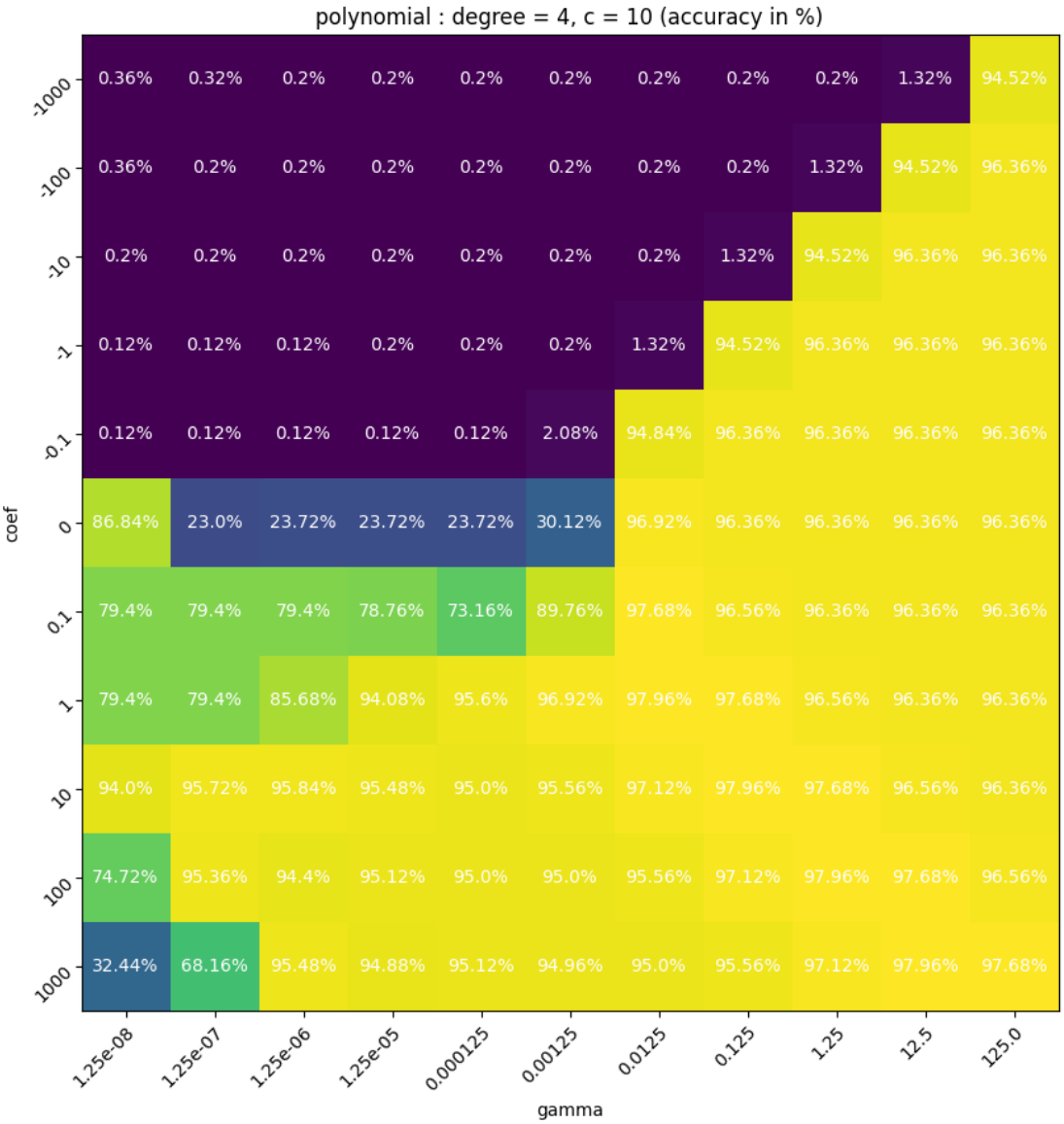

Fig 13. the testing accuracy of polynomial kernel with `degree` = 4 and `c`=10

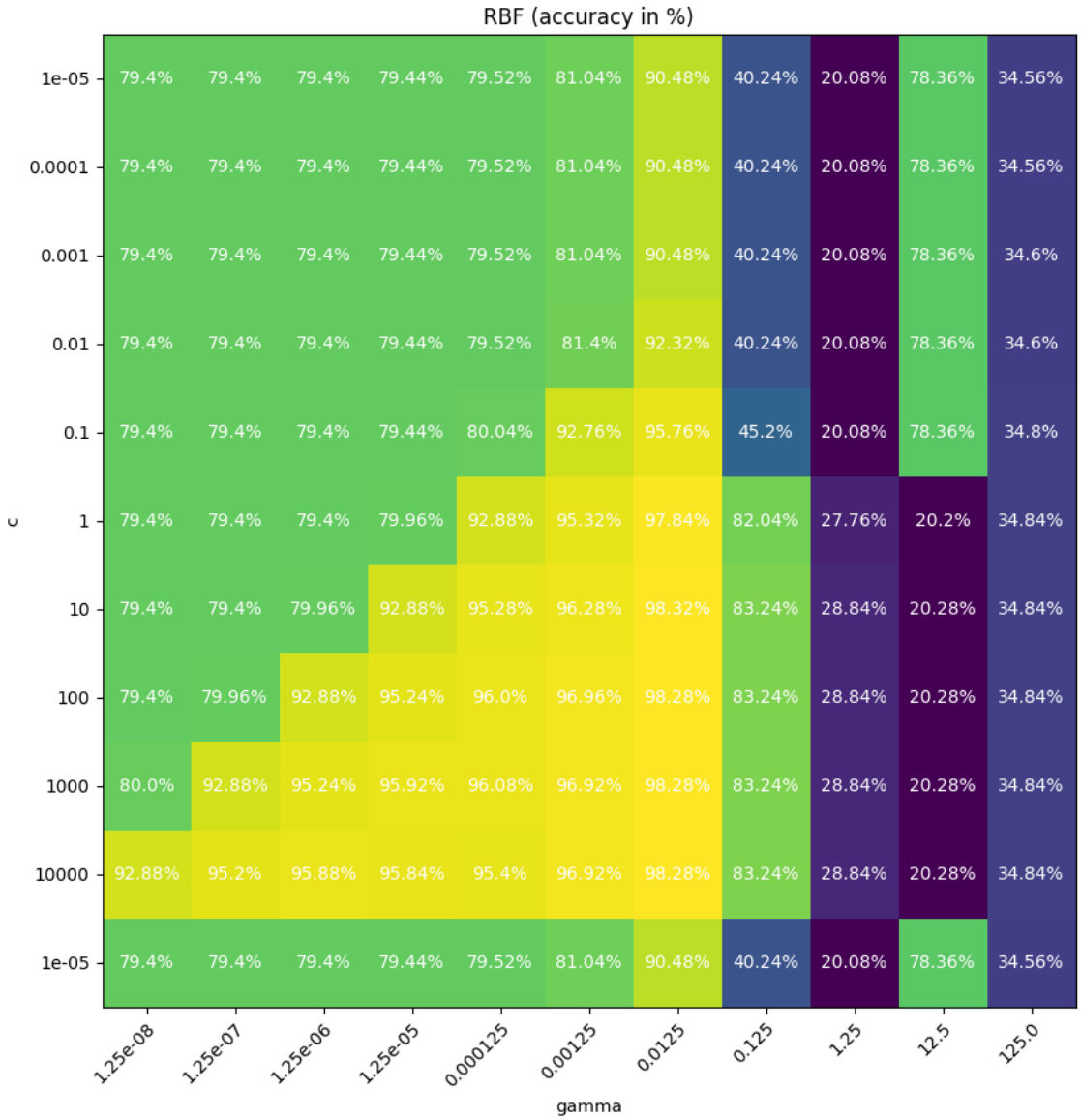

Fig 14. the testing accuracy of polynomial kernel with degree set to 4

Task 2.3

In this section, linear + RBF kernel was used as customized kernel. As described in **Code** section, I compared the testing accuracy of linear, RBF, linear + RBF respectively with same `c` parameter (set to default value 1) and 11 different `gamma` values. The testing accuracy result was shown in **Fig 8.**

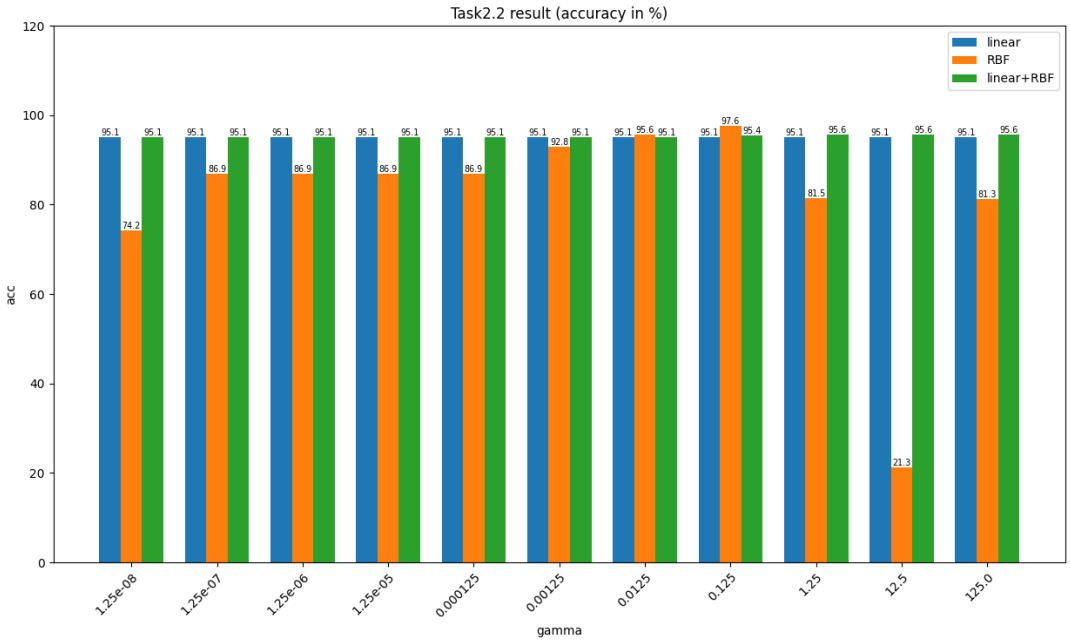

Fig 15. the testing accuracy of linear, RBF, and linear+RBF with 11 different `gamma` values

Observations and Discussion

Task 2.1

Some properties could be observed form the experimental results (**Fig 4**):

- linear kernel usually performed better that other two kernels when `c` parameter was smaller than 1, while RBF kernel outperformed linear kernel when `c` parameter was greater than than 1.
- polynomial kernel was quite sensitive to the `c` parameter. It performed really bad when `c` paramter smaller than 10 or larger than 10000,. On the contrary, it performed the best when `c` was 1000 or 10000.
- linear kernel was relatively stable with different `c` parameters than the other two kernels
- the best testing accuracy (97.6%) was the process that applied linear kernel with `c` set to 1000

Task 2.2

I ran SVM process for about 500 times in order to do the grid research, and the testing accuracy result could be seen in **Fig 5-13**. From the result we could know that:

Polynomial kernel

- the processes that applied `degree=3` performed averagely better than other values (2 and 4)
- larger `coef` and `gamma` tended to perform better on testig accuracy
- the accuracy map existed two quite different groups when the `degree` was 2 and 4 (top-left, bottom-right) that the boundary was a oblique line from bottom-right to top-left part of the figure, the former performed much worse than the latter.
- when `degree` was set to 3 and `coef` was set to 0, the testing accuracy will perform much worse than other parameter pairs when `gamma` less than 0.00125
- the best tesing accuracy (98.0%) was the process that took (`c`, `degree`, `gamma`, `coef0`) = (1, 4, 0.0125, 1)

RBF kernel

- the testing accuracy would fluctuate when `gamma` set greater than 0.0125 with different `c` applied
- the testing accuracy was averagely the best when `gamma` set to 0.0125
- the best accuracy (98.32%) was the process that took (`gamma`, `c`) = (0.0125, 10)

Task 2.3

In **Code** section, the experiment setup had been described. In this section I wanted to survey that when we kept the performance of linear kernel fixed (because `gamma` was not the parameter of it), how would different `gamma` values influence the performance of linear + RBF kerne. From the testing results shown in **Fig 14**, we could see that

- linear + RBF kernel tended to perform the same or slightly better than linear kernel
- RBF kernel was sensitive to the parameter `gamma`, it performed the best than other kernels when it was set to `0.125` (97.6%) while it performed the worst when it was set to `12.5` (21.3%) on testing accuracy.

From the result we could know that even when the RBF kernel performed worse than linear kernel in many cases, linear + RBF kernel could perform stably around 95% and the testing accuracy was always better than or equal to linear kernel.

References

- **[1]** scikit-learn online documentation for SVM parameters : https://scikit-learn.org/stable/auto_examples/svm/plot_rbf_parameters.html