ML HW5 - GP & SVM

多工所 王彥儒 0856621

Gaussian Process

Implementation Procedure & Discuss

1-1 Apply Gaussian Process Regression

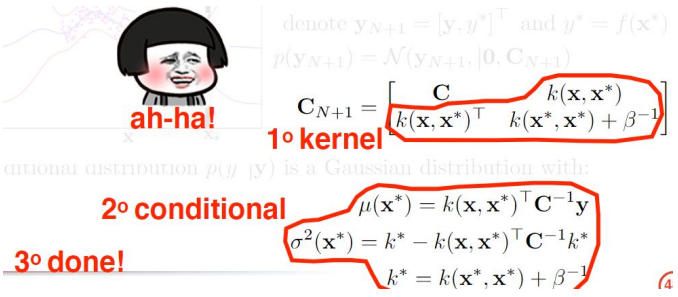
According to the teaching material, the first thing to do is to find to kernel function.
The rational quadratic kernel is as following:

1. Kernel Function

$$\widetilde{K}(x_1 - x_2) = \sigma^2\left(1 + \frac{(x_1-x_2)^2}{2\alpha l^2}\right)^{-\alpha}$$

```python
def Rational_Quardratic_Kernel(x1, x2, sigma=1, alpha=1, length_scale=1):
    """
    # Input
        x1: data feature 1 (n,1)
        x2: data feature 2 (1,n)
        sigma: Kernel vertical variation
        alpha: Control length
        length_scale: Kernel length, larger smoother

    # Return
        kernel: data feature combination (n,n)
    """
    x1 = x1.reshape(-1, 1)
    x2 = x2.reshape(1, -1)
    d = (x1 - x2)**2 / (2*alpha*(length_scale**2))
    kernel = np.power((1 + d), -alpha) * sigma**2

    return kernel
```

The rational quadratic kernel has 3 hyper-parameters that we can tune, sigma, alpha and length scale. In the initial state, I used all hyper-parameters with value 1.



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

2. Conditional

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

$$\mu\left(\mathbf{x}^*\right) = k\left(\mathbf{x}, \mathbf{x}^*\right)^{\mathrm{T}}\mathbf{C}^{-1}\mathbf{y}$$

$$\sigma^2\left(\mathbf{x}^*\right) = k^* - k\left(\mathbf{x}, \mathbf{x}^*\right)^{\mathrm{T}}\mathbf{C}^{-1}k\left(\mathbf{x}, \mathbf{x}^*\right)$$

$$k^* = k\left(\mathbf{x}^*, \mathbf{x}^*\right) + \beta^{-1}$$

```
def Gaussian_Process(x, y, s, a, l):
    """
    # Input
        x: data feature
        y: label
        s: sigma (Kernel vertical variation)
        a: alpha (Control length)
        l: length_scale (Kernel length, larger smoother)
    """
    beta = 5 # Noise, lower more noise, avoid overfitting
    k = Rational_Quardratic_Kernel(x, x, sigma=s, alpha=a, length_scale=l) # (34,34)
    C = k + (1/beta)*np.identity(len(k)) # (34,34)
    line = np.linspace(-60, 60, 100) # (100,)

    kx_kxstar = Rational_Quardratic_Kernel(x, line, sigma=s, alpha=a, length_scale=l) # (34, 100)
    kxstar_kxstar = Rational_Quardratic_Kernel(line, line, sigma=s, alpha=a, length_scale=l) # (100, 100)

    mu = kx_kxstar.T @ np.linalg.inv(C) @ y # (100,34) @ (34,34) @ (34,) = (100,)

    kstar = kxstar_kxstar + (1/beta) # (100,100)
    var = kstar - kx_kxstar.T @ np.linalg.inv(C) @ kx_kxstar # (100,34) @ (34,34) @ (34,100) = (100,100)
    var = np.sqrt(np.diag(var))

    # Visualization
    hyperparameter = f'sigma = {s:.3f}\nalpha= {a:.3f}\nlength_scale= {l:.3f}'
    plt.style.use('bmh')
    plt.plot(x, y, 'ro')
    plt.plot(line, mu, 'b-', label=hyperparameter)

    plt.fill_between(line, mu+2*var, mu-2*var, alpha=0.5)
    plt.xlim([-60, 60])
    plt.legend(loc=8)
    plt.show()

init_sigma = init_alpha = init_length_scale = 1
plt.title('Initial')
Gaussian_Process(x, y, init_sigma, init_alpha, init_length_scale)
```

Second step, with conditional parameter, we can easily calculate the mean and variance on each points. Beta is used as a noise parameter to prevent model from overfitting, the lower it is, the large noise you will get.

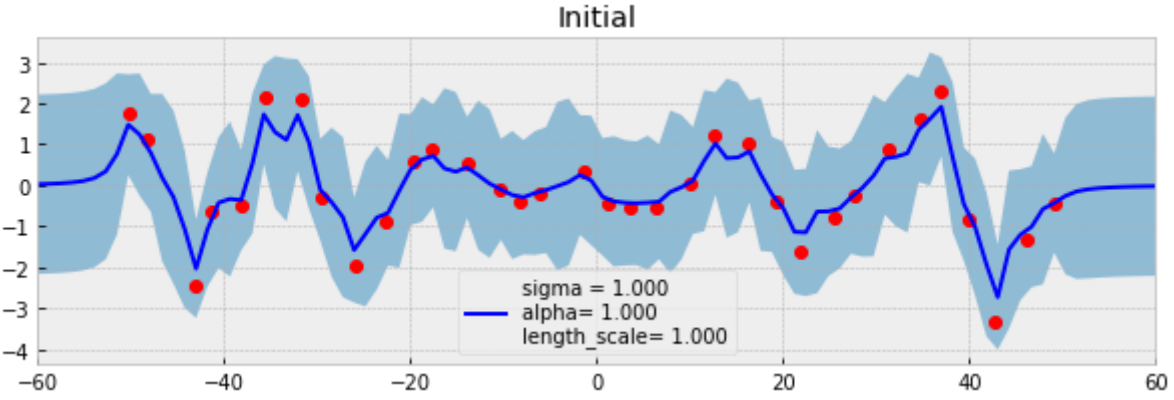
The capital C (covariance matrix is kernel of data, and the delta is the identity matrix).

x is the training data features and x* (x_star) is the new data features, which in our homework is the line x coordinate.

The new data variance is the diagonal of the matrix manuplication. In order to get the 95% confidence interval, we have to do the square-root to get the standard deviation, and finally the mean plus(minus) two standard deviation is the 95% confidence boundary.

There's some mistake in the teaching material on page 48. In the second line of variance formula, the k should be k(x,x*).

The Initial state result:



In the second mission, optimize the kernel parameters. According to the teaching material, the log likelihood is as following. Our goal is to maximize the log likelihood. Therefore, we can derive the formula into finding the minimum value, and using *scipy.optimize.minimize* to optimize the parameters.

$$p(\mathbf{y}|\theta) = \mathcal{N}(\mathbf{y}|\mathbf{0}, \mathbf{C}_\theta)$$

$$\ln p(\mathbf{y}|\theta) = -\frac{1}{2}\ln |\mathbf{C}_\theta| - \frac{1}{2}\mathbf{y}^\top \mathbf{C}_\theta^{-1}\mathbf{y} - \frac{N}{2}\ln (2\pi) \text{ ☞ } \frac{\partial \ln p(\mathbf{y}|\theta)}{\partial \theta}$$

By the way, **the symbol like absolute value is actually the determinant of** C_θ

1-2 Optimize the kernel parameters

3. Optimize the Kernel Parameters

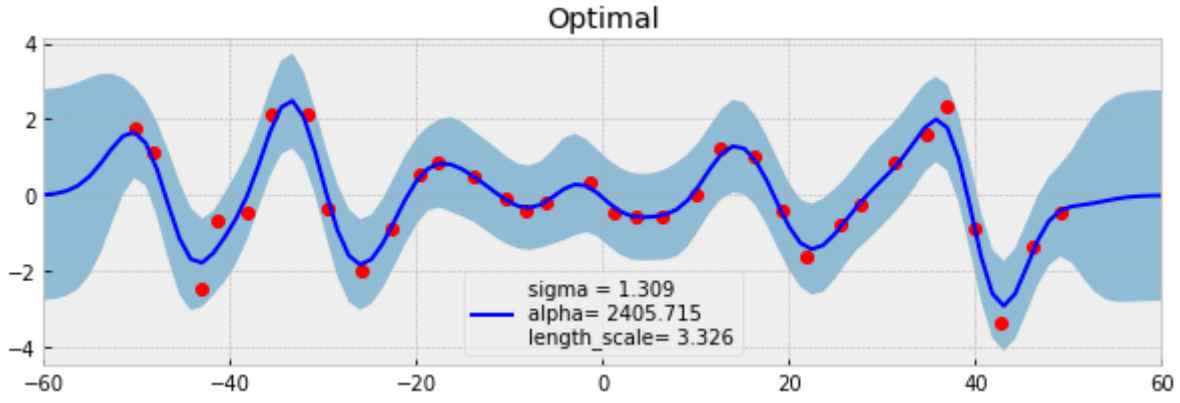
$$\text{argmax}(\ln p(y|\theta)) = -\frac{1}{2}\ln|C_\theta| - \frac{1}{2}y^T C_\theta^{-1}\mathbf{y} - \frac{N}{2}\ln(2\pi) \propto -\ln|C_\theta| - y^T C_\theta^{-1} y$$

$$= \text{argmin}\left(\ln|C_\theta| + y^T C_\theta^{-1} y\right)$$

```
from scipy.optimize import minimize

def objective_function(x, y, beta):
    """
    # Input
        x: data feature
        y: label
        beta: noise, lower more noise, avoid overfitting
    """
    def objective(theta):
        """
        # Input
            theta: hyperparameters
        """
        k = Rational_Quardratic_Kernel(x, x, sigma=theta[0], alpha=theta[1], length_scale=theta[2])
        C = k + (1/beta)*np.identity(len(k))
        return np.log(np.linalg.det(C)) + y.T @ np.linalg.inv(C) @ y

    return objective

beta = 5

objective_value = np.inf
inits = [1e-2, 1e-1, 0, 1e1, 1e2]

for init_sigma in inits:
    for init_alpha in inits:
        for init_length_scale in inits:
            res = minimize(objective_function(x, y, beta), x0=[init_sigma, init_alpha, init_length_scale], bounds=((1e-5,None),
            if res.fun < objective_value:
                objective_value = res.fun
                sigma_opt, alpha_opt, length_scale_opt = res.x

print('sigma: ', sigma_opt)
print('alpha: ',alpha_opt)
print('length_scale: ',length_scale_opt)
```

```
sigma:  1.3086673427082567
alpha:  2405.7146900275693
length_scale:  3.3260099629544553
```

In the grid search of Gaussian Process hyper-parameters (sigma, alpha, length scale), I used 1e-5 as the lowest bound value. After minmizing the loss, we got the best hyper-parameter to fit our model.

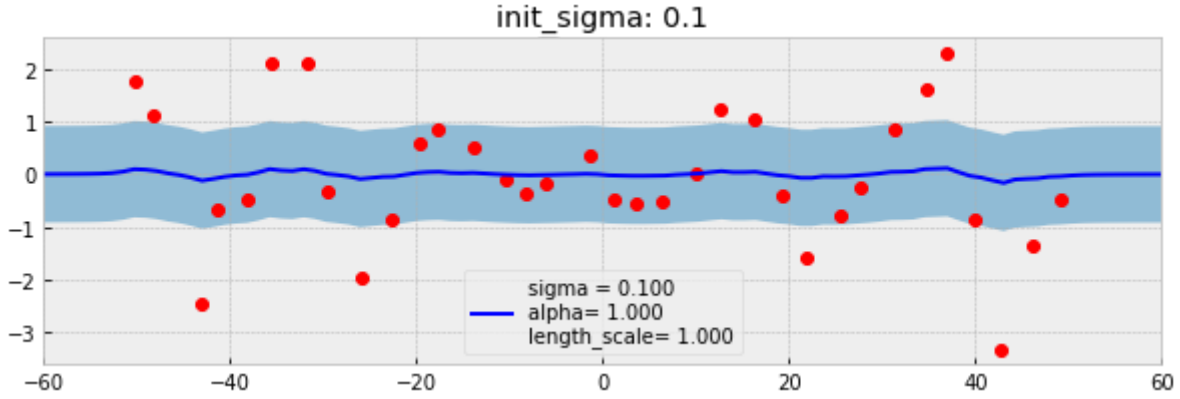


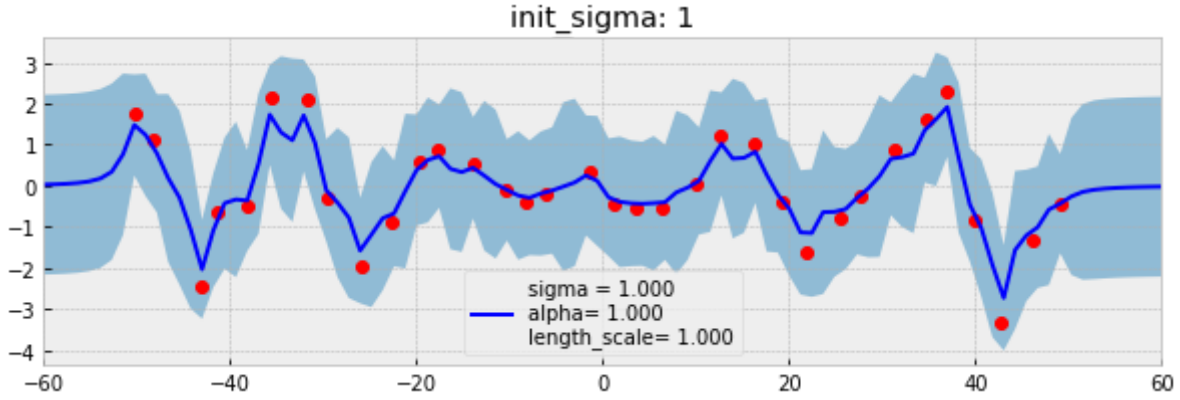
Experiment & Result

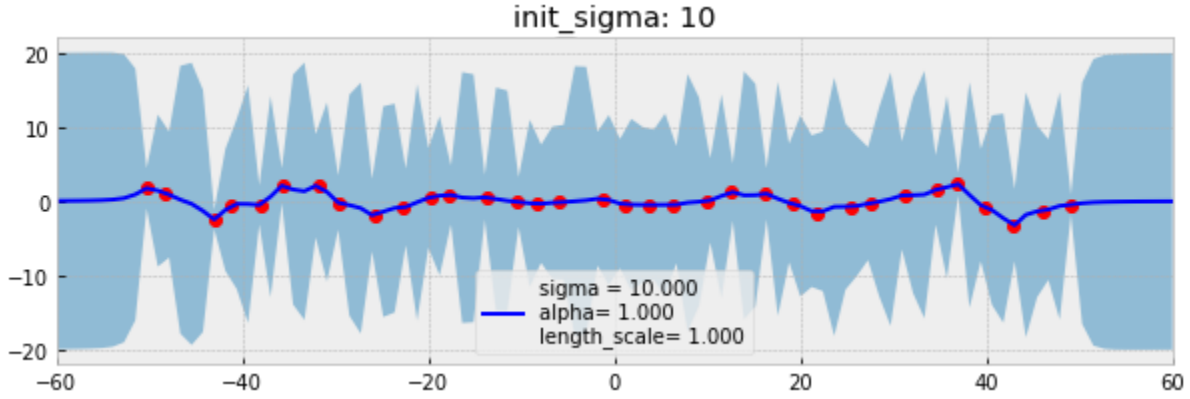
In the rational quadratic kernel, we have three hyper-parameters. It's quite intersting to find what exactly their functions are. Thus, I do some experiment on them.

$$\widetilde{K}(x_1 - x_2) = \sigma^2\left(1 + \frac{(x_1-x_2)^2}{2\alpha l^2}\right)^{-\alpha}$$

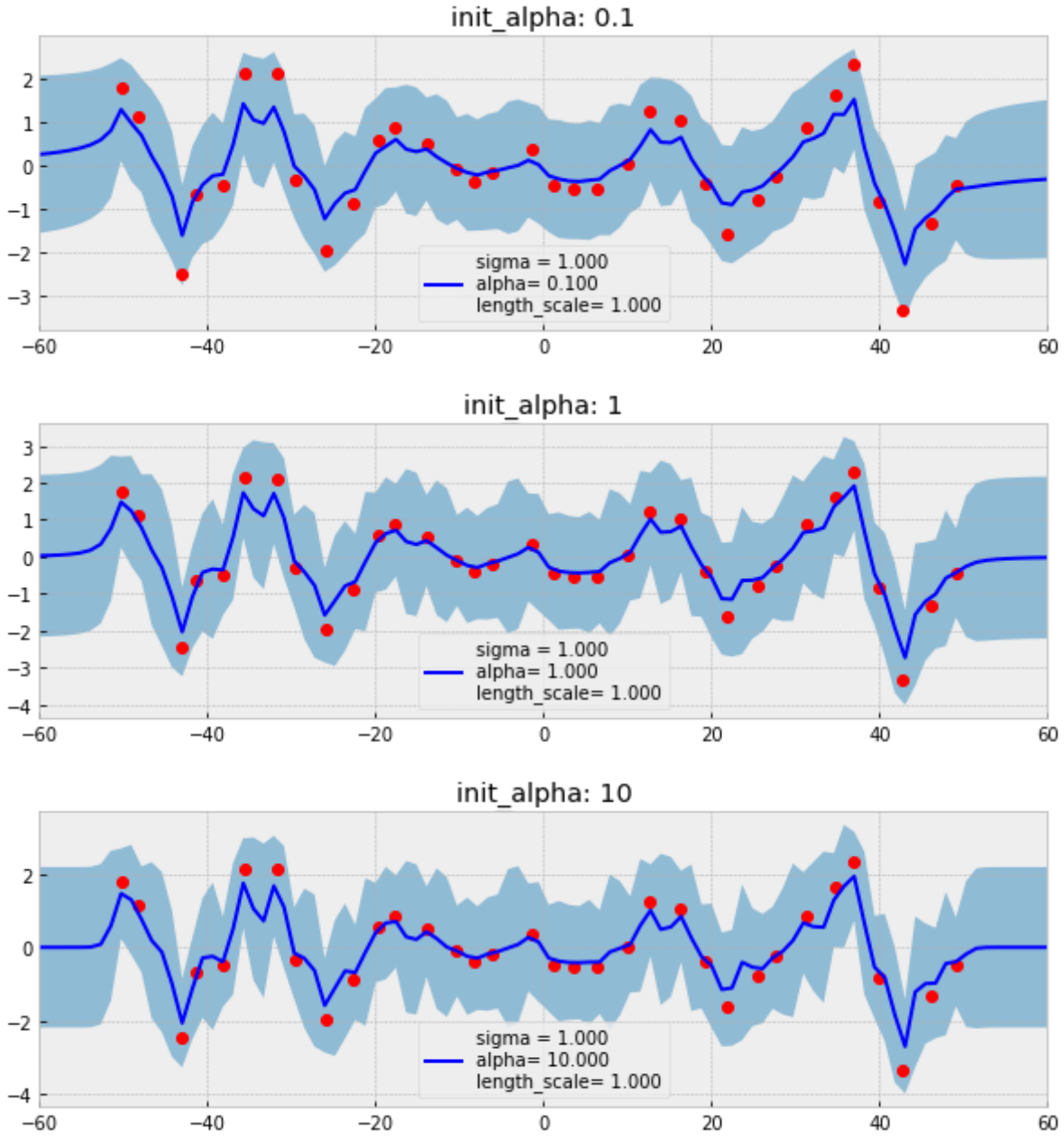
The first experiment is on parameter sigma. In our experiment results, sigma controls the fitting line variance, which is as its formula as a standard deviation.



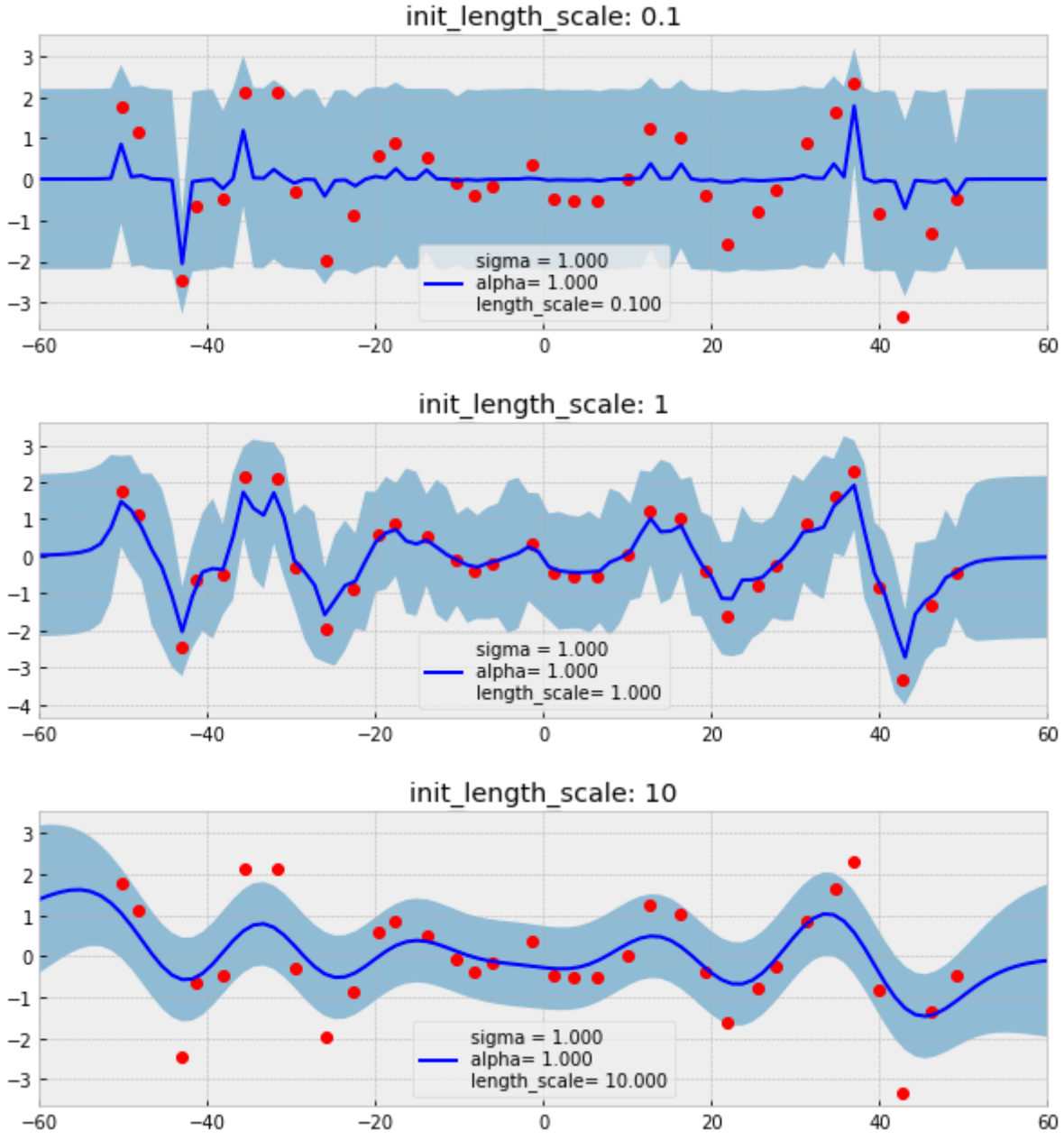




The second experiment is on the parameter alpha. The experiment result is quite the same. However with small alpha, the fitting line and its variance are smoother than the large one.



The last experiment is on the parameter length scale. It's seems like the higher length scale value lead to smoother functions and therefore coarser approximations of the training data. Lower length scale values make functions more wiggly with wide confidence intervals between training data points.



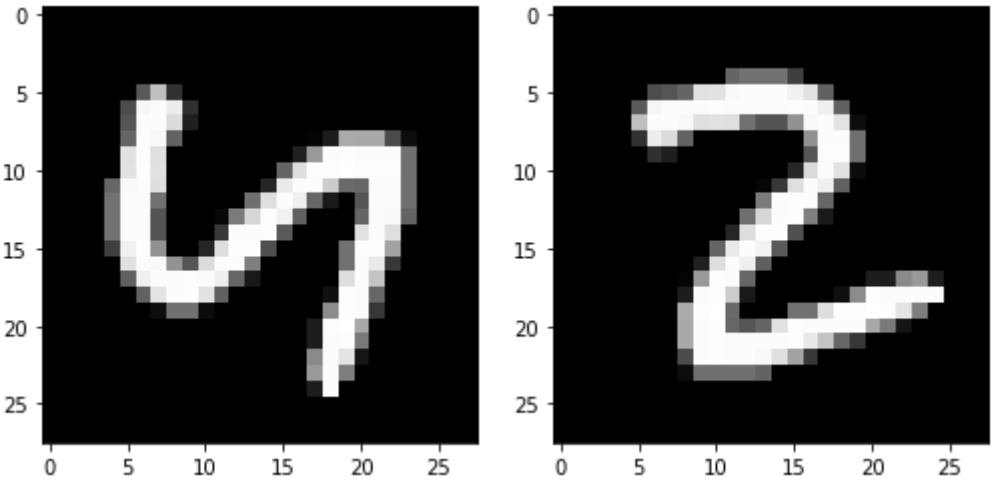
SVM on MNIST dataset

Implementation Procedure & Discuss

2-1 Use different kernel functions

```
import numpy as np
import csv

def CSV_image_load(path):
    image_list = []
    with open(path) as csvfile:
        rows = csv.reader(csvfile)
        for row in rows:
            i = 0
            j = 0
            image = np.zeros((28,28))
            for col in row:
                image[i,j] = col
                i += 1
                if i == 28:
                    i = 0
                    j += 1
            image_list.append(image)
    return np.asarray(image_list, dtype='float').reshape(-1, 784)

def CSV_label_load(path):
    label_list = []
    with open(path) as csvfile:
        rows = csv.reader(csvfile)
        for row in rows:
            label_list.append(row)

        return np.asarray(label_list, dtype='float').reshape(-1,)

x_train = CSV_image_load('X_train.csv') # (n, 784)
x_test = CSV_image_load('X_test.csv') # (n,)
y_train = CSV_label_load('Y_train.csv') # (n, 784)
y_test = CSV_label_load('Y_test.csv') # (n,)
```

Intuitively, loading the image from csv file is quite easy to us. Yet, the pixel value order is not as we imagine. **When we load the image pixel value, it's seems like they're Fortran-like index order (column major).** Hence, **if we reshape the shape (784,) into (28,28), the image will become transpose.** Furthermore, this issue will **influence the SVM classification score when applying Polynomial kernel function (-0.44%).**



```
Accuracy = 95.08% (2377/2500) (classification)  Accuracy = 95.08% (2377/2500) (classification)
Accuracy = 34.68% (867/2500) (classification)   Accuracy = 35.12% (878/2500) (classification)
Accuracy = 95.32% (2383/2500) (classification)  Accuracy = 95.32% (2383/2500) (classification)
```

-t kernel_type : set type of kernel function (default 2) ¶

- 0 -- linear: u'*v
- 1 -- polynomial: (gamma*u'*v + coef0)^degree
- 2 -- radial basis function: exp(-gamma*|u-v|^2)
- 3 -- sigmoid: tanh(gamma*u'*v + coef0)

```
from libsvm.svmutil import *

kernel_types = {'linear':'-t 0', 'polynomial':'-t 1', 'radial basis function':'-t 2'}
accuracy = []
for k,param in kernel_types.items():
    model = svm_train(y_train, x_train, param)
    p_labs, p_acc, p_vals = svm_predict(y_test, x_test, model)
    accuracy.append(p_acc[0])
```

```
Accuracy = 95.08% (2377/2500) (classification)
Accuracy = 35.12% (878/2500) (classification)
Accuracy = 95.32% (2383/2500) (classification)
```

```
i = 0
for k, v in kernel_types.items():
    print(f'{k} kernel accuracy: {accuracy[i]:.2f}%')
    i += 1
```

```
linear kernel accuracy: 95.08
polynomial kernel accuracy: 35.12
radial basis function kernel accuracy: 95.32
```

The first work is trying to use SVM with different kernel on MNIST dataset. All the parameters are default setting. The Radial basis function kernel got the best score in the testing dataset, with 95.32% accuracy. The polynomial kernel got the worst score. The reason may result from its hyper-parameters (cost, gamma, coefficient and kernel degree)

2-2 C-SVC Grid Search

-s svm_type : set type of SVM (default 0)

- 0 -- C-SVC
- 1 -- nu-SVC
- 2 -- one-class SVM
- 3 -- epsilon-SVR
- 4 -- nu-SVR

- -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/k)
- -r coef0 : set coef0 in kernel function (default 0)
- -d degree : set degree in kernel function (default 3)

Linear kernel Grid Search

```
def grid_search_linear(log2c, x_train, y_train, x_test, y_test):
    best_log2c = best_acc = 0
    confusion_matrix = np.zeros((len(log2c)))

    for i in range(len(log2c)):
        param = f'-q -t 0 -v 5 -c {2**log2c[i]}'
        acc = svm_train(y_train, x_train, param)
        print(f'[{i+1} / {len(log2c)}] Cost: {2**log2c[i]}, Accuracy: {acc:.2f}%\n')
        confusion_matrix[i] = acc
        if acc > best_acc:
            best_acc = acc
            best_log2c = log2c[i]

    print(f'Best setting: Cost: {2**best_log2c}, Accuracy: {best_acc:.3f}%')

    return confusion_matrix, best_log2c

log2c = [-7, -6, -5, -4, -3, -2, -1, 0, 1, 2]
confusion_matrix, best_log2c = grid_search_linear(log2c, x_train, y_train, x_test, y_test)
```

The second work is using grid search to find the best hyper-parameters to fit the chosen kernel soft-margin SVM (C-SVC). I tried to use 10 values applied on the cost of linear kernel function. The parameter -q: quiet mode, -t: kernel type, -v: k-fold cross validation.
We got the best accuracy 97.04% with 0.03125 cost on linear kernel. However, there's only 96% accuracy in the testing set.

```
Cross Validation Accuracy = 96.98%
[1 / 10] Cost: 0.0078125, Accuracy: 96.98%

Cross Validation Accuracy = 96.98%
[2 / 10] Cost: 0.015625, Accuracy: 96.98%

Cross Validation Accuracy = 97.04%
[3 / 10] Cost: 0.03125, Accuracy: 97.04%

Cross Validation Accuracy = 97%
[4 / 10] Cost: 0.0625, Accuracy: 97.00%

Cross Validation Accuracy = 96.96%
[5 / 10] Cost: 0.125, Accuracy: 96.96%

Cross Validation Accuracy = 96.72%
[6 / 10] Cost: 0.25, Accuracy: 96.72%

Cross Validation Accuracy = 96.42%
[7 / 10] Cost: 0.5, Accuracy: 96.42%

Cross Validation Accuracy = 96.52%
[8 / 10] Cost: 1, Accuracy: 96.52%

Cross Validation Accuracy = 96.18%
[9 / 10] Cost: 2, Accuracy: 96.18%

Cross Validation Accuracy = 96.38%
[10 / 10] Cost: 4, Accuracy: 96.38%

Best setting: Cost: 0.03125, Accuracy: 97.040%
```

```
Accuracy = 96% (2400/2500) (classification)
Optimal Parameters Linear kernel, Cost: 0.03125, Accuracy: 96.00%
```

Polynomial kernel Grid Search

```
def grid_search_poly(log2c, log2g, log2r, x_train, y_train, x_test, y_test):
    best_log2c = best_log2g = best_log2r = best_acc = 0
    confusion_matrix = np.zeros((len(log2c), len(log2g), len(log2r)))

    for i in range(len(log2c)):
        for j in range(len(log2g)):
            for k in range(len(log2r)):
                param = f'-q -t 1 -v 5 -c {2**log2c[i]} -g {2**log2g[j]} -r {2**log2r[k]} -d 2'
                acc = svm_train(y_train, x_train, param)
                print(f'[{k+j*len(log2g)+i*len(log2c)*len(log2g)+1} / {len(log2c)*len(log2g)*len(log2r)}] Cost: {2**log2c[i]}, Ga
                confusion_matrix[i,j] = acc
                if acc > best_acc:
                    best_acc = acc
                    best_log2c = log2c[i]
                    best_log2g = log2g[j]
                    best_log2r = log2r[k]

    print(f'Best setting: Cost: {2**best_log2c}, Gamma: {2**best_log2g}, coef0 : {2**log2r[k]}, Accuracy: {best_acc:.3f}%')

    return confusion_matrix, best_log2c, best_log2g, best_log2r

log2c = [-4, -3, -2]
log2g = [12, 13, 14]
log2r = [4, 5, 6]
confusion_matrix, best_log2c, best_log2g, best_log2r = grid_search_poly(log2c, log2g, log2r, x_train, y_train, x_test, y_test)
```

```
Cross Validation Accuracy = 98.14%
[1 / 27] Cost: 0.0625, Gamma: 4096, coef0 : 16, Accuracy: 98.14%

Cross Validation Accuracy = 97.94%
[2 / 27] Cost: 0.0625, Gamma: 4096, coef0 : 32, Accuracy: 97.94%

Cross Validation Accuracy = 98.16%
[3 / 27] Cost: 0.0625, Gamma: 4096, coef0 : 64, Accuracy: 98.16%

Cross Validation Accuracy = 97.98%
[4 / 27] Cost: 0.0625, Gamma: 8192, coef0 : 16, Accuracy: 97.98%

Cross Validation Accuracy = 97.98%
[5 / 27] Cost: 0.0625, Gamma: 8192, coef0 : 32, Accuracy: 97.98%
```

In the polynomial kernel, I used 3 different cost value ($2^{-4} \sim 2^{-3}$), 3 gamma value ($2^{12} \sim 2^{14}$) ,3 coefficient value ($2^{4} \sim 2^{6}$) and fix the degree of kernel to 2. It's totally 27 combinations.

```
Cross Validation Accuracy = 98.14%
[27 / 27] Cost: 0.25, Gamma: 16384, coef0 : 64, Accuracy: 98.14%

Best setting: Cost: 0.25, Gamma: 4096, coef0 : 64, Accuracy: 98.320%
```

Polynomial kernel Prediction on Testing Set

```
param = f'-q -t 1 -c {2**best_log2c} -g {2**best_log2g} -r {2**best_log2r} -d 2'
model = svm_train(y_train, x_train, param)

p_label, p_acc, p_vals = svm_predict(y_test, x_test, model)
print(f'Optimal Parameters Polynomial kernel, Cost: {2**best_log2c}, Gamma: {2**best_log2g}, coef0 : {2**best_log2r}, degree : {2
```

```
Accuracy = 97.68% (2442/2500) (classification)
Optimal Parameters Polynomial kernel, Cost: 0.125, Gamma: 4096, coef0 : 32, degree : 2, Accuracy: 97.68%
```

The best result is in Polynomial kernel is cost value: 0.125, gamma: 4096, coefficient: 32 , degree: 2, Accuracy: 98.32%.

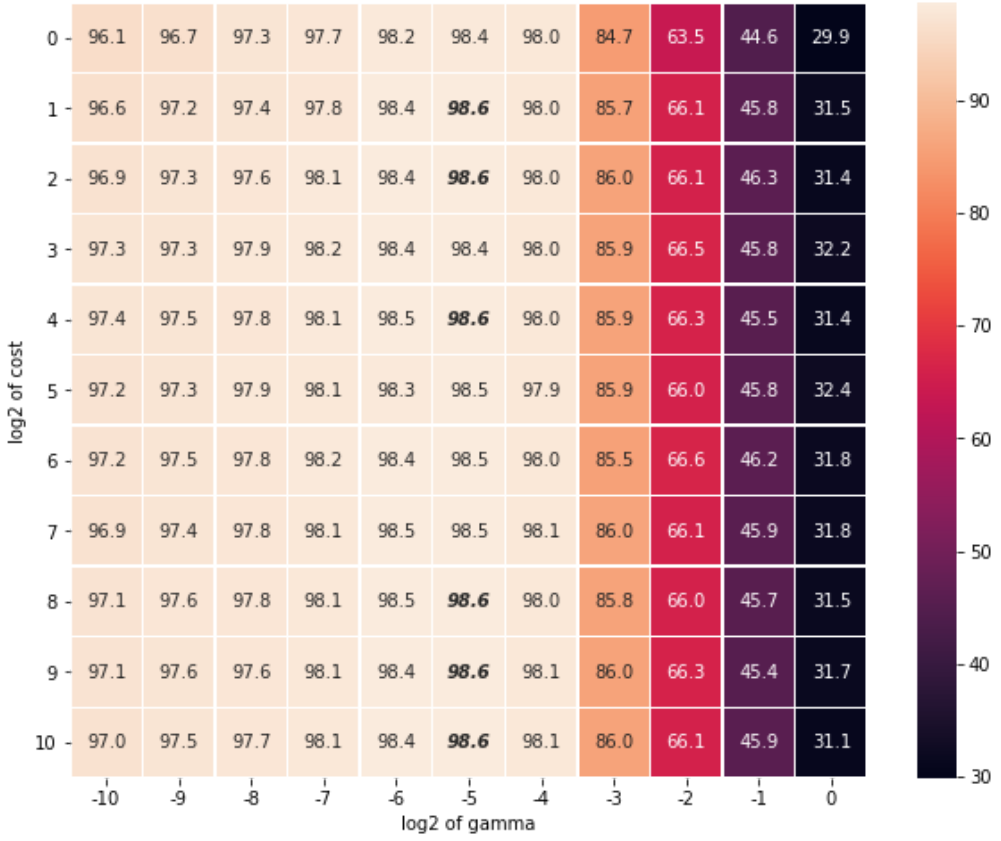
And the best hyper-parameters model got 97.68% accuracy in the testing dataset.

RBF kernel Grid Search

```
def grid_search(log2c, log2g, x_train, y_train, x_test, y_test):
    best_log2c = best_log2g = best_acc = 0
    confusion_matrix = np.zeros((len(log2c), len(log2g)))

    for i in range(len(log2c)):
        for j in range(len(log2g)):
            param = f'-q -t 2 -v 5 -c {2**log2c[i]} -g {2**log2g[j]}'
            acc = svm_train(y_train, x_train, param)
            print(f'[{j+i*len(log2g)+1} / {len(log2c)*len(log2g)}] Cost: {2**log2c[i]}, Gamma: {2**log2g[j]}, Accuracy: {acc:.2f}
            confusion_matrix[i,j] = acc
            if acc > best_acc:
                best_acc = acc
                best_log2c = log2c[i]
                best_log2g = log2g[j]

    print(f'Best setting: Cost: {2**best_log2c}, Gamma: {2**best_log2g}, Accuracy: {best_acc:.3f}%')

    return confusion_matrix, best_log2c, best_log2g

log2c = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
log2g = [-10, -9, -8, -7, -6, -5, -4, -3, -2, -1, 0]
confusion_matrix, best_log2c, best_log2g = grid_search(log2c, log2g, x_train, y_train, x_test, y_test)
```

```
Cross Validation Accuracy = 98.6%
[116 / 121] Cost: 1024, Gamma: 0.03125, Accuracy: 98.60%

Cross Validation Accuracy = 98.06%
[117 / 121] Cost: 1024, Gamma: 0.0625, Accuracy: 98.06%

Cross Validation Accuracy = 85.96%
[118 / 121] Cost: 1024, Gamma: 0.125, Accuracy: 85.96%

Cross Validation Accuracy = 66.08%
[119 / 121] Cost: 1024, Gamma: 0.25, Accuracy: 66.08%

Cross Validation Accuracy = 45.92%
[120 / 121] Cost: 1024, Gamma: 0.5, Accuracy: 45.92%
```

In the radial basis function kernel, I used 11 different cost value ($2^0 \sim 2^{10}$) and 11 different gamma value ($2^{-10} \sim 2^0$). It's totally 121 combinations.



The best result is in RBF kernel is cost value: 512, gamma: 0.03125, Accuracy: 98.64%

RBF kernel Prediction on Testing Set

```
param = f'-q -t 2 -c {2**best_log2c} -g {2**best_log2g}'
model = svm_train(y_train, x_train, param)

p_label, p_acc, p_vals = svm_predict(y_test, x_test, mod
print('Optimal Parameters RBF kernel accuracy: {:.2f}%'.format(p_acc[0]))
```

```
Accuracy = 98.52% (2463/2500) (classification)
Optimal Parameters RBF kernel accuracy: 98.52%
```

And the best hyper-parameters model got 98.52% accuracy in the testing dataset.

2-3 Linear kernel + RBF kernel

The final work is to use linear kernel and RBF kernel together to create a brand new kernel function.

Linear kernel: $K(\mathbf{x}, \mathbf{x}') = \mathbf{x}^T\mathbf{x}', \quad \mathbf{x}, \mathbf{x}' \in \mathbb{R}^d$ ¶

Radial basis function kernel: $K\left(\mathbf{x}, \mathbf{x}'\right) = \exp\left(-\frac{\|\mathbf{x}-\mathbf{x}'\|^2}{2\sigma^2}\right)$

$K\left(\mathbf{x}, \mathbf{x}'\right) = \exp\left(-\gamma\|\mathbf{x} - \mathbf{x}'\|^2\right)$

What is the fastest way to compute an RBF kernel in python?

New training instance for xi:

0:i 1:K(xi,x1) ... L:K(xi,xL)

New testing instance for any x:

0:? 1:K(x,x1) ... L:K(x,xL)

That is, in the training file the first column must be the "ID" of xi. In testing, ? can be any value.

```
def linear_RBF_kernel(x, x_prime, gamma):
    linear_kernel = x @ x_prime.T
    x_norm = np.sum(x**2, axis = -1)
    x_prime_norm = np.sum(x_prime**2, axis = -1)

    RBF = 1 * np.exp(-gamma * (x_norm[:,None] + x_prime_norm[None,:] - 2 * linear_kernel))
    kernel = RBF + linear_kernel
    kernel = np.hstack((np.arange(1, len(x)+1).reshape(-1,1), kernel)) # Add ID
    return kernel
```

1. The first issue is about the **RBF formula**. There're lots of different kind implementation (pure numpy, numexpr, scipy.spatial.distance.pdist, ~~sklearn.metrics.pairwise.rbf_kernel)~~

Method	Time
Numpy	24.2 s ± 1.06 s
Numexpr	8.89 s ± 314ms
scipy.spatial.distance.pdist	2 min 59s ± 312 ms
sklearn.metrics.pairwise.rbf_kernel	13.9 s ± 757 ms

Although numexpr is the fast method of all, the numpy as quite straightforward and easy to use. *scipy.spatial.distance.pdist* is much slower even with the small features space like our task.

By the way, the differences between *scipy.spatial.distance.pdist* & *scipy.spatial.distance.cdist* is their input. (cdist: 2 inputs, pdist: 1 inputs)

2. The second issue is about the **libsvm feature format**. When we just simply used RBF + linear kernel feature, the training accuracy would become 20% only. The reason is that the feature format should add an ID (index) at the first column. So, the original features shape would convert (5000,5000) to (5000,5001). After adding the index, the accuracy restored to 95%

3. The last issue is about the kernel, we should used the **training data kernel feature combining with the testing** instead of just using testing set.
   *linear_RBF_kernel(x_test, x_train, 2**best_log2c) => (2500, 5001)*
   *linear_RBF_kernel(x_test, x_test, 2**best_log2c) => (2500, 2501)*

   Moreover, the order of testing and training is important as well.
   *linear_RBF_kernel(x_test, x_train, 2**best_log2c) => (2500, 5001)*
   *linear_RBF_kernel(x_train, x_test, 2**best_log2c) => (5001, 2500)*

```
def linear_RBF_kernel(x, x_prime, gamma):
    linear_kernel = x @ x_prime.T
    x_norm = np.sum(x**2, axis = -1)
    x_prime_norm = np.sum(x_prime**2, axis = -1)

    RBF = 1 * np.exp(-gamma * (x_norm[:,None] + x_prime_norm[None,:] - 2 * linear_kernel))
    kernel = RBF + linear_kernel
    kernel = np.hstack((np.arange(1, len(x)+1).reshape(-1,1), kernel)) # Add ID
    return kernel
```

```
best_log2c = 7
best_log2g = -5

kernel_train = linear_RBF_kernel(x_train, x_train, 2**best_log2c) # (5000,5001)
kernel_test = linear_RBF_kernel(x_test, x_train, 2**best_log2c) # (2500, 5001)

prob = svm_problem(y_train, kernel_train, isKernel=True)
param = svm_parameter(f'-q -t 4 -c {2**best_log2g}')
model = svm_train(prob, param)


p_label, p_acc, p_vals = svm_predict(y_test, kernel_test, model, '-q')
print('linear kernel + RBF kernel accuracy: {:.2f}%'.format(p_acc[0]))
```

```
linear kernel + RBF kernel accuracy: 95.96%
```

Finally, we used the hyper-parameters from the grid search above. The testing accuracy is 95.96%

Kernel Type	Training Accuracy	Testing Accuracy
Linear	97.04 %	96%
Polynomial	98.32 %	97.68 %
Radial basis function	98.64 %	98.52 %
Linear + RBF		95.96 %

Additional references

gaussian-processes
LIBSVM -- A Library for Support Vector Machines
cjlin1/libsvm
关于svm_train的参数问题
What is the fastest way to compute an RBF kernel in python?
libsvm data format - Cross Validated