Report

I. Gaussian Process

a. Code Explanation

- Load Dataset

```
Untitled-1

1   def load_dataset(path):
2       X_train, y_train = [], []
3       with open(path, 'r', encoding='utf-8') as f:
4           for line in f:
5               x, y = line.split(' ')
6               X_train.append(float(x))
7               y_train.append(float(y))
8       X_train = np.array(X_train, dtype=np.float64).reshape(-1, 1)
9       y_train = np.array(y_train, dtype=np.float64).reshape(-1, 1)
10      return X_train, y_train
```

The load_dataset() function reads all lines of numbers in input.data file. The input.data file contains n numbers for X_train and y_train. Both X_train and y_train are converted into 2-dimension numpy array with shape n by 1.

- Part 1

In the part 1, I use gaussian process regression with rational quadratic kernel to predict the distribution of data point. The image below is mathematic formula of this kernel function,

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

with:

- σ2 the overall variance (σ is also known as amplitude).
- ℓ the lengthscale.
- α the scale-mixture (α > 0).
- X_a and X_b data points.

The code below is implementation of rational quadratic kernel.

```
Untitled-1

def rational_quadratic_kernel(X1, X2, sigma, alpha, length_scale):
    nume = np.power(X1, 2) + np.power(X2, 2).flatten() - 2 * np.matmul(X1, np.transpose(X2))
    return (sigma ** 2) * ((1 + nume / (2 * alpha * (length_scale ** 2))) ** (-1 * alpha))
```

With this basic kernel function, we can use gaussian process regression to predict distribution of data points. For the purpose of visualization, I generate 1000 numbers evenly distributed from -60.0 to 60.0.

```
Untitled-1

X_test = np.linspace(-60.0, 60.0, 1000).reshape(-1, 1)
```

I set the default parameters of gaussian process, and call gaussain_process() function.

```
Untitled-1

# default parameter
beta = 5
sigma = 1
alpha = 1
length_scale = 1

# apply gaussian process
gaussian_process(
    X_train,
    y_train,
    X_test,
    beta,
    sigma,
    alpha,
    length_scale
)
```

In gaussian_process() function, I will find the mean and variance of testing data points with these formula.

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

We can find two kinds of kernel in above formula, k(x, x*) and k(x*, x*). The former is the kernel of training and testing data points and the latter is the kernel of testing data points.

Additionally, what the C means is:

$$C(X_i, X_j) = k(x_i, x_j) + \beta^{-1}\delta_{ij}$$
$$\delta_{ij} = \begin{cases} 1, \ i = j, \\ 0, \ otherwise. \end{cases}$$

We can find another type of kernel, $k(x_i, x_j)$ in formula. It is a kernel of training data points.

In the code below, I create three kinds of kernels and C.

Untitled-1

```python
def gaussian_process(
        X_train,
        y_train,
        X_test,
        beta,
        sigma,
        alpha,
        length_scale
    ):

    kernel_Xtr_Xtr = rational_quadratic_kernel(X_train, X_train, sigma, alpha, length_scale)
    kernel_Xtr_Xte = rational_quadratic_kernel(X_train, X_test, sigma, alpha, length_scale)
    kernel_Xte_Xte = rational_quadratic_kernel(X_test, X_test, sigma, alpha, length_scale)
    C = kernel_Xtr_Xtr + (1 / beta) * np.identity(X_train.shape[0], dtype=np.float64)

    ...
```

With these kernels, I can compute the mean and variance of testing data point.

Untitled-1

```python
mean_Xte = np.matmul(np.matmul(np.transpose(kernel_Xtr_Xte), np.linalg.inv(C)), y_train)
variance_Xte = kernel_Xte_Xte + (1 / beta) * np.identity(len(X_test), dtype=np.float64)
variance_Xte -= np.matmul(np.matmul(np.transpose(kernel_Xtr_Xte), np.linalg.inv(C)), kernel_Xtr_Xte)
```

Finally, I visualize the result of gaussian process regression.

```
Untitled-1

plt.plot(X_test, mean_Xte, color='red')
plt.scatter(X_train, y_train, color='blue', s=5)

interval = 1.96 * np.sqrt(np.diag(variance_Xte))
X_test = X_test.flatten()
mean_Xte = mean_Xte.flatten()

plt.plot(X_test, mean_Xte + interval, color='coral')
plt.plot(X_test, mean_Xte - interval, color='coral')
plt.fill_between(X_test, mean_Xte + interval, mean_Xte - interval, color='coral', alpha=0.1)
```

Firstly, I plot the mean of distribution with a red line, and plot all training data points as blue dots. Because I want to show the 95% confidence interval of distribution, I multiply 1.96 with sigma, and represent this area with coral color.

- Part 2

In this part, we want to optimize the parameters of rational quadratic kernel in gaussian process regression by minimizing marginal negative log-likelihood. Here is the mathematic formula of negative log-likelihood and its implementation.

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

```
Untitled-1

def negative_log_likelihood(params, X_train, y_train, beta):
    sigma, alpha, length_scale = params.flatten()

    kernel_Xtr_Xtr = rational_quadratic_kernel(X_train, X_train, sigma, alpha, length_scale)
    C = kernel_Xtr_Xtr + (1 / beta) * np.identity(X_train.shape[0], dtype=np.float64)

    nll = 0.5 * np.log(np.linalg.det(C))
    nll += 0.5 * np.matmul(np.matmul(np.transpose(y_train), np.linalg.inv(C)), y_train)
    nll += 0.5 * X_train.shape[0] * np.log(2 * np.pi)
    return nll
```

The first parameter, params, contains the parameters of kernel, and the computation of C is same in part 1.

After that, I use scipy.optimize() function to find the best parameters of kernel with the bound (1e-6, 1e6), and apply these parameters to gaussian_process() again.

```
Untitled-1

opt = scipy.optimize.minimize(
    negative_log_likelihood,
    [sigma, alpha, length_scale],
    bounds=((1e-6, 1e6), (1e-6, 1e6), (1e-6, 1e6)),
    args=(X_train, y_train, beta)
)

best_sigma = opt.x[0]
best_alpha = opt.x[1]
best_length_scale = opt.x[2]

gaussian_process(
    X_train,
    y_train,
    X_test,
    beta,
    best_sigma,
    best_alpha,
    best_length_scale
)
```

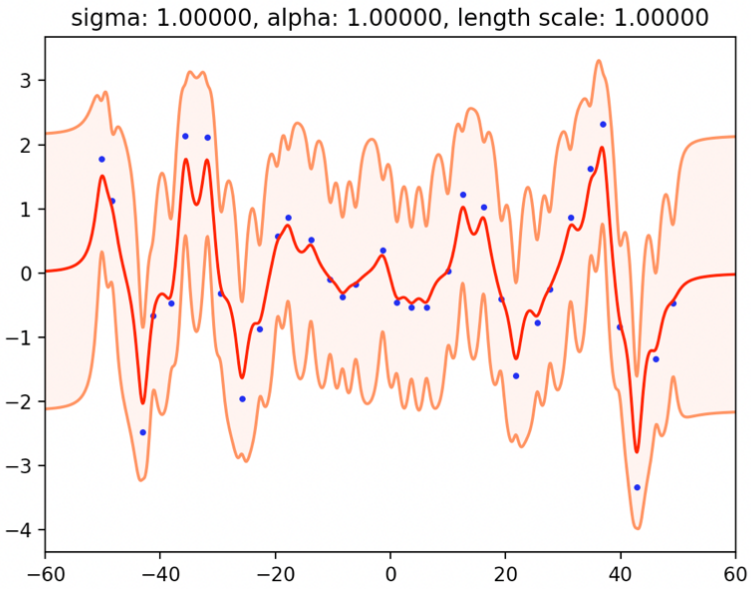
b. Experiment Setting and Result

- Part 1

  In this part, I build a gaussian process regression model to predict the distribution of data points. The table below shows the default hyperparameters I use.

Kernel Parameters	Value
Alpha	1
Beta	5
Sigma	1
Length scale	1

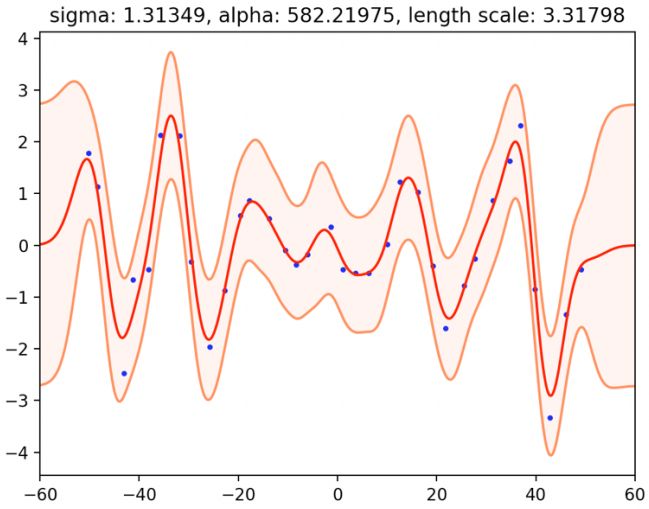
The figure below shows the visualization of gaussian process regression.



- Part 2

Kernel Parameters	Value
Alpha	582.21975
Beta	5
Sigma	1.31349
Length scale	3.31798

The figure below shows the visualization of gaussian process regression.



c. Observations and Discussion

- Performance of different hyperparameters.

  - Sigma: control the distance between two coral lines

  - Length scale: control the smoothness of regression model. In part 1, the length scale is too small which leads to wiggy regression model (overfitting). In part 2, the length scale is large enough which leads to smooth regression model. However, if length scale is too big, it is difficult for regression model to fit in data points (underfitting).

II. Support Vector Machine

a. Code Explanation

- Load Dataset

```
Untitled-1

def read_dataset(path="data"):

    # X_train
    with open(f"{path}/X_train.csv", "r") as f:
        contents = f.readlines()
    X_train = [list(map(float, text.strip().split(","))) for text in contents]
    X_train = np.array(X_train, dtype=np.float32)

    # y_train
    with open(f"{path}/Y_train.csv", "r") as f:
        contents = f.readlines()
    y_train = [[int(text.strip())] for text in contents]
    y_train = np.array(y_train, dtype=np.int8).flatten()

    ...
```

I read all dataset with read_dataset() function. In this function, it uses different way to process feature files and label files.

When processing feature file, it reads all lines of texts in csv file, splitting each line with comma and converts each number into float type.

When processing label file, it also reads all lines of texts in csv file; however, it will not split the line because the line only contains one number. The number (label) is converted into int type.

Both feature file and label file are converted into 2-dimension and 1-dimension numpy array.

- Part 1

```
Untitled-1

print("Linear Kernel: ", end="")
model = svm.svm_train(y_train, X_train, f"-t 0 -q")
pred_label, pred_acc, pred_val = svm.svm_predict(y_test, X_test, model)

print("Polynomial Kernel: ", end="")
model = svm.svm_train(y_train, X_train, f"-t 1 -q")
pred_label, pred_acc, pred_val = svm.svm_predict(y_test, X_test, model)

print("RBF Kernel: ", end="")
model = svm.svm_train(y_train, X_train, f"-t 2 -q")
pred_label, pred_acc, pred_val = svm.svm_predict(y_test, X_test, model)
```

In part 1, I use different kernels for SVM, and compare their performance. It is easy to train SVM model with svm_train() function provided by libsvm package.

The parameter -t which specifies the kernel of SVM is passed into svm_train() function. The meaning of -t is listed below:

t	*Kernel*
0	Linear
1	Polynomial
2	RBF

The parameter -q means quiet mode.

- Part 2

In part 2, I try C-SVC with different kernel, finding the best hyperparameters for each setting with grid search.

Untitled-1

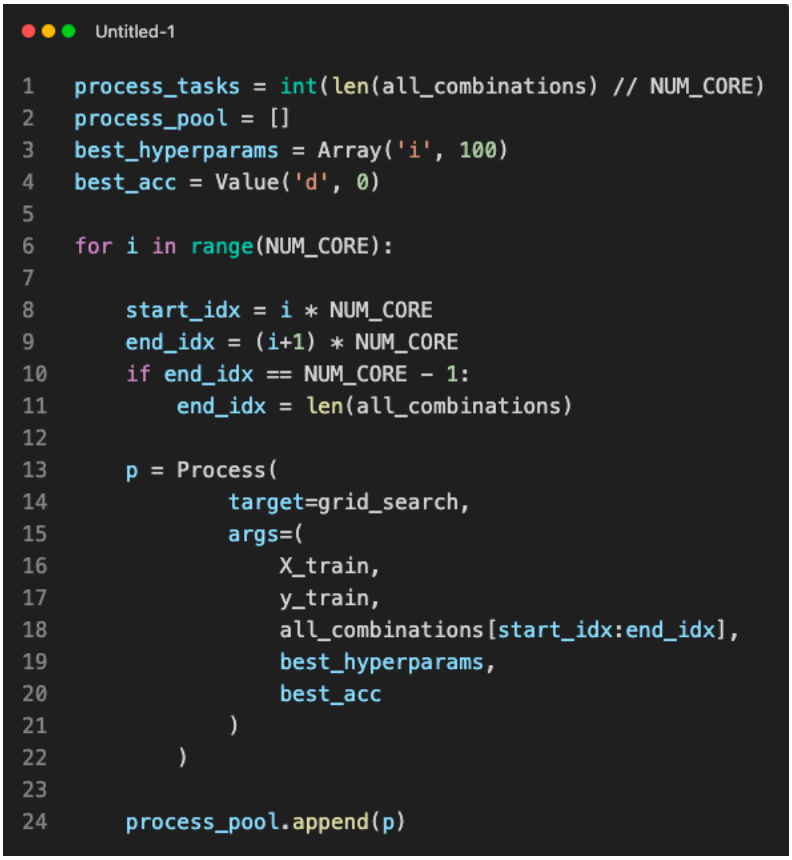
```
hyperparams = {
        'degree': [2, 3, 4, 5], # polynomail kernel (default = 3)
        'gamma': [
        0.00127551*(1/2),
        0.00127551*1,
        0.00127551*4,
        0.00127551*8,
        0.00127551*16,
        0.00127551*32,
        0.00127551*64,
        ],  # polynomail and RBF kernel (default = 1/784 = 0.00127551)
        'coef0': [0, 1, 2, 3, 4], # polynomail kernel (default = 0)
        'cost': [0.01, 0.1, 1, 10, 100, 200], # C-SVC (default = 1)
}
```

Firstly, I define the search space of hyperparameters. Each kind of hyperparameters is applied to different model's setting. For example, degree is for polynomial kernel, gamma is for polynomial and RBF kernel, coef0 is for polynomial kernel and cost is for C-SVC.

Untitled-1

```
all_combinations = []

for kernel_idx, _ in enumerate(["linear", "polynomial", "rbf"]):

    # linear kernel
    if kernel_idx == 0:
        for c in hyperparams["cost"]:
            params = f"-t {kernel_idx} -c {c} -v 3 -q"
            all_combinations.append(params)

    # polynomail kernel
    elif kernel_idx == 1:
        for c in hyperparams["cost"]:
            for g in hyperparams["gamma"]:
                for d in hyperparams["degree"]:
                    for co in hyperparams["coef0"]:
                        params = f"-t {kernel_idx} -c {c} -g {g} -d {d} -r {co} -v 3 -q"
                        all_combinations.append(params)

    # RBF kernel
    else:
        for c in hyperparams["cost"]:
            for g in hyperparams["gamma"]:
                params = f"-t {kernel_idx} -c {c} -g {g} -v 3 -q"
                all_combinations.append(params)
```

After that, I build all combinations of hyperparameters. There are 2379 combinations in total.



In order to speed up the process of grid search, I use multiprocessing technique to utilize all cores on my computer. The target function of each process is grid_search(), and I evenly distribute all 2379 combinations of hyperparameters to these processes.

```
Untitled-1

def grid_search(X_train, y_train, all_combinations, best_hyperparams, best_acc):

    for idx, params in enumerate(all_combinations):
        print(f"#{idx}: ", end="")
        train_acc = svm.svm_train(y_train, X_train, params)
        if train_acc > best_acc.value:
            with best_acc.get_lock() and best_hyperparams.get_lock():
                best_acc.value = train_acc
                for i in range(100):
                    best_hyperparams[i] = 32
                for i in range(len(params)):
                    best_hyperparams[i] = ord(params[i])
```

In grid_search() function, each process will train a SVM model with specific hyperparameters, using cross validation with 3 folds to evaluate the performance of model. In order to enable cross validation, I pass -v parameter to svm_train() function.

If cross validation accuracy is higher than the best one, I save current hyperparameters setting. Because I implement grid search with multiprocessing technique, I have to ensure that two or more process must not change the value of same variable at the same time. Therefore, get_lock() method is used to avoid this condition.

```
Untitled-1

model = svm.svm_train(y_train, X_train, "-t 1 -c 0.1 -g 0.02040816 -d 4 -r 2 -q")
pred_label, pred_acc, pred_val = svm.svm_predict(y_test, X_test, model)
```

After finding best hyperparameters, I retrain model with this specific hyperparameters, and predict on testing dataset.

- Part 3

In part 3, I try to create a custom kernel by adding linear kernel with RBF kernel, and apply svm_train() function with my custom kernel.

At first, I try to understand the exact mathematic formula of these kernel function, the source of below image is from ResearchGate.

Kernel Functions	Mathematical Expressions
Linear [26]	$E(y, y_z) = y_z^T y$
Polynomial [26]	$E(y, y_z) = (y_z^T y + 1)^x$
Radial basis function (RBF) [46]	$E(y, y_z) = e^{\frac{-\|y - y_z\|^2}{2\sigma^2}}$
Morlet wavelet [47,48]	$E(y, y_z) = \prod_{n=1}^{D} \cos\left[k_0 \frac{y^n - y_z^n}{q}\right] e^{\frac{-\|y^n - y_z^n\|^2}{2q^2}}$

After that, I implement the linear kernel and RBF kernel.

```
Untitled-1

def linear_kernel(X1, X2):
    return np.matmul(X1, np.transpose(X2))

def rbf_kernel(X1, X2, gamma):
    dist = np.sum(X1 ** 2, axis=1).reshape(-1, 1) + np.sum(X2 ** 2, axis=1) - 2 * np.matmul(X1, np.transpose(X2))
    kernel = np.exp((-1 * gamma * dist))
    return kernel
```

With these two basic kernel functions, I can compute linear and RBF kernel on training and testing dataset.

```
Untitled-1

linear_kernel_train = linear_kernel(X_train, X_train)
rbf_kernel_train = rbf_kernel(X_train, X_train, 1 / 784)

linear_kernel_test = np.transpose(linear_kernel(X_train, X_test))
rbf_kernel_test = np.transpose(rbf_kernel(X_train, X_test, 1 / 784))
```

In libsvm package, the default gamma value of RBF kernel is 1 divided by number of features. Therefore, I use 1/784 as gamma value for RBF kernel.

After computing linear and RBF on training and testing dataset, I combine these two kernels by adding them together.

```
Untitled-1

X_kernel_id_train = np.arange(1, 5001).reshape((-1, 1))
X_kernel_train = np.concatenate([X_kernel_id_train, linear_kernel_train+rbf_kernel_train], axis=1)

X_kernel_id_test = np.arange(1, 2501).reshape((-1, 1))
X_kernel_test = np.concatenate([X_kernel_id_test, linear_kernel_test+rbf_kernel_test], axis=1)
```

Because the additional ID column is required for kernel, I concatenate ID column with kernel.

```
Untitled-1

model = svm.svm_train(y_train, X_kernel_train, "-t 4 -q")
pred_label, pred_acc, pred_val = svm.svm_predict(y_test, X_kernel_test, model)
```

Ultimately, we have to tell svm_train() function that we want to use our custom kernel by using -t 4 parameter.

In addition to using default hyperparameters, I also use grid search to find the best hyperparameters. Similar to finding best hyperparameters in part 2, I firstly define the search space of hyperparameters.

```
Untitled-1

hyperparams = {
    'gamma': [
        0.00127551*(1/2),
        0.00127551*1,
        0.00127551*4,
        0.00127551*8,
        0.00127551*16,
        0.00127551*32,
        0.00127551*64,
    ],  # polynomail and RBF kernel (default = 1/784 = 0.00127551)
    'cost': [0.01, 0.1, 1, 10, 100, 200], # C-SVC (default = 1)
}
```

After that, I build all combinations of hyperparameters. There are 42 combinations in total.

```
Untitled-1

all_combinations = []
for c in hyperparams["cost"]:
    for g in hyperparams["gamma"]:
        params = f"-t 4 -c {c} -g {g} -v 3 -q"
        all_combinations.append(params)

print(f"Total number of combinations: {len(all_combinations)}")
```

Using same way (grid search + multiprocessing) in part 2 to find the best hyperparameters, the model is retrained and evaluated on testing dataset.

b. Experiment Setting and Result

- Part 1

  In this part, I compare the performance of SVM with different type of kernel with its default hyperparameters. The table shows the accuracy of model on testing dataset:

Kernel	*Accuracy*
Linear	95.08% (2377/2500)
Polynomial	34.68% (867/2500)
RBF	95.32% (2383/2500)

- Part 2

  In this part, I try C-SVC with three kinds of kernel, and find the best hyperparameters with grid search. As mentioned above, I firstly define the hyperparameter space:

Untitled-1

```
hyperparams = {
        'degree': [2, 3, 4, 5], # polynomail kernel (default = 3)
        'gamma': [
        0.00127551*(1/2),
        0.00127551*1,
        0.00127551*4,
        0.00127551*8,
        0.00127551*16,
        0.00127551*32,
        0.00127551*64,
        ],  # polynomail and RBF kernel (default = 1/784 = 0.00127551)
        'coef0': [0, 1, 2, 3, 4], # polynomail kernel (default = 0)
        'cost': [0.01, 0.1, 1, 10, 100, 200], # C-SVC (default = 1)
}
```

Secondly, I generate all combinations of these hyperparameters. There are 2379 combinations in total. I use grid search to find the best hyperparameters by evaluating model's performance with three-fold cross validation.

The best hyperparameters are found, and model is retrained. Finally, model gets 98.04% accuracy on testing dataset.

Untitled-1

```
model = svm.svm_train(y_train, X_train, "-t 1 -c 0.1 -g 0.02040816 -d 4 -r 2 -q")
pred_label, pred_acc, pred_val = svm.svm_predict(y_test, X_test, model)
```

From the image, we can find that the best hyperparameters are "-t 1 -c 0.1 -g 0.02040816 -d 4 -r 2 -q". The table shows the meaning of each parameter:

Parameter	Meaning
-t 1	Polynomial kernel
-c 0.1	Cost = 0.1 for C-SVC
-g 0.02040816	Gamma = 0.02040816 for polynomial kernel
-d 4	Degree = 4 for polynomial kernel
-r 2	Coef0 = 2 for polynomial kernel
-q	Enable quiet mode when training model

- Part 3

  In this part, I create a custom kernel by combing linear and RBF kernel. I use two kinds of hyperparameters to evaluate model on testing dataset. One is default hyperparameters, the other is best hyperparameters found by grid search.

Hyperparameters	*Accuracy*
Default	95.08% (2377/2500)
Best	95.96% (2399/2500)

c. Observations and Discussion

- Why some kernels have better performance than others?

  Intuitively, I think more complex feature mapping involved in kernel function will result in better performance of SVM. Take part 1 for instance, because RBF kernel is more complex than linear kernel, the performance of RBF kernel is also better than linear kernel.

  However, if we compare the result of linear and polynomial kernel, we will find the contradiction. Polynomial kernel is more complex than linear kernel, but getting worse result.

  I think the experiment in part 2 can explain this contradiction. If we find a set of proper hyperparameters for polynomial kernel, we can enhance the accuracy from 34.68% to 98.04%.

  In conclusion, I think the complexity of kernel affects the performance of SVM to some degree, and appropriate hyperparameters play an important role as well.

- Try different user-defined kernel function and compare the result.

In part 3, I combine linear kernel and RBF kernel by adding them together. However, the result is almost same as only using linear kernel.

Because I think that linear kernel may dominates the result I try to change the weight of linear kernel and RBF kernel when adding them together, the result accuracy on testing dataset is listed below.

Different User-Defined Kernel	Accuracy (Best HP)	Accuracy (Default HP)
0.9 x Linear + 0.1 x RBF	95.92%	95.08
0.8 x Linear + 0.2 x RBF	95.88%	95.16%
0.7 x Linear + 0.3 x RBF	95.76%	95.24%
0.6 x Linear + 0.4 x RBF	95.56%	95.32%
0.5 x Linear + 0.5 x RBF	95.56%	95.52%
0.4 x Linear + 0.6 x RBF	95.56%	95.64%
0.3 x Linear + 0.7 x RBF	95.28%	95.84%
0.2 x Linear + 0.8 x RBF	95.16%	95.72%
0.1 x Linear + 0.9 x RBF	94.68%	95.80%

From the above table, I find that if using best hyperparameters found with grid search, the bigger the weight of linear kernel, the better the performance of model. However, if using default hyperparameters, it seems that the smaller weight of linear kernel yields better result.