

HW5: GP and SVM

Gaussian Process

Task 1: Basic Mode

Code Explanation

the main structure

```
1  if __name__ == '__main__':
2      X_train, y_train = read_file('./data/input.data')
3      beta_inv = 1/5
4      params = np.array([1.0, 1.0, 1.0])
5      n = 1000
6
7      # training
8      mu, cov_inv = training(X_train, beta_inv, params)
9      # testing
10     X_test, y_test, y_test_Upper, y_test_Lower = testing(X_train, y_train,
11                                                         beta_inv, params, cov_inv, n)
12     # drawing out result
13     drawout(X_train, y_train, X_test, y_test, y_test_Upper, y_test_Lower,
14             params, 'Basic')
```

Firstly, read the input file from `./data/input.data`, using the function `read_file()`. The data will be separated into 2 numpy arrays: `X_train` and `y_train`, the explanatory variable X_i for all training data points and the target variable Y_i for all training data points, respectively.

Then, initializing some parameters:

1. `beta_inv` is the inverse of the precision β , which is set to **5** in this implementation.

β^{-1} is the variance of the white noise of data and is set to $\frac{1}{5}$.

For each data point (X_i, Y_i)

$$Y_i = f(X_i) + \epsilon_i, \text{ where } \epsilon_i \sim N(0, \beta^{-1})$$

2. `params` is a numpy array of length 3, containing the 3 parameters used in **rational quadratic kernel**, listed in this order: α, l, σ^2 . For detailed descriptions, please refer to the `kernel1()` function section. For now, all 3 parameters are initialized as **1.0**.

3. n is the number of testing data. Note that our testing data are generating from segmenting the interval **[-60, 60]** evenly. The testing size is set to be $n = 1000$ here.

Basic mode (without optimization)

In the **training** session, put the the explanatory variable array of training data x_{train} , the variance of white noise β_{inv} and the parameters params into the function `training()`.

After that, we will get the mean vector μ and the inverse of the covariance matrix cov_inv which will be used in the testing session.

In the **testing** session, put both the explanatory variable array and the target variable array of training data x_{train} and y_{train} , the variance of white noise β_{inv} and the parameters params , the inverse of covariance matrix cov_inv , the size of testing data n into the function `testing()`.

After that, we will get the testing data x_{test} , the mean of predictive distributions over the distributions f , y_{test} , and both the upper bound and the lower bound of 95% confidence interval $y_{\text{test_Upper}}$ and $y_{\text{test_Lower}}$.

Finally, we will use both the data points from training data and testing data, as well as the bounds of 95% confidence interval and the parameters, to draw out the result figure in function `drawout()`.

function: **read_file()**

```
1 def read_file(path: str):
2     X = []
3     y = []
4     with open(path, 'r') as file:
5         for line in file.readlines():
6             X.append(float(line.split()[0]))
7             y.append(float(line.split()[1]))
8     return np.array(X).reshape(-1, 1), np.array(y).reshape(-1, 1)
```

This function is to read data lines from the input file and return two numpy x and y , each with the shape (34, 1). The input data is a 34x2 matrix.

function: **kernel()**

```
1 def kernel(X1, X2, params):
2     alpha = params[0]
3     length = params[1]
4     sigma2 = params[2]
5     return sigma2 * ((1 + cdist(X1, X2, metric = 'sqeuclidean')
6     / (2 * alpha * (length ** 2))) ** (-alpha))
```

This is the kernel function for computing similarities between different points.

There are 3 parameters in `param`, in this order `alpha` for α , `length` for l , and `sigma2` for σ^2 .

```
1 alpha = params[0]
2 length = params[1]
3 sigma2 = params[2]
```

The kernel function used here is ***rational quadractic kernel***, which is defined as

$$k(X_1, X_2) = \sigma^2 \left(1 + \frac{\|X_1 - X_2\|^2}{2\alpha l^2}\right)^{-\alpha}$$

with σ^2 being the **overall variance**, α is the **scale-mixture** and l is the **lengthscale**. So, there are 3 adjustable parameters. For the square Euclidean distance part $\|X_1 - X_2\|^2$, we use the function `cdist()` from ***scipy.spatial.distance***, with the metric argument being `sqeuclidean`.

More discussions around this kernel will be described in the **Observation and Discussion** section in **task 2 (optimized mode)**.

function: **training()**

```
1 def training(X, beta_inv, params):
2     mu = np.zeros(X.shape)
3     cov = kernel(X, X, params) + beta_inv * np.identity(X.shape[0])
4     cov_inv = np.linalg.inv(cov)
5
6     return mu, cov_inv
```

In the training session, we simply calculate the mean vector and the covariance matrix of the marginal likelihood, which follows a multivariate Gaussian

distribution. For $\vec{f} = \begin{bmatrix} f(x_1) \\ \vdots \\ f(x_{34}) \end{bmatrix}$ and $\vec{y} = \begin{bmatrix} y_1 \\ \vdots \\ y_{34} \end{bmatrix}$, we have

the prior and likelihood as:

$$P(\vec{y}|\vec{f}) \sim N(\vec{f}, \beta^{-1}I_N) = N(\vec{f}, \frac{1}{5}I_{34})$$

$$P(\vec{f}) \sim N(\vec{0}, K)$$

where K is a Gram matrix with each element being a kernel function $k(X_1, X_2)$.

I is an identity matrix.

The marginal likelihood will then be

$$P(\vec{y}) = \int P(\vec{y}|\vec{f})P(\vec{f})d\vec{f} \sim N(\vec{0}, C)$$

Thus, the mean vector μ is a zero vector with shape of $(34, 1)$.

And the covariance matrix cov is a $(34, 34)$ matrix, calculated as

$$C = K + \beta^{-1}I = K + \frac{1}{5}I$$

Later on testing session, only the **inverse** of the covariance matrix will be used.

So, we return the inverse of the covariance matrix, C^{-1} only.

function: **testing()**

```

1 def testing(X, y, beta_inv, params, cov_inv, n):
2     X_test = np.linspace(-60, 60, n).reshape(-1, 1)
3     y_test = np.empty(n).reshape(-1, 1)
4     y_test_Upper = np.empty_like(y_test)
5     y_test_Lower = np.empty_like(y_test)
6
7     kxxs = kernel(X, X_test, params)
8     kxsxs = kernel(X_test, X_test, params)
9
10    y_test = np.linalg.multi_dot([kxxs.T, cov_inv, y])
11    var = np.diag(kxsxs + beta_inv * np.identity(X_test.shape[0])
12                - np.linalg.multi_dot([kxxs.T, cov_inv, kxxs]))
13    std = np.sqrt(var).reshape(-1, 1)
14    y_test_Upper = y_test + 1.96 * std
15    y_test_Lower = y_test - 1.96 * std
16
17    return X_test, y_test, y_test_Upper, y_test_Lower

```

Given some new testing data points, (\vec{x}^*, \vec{y}^*) , we can combine the original result vector \vec{y} with \vec{y}^* , so that $\vec{y}_{N+1} = \begin{bmatrix} \vec{y} \\ \vec{y}^* \end{bmatrix}$. Thus, this vector also follows a multivariate Gaussian distribution.

$$P(\vec{y}_{N+1}) \sim N(\vec{0}, C_{N+1}), \text{ where}$$

$$C_{N+1} = \begin{bmatrix} C & k(\vec{x}, \vec{x}^*) \\ k(\vec{x}, \vec{x}^*)^T & k(\vec{x}^*, \vec{x}^*) + \beta^{-1}I \end{bmatrix}$$

C is the original covariance matrix, $k(\vec{x}, \vec{x}^*)$ is the vector of kernel functions between testing data \vec{x}^* and other training data \vec{x} . $k(\vec{x}^*, \vec{x}^*)$ is the kernel function between the testing data \vec{x}^* and itself.

Thus, the predictions of \vec{y}^* given \vec{y} , can be derive from conditioning on the distribution of \vec{y}_{N+1} . $P(y^*|\vec{y})$ follows a Gaussian distribution as well, with the mean and the variance being:

$$\mu(\vec{x}^*) = k(\vec{x}, \vec{x}^*)^T C^{-1} \vec{y}$$

$$\sigma^2(\vec{x}^*) = [k(\vec{x}^*, \vec{x}^*) + \beta^{-1}I] - k(\vec{x}, \vec{x}^*)^T C^{-1} k(\vec{x}, \vec{x}^*)$$

The identity matrix I here has the size of 1000×1000 , where 1000 is the size of testing data, in this case.

In testing session, firstly we construct the X_i for all $n=1000$ data points, which are *evenly* segmented from the interval **[-60, 60]**. Thus, the test data `x_test` has shape of (1000, 1).

As for the \bar{Y}_i , the predictions, we set it as the **mean** of the conditional distribution $P(y^*|\vec{y})$. Again, the predictions `y_test` has a shape of (1000, 1), and can be calculated as the formula mentioned above: $\mu(\vec{x}^*) = k(\vec{x}, \vec{x}^*)^T C^{-1} \vec{y}$
numpy.linalg.multi_dot() is a function for matrix multiplication, but is faster when the matrixes sizes are huge. This function will automatically decide the more efficient order while conducting dot product.

As for `kxxs`, it's just $k(\vec{x}, \vec{x}^*)$, the matrix of kernel functions between the training data \vec{x} and the testing data \vec{x}^* .

Not only the mean, we also want to draw the 95% confidence interval, so we need other two lines: an upper bound `y_test_Upper` and a lower bound `y_test_Lower`.

The shapes are the same as `y_test` above.

Also, we need to compute the standard deviation `std`, which is just the square root of variance:

$\sigma^2(\vec{x}^*) = [k(\vec{x}^*, \vec{x}^*) + \beta^{-1}I] - k(\vec{x}, \vec{x}^*)^T C^{-1} k(\vec{x}, \vec{x}^*)$
`kxsxs` is just $k(\vec{x}^*, \vec{x}^*)$, the kernel function between testing data \vec{x}^* and itself.

Last step, the 95% confidence upper bound and lower bound `y_test_Upper` and `y_test_Lower` are just $\mu(x^*) + 1.96 \times \sigma(x^*)$ and $\mu(x^*) - 1.96 \times \sigma(x^*)$, respectively.

function: **drawout()**

```

1  def drawout(X_train, y_train, X_test, y_test,
2  y_test_Upper, y_test_Lower, params, mode):
3      plt.figure()
4      plt.xlim(-60, 60)
5      plt.plot(X_test.ravel(), y_test.ravel(), color = 'black')
6      plt.plot(X_test.ravel(), y_test_Lower.ravel(), color = 'red')
7      plt.plot(X_test.ravel(), y_test_Upper.ravel(), color = 'red')
8      plt.fill_between(X_test.ravel(), y_test_Upper.ravel(),
9      y_test_Lower.ravel(), facecolor = 'pink')
10     plt.scatter(X_train, y_train, color = 'blue')
11     plt.title("Gaussian Process\n" + mode + ' Mode')
12     plt.text(-55, -4, f'alpha = {params[0]}\nlength = {params[1]}\n
13     sigma2 = {params[2]}',
14     fontsize = 12, color = 'green')
15     plt.savefig("GP_" + mode.lower() + "_mode.png")

```

This function is used to draw out the result figures. The last argument `mode` is a string, indicating whether current mode is **basic** mode or **optimized** mode.

Use `xlim()` to set the range of x-axis from **-60** to **60**.

Note that `x_test`, `y_test`, `y_test_Upper` and `y_test_Lower` are all (1000, 1) **matrixes** now. To pass them as arguments, we need to flatten them into **1D vectors**, by using `ravel()`.

1. `y_test`, the **mean** of the predictive distributions of f , colored in **black**.
2. `y_test_Lower`, the lower bound of the confidence interval, colored in **red**.
3. `y_test_Upper`, the upper bound of the confidence interval, colored in **red**.

Then, the area *between the upper bound and lower bound* is filled with color **pink** by `fill_between()`.

We also plot all the training data points `x_train` and `y_train` by **scatter** plot `scatter()`, colored in **blue**.

Experiment

All the 3 hyper parameters used in **rational quadratic kernel**, namely α , l and σ^2 are left as default settings. Our default settings are:

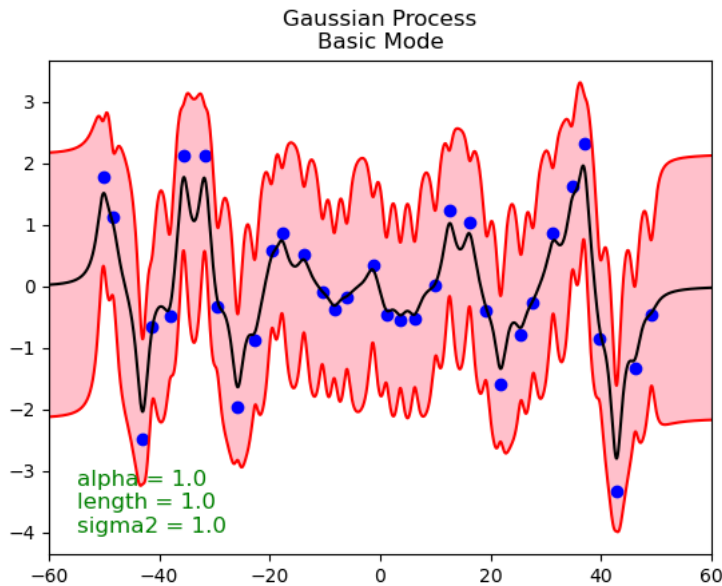
$$\alpha = l = \sigma^2 = 1$$

Again, the formula of this kernel is defined as:

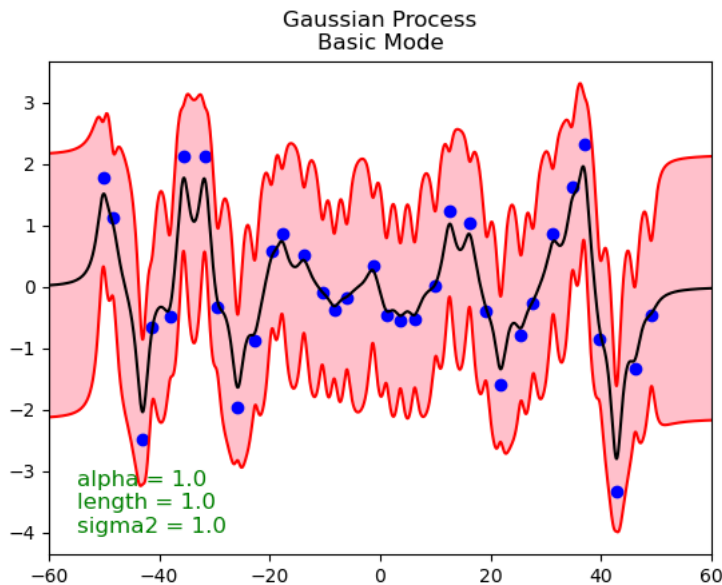
$$k(X_1, X_2) = \sigma^2 \left(1 + \frac{\|X_1 - X_2\|^2}{2\alpha l^2} \right)^{-\alpha}$$

with σ^2 being the **overall variance**, α is the **scale-mixture** and l is the **lengthscale**.

The result figure of **basic mode** (without any optimization) looks like this



Observation and Discussion



From the figure above, we can see that in basic mode, parameters are all default to **1.0** without any optimization. The **mean** (black line) seems to be pretty good, but those regions without training data seems to be unstable. Also, the **95% confidence interval** tells use that those places **with training data** generally have **smaller variance** than other places, indicating that the predictive results are more “sure” there.



The green arrow where there's a training data point, is certainly *shorter* than the orange arrow.

Task 2: Optimized Mode

Code Explanation

the main strucure

```

1  if __name__ == '__main__':
2      X_train, y_train = read_file('./data/input.data')
3      beta_inv = 1/5
4      params = np.array([1.0, 1.0, 1.0])
5      n = 1000
6
7      # optimize parameters
8      result = minimize(fun = negativeLogL, x0 = params, args = (X_train, y_train,
9      beta_inv), bounds = ((0, None), (0, None), (0, None)), method = 'L-BFGS-B')
10     print("The parameters are: ", result.x)
11
12     # training
13     mu, cov_inv = training(X_train, beta_inv, result.x)
14     # testing
15     X_test, y_test, y_test_Upper, y_test_Lower = testing(X_train, y_train,
16     beta_inv, result.x, cov_inv, n)
17     # drawing out result
18     drawout(X_train, y_train, X_test, y_test, y_test_Upper, y_test_Lower,
19     result.x, 'Optimized')

```

Firstly, read the input file from `./data/input.data`, using the function `read_file()`. The data will be separated into 2 numpy arrays: `X_train` and `y_train`, the explanatory variable X_i for all training data points and the target variable Y_i for all training data points, respectively.

Then, initializing some parameters:

1. `beta_inv` is the inverse of the precision β , which is set to **5** in this implementation.

β^{-1} is the variance of the white noise of data and is set to $\frac{1}{5}$.

For each data point (X_i, Y_i)

$$Y_i = f(X_i) + \epsilon_i, \text{ where } \epsilon_i \sim N(0, \beta^{-1})$$

2. `params` is a numpy array of length 3, containing the 3 parameters used in **rational quadratic kernel**, listed in this order: α, l, σ^2 . For detailed descriptions, please refer to the `kernel1()` function section. For now, all 3 parameters are initialized as **1.0**.
3. `n` is the number of testing data. Note that our testing data are generating from segmenting the interval **[-60, 60]** evenly. The testing size is set to be `n = 1000` here.

Optimized mode

As for optimized mode, the whole processes are basically the same. The only difference is that the parameters used `result.x` are previously optimized by minimizing (with the help of function `minimize()` from ***scipy.optimize***) the **negative marginal log-likelihood** which is calculated by function `negativeLogL()`. The computation details will be given in the description about `negativeLogL()`.

About the arguments in `minimize()`

1. `fun` is the function to be minimized, which is the negative marginal log-likelihood `negativeLogL()` here.
2. `x0` is the parameters to be optimized, which is the original parameters `params` here.
3. `args` are other arguments to be passed to the function in `fun` besides those in `x0`. Here, besides `params`, `negativeLogL()` also takes 2 other arguments `x_train` and `y_train`.
4. `bounds` are the bounds of these parameters. Since all 3 parameters α , l , σ^2 should be **positive**, the lower bounds are set to **0**. There are no limitation against upper bounds.
5. Finally, `method` is the optimization method to be used. The default method is **BFGS**, but since these parameters are bounded, more suitable method will be **L-BFGS-B**, according to the documents of `minimize()`.

function: **read_file()**

The code content of this function and the explanation about this function are all the same as those in **task 1**. So, please refer to the corresponding section. We will not list it out again because the length of the report will become too long.

function: **kernel()**

The code content of this function and the explanation about this function are all the same as those in **task 1**. So, please refer to the corresponding section. We will not list it out again because the length of the report will become too long.

function: **negativeLogL()**

In the optimized mode, we need to optimize the kernel parameters by minimizing the negative marginal log-likelihood. This function is to compute and return the **negative log-likelihood**.

```
1 def negativeLogL(params, X_train, y_train, beta_inv):
2     K = kernel(X_train, X_train, params)
3     K += np.identity(X_train.shape[0]) * beta_inv
4     term1 = 0.5 * X_train.shape[0] * np.log(2 * np.pi)
5     term2 = 0.5 * np.log(np.linalg.det(K))
6     term3 = 0.5 * np.linalg.multi_dot([y_train.T, np.linalg.inv(K), y_train])[0][0]
7     negative_log_likelihood = term1 + term2 + term3
8
9     return negative_log_likelihood
```

Given the hyper parameters θ , the conditional marginal likelihood $P(\vec{y}|\theta)$ is a function of θ .

$$P(\vec{y}|\theta) \sim N(\vec{0}, C_\theta)$$

So, the marginal log-likelihood would be

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

Thus the negative log-likelihood is

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

To maximize $\ln P(\vec{y}|\theta)$ is to minimize $-\ln P(\vec{y}|\theta)$.

C_θ here is the covariate function, which is the addition of a gram matrix with each entry being a kernel function between training data \vec{x} and itself, and the identity matrix

I times the variance β^{-1} which is `beta_inv`. Here, we use `K` to store this covariance matrix.

Now, the negative log-likelihood

`negative_log_likelihood` is just the summation over 3 terms: `term1`, `term2` and `term3`. These 3 terms are:

1. **Term1**: $\frac{N}{2} \ln(2\pi)$, where $N = 34$ is the size of training data.
2. **Term2**: $\frac{1}{2} \ln |C_\theta|$. Use `numpy.linalg.det()` to calculate the determinant of a matrix.
3. **Term3**: $\frac{1}{2} \vec{y}^T C_\theta^{-1} \vec{y}$. Again, we use `numpy.linalg.multi_dot()` to accelerate the matrix multiplication. Note that the result is a 1x1 matrix. we need to index `[0][0]` to access the value.

Finally, summing these 3 terms and return the negative log-likelihood.

function: **training()**

The code content of this function and the explanation about this function are all the same as those in **task 1**. So, please refer to the corresponding section. We will not list it out again because the length of the report will become too long.

function: **testing()**

The code content of this function and the explanation about this function are all the same as those in **task 1**. So, please refer to the corresponding section. We will not list it out again because the length of the report will become too long.

function: **drawout()**

The code content of this function and the explanation about this function are all the same as those in **task 1**. So, please refer to the corresponding section. We will not list it out again because the length of the report will become too long.

Experiment

Start from the initial settings of the hyper parameters as those in **task 1**

$$\alpha = l = \sigma^2 = 1$$

We minimize the negative log likelihood by `minimize()` from **scipy.optimize**, as described in the `function: negativeLogL()` section previously. Again, the negative log likelihood is

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

where N is the total number of **training data**, which is **34** in this implementation.

As for some worth-mentioning arguments setting in `minimize()`

```
1 result = minimize(fun = negativeLogL, x0 = params, args = (X_train, y_train,
2 beta_inv), bounds = ((0, None), (0, None), (0, None)), method = 'L-BFGS-B')
```

1. The function to be minimized is the **negative marginal log-likelihood**, which is calculated via function

`negativeLogL()` .

2. `x0` is the parameters to be optimized, which is the original hyper parameters `params` here.

```
1 params = np.array([1.0, 1.0, 1.0])
```

3. `bounds` are the bounds of these hyper parameters.

Since all 3 hyper parameters α , l , σ^2 should be **positive**, the lower bounds are set to **0**. There are no limitation against upper bounds.

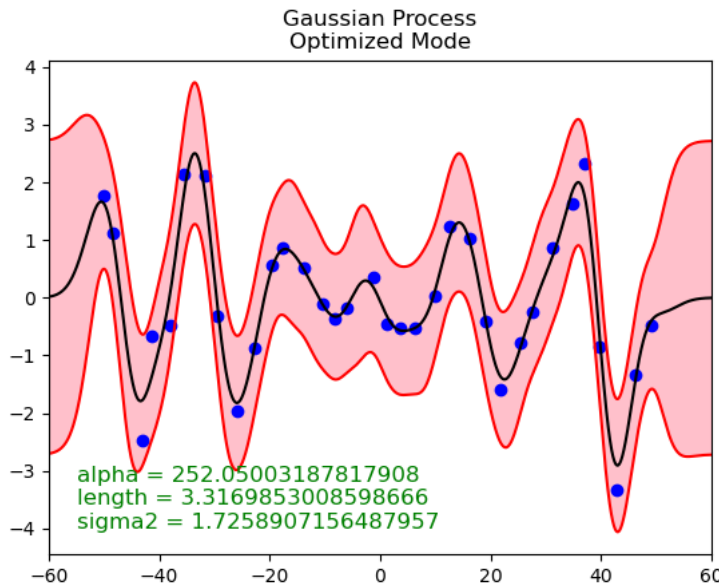
4. Finally, `method` is the optimization method to be used.

The default method is **BFGS**, but since these parameters

are **bounded**, more suitable method will be **L-BFGS-B**, according to the documents of `minimize()`. The optimized results for these 3 parameters are

$$\begin{aligned}\alpha &= 252.05003187817908 \\ l &= 3.3169853008598666 \\ \sigma^2 &= 1.7258907156487957\end{aligned}$$

The result figure with **optimized** parameters is



Observation and Discussion

One can see that the uncertainty in those areas without training data have **decreased a lot**. The **mean** (black line) and **boundaries** (red line) are way **smoother**. They do not look *serrated* anymore.

Some other discussions about the kernel parameters

The **rational quadratic kernel** will lead to a *smoother* prior on functions sampled from Gaussian Process. This kernel can be deemed as an **infinite sum of different exponentiated quadratic kernels with different lengthscale l** . And the parameter α is the **weighting** between these different lengthscales.

The **exponentiated quadratic kernel** can be computed as:

$$k(X_1, X_2) = \sigma^2 e^{-\frac{\|X_1 - X_2\|^2}{2l^2}}$$

Again, let's see the formula for **rational quadratic kernel**:

$$k(X_1, X_2) = \sigma^2 \left(1 + \frac{\|X_1 - X_2\|^2}{2\alpha l^2} \right)^{-\alpha}$$

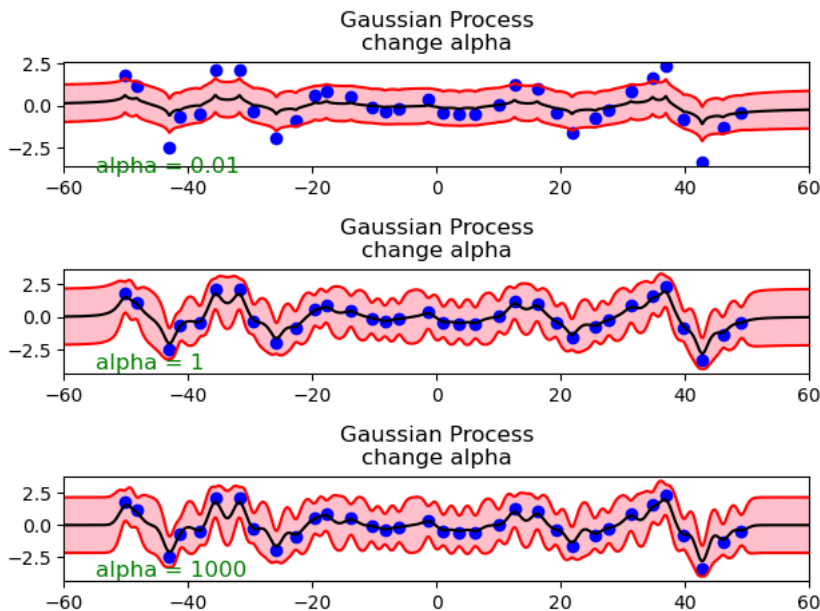
One can see that while $\alpha \rightarrow \infty$, The rational quadratic kernel actually converge into the exponentiated quadratic kernel.

$$k(X_1, X_2) = \sigma^2 \left(1 + \frac{\|X_1 - X_2\|^2}{2\alpha l^2} \right)^{-\alpha} = \sigma^2 \left(1 + \frac{\frac{\|X_1 - X_2\|^2}{2l^2}}{\alpha} \right)^{-\alpha}$$

$$\therefore \lim_{\alpha \rightarrow \infty} \left(1 + \frac{f(x)}{\alpha} \right)^{\alpha} = e^{f(x)} \text{ by definition of } e$$

$$\therefore \lim_{\alpha \rightarrow \infty} \sigma^2 \left(1 + \frac{\frac{\|X_1 - X_2\|^2}{2l^2}}{\alpha} \right)^{-\alpha} = \sigma^2 e^{-\frac{\|X_1 - X_2\|^2}{2l^2}}$$

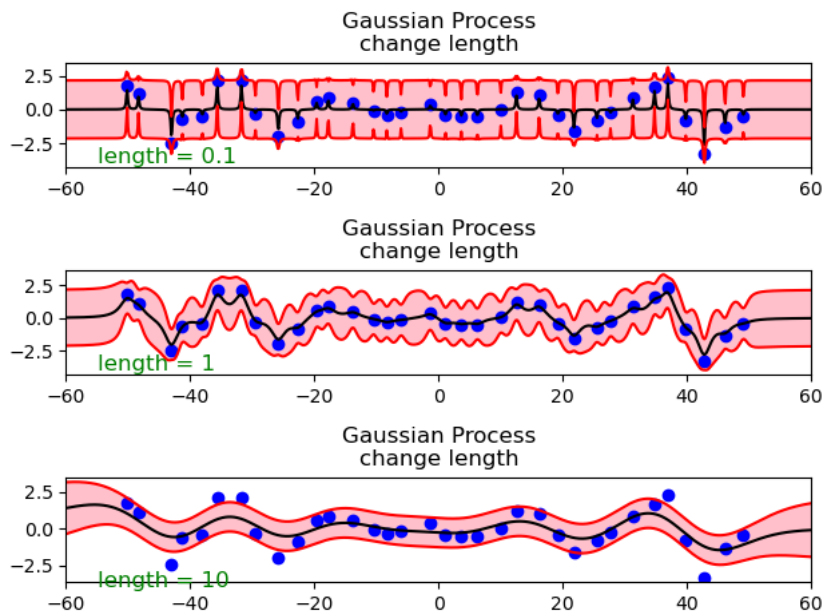
We have also tried to see the effect of the 3 parameters. First, we adjust α to be 3 different scales: **0.01**, **1** and **1000** while keeping other parameters fixed. The result figure is shown below:



One can see that α controls the minor local variations.

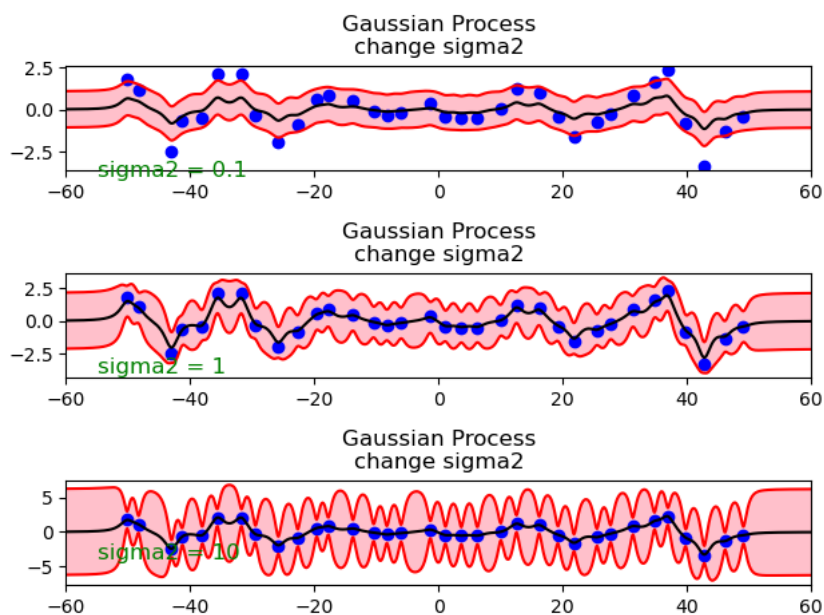
Decreasing α will allow more minor local variations.

Then, we adjust l to be 3 different scales: **0.1**, **1** and **10** while keeping other parameters fixed. The result figure is shown below:



One can see that the larger l is, the less wiggly our functions become. With big l , the functions become very **smooth**.

Finally, we adjust σ^2 to be 3 different scales: **0.1**, **1** and **10** while keeping other parameters fixed. The result figure is shown below:



This one is interesting. One can see that with large value of σ^2 , the variances where the training data are become way smaller, while on the other hand the variances of those places without training data become very very big.

SVM on MNIST

Task 1

Code Explanation

the main structure

```
1 import numpy as np
2 import csv
3 from libsvm.svmutil import *
4
5 if __name__ == "__main__":
6     X_train, y_train, X_test, y_test = read_file("./data/X_train.csv",
7     "./data/Y_train.csv", "./data/X_test.csv", "./data/Y_test.csv")
8     acc = dict()
9
10    print("Linear Kernel:")
11    problem = svm_problem(y_train, X_train)
12    param = svm_parameter("-t 0 -q")
13    model = svm_train(problem, param)
14    _, p_acc, _ = svm_predict(y_test, X_test, model)
15    acc.setdefault("Linear", p_acc[0])
16
17    print("Polynomial Kernel:")
18    problem = svm_problem(y_train, X_train)
19    param = svm_parameter("-t 1 -q")
20    model = svm_train(problem, param)
21    _, p_acc, _ = svm_predict(y_test, X_test, model)
22    acc.setdefault("Polynomial", p_acc[0])
23
24    print("RBF Kernel:")
25    problem = svm_problem(y_train, X_train)
26    param = svm_parameter("-t 2 -q")
27    model = svm_train(problem, param)
28    _, p_acc, _ = svm_predict(y_test, X_test, model)
29    acc.setdefault("RBF", p_acc[0])
```

First, import all the libraries needed, and open all 4 data csv files.

Then, for each kernel, we use the functions in the package ***libsvm.svmutil*** to conduct SVM training and inference.

For task 1, we will leave all the parameters as the default settings.

Finally, we will set a dictionary called `acc` to store the accuracies of all 3 kernels.

function: **read_file()**

```

1 def read_file(X_train_path, y_train_path, X_test_path, y_test_path):
2     with open(X_train_path, 'r') as file:
3         csv_reader = csv.reader(file)
4         X_train = list(csv_reader)
5         X_train = [[float(col) for col in row] for row in X_train]
6
7     with open(y_train_path, 'r') as file:
8         csv_reader = csv.reader(file)
9         y_train = list(csv_reader)
10        y_train = [int(label) for line in y_train for label in line]
11
12    with open(X_test_path, 'r') as file:
13        csv_reader = csv.reader(file)
14        X_test = list(csv_reader)
15        X_test = [[float(col) for col in row] for row in X_test]
16
17    with open(y_test_path, 'r') as file:
18        csv_reader = csv.reader(file)
19        y_test = list(csv_reader)
20        y_test = [int(label) for line in y_test for label in line]
21
22    return np.array(X_train), np.array(y_train), np.array(X_test), np.array(y_test)

```

This function is used to open all 4 input data csv files. It takes the paths of these 4 files as input arguments.

We get 4 lists: `x_train`, `y_train`, `x_test`, `y_test`. But before we return them, we will turn them into ***ndarray*** (numpy array) by `numpy.array()` first.

Their shapes are: `x_train = (5000, 784)`, `y_train = (5000,)`, `x_test = (2500, 784)`, `y_test = (2500,)`.

functions: functions from `svmutil`

`svm_problem()`

`svm_parameter()`

`svm_train()`

`svm_predict()`

First, use `svm_problem()` to construct a classification problem, by the training labels `y_train` and the training features `x_train`. Note that the labels come first in argument position.

Then, set the parameters to be passed in the training function globally, with the help of `svm_parameter()`. For current task 1, only 2 flags will be set.

1. **-t**: the type of kernel to be used. **1** for **linear** kernel, **2** for **polynomial** kernel, **3** for **RBF** kernel.
2. **-q**: the quiet mode, no output for the training session.

Next, we train an SVM model, simply by applying function `svm_train()` , which takes the problem we built `problem` and the parameter command we set, `param` , as arguments.

Finally, use `svm_predict` to predict the labels for testing data, and check the accuracy. This function takes the labels `y_test` and the features `x_test` of testing data as well as the model we just trained `model` as input arguments.

The function returns 3 things:

1. **p_label**: the prediction (label) for each data point.
2. **p_acc**: it's a tuple, containing 3 things
 - 2.1 the accuracy (for classification)
 - 2.2 the MSE (for regression)
 - 2.3 the R^2 (for regression)
3. **p_vals**: in **classification** case (argument flag: **-b 0**), it's a list of *decision values*.

If there are k classes, then every element in **p_vals** is a tuple containing the decision values from $\frac{k(k-1)}{2}$ binary-class SVMs.

We only need the accuracy from 2-1. Also, we will store accuracy for each 3 kernels in a dictionary `acc` by `setDefault()` method, for the purpose of printing and comparing all the results at once.

All 3 kernels: **linear**, **polynomial**, **RBF** have the same processes.

The only difference is the flag of kernel type, **-t** when passing parameters by `svm_parameter()`

1. **1** for **linear** kernel
2. **2** for **polynomial** kernel
3. **3** for **RBF** kernel.

Also, each kernel's name will be used as a **key** along with their respective accuracy as **value**, to store in the dictionary `acc` for results printing.

Experiment

For current task 1, only 2 flags will be set when passing parameters.

1. **-t**: the type of kernel to be used.

1.1 **1** for **linear** kernel

1.2 **2** for **polynomial** kernel

1.3 **3** for **RBF** kernel.

2. **-q**: the quiet mode, no output for the training session.

```
1 | param = svm_parameter("-t 0 -q") #EX: linear kernel
```

All the kernel parameters for these kernels are left as **default settings**, which can be found in the official documentation of `libsvm`.

Kernel	Parameters
Linear	$C = 1$
Polynomial	$C = 1, d = 3, \gamma = \frac{1}{num_features}, c_0 = 0$
RBF	$C = 1, \gamma = \frac{1}{num_features}$

The number of features *num_features* in this case is the number of pixels, which is

$$28 \times 28 = 784$$

Below are the formulas for each kernels. Let \vec{u} be the *feature map* of some input vector \vec{x}_1 , and \vec{v} be that of another input vector \vec{x}_2 .

1. **linear kernel**: the formula is

$$k(\vec{x}_1, \vec{x}_2) = \vec{u}^T \vec{v}$$

No parameter for this kernel.

2. **polynomial kernel**: the formula is

$$k(\vec{x}_1, \vec{x}_2) = (\gamma \vec{u}^T \vec{v} + c_0)^d$$

There are 3 parameters for this kernel: γ , c_0 and d , denoted as `gamma`, `coef0` and `degree`, respectively, in my code. Also, when passing arguments into

`svm_parameter()` , use these flags: **-g**, **-r** and **-d** for these 3 parameters, respectively.

3. **RBF kernel**: the formula is

$$k(\vec{x}_1, \vec{x}_2) = e^{-\gamma \|\vec{u} - \vec{v}\|^2}$$

There is 1 parameter for this kernel: γ , denoted as `gamma` in my code.

Since we use the default SVM, which is the **C-SVC (soft-margin SVM)**, all kernels have an additional parameter called C , denoted as `c` in my code. When passing this parameter, set the flag **-c**. The rest parameters are the parameters belong to the kernel itself.

Here are the results of task 1.

```
Linear Kernel:
Accuracy = 95.08% (2377/2500) (classification)
Polynomial Kernel:
Accuracy = 34.68% (867/2500) (classification)
RBF Kernel:
Accuracy = 95.32% (2383/2500) (classification)
```

Observation and Discussion

```
Linear Kernel:
Accuracy = 95.08% (2377/2500) (classification)
Polynomial Kernel:
Accuracy = 34.68% (867/2500) (classification)
RBF Kernel:
Accuracy = 95.32% (2383/2500) (classification)
```

The **RBF** kernel has the *highest* accuracy of **95.32%**.

Following is the **linear** kernel with the accuracy of **95.08%**, slightly worse than the that of the RBF kernel, but still very good.

Last we have the **polynomial** kernel with the accuracy of **34.68%**, which is pretty bad.

Task 2

Code Explanation

the main structure

```

1 import numpy as np
2 from libsvm.svmutil import *
3 import csv
4 import yaml
5
6 if __name__ == "__main__":
7     X_train, y_train, X_test, y_test = read_file("./data/X_train.csv",
8         "./data/Y_train.csv", "./data/X_test.csv", "./data/Y_test.csv")
9     result = dict()
10
11     cv_acc, acc, best_param = Grid_Search("Linear")
12     result.setdefault("Linear", {"validation accuracy":cv_acc,
13         "testing accuracy":acc, "best parameters":best_param})
14     cv_acc, acc, best_param = Grid_Search("Polynomial")
15     result.setdefault("Polynomial", {"validation accuracy":cv_acc,
16         "testing accuracy":acc, "best parameters":best_param})
17     cv_acc, acc, best_param = Grid_Search("RBF")
18     result.setdefault("RBF", {"validation accuracy":cv_acc,
19         "testing accuracy":acc, "best parameters":best_param})
20
21     print(yaml.dump(result, default_flow_style=False))

```

Firstly, open the features and labels for both training data and testing data and load them in `X_train`, `y_train`, `X_test`, `y_test` by the function `read_file()`. The whole processes are the same as described in **task 1**, so please refer to the `function: read_file()` section there.

And again, we set up a dictionary called `result` that will be used to store the *validation accuracy*, *testing accuracy* and the *best parameters* for each of the 3 kernels used. So, there are 3 kernels to be tested: **linear**, **polynomial** and **RBF** kernel functions. For each of them, we call the function `Grid_Search()` to find the *best parameters* with the highest *validation accuracy* as well as use this combination of parameters to make the prediction on testing data.

The function returns *validation accuracy*, *testing accuracy* and the *best parameters* for the current kernel. We will use `cv_acc`, `acc` and `best_param` to temporarily store these values, and then pass them in `result` by `setdefault()` method.

Finally, we will use the function `dump()` from the library **yaml**, to dump the `result` into **yaml** format. This step is for the purpose of **better visualization** when printing nested dictionary.

function: **Grid_Search()**

```

1 def Grid_Search(SVM_type, nfold = 5):
2     problem = svm_problem(y_train, X_train)
3     max_acc = 0.0
4
5     C = [0.001, 0.01, 0.1, 1, 10]
6     gamma = [0.001, 1/784, 0.01, 0.1, 1, 10]
7
8     if SVM_type == 'Linear':
9         param_str = "-t 0 -q"
10        best_param = {"C":None}
11        if nfold:
12            param_str += " -v "+str(nfold)
13        for c in C:
14            param = svm_parameter(param_str+" -c "+str(c))
15            cv_acc = svm_train(problem, param)
16            if cv_acc > max_acc:
17                max_acc = cv_acc
18                best_param.update({"C":c})
19        model = svm_train(problem, svm_parameter("-t 0 -q -c "+str(best_param["C"]
20        ), p_acc, _ = svm_predict(y_test, X_test, model)
21
22    elif SVM_type == 'Polynomial':
23        param_str = "-t 1 -q"
24        best_param = {"C":None, "gamma":None, "degree":3, "coef0":0}
25        if nfold:
26            param_str += " -v "+str(nfold)
27        for c in C:
28            for g in gamma:
29                param = svm_parameter(param_str+" -c "+str(c)+" -g "+str(g))
30                cv_acc = svm_train(problem, param)
31                if cv_acc > max_acc:
32                    max_acc = cv_acc
33                    best_param.update({"C":c, "gamma":g})
34        model = svm_train(problem, svm_parameter("-t 1 -q -c "+str(best_param["C"]
35       )+" -g "+str(best_param["gamma"])))
36        _, p_acc, _ = svm_predict(y_test, X_test, model)
37
38    elif SVM_type == 'RBF':
39        param_str = "-t 2 -q"
40        best_param = {"C":None, "gamma":None}
41        if nfold:
42            param_str += " -v "+str(nfold)
43        for c in C:
44            for g in gamma:
45                param = svm_parameter(param_str+" -c "+str(c)+" -g "+str(g))
46                cv_acc = svm_train(problem, param)
47                if cv_acc > max_acc:
48                    max_acc = cv_acc
49                    best_param.update({"C":c, "gamma":g})
50        model = svm_train(problem, svm_parameter("-t 2 -q -c "+str(best_param["C"]
51       )+" -g "+str(best_param["gamma"])))
52        _, p_acc, _ = svm_predict(y_test, X_test, model)
53
54    else:
55        raise ValueError("Kernel Not Available")
56
57    return float(np.round(max_acc, 2)), float(np.round(p_acc[0], 2)), best_param

```

Firstly, we again set the classification problem by

`svm_problem()` as described in **task 1**, the functions:
functions from `svmutil` section.

And then, we set up some values. `max_acc` is the current maximum accuracy in the process of grid search, and is initialized to **0.0**. Also, we need to set up candidate values for parameters to be tuned. We will simply introduce how

each kernel is calculated and what their parameters are.

Let \vec{u} be the *feature map* of some input vector \vec{x}_1 , and \vec{v} be that of another input vector \vec{x}_2 .

1. **linear kernel**: the formula is

$$k(\vec{x}_1, \vec{x}_2) = \vec{u}^T \vec{v}$$

No parameter for this kernel.

2. **polynomial kernel**: the formula is

$$k(\vec{x}_1, \vec{x}_2) = (\gamma \vec{u}^T \vec{v} + c_0)^d$$

There are 3 parameters for this kernel: γ , c_0 and d , denoted as `gamma`, `coef0` and `degree`, respectively, in my code. Also, when passing arguments into `svm_parameter()`, use these flags: **-g**, **-r** and **-d** for these 3 parameters, respectively.

3. **RBF kernel**: the formula is

$$k(\vec{x}_1, \vec{x}_2) = e^{-\gamma \|\vec{u} - \vec{v}\|^2}$$

There is 1 parameter for this kernel: γ , denoted as `gamma` in my code.

Moreover, since **C-SVM** is used here, we will have an additional parameter for penalty term called C , denoted as `c` in my code. The flag used to set up this parameter is **-c**.

Now, we can set the candidate values for these parameters

For `c` and `gamma`, we set the values ranging from **0.001** to **10**. The default value is **1** for `c`. As for `gamma`, the default value is set to the reciprocal of number of features which is the total number of pixels in this case. Thus the default value = $\frac{1}{28 \times 28} = \frac{1}{784}$.

Note that we only consider these 2 values, even for the polynomial kernel. The reason for not considering grid-searching `coef0` and `degree` is simply because of *combination explosion*. Tuning all 4 parameters at the same time is just too time-consuming. **(1 minute for each combination)**

Following are the processes of *grid-searching* the best parameters combination with highest validation accuracy by **5-fold cross-validation**. The fold number is passed as an argument `nfold` and is default to **5**.

So basically, we first set a dictionary called `best_param` which stores the current best parameters and will be updated by `update()` method when the model with the current parameters has higher validation accuracy than the value in `max_acc`. In this case, we will not only update the parameters `best_param`, but also update the current best accuracy `max_acc`.

We will use multiple *for loop* to generate different combinations of these parameters, and set the flags **-c** and **-g** to pass the C and γ values into `svm_parameter()`. For the **N-fold cross-validation** part, simply set a **-v** flag, followed by the fold number which is `nfold = 5` in this case.

When the validation mode is activated, `svm_train()` will return the **validation accuracy** instead of the model. We will use `cv_acc` to store the value.

Outside the for loops, we will use the best parameters in `best_param` to build a model, using `svm_train()`.

Finally, use `svm_predict` to make predictions, using `y_test`, `X_test` and the `model` with the parameters `best_param`.

Note that we only consider these 3 types of kernels. So, for dummy check, the code will raise a **ValueError** with message *"Kernel Not Available"* if the kernel type indicated from the argument `svm_type` does not belong to these 3. Finally, before returning the *validation accuracy* and *testing accuracy* with the best parameters `best_param`, we first use `numpy.round()` to round the numbers to **2nd**

decimal places, and then turn it back to the *float* type in Python by `float()` .

Experiment

Our candidate values for these parameters are set as follows: `c` for C in all 3 kernels, which is the penalty parameter in C-SVC. `gamma` for γ , which is used in both **polynomial** kernel and **RBF** kernel. We will list out the formula for these 2 kernels again. For **polynomial** kernel:

$$k(\vec{x}_1, \vec{x}_2) = (\gamma \vec{u}^T \vec{v} + c_0)^d$$

For **RBF** kernel:

$$k(\vec{x}_1, \vec{x}_2) = e^{-\gamma \|\vec{u} - \vec{v}\|^2}$$

```
1 c = [0.001, 0.01, 0.1, 1, 10]
2 gamma = [0.001, 1/784, 0.01, 0.1, 1, 10]
```



For `c` and `gamma` , we set the values ranging from **0.001** to **10**. The default value is **1** for `c` . As for `gamma` , the default value is set to the reciprocal of number of features which is the total number of pixels in this case. Thus the default value = $\frac{1}{28 \times 28} = \frac{1}{784}$.

Note that we only consider these 2 values, even for the polynomial kernel. The reason for not considering grid-searching `coef0` and `degree` is simply because of *combination explosion*. Tuning all 4 parameters at the same time is just too time-consuming. **(1 minute for each combination)**

To pass these 2 parameters, set up flags as: **-g** for γ and **-c** for C .



The results are shown as belows

```
Linear:
best parameters:
C: 0.01
testing accuracy: 95.96
validation accuracy: 97.04
Polynomial:
best parameters:
C: 10
coef0: 0
degree: 3
gamma: 1
testing accuracy: 97.48
validation accuracy: 97.92
RBF:
best parameters:
C: 10
gamma: 0.01
testing accuracy: 98.2
validation accuracy: 98.2
```

Kernel	validation acc(%)	testing acc(%)	best parameters
Linear	97.04	95.96	$C = 0.01$
Polynomial	97.92	97.48	$C = 10$ $c_0 = 0$ $d = 3$ $\gamma = 1$
RBF	 98.2	 98.2	$C = 10$ $\gamma = 0.01$

Observation and Discussion




```
Linear:
best parameters:
C: 0.01
testing accuracy: 95.96
validation accuracy: 97.04
Polynomial:
best parameters:
C: 10
coef0: 0
degree: 3
gamma: 1
testing accuracy: 97.48
validation accuracy: 97.92
RBF:
best parameters:
C: 10
gamma: 0.01
testing accuracy: 98.2
validation accuracy: 98.2
```

Kernel	validation acc(%)	testing acc(%)	best parameters
Linear	97.04	95.96	$C = 0.01$
Polynomial	97.92	97.48	$C = 10$ $c_0 = 0$ $d = 3$ $\gamma = 1$
RBF	 98.2	 98.2	$C = 10$ $\gamma = 0.01$

Again, $c_0 = 0$ and $d = 3$ are both default settings. We did not tune these two parameters, simply because tuning all 4 parameters will take way too long due to combination explosion.

From the table above, we can see that **RBF** kernel has the best performance, with the validation accuracy and the testing accuracy both being **98.2%**. As for the other 2 kernels, the performances are still pretty good, even the lowest one is at **95.96%**.

Comparing with the testing results of task 1, one can see that tuning parameters by grid search does help a lot in performances. Especially **polynomial** kernel, whose performance skyrockets from 34.68% to 97.48%. This further demonstrates the importance of tuning parameters.

Kernel	acc before tuning (%)	acc after tuning (%)
Linear	95.08	95.96 
Polynomial	34.68	97.48 
RBF	95.32	98.2 

Task 3

Code Explanation

the main structure

```
1 import numpy as np
2 from libsvm.svmutil import *
3 import csv
4 from scipy.spatial.distance import cdist
5
6 if __name__ == "__main__":
7     X_train, y_train, X_test, y_test = read_file("./data/X_train.csv",
8         "./data/Y_train.csv", "./data/X_test.csv", "./data/Y_test.csv")
9     # The best parameters task 2: C = 10, gamma = 0.01 for RBF, C = 0.01 for linea
10     gamma = 0.01
11
12     # calculate the precomputed kernels for both training data and testing data.
13     X_train_kernel_ver = linear_and_RBF(X_train, X_train, gamma)
14     X_test_kernel_ver = linear_and_RBF(X_test, X_train, gamma)
15
16     # Use all the things to train an SVM and make predictions.
17     # Before that, we will tune parameter C by grid-searching again.
18     acc, C = SVM(X_train_kernel_ver, y_train, X_test_kernel_ver, y_test, gamma)
19     print("The accuracy for linear kernel+RBF kernel is {}".format(acc))
20     print("The corresponding parameters are gamma = {} and C = {}".format(gamma, C))
```

In this task, first we simply load in the features and labels from training data and testing data to `X_train`, `y_train`, `X_test`, `y_test` by the function `read_file()`, as described in **task 1**, section function: `read_file()`.

As for the parameters, the best parameters from **task 2** are $C = 0.01$ for **linear** kernel and $C = 10$, $\gamma = 0.01$ for **RBF** kernel. Since there are 2 different values for C , we need to tune the parameter C by grid-searching again. This will be done in function `SVM()`. So, for now we will only need to set the γ value in `gamma` as **0.01**.

After that, we combine the **linear** kernel and the **RBF** kernel with the tuned parameters `gamma` into a newly precomputed kernel function by using the function `linear_and_RBF()`. Note that at the current stage (combining the 2 kernels), we still only need `gamma`.

For training data `X_train`, we use `X_train_kernel_ver` to store the precomputed kernel from `linear_and_RBF()`. Similarly, `X_test_kernel_ver` stores the precomputed kernel of testing data `X_test` from `linear_and_RBF()`. Note that in training session, we compute the kernels between training data `X_train` and itself, while in testing session, we compute the kernels of testing data `X_test` with respect to training data `X_train`.

Finally, we pass the original labels for training and testing data, `y_train` and `y_test`, as well as the precomputed kernels `X_train_kernel_ver` and `X_test_kernel_ver` and the tuned parameter γ `gamma` into the function `SVM()`. This function will first do the **grid-searching** with **N-fold validation** to find the best value for parameter C , and then train an SVM model with the best parameters and make predictions. More details will be reveal later at the `function: SVM()` section.

The function will then return the *testing accuracy* as well as the best value for parameter C , which will be stored in the variables `acc` and `c`, respectively. Also, print the result out.

function: **Linear()**

```
1 def Linear(u, v):  
2     return np.matmul(u, v.T)
```

This function is the **linear** kernel. The function takes **2** input vectors (or matrixes), and compute the *linear* kernels between them.

Since each data point is presented in a **row** vector, the linear kernel between vectors(matrixes) \vec{u} and \vec{v} can then be computed as

$$k(\vec{u}, \vec{v}) = \vec{u}\vec{v}^T$$

function: **RBF()**

```
1 def RBF(u, v, gamma):  
2     return np.exp(-gamma * cdist(u, v, 'sqeuclidean'))
```

This function is the **RBF** kernel. The function takes **2** input vectors (or matrixes) as well as a parameter `gamma`, and compute the *RBF* kernels between them.

For vectors \vec{u} and \vec{v} , the kernel is computed as

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

The parameter γ here is from `gamma` .

function: **linear_and_RBF()**

```
1 def linear_and_RBF(data1, data2, gamma):
2     linear = Linear(data1, data2)
3     rbf = RBF(data1, data2, gamma)
4     combined = np.add(linear, rbf)
5     X_kernel_ver = np.hstack((np.arange(1, data1.shape[0]+1).reshape(-1, 1),
6     combined))
7     return X_kernel_ver
```

Firstly, use the input data `data1` and `data2` to compute their **linear** kernels by `Linear()` , and store in the variable `linear` . Simarly, use `data1` and `data2` to compute their **RBF** kernels by `RBF()` , and store in the variable `rbf` . Then, use `numpy.add()` to combine them together, and store in the variable `combined` .

Next, we need to add an **index column** as the *first* column, from 1 to the total number of `data1` , according to the format of precomputed kernels in ***libsvm.svmutil***.

Note: `data1` stores **training** data in *training session* and **testing** data in *testing session*.

If we have N data in `data1` , we can generate a contiguous index sequence

$$1, 2, \dots, N$$

by `numpy.arange()` .

Finally, we use `numpy.hstack()` to concatenate the index column (listed above) and the precomputed kernels `combined` *horizontally*.

function: **SVM()**


```

1 def SVM(X_train, y_train, X_test, y_test, gamma, nfold = 5):
2     C = [0.001, 0.01, 0.1, 1, 10] # for RBF C = 10, while for linear C = 0.01
3     max_acc = 0.0
4     best_C = None
5
6     # training
7     problem = svm_problem(y_train, X_train, isKernel = True)
8     for c in C:
9         param_str = "-t 4 -c {} -g {}".format(c, gamma)
10        if nfold:
11            param_str += " -v {}".format(nfold)
12        param = svm_parameter(param_str)
13        cv_acc = svm_train(problem, param)
14        if cv_acc > max_acc:
15            max_acc = cv_acc
16            best_C = c
17    # testing
18    model = svm_train(problem, svm_parameter("-t 4 -c {} -g {}".format
19        (best_C, gamma)))
20    _, p_acc, _ = svm_predict(y_test, X_test, model)
21
22    return float(np.round(p_acc[0], 2)), best_C

```

Since the the best value of parameter C is different between the RBF kernel and the linear kernel,

For **RBF** = 10, while for **linear** = 0.01

we must tune the parameter by grid-searching again to find the best value. So, we first set the list `c` which contains the candidate values for C . These values are the same as those used in **task 2**.

```

1 | C = [0.001, 0.01, 0.1, 1, 10]

```

Also, `max_acc` is used to store the current best *validation* accuracy whereas `best_C` is used to store the current best value of C . `max_acc` and `best_C` are initialized as **0.0** and **None**, respectively.

As in **task 2**, we first set up a classification problem using the training features and labels, `y_train` and `X_train` with the help of function `svm_problem()`. Note that the argument **isKernel** must be set to **True** since the features `X_train` we are using now are **precomputed kernels**, not raw values.

Next, we set a *for loop* to try each different value of C in the candidate list `c`. As in task 2, we use `svm_parameter()` to pass the parameters `param`. Note that in the parameter string `param_str`, the flag **-t** indicating kernel type should be set to **4** for **precomputed kernel**. Also, use **-g** and **-c** to pass the best γ value and the current C value.

Finally, we will do the **N-fold cross-validation** here, as described in **task 2**, function: `Grid_Search()` section. So, set the flag **-v** to the fold number `nfold` which is **5** in this case.

Again, with *validation* mode on, `svm_train()` will not return a model, but the *validation accuracy*. We use `cv_acc` to store the value, and compare it with `max_acc`. If it gets better, we will update the accuracy in `max_acc` and the best value of C in `best_C`.

```
1 cv_acc = svm_train(problem, param)
2 if cv_acc > max_acc:
3     max_acc = cv_acc
4     best_C = c
```

Finally, outside the loop, when the best value of C is determined in `best_C`, we can use it along with `gamma` as parameters. We then use these parameters as well as the classification problem built `problem` to train a model `model` by `svm_train()`.

And then, in testing session, pass the **testing** data `y_test` and `x_test` as well as the SVM model `model` to make predictions, with the help of `svm_predict()`. Again, we only care about the *testing accuracy*, which is stored at the first index of `p_acc`.

This function returns 2 things: one is the **rounded testing accuracy** and the other is the best value of C , `best_C`.

```
1 return float(np.round(p_acc[0], 2)), best_C
```

Experiment

To combine the **linear** kernel and **RBF** kernel, we need to know the best parameters from **task 2**.

Kernel	Parameter C	Parameter γ
Linear	0.01	None
RBF	10	0.01

For γ , there won't be any problem because the linear kernel does not have this parameter. However, both linear kernel and RBF kernel have parameter C since we are using soft-margin SVM here, and that the best values of C are different between these two kernels!

We don't know which value to use in the mixed kernel, so we have to do grid-searching on the parameter C again. The candidate values are the same as those in **task 2**.

```
1 | C = [0.001, 0.01, 0.1, 1, 10]
```

Final result shows that the best value of C when using the combined kernel is 0.01. This can be seen from the result printed.

```
Accuracy = 96% (2400/2500) (classification)
The accuracy for linear kernel+RBF kernel is 96.0.
The corresponding parameters are gamma = 0.01 and C = 0.01.
```

We have also tried another combined kernel: **linear** kernel + **polynomial** kernel.

The whole processes are similar. The best parameters are

$$C = 0.1 \text{ and } \gamma = 0.01$$

where the other 2 parameters of polynomial kernel are left as default settings:

$$d = 3 \text{ and } c_0 = 0$$

The process are the mostly the same, just we use polynomial kernel

function: **Polynomial()**

```
1 | def Polynomial(u, v, gamma):
2 |     return np.power(gamma * np.matmul(u, v.T), 3)
```

The code is according the formula of the polynomial kernel, and the parameters' settings are described as above.

function: **linear_and_polynomial()**

Basically the same as function: `linear_and_RBF()`, just replace these 2 lines

```
1 | rbf = RBF(data1, data2, gamma)
2 | combined = np.add(linear, polynomial)
```

with the following 2 lines

```
1 | polynomial = Polynomial(data1, data2, gamma)
2 | combined = np.add(linear, polynomial)
```

to change the RBF into polynomial kernel.

Finally, in the **main structure**, when calculating the ***precomputed combined kernels*** of training data and testing data, we use the function

`linear_and_polynomial()` instead of `linear_and_RBF()` .

```
1 X_train_kernel_ver = linear_and_polynomial(X_train, X_train, gamma)
2 X_test_kernel_ver = linear_and_polynomial(X_test, X_train, gamma)
```

The result figure is

```
Accuracy = 98.08% (2452/2500) (classification)
The accuracy for linear kernel+polynomial kernel is 98.08
with the parameters: C = 0.1 and gamma = 0.01.
```

Observation and Discussion

One can see that the result shows that the **testing** accuracy for the combined kernel is **96%**, which is pretty good.

It is better than both **linear** kernel and **RBF** kernel in the default settings, which both have the accuracy at around **95%**. But it is worse than the **RBF** kernel after tuned, as we can see from **task 2** that the kernel has high accuracy at around **98%** after tuned.

As for the little experiment we did about comparing the 2 different combined kernels, namely **linear+RBF** one and **linear+polynomial** one, the results show that **linear+polynomial** has the *testing* accuracy at **98.08%** which is higher than 96% of linear+RBF kernel.