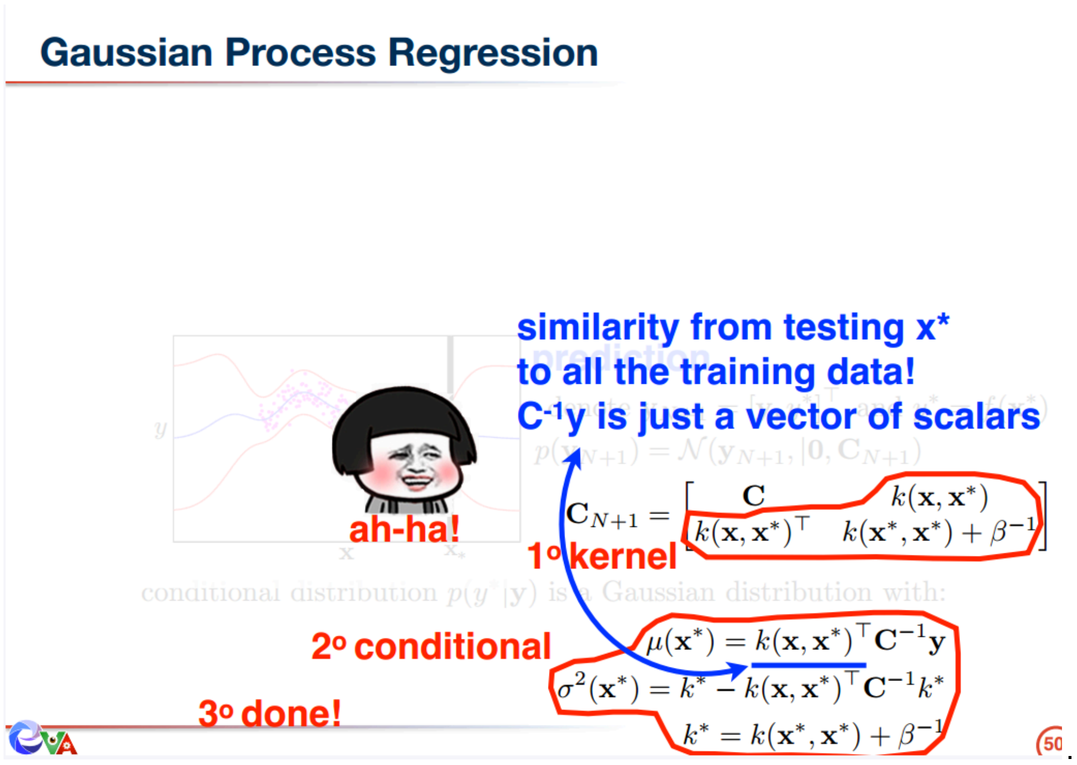
Machine Learning HW5

1. Gaussian Process:

In this part, I need to **implement the Gaussian process** and **visualize the result.** The following is the problem formulation.

- Input: X (one dimension data)
- Output: Y (one dimension data)

Follow the instructions taught from the lecture. I have three steps to execute.



Gaussian Process Regression

similarity from testing x* to all the training data!
C-1y is just a vector of scalars

$$p(\mathbf{y}_{N+1}) = \mathcal{N}(\mathbf{y}_{N+1}|\mathbf{0}, \mathbf{C}_{N+1})$$

$$\mathbf{C}_{N+1} = \begin{bmatrix} \mathbf{C} & k(\mathbf{x},\mathbf{x}^*) \\ k(\mathbf{x},\mathbf{x}^*)^\top & k(\mathbf{x}^*,\mathbf{x}^*)+\beta^{-1} \end{bmatrix}$$

ah-ha!

1o kernel

conditional distribution $p(y^*|\mathbf{y})$ is a Gaussian distribution with:

2o conditional

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

3o done!

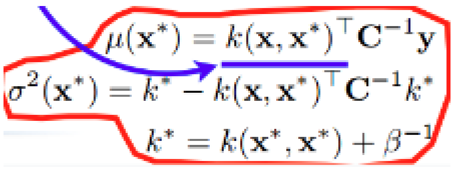
50

1. Choose kernel and construct
   I use a rational quadratic kernel.

$$\left(1+\frac{d^2}{2\alpha k^2}\right)^{-\alpha}$$

```
def calculate_rational_quatratic_kenerl(x1,x2,alpha,length_scale):
    return np.power((1 + distance.cdist(x1,x2,'euclidean'))/(2*alpha*length_scale*length_scale), -alpha)
```

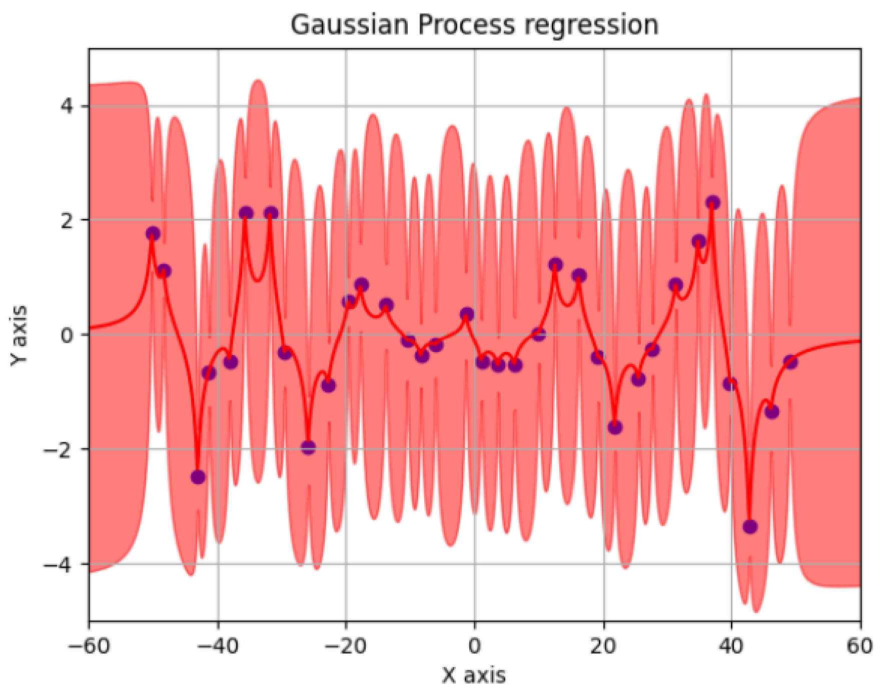
2. Construct conditional multivariate gaussian mean and variant
   I implement the equation.



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

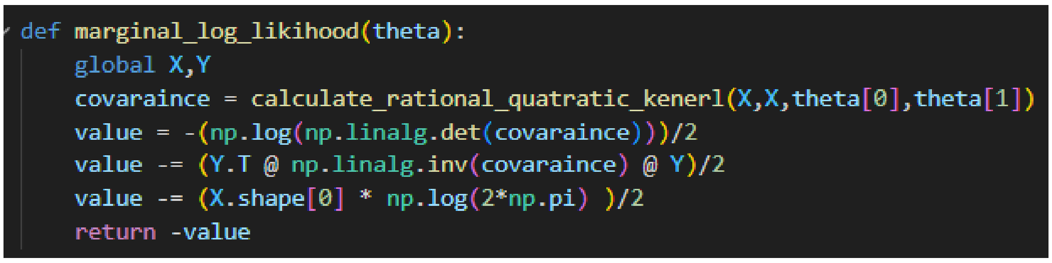
```
covariance = calculate_rational_quatratic_kenerl(X,X,alpha,length)
covariance_train_test = calculate_rational_quatratic_kenerl(X,X_test,alpha,length)
covariance_test_test = calculate_rational_quatratic_kenerl(X_test,X_test,alpha,length) + (np.eye(1000)/5)

conditional_mean = covariance_train_test.T @ np.linalg.inv(covariance) @ Y
conditional_variance = covariance_test_test - covariance_train_test.T @ np.linalg.inv(covariance) @ covariance_train_test
```

3. Visualize the result:
   According to the spec, I need to meet the following requests:
   - Show all training data points.
     I utilize matplotlib package to plot all training data on figure
     ```
     plt.scatter(X.T, Y.T, color='blue', marker='o')
     ```
   - Draw a line to represent the mean of f in range [-60,60].
     I use np.linspace to generate 1000 points from range [-60,60] and plot its corresponding value y with matplotlib.
     ```
     X_test = np.linspace(-60,60,1000)[:,None]
     ```
     ```
     plt.plot(X_test, conditional_mean, color='red', linestyle='-')
     ```
   - Mark the 95% confidence interval of f.
     Since the 95% confidence interval of f is corresponding in 1.96 standard deviation, we need to construct upper bound and lower bound like following.
     ```
     upper_bound = conditional_mean + 1.96 * conditional_variance.diagonal()[:,None]
     lower_bound = conditional_mean - 1.96 * conditional_variance.diagonal()[:,None]
     ```
     ```
     plt.fill_between(X_test[:,0], upper_bound[:,0], lower_bound[:,0], color='r', alpha=0.5)
     ```
   The visualization result is:

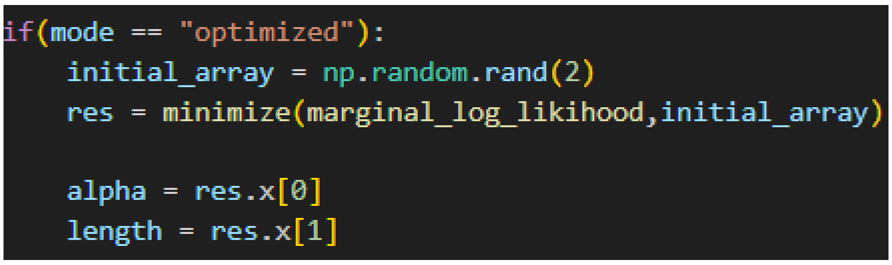


Besides, I need to optimize the parameter of kernel by **minimizing negative marginal log-likelihood**, and **visualize the result again.** I utilize the package provided by scipy.optimize to help me minimize negative marginal log-likelihood.

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

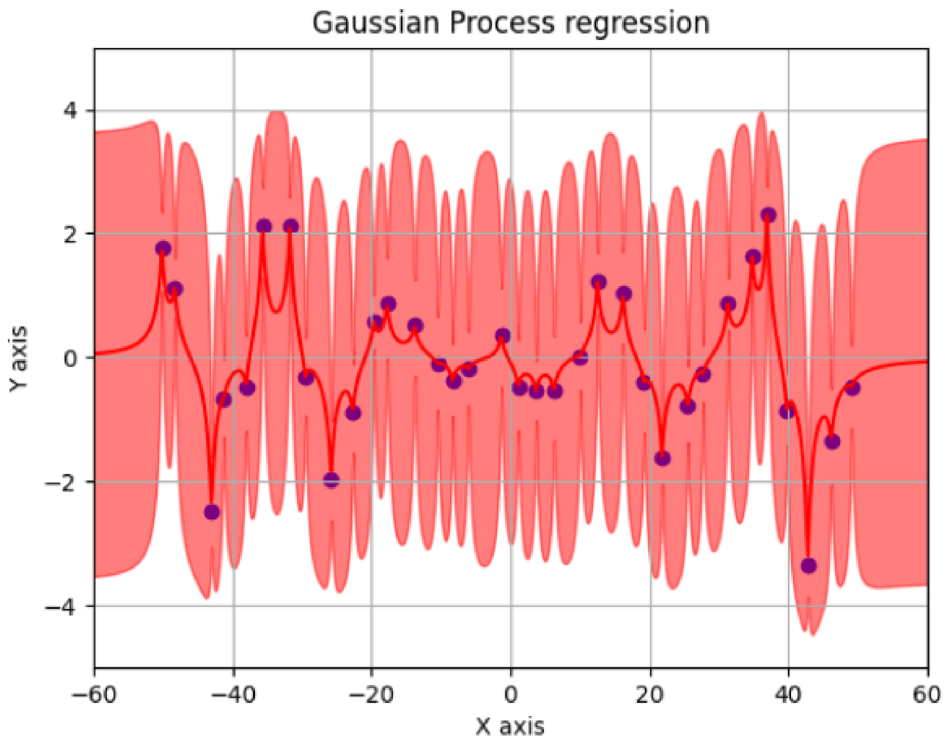


```python
def marginal_log_likihood(theta):
    global X,Y
    covaraince = calculate_rational_quatratic_kenerl(X,X,theta[0],theta[1])
    value = -(np.log(np.linalg.det(covaraince)))/2
    value -= (Y.T @ np.linalg.inv(covaraince) @ Y)/2
    value -= (X.shape[0] * np.log(2*np.pi) )/2
    return -value
```



```python
if(mode == "optimized"):
    initial_array = np.random.rand(2)
    res = minimize(marginal_log_likihood,initial_array)

    alpha = res.x[0]
    length = res.x[1]
```

The following are the visualization results with optimized parameters.

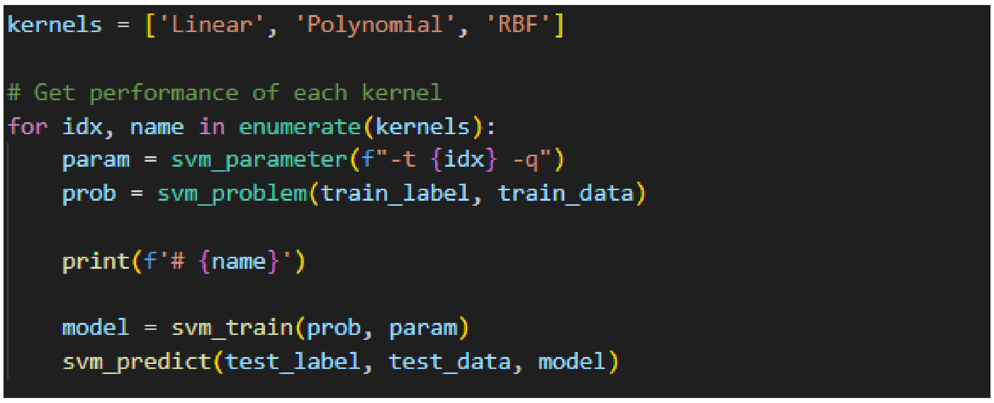


2. SVM on MNIST dataset:

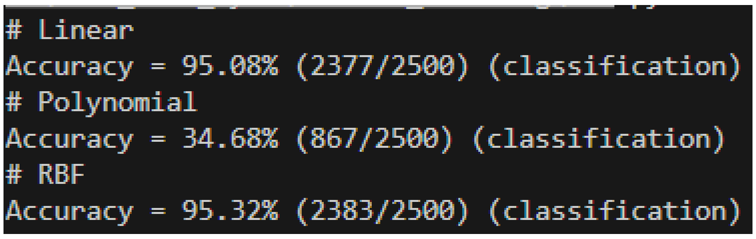
In this section, I need to complete three tasks.

- Use different kernel functions (linear, polynomial, and RBF kernels) and have a comparison between their performance.
  I utilize the package “libsvm”. First, I need to fill appropriate “parameter” and “problem” in svm_train to train the model. Finally, I use this model to predict

on test dataset.

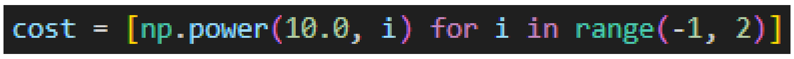


```python
kernels = ['Linear', 'Polynomial', 'RBF']

# Get performance of each kernel
for idx, name in enumerate(kernels):
    param = svm_parameter(f"-t {idx} -q")
    prob = svm_problem(train_label, train_data)

    print(f'# {name}')

    model = svm_train(prob, param)
    svm_predict(test_label, test_data, model)
```

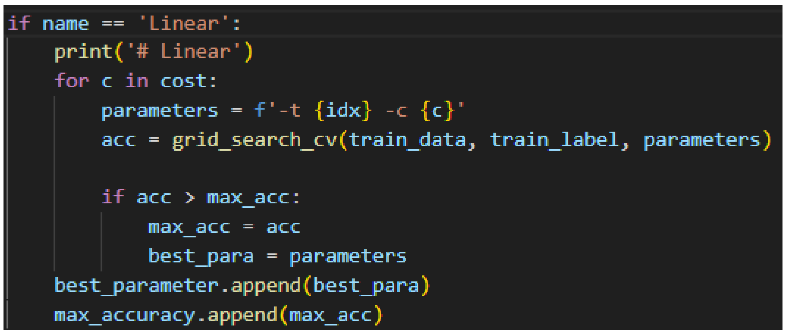


```
# Linear
Accuracy = 95.08% (2377/2500) (classification)
# Polynomial
Accuracy = 34.68% (867/2500) (classification)
# RBF
Accuracy = 95.32% (2383/2500) (classification)
```

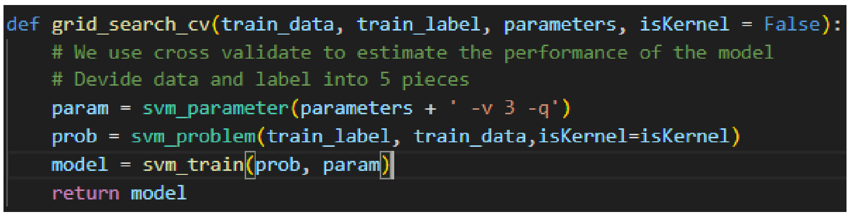
- Please use C-SVC (you can choose by setting parameters in the function input, C-SVC is soft-margin SVM). Since there are some parameters you need to tune for, please do the grid search for finding parameters of the best performing model. For instance, in C-SVC you have a parameter C, and if you use RBF kernel you have another parameter γ, you can search for a set of (C, γ) which gives you best performance in cross-validation.
  I need to do a grid search to find the best parameter for each corresponding parameter.
  - Linear
    I try different C in the linear kernel with [-0.1,0,10] and I use cross-validation to estimate the performance of each hyperparameter. Finally, I choose the parameter which has the best performance on cross-validation.



```python
cost = [np.power(10.0, i) for i in range(-1, 2)]
```

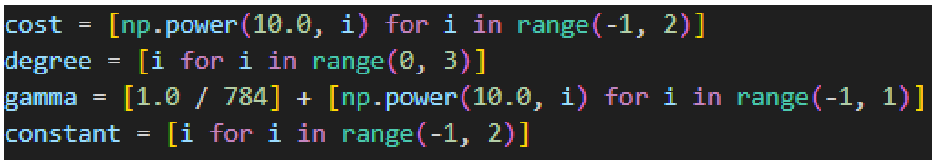


```python
if name == 'Linear':
    print('# Linear')
    for c in cost:
        parameters = f'-t {idx} -c {c}'
        acc = grid_search_cv(train_data, train_label, parameters)

        if acc > max_acc:
            max_acc = acc
            best_para = parameters
    best_parameter.append(best_para)
    max_accuracy.append(max_acc)
```

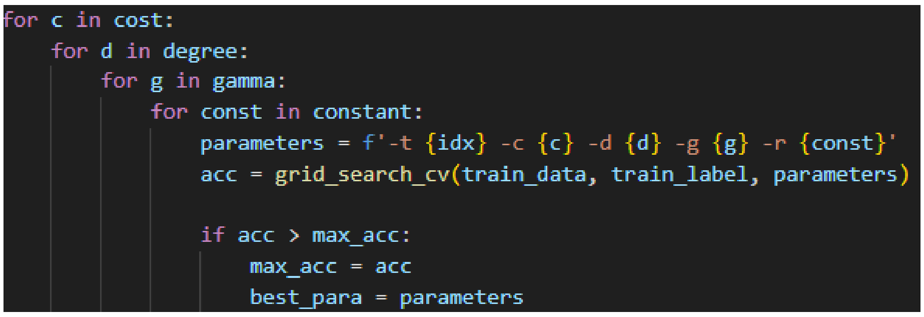


```python
def grid_search_cv(train_data, train_label, parameters, isKernel = False):
    # We use cross validate to estimate the performance of the model
    # Devide data and label into 5 pieces
    param = svm_parameter(parameters + ' -v 3 -q')
    prob = svm_problem(train_label, train_data,isKernel=isKernel)
    model = svm_train(prob, param)
    return model
```

- Polynomial
  I try different C, degree, gamma and constant in the polynomial kernel and I use cross-validation to estimate the performance of each hyperparameter.Finally, I choose the parameter which has the best performance on cross-validation.

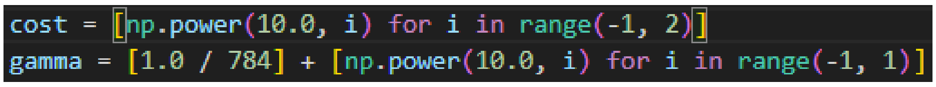


```
cost = [np.power(10.0, i) for i in range(-1, 2)]
degree = [i for i in range(0, 3)]
gamma = [1.0 / 784] + [np.power(10.0, i) for i in range(-1, 1)]
constant = [i for i in range(-1, 2)]
```

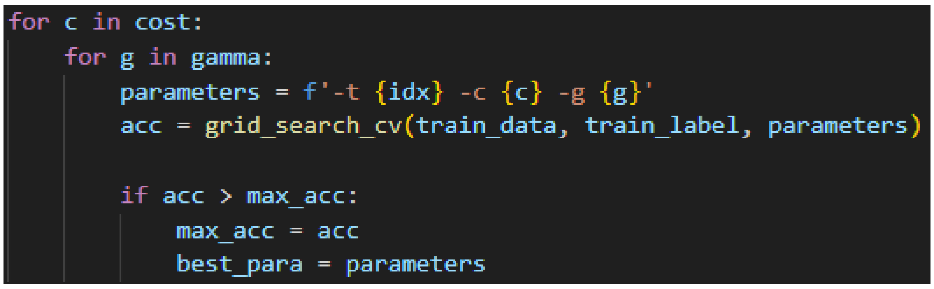


```
for c in cost:
    for d in degree:
        for g in gamma:
            for const in constant:
                parameters = f'-t {idx} -c {c} -d {d} -g {g} -r {const}'
                acc = grid_search_cv(train_data, train_label, parameters)

                if acc > max_acc:
                    max_acc = acc
                    best_para = parameters
```

- RBF
  I try different C, and gamma in the RBF kernel and I use cross-validation to estimate the performance of each hyperparameter.Finally, I choose the parameter which has the best performance on cross-validation.

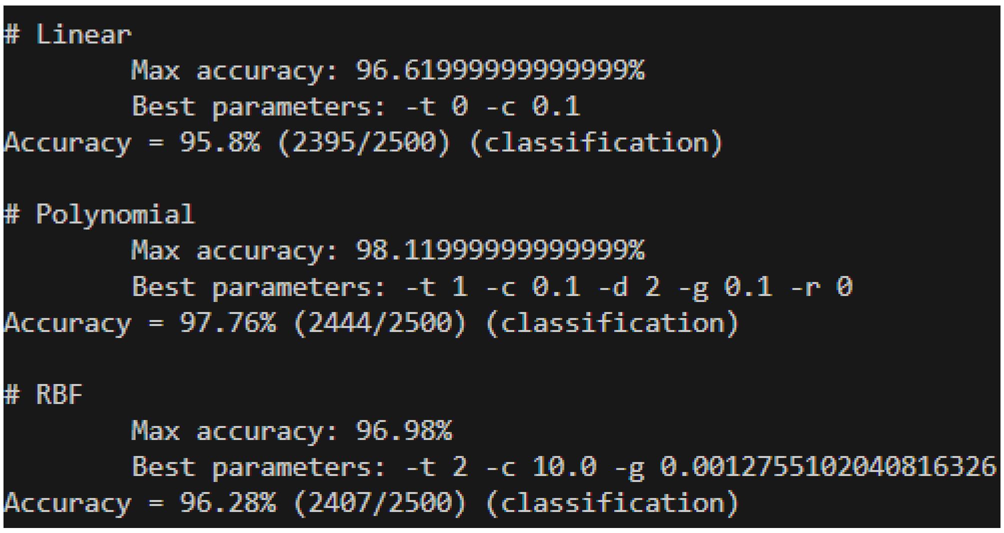


```
cost = [np.power(10.0, i) for i in range(-1, 2)]
gamma = [1.0 / 784] + [np.power(10.0, i) for i in range(-1, 1)]
```



```
for c in cost:
    for g in gamma:
        parameters = f'-t {idx} -c {c} -g {g}'
        acc = grid_search_cv(train_data, train_label, parameters)

        if acc > max_acc:
            max_acc = acc
            best_para = parameters
```

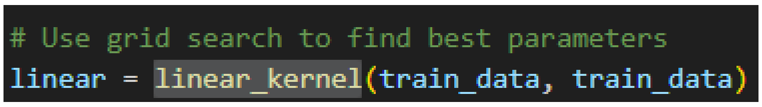
The following is the result of grid search.



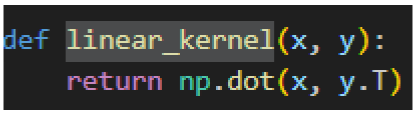
```
# Linear
        Max accuracy: 96.61999999999999%
        Best parameters: -t 0 -c 0.1
Accuracy = 95.8% (2395/2500) (classification)

# Polynomial
        Max accuracy: 98.11999999999999%
        Best parameters: -t 1 -c 0.1 -d 2 -g 0.1 -r 0
Accuracy = 97.76% (2444/2500) (classification)

# RBF
        Max accuracy: 96.98%
        Best parameters: -t 2 -c 10.0 -g 0.0012755102040816326
Accuracy = 96.28% (2407/2500) (classification)
```

3. Use linear kernel + RBF kernel together (therefore a new kernel function) and compare its performance with respect to others. You would need to find out how to use a user-defined kernel in libsvm.

To use a user-defined kernel, I need to precompute training data. In this case I need to precompute training data with a linear kernel.

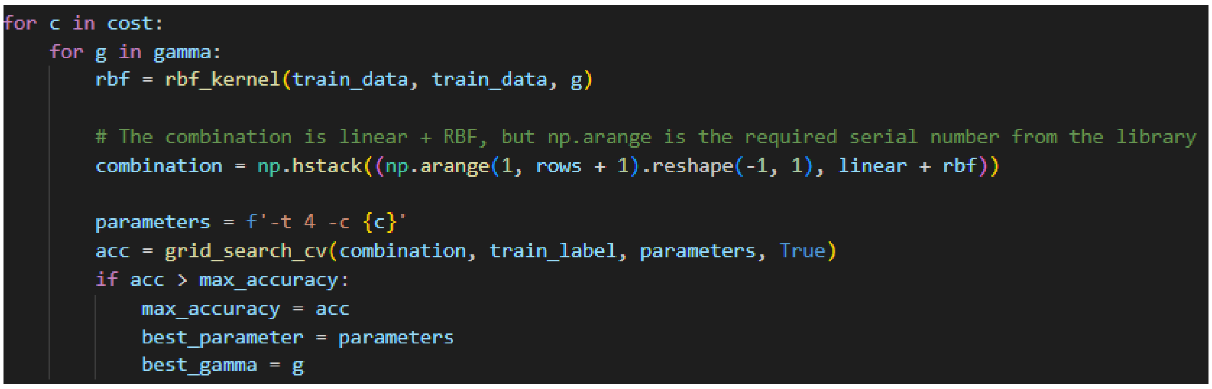


```
# Use grid search to find best parameters
linear = linear_kernel(train_data, train_data)
```



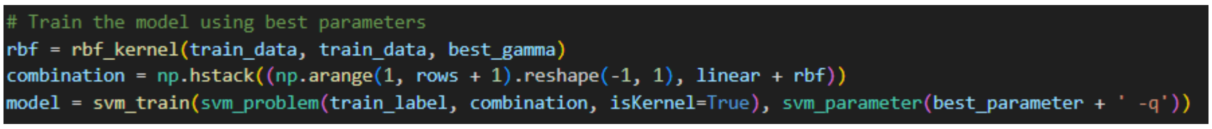
```
def linear_kernel(x, y):
    return np.dot(x, y.T)
```

I did a grid search for the RBF kernel to find out the best parameter for the RBF kernel in combination with the linear kernel.



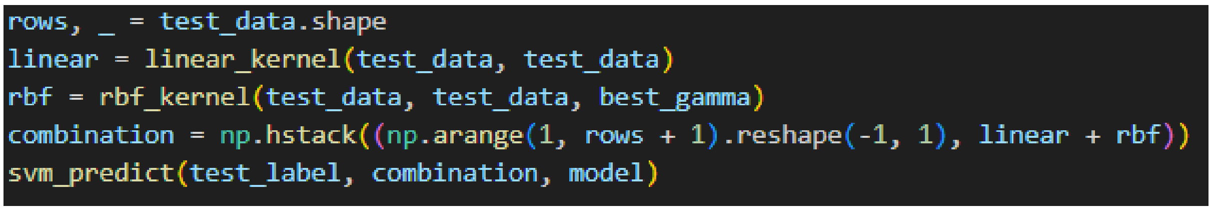
```
for c in cost:
    for g in gamma:
        rbf = rbf_kernel(train_data, train_data, g)

        # The combination is linear + RBF, but np.arange is the required serial number from the library
        combination = np.hstack((np.arange(1, rows + 1).reshape(-1, 1), linear + rbf))

        parameters = f'-t 4 -c {c}'
        acc = grid_search_cv(combination, train_label, parameters, True)
        if acc > max_accuracy:
            max_accuracy = acc
            best_parameter = parameters
            best_gamma = g
```

After finding out the best parameter, we use this set of parameters to train the model.



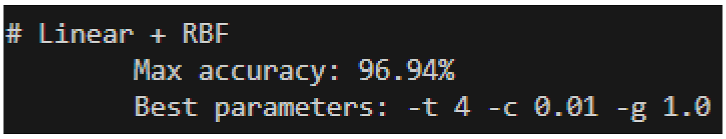
```
# Train the model using best parameters
rbf = rbf_kernel(train_data, train_data, best_gamma)
combination = np.hstack((np.arange(1, rows + 1).reshape(-1, 1), linear + rbf))
model = svm_train(svm_problem(train_label, combination, isKernel=True), svm_parameter(best_parameter + ' -q'))
```

After training the model, we use this model to do inference on the test dataset.



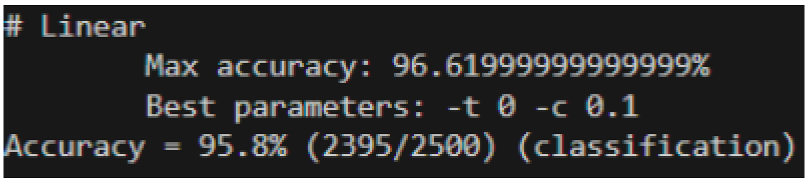
```
rows, _ = test_data.shape
linear = linear_kernel(test_data, test_data)
rbf = rbf_kernel(test_data, test_data, best_gamma)
combination = np.hstack((np.arange(1, rows + 1).reshape(-1, 1), linear + rbf))
svm_predict(test_label, combination, model)
```

The result of combination kernel is:

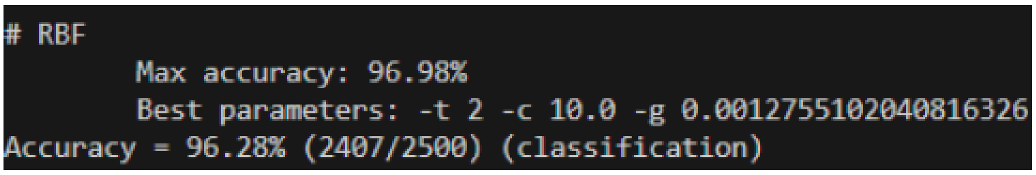


```
# Linear + RBF
        Max accuracy: 96.94%
        Best parameters: -t 4 -c 0.01 -g 1.0
```

This result is better than linear kernel and RBF kernel



```
# Linear
        Max accuracy: 96.61999999999999%
        Best parameters: -t 0 -c 0.1
Accuracy = 95.8% (2395/2500) (classification)
```



```
# RBF
        Max accuracy: 96.98%
        Best parameters: -t 2 -c 10.0 -g 0.0012755102040816326
Accuracy = 96.28% (2407/2500) (classification)
```