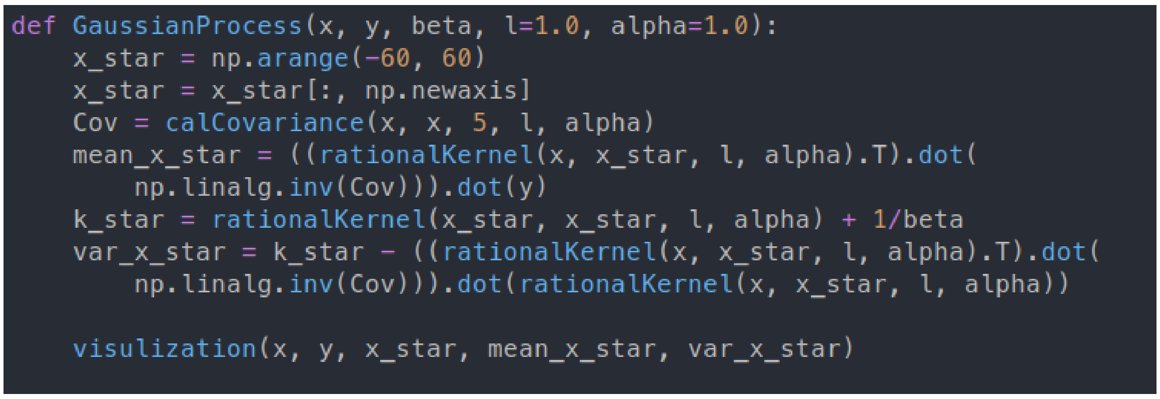
ML HW5 Report
309551131 楊子侑

I. Gaussian Process

a. Code with Detailed Explanation

Part1: Gaussian Process to predict the distribution of f



```python
def GaussianProcess(x, y, beta, l=1.0, alpha=1.0):
    x_star = np.arange(-60, 60)
    x_star = x_star[:, np.newaxis]
    Cov = calCovariance(x, x, 5, l, alpha)
    mean_x_star = ((rationalKernel(x, x_star, l, alpha).T).dot(
        np.linalg.inv(Cov))).dot(y)
    k_star = rationalKernel(x_star, x_star, l, alpha) + 1/beta
    var_x_star = k_star - ((rationalKernel(x, x_star, l, alpha).T).dot(
        np.linalg.inv(Cov))).dot(rationalKernel(x, x_star, l, alpha))

    visulization(x, y, x_star, mean_x_star, var_x_star)
```

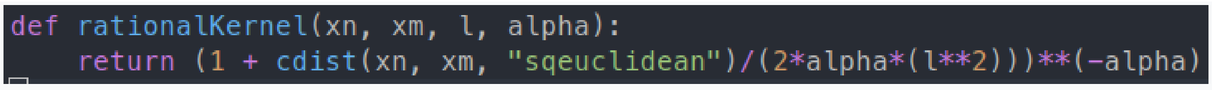
In this function, I called three other functions, that are "calCovariance" , "rationalKernel", and "visualization".

First, the formula of rational Kernel is like this:

$$k(x_i, x_j) = \left(1 + \frac{d(x_i, x_j)^2}{2\alpha l^2}\right)^{-\alpha}$$

It is parameterized by a length scale parameter l >0 and a scale mixture parameter α >0. According to the formula, I defined my function in this form:

note that the default value of l and α are 1.0



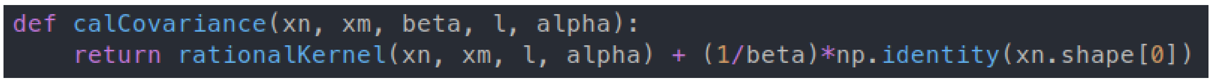
```python
def rationalKernel(xn, xm, l, alpha):
    return (1 + cdist(xn, xm, "sqeuclidean")/(2*alpha*(l**2)))**(-alpha)
```

the cdist function is to calculate the distance of two point in squared Euclidean distance metric.

Second, according to the slides, the covariance is defined as follows:

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

So the calCovariance function is written as this:



```python
def calCovariance(xn, xm, beta, l, alpha):
    return rationalKernel(xn, xm, l, alpha) + (1/beta)*np.identity(xn.shape[0])
```

Last, in Gaussian Process, we need to compute the mean and variance, the formulas are defined as follows:

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

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

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

So the 4th line to the 8th line in the Gaussian Process are calculating these. After calculating, we can obtain the mean and variance of each x*, which represents the distribution of f.
Once getting the prediction, we can visualize the results:

```python
def visulization(x, y, x_star, mean_x_star, var_x_star):
    upper = np.zeros(x_star.shape[0])
    lower = np.zeros(x_star.shape[0])
    for i in range(x_star.shape[0]):
        upper[i] = mean_x_star[i, 0] + 1.96*var_x_star[i, i]
        lower[i] = mean_x_star[i, 0] - 1.96*var_x_star[i, i]
    plt.xlim(-60, 60)
    plt.scatter(x, y, s=7.0, c='m')
    plt.plot(x_star.ravel(), mean_x_star.ravel(), 'b')
    plt.fill_between(x_star.ravel(), upper, lower, alpha=0.5)
    plt.show()
    return
```

because we want to get the 95% of confidence interval. The z-score of the 95% confidence interval is 1.96, and the formula of z-score is:

$$z = \frac{x - \mu}{\sigma}$$

so by substituting the z with 1.96, we can calculate the x, which is the upper in the function, and to calculate the lower is to substitute z with -1.96. After getting the upper and lower, we can use fill_between to plot.

Part2: Optimize the kernel parameters
According to the slides, the marginal log likelihood is as following:

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

to maximize the log likelihood equals to minimize the negative log likelihood.
So we can define the negative log likelihood as follows:

```python
def negativeMarginal(theta, args):
    C_theta = rationalKernel(
        args[0], args[0], theta[0], theta[1]) + (1/beta)*np.identity(args[0].shape[0])
    result = 1/2 * (np.log(np.linalg.det(C_theta))) + 1/2 * (args[1].T.dot(
        np.linalg.inv(C_theta))).dot(args[1]) + args[0].shape[0]/2 * np.log(2*math.pi)
    return result[0]
```

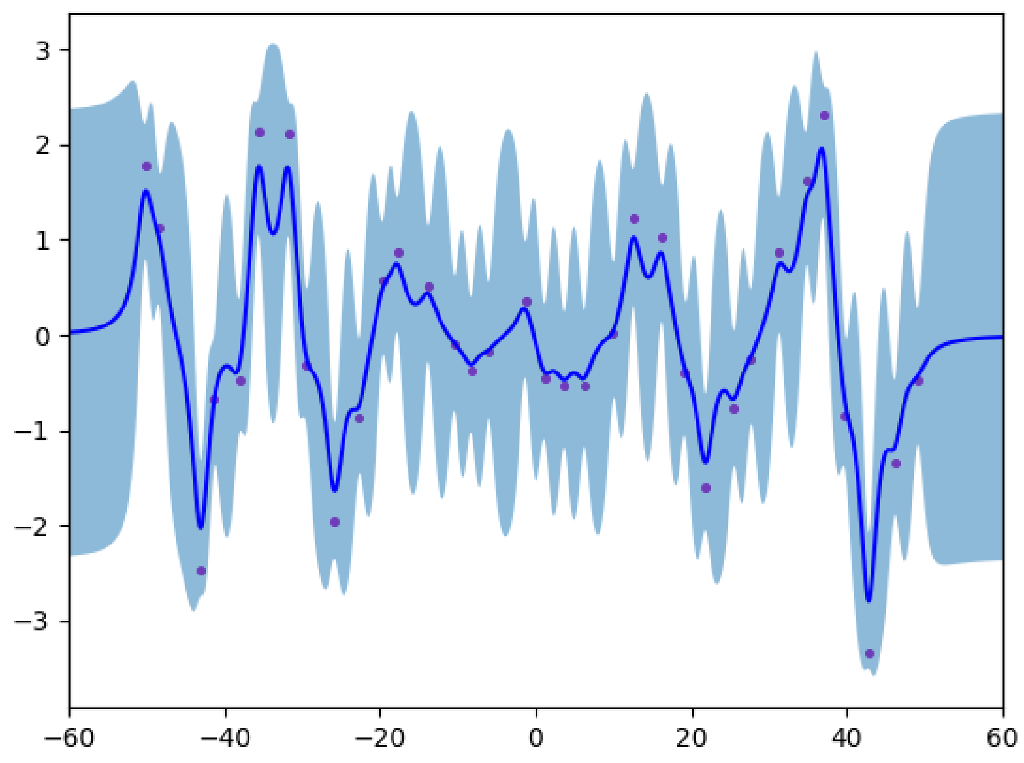
Then I use minimize in scipy to do the optimization:

```python
opt = minimize(negativeMarginal, x0=[1, 1], args=[train_x, train_y])
```

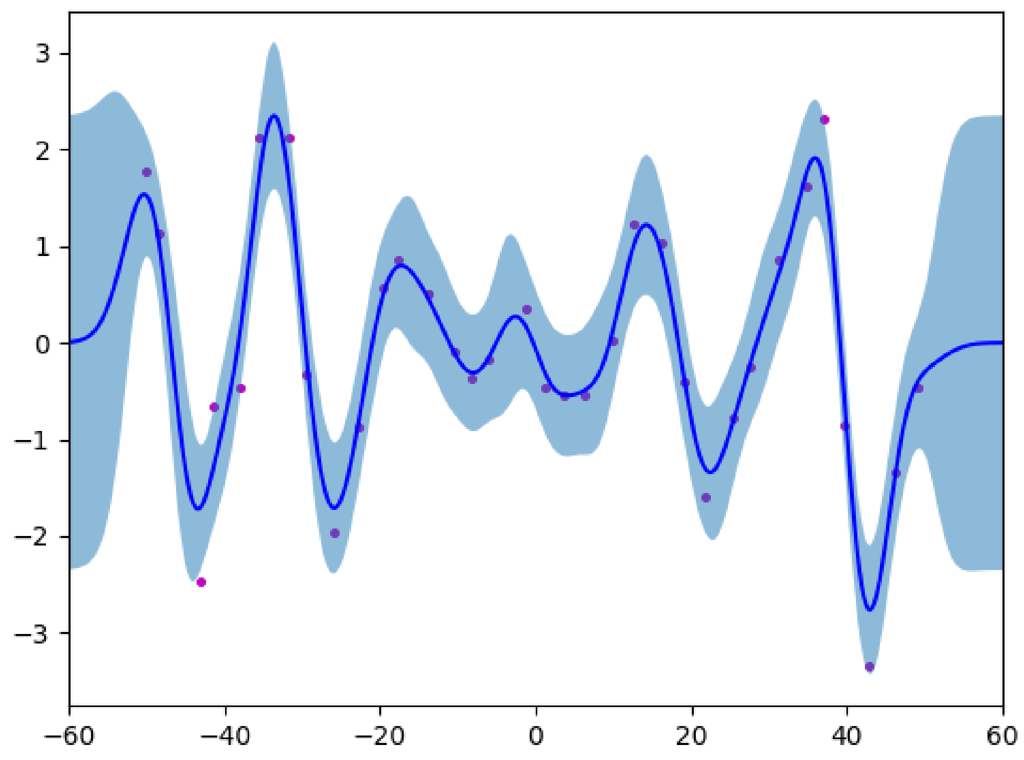
The initial value of the parameters are [1,1], and the opt represents the optimal solutions.
Then we can do the Gaussian Process again and visualize the results.

b. Experiments Settings and Results:

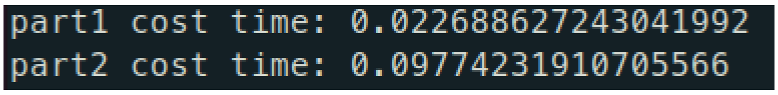
Before optimize: l = 1.0, α= 1.0



After optimize: l = 2.9638548, α = 797.27778614



c. Observations and Discussions:
   1. The optimized version has overall smaller confidence interval than original version.
   2. In the un-optimized version, the confidence interval is wider between two data points.
   3. In both ends of the two figures, the confidence intervals are wider, this means when there are no data points, it’s harder to predict.
   4. The un-optimized version (l, α = 1.0)is about 4.3x times faster then the optimized version.

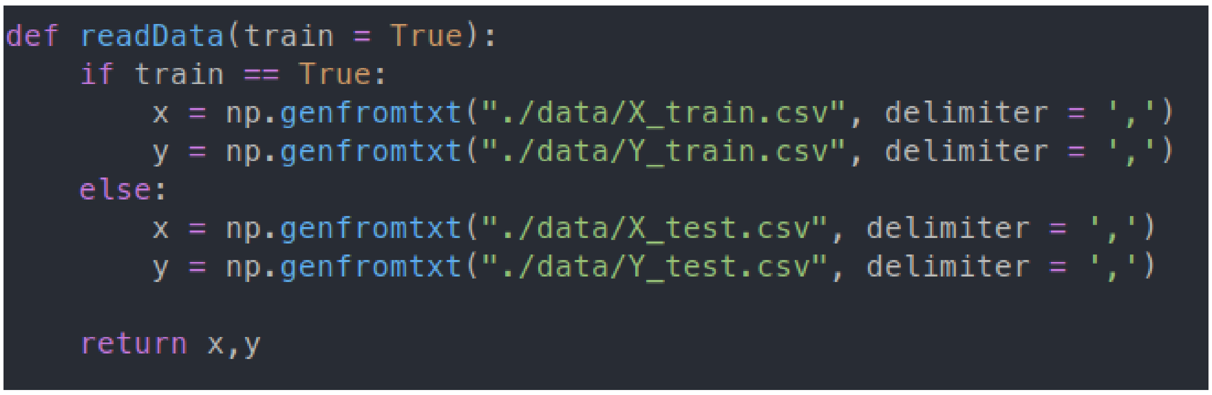


```
part1 cost time: 0.022688627243041992
part2 cost time: 0.09774231910705566
```

II. SVM

a. Code with Detailed Explanations

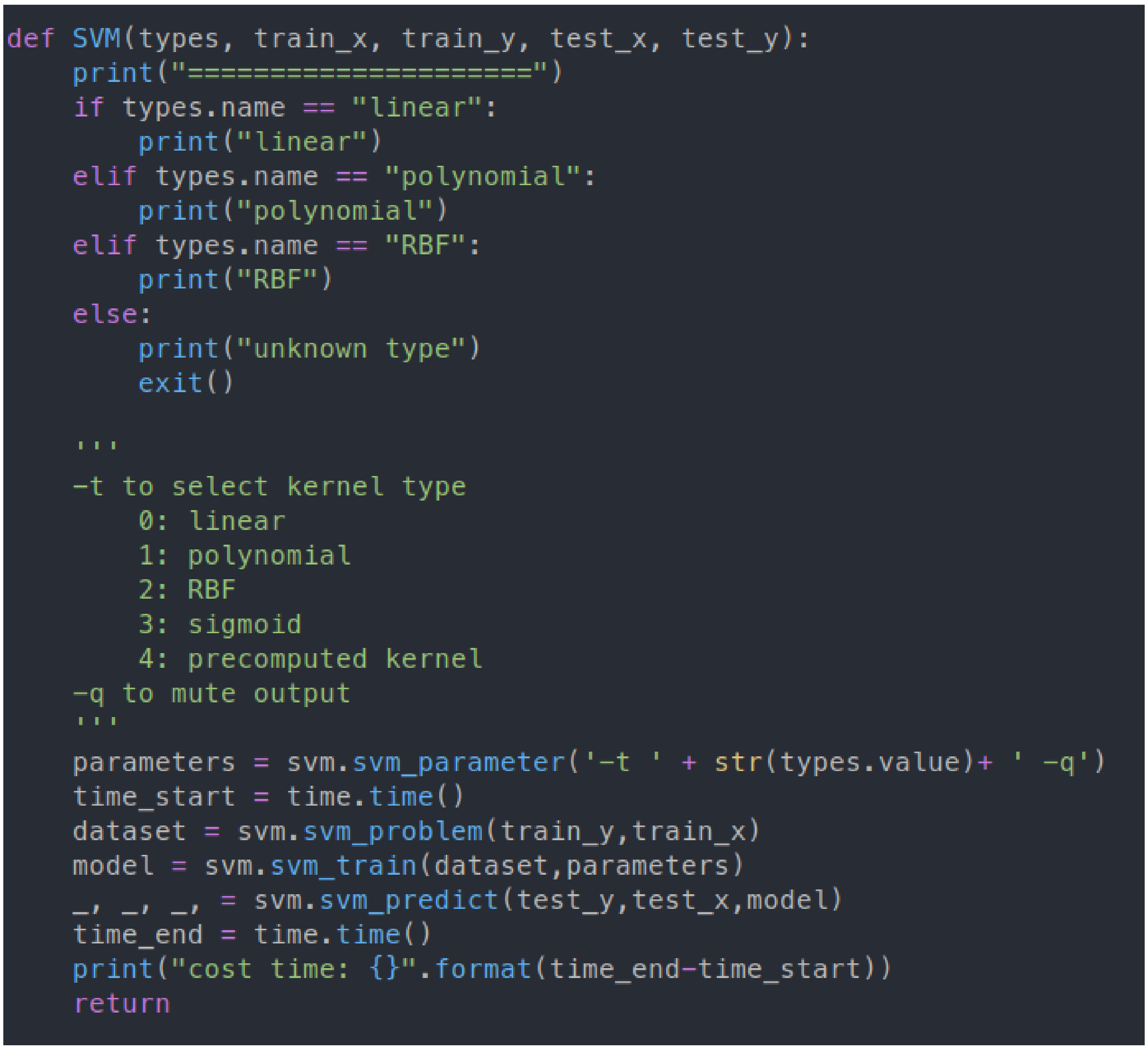
Step 0: Read Dataset:



```
def readData(train = True):
    if train == True:
        x = np.genfromtxt("./data/X_train.csv", delimiter = ',')
        y = np.genfromtxt("./data/Y_train.csv", delimiter = ',')
    else:
        x = np.genfromtxt("./data/X_test.csv", delimiter = ',')
        y = np.genfromtxt("./data/Y_test.csv", delimiter = ',')

    return x,y
```

Part1: Use different Kernel functions

In this part, we use 3 different kernels: linear, polynomial, and RBF.
I implemented the task in a function:(next page)

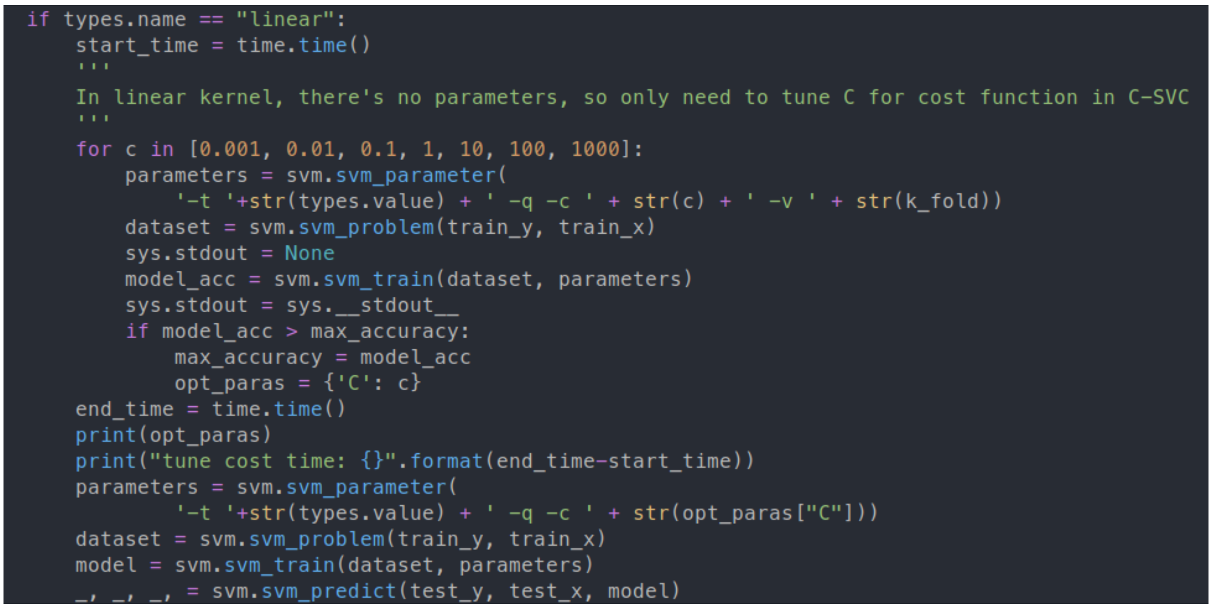


```python
def SVM(types, train_x, train_y, test_x, test_y):
    print("=====================")
    if types.name == "linear":
        print("linear")
    elif types.name == "polynomial":
        print("polynomial")
    elif types.name == "RBF":
        print("RBF")
    else:
        print("unknown type")
        exit()

    '''
    -t to select kernel type
        0: linear
        1: polynomial
        2: RBF
        3: sigmoid
        4: precomputed kernel
    -q to mute output
    '''
    parameters = svm.svm_parameter('-t ' + str(types.value)+ ' -q')
    time_start = time.time()
    dataset = svm.svm_problem(train_y,train_x)
    model = svm.svm_train(dataset,parameters)
    _, _, _, = svm.svm_predict(test_y,test_x,model)
    time_end = time.time()
    print("cost time: {}".format(time_end-time_start))
    return
```

First, I uses a variable to specify the type of kernel. The libsvm package provides -t option for us to choose some built-in kernel. 0 for linear kernel, 1 for polynomial, 2 for RBF. Then I use time package to measure performance. The svm_problem is to transform data to the format of SVM. The svm_train is to produce the model. Last, we use svm_predict to do the classification.

Part2: C-SVC and Grid Search
C-SVC is soft-margin SVM and Grid Search is a method to find the best parameter, basically, grid search is just to search all the possible combination of parameters user provided.

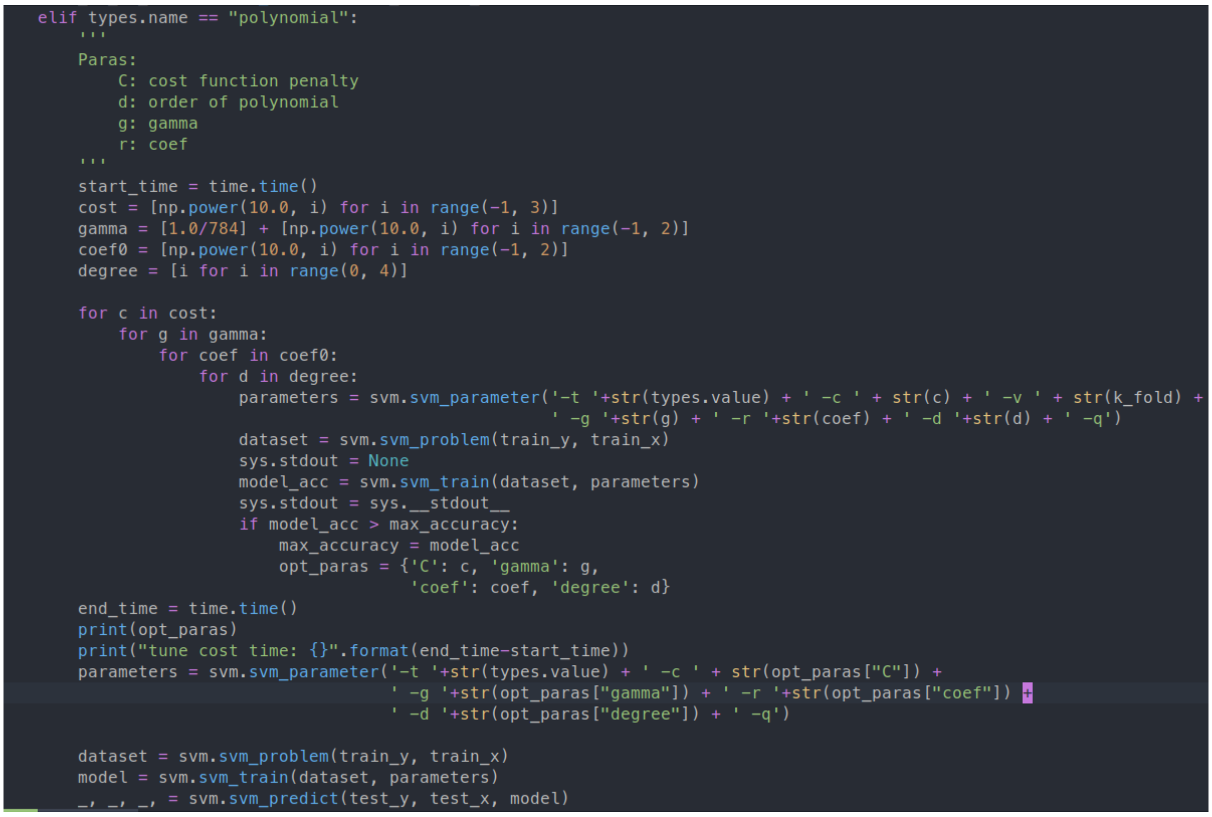
In this part, I also implemented a function for it, first we tune for linear kernel:



```python
if types.name == "linear":
    start_time = time.time()
    '''
    In linear kernel, there's no parameters, so only need to tune C for cost function in C-SVC
    '''
    for c in [0.001, 0.01, 0.1, 1, 10, 100, 1000]:
        parameters = svm.svm_parameter(
            '-t '+str(types.value) + ' -q -c ' + str(c) + ' -v ' + str(k_fold))
        dataset = svm.svm_problem(train_y, train_x)
        sys.stdout = None
        model_acc = svm.svm_train(dataset, parameters)
        sys.stdout = sys.__stdout__
        if model_acc > max_accuracy:
            max_accuracy = model_acc
            opt_paras = {'C': c}
    end_time = time.time()
    print(opt_paras)
    print("tune cost time: {}".format(end_time-start_time))
    parameters = svm.svm_parameter(
        '-t '+str(types.value) + ' -q -c ' + str(opt_paras["C"]))
    dataset = svm.svm_problem(train_y, train_x)
    model = svm.svm_train(dataset, parameters)
    _, _, _ = svm.svm_predict(test_y, test_x, model)
```

in linear kernel, there’s no parameter, so we only need to tune for parameter c, which is the parameter for C-SVC. For tuning parameter c, I provide 7 possible c values. And I set the number of folds to 3 in cross-validation.

Next, the tuning for polynomial kernel is as follows:



```python
elif types.name == "polynomial":
    '''
    Paras:
        C: cost function penalty
        d: order of polynomial
        g: gamma
        r: coef
    '''
    start_time = time.time()
    cost = [np.power(10.0, i) for i in range(-1, 3)]
    gamma = [1.0/784] + [np.power(10.0, i) for i in range(-1, 2)]
    coef0 = [np.power(10.0, i) for i in range(-1, 2)]
    degree = [i for i in range(0, 4)]

    for c in cost:
        for g in gamma:
            for coef in coef0:
                for d in degree:
                    parameters = svm.svm_parameter('-t '+str(types.value) + ' -c ' + str(c) + ' -v ' + str(k_fold) +
                                                   ' -g '+str(g) + ' -r '+str(coef) + ' -d '+str(d) + ' -q')
                    dataset = svm.svm_problem(train_y, train_x)
                    sys.stdout = None
                    model_acc = svm.svm_train(dataset, parameters)
                    sys.stdout = sys.__stdout__
                    if model_acc > max_accuracy:
                        max_accuracy = model_acc
                        opt_paras = {'C': c, 'gamma': g,
                                     'coef': coef, 'degree': d}
    end_time = time.time()
    print(opt_paras)
    print("tune cost time: {}".format(end_time-start_time))
    parameters = svm.svm_parameter('-t '+str(types.value) + ' -c ' + str(opt_paras["C"]) +
                                   ' -g '+str(opt_paras["gamma"]) + ' -r '+str(opt_paras["coef"]) +
                                   ' -d '+str(opt_paras["degree"]) + ' -q')

    dataset = svm.svm_problem(train_y, train_x)
    model = svm.svm_train(dataset, parameters)
    _, _, _ = svm.svm_predict(test_y, test_x, model)
```

in polynomial kernel, there are 3 parameters: degree of polynomial, gamma, and coef plus the c for C-SVC. The number of fold is also set to 3. Because it has more parameters, the tuning time is much longer. The results are shown in the next section. For c, gamma, and degree, I select 4 different values; 3 for coef. So the number of possible combinations is 192.

Last, as for RBF kernel:

```
elif types.name == "RBF":
    '''
    Paras:
        C: cost function penalty
        g: gamma
    '''

    start_time = time.time()
    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, 2)]

    for c in cost:
        for g in gamma:
            parameters = svm.svm_parameter(
                '-t '+str(types.value)+' -c ' + str(c)+' -v '+str(k_fold) + ' -g '+str(g) + ' -q')
            dataset = svm.svm_problem(train_y, train_x)
            sys.stdout = None
            model_acc = svm.svm_train(dataset, parameters)
            sys.stdout = sys.__stdout__
            if model_acc > max_accuracy:
                max_accuracy = model_acc
                opt_paras = {'C': c, 'gamma': g}
    end_time = time.time()
    print(opt_paras)
    print("tune cost time: {}".format(end_time-start_time))
    parameters = svm.svm_parameter(
        '-t '+str(types.value)+' -c ' + str(opt_paras["C"]) + ' -g '+str(opt_paras["gamma"]) + ' -q')
    dataset = svm.svm_problem(train_y, train_x)
    model = svm.svm_train(dataset, parameters)
    _, _, _ = svm.svm_predict(test_y, test_x, model)
```

there is a parameter, gamma, for RBF kernel plus c for C-SVC. I select 4 values for gamma, 3 for cost. And the number of fold is also 3.

Part3: linear+RBF kernel
To use user-defined, we need to set -t option to 4 and set isKernel = True when preparing problem set, the main function of performing user-defined kernel is as follows:

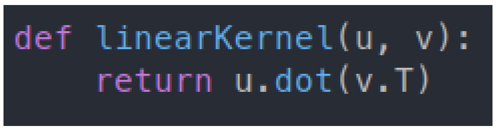
```
def SVMLinearRBF(train_x, train_y, test_x, test_y):
    print("=========part3===========")
    linear = linearKernel(train_x, train_x)
    RBF = RBFKernel(train_x, train_x)
    LinearRBF_kernel = np.hstack(
        (np.arange(1, train_x.shape[0]+1).reshape(-1, 1), linear + RBF))
    parameters = svm.svm_parameter('-t 4 -q')
    problem = svm.svm_problem(train_y, LinearRBF_kernel, isKernel=True)
    model = svm.svm_train(problem, parameters)

    linearTest = linearKernel(test_x, test_x)
    rbfTest = RBFKernel(test_x, test_x)
    new_test_x = np.hstack(
        (np.arange(1, test_x.shape[0]+1).reshape(-1, 1), linearTest+rbfTest))
    _, model_acc, _ = svm.svm_predict(test_y, new_test_x, model)

    return
```

when we use user-defined, we need to transform the data to the correct format, that is the 4th and the 12th line of the function.
The functions “linearKernel” and “RBFKernel” are to compute linear and RBF kernel respectively.

The formula of linear kernel is:

$$\text{Linear kernel}: k(x, y) =< x, y >$$

so I defined the function as this:

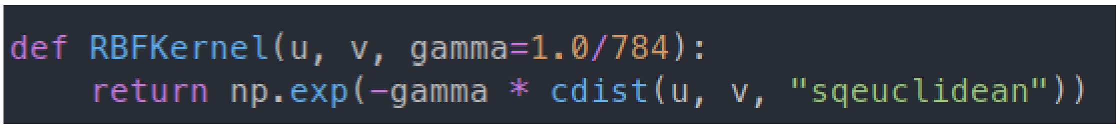


```
def linearKernel(u, v):
    return u.dot(v.T)
```

The formula of RBF kernel is:

$$K(x,x_n)=exp(\gamma\|x-x_n||^2)=RBF(x,x_n)$$

accordingly, the function is like:



```
def RBFKernel(u, v, gamma=1.0/784):
    return np.exp(-gamma * cdist(u, v, "sqeuclidean"))
```

b. Experiments settings and results
Part1: Default parameter

Kernel Type	Accuracy	Cost time(s)
Linear	95.08%	2.8115
Polynomial	34.68%	28.2943
RBF	95.32%	6.3585

Part2: Grid Search

Kernel TYpe	Accuracy	Grid Search Cost Time (s)	Best parameters	number of parameter combination
Linear	95.96%	38.8039	c: 0.01	7
Polynomial	97.72%	3106.7588	c: 0.1 gamma: 1.0 coef: 1.0 degree: 2	192
RBF	96.28%	400.867	c: 10.0 gamma: 1/784	12

Part3: Linear+RBF

Kernel Type	Accuracy	Cost time (s)
Linear + RBF	27.36%	13.5321

c. Observations and Discussions
   1. Polynomial Kernel costs much time, especially when doing the grid search, because it has more parameters to tune.
   2. After optimizing the parameters, the accuracies in part2 are all better than part1.
   3. Though Linear Kernel is simple, it performs good in part1 and part2.
   4. Parameter C controls penalty for error term, so increasing the C may make results more likely to overfit to training data.
   5. Combine linear with RBF didn’t make results better and needs more time to complete.
   6. If we choose the right parameters, polynomial can perform better than RBF. However, because RBF can map data into indefinite dimension of feature space, theoretically, it can separate data better.