I. Gaussian Process:

1. Code(20%):

First of all, we introduce the kernel function, specifically the rational quadratic kernel:

$$\mathrm{k}(x_1, x_2) = \left(1 + \frac{(x_1 - x_2)^2}{2\alpha \mathrm{L}^2}\right)^{-\alpha}$$

```
def kernel(x1, x2, alpha=1, L=1):
    return (1 + ((x1 - x2)**2 / (2 * alpha * L * L))) ** (-alpha)
```

Next, let's discuss the main function. In Task 1, we use parameter $\alpha = 1, \mathrm{L} = 1$. We input data from the 'input.data' file and store it in the 'data' array. We use the notation (x_i, y_i) to represent the i-th data, $\boldsymbol{x}$ be the 34*1 array of all x value and $\boldsymbol{y}$ be the 34*1 array of all y values in the report.

```
data = np.zeros((34,2))
with open('/content/input.data', 'r') as file:
    lines = file.readlines()
    count = 0
    for line in lines:
        data[count][0], data[count][1] = map(float, line.split())
        count += 1
```

Given a new input x^*, our goal is to find the conditional predictive distribution $p(y^*|\boldsymbol{y}, x^*)$, Using the lecture formula, we have:

$$p(y^*|y, x^*) = \mathcal{N}(k(\boldsymbol{x}, x^*)^T C^{-1} \boldsymbol{y}, k(x^*, x^*) + \beta^{-1} - k(\boldsymbol{x}, x^*)^T C^{-1} k(\boldsymbol{x}, x^*))$$

Here, *k* is the kernel function defined above, and *C* is 34*34 matrix with $C[x_i][x_j] = k(x_i, x_j) + \beta^{-1}\delta_{ij}$.

In the code, we first use the formula to compute C^{-1}. Then, we segment the range [-60,60] into 1200 points. For each point x^* we compute the mean and standard deviation of $p(y^*|y, x^*)$, and finally, we plot the graph.

```
#compute matrix C^-1
C = np.zeros((34,34))
for i in range(34):
    for j in range(34):
        if i==j :
            C[i][j] = kernel(data[i][0],data[j][0]) + (1/5)
        else:
            C[i][j] = kernel(data[i][0],data[j][0])
C_inv = np.linalg.inv(C)

#Draw a line to represent mean of f and 95% confidence interval of f
x = np.linspace(-60,60,1200)
y = np.zeros((1200))
y_2std_add = np.zeros((1200))
y_2std_min = np.zeros((1200))
for i in range(1200):
    k = np.zeros((34,1))
    for j in range(34):
        k[j][0] = kernel(data[j][0],x[i])
    mean = k.T @ C_inv @ data[:,1]
    var = kernel(x[i],x[i]) + (1/5) - k.T @ C_inv @ k

    y[i] = mean[0]
    y_2std_add[i] = mean[0] + 1.96 * math.sqrt(var[0])
    y_2std_min[i] = mean[0] - 1.96 * math.sqrt(var[0])

# Plot on the first subplot
fig, axes = plt.subplots(1, 2,figsize=(12,6))
axes[0].plot(x, y, label='mean of f',color='red')
axes[0].set_xlim(-60, 60)
axes[0].set_ylim(-5, 5)
axes[0].fill_between(x, y, y_2std_add, color='gray', alpha=0.5)
axes[0].fill_between(x, y, y_2std_min, color='gray', alpha=0.5)
axes[0].scatter(data[:,0], data[:,1],label='Training Data', color='blue', marker='o')
axes[0].set_title('initilized_parameter:\nalpha: {:.3f}, L: {:.3f}'.format(1,1))
```

In Task 2, we initially employ scipy.optimize.minimize to determine the optimal parameters:

```
#-------------------------Task 2-------------------------#
#Use scipy.optimize.minimize to find optimal parameters
result = minimize(log_likelihood_obj, [1e0, 1e0] ,bounds=((1e-4, 1e8), (1e0, 1e6)))
alpha, L = result.x #optimal value of alpha and L
```

Here, log_likelihood_obj represents the negative marginal log-likelihood, given by:

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

,where $\theta = (\alpha, \mathrm{L})$.

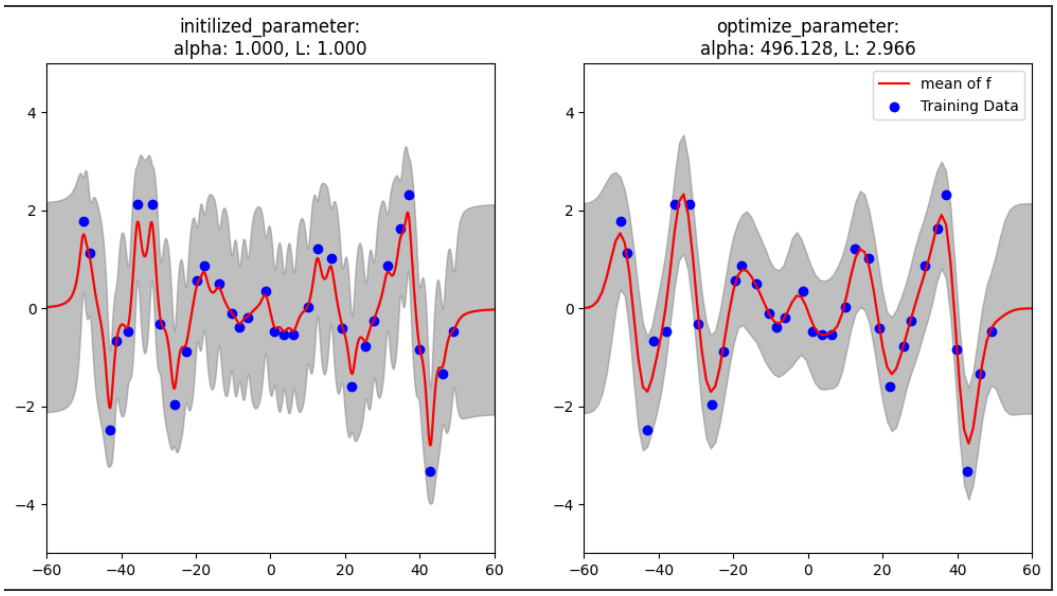
```
def log_likelihood_obj(par):
    alpha,L = par
    kernel_matrix = np.zeros((34,34))
    for i in range(34):
        for j in range(34):
            if i==j :
                kernel_matrix[i][j] = kernel(data[i][0],data[j][0], alpha, L) + (1/5)
            else:
                kernel_matrix[i][j] = kernel(data[i][0],data[j][0], alpha, L)
    return (1/2) * math.log(np.linalg.det(kernel_matrix)+1e-8) + (1/2) * data[:,1].T @ np.linalg.inv(kernel_matrix) @ data[:,1] + 17 * math.log(2*math.pi)
```

Next, we utilize the optimal parameters $\theta^* = (\alpha^*, \mathrm{L}^*)$ to compute $p(y^*|\boldsymbol{y}, x^*)$, and this process is identical to Task 1.

```
#Use optimal value of parameters to compute matrix C^-1
C = np.zeros((34,34))
for i in range(34):
    for j in range(34):
        if i==j :
            C[i][j] = kernel(data[i][0],data[j][0], alpha, L) + (1/5)
        else:
            C[i][j] = kernel(data[i][0],data[j][0], alpha, L)
C_inv = np.linalg.inv(C)

#Draw a line to represent mean of f and 95% confidence interval of f
x = np.linspace(-60,60,1200)
y = np.zeros((1200))
y_2std_add = np.zeros((1200))
y_2std_min = np.zeros((1200))

for i in range(1200):
    k = np.zeros((34,1))
    for j in range(34):
        k[j][0] = kernel(data[j][0],x[i], alpha, L)
    mean = k.T @ C_inv @ data[:,1]
    var = kernel(x[i],x[i], alpha, L) + (1/5) - k.T @ C_inv @ k

    y[i] = mean[0]
    y_2std_add[i] = mean[0] + 1.96 * math.sqrt(var[0])
    y_2std_min[i] = mean[0] - 1.96 * math.sqrt(var[0])
# Plot on the second subplot
axes[1].plot(x, y, label='mean of f',color='red')
axes[1].set_xlim(-60, 60)
axes[1].set_ylim(-5, 5)
axes[1].fill_between(x, y, y_2std_add, color='gray', alpha=0.5)
axes[1].fill_between(x, y, y_2std_min, color='gray', alpha=0.5)
axes[1].scatter(data[:,0], data[:,1],label='Training Data', color='blue', marker='o')
axes[1].set_title('optimize_parameter:\nalpha: {:.3f}, L: {:.3f}'.format(alpha,L))
plt.legend()
plt.show()
```

2. Experiments (20%):

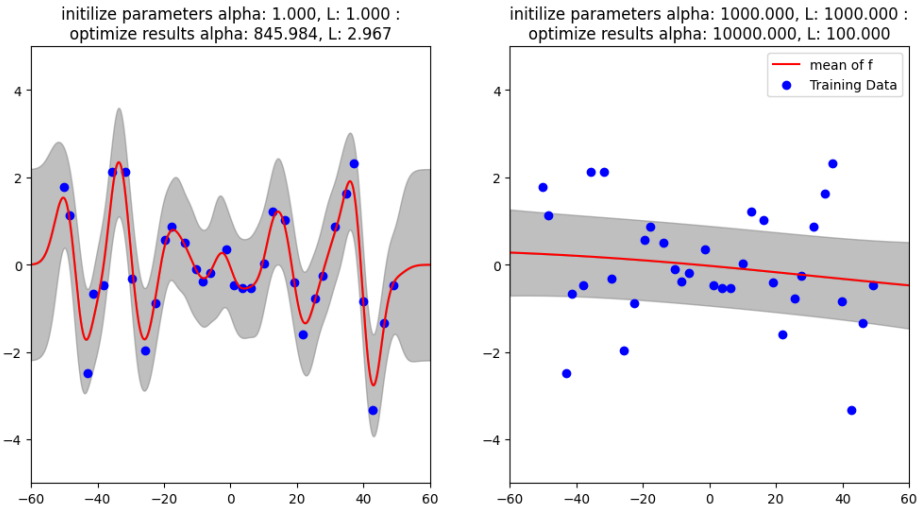
The left subplot shows the results of Task 1 with parameters $\theta = (\alpha = 1, \mathrm{L} = 1)$, while the right subplot displays the results of Task 2 with parameters $\theta = (\alpha = 496.128, \mathrm{L} = 2.966)$.In each subplot, blue dots represent the training data points, the red line represents the mean of f in range [-60,60], and the gray area represents 95% confidence interval of f.



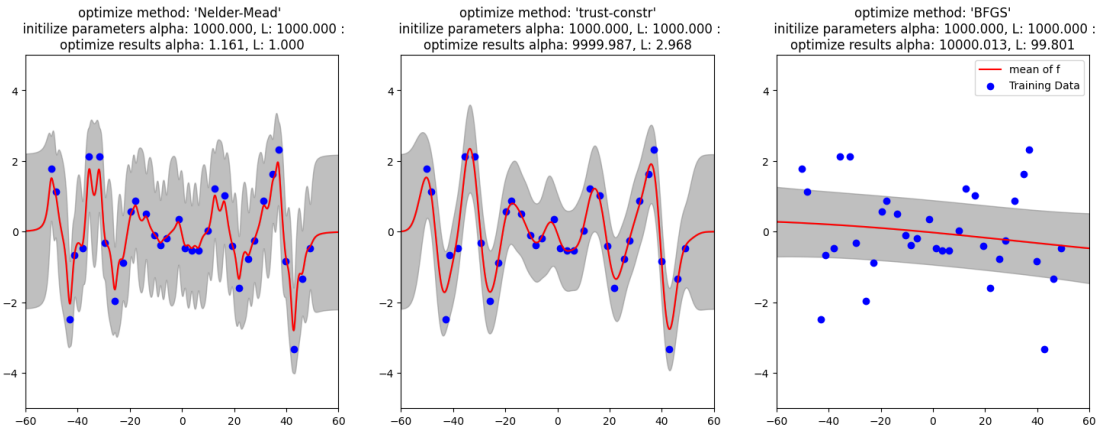
3. Observations and Discussion (10%)

We can delve deeper into the discussion on scipy.optimize.minimize.

- Using different initial guesses may yield different results, potentially leading to suboptimal outcomes in the optimization process.



- Use different optimized method to optimize may also yield different results.



II. SVM on MNIST dataset

1. Code(20%):

In Task 1, we initially read all training data, training labels, testing data, and testing labels into specify format.

```
#read training data
file = open('/content/X_train.csv', 'r')
lines = file.read()
training_x = []
for line in lines.split('\n'):
        pixels = line.split(',')
        training_x.append([(i+1):float(pixels[i]) for i in range(len(pixels)) if pixels[i].strip()])
training_x.pop()

#read training label
file = open('/content/Y_train.csv', 'r')
lines = file.read()
training_y = []
for line in lines.split('\n'):
        if line.strip():
                training_y.append(int(line))

#read testing data
file = open('/content/X_test.csv', 'r')
lines = file.read()
testing_x = []
for line in lines.split('\n'):
        pixels = line.split(',')
        testing_x.append([(i+1):float(pixels[i]) for i in range(len(pixels)) if pixels[i].strip()])
testing_x.pop()

#read testing label
file = open('/content/Y_test.csv', 'r')
lines = file.read()
testing_y = []
for line in lines.split('\n'):
        if line.strip():
                testing_y.append(int(line))
```

Next, we utilize svm_problem to generate the training data with labels. Subsequently, we employ svm_train to obtain three models with different kernel functions(-t allows us to specify which kernel function to use for training the model.). Finally, we use svm_predict to predict the testing data with each model and evaluate the prediction accuracy.

```
#generate the training data with labels
training = svm_problem(training_y, training_x)

#training svm on three different kernel function( 0:linear, 1: polynomial, 2: RBF )
linear_model = svm_train(training, svm_parameter('-q -t 0'))
poly_model = svm_train(training, svm_parameter('-q -t 1'))
RBF_model = svm_train(training, svm_parameter('-q -t 2'))

#prdict testing data, output is accuracy
print("predict with linear kernel model: ")
linear_label, linear_acc, linear_vals = svm_predict(testing_y, testing_x, linear_model)
print("\npredict with polynomial kernel model: ")
poly_label, poly_acc, poly_vals = svm_predict(testing_y, testing_x, poly_model)
print("\npredict with RBF kernel model: ")
RBF_label, RBF_acc, RBF_vals = svm_predict(testing_y, testing_x, RBF_model)
```

In Task 2, we initially set the kernel as RBF and perform a grid search on the parameters –c(C-SVC parameter) and –g(gamma parameter) using the values$[e^{-8}, e^{-6}, e^{-4}, e^{-2}, e^{0}]$, aiming to identify the parameter combination that yields the highest training performance.

```
#grid search for RBF kernel
best_acc_RBF = 0
best_c_RBF = 0
best_g_RBF = 0
valueChoice_lgc = [-8, -6, -4, -2, 0]
valueChoice_lgg = [-8, -6, -4, -2, 0]
results_RBF = np.zeros((5,5))
for i, lgc_RBF in enumerate(valueChoice_lgc):
    for j, lgg_RBF in enumerate(valueChoice_lgg):
        parameter = svm_parameter(f'-q -v 3 -t 2 -c {math.e**lgc_RBF} -g {math.e**lgg_RBF}')
        model = svm_train(training, parameter)
        results_RBF[i][j] = round(model, 2)
        if results_RBF[i][j] > best_acc_RBF:
            best_acc_RBF = results_RBF[i][j]
            best_c_RBF = lgc_RBF
            best_g_RBF = lgg_RBF
```

Then, we use the best training performance parameters with RBF kernel to test the testing data.

```
best_parameter = svm_parameter(f'-q -t 2 -c {math.e**best_c_RBF} -g {math.e**best_g_RBF}')
model = svm_train(training, best_parameter)
_, test_RBF, _ = svm_predict(testing_y, testing_x, model)
```

Next, we set the kernel as polynomial and perform a grid search on the parameters –d(the degree of polynomial) using the value $[0,2,4,6,8,10]$ and -c using the values$[e^{-8}, e^{-6}, e^{-4}, e^{-2}, e^{0}]$, aiming to identify the parameter combination that yields the highest training performance.

```
#grid search for polynomial kernel
best_acc_poly = 0
best_d_poly = 0
best_c_poly = 0
valueChoice_d = [0, 2, 4, 6, 8 ,10]
valueChoice_lgc = [-8, -6, -4, -2, 0]
results_poly = np.zeros((6,5))

for i, d in enumerate(valueChoice_d):
    for j, lgc in enumerate(valueChoice_lgc):
        parameter = svm_parameter(f'-q -v 3 -t 1 -d {d} -c {math.e**lgc}')
        model = svm_train(training, parameter)
        results_poly[i][j] = round(model, 2)
        if results_poly[i][j] > best_acc_poly:
            best_acc_poly = results_poly[i][j]
            best_d_poly = d
            best_c_poly = lgc
```

Then, we use the best training performance parameters with polynomial kernel to test the testing data.

```
best_parameter = svm_parameter(f'-q -t 1 -d {best_d_poly} -c {math.e**best_c_poly}')
model = svm_train(training, best_parameter)
_, test_poly, _ = svm_predict(testing_y, testing_x, model)
```

Lastly, we set the kernel as linear and perform a grid search on the parameters -c using the values$[e^{-10}, e^{-8}, e^{-6}, e^{-4}, e^{-2}, e^{0}, e^{2}, e^{4}, e^{6}, e^{8}, e^{10}]$, aiming to identify the parameter combination that yields the highest training performance.

```
#grid search for linear kernel
best_acc_linear = 0
best_c_linear = 0
valueChoice_lgc_L = [-10, -8, -6, -4, -2, 0, 2, 4, 6, 8, 10]
results_linear = np.zeros((11))

for i, lgc in enumerate(valueChoice_lgc):
        parameter = svm_parameter(f'-q -v 3 -t 0 -c {math.e**lgc}')
        model = svm_train(training, parameter)
        results_linear[i] = round(model, 2)
        if results_linear[i] > best_acc_linear:
                best_acc_linear = results_linear[i]
                best_c_linear = lgc
```

Then, we use the best training performance parameters with linear kernel to test the testing data.

```
best_parameter = svm_parameter(f'-q -t 0 -c {math.e**best_c_linear}')
model = svm_train(training, best_parameter)
_, test_linear, _ = svm_predict(testing_y, testing_x, model)
```

For the results, we save the grid search results into an Excel file for easy reading. Subsequently, we print the results of three different kernel functions with their respective best parameters.

```
#save results to excel
#RBF results
row_labels = ["c: e**" + str(i) for i in valueChoice_lgc]
column_labels = ["g: e**" + str(i) for i in valueChoice_lgg]
df = pd.DataFrame(results_RBF, index=row_labels, columns=column_labels)
excel_file_path = "RBF_grid_results.xlsx"
df.to_excel(excel_file_path, index=True)

#polynomial results
row_labels = ["d: " + str(i) for i in valueChoice_d]
column_labels = ["c: e**" + str(i) for i in valueChoice_lgc]
df = pd.DataFrame(results_poly, index=row_labels, columns=column_labels)
excel_file_path = "polynomail_grid_results.xlsx"
df.to_excel(excel_file_path, index=True)

#linear results
row_labels = ["c: e**" + str(i) for i in valueChoice_lgc_L]
df = pd.DataFrame(results_linear, index=row_labels)
excel_file_path = "linear_grid_results.xlsx"
df.to_excel(excel_file_path, index=True)
```

```
#print results
print("-------------------------------------------------------------------------")
print("The best parameter for RBF kernel:")
print(f"c: e**{best_c_RBF}")
print(f"g: e**{best_g_RBF}")
print(f"model training performance: {best_acc_RBF}")
print(f"model testing performance: {test_RBF[0]}")
print("-------------------------------------------------------------------------")
print("The best parameter for Polynomial kernel:")
print(f"d: {best_d_poly}")
print(f"g: e**{best_c_poly}")
print(f"model training performance: {best_acc_poly}")
print(f"model testing performance: {test_poly[0]}")
print("-------------------------------------------------------------------------")
print("The best parameter for Linear kernel:")
print(f"c: e**{best_c_linear}")
print(f"model training performance: {best_acc_linear}")
print(f"model testing performance: {test_linear[0]}")
print("-------------------------------------------------------------------------")
```

In Task_3, we first define the new_kernel which is linear combination of linear kernel and RBF kernel using different weight and different gamma value in RBF.

```
def new_kernel(x1,x2,w,g):
        data = np.zeros((len(x1), len(x2) + 1))
        linear = x1 @ x2.T
        RBF = np.exp(- math.e**g * cdist(x1, x2, 'sqeuclidean'))
        data[:, 1:] = w* linear + (1-w) * RBF
        data[:, :1] = np.arange(len(x1))[:, np.newaxis]+1
        return data
```

In the main function, we perform the grid search on the weight(value range [0.0, 0.2, 0.4, 0.6, 0.8, 1.0]) and gamma (value range: $[e^{-4}, e^{-2}, e^{0}, e^{2}, e^{4}]$), aiming to identify the parameter combination that yields the highest training performance.

```
wChoice = [0.0, 0.2, 0.4, 0.6, 0.8, 1.0]
valueChoice_lgg = [-4, -2, 0, 2, 4]
best_acc_con = 0
best_w_con = 0
best_g_con = 0
result_con = np.zeros((6,5))
for i, w in enumerate(wChoice):
    for j, lgg in enumerate(valueChoice_lgg):
        data = new_kernel(np.array([list(data.values()) for data in training_x]), np.array([list(data.values()) for data in training_x]), w, lgg)
        param = f"-q -t 4 -v 3"
        model = svm_train(training_y, [list(row) for row in data], param)
        result_con[i][j] = round(model, 2)
        if result_con[i][j] > best_acc_con:
            best_acc_con = result_con[i][j]
            best_w_con = w
            best_g_con = lgg
```

After grid search, we use best parameter of grid search to test the testing data.

```
data = new_kernel(np.array([list(data.values()) for data in training_x]), np.array([list(data.values()) for data in training_x]), best_w_con, best_g_con)
data_t = new_kernel(np.array([list(data.values()) for data in testing_x]), np.array([list(data.values()) for data in training_x]), best_w_con, best_g_con)
param = f"-q -t 4"
model = svm_train(training_y, [list(row) for row in data], param)
_, test_con, _ = svm_predict(testing_y, [list(row) for row in data_t], model)
```

Lastly, we save the grid search results into an Excel file for easy reading.

```
#combination results
row_labels = ["w: " + str(i) for i in wChoice]
column_labels = ["g: e**" + str(i) for i in valueChoice_lgg]
df = pd.DataFrame(result_con, index=row_labels, columns=column_labels)
excel_file_path = "combination_grid_results.xlsx"
df.to_excel(excel_file_path, index=True)
```

Subsequently, we print the results of best parameters.

```
#print resutls
print("-------------------------------------------------------------------------------")
print("The best parameter for combination kernel:")
print(f"w: {best_w_con}")
print(f"g: e**{best_g_con}")
print(f"model training performance: {best_acc_con}")
print(f"model testing performance: {test_con[0]}")
```

2. Experiments (20%):

- Experiment for Task 1 (6%)

```
predict with linear kernel model:
Accuracy = 95.08% (2377/2500) (classification)

predict with polynomial kernel model:
Accuracy = 34.68% (867/2500) (classification)

predict with RBF kernel model:
Accuracy = 95.32% (2383/2500) (classification)
```

- Experiment for Task 2 (8%)

1. The grid search results(training performance) with the RBF kernel function.

	g: e**-8	g: e**-6	g: e**-4	g: e**-2	g: e**0
c: e**-8	79.86	83.08	92.98	39.12	20.82
c: e**-6	80.2	82.94	92.76	40.32	20.54
c: e**-4	80.08	87.26	94.22	40.52	20.48
c: e**-2	87.74	94.82	96.88	45.18	20.84
c: e**0	95	96.68	98.14	81.34	29.48

2. The grid search results(training performance) with the polynomial kernel function.

	c: e**-8	c: e**-6	c: e**-4	c: e**-2	c: e**0
d: 0	20	20	20	20	20
d: 2	45.98	45.9	45.8	47.52	84.08
d: 4	23.76	23.74	23.76	23.92	23.58
d: 6	21.38	21.44	21.62	21.56	21.6
d: 8	20.78	20.72	20.72	20.76	20.72
d: 10	20.48	20.52	20.5	20.48	20.62

3. The grid search results(training performance) with the linear kernel function.

	0
c: e**-10	80.04
c: e**-8	93.48
c: e**-6	96.12
c: e**-4	96.96
c: e**-2	96.84
c: e**0	96
c: e**2	96.1
c: e**4	96.12
c: e**6	96.14
c: e**8	96.24
c: e**10	96.06

4. The results include the values and training/testing performances of the models with the best parameters for each kernel function.

```
The best parameter for RBF kernel:
c: e**0
g: e**-4
model training performance: 98.14
model testing performance: 98.24000000000001

The best parameter for Polynomial kernel:
d: 2
g: e**0
model training performance: 84.08
model testing performance: 88.24

The best parameter for Linear kernel:
c: e**-4
model training performance: 96.96
model testing performance: 95.96000000000001
```

- Experiment for Task 3 (6%)

  1. The grid search results of a linear combination of the linear kernel and RBF kernel with varying weights and gamma values.

	g: e**-4	g: e**-2	g: e**0	g: e**2	g: e**4
w: 0.0	98.16	81.04	30.02	20.5	20.98
w: 0.2	96.94	96.44	96.86	96.56	96.62
w: 0.4	96.4	96.34	96.38	96.56	96.12
w: 0.6	96.26	96.14	96.54	95.92	96.52
w: 0.8	96.6	96.48	96.48	96.54	96.54
w: 1.0	96.12	96.5	95.96	95.78	96.18

  2. The results include the values and training/testing performances of the models with the best parameters.

```
The best parameter for combination kernel:
w: 0.0
g: e**-4
model training performance: 98.16
model testing performance: 98.24000000000001
```

3. Observations and Discussion (10%)

   1. The polynomial kernel requires more time for both training and testing, as observed in Task 2.

   2. In Task 3, we can observe that the linear combination does not improve the performance because the best weight (w) is 0.0.

   3. The polynomial kernel has lower performance compared to other kernels, even though we have used grid search to find better parameters.