ML_HW5

312657008_ 江祐姍

In this homework, I used Python and imported the following packages: numpy, matplotlib.pyplot, scipy.optimize.minimize, libsvm.svmutil, pandas, and scipy.spatial.distance.cdist.

I. Gaussian Process

Load data

```
df = open(r"C:\Users\youshu\Desktop\機器學習\HW5\data\input.data","r")
file_path = r"C:\Users\youshu\Desktop\機器學習\HW5\data\input.data"
x = []
y = []
with open(file_path, "r") as df:
    lines = df.readlines()
points = []
for line in lines:
    values = list(map(float, line.split()))
    for i in range(0, len(values), 2):
        x_point, y_point = values[i], values[i+1]
        x.append(x_point)
        y.append(y_point)
```

1. Code

- Code for Task 1

The rational quadratic kernel is $k_{RQ}(x, x') = \sigma^2(1 + \frac{(x-x')^2}{2\alpha l^2})^{-\alpha}$

where

σ^2 is the signal scale

α is the noise variance

l is the length scale, controls similarity

The following code is the kernel mentioned above.

```
def kernel(x1, x2, sigma = 1, alpha = 1, l = 1):
    return sigma**2 * (1 + np.power(x1 - x2, 2)/(2 * alpha * l**2)) ** (-alpha)
```

Using the kernel mentioned above, we can apply the Gaussian function to obtain the covariance matrix for the training data. The formula is as follows:

$C(x_n, x_m) = k(x_n, x_m) + \beta^{-1}\delta_{mn}$

where

$$\delta_{mn} = \begin{cases} 1, & \text{if } m = n, \\ 0, & \text{otherwise.} \end{cases}$$

```
def Gaussian(x, beta, sigma, alpha , l):
    k = kernel(x, np.transpose(x), sigma, alpha, l)
    c = k + (1 / beta) * np.eye(len(x))
    return c
```

Next, we can use the model to predict the new data point x^*

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

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

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

```
def predict(x, y, c, beta, sigma, alpha, l):
    x_star = np.linspace(-60, 60, 1000)
    pred_mean = np.zeros(1000)
    pred_var = np.zeros(1000)
    c_inv = np.linalg.inv(c)
    for i in range(1000):
        k_x_xs = kernel(x, [x_star[i]], sigma, alpha, l)
        pred_mean[i] = np.transpose(k_x_xs) @ c_inv @ y

        k_xs_xs = kernel([x_star[i]], [x_star[i]], sigma, alpha, l) + (1/beta)
        pred_var[i] = np.abs(k_xs_xs - np.transpose(k_x_xs) @ c_inv @ k_x_xs)

    return x_star, pred_mean, pred_var
```

- Code for Task 2

In this task, the Gaussian process, kernel, and prediction functions remain same as Task 1.

The marginal likelihood is $L = \frac{1}{2} log(det(C)) + y^T C^{-1} y + N * log(2\pi)$

```
def marginal_likelihood(theta, x, y, beta):
    theta = theta.ravel()
    c = Gaussian(x, beta, theta[0], theta[1], theta[2])
    L = 0.5 * (np.log(np.linalg.det(c)) + np.transpose(y) @ np.linalg.inv(c) @ y
        + np.log(2*np.pi)* len(x))
    return L.ravel()
```

σ^2 (Signal scale): This is usually related to the variability of the data. The range of sigma is typically set between (1e-6, 1e6), which can cover both small and large variations.

α (Noise variance): alpha is the noise parameter in Gaussian processes, and its range is generally set between (1e-6, 1e2). A too small alpha may lead to overfitting, while a too large alpha might cause the model to ignore changes in the data.

l (Length scale, controls similarity): l controls the similarity between input points in the Gaussian process. Its range is usually set between (1e-6, 1e2), and this range should be adjusted based on the characteristics of the problem.

To optimize the parameters σ, α, l , I employed scipy.optimize.minimize to determine the values that minimize the marginal likelihood function described above.

```
optimize = minimize(marginal_likelihood, [sigma, alpha, l],
                    bounds = ((1e-6, 1e6),  # the range of sigma
                              (1e-6, 1e2),  # the range of alpha
                              (1e-6, 1e2)),  # the range of l
                     args=(x, y, beta))
sigma_hat = optimize.x[0]
alpha_hat = optimize.x[1]
l_hat = optimize.x[2]
```

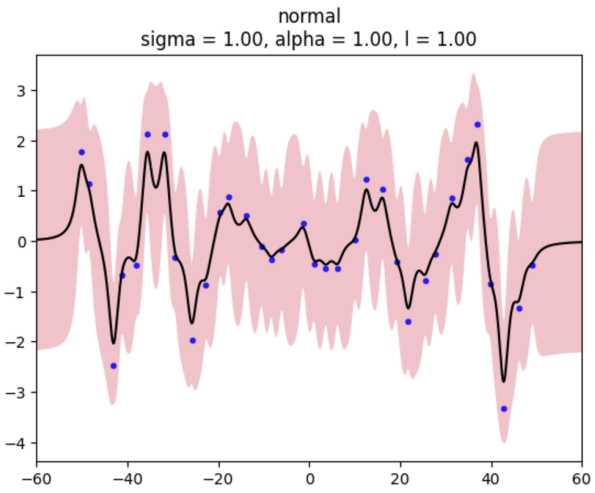
2. Experiments

- Experiment for Task 1

Using the steps outlined above, we can obtain the predicted mean and variance. These can then be used for plotting. Note that when plotting, we should treat the variance as the standard deviation, which means we need to take the square root of the variance. Additionally, since we require a 95% confidence interval, we multiply the standard deviation by 2.

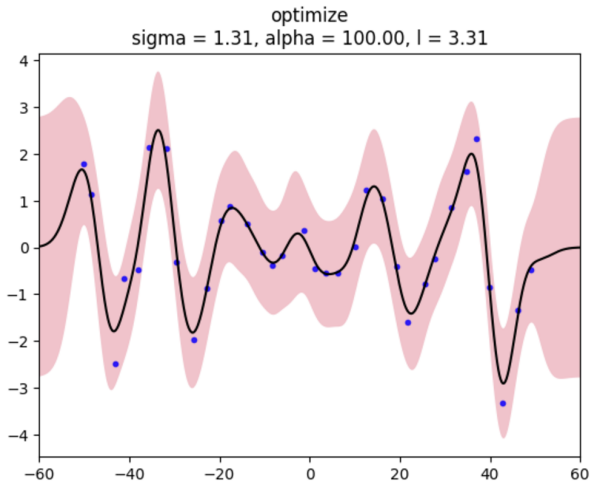
```
sigma = 1
alpha = 1
l = 1
beta = 5
c = Gaussian(x, beta, sigma, alpha , l)

x_star, pred_mean, pred_var = predict(x, y, c, beta, sigma, alpha, l)
plt.plot(x, y, 'bo',markersize=3)
plt.plot(x_star, pred_mean, 'k-')
plt.fill_between(x_star, pred_mean + 2 * (pred_var)**(1/2), pred_mean - 2 *
                 (pred_var)**(1/2), facecolor = 'pink')
plt.xlim(-60, 60)
plt.title(f"normal\nsigma = {sigma:.2f}, alpha = {alpha:.2f}, l = {l:.2f}")
plt.show()
```



- Experiment for Task 2

```
sigma = 1
alpha = 1
l = 1
beta = 5
c_hat = Gaussian(x, beta, sigma_hat, alpha_hat, l_hat)
x_star, pred_mean, pred_var = predict(x, y, c_hat, beta, sigma_hat, alpha_hat, l_hat)
plt.plot(x, y, 'bo',markersize=3)
plt.plot(x_star, pred_mean, 'k-')
plt.fill_between(x_star, pred_mean + 2 * (pred_var)**(1/2), pred_mean - 2 * (pred_var)**(1/2), facecolo
plt.xlim(-60, 60)
plt.title(f"optimize\nsigma = {sigma_hat:.2f}, alpha = {alpha_hat:.2f}, l = {l_hat:.2f}")
plt.show()
```



3.Observations and Discussion

By applying optimization techniques, the curve becomes more stable and exhibits a better overall fit to the data, with the 95% confidence interval narrowing in most areas. This enhancement helps to avoid overfitting by preventing the curve from excessively conforming to noise or outliers in the data. While the variance decreases in regions where the data is dense and well-represented, there is an observable increase in variance near the decision boundaries. This increase may be attributed to a lack of sufficient training samples in these boundary areas, leading to greater uncertainty and less precise predictions. The optimized model strikes a balance between capturing the complexity of the data and maintaining generalization, but in regions with fewer training instances, the model's confidence naturally decreases.

II. SVM on MNIST

Load data

```
train_image_path =  r"C:\Users\youshu\Desktop\機器學習\HW5\data\X_train.csv"
train_label_path = r"C:\Users\youshu\Desktop\機器學習\HW5\data\Y_train.csv"
test_image_path = r"C:\Users\youshu\Desktop\機器學習\HW5\data\X_test.csv"
test_label_path = r"C:\Users\youshu\Desktop\機器學習\HW5\data\Y_test.csv"
train_label = np.array(pd.read_csv(train_label_path, header=None)).ravel().tolist()
test_label = np.array(pd.read_csv(test_label_path, header=None)).ravel().tolist()
```

```
train_image = []
df = pd.read_csv(train_image_path, header=None)
for i in range(len(df)):
    train_image.append(df.iloc[i].tolist())

test_image = []
df = pd.read_csv(test_image_path, header=None)
for i in range(len(df)):
    test_image.append(df.iloc[i].tolist())
```

1. Code

- Code for Task 1

Generate the training data with labels: Use svm_problem to create the training dataset, which consists of feature vectors and their corresponding labels. The labels should indicate the class each sample belongs to.

Set training parameters: Use svm_parameter to configure the SVM's training parameters. The -q option suppresses any output messages during the training process, ensuring a quieter execution. The -t option specifies the kernel type used in the SVM model. The three available kernel types are:

0: Linear kernel, 1: Polynomial kernel, 2: Radial Basis Function (RBF) kernel. By default, the SVM configuration uses the -c 0 argument, which refers to C-SVC (C-Support Vector Classification). This is a standard SVM used for multi-class classification. The default setup does not include any penalty (or regularization) parameter, which may be adjusted if needed.

Train the SVM model: Use svm_train to train the SVM model using the training data for each kernel type (linear, polynomial, or RBF). For each kernel, you will obtain a corresponding trained model. Since there are four classes in the classification task, a one-class SVM (which is designed for binary classification tasks) will not be sufficient, and thus the multi-class classification capabilities of C-SVC will be used to handle this task effectively.

```
train = svm_problem(train_label, train_image)

# linear
linear_parm = svm_parameter('-q -t 0')
linear_model = svm_train(train, linear_parm)
# testing
print("linear kernel:")
svm_predict(test_label, test_image, linear_model)
print("\n")

#polynomial
polynomail_parm = svm_parameter('-q -t 1')
polynomial_model = svm_train(train, polynomail_parm)
# testing
print("polynomial kernel:")
svm_predict(test_label, test_image, polynomial_model)
print("\n")

#RBF
RBF_parm = svm_parameter('-q -t 2')
RBF_model = svm_train(train, RBF_parm)
# testing
```

```
print("RBF kernel:")
svm_predict(test_label, test_image, RBF_model)
```

- Code for Task 2

The below code implements a grid search for hyperparameter optimization in a Support Vector Machine (SVM) classifier. The main goal is to find the best combination of hyperparameters, specifically C(the regularization parameter) and γ(the kernel parameter), for different kernel types. The performance of these parameter combinations is evaluated using cross-validation accuracy (AUC), with 3-fold cross-validation employed, which is generally sufficient for this task. Once the best parameters are identified, the model is trained with these optimal hyperparameters and used for prediction on the test set.

Explanation of the Impact of C and γ

C(Regularization Parameter):

The parameter Ccontrols the trade-off between achieving a low error on the training set and maintaining a simpler decision boundary (i.e., avoiding overfitting).

A larger value of C gives the SVM a higher penalty for misclassifying training points, leading to a decision boundary that fits the training data more closely. However, this can result in overfitting, especially if the model becomes too sensitive to noise in the data.

A smaller value of C allows for a simpler decision boundary, possibly underfitting the data, but it may perform better on unseen data by generalizing more effectively.

γ(Kernel Parameter):

The parameter γ defines the influence of a single training example in the decision boundary. Specifically, it controls the curvature of the decision boundary in the non-linear kernel cases (e.g., Polynomial, RBF).

A larger γresults in a more complex model with a more flexible decision boundary, making it sensitive to individual training examples. This can lead to overfitting if the boundary becomes too detailed and starts capturing noise.

A smaller γ leads to a simpler decision boundary, which might underfit the data by not capturing enough complexity, but it is generally better for generalizing on unseen data.

In summary, finding the optimal values for C and γ is crucial for balancing the trade-off between bias and variance, ensuring that the model generalizes well while still fitting the data effectively.

```
def svm_grid_search(train_label, train_image, test_label, test_image):
    train = svm_problem(train_label, train_image)
    log2c = [i for i in range(-6, 7)]
    log2g = [i for i in range(-6, 7)]
    kernel = int(input("Please select the kernel type by entering the
             corresponding number: (0: Linear, 1: Polynomial, 2: RBF) "))
    best = 0.0
    best_log2c = 0
    best_log2g = 0

    if kernel == 0:  # Linear kernel
        auc = np.zeros(len(log2c))
        for j in range(len(log2c)):
            parm = f"-q -t {kernel} -v 3 -c {2**log2c[j]}"
            model = svm_train(train, parm)
            auc[j] = model
            if best < model:
```

```
                best = model
                best_log2c = log2c[j]


        plt.figure(figsize=(8, 6))
        plt.plot(log2c, auc, marker='o')
        plt.xlabel('log2(C)')
        plt.ylabel('Cross Validation Accuracy (%)')
        plt.title("Grid Search (Linear Kernel)")
        plt.grid(True)
        for j in range(len(log2c)):
            plt.text(log2c[j], auc[j], f"{auc[j]:.2f}", ha='center', va='bottom', color='black')
        plt.show()
    else:  # Non-linear kernels (e.g., Polynomial, RBF)
        auc = np.zeros((len(log2g), len(log2c)))
        for i in range(len(log2g)):
            for j in range(len(log2c)):
                parm = f"-q -t {kernel} -v 3 -c {2**log2c[j]} -g {2**log2g[i]}"
                model = svm_train(train, parm)
                auc[i][j] = model
                if best < model:
                    best = model
                    best_log2c = log2c[j]
                    best_log2g = log2g[i]

        plt.figure(figsize=(12, 12))
        plt.imshow(auc, interpolation='nearest', cmap='Blues', origin='lower')
        plt.colorbar(label='Cross Validation Accuracy (%)')
        plt.xticks(range(len(log2c)), [f"{c}" for c in log2c])
        plt.yticks(range(len(log2g)), [f"{g}" for g in log2g])
        plt.xlabel('log2(C)')
        plt.ylabel('log2(Gamma)')
        plt.title(f"Grid Search Heatmap (Kernel={kernel})")
        for i in range(len(log2g)):
            for j in range(len(log2c)):
                plt.text(j, i, f"{auc[i, j]:.2f}", ha='center', va='center',
                         color='pink')
        plt.show()

    parm = f"-q -t {kernel} -c {2**best_log2c}"
    if kernel != 0:
        parm += f" -g {2**best_log2g}"
    model = svm_train(train, parm)
    _, p_auc, _ = svm_predict(test_label, test_image, model)

    return best, best_log2c, best_log2g if kernel != 0 else None, auc, p_auc[0]
```

- Code for Task 3

Function new_kernel(x1, x2, gamma):

Purpose: This function computes a custom kernel that combines a linear kernel and a Radial Basis Function (RBF) kernel.

Linear kernel: The linear kernel is computed as the dot product between the vectors x1 and x2

RBF kernel: The RBF kernel is computed using the formula exp(-gamma * cdist(x1, x2, 'sqeuclidean'))

where cdist computes the squared Euclidean distance between vectors in x1 and x2.

Combining the kernels: The final kernel is a combination of the linear kernel and the RBF kernel. The kernel is stored in a matrix, where the first column is filled with values from 1 to n, and the remaining columns are filled with the sum of the linear and RBF kernels. grid Search for best parameters: The code defines two ranges for the parameters c and γ which are same methods in task2. After identifying the optimal parameters, the model is trained and predictions are made using these parameters.

Prediction: The trained model is then used to predict the labels for the test data, and the prediction accuracy is computed using svm_predict.

note. the little differents in this task, that is kernel number needs to modify to 4 -t 4: This specifies the kernel type for the SVM.

By using -t 4, the code tells the SVM to apply this custom kernel rather than the built-in kernels. The combination of linear and RBF kernels allows the model to capture both linear and non-linear patterns in the data, potentially leading to better performance for more complex datasets.

```
def new_kernel(x1, x2, gamma):
    kernel_linear = x1 @ x2.T
    kernel_RBF = np.exp(-gamma * cdist(x1, x2, 'sqeuclidean'))
    kernel = np.zeros((len(x1), len(x2) + 1))
    kernel[:, 1:] = kernel_linear + kernel_RBF
    kernel[:, :1] = np.arange(len(x1))[:, np.newaxis] + 1
    return kernel


log2c = [i for i in range(-6, 7)]
log2g = [i for i in range(-6, 7)]
best = 0.0
best_log2c = 0
best_log2g = 0

auc = np.zeros((len(log2g), len(log2c)))

for i in range(len(log2g)):
    for j in range(len(log2c)):
        parm = f"-q -t 4 -v 3 -c {2**log2c[j]} -g {2**log2g[i]}"
        kernel = new_kernel(np.array(train_image), np.array(train_image), 2**log2g[i])
        model = svm_train(train_label, [list(row) for row in kernel], parm)
        auc[i][j] = model
        if (best < model):
                best = model
                best_log2g = log2g[i]
                best_log2c = log2c[j]

plt.figure(figsize=(12, 12))
plt.imshow(auc, interpolation='nearest', cmap='Blues', origin='lower')
plt.colorbar(label='Cross Validation Accuracy (%)')
plt.xticks(range(len(log2c)), [f"{c}" for c in log2c])
plt.yticks(range(len(log2g)), [f"{g}" for g in log2g])
plt.xlabel('log2(C)')
plt.ylabel('log2(Gamma)')
```

```
plt.title(f"Grid Search Heatmap")
for i in range(len(log2g)):
    for j in range(len(log2c)):
        plt.text(j, i, f"{auc[i, j]:.2f}", ha='center', va='center', color='black')
plt.show()


parm = f"-q -t 4 -c {2**best_log2c} -g {2**best_log2g}"
best_train_kernel = new_kernel(np.array(train_image), np.array(train_image), 2**best_log2g)
model = svm_train(train_label, [list(row) for row in best_train_kernel], parm)


test_kernel = new_kernel(np.array(test_image), np.array(train_image), 2**best_log2g)
_, p_acc, _ = svm_predict(test_label, [list(row) for row in test_kernel], model)

print(p_acc)
```

2. Experiments

- Experiment for Task 1

linear kernel: Accuracy = 95.08% (2377/2500) (classification)

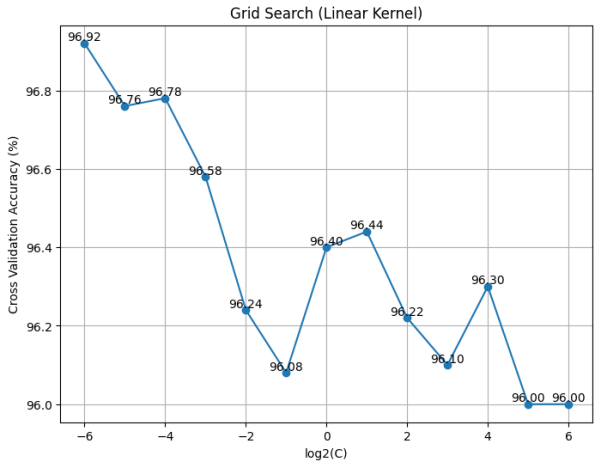
polynomial kernel: Accuracy = 34.68% (867/2500) (classification)

RBF kernel: Accuracy = 95.32% (2383/2500) (classification)

- Experiment for Task 2

```
best, best_log2c, best_log2g, auc, p_auc = svm_grid_search(train_label, train_image, test_label, test_i
```

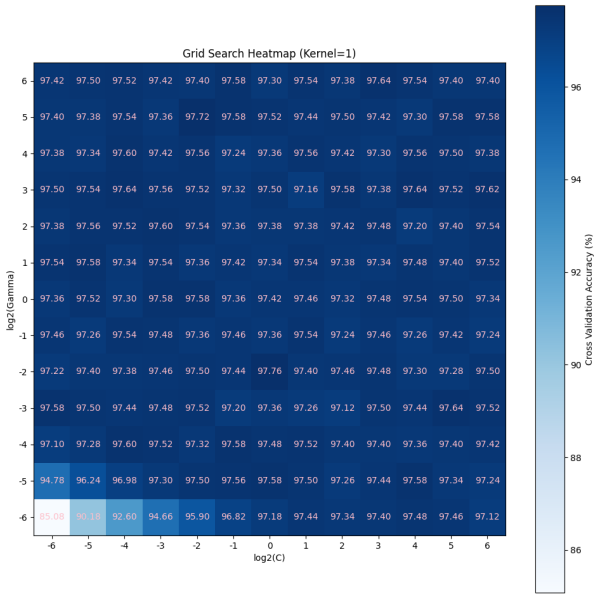
Select the kernel number (0 for linear), and view the accuracy on the line graph below.



Select the optimal parameter, then obtain the prediction results using svm_predict.

```
Accuracy = 95.92% (2398/2500) (classification)
```

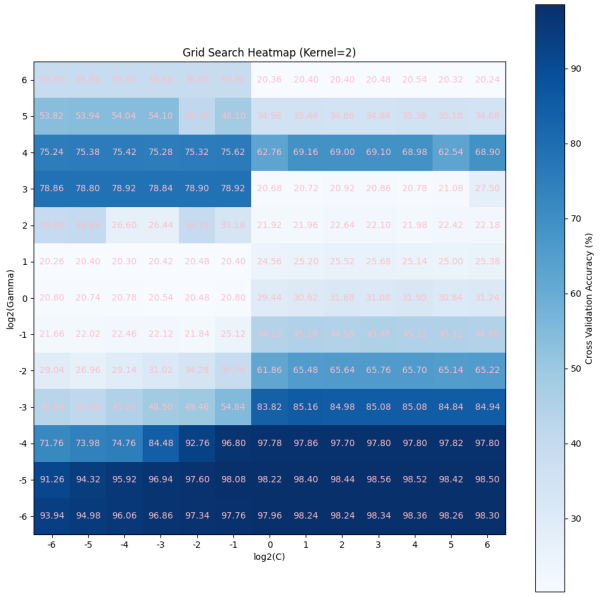
Select the kernel number (1 for polynomial), and view the accuracy on the heatmap below.



Select the optimal parameter, then obtain the prediction results using svm_predict.

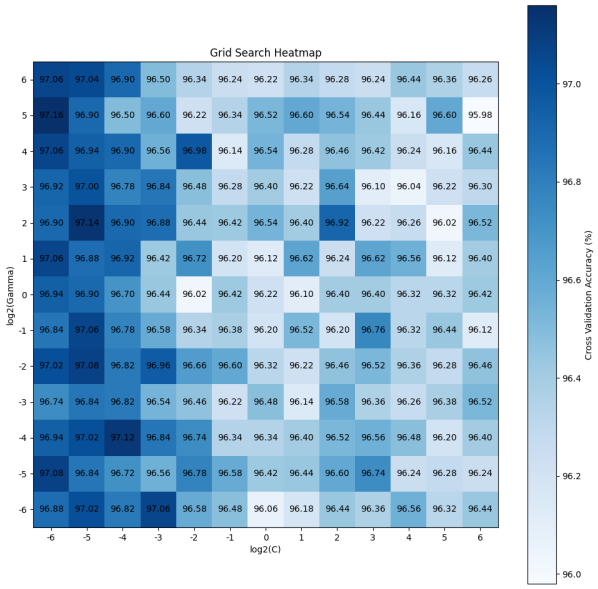
```
Accuracy = 97.48% (2437/2500) (classification)
```

Select the kernel number (2 for RBF), and view the accuracy on the heatmap below.




```
Accuracy = 98.52% (2463/2500) (classification)
```

- Experiment for Task 3



Select the optimal parameter, then obtain the prediction results using svm_predict.

```
Accuracy = 95.96% (2399/2500) (classification)
```

3.Observations and Discussion

- for Task 1

Linear Kernel (95.08%): The linear kernel performed quite well with an accuracy of 95.08%. This suggests that the decision boundary for this dataset is approximately linear, or that the linear kernel was able to effectively separate the data despite its simplicity. It's worth noting that linear kernels often work well when the dataset is not overly complex or when the classes are almost linearly separable.

Polynomial Kernel (34.68%): The polynomial kernel, however, showed a significant drop in accuracy at just 34.68%. This is a much poorer result compared to the linear and RBF kernels. The polynomial kernel has a higher risk of overfitting, especially when the degree of the polynomial is high, which may be the case here. Overfitting occurs when the model becomes too complex, fitting the noise in the training data rather than the actual patterns. In this case, the polynomial kernel likely failed to generalize well to the unseen test data.

RBF Kernel (95.32%): The RBF kernel achieved the highest accuracy at 95.32%, slightly outperforming the linear kernel. The RBF kernel is known for its ability to handle non-linear relationships, and its slight improvement here suggests that the data may have some underlying non-linear patterns that the RBF kernel is better equipped to capture. The RBF kernel's flexibility allows it to fit the data more precisely compared to the linear kernel, which is why it might have a marginally better performance.

- for Task 2

The range log2c = [i for i in range(-6, 7)] is used to define the possible values for the logarithm of the kernel parameter $c$in the grid search. This range specifies the exponents of 2, covering values from 2^{-6} 2^{6}

Why -6 to 6?: The values from $ -6 ~ 6$represent γvalues spanning from very small (i.e., $2^{-6} \approx 0.0156$) to very large (i.e., $2^{6} = 64$).

This range is chosen to cover a broad range of regularization strengths, ensuring that both over-regularized (underfitting) and under-regularized (overfitting) models are considered.

The range log2g = [i for i in range(-6, 7)] is used to define the possible values for the logarithm of the kernel parameter γin the grid search. This range specifies the exponents of 2, covering values from 2^{-6} 2^{6}

Why -6 to 6?: The values from -6 6represent γvalues spanning from very small (i.e., $2^{-6} \approx 0.0156$) to very large (i.e., $2^{6} = 64$)This allows the grid search to test both highly localized and more general decision boundaries:

Small values: lead to a smoother, less complex decision boundary that may underfit the data.

Large values: lead to a more flexible, highly sensitive decision boundary that may overfit the data.

The accuracy of the model on the test data demonstrates a clear performance hierarchy, with the Radial Basis Function (RBF) kernel yielding the highest accuracy at 98.52%, followed by the polynomial kernel at 97.48%, and the linear kernel at 95.52%. This result suggests that the RBF kernel is the most effective for capturing the underlying patterns in the data, likely due to its ability to model non-linear relationships. The polynomial kernel also performs well, though slightly less effectively than the RBF, as it is capable of modeling more complex decision boundaries but with slightly less flexibility. On the other hand, the linear kernel, while efficient for simpler, linearly separable data, tends to underperform in comparison to the other two kernels, possibly because it fails to capture more intricate, non-linear relationships in the dataset. The observed performance differences between the kernels can be attributed to the nature of the data and the capacity of each kernel to represent the complexity of the underlying decision boundaries.Overall, all three kernels achieved high accuracy, demonstrating the effectiveness of the chosen methods in producing reliable results.

- for Task 3

Compared to Task 2, where the kernel functions used were linear, polynomial, and RBF, in Task 3, using a combination of linear and RBF kernels yielded slightly better accuracy than using only the linear kernel. However, the best results were obtained when using only the RBF kernel.

This improvement with the RBF kernel can be attributed to its ability to map data into a higher-dimensional space, which is beneficial when dealing with non-linear relationships. The RBF kernel has the advantage of better capturing complex patterns in the data, leading to higher generalization performance compared to the linear kernel. Although combining kernels can sometimes enhance the model's ability to generalize, in this case, the pure RBF kernel outperformed the combination, likely because it more effectively handled the underlying non-linearities present in this dataset.