# Homework 5 Report

# I. Gaussian Process

## 1. Code

### Task 1

Apply Gaussian Process Regression to predict the distribution of $f$ and visualize the result. Use rational quadratic kernel to compute similarities between different points.

```python
class GaussianProcess:
    def __init__(self, data, beta=5):
        self.X = data.T[0]
        self.Y = data.T[1]
        self.beta = beta

    def kernel(self, x1, x2, sigma, alpha, l):
        return sigma * (1 + ((x1 - x2)**2) / (2 * alpha * l**2)) ** (-alpha)

    def gaussian(self, sigma, alpha, l):
        X = self.X.reshape(-1, 1)
        C = self.kernel(X, X.T, sigma, alpha, l)
        C += np.eye(len(X)) / self.beta
        return C
```

1. `__init__`: Initializes the Gaussian Process with the training data and the noise precision ( `beta` ).

   `self.X` stores the training data's input values, and `self.Y` stores the corresponding target values. where `beta` is set to 5 as specified in the problem statement.

2. `kernel`: This function defines the **Rational Quadratic Kernel**, which is commonly used in Gaussian Process regression. The formula is:

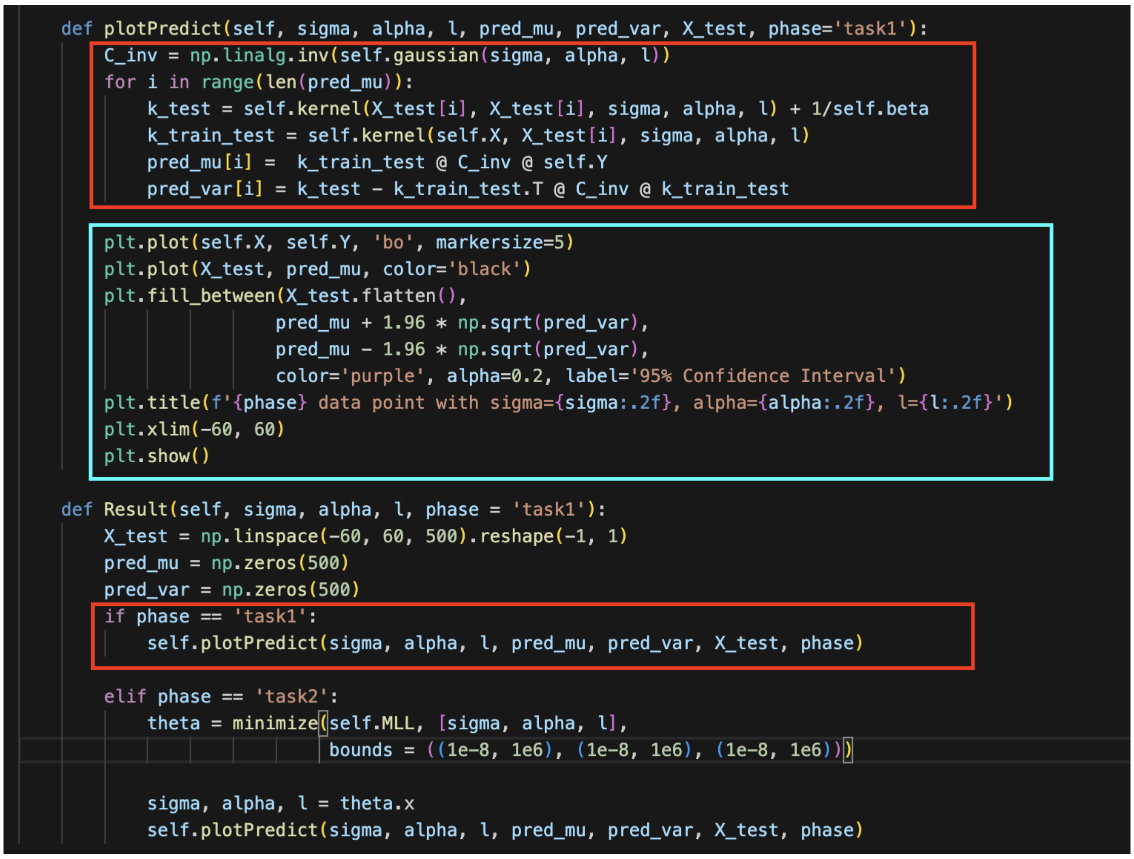
$$k(x_1, x_2) = \sigma^2 \left(1 + \frac{(x_1 - x_2)^2}{2\alpha\ell^2}\right)^{-\alpha}$$

   where the parameters represent:

   - `sigma` $=\sigma^2$ is the kernel scale.
   - `alpha` $=\alpha$ controls the relative weight of scales.
   - `l` $=\ell$ is the length scale that determines the smoothness of the function.

3. `gaussian`: This function calculates the covariance matrix between all pairs of training points, then adds the noise term $\frac{1}{\beta}$ to the diagonal to account for observational noise. This makes the matrix invertible and positive definite. The formula is:

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



```python
def plotPredict(self, sigma, alpha, l, pred_mu, pred_var, X_test, phase='task1'):
    C_inv = np.linalg.inv(self.gaussian(sigma, alpha, l))
    for i in range(len(pred_mu)):
        k_test = self.kernel(X_test[i], X_test[i], sigma, alpha, l) + 1/self.beta
        k_train_test = self.kernel(self.X, X_test[i], sigma, alpha, l)
        pred_mu[i] =  k_train_test @ C_inv @ self.Y
        pred_var[i] = k_test - k_train_test.T @ C_inv @ k_train_test

    plt.plot(self.X, self.Y, 'bo', markersize=5)
    plt.plot(X_test, pred_mu, color='black')
    plt.fill_between(X_test.flatten(),
                        pred_mu + 1.96 * np.sqrt(pred_var),
                        pred_mu - 1.96 * np.sqrt(pred_var),
                        color='purple', alpha=0.2, label='95% Confidence Interval')
    plt.title(f'{phase} data point with sigma={sigma:.2f}, alpha={alpha:.2f}, l={l:.2f}')
    plt.xlim(-60, 60)
    plt.show()

def Result(self, sigma, alpha, l, phase = 'task1'):
    X_test = np.linspace(-60, 60, 500).reshape(-1, 1)
    pred_mu = np.zeros(500)
    pred_var = np.zeros(500)
    if phase == 'task1':
        self.plotPredict(sigma, alpha, l, pred_mu, pred_var, X_test, phase)

    elif phase == 'task2':
        theta = minimize(self.MLL, [sigma, alpha, l],
                         bounds = ((1e-8, 1e6), (1e-8, 1e6), (1e-8, 1e6)))

        sigma, alpha, l = theta.x
        self.plotPredict(sigma, alpha, l, pred_mu, pred_var, X_test, phase)
```

4. **`plotPredict`**: The **predict part** is defined in the plotPredict and highlight by the red box.This function uses the kernel function( `k_test` ) and the inverse of the kernel matrix( `k_inv` ) to predict the mean ( `pred_mu` ) and variance ( `pred_var` ) for the test points ( `X_test` ), where the formula for the predicted mean for a test point $x^*$ is:

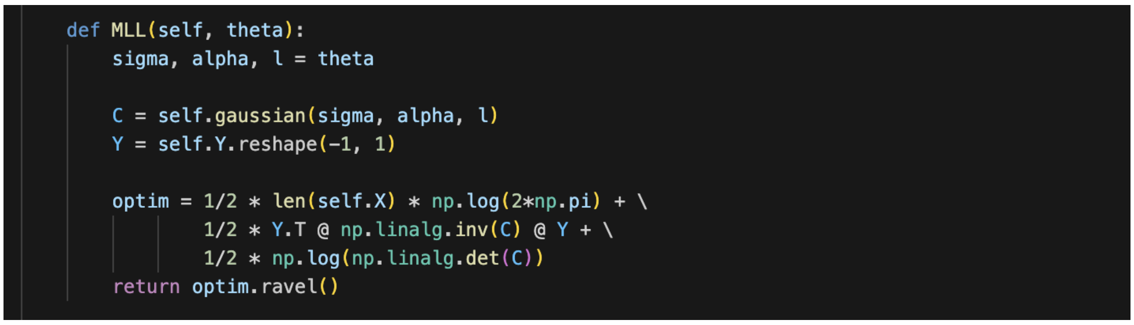
   - `k_inv` : $K^{-1}$
   - `k_test` : $k(x, x^*) + \beta^{-1}$
   - `pred_mu` : $\mu(x^*) = k(x, x^*)^T K^{-1} y$
   - `pred_var` : $\sigma^2(x^*) = k(x^*, x^*) - k(x, x^*) K^{-1} k(x, x^*)$

   After the predict, we can then visualizes the training data, predicted mean, and 95% confidence interval (using the predicted variance) on a plot, this part is  highlight by the blue box.

5. **`Result`**: This function defines the range of test points ( `X_test` ) and sets initial values for the predicted mean ( `pred_mu` ) and variance ( `pred_var` ), as specified in the problem statement, we need to predict mean and variance for each test point in range x=[-60, 60]. In Task 1, it directly calls `plotResult` with predefined values for σ, α, and l.

## Task2

In task2, the function for kernel function, Gaussian process, prediction and train/test data are same as task1.



```python
def MLL(self, theta):
    sigma, alpha, l = theta

    C = self.gaussian(sigma, alpha, l)
    Y = self.Y.reshape(-1, 1)

    optim = 1/2 * len(self.X) * np.log(2*np.pi) + \
            1/2 * Y.T @ np.linalg.inv(C) @ Y + \
            1/2 * np.log(np.linalg.det(C))
    return optim.ravel()
```

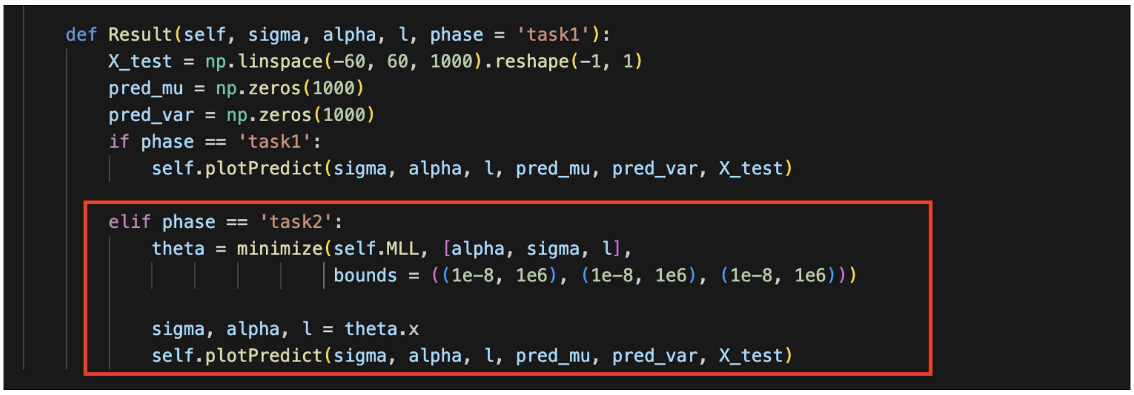
1. `MLL` : This function calculates the **Marginal Log-Likelihood** (MLL) for the Gaussian Process, the formula is:

$$\mathcal{L}(\theta) = \frac{N}{2}\log 2\pi + \frac{1}{2}\mathbf{y}^T K^{-1}\mathbf{y} + \frac{1}{2}\log|K|$$

   Where:

   - `K` is the covariance matrix from Gaussian process.
   - `len(self.X)` =N is the number of training points.

   The log-likelihood is used for optimizing the kernel parameters $\sigma, \alpha, \ell$.

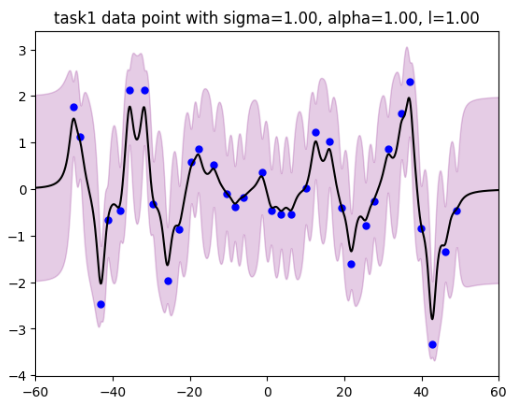


```python
def Result(self, sigma, alpha, l, phase = 'task1'):
    X_test = np.linspace(-60, 60, 1000).reshape(-1, 1)
    pred_mu = np.zeros(1000)
    pred_var = np.zeros(1000)
    if phase == 'task1':
        self.plotPredict(sigma, alpha, l, pred_mu, pred_var, X_test)

    elif phase == 'task2':
        theta = minimize(self.MLL, [alpha, sigma, l],
                         bounds = ((1e-8, 1e6), (1e-8, 1e6), (1e-8, 1e6)))

        sigma, alpha, l = theta.x
        self.plotPredict(sigma, alpha, l, pred_mu, pred_var, X_test)
```

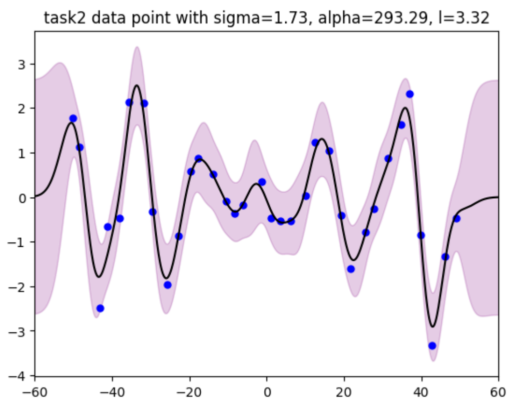
2. `Result` : This part is same as task1, but in task 2, we need to minimizes the Marginal Log-Likelihood (MLL) using `scipy.optimize.` `minimize` function to optimize the parameters,  and set bounds = $(10^{-8}, 10^{6})$ for each parameter.

## 2. Experiments

### Task1



### Task2



## 3. Observations and Discussion:

### a. Performance Comparison

The two tasks highlight the impact of hyperparameters $\sigma, \alpha, \ell$ on the performance of Gaussian Process Regression:

| Metric | **Task 1** | **Task 2** |
|---|---|---|
| **sigma (σ)** | 1.00 | 1.73 |
| **alpha (α)** | 1.00 | 293.29 |
| **Length Scale ($\ell$)** | 1.00 | 3.32 |
| **Model Confidence** | Low in non-data regions | Higher in all regions |
| **Observations** | 1. Predictions are smooth but fail to capture oscillations in the training data.<br>2. Confidence intervals grow quickly in regions far from training points. | 1. Predictions capture both sharp and smooth transitions in the training data.<br>2. Confidence intervals are consistently lower, especially near training points. |
| **Reasons** | 1. Small $\ell$, focuses on local variations, struggles with long-range dependencies.<br>2. Small $\alpha$ limits the ability to balance different scales. | 1. Larger $\ell$, models long-range dependencies, provides smoother predictions.<br>2. Higher $\alpha$ enhances adaptability to sharp transitions. |

## b. Importance of Hyperparameter

The comparison demonstrates that the choice of hyperparameters is crucial in Gaussian Process Regression.

In Task 1, inappropriate hyperparameters may lead to underfitting or poor prediction performance, with overly large confidence intervals. In Task 2, the optimized hyperparameters make the model more confident (narrower confidence intervals) and more accurate in its predictions, particularly in capturing the oscillations in the training data.

## c. Confidence Intervals

In task 1, The confidence intervals expand significantly in regions with insufficient training data (e.g., near $x = \pm 60$ ), indicating high uncertainty. In the confidence intervals are more uniform and narrower across the test range, showing that the optimized hyperparameters improve the model's confidence in its predictions.

The addition of more training data in sparse regions would further reduce uncertainty and improve prediction accuracy, even without re-optimizing hyperparameters.

---

# II. SVM on MNIST

In this part, we will use 3 different kernel function to  tackle classification on images of hand-written digits 0-4.

### a. Linear Kernel

The linear kernel computes the dot product of two feature vectors, suitable for linearly separable data. It does not introduce non-linearity and works directly in the original feature space.

where $x_i$ and $x_j$ are input feature vectors.

### b. Polynomial Kernel

The polynomial kernel adds non-linearity by computing the power of the dot product, optionally scaled and shifted by coefficients.

$$K(x_i, x_j) = \left(\gamma \cdot x_i^T x_j + r\right)^d$$

$$\text{where} \begin{cases} \gamma : \text{scaling factor} \\ r : \text{Coefficient} \\ d : \text{Degree of the polynomial} \end{cases}$$

**c. RBF (Radial Basis Function) Kernel**

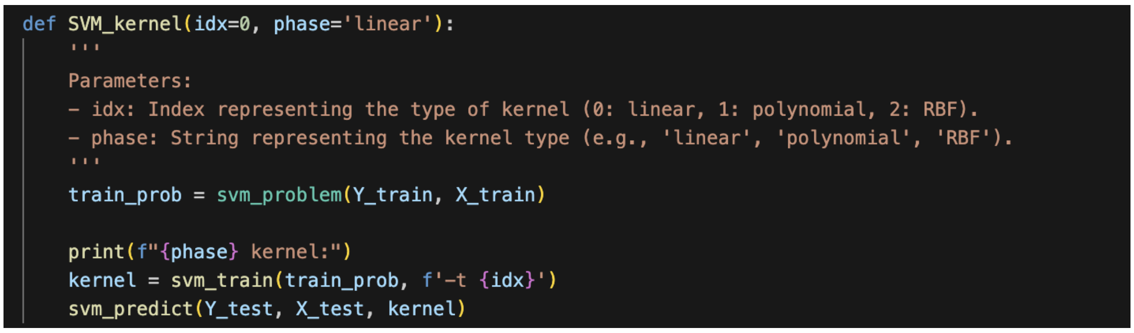
The RBF kernel is a Gaussian kernel that measures the similarity between two points based on their Euclidean distance (maps points that are close in the original space to a high similarity value). It is highly flexible and widely used for non-linear data.

$$K(x_i, x_j) = \exp\left(-\gamma \|x_i - x_j\|^2\right)$$

where $\gamma > 0$, controls the kernel's width (higher  makes the kernel more localized).

## Task1

### 1. Code



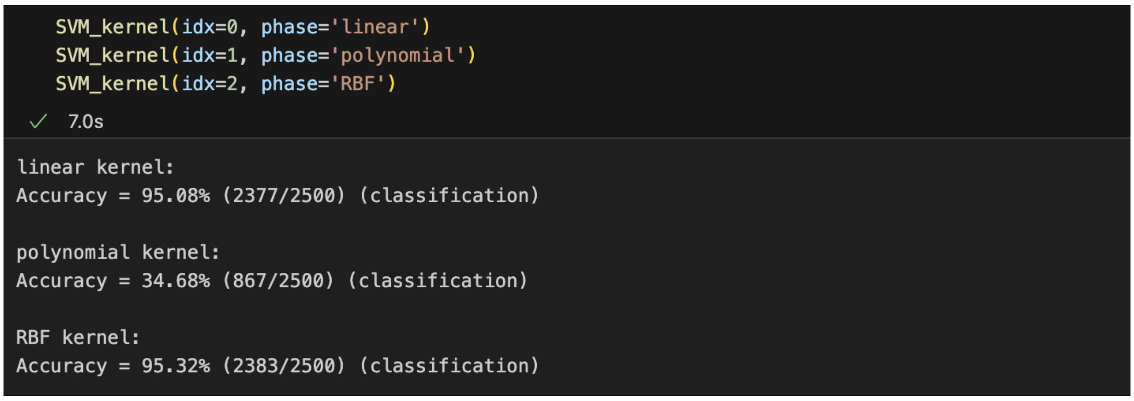
```python
def SVM_kernel(idx=0, phase='linear'):
    '''
    Parameters:
    - idx: Index representing the type of kernel (0: linear, 1: polynomial, 2: RBF).
    - phase: String representing the kernel type (e.g., 'linear', 'polynomial', 'RBF').
    '''
    train_prob = svm_problem(Y_train, X_train)

    print(f"{phase} kernel:")
    kernel = svm_train(train_prob, f'-t {idx}')
    svm_predict(Y_test, X_test, kernel)
```

The function `SVM_kernel` loops through different kernel types by changing the `idx` and `phase` arguments. For each kernel:

- `svm_problem` defines the training data for the SVM model, where `Y_train` is training labels and `X_train` is feature vectors.
- `svm_train` trains the SVM model using the specified kernel.The option `-t { idx }` represent different type of kernel (0: linear, 1: polynomial, 2: RBF).

- `svm_predict` tests the trained model on the test data (`X_test`) and compares predictions with the ground truth (`Y_test`) and print the results.

## 2. Experiments



```
    SVM_kernel(idx=0, phase='linear')
    SVM_kernel(idx=1, phase='polynomial')
    SVM_kernel(idx=2, phase='RBF')
 ✓  7.0s
linear kernel:
Accuracy = 95.08% (2377/2500) (classification)

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

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

## 3. Observations and Discussion

### Observations

- **Linear Kernel**:

  The linear kernel achieved an accuracy of **95.08%**, demonstrating strong performance on this dataset. This suggests that the data may be nearly linearly separable, allowing the linear kernel to effectively classify the samples.

- **Polynomial Kernel**:

  The polynomial kernel performed poorly with an accuracy of **34.68%**, indicating that the chosen degree or scaling parameters might not be suitable for this dataset. The kernel likely overfits or fails to capture the underlying patterns, resulting in significantly lower performance.

- **RBF Kernel**:

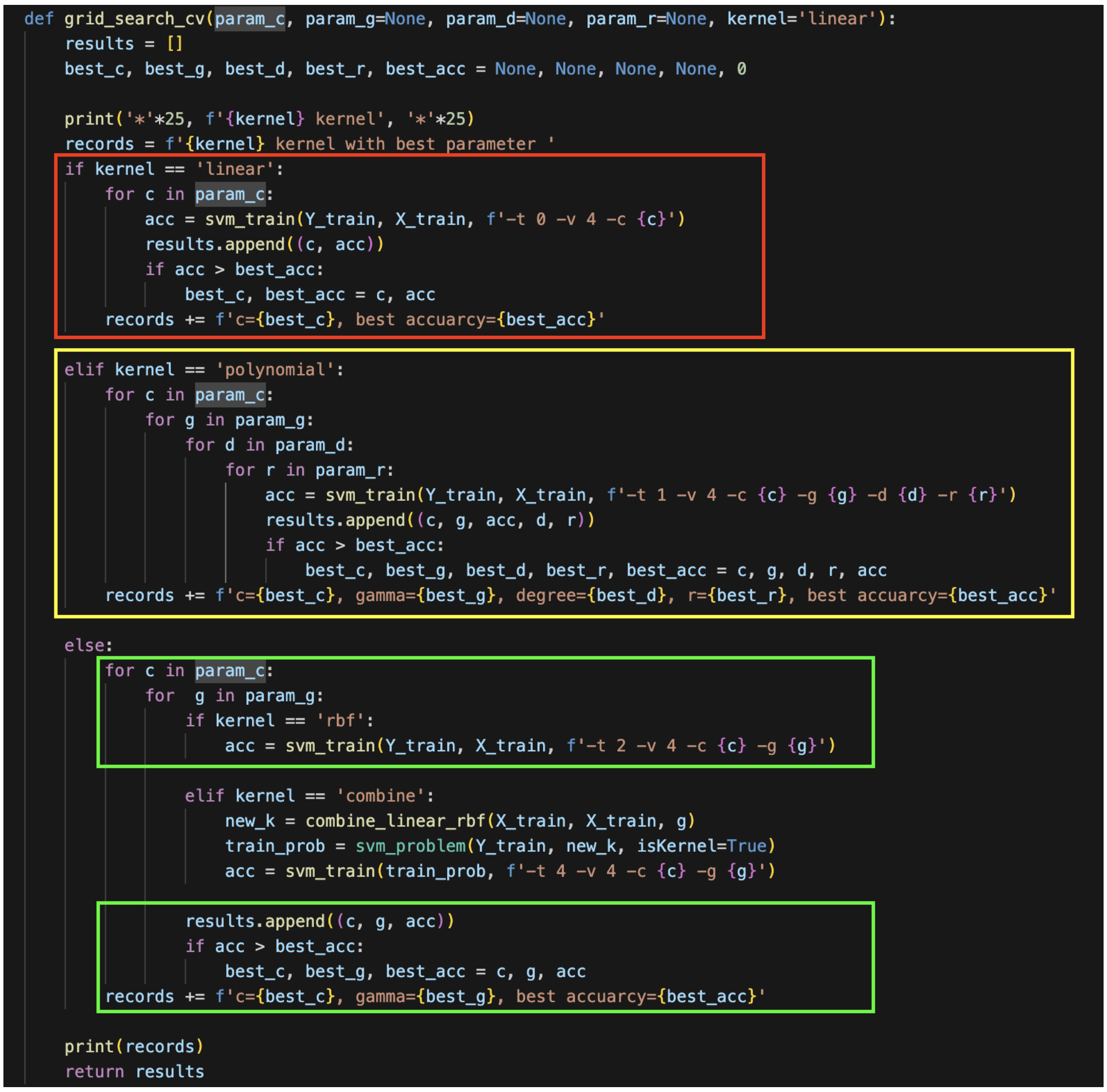
  The RBF kernel achieved the highest accuracy of 95.32%, marginally outperforming the linear kernel. This suggests that the RBF kernel's ability to model non-linear relationships provided some benefit, though the slight improvement indicates that the data is primarily linear, with the RBF kernel enhancing classification by capturing subtle non-linear patterns.

### Discussion

- Further optimize the polynomial kernel's parameters (C, $\gamma$, d, r) to evaluate its potential performance.
- Use grid search or cross-validation to validate the robustness of the kernel's parameters.

## Task2

### 1. Code



```python
def grid_search_cv(param_c, param_g=None, param_d=None, param_r=None, kernel='linear'):
    results = []
    best_c, best_g, best_d, best_r, best_acc = None, None, None, None, 0

    print('*'*25, f'{kernel} kernel', '*'*25)
    records = f'{kernel} kernel with best parameter '
    if kernel == 'linear':
        for c in param_c:
            acc = svm_train(Y_train, X_train, f'-t 0 -v 4 -c {c}')
            results.append((c, acc))
            if acc > best_acc:
                best_c, best_acc = c, acc
        records += f'c={best_c}, best accuarcy={best_acc}'

    elif kernel == 'polynomial':
        for c in param_c:
            for g in param_g:
                for d in param_d:
                    for r in param_r:
                        acc = svm_train(Y_train, X_train, f'-t 1 -v 4 -c {c} -g {g} -d {d} -r {r}')
                        results.append((c, g, acc, d, r))
                        if acc > best_acc:
                            best_c, best_g, best_d, best_r, best_acc = c, g, d, r, acc
        records += f'c={best_c}, gamma={best_g}, degree={best_d}, r={best_r}, best accuarcy={best_acc}'

    else:
        for c in param_c:
            for  g in param_g:
                if kernel == 'rbf':
                    acc = svm_train(Y_train, X_train, f'-t 2 -v 4 -c {c} -g {g}')

                elif kernel == 'combine':
                    new_k = combine_linear_rbf(X_train, X_train, g)
                    train_prob = svm_problem(Y_train, new_k, isKernel=True)
                    acc = svm_train(train_prob, f'-t 4 -v 4 -c {c} -g {g}')

                results.append((c, g, acc))
                if acc > best_acc:
                    best_c, best_g, best_acc = c, g, acc
        records += f'c={best_c}, gamma={best_g}, best accuarcy={best_acc}'

    print(records)
    return results
```

The function `grid_search_cv` is to use C-SVC (Soft-Margin Support Vector Classification) to train an SVM model and perform **grid search cross-validation** to identify the best-performing hyperparameters.

- `param_c` : A list of candidate values for regularization parameter C , which will be used for every kernel.

  It controls the trade-off between margin size and misclassification. Larger C reduces training errors but risks overfitting, while smaller  C increases the margin but may lead to underfitting.

- `param_g` : A list of candidate values for γ, which will be used for RBF or polynomial).

  For RBF, it determines the influence of individual data points. Larger  $\gamma$ have localized influence, potentially leading to overfitting, while smaller $\gamma$ have broader influence, potentially leading to underfitting.

  For Polynomial, it acts as a scaling factor that adjusts the influence of input feature magnitudes in the kernel calculation.

- `param_d` : A list of candidate values for the degree of the polynomial kernel.

  Determines the complexity of the polynomial decision boundary. Higher d Allows more complex decision boundaries but increases the risk of overfitting.

- `param_r` : A list of candidate values for the coefficient r in the polynomial kernel.

  A constant term added in the polynomial kernel, controls the baseline influence of the polynomial features.

- `kernel` : Specifies the kernel type.

The arguments for `-t` , `-c` , `-g` , `-d` , and `-r` refer to the kernel type, C , $\gamma$, degree, and $r$, respectively, as mentioned before. The `-v` argument specifies the number of folds for cross-validation.

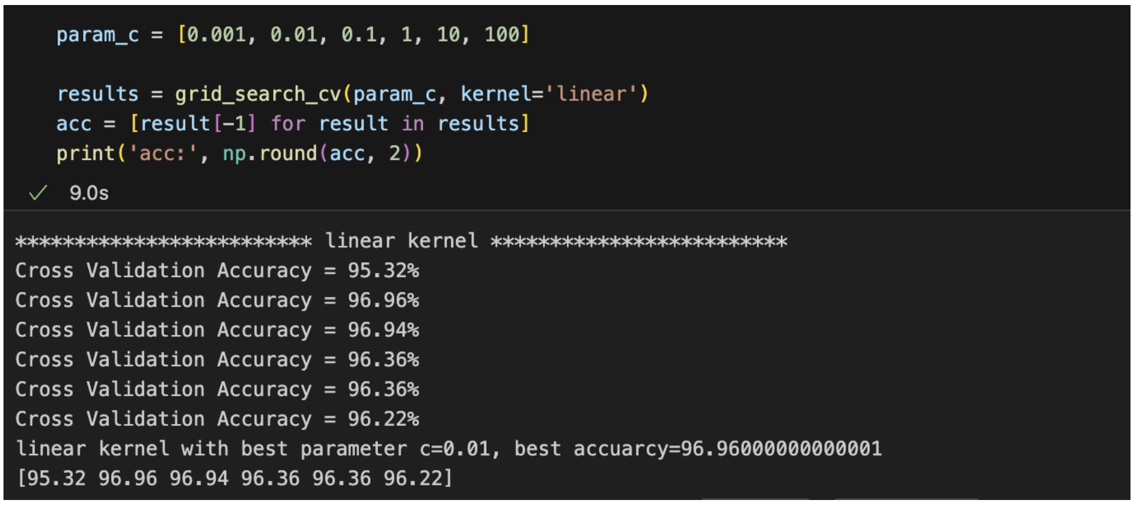
Set the training data and perform a grid search for different types of kernels (**r**ed box for linear, yellow for polynomial, and green for RBF). The grid search uses 4-fold cross-validation to evaluate different parameter combinations and identifies the optimal parameters for each kernel function.

Specifically, it records the best values for C ( `best_c` ), $\gamma$ ( `best_g` ), degree ( `best_d` ), and $r$ ( `best_r` ) along with their corresponding accuracies in the variable `best_acc` At the same time, it uses `records` to print the parameter combination that achieves the best accuracy.

After training, use the best pair parameters to do testing and return all values we need.

## 2. Experiments

## linear kernel



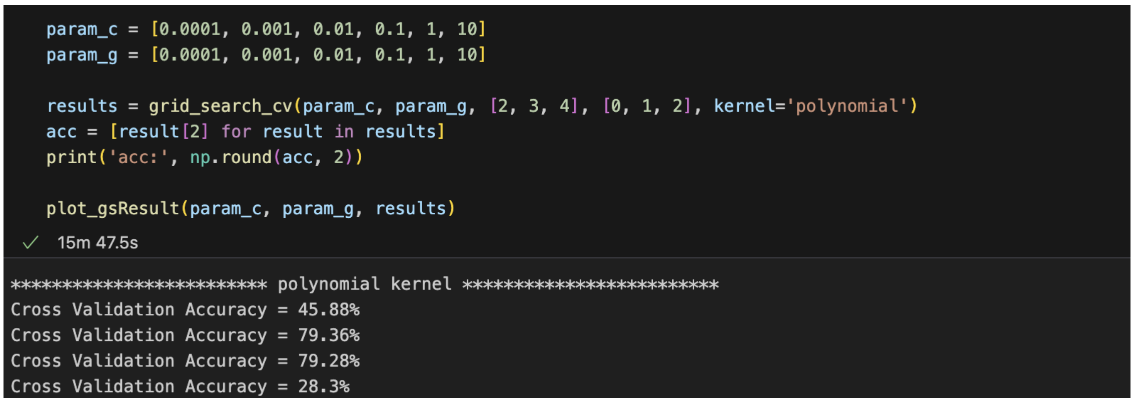
```
param_c = [0.001, 0.01, 0.1, 1, 10, 100]

results = grid_search_cv(param_c, kernel='linear')
acc = [result[-1] for result in results]
print('acc:', np.round(acc, 2))
```

✓ 9.0s

```
************************* linear kernel *************************
Cross Validation Accuracy = 95.32%
Cross Validation Accuracy = 96.96%
Cross Validation Accuracy = 96.94%
Cross Validation Accuracy = 96.36%
Cross Validation Accuracy = 96.36%
Cross Validation Accuracy = 96.22%
linear kernel with best parameter c=0.01, best accuarcy=96.96000000000001
[95.32 96.96 96.94 96.36 96.36 96.22]
```

Set param_c = [0.001, 0.01, 0.1, 1, 10, 100], with best parameter c=0.01 and best accuracy=96.96%

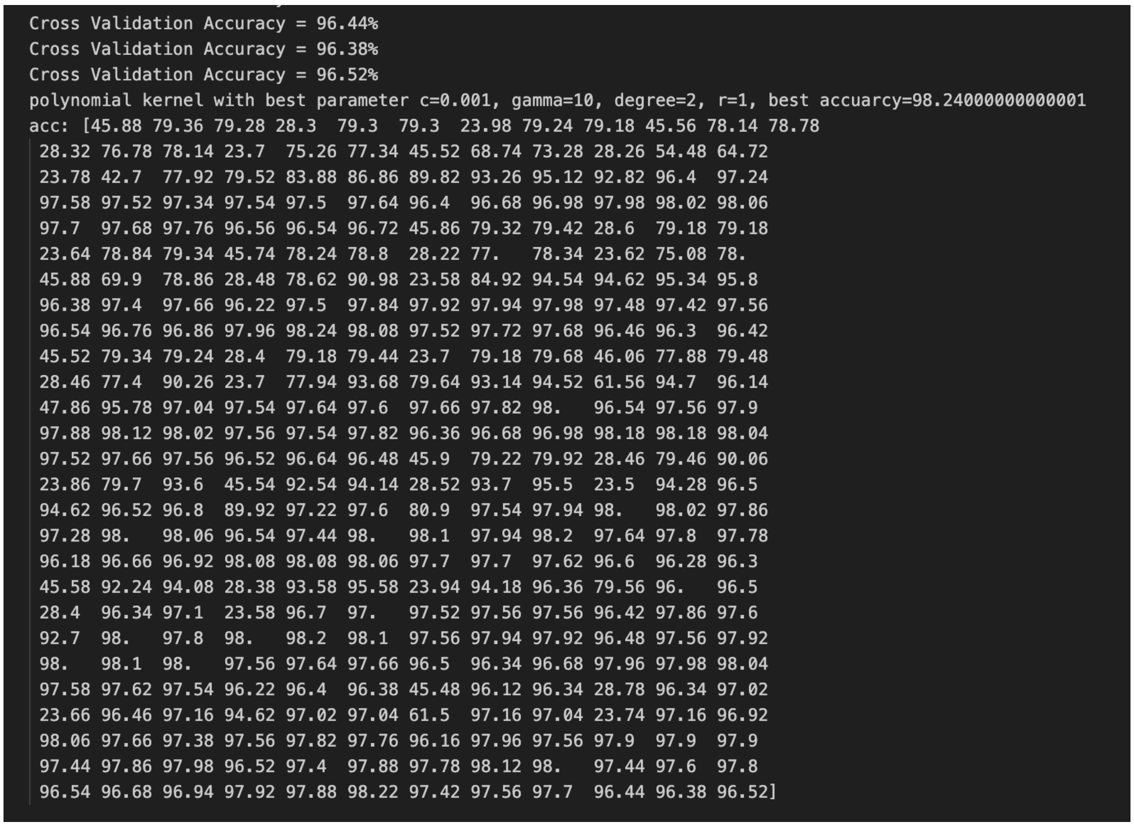
## polynomial kernel



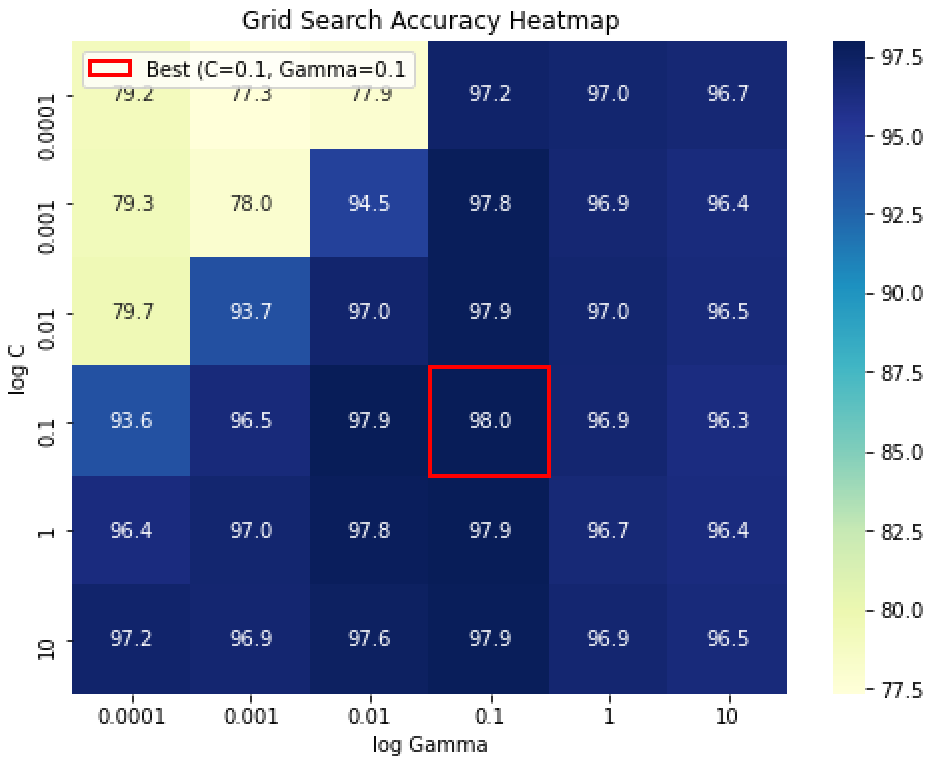
```
param_c = [0.0001, 0.001, 0.01, 0.1, 1, 10]
param_g = [0.0001, 0.001, 0.01, 0.1, 1, 10]

results = grid_search_cv(param_c, param_g, [2, 3, 4], [0, 1, 2], kernel='polynomial')
acc = [result[2] for result in results]
print('acc:', np.round(acc, 2))

plot_gsResult(param_c, param_g, results)
```

✓ 15m 47.5s

```
************************* polynomial kernel *************************
Cross Validation Accuracy = 45.88%
Cross Validation Accuracy = 79.36%
Cross Validation Accuracy = 79.28%
Cross Validation Accuracy = 28.3%
```



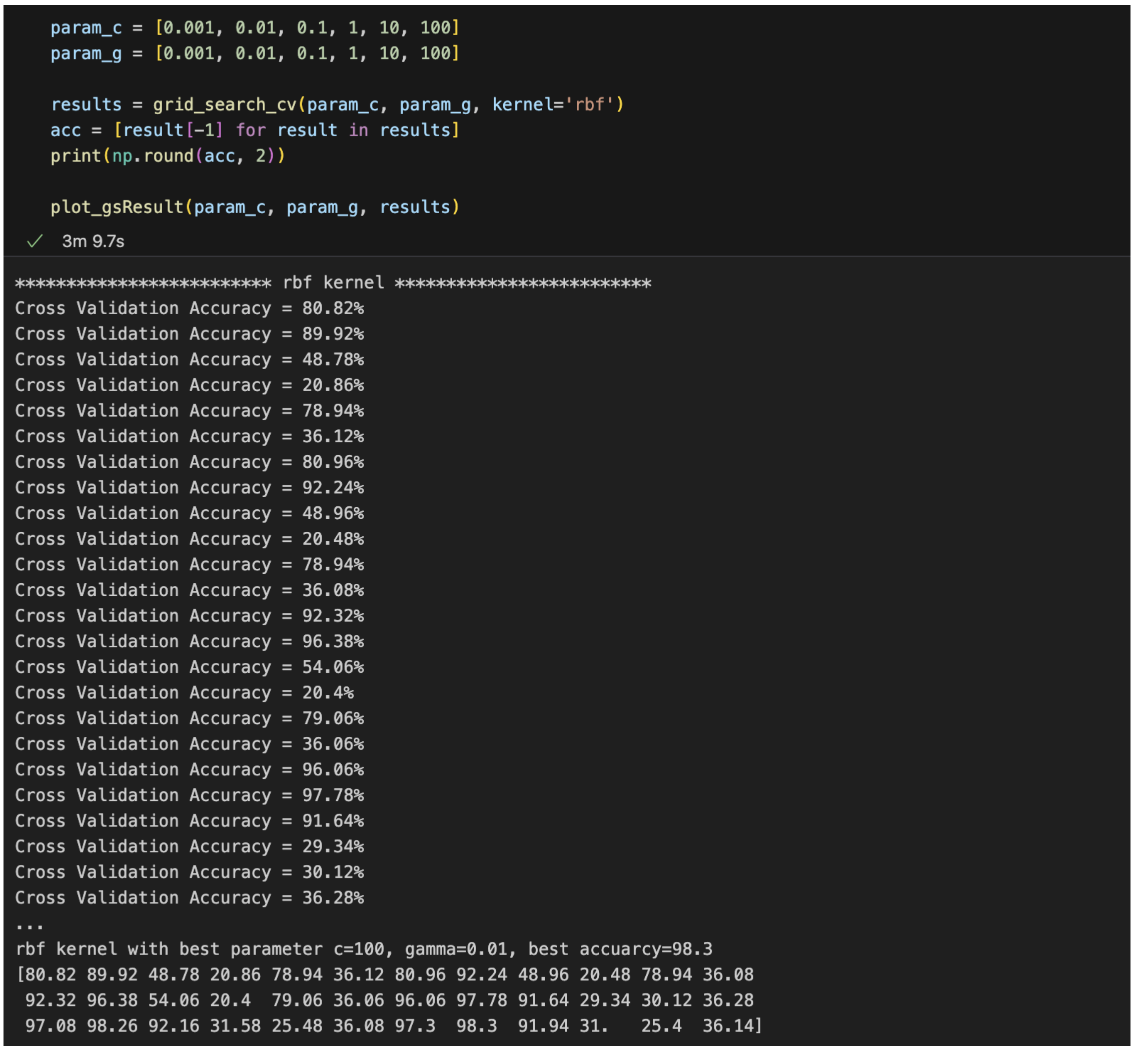
```
Cross Validation Accuracy = 96.44%
Cross Validation Accuracy = 96.38%
Cross Validation Accuracy = 96.52%
polynomial kernel with best parameter c=0.001, gamma=10, degree=2, r=1, best accuarcy=98.24000000000001
acc: [45.88 79.36 79.28 28.3  79.3  79.3  23.98 79.24 79.18 45.56 78.14 78.78
 28.32 76.78 78.14 23.7  75.26 77.34 45.52 68.74 73.28 28.26 54.48 64.72
 23.78 42.7  77.92 79.52 83.88 86.86 89.82 93.26 95.12 92.82 96.4  97.24
 97.58 97.52 97.34 97.54 97.5  97.64 96.4  96.68 96.98 97.98 98.02 98.06
 97.7  97.68 97.76 96.56 96.54 96.72 45.86 79.32 79.42 28.6  79.18 79.18
 23.64 78.84 79.34 45.74 78.24 78.8  28.22 77.   78.34 23.62 75.08 78.
 45.88 69.9  78.86 28.48 78.62 90.98 23.58 84.92 94.54 94.62 95.34 95.8
 96.38 97.4  97.66 96.22 97.5  97.84 97.92 97.94 97.98 97.48 97.42 97.56
 96.54 96.76 96.86 97.96 98.24 98.08 97.52 97.72 97.68 96.46 96.3  96.42
 45.52 79.34 79.24 28.4  79.18 79.44 23.7  79.18 79.68 46.06 77.88 79.48
 28.46 77.4  90.26 23.7  77.94 93.68 79.64 93.14 94.52 61.56 94.7  96.14
 47.86 95.78 97.04 97.54 97.64 97.6  97.66 97.82 98.   96.54 97.56 97.9
 97.88 98.12 98.02 97.56 97.54 97.82 96.36 96.68 96.98 98.18 98.18 98.04
 97.52 97.66 97.56 96.52 96.64 96.48 45.9  79.22 79.92 28.46 79.46 90.06
 23.86 79.7  93.6  45.54 92.54 94.14 28.52 93.7  95.5  23.5  94.28 96.5
 94.62 96.52 96.8  89.92 97.22 97.6  80.9  97.54 97.94 98.   98.02 97.86
 97.28 98.   98.06 96.54 97.44 98.   98.1  97.94 98.2  97.64 97.8  97.78
 96.18 96.66 96.92 98.08 98.08 98.06 97.7  97.7  97.62 96.6  96.28 96.3
 45.58 92.24 94.08 28.38 93.58 95.58 23.94 94.18 96.36 79.56 96.   96.5
 28.4  96.34 97.1  23.58 96.7  97.   97.52 97.56 97.56 96.42 97.86 97.6
 92.7  98.   97.8  98.   98.2  98.1  97.56 97.94 97.92 96.48 97.56 97.92
 98.   98.1  98.   97.56 97.64 97.66 96.5  96.34 96.68 97.96 97.98 98.04
 97.58 97.62 97.54 96.22 96.4  96.38 45.48 96.12 96.34 28.78 96.34 97.02
 23.66 96.46 97.16 94.62 97.02 97.04 61.5  97.16 97.04 23.74 97.16 96.92
 98.06 97.66 97.38 97.56 97.82 97.76 96.16 97.96 97.56 97.9  97.9  97.9
 97.44 97.86 97.98 96.52 97.4  97.88 97.78 98.12 98.   97.44 97.6  97.8
 96.54 96.68 96.94 97.92 97.88 98.22 97.42 97.56 97.7  96.44 96.38 96.52]
```



Set param_c = [0.0001, 0.001, 0.01, 0.1, 1, 10], param_g = [0.0001, 0.001, 0.01, 0.1, 1, 10], without considering d and r, the best parameters are C=0.1 and γ=0.1, achieving an accuracy of **98.0%**.

Considering param_d=[2, 3, 4], param_r=[0, 1, 2], with best parameter C=0.001, γ=10, d=2, r=1, best accuracy =98.24.

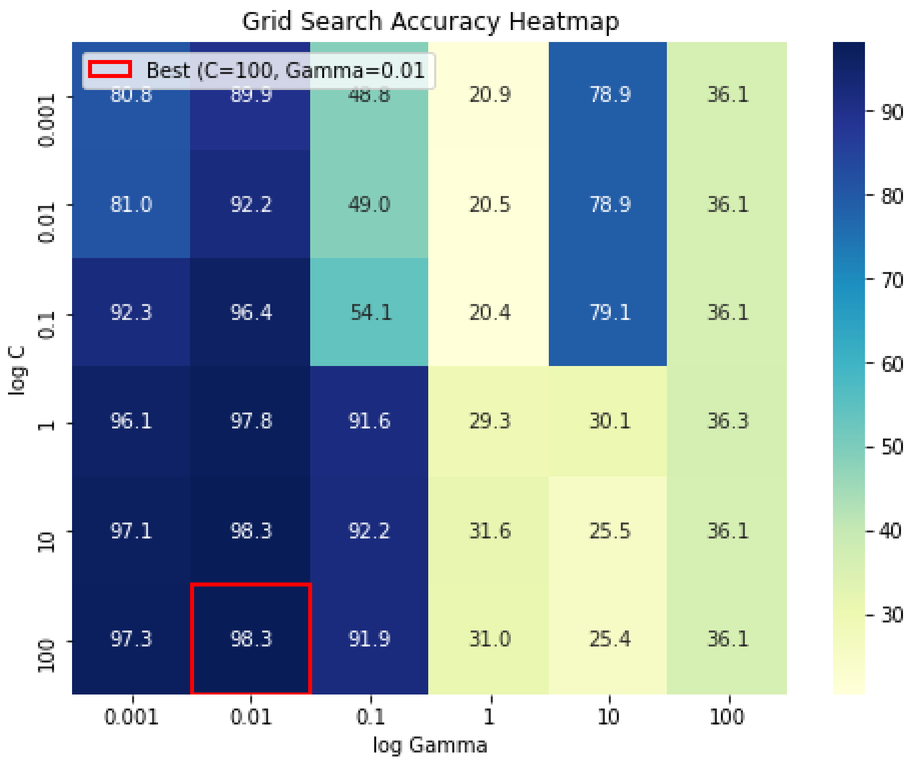
## RBF kernel



```
param_c = [0.001, 0.01, 0.1, 1, 10, 100]
param_g = [0.001, 0.01, 0.1, 1, 10, 100]

results = grid_search_cv(param_c, param_g, kernel='rbf')
acc = [result[-1] for result in results]
print(np.round(acc, 2))

plot_gsResult(param_c, param_g, results)
```

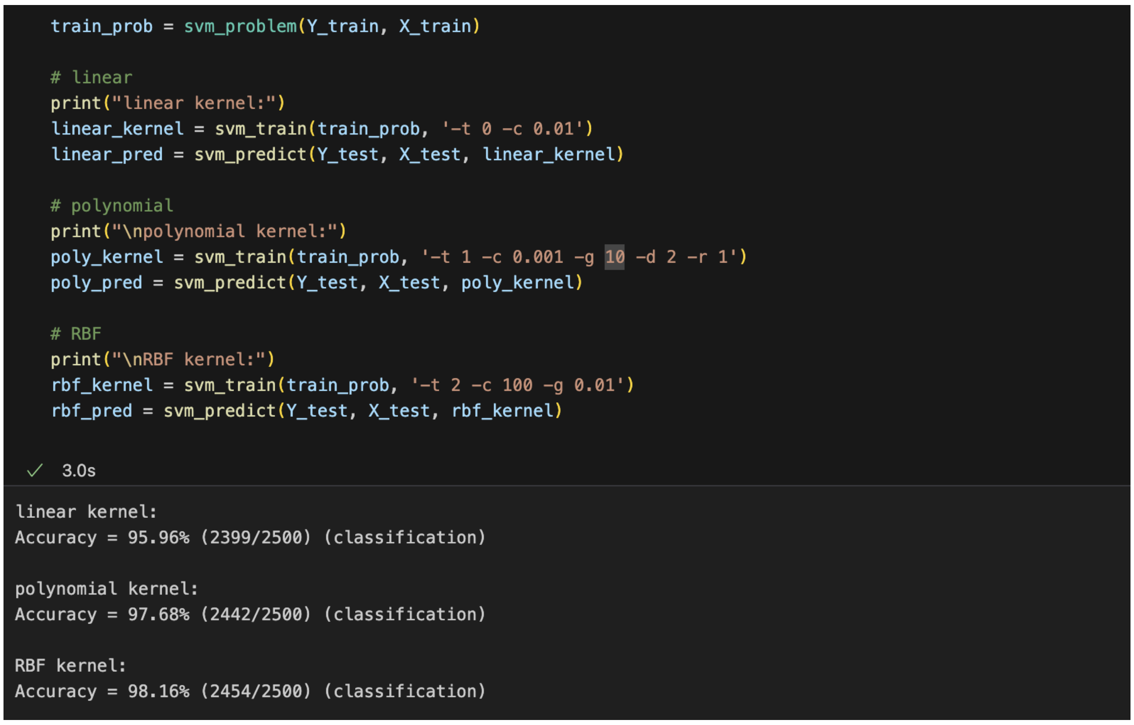
✓ 3m 9.7s

```
************************* rbf kernel *************************
Cross Validation Accuracy = 80.82%
Cross Validation Accuracy = 89.92%
Cross Validation Accuracy = 48.78%
Cross Validation Accuracy = 20.86%
Cross Validation Accuracy = 78.94%
Cross Validation Accuracy = 36.12%
Cross Validation Accuracy = 80.96%
Cross Validation Accuracy = 92.24%
Cross Validation Accuracy = 48.96%
Cross Validation Accuracy = 20.48%
Cross Validation Accuracy = 78.94%
Cross Validation Accuracy = 36.08%
Cross Validation Accuracy = 92.32%
Cross Validation Accuracy = 96.38%
Cross Validation Accuracy = 54.06%
Cross Validation Accuracy = 20.4%
Cross Validation Accuracy = 79.06%
Cross Validation Accuracy = 36.06%
Cross Validation Accuracy = 96.06%
Cross Validation Accuracy = 97.78%
Cross Validation Accuracy = 91.64%
Cross Validation Accuracy = 29.34%
Cross Validation Accuracy = 30.12%
Cross Validation Accuracy = 36.28%
...
rbf kernel with best parameter c=100, gamma=0.01, best accuarcy=98.3
[80.82 89.92 48.78 20.86 78.94 36.12 80.96 92.24 48.96 20.48 78.94 36.08
 92.32 96.38 54.06 20.4  79.06 36.06 96.06 97.78 91.64 29.34 30.12 36.28
 97.08 98.26 92.16 31.58 25.48 36.08 97.3  98.3  91.94 31.   25.4  36.14]
```



Set param_c = [0.001, 0.01, 0.1, 1, 10, 100], param_g = [0.001, 0.01, 0.1, 1, 10, 100], with best parameter c=100, gamma=0.01, best accuracy=98.3.

## After grid search



```
train_prob = svm_problem(Y_train, X_train)

# linear
print("linear kernel:")
linear_kernel = svm_train(train_prob, '-t 0 -c 0.01')
linear_pred = svm_predict(Y_test, X_test, linear_kernel)

# polynomial
print("\npolynomial kernel:")
poly_kernel = svm_train(train_prob, '-t 1 -c 0.001 -g 10 -d 2 -r 1')
poly_pred = svm_predict(Y_test, X_test, poly_kernel)

# RBF
print("\nRBF kernel:")
rbf_kernel = svm_train(train_prob, '-t 2 -c 100 -g 0.01')
rbf_pred = svm_predict(Y_test, X_test, rbf_kernel)
```

✓ 3.0s

```
linear kernel:
Accuracy = 95.96% (2399/2500) (classification)

polynomial kernel:
Accuracy = 97.68% (2442/2500) (classification)

RBF kernel:
Accuracy = 98.16% (2454/2500) (classification)
```

## 3. Observations and Discussion

- **Linear Kernel**

  The linear kernel performs well with simple data, achieving high accuracy without the need for additional parameters. The performance remains relatively stable across C values, showing robustness for linearly separable data.

- **Polynomial kernel**

  Including d and r slightly improves accuracy, indicating that polynomial kernels benefit from carefully tuned degrees and coefficients.

  Higher γ captures finer data patterns but requires regularization through lower C, indicating that it benefits from higher regularization and scaling factors.

  Without considering d and r, the best parameters are C=0.1 and γ=0.1, achieving an accuracy of **98.0%**. This indicates that with just C and γ, the model already performs well.

  After incorporating d and r, the best parameters change to C = 0.001, γ=10, d=2, and r=1, achieving a slightly higher accuracy of **98.24%**. This improvement suggests that polynomial kernel-specific parameters d and r contribute to further optimizing the model's performance.

- **RBF kernel**

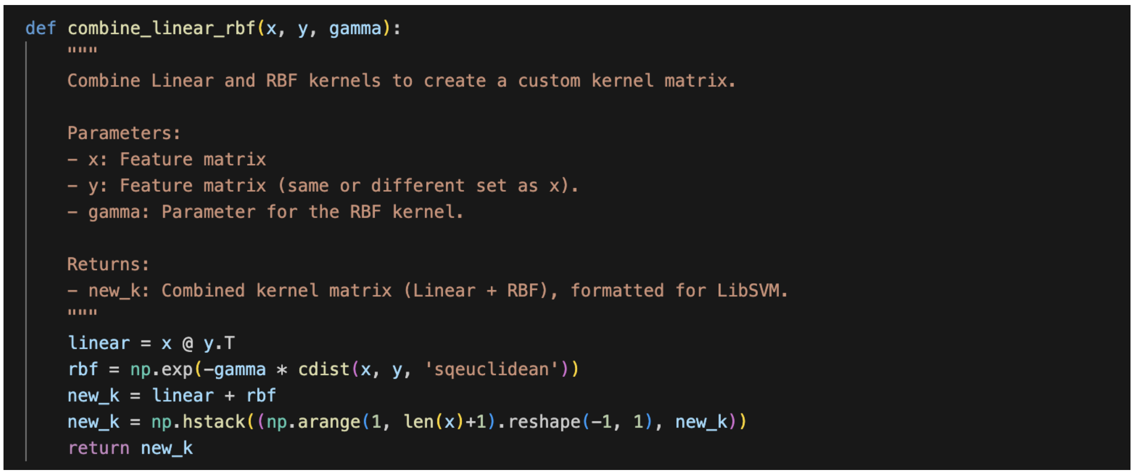
  The RBF kernel outperforms others in this experiment, showcasing its flexibility to model complex, non-linear relationships. Large C ensures tight margins, while small γ prevents overfitting by smoothing decision boundaries.

- **General Observation**

  Regardless of the kernel type, achieving high performance depends on selecting suitable parameters. Proper tuning of C, γ, and other kernel-specific parameters is essential for optimal results.

# Task3

## 1. Code

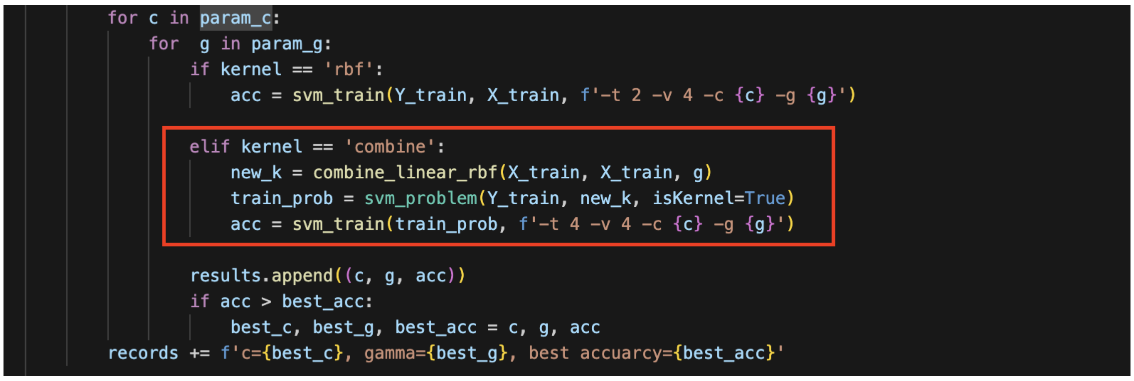


```python
def combine_linear_rbf(x, y, gamma):
    """
    Combine Linear and RBF kernels to create a custom kernel matrix.

    Parameters:
    - x: Feature matrix
    - y: Feature matrix (same or different set as x).
    - gamma: Parameter for the RBF kernel.

    Returns:
    - new_k: Combined kernel matrix (Linear + RBF), formatted for LibSVM.
    """
    linear = x @ y.T
    rbf = np.exp(-gamma * cdist(x, y, 'sqeuclidean'))
    new_k = linear + rbf
    new_k = np.hstack((np.arange(1, len(x)+1).reshape(-1, 1), new_k))
    return new_k
```

1. `combine_linear_rbf` combines the linear and RBF kernels:

$$K_{\text{combined}}(x, y) = K_{\text{linear}}(x, y) + K_{\text{RBF}}(x, y)$$

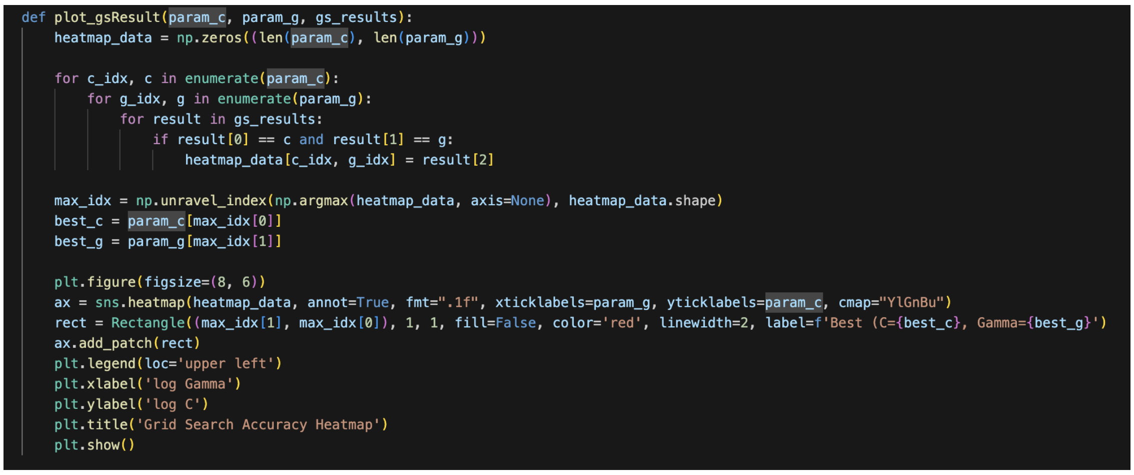
   The input `x`, `y` represents the feature matrix of the first and second data set, respectively, while `gamma` is the parameter of the RBF kernel. The function returns a combined kernel matrix formatted specifically for `svm_train`.



```python
        for c in param_c:
            for  g in param_g:
                if kernel == 'rbf':
                    acc = svm_train(Y_train, X_train, f'-t 2 -v 4 -c {c} -g {g}')

                elif kernel == 'combine':
                    new_k = combine_linear_rbf(X_train, X_train, g)
                    train_prob = svm_problem(Y_train, new_k, isKernel=True)
                    acc = svm_train(train_prob, f'-t 4 -v 4 -c {c} -g {g}')

                results.append((c, g, acc))
                if acc > best_acc:
                    best_c, best_g, best_acc = c, g, acc
        records += f'c={best_c}, gamma={best_g}, best accuarcy={best_acc}'
```

2. The `grid_search_cv` function is the same as in Task2, with the combined kernel highlighted by the red box. The combined kernel is passed into the `svm_problem` function using the `isKernel= True` argument. The SVM model is then trained with the custom kernel, and 4-fold cross-validation(`-v 4`) is performed to evaluate its performance. The function returns the best parameters and accuracy.

   The parameters C and γ are tuned via grid search. The parameter combination yielding the highest cross-validation accuracy is stored in `best_c`, `best_g` and `best_acc`.



```python
def plot_gsResult(param_c, param_g, gs_results):
    heatmap_data = np.zeros((len(param_c), len(param_g)))

    for c_idx, c in enumerate(param_c):
        for g_idx, g in enumerate(param_g):
            for result in gs_results:
                if result[0] == c and result[1] == g:
                    heatmap_data[c_idx, g_idx] = result[2]

    max_idx = np.unravel_index(np.argmax(heatmap_data, axis=None), heatmap_data.shape)
    best_c = param_c[max_idx[0]]
    best_g = param_g[max_idx[1]]

    plt.figure(figsize=(8, 6))
    ax = sns.heatmap(heatmap_data, annot=True, fmt=".1f", xticklabels=param_g, yticklabels=param_c, cmap="YlGnBu")
    rect = Rectangle((max_idx[1], max_idx[0]), 1, 1, fill=False, color='red', linewidth=2, label=f'Best (C={best_c}, Gamma={best_g}')
    ax.add_patch(rect)
    plt.legend(loc='upper left')
    plt.xlabel('log Gamma')
    plt.ylabel('log C')
    plt.title('Grid Search Accuracy Heatmap')
    plt.show()
```

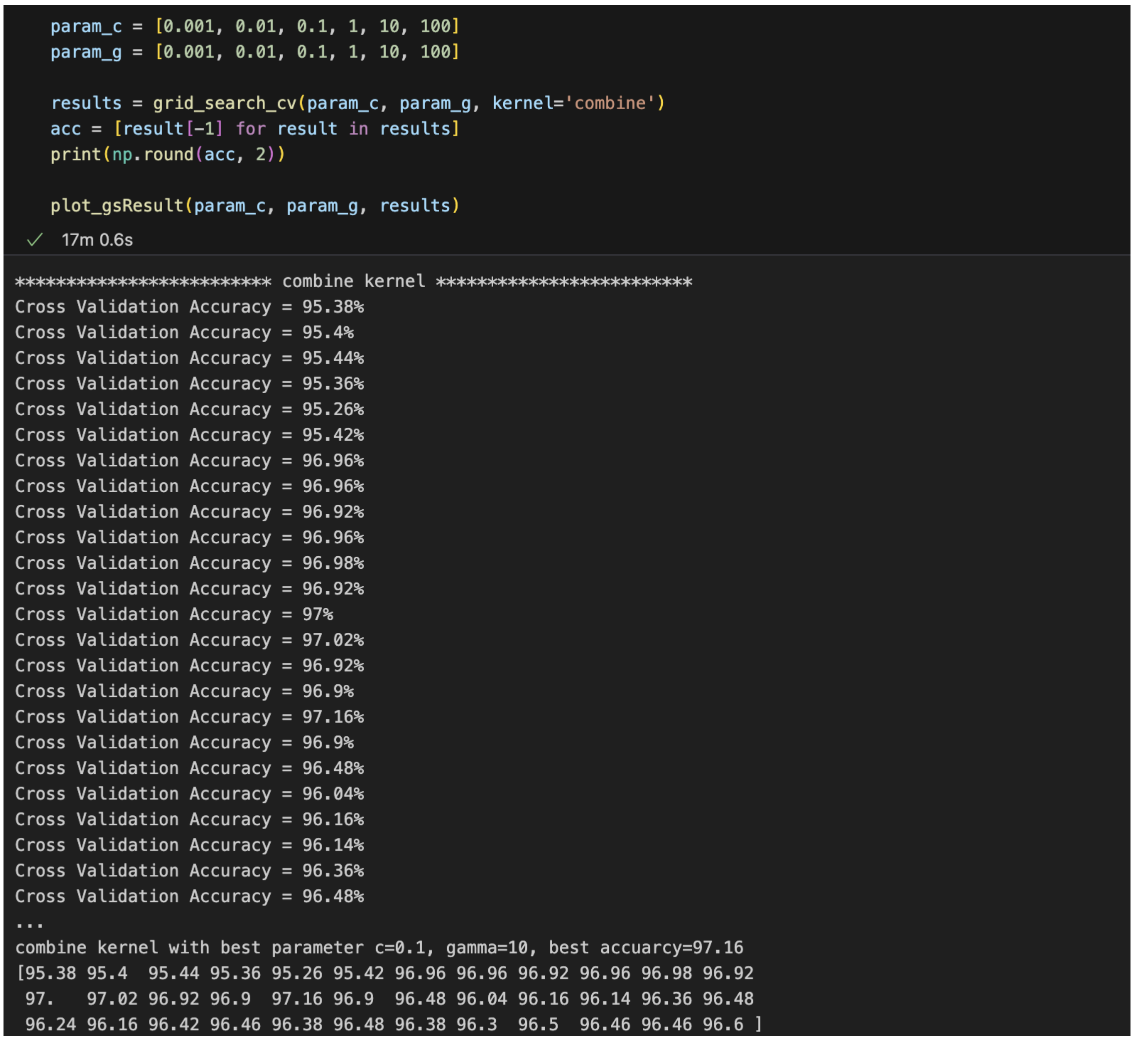
3. `plot_gsResult` : This function using a heatmap to visualize the results of grid search for hyperparameter tuning. The heatmap displays the relationship between the parameter C and $\gamma$, with color intensity representing cross-validation accuracy. The function also highlights the best-performing C-γ combination in the heatmap.

### Why Focus Only on C and $\gamma$?

Both C and $\gamma$ have the most significant and interpretable impact on accuracy for both RBF and Polynomial kernels. They directly control model flexibility and generalization.

Although d and r are essential for Polynomial kernels, their impact is secondary. Typically, d is set to a small integer (e.g., 2, 3) and r is chosen from a limited range (e.g., 0 or 1).

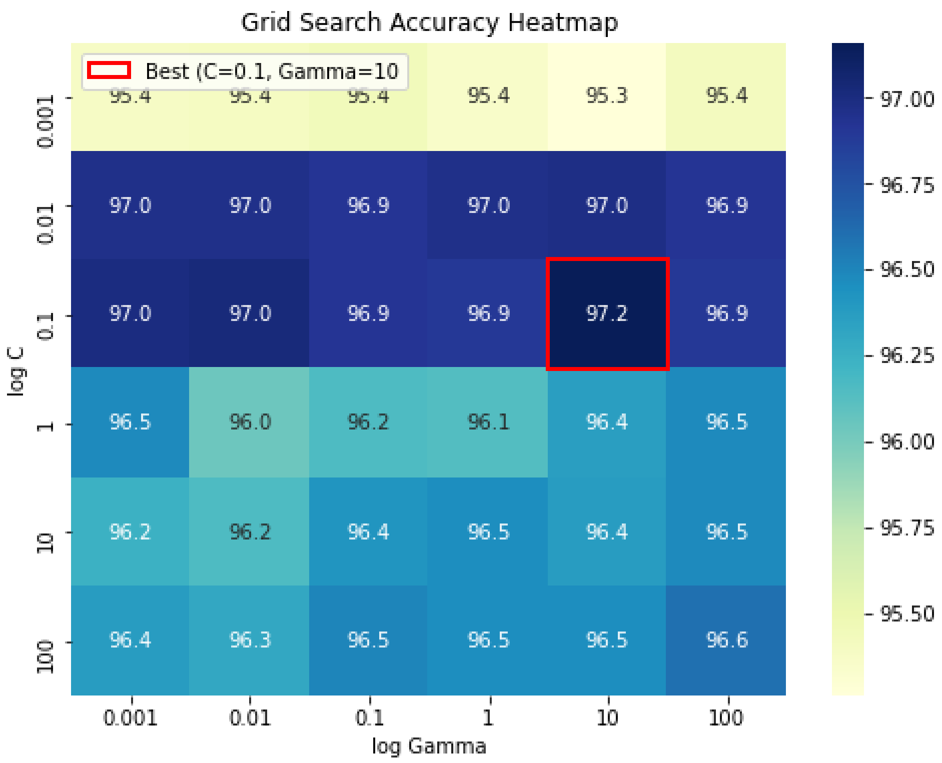
## 2. Experiments



```
param_c = [0.001, 0.01, 0.1, 1, 10, 100]
param_g = [0.001, 0.01, 0.1, 1, 10, 100]

results = grid_search_cv(param_c, param_g, kernel='combine')
acc = [result[-1] for result in results]
print(np.round(acc, 2))

plot_gsResult(param_c, param_g, results)
```

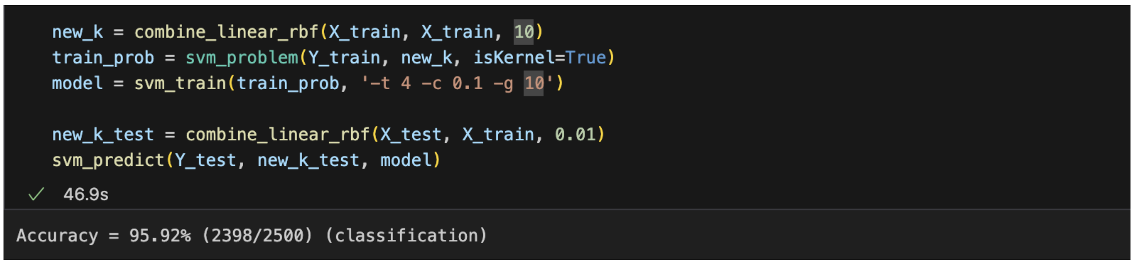
✓ 17m 0.6s

```
************************ combine kernel ************************
Cross Validation Accuracy = 95.38%
Cross Validation Accuracy = 95.4%
Cross Validation Accuracy = 95.44%
Cross Validation Accuracy = 95.36%
Cross Validation Accuracy = 95.26%
Cross Validation Accuracy = 95.42%
Cross Validation Accuracy = 96.96%
Cross Validation Accuracy = 96.96%
Cross Validation Accuracy = 96.92%
Cross Validation Accuracy = 96.96%
Cross Validation Accuracy = 96.98%
Cross Validation Accuracy = 96.92%
Cross Validation Accuracy = 97%
Cross Validation Accuracy = 97.02%
Cross Validation Accuracy = 96.92%
Cross Validation Accuracy = 96.9%
Cross Validation Accuracy = 97.16%
Cross Validation Accuracy = 96.9%
Cross Validation Accuracy = 96.48%
Cross Validation Accuracy = 96.04%
Cross Validation Accuracy = 96.16%
Cross Validation Accuracy = 96.14%
Cross Validation Accuracy = 96.36%
Cross Validation Accuracy = 96.48%
...
combine kernel with best parameter c=0.1, gamma=10, best accuarcy=97.16
[95.38 95.4  95.44 95.36 95.26 95.42 96.96 96.96 96.92 96.96 96.98 96.92
 97.   97.02 96.92 96.9  97.16 96.9  96.48 96.04 96.16 96.14 96.36 96.48
 96.24 96.16 96.42 96.46 96.38 96.48 96.38 96.3  96.5  96.46 96.46 96.6 ]
```



Set param_c = [0.001, 0.01, 0.1, 1, 10, 100], param_g = [0.001, 0.01, 0.1, 1, 10, 100] with best parameter c=0.1, gamma=10, best accuracy=97.16.

### After grid search



```
new_k = combine_linear_rbf(X_train, X_train, 10)
train_prob = svm_problem(Y_train, new_k, isKernel=True)
model = svm_train(train_prob, '-t 4 -c 0.1 -g 10')

new_k_test = combine_linear_rbf(X_test, X_train, 0.01)
svm_predict(Y_test, new_k_test, model)
```

✓ 46.9s

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

## 3. Observations and Discussion

The experiment results for the linear + RBF combined kernel show that it achieves a best accuracy of **97.16%** with C=0.1 and γ=10. The heatmap visualization highlights a stable performance across a range of C and γ, indicating that the combined kernel is robust to parameter changes.

This combined kernel effectively leverages the strengths of both linear and RBF kernels, producing results comparable to the standalone RBF kernel. While the RBF kernel slightly outperforms in peak accuracy, the combined kernel offers enhanced stability and consistent performance across different parameter combinations, making it a reliable option for scenarios where robustness is critical.

As with other kernels, proper tuning of C and γ is crucial to achieve optimal performance.