I. Gaussian Process

1. Code(20%)

```
import numpy as np

data = np.loadtxt('./data/input.data')

print(data.shape)
print(data[:5])

X_train = data[:, 0].reshape(-1, 1)
y_train = data[:, 1].reshape(-1, 1)
```

Using the `numpy.loadtxt()` function to load the training data from the `input.data` file. Each data point consists of (X_i, Y_i), where X_i represents the input location, and Y_i is the observed value at that location with added random noise.

Task 1

Rational Quadratic kernel

In this task, the Rational Quadratic kernel function is used as the similarity measure for Gaussian Process Regression. Its mathematical form is given as follows:

$$k(\mathbf{x}, \mathbf{x}') = \sigma^2 \left(1 + \frac{\|\mathbf{x} - \mathbf{x}'\|^2}{2\alpha\ell^2}\right)^{-\alpha}$$

```
import numpy as np
import matplotlib.pyplot as plt

# Rational Quadratic kernel
def rational_quadratic_kernel(X1, X2, length_scale=1.0, alpha=1.0,
variance=1.0):
    sqdist = np.sum(X1**2, axis=1).reshape(-1,1) + np.sum(X2**2, axis=1) - 2 *
np.dot(X1, X2.T)
    return variance * (1 + sqdist / (2 * alpha * length_scale**2)) ** (-alpha)
```

In the expression, $|\mathbf{x} - \mathbf{x}'|^2$ represents the squared distance between the input points $\mathbf{x}$ and $\mathbf{x}'$. The other three hyperparameters are interpreted as follows:

- ℓ (`length_scale`): Controls how distance between data points affects similarity. A larger value means the model assumes longer-range correlations (smoother behavior).

- α: Determines the overall shape of the kernel, controlling how quickly similarity decays with distance. As α increases, the kernel approaches the Radial Basis Function (RBF) kernel.
- σ^2 (`variance`): Controls the overall scale of output variations.

Gaussian Process Regression

What is a Gaussian Process?

A Gaussian Process (GP) is a non-parametric machine learning model commonly used for regression problems. For any set of input points, the corresponding outputs are assumed to follow a joint multivariate Gaussian distribution.

In other words, a Gaussian Process does not directly learn an explicit function, but rather learns a distribution over functions. By defining a similarity measure between inputs (typically through a kernel function, such as the Rational Quadratic kernel used in this task), it constructs a covariance structure over the outputs. This naturally allows the model to express uncertainty in unseen regions, making it highly suitable for applications that require confidence estimation, such as active learning or Bayesian optimization.

How Does a Gaussian Process Predict a Distribution?

In Gaussian Process Regression (GPR), given the training data $(X_{\text{train}}, y_{\text{train}})$, aim to infer the **predictive distribution** at new input locations X_{test}.

By applying the conditional properties of the multivariate Gaussian distribution, it will derive the following:

- **Predictive mean**: the expected output at each test point

$$\mu(X_*) = K(X_*, X)\, K(X, X)^{-1}\, y$$

- **Predictive covariance**: the uncertainty of the prediction at each test point

$$\Sigma(X_*) = K(X_*, X_*) - K(X_*, X)\, K(X, X)^{-1}\, K(X, X_*)^\top$$

$K(\cdot, \cdot)$ denotes the **kernel matrix** that computes similarity between inputs using the chosen kernel function.

```
# Gaussian Process Regression
def gaussian_process(X_train, y_train, X_test, length_scale=1.0, alpha=1.0,
variance=1.0, noise=1e-10):

    X_train = np.asarray(X_train).reshape(-1, 1)
    y_train = np.asarray(y_train).ravel()
    X_test = np.asarray(X_test).reshape(-1, 1)
```

```
    K = rational_quadratic_kernel(X_train, X_train, length_scale, alpha,
variance) + noise * np.eye(len(X_train))
    K_s = rational_quadratic_kernel(X_train, X_test, length_scale, alpha,
variance)
    K_ss = rational_quadratic_kernel(X_test, X_test, length_scale, alpha,
variance)  + noise

    K_inv = np.linalg.inv(K)

    mu = K_s.T @ K_inv @ y_train
    cov = K_ss - K_s.T @ K_inv @ K_s

    return mu.ravel(), cov
```

In the `gaussian_process()` function, I compute the following using the Rational Quadratic kernel:

- `K` : the kernel matrix between training points, denoted as $K(X, X)$, with added noise on the diagonal.
- `K_s` : the kernel matrix between training and test points, denoted as $K(X, X_*)$.
- `K_{ss}` : the kernel matrix between test points, denoted as $K(X_*, X_*)$.

Then, using the closed-form Gaussian Process formulas, compute:

- `mu` : the predicted mean at each test point, derived through linear algebra.
- `cov` : the predictive covariance matrix for the test points, which reflects the model's confidence (or uncertainty) across different input locations.

```
length_scale = 1.0
alpha = 1.0
variance = 1.0
noise = 1/5

X_pred = np.linspace(-60, 60, 1000).reshape(-1, 1) #Draw a line to represent the
mean of f in range [-60,60].
mu, cov = gaussian_process(X_train, y_train, X_pred,
                           length_scale=length_scale, alpha=alpha,
                           variance=variance, noise=noise)
```

Visualizing Gaussian Process Regression Results

The purpose of the following `matplotlib` code is to **visualize the predictions made by a Gaussian Process Regression model** using the Rational Quadratic kernel. The plot displays not only the predicted values but also the **uncertainty (confidence interval)** associated with each prediction.

```
import matplotlib.pyplot as plt
plt.figure(figsize=(10, 6))
plt.scatter(X_train, y_train, color='red', label='Training data')
plt.plot(X_pred, mu, color='blue', label='Mean prediction')
plt.fill_between(X_pred.ravel(),
                 mu - 1.96 * np.sqrt(np.diag(cov)),
                 mu + 1.96 * np.sqrt(np.diag(cov)),
                 alpha=0.2, label='95% confidence interval')
plt.title('Rational Quadratic Kernel Regression')
plt.xlabel('X')
plt.ylabel('Y')
plt.xlim(-60, 60)
plt.legend()
plt.grid(True)
plt.show()
```

- Mean prediction helps understand what the model believes is the underlying trend of the data.
- Confidence intervals provide insight into how certain or uncertain the model is in different regions:
  - The band is narrow near training points (high confidence).
  - The band is wide in regions far from training data (low confidence).

Task 2

Optimizing Kernel Parameters by Minimizing Negative Log Marginal Likelihood

In Gaussian Process Regression (GPR), the performance of the model heavily depends on the kernel hyperparameters — such as the length scale, variance, shape parameter α, and observation noise. These parameters control how smooth, flexible, and expressive the function space is.

To find the optimal combination of these parameters, maximize the marginal likelihood — a Bayesian criterion that balances model fit and complexity. The marginal likelihood evaluates how probable the observed data is under the current model configuration.

Log Marginal Likelihood Formula

The (log) marginal likelihood of the observed targets $\mathbf{y}$ under a Gaussian Process with kernel parameters $\boldsymbol{\theta}$ is given by:

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

Where:

- $\mathbf{C}_{\boldsymbol{\theta}} = K_\theta(X, X) + \sigma^2 I$ is the covariance matrix of the training data, combining the kernel and observation noise.

- The first term penalizes **model complexity** (the larger the determinant, the more flexible the model).
- The second term measures **data fit** (how well the model explains the data).
- The third term is a constant normalization factor.

```
def negative_log_likelihood(log_params, X_train, y_train):
    length_scale, alpha, variance, noise = np.exp(log_params)

    # Compute the covariance matrix with Rational Quadratic kernel
    K = rational_quadratic_kernel(X_train, X_train, length_scale, alpha,
variance) \
        + np.eye(len(X_train)) * noise

    try:
        # Cholesky decomposition for numerical stability
        L = np.linalg.cholesky(K)
    except np.linalg.LinAlgError:
        return np.inf  # Return infinity if matrix is not positive definite

    # Solve for alpha: α = K⁻¹y using Cholesky
    alpha_vec = np.linalg.solve(L.T, np.linalg.solve(L, y_train))

    # Log-determinant of K
    log_det = 2 * np.sum(np.log(np.diag(L)))

    # Compute negative log marginal likelihood
    nll = 0.5 * y_train.T @ alpha_vec + 0.5 * log_det + 0.5 * len(y_train) *
np.log(2 * np.pi)
    return nll
```

Using the `L-BFGS-B` optimization algorithm to minimize the NLL and find the optimal values of the hyperparameters. Since all parameters must be strictly positive, **optimizing in log-space** and apply `np.exp()` inside the function to recover actual values.

```
from scipy.optimize import minimize

init_log_params = np.log([1.0, 1.0, 1.0, 1/5])  # [length_scale, alpha,
variance, noise]

res = minimize(negative_log_likelihood,
               init_log_params,
               args=(X_train, y_train),
               method='L-BFGS-B')

opt_length_scale, opt_alpha, opt_variance, opt_noise = np.exp(res.x)
```

Once the optimal hyperparameters are found, using them in the Gaussian Process prediction function:

```
length_scale = opt_length_scale
alpha = opt_alpha
variance = opt_variance
noise = opt_noise

X_pred = np.linspace(-60, 60, 1000).reshape(-1, 1)

mu, cov = gaussian_process(X_train, y_train, X_pred,
                           length_scale=length_scale, alpha=alpha,
                           variance=variance, noise=noise)
```

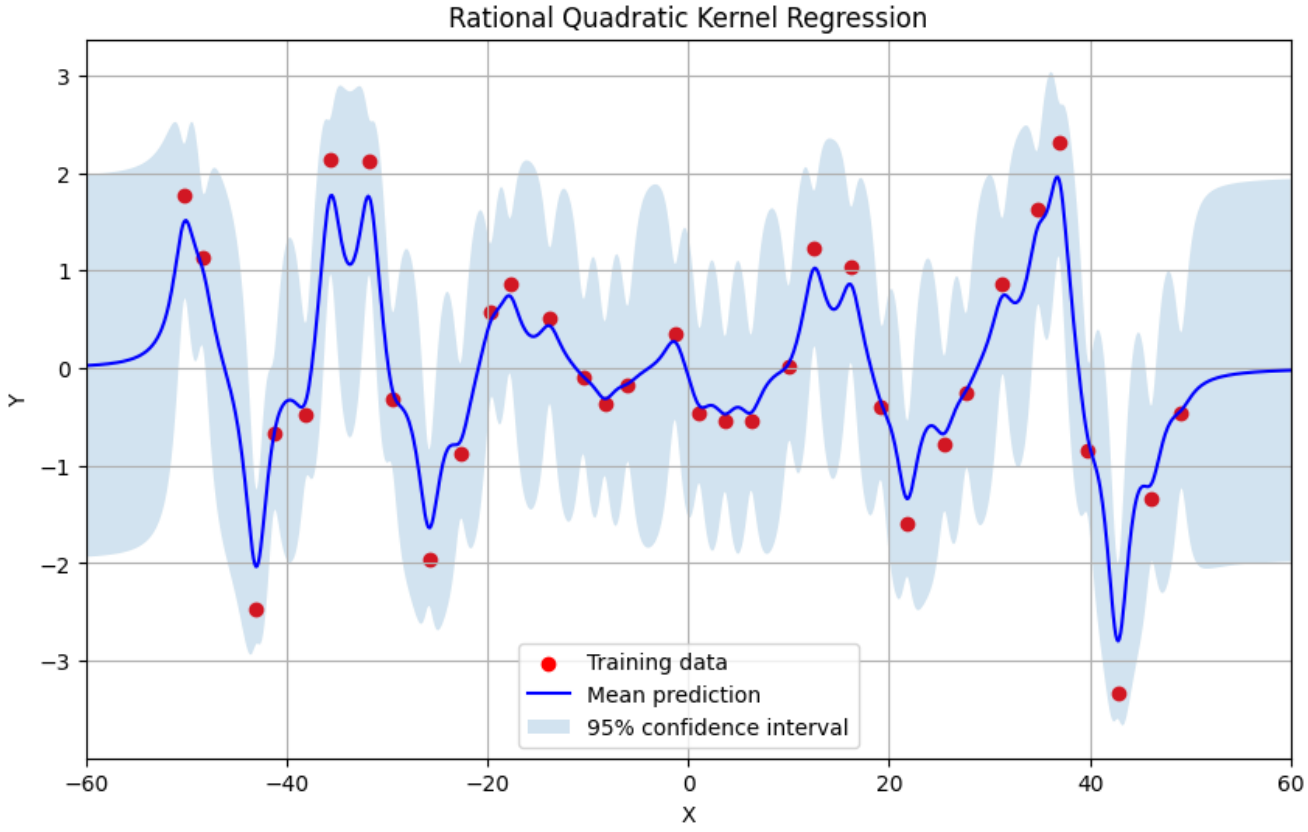
- `mu` : predicted mean values at test inputs.
- `cov` : predictive covariance matrix, used to derive confidence intervals.

2. Experiments (20%)

The prediction is performed over a dense grid of 1000 points between $[-60, 60]$.

The model uses the following initial hyperparameters

- **Length scale** (`length_scale`): 1.0
- **Shape parameter** (`alpha`): 1.0
- **Variance** (`variance`): 1.0
- **Noise level** (`noise`): 1/5

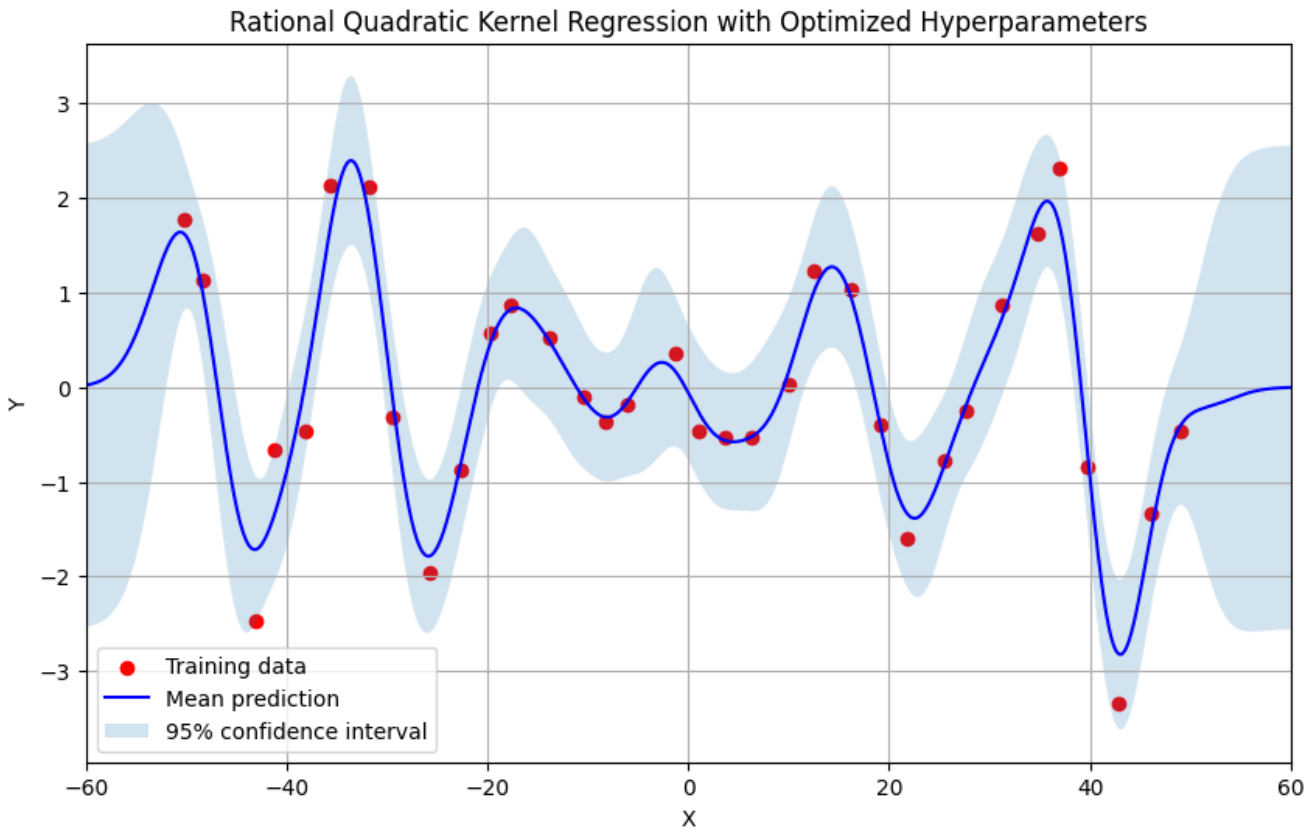


Plot 1

After optimizing the kernel parameters via negative log marginal likelihood, now use the `gaussian_process()` function to make predictions over a defined input range.

The following hyperparameters were obtained from the optimization process:

- **Length scale** (`length_scale`): 3.402
- **Shape parameter** (`alpha`): 4363.323
- **Variance** (`variance`): 1.699
- **Noise level** (`noise`): 0.2366



Plot 2

3. Observation and Discussion (10%)

	Initial Parameters	Optimized Parameters
Length scale (ℓ)	1.0	3.402
Alpha (α)	1.0	4363.323
Variance (σ^2)	1.0	1.699
Noise (σ_noise^2)	0.2 (i.e. 1/5)	0.2366

Plot 1: Initial Hyperparameters

- **Behavior**:
  The prediction line is very **wiggly and fluctuates sharply**, trying to closely fit the noise in the training data.
- **Confidence Region**:
  The **95% confidence interval** is wide and erratic in some regions, particularly far from training points. This suggests **high uncertainty** and possibly **overfitting** to noisy data.
- **Interpretation**:
  A small `length_scale` causes the model to assume **short-range dependencies**, leading it to react aggressively to local variations. Also, a small `alpha` gives the Rational Quadratic kernel heavier tails, making the model more sensitive to outliers.

Plot 2: Optimized Hyperparameters

- **Behavior**:
  The prediction curve is **smoother** and fits the **overall trend** of the data more realistically. It avoids overreacting to noise.
- **Confidence Region**:
  The shaded region representing the 95% confidence interval is more **stable and narrower**, especially near training points, reflecting increased confidence in those predictions.
- **Interpretation**:
  - A **larger** `length_scale` implies the model assumes that outputs vary more smoothly with inputs.
  - A **very large** `alpha` (~4363) causes the Rational Quadratic kernel to approximate an RBF kernel, resulting in **stable and localized similarity**.
  - The **optimized** `variance` and **noise level** allow the model to balance between fitting the signal and ignoring noise.

Conclusion

- The **initial model** attempts to fit every fluctuation, resulting in **overfitting** and high predictive variance.
- The **optimized model** produces a **more generalizable fit**, with a smoother mean and more trustworthy confidence intervals.
- This demonstrates the importance of **hyperparameter tuning**, especially via **marginal likelihood optimization**, in Gaussian Process models.

II. SVM on MNIST

1. Code (20%)

Task 1 SVM Classification with Different Kernels

The goal of this task is to train a Support Vector Machine (SVM) to find an optimal hyperplane that separates different classes in the training data by maximizing the margin between them. For non-linearly separable data, kernel functions are used to implicitly map the data into a higher-dimensional space where linear separation becomes possible.

Data Loading

```
import pandas as pd
X_train = pd.read_csv('./data/X_train.csv', header=None).values.tolist()
X_test = pd.read_csv('./data/X_test.csv', header=None).values.tolist()
y_train = pd.read_csv('./data/Y_train.csv', header=None).squeeze().tolist()
y_test = pd.read_csv('./data/Y_test.csv', header=None).squeeze().tolist()
```

Kernel Functions in SVM

In SVM, **kernel functions** allow us to compute dot products in high-dimensional feature spaces **without explicitly transforming the data**. This is known as the **kernel trick**.

① Linear Kernel

$$k(\mathbf{x}, \mathbf{x}') = \mathbf{x}^\top \mathbf{x}$$

- No nonlinear transformation — performs well when data is linearly separable.
- Fast and interpretable.
- **Parameter**:
  - `-c` : Penalty parameter C, balancing margin size and classification error.

② Polynomial Kernel

$$k(\mathbf{x}, \mathbf{x}') = (\gamma \cdot \mathbf{x}^\top \mathbf{x}' + r)^d$$

- Captures interactions between features.
- Can model curved decision boundaries.
- **Parameters**:
  - `-d` : Degree ddd of the polynomial.
  - `-g` : Gamma γ, controls scaling of input dot products.
  - `-r` : Coefficient rrr, adjusts balance between lower and higher-degree terms.
  - `-c` : Penalty parameter C.

③ RBF Kernel

$$k(\mathbf{x}, \mathbf{x}') = \exp\left(-\gamma \|\mathbf{x} - \mathbf{x}'\|^2\right)$$

- Most popular kernel for non-linear problems.
- Effective for complex, non-linear decision boundaries.
- **Parameters**:
  - `-g` : Gamma γ, controls how far each training point's influence reaches.
  - `-c` : Penalty parameter C.

```
from libsvm.svmutil import svm_train, svm_predict, svm_problem, svm_parameter
# kernel : 0 = linear, 1 = polynomial, 2 = RBF
kernel_names = {0: "linear", 1: "polynomial", 2: "rbf"}
kernel_results = {}

for k in [0, 1, 2]:
    prob = svm_problem(y_train, X_train)
    if k == 1:
        param = svm_parameter('-s 0 -t 1 -d 3 -c 1')
    else:
        param = svm_parameter(f'-s 0 -t {k} -c 1')
    model = svm_train(prob, param)
    pred_label, acc, _ = svm_predict(y_test, X_test, model, options='-q')
    kernel_results[kernel_names[k]] = acc[0]

print("Kernel results: ", kernel_results)
```

Task 2 Grid Search & Cross-Validation for Kernel Parameter Optimization

Implementing a **Grid Search** strategy combined with **5-fold Cross-Validation** to identify the best-performing hyperparameters for various kernel types in Support Vector Machines (SVMs), including a custom-defined mixed kernel.

What is Grid Search?

Grid Search is a systematic approach to hyperparameter tuning. It exhaustively tries all possible combinations from a user-defined parameter grid and evaluates each configuration. The model with the best performance (e.g., highest validation accuracy) is selected as the final model.

In this code, we define a search space using `param_grid` :

- **Linear kernel**: Search over penalty parameter `C` .
- **Polynomial kernel**: Search over `degree` , `C` , and `gamma` .
- **RBF kernel**: Search over `gamma` and `C` .
- **Mixed kernel** (Linear + RBF): Search over `gamma` and `C` .

What is Cross-Validation?

Cross-Validation is a method to estimate a model's ability to generalize to unseen data. It helps mitigate overfitting. In **k-fold cross-validation**, the dataset is split into `k` parts (folds). The model is trained on `k-1` folds and validated on the remaining fold. This process repeats `k` times with different validation folds, and the average performance is used as the metric.

In this implementation, use:

- `libsvm`'s built-in `-v 5` flag for automatic **5-fold CV** for linear, polynomial, and RBF kernels.
- A **manual CV function** for the **custom kernel**, because `libsvm` does not natively support cross-validation with precomputed kernels (`-t 4`).

This function performs **grid search with cross-validation** to find the best parameter set for a specified kernel.

Grid search function

1. **Enumerate all combinations** of parameters from the `param_grid`.
2. **For each combination**:
   - Construct an SVM training problem (`svm_problem`).
   - Define model settings (`svm_parameter`) including kernel type, penalty, and other hyperparameters.
   - Perform **5-fold cross-validation** using `-v 5`.
   - Store the accuracy and parameters if it's the best seen so far.
3. **Special handling for custom kernel in Grid Search Function**:
   - Compute the full Gram matrix using the `mixed_kernel(...)`.
   - Use `cross_validate_custom_kernel(...)` to simulate 5-fold CV manually.
4. **Return the best parameter set and its corresponding accuracy**.

```
def grid_search(X_train, y_train, X_test, y_test, kernel_type, param_grid):
    best_acc = 0
    best_params = None

    combinations = []

    if kernel_type == 'linear':
        for C in param_grid['C']:
            combinations.append([C])

    elif kernel_type == 'polynomial':
        for d in param_grid['degree']:
            for C in param_grid['C']:
                for g in param_grid['gamma']:
                    combinations.append([d, C, g])
```

```
    elif kernel_type == 'rbf':
        for g in param_grid['gamma']:
            for C in param_grid['C']:
                combinations.append([g, C])

    elif kernel_type == 'mixed':
        for C in param_grid['C']:
            for gamma in param_grid['gamma']:
                combinations.append([C, gamma])

    else:
        raise ValueError("Unsupported kernel type.")

    for params in combinations:
        if kernel_type == 'linear':
            C = params[0]
            param_str = f'-s 0 -t 0 -c {C} -v 5'
            prob = svm_problem(y_train, X_train)
            param = svm_parameter(param_str)
            acc = svm_train(prob, param, )  # acc is average CV accuracy
            print(f"Linear kernel: C={C}, acc={acc}")

        elif kernel_type == 'polynomial':
            degree, C, gamma = params
            param_str = f'-s 0 -t 1 -d {degree} -c {C} -g {gamma} -v 5'
            prob = svm_problem(y_train, X_train)
            param = svm_parameter(param_str)
            acc = svm_train(prob, param)
            print(f"Polynomial kernel: degree={degree}, C={C}, gamma={gamma},
acc={acc}")

        elif kernel_type == 'rbf':
            gamma, C = params
            param_str = f'-s 0 -t 2 -g {gamma} -c {C} -v 5'
            prob = svm_problem(y_train, X_train)
            param = svm_parameter(param_str)
            acc = svm_train(prob, param)
            print(f"RBF kernel: gamma={gamma}, C={C}, acc={acc}")

        elif kernel_type == 'mixed':
            C, gamma = params

            K_full = mixed_kernel(X_train, X_train, gamma)
            acc = cross_validate_custom_kernel(K_full, y_train, C, gamma)
            print("mixed kernel: C={}, gamma={}, acc={}".format(C, gamma, acc))

        else:
```

```
            continue

        if acc > best_acc:
            best_acc = acc
            best_params = params

    return best_params, best_acc
```

Task 3

1. **Combining Linear and RBF Kernels** To capture both global linear trends and local non-linear variations, we define a **custom mixed kernel** by adding a linear kernel and a radial basis function (RBF) kernel:

$$k_{\text{mixed}}(\mathbf{x}, \mathbf{x}') = \mathbf{x}^\top \mathbf{x}' + \exp\left(-\gamma \|\mathbf{x} - \mathbf{x}'\|^2\right)$$

```
def mixed_kernel(X, Y, gamma=0.5):
    X = np.array(X)
    Y = np.array(Y)

    linear_part = X @ Y.T
    X_norm = np.sum(X ** 2, axis=1).reshape(-1, 1)
    Y_norm = np.sum(Y ** 2, axis=1).reshape(1, -1)
    rbf_part = np.exp(-gamma * (X_norm + Y_norm - 2 * np.dot(X, Y.T)))

    return  linear_part +  rbf_part
```

2. How to Use Custom Kernel in LIBSVM

LIBSVM allows the use of **precomputed kernel matrices** through the `-t 4` option. However, for this to work, the kernel matrix must be manually formatted by prepending a **sample index (starting from 1)** as the first column of the matrix:

```
def add_index_column(K):
    n = K.shape[0]
    index = np.arange(1, n + 1, dtype=int).reshape(-1, 1)
    return np.hstack((index, K))
```

3. Manual Cross-Validation for Custom Kernels

Since LIBSVM's built-in cross-validation (`-v <n>`) **does not support precomputed kernels**, we manually implement **k-fold cross-validation**:

```python
def cross_validate_custom_kernel(K_full, y, C, gamma, num_folds=5):
    n = len(y)
    indices = list(range(n))
    np.random.seed(42)
    np.random.shuffle(indices)

    fold_size = n // num_folds
    folds = [indices[i * fold_size:(i + 1) * fold_size] for i in
range(num_folds)]
    if n % num_folds != 0:
        for i in range(n % num_folds):
            folds[i].append(indices[num_folds * fold_size + i])

    acc_list = []

    for i in range(num_folds):
        val_idx = folds[i]
        train_idx = [j for j in indices if j not in val_idx]

        K_train = K_full[np.ix_(train_idx, train_idx)]
        K_val = K_full[np.ix_(val_idx, train_idx)]

        K_train_indexed = add_index_column(K_train)
        K_val_indexed = add_index_column(K_val)

        y_train = [y[j] for j in train_idx]
        y_val = [y[j] for j in val_idx]

        # prob = svm_problem(y_train, K_train_indexed.tolist())
        # param = svm_parameter(f'-t 4 -c {C} -q')
        # model = svm_train(prob, param)
        model = svm_train(y_train, K_train_indexed.tolist(), f'-t 4 -c {C} -g
{gamma} -q ')

        _, acc, _ = svm_predict(y_val, K_val_indexed.tolist(), model, options='-
q')
        acc_list.append(acc[0])

    return np.mean(acc_list)
```

2. Experiments (20%)

Task 1 - Kernel Comparison (Default Parameters)

Kernel Type	Accuracy

Kernel Type	Accuracy
linear	95.08%
Polynomial	34.68%
RBF	95.32%

- **Linear and RBF kernels** perform very well by default, suggesting that the dataset is either linearly separable or has local structures that RBF can capture.
- **Polynomial kernel (default)** performs poorly (34.68%), likely due to unsuitable default hyperparameters (e.g., high-degree polynomial or unscaled features). This shows that polynomial kernels are **very sensitive to parameter settings**.

Task 2

Linear Kernel

C	Accuracy
0.1	96.96%
1	96.3%
10	96.1%

- Smaller `C = 0.1` gave the best result, meaning the model benefits from **wider margin** and **less emphasis on misclassification**.
- Higher `C` values may lead to overfitting on training noise.

Polynomial Kernel

degree	c	gamma	Accuracy
2	0.1	0.01	94.78%
2	0.1	0.1	98.18%
2	1	0.01	97.66%
2	1	0.1	98.2%
2	10	0.01	98.04%
2	10	0.1	97.98%
3	0.1	0.01	90.14%
3	0.1	0.1	97.92%

degree	c	gamma	Accuracy
3	1	0.01	96.54%
3	1	0.1	97.54%
3	10	0.01	97.72%
3	10	0.1	97.6%

- After tuning, **polynomial kernel becomes the best performer** (98.2%).
- Low-degree polynomials (especially degree 2) with well-scaled gamma values work well, capturing **quadratic interactions** in data without overfitting.
- Degree 3 gives slightly lower performance, suggesting additional complexity doesn't help much.

RBF kernel

gamma	c	Accuracy
0.01	0.1	96.46%
0.01	1	97.56%
0.01	10	98.12%
0.1	0.1	53.34%
0.1	1	92.02%
0.1	10	92.4%

- RBF performs best with **small gamma (0.01)** and **large C (10)**:
  - Low gamma ⇒ smooth, wide influence per point.
  - High C ⇒ allows the model to correct more misclassified points.
- High gamma (0.1) leads to overfitting and **poor generalization** (e.g., 53.34% with C=0.1).

Task 3

Combines the strengths of both:

- **Linear component**: captures global, linear trends.
- **RBF component**: captures local, nonlinear variations.

c	gamma	Accuracy
0.1	0.01	97.16

c	gamma	Accuracy
0.1	0.1	97.16
1	0.01	96.62
1	0.1	96.72
10	0.01	96.62
10	0.1	96.72

- The **best accuracy (97.16%)** is achieved when **C = 0.1** with **either gamma = 0.01 or 0.1**, indicating the model benefits from:
  - **Lower regularization (C)** → allows for a larger margin and better generalization.
  - The **mixed kernel is robust to gamma** within this tested range.
- Larger values of **C = 1 or 10** slightly reduce performance, suggesting a risk of **overfitting** due to higher penalty on misclassified points.

3. Observation and Discussion (10%)

Method	Kernel	Best Accuracy	Parameters (C, γ, d)
Task 1: Default Kernels	Linear	95.08%	default `C=1`
	Polynomial	34.68%	default `C=1`, `d=3`
	RBF	95.32%	default `C=1`, `γ=1/num_feat`
Task 2: Tuned Kernels	Linear	96.96%	`C=0.1`
	Polynomial	**98.20%**	`C=1`, `d=2`, `γ=0.1`
	RBF	98.12%	`C=10`, `γ=0.01`
Task 3: Mixed Kernel	Linear + RBF	97.16%	`C=0.1`, `γ=0.01` or `0.1`

The **tuned polynomial kernel** achieves the highest accuracy (98.2%), likely due to its ability to model complex nonlinear patterns, but only when hyperparameters such as degree and gamma are properly selected. The **RBF kernel** performs nearly as well (98.12%) and demonstrates strong generalization with relatively robust performance across different parameter settings. The **custom mixed kernel**, which combines linear and RBF components, also performs strongly (97.16%) and shows robustness to gamma values, making it a flexible option for capturing both global and local patterns. In contrast, the **default polynomial kernel** performs poorly (34.68%) without parameter tuning, emphasizing the critical role of hyperparameter selection in kernel-based models.