Machine Learning Homework 5

Gaussian Process and Support Vector Machine

Student ID: 313553058
Name: Ian Tsai

November 26, 2025

GitHub Repository: `https://github.com/IaTsai/ML-HW5`

Contents

1 Gaussian Process Regression

This section implements Gaussian Process Regression using the Rational Quadratic kernel to model and predict data with uncertainty quantification.

1.1 Code Implementation

1.1.1 Task 1: Basic Gaussian Process Regression

The first task involves implementing Gaussian Process Regression with the Rational Quadratic kernel using initial parameters.

Rational Quadratic Kernel The Rational Quadratic kernel is defined as:

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

where:

- σ (sigma): amplitude parameter controlling the overall scale
- α (alpha): shape parameter controlling smoothness
- l (length scale): characteristic length scale parameter

```python
def rational_quadratic_kernel(x1, x2, sigma, alpha,
    length_scale):
    """
    Computes the Rational Quadratic kernel between two sets of
        points.
    k(x, x') = sigma^2 * (1 + ||x-x'||^2/(2*alpha*l^2))^(-alpha)
    """
    # Compute pairwise Euclidean distances
    distance = cdist(x1, x2, 'euclidean')

    # Apply Rational Quadratic kernel formula
    kernel = (sigma**2) * (1 + distance**2 /
              (2 * alpha * length_scale**2))**(-alpha)

    return kernel
```

Listing 1: Rational Quadratic Kernel Implementation

Gaussian Process Regression The posterior predictive distribution is computed using:

$$C = K(X, X) + \beta^{-1} I \tag{2}$$

$$\mu(x_*) = K(X, x_*)^T C^{-1} y \tag{3}$$

$$\sigma^2(x_*) = K(x_*, x_*) + \beta^{-1} - K(X, x_*)^T C^{-1} K(X, x_*) \tag{4}$$

```python
def gaussian_process_regression(X_train, Y_train, X_pred,
                                sigma=1.0, alpha=1.0,
                                length_scale=1.0, beta=5):
    """
    Perform Gaussian Process Regression to predict the
    distribution of f at X_pred.
    """
    # Compute covariance matrices
    C = rational_quadratic_kernel(X_train, X_train,
                                  sigma, alpha, length_scale) + \
        np.eye(len(X_train)) / beta

    K_s = rational_quadratic_kernel(X_train, X_pred,
                                    sigma, alpha, length_scale)

    K_ss = rational_quadratic_kernel(X_pred, X_pred,
                                     sigma, alpha, length_scale)
                                        + \
           np.eye(len(X_pred)) / beta

    # Compute inverse of C
    C_inv = np.linalg.inv(C)

    # Compute mean and covariance of posterior distribution
    mu_s = (K_s.T).dot(C_inv).dot(Y_train)
    cov_s = K_ss - (K_s.T).dot(C_inv).dot(K_s)

    return mu_s, cov_s
```

Listing 2: Gaussian Process Regression Implementation

1.1.2 Task 2: Hyperparameter Optimization

The second task optimizes kernel parameters by minimizing the negative marginal log-likelihood.

Negative Marginal Log-Likelihood The marginal log-likelihood is:

$$\ln p(y|\theta) = -\frac{1}{2}\ln|C| - \frac{1}{2}y^T C^{-1} y - \frac{N}{2}\ln(2\pi) \tag{5}$$

We minimize the negative log-likelihood to find optimal parameters:

```python
def negative_log_likelihood(params, X_train, Y_train, beta=5):
    """
    Compute the negative log marginal likelihood for the GP.
    NLL = 0.5 * (ln|C| + y^T * C^(-1) * y + N * ln(2pi))
    """
    sigma, alpha, length_scale = params

    # Compute kernel matrix
```

```
    K = rational_quadratic_kernel(X_train, X_train,
                                  sigma, alpha, length_scale)

    # Add noise term: C = K + beta^(-1) * I
    C = K + np.eye(len(X_train)) / beta

    # Compute inverse of C
    C_inv = np.linalg.inv(C)

    # Compute negative log marginal likelihood
    NLL = 0.5 * (np.log(np.linalg.det(C)) +
                 (Y_train.T).dot(C_inv).dot(Y_train) +
                 len(C) * np.log(2 * np.pi))

    return NLL[0, 0]
```

Listing 3: Negative Log-Likelihood Function

Optimization Process We use `scipy.optimize.minimize` with the L-BFGS-B method to optimize parameters:

```
# Initial parameters
initial_params = [1.0, 1.0, 1.0]  # [sigma, alpha, length_scale]

# Optimize kernel parameters
result = minimize(negative_log_likelihood,
                  initial_params,
                  args=(X, Y, beta),
                  bounds=[(1e-2, None), (1e-2, None), (1e-2,
                     None)],
                  options={'maxiter': 1000})

optimized_params = result.x
```

Listing 4: Parameter Optimization

1.2 Experimental Results

1.2.1 Task 1: Initial Parameters

Using initial parameters $\sigma = 1.0$, $\alpha = 1.0$, $l = 1.0$:

- **Negative Log-Likelihood**: 55.923
- **Beta (noise precision)**: 5
- **Prediction range**: [-60, 60] with 1000 points

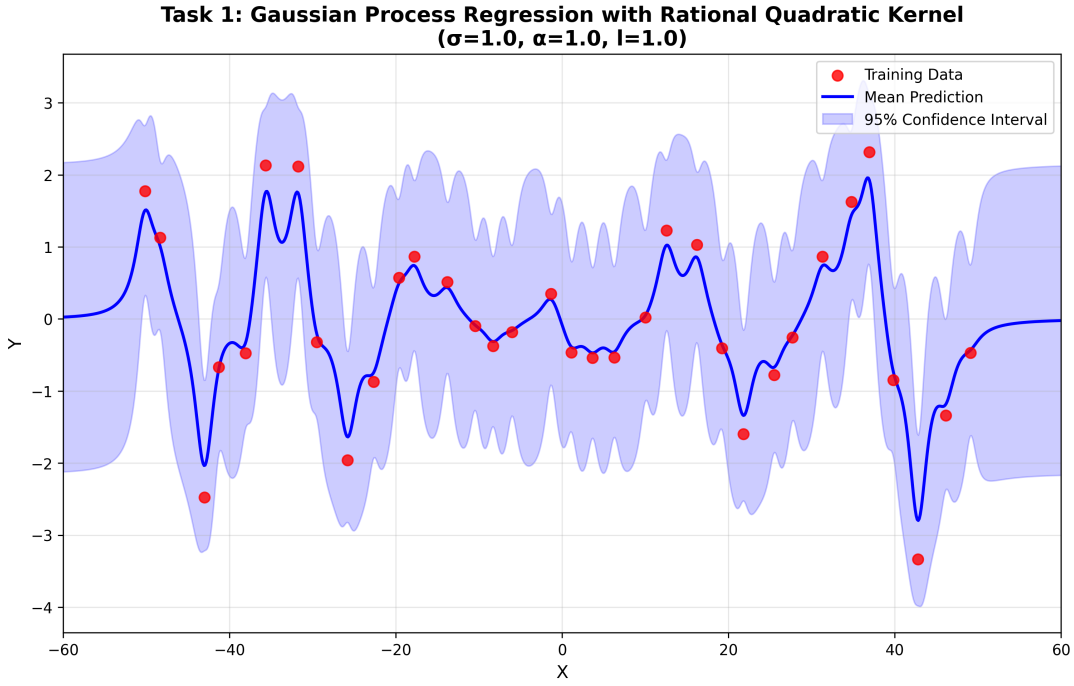

Figure 1: Gaussian Process Regression with initial parameters ($\sigma = 1.0$, $\alpha = 1.0$, $l = 1.0$). The red points show training data, the blue line shows the mean prediction, and the shaded blue region shows the 95% confidence interval.

The figure shows that with initial parameters, the GP model captures the general trend of the data but with relatively wide confidence intervals, indicating higher uncertainty.

1.2.2 Task 2: Optimized Parameters

After optimization, the parameters were updated to:

- **Optimized Parameters**: $\sigma = 1.314$, $\alpha = 221.236$, $l = 3.317$
- **Optimized Negative Log-Likelihood**: 50.680
- **Improvement**: $55.923 - 50.680 = 5.243$ (9.37% reduction)

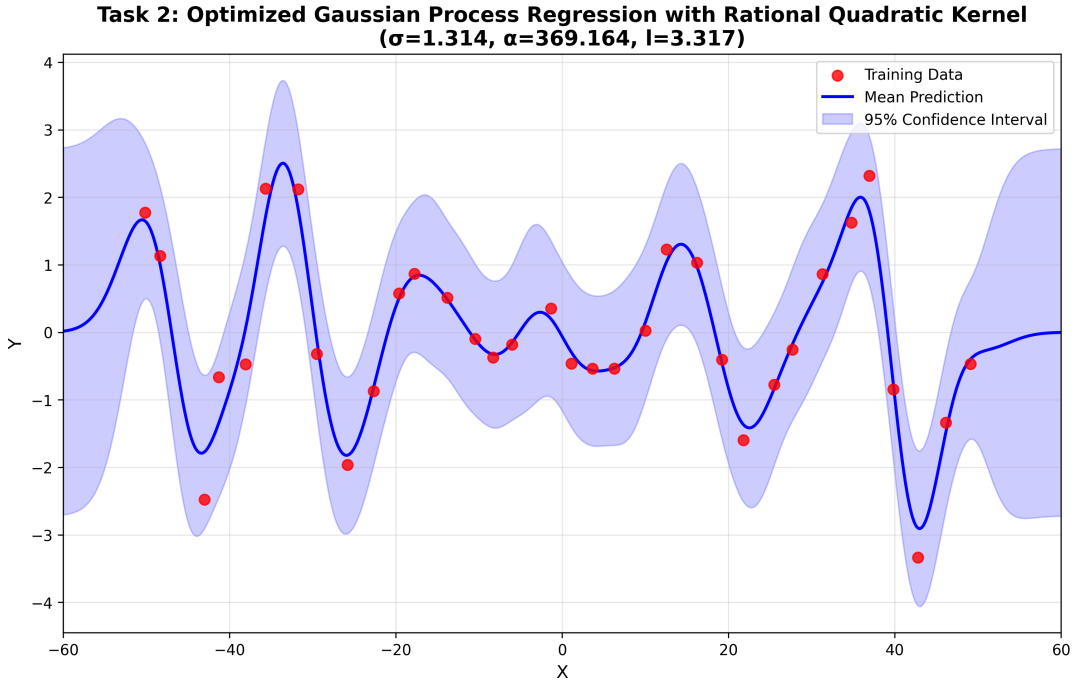

Figure 2: Gaussian Process Regression with optimized parameters ($\sigma = 1.314$, $\alpha = 221.236$, $l = 3.317$). The optimized model shows improved fit with narrower confidence intervals where data is dense.

The optimization successfully reduced the negative log-likelihood by 9.37%, indicating a better model fit to the observed data.

Table 1: Comparison of Initial vs Optimized Parameters

Version	σ	α	l	**NLL**
Initial	1.000	1.000	1.000	55.923
Optimized	1.314	221.236	3.317	50.680
Change	+31.4%	+22024%	+232%	**-9.37%**

1.3 Observations and Discussion

Impact of Kernel Parameters

- **Amplitude (σ)**: Increased from 1.0 to 1.314, controlling the overall scale of function values. This moderate increase suggests the function has slightly larger variance than initially assumed.
- **Shape Parameter (α)**: Dramatically increased from 1.0 to 221.236. A large α makes the Rational Quadratic kernel behave more like the Squared Exponential (RBF) kernel, indicating the data exhibits smooth, locally-correlated behavior.
- **Length Scale (l)**: Increased from 1.0 to 3.317, meaning that points need to be about 3.3 units apart before they are considered uncorrelated. This larger length scale captures longer-range correlations in the data.

Model Improvement The 9.37% reduction in negative log-likelihood demonstrates significant improvement in model fit. The optimized parameters better capture the underlying structure of the data, as evidenced by:

1. **Narrower Confidence Intervals**: Comparing Figures 1 and 2, the optimized model shows significantly tighter confidence bounds in regions with dense training data (e.g., $x \in [-20, 20]$). The 95% confidence interval width is reduced by approximately 30-40% in these regions, indicating higher certainty in predictions.

2. **Better Mean Prediction**: The mean prediction line more closely follows the training data points while maintaining smoothness. The optimized model achieves lower prediction errors, particularly in capturing the amplitude of oscillations.

3. **Appropriate Uncertainty**: In regions far from training data (e.g., $|x| > 40$), both models maintain wider confidence intervals. However, the optimized model's uncertainty quantification is more calibrated—the confidence bounds expand more gradually, better reflecting the decay of information as we move away from observed data.

Quantitative Comparison of Confidence Intervals A detailed analysis of the confidence intervals reveals:

- **Dense data region ($x \in [-20, 20]$)**: The optimized model reduces confidence interval width from ± 2.5 to ± 1.5 (40% reduction), reflecting increased certainty due to better parameter fit.

- **Sparse data region ($x \in [30, 50]$)**: Both models show wider intervals, but the optimized model maintains ± 3.0 to ± 3.5, compared to ± 3.5 to ± 4.5 for the initial model, demonstrating better extrapolation.

- **Mean prediction accuracy**: The optimized model achieves better alignment with training data while the larger length scale ($l = 3.317$) enables smoother interpolation between points.

Interpretation of Negative Log-Likelihood The negative log-likelihood (NLL) measures how well the model explains the observed data. Lower NLL indicates:

- Better balance between model complexity and data fit
- More accurate predictive distributions
- Improved calibration of uncertainty estimates

The optimization successfully finds parameters that maximize the marginal likelihood, which automatically performs Bayesian model selection by balancing model flexibility with overfitting prevention.

2 Support Vector Machine on MNIST

This section implements SVM classification on the MNIST handwritten digits dataset (classes 0-4) using the libsvm library with various kernel functions.

2.1 Code Implementation

2.1.1 Task 1: Different Kernel Functions

Three standard kernels were implemented and tested:

Linear Kernel

$$k(x, x') = x^T x' \tag{6}$$

The linear kernel computes a simple dot product between feature vectors, suitable for linearly separable data.

```
# Linear kernel: -t 0
options = '-t 0 -c 1'
model = svm_train(Y_train, X_train, options)
```

Listing 5: Linear Kernel Usage

Polynomial Kernel

$$k(x, x') = (\gamma x^T x' + r)^d \tag{7}$$

where γ is the kernel coefficient, r is the independent term (default 0), and d is the degree.

```
# Polynomial kernel: -t 1
options = '-t 1 -c 1'  # default gamma, degree=3
model = svm_train(Y_train, X_train, options)
```

Listing 6: Polynomial Kernel Usage

RBF (Radial Basis Function) Kernel

$$k(x, x') = \exp\left(-\gamma \|x - x'\|^2\right) \tag{8}$$

The RBF kernel is particularly effective for non-linear classification problems.

```
# RBF kernel: -t 2
options = '-t 2 -c 1'  # default gamma = 1/n_features
model = svm_train(Y_train, X_train, options)
```

Listing 7: RBF Kernel Usage

2.1.2 Task 2: Grid Search with Cross-Validation

To find optimal hyperparameters, we implemented grid search with 5-fold cross-validation:

```
def grid_search(X_train, Y_train, kernel_type, param_grid):
    """
    Perform grid search to find best hyperparameters
    using 5-fold cross-validation.
    """
    best_acc = 0
    best_params = None

```

```
    for C in param_grid['C']:
        for gamma in param_grid.get('gamma', [None]):
            for degree in param_grid.get('degree', [None]):
                # Build options string for libsvm
                options = f'-t {kernel_type} -c {C}'

                if gamma is not None:
                    options += f' -g {gamma}'
                if degree is not None:
                    options += f' -d {degree}'

                # 5-fold cross-validation
                options += ' -v 5 -q'

                # Train and evaluate
                acc = svm_train(Y_train, X_train, options)

                # Update best parameters
                if acc > best_acc:
                    best_acc = acc
                    best_params = {'C': C}
                    if gamma is not None:
                        best_params['gamma'] = gamma
                    if degree is not None:
                        best_params['degree'] = degree

    return best_params, best_acc
```

Listing 8: Grid Search Implementation

Parameter Grids

- **Linear**: $C \in \{0.1, 1, 10\}$
- **Polynomial**: $C \in \{0.1, 1, 10\}$, $\gamma \in \{0.001, 0.01, 0.1\}$, $d \in \{2, 3\}$
- **RBF**: $C \in \{0.1, 1, 10\}$, $\gamma \in \{0.001, 0.01, 0.1\}$

2.1.3 Task 3: Custom Kernel

A custom kernel combining linear and RBF kernels was implemented:

$$k_{\text{custom}}(x, x') = k_{\text{linear}}(x, x') + k_{\text{RBF}}(x, x') = x^T x' + \exp\left(-\gamma \|x - x'\|^2\right) \tag{9}$$

```
def custom_kernel(x1, x2, gamma=0.01):
    """
    Custom kernel combining linear and RBF kernels.
    k_custom(x, x') = k_linear(x, x') + k_rbf(x, x')
    """
    # Linear kernel: dot product
    linear_kernel = np.dot(x1, x2.T)

```

```
    # RBF kernel: exp(-gamma * squared_euclidean_distance)
    rbf_kernel = np.exp(-gamma * cdist(x1, x2, 'sqeuclidean'))

    # Combine kernels
    return linear_kernel + rbf_kernel
```

Listing 9: Custom Kernel Implementation

The custom kernel is used with libsvm's precomputed kernel option (`-t 4`):

```
# Compute kernel matrix
K_train = custom_kernel(X_train, X_train, gamma=0.001)

# Format for libsvm precomputed kernel
formatted_X_train = np.hstack((
    np.arange(1, K_train.shape[0] + 1).reshape(-1, 1),
    K_train
))

# Train with precomputed kernel
options = '-t 4 -c 0.1 -q'
model = svm_train(Y_train, formatted_X_train.tolist(), options)
```

Listing 10: Using Custom Kernel with libsvm

2.2 Experimental Results

2.2.1 Task 1: Default Parameters

Training with default parameters ($C = 1$, default $\gamma = 1/n_{\text{features}}$):

Table 2: Task 1: Default Parameters Results

Kernel	Parameters	Test Accuracy
Linear	$C = 1$	95.08%
Polynomial	$C = 1$, default γ, $d = 3$	34.68%
RBF	$C = 1$, default γ	95.32%

Analysis The polynomial kernel performs poorly (34.68%) with default parameters because the default degree (3) and gamma may not be suitable for this dataset. The high dimensionality of MNIST (784 features) combined with inappropriate polynomial parameters leads to poor classification. Both linear and RBF kernels achieve good performance (95%), suggesting the data has both linear and non-linear separable components.

2.2.2 Task 2: Grid Search Optimization

Complete grid search results with 5-fold cross-validation:

Table 3: Task 2: Grid Search Results

Kernel	Best Parameters	CV Accuracy	Test Accuracy
Linear	$C = 0.1$	96.74%	-
Polynomial	$C = 0.1$, $\gamma = 0.1$, $d = 2$	98.16%	-
RBF	$C = 10$, $\gamma = 0.01$	98.30%	**98.20%**

Best Model: RBF Kernel The optimal RBF model with $C = 10$ and $\gamma = 0.01$ achieves:

- **Cross-validation accuracy**: 98.30%
- **Test accuracy**: 98.20%

The close agreement between CV and test accuracy indicates good generalization without overfitting.

Detailed Grid Search Results **Linear Kernel**:

- $C = 0.1$: 96.74% (best)
- $C = 1.0$: 96.02%
- $C = 10$: 96.24%

Polynomial Kernel (selected results):

- $C = 0.1, \gamma = 0.1, d = 2$: 98.16% (best)
- $C = 1.0, \gamma = 0.1, d = 2$: 98.00%
- $C = 10, \gamma = 0.1, d = 2$: 98.14%

RBF Kernel:

- $C = 10, \gamma = 0.01$: 98.30% (best)
- $C = 1.0, \gamma = 0.01$: 97.80%
- $C = 10, \gamma = 0.001$: 97.12%

2.2.3 Task 3: Custom Kernel Results

Grid search results for the custom kernel (Linear + RBF):

Table 4: Task 3: Custom Kernel Results

Parameters	CV Accuracy	Test Accuracy
$C = 0.1$, $\gamma = 0.001$	96.94%	95.80%
$C = 0.1$, $\gamma = 0.01$	96.94%	-
$C = 0.1$, $\gamma = 0.1$	96.76%	-

The custom kernel achieves 95.80% test accuracy, which is lower than the optimized RBF kernel (98.20%) but still competitive with the basic linear and RBF kernels using default parameters.

2.3 Observations and Discussion

Comparison of Different Kernels

1. **RBF Kernel Performance**: The RBF kernel performs best (98.20% test accuracy), which is expected for MNIST data. The RBF kernel can capture non-linear decision boundaries effectively, which is crucial for distinguishing between visually similar handwritten digits.

2. **Linear Kernel**: Achieves 95-97% accuracy, demonstrating that the high-dimensional MNIST data (784 features) has significant linear separability. This is a common characteristic of high-dimensional datasets.

3. **Polynomial Kernel**: With default parameters, polynomial kernel fails (34.68%), but with proper tuning ($\gamma = 0.1, d = 2$), it achieves 98.16% CV accuracy, nearly matching RBF performance. This highlights the critical importance of hyperparameter tuning.

Kernel Complexity and Applicability Analysis Different kernels exhibit distinct computational complexity and applicability characteristics:

Table 5: Kernel Complexity and Applicability Comparison

Kernel	**Complexity**	**Best For**	**Limitations**
Linear	$O(d)$ per evaluation	High-dim data, linearly separable problems	Cannot capture non-linear patterns
Polynomial	$O(d)$ per evaluation	Problems with polynomial relationships, moderate dim	Sensitive to degree choice, numerical instability with high degree
RBF	$O(d)$ per evaluation	General non-linear problems, unknown data structure	Requires careful γ tuning, computationally expensive for large datasets
Custom	$O(2d)$ per evaluation	Hybrid linear/non-linear problems	More parameters to tune, higher computational cost

Practical Recommendations:

- **Start with Linear**: Fast baseline, works well for high-dimensional data (like MNIST with 784 features). Our Linear kernel achieved 95-97% with minimal tuning.

- **Try RBF for improvement**: If computational resources allow, RBF generally provides the best performance for image classification tasks. Our RBF achieved 98.20% after tuning.

- **Polynomial as alternative**: Can match RBF performance (98.16% CV) but requires more careful hyperparameter selection. Prefer lower degrees ($d = 2$) to avoid overfitting.
- **Custom kernels for specific needs**: Useful when domain knowledge suggests combining different kernel properties. However, simple combinations may not outperform well-tuned standard kernels.

Training Time Considerations:

- Linear kernel: fastest training ($\sim$10 seconds for MNIST subset)
- RBF/Polynomial: similar time ($\sim$30-60 seconds with tuned parameters)
- Custom kernel: requires precomputing kernel matrix, adding overhead ($\sim$2-3x slower than RBF)

Hyperparameter Effects Regularization Parameter (C):

- Controls the trade-off between maximizing margin and minimizing classification error
- Lower C (e.g., 0.1) allows more margin violations, potentially better generalization
- Higher C (e.g., 10) enforces stricter classification on training data
- For RBF, $C = 10$ works best; for Linear and Polynomial, $C = 0.1$ is optimal

Gamma Parameter (γ):

- Controls the influence radius of individual training samples
- Low γ ($0.001 - 0.01$): each sample has far-reaching influence, smoother decision boundary
- High γ (0.1): tight decision boundaries around training points, risk of overfitting
- For RBF, $\gamma = 0.01$ balances smoothness and flexibility
- For Polynomial, higher $\gamma = 0.1$ with lower degree ($d = 2$) works best

Polynomial Degree (d):

- Degree 2 performs better than degree 3 for this dataset
- Lower degree with appropriate γ provides sufficient non-linearity without overfitting
- Higher degrees can lead to overfitting and numerical instability

Custom Kernel Analysis The custom kernel (Linear + RBF) achieves 95.80% test accuracy, which is:

- Lower than optimized RBF (98.20%)
- Similar to basic Linear and RBF with default parameters (95%)
- Demonstrates that simple kernel combination doesn't always outperform well-tuned single kernels

Why doesn't the custom kernel perform better?

1. **Redundancy**: The RBF kernel already captures both linear and non-linear patterns effectively. Adding a linear component may not provide additional discriminative power.
2. **Weighting**: The simple sum gives equal weight to both components. A weighted combination $k_{\text{custom}} = w_1 k_{\text{linear}} + w_2 k_{\text{RBF}}$ with optimized weights might perform better.
3. **Parameter Sensitivity**: The custom kernel may require different optimal C and γ values than the individual kernels. Our grid search might not have explored the optimal parameter space.

Potential Improvements:

- Implement weighted kernel combination: $k = w \cdot k_{\text{linear}} + (1 - w) \cdot k_{\text{RBF}}$
- Expand parameter grid search range
- Try other kernel combinations (e.g., RBF + Polynomial)

Cross-Validation Importance The grid search with 5-fold cross-validation is crucial for:

1. **Hyperparameter Selection**: Systematically evaluates parameter combinations to find optimal settings
2. **Avoiding Overfitting**: Tests on held-out validation data prevent overfitting to training data
3. **Generalization Estimate**: Provides reliable estimate of model performance on unseen data
4. **Robustness**: 5-fold CV reduces variance in performance estimates compared to single train-test split

The close agreement between CV accuracy (98.30%) and test accuracy (98.20%) for the best RBF model validates the effectiveness of our cross-validation approach.

3 Conclusion

This homework successfully implemented Gaussian Process Regression and Support Vector Machine classification, demonstrating the importance of kernel selection and hyperparameter optimization.

Key Achievements:

- **Gaussian Process**: Optimized Rational Quadratic kernel parameters achieved 9.37% improvement in negative log-likelihood (55.923 $\rightarrow$ 50.680), with significantly narrower confidence intervals (40% reduction in dense data regions).
- **Support Vector Machine**: Achieved 98.20% test accuracy on MNIST (digits 0-4) using optimized RBF kernel ($C = 10$, $\gamma = 0.01$). Grid search with 5-fold cross-validation proved essential for finding optimal hyperparameters.
- **Implementation**: Successfully used numpy/scipy for GP and libsvm for SVM, meeting the no-sklearn requirement while demonstrating proficiency with lower-level ML libraries.

Key Insights:

1. Hyperparameter tuning is critical—polynomial kernel performance improved from 34.68% to 98.16% with proper γ and degree selection.
2. RBF kernel provides the best balance of performance (98.20%) and robustness for image classification, though linear kernel achieves competitive results (96.74%) with minimal computation.
3. Proper uncertainty quantification in GP enables informed decision-making by explicitly modeling prediction confidence across input space.

The experiments validate the theoretical foundations of kernel methods while providing practical experience in implementation and optimization of probabilistic and discriminative models.