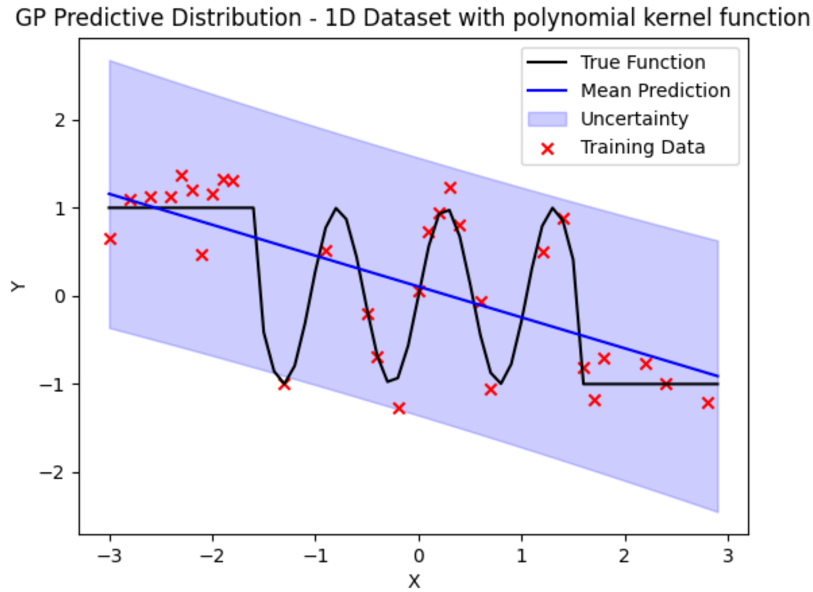
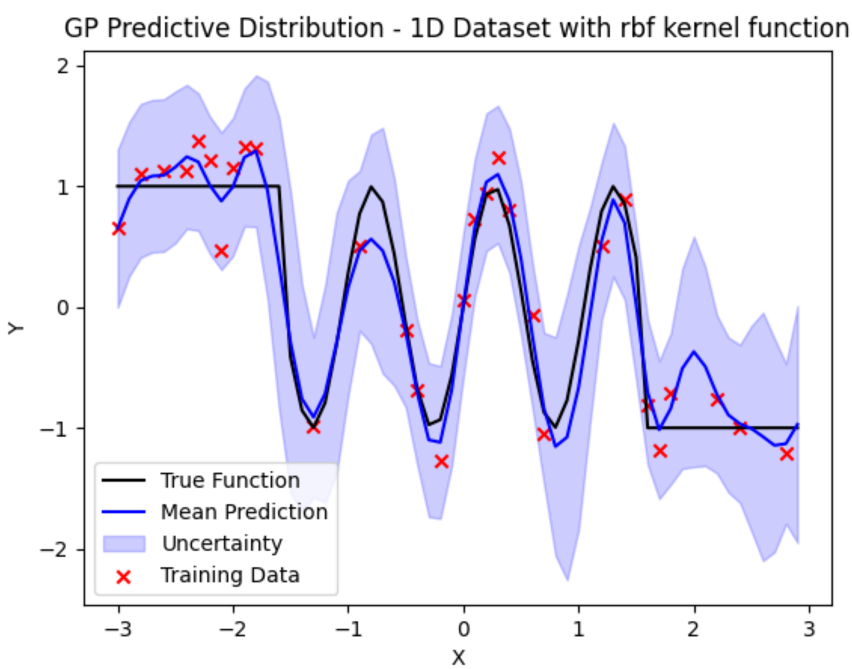
Sunilsakthivel Sakthi Velavan
Roni Khardon
CSCI-B 555
20 Nov 2023

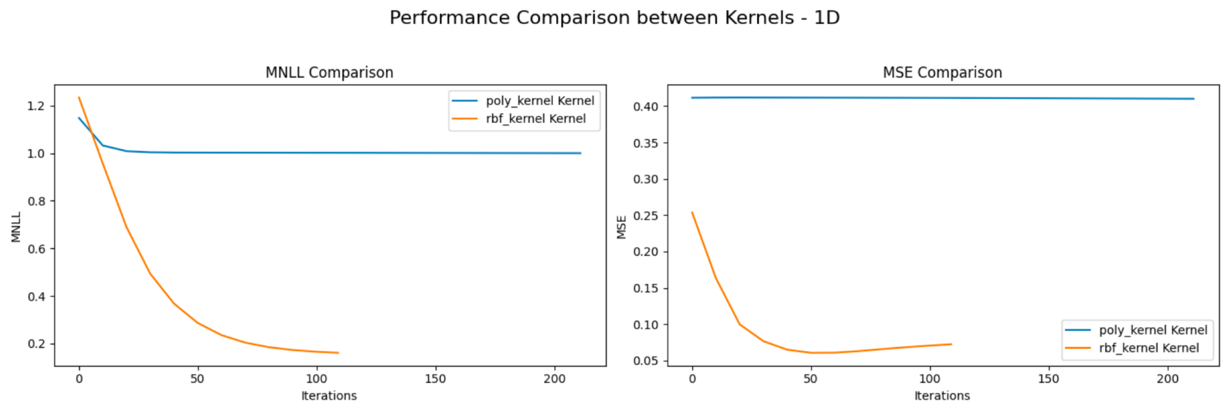
Programming Project 4

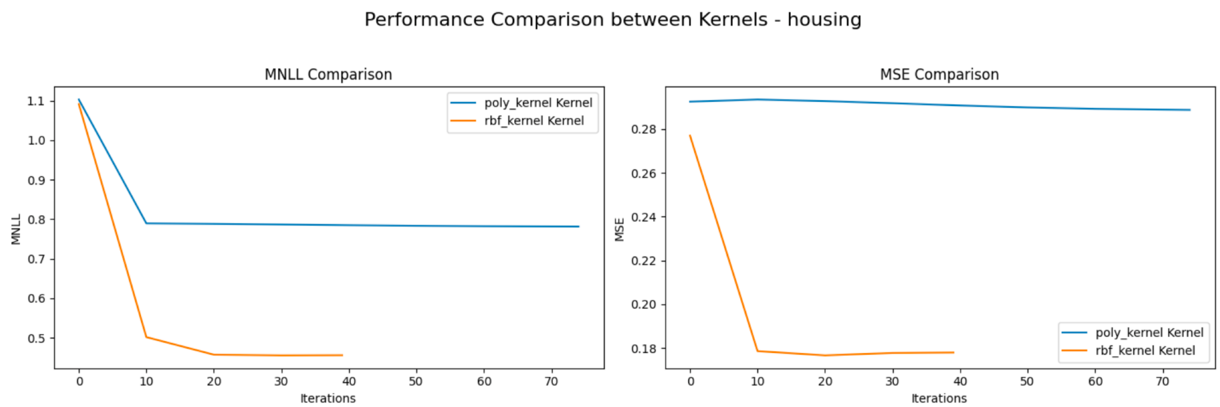
Visualizing performance on the 1D dataset:

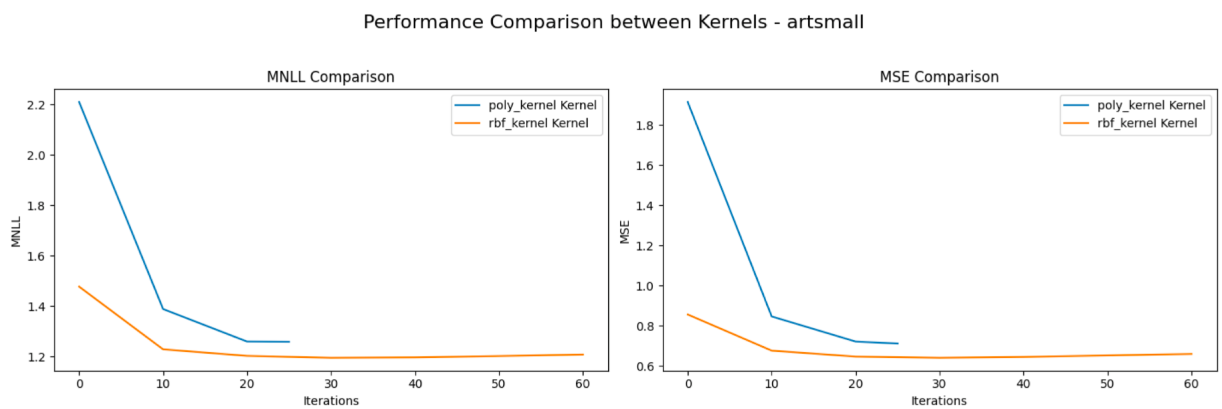


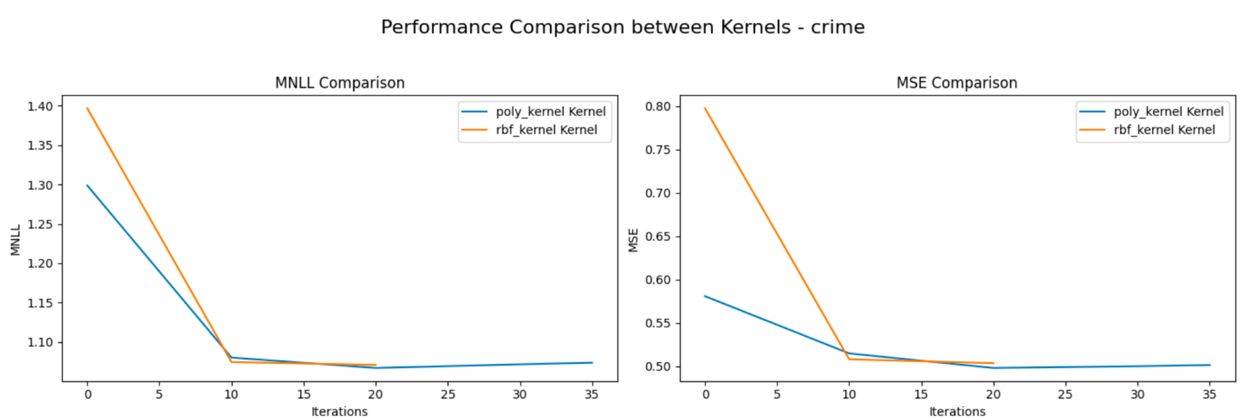


Performance as a function of Iterations:









Comparison of Bayesian Linear Regression:

Gaussian Process Results:

Dataset	*MSE*	*alpha*	*beta*
crime	0.50151866890626	320.669764901518	2.61138271021256
artsmall	0.71240591135867	140.89509999224	4.13443167341271
housing	0.28866544127906	18.7018324256571	4.00469585550641
1D	0.41025834394749	5.25808956496352	1.92914059346791

Bayesian Linear Regression Results:

Dataset	*MSE*	*alpha*	*beta*
crime	0.5	357.5	2.6
artsmall	0.716	141.4	4.23
housing	0.288	20.4	4.0
1D	0.39	7.5	1.9

Discussion of Results:

The recorded Mean Squared Error (MSE) values for both Bayesian Linear Regression (BLR) and Gaussian Processes (GP) with a linear kernel demonstrate a noteworthy similarity. In general, the two algorithms exhibit comparable test errors across different datasets. This consistency in performance indicates that, despite differences in their underlying principles and the way we generate the hyperparameters, both models are adept at capturing the relationships within the given datasets. As one can tell the MSE values on Gaussian Process pretty closely reflect the MSE values in the BLR method. We can also see that, for the most part, both the alpha and beta values in Gaussian Process seem to mirror the BLR values. We do notice that for the crime and 1D dataset, the alpha values seem to deviate from the BLR alphas. While this isn't necessarily a bad thing considering that the MSE is more or less the same, The deviation in alpha values for the crime and 1D datasets suggests a nuanced relationship between the Gaussian Process hyperparameter tuning and the underlying characteristics of the data. This divergence may stem from the inherent differences in the modeling assumptions and methodologies between the two approaches.

As for the RBF kernel, we can take a closer look at the graphs generated as part of the “Performance as a function of iterations” to see the recorded Mean Negative Log Likelihood (MNLL) and the MSE values given the selected alpha, beta, and s and the respective predicted values they generated. A notable trend emerges where the RBF kernel consistently converges at a lower error value than the polynomial (poly) kernel, except for the crime dataset. This divergence in performance prompts a closer inspection into the intrinsic qualities of the crime dataset that challenge the RBF kernel's conventional superiority. Moreover, the RBF kernel consistently achieves its lowest error at fewer iterations compared to the linear kernel, except for the artsmall dataset, where the flexibility of the RBF kernel seems to require additional iterations for convergence. This observation underscores the importance of tailoring kernel choices to specific dataset characteristics. The RBF kernel, with its capacity to capture non-linear relationships, excels in datasets demanding a more complex model, while the linear kernel remains proficient in situations where linearity is predominant.

While the linear kernel offers computational efficiency and interpretability, it may falter in the face of intricate relationships present in datasets such as artsmall. On the other hand, the RBF kernel's enhanced flexibility comes at the cost of increased computational complexity, making it crucial to weigh the trade-offs based on the specific demands of the given dataset.

README:

```
README:

Just run the PP4.py file to get the necessary graphs.
The oneD_main() essentially runs all of "Visualizing performance on the 1D
dataset" section.
plot_performance_curves() should take care of "Performance as a function of
iterations" and
"Comparison to Bayesian Linear Regression" plots and actual MSE and
alpha, beta values.
```

Codebase:

```
import numpy as np
import matplotlib

matplotlib.use('TkAgg')
import matplotlib.pyplot as plt
import pandas as pd


def poly_kernel(x1, x2, s):
```

```
    return np.dot(x1.T, x2) + 1


def rbf_kernel(x1, x2, s):
    return np.exp(-0.5 * np.linalg.norm(x1 - x2) ** 2 / s ** 2)


def covariance_matrix(Xi, Xj, kernel, s):
    ni = len(Xi)
    nj = len(Xj)

    K = np.zeros((ni, nj))

    for i in range(ni):
        for j in range(nj):
            K[i, j] = kernel(Xi[i], Xj[j], s)
    return K


def load_dataset(dataset):
    train_data = pd.read_csv('pp4data/train-' + dataset + '.csv', header=None)
    train_labels = pd.read_csv('pp4data/trainR-' + dataset + '.csv',
header=None)

    X_train = train_data.values
    y_train = train_labels.values.flatten()

    test_data = pd.read_csv('pp4data/test-' + dataset + '.csv', header=None)
    test_labels = pd.read_csv('pp4data/testR-' + dataset + '.csv', header=None)

    X_test = test_data.values
    y_test = test_labels.values.flatten()

    # Define the true function for visualization purposes
    def true_function(x):
        return np.where(x > 1.5, -1, np.where(x < -1.5, 1, np.sin(6 * x)))

    return X_train, y_train, X_test, y_test, true_function


def gp_predict(X_train, y_train, X_test, kernel, s, alpha, beta):

    # Cn
    K_train_train = covariance_matrix(X_train, X_train, kernel, s)
    Cn = ((1 / beta) * np.identity(len(X_train))) + ((1 / alpha) *
K_train_train)
    Cn_inv = np.linalg.inv(Cn)

    # vT
```

```
    vT = (1/alpha) * covariance_matrix(X_test, X_train, kernel, s)

    # c
    c = (1/alpha) * covariance_matrix(X_test, X_test, kernel, s) + ((1 / beta) *
np.identity(len(X_test)))

    mean = vT.dot(Cn_inv).dot(y_train)
    covariance = c - vT.dot(Cn_inv).dot(vT.T)

    return mean, covariance


def log_evidence(X, y, alpha, beta, s, kernel):
    n = len(X)
    K = covariance_matrix(X, X, kernel, s)
    Cn = ((1/beta) * np.identity(n)) + ((1/alpha) * K)
    Cn_inv = np.linalg.inv(Cn)

    a = -(n/2)*np.log(2*np.pi)
    b = -(1/2)*np.log(np.linalg.det(Cn))
    c = -(1/2)*(y.T.dot(Cn_inv).dot(y))

    return a + b + c


def compute_gradients(X, y, alpha, beta, s, kernel):
    n = len(X)
    K = covariance_matrix(X, X, kernel, s)
    Cn = ((1/beta) * np.identity(n)) + ((1/alpha) * K)
    Cn_inv = np.linalg.inv(Cn)

    # Compute the gradient with respect to alpha
    d_Cn_d_alpha = (-K/alpha**2)
    d_log_evidence_d_alpha = ((-1/2) * np.trace(Cn_inv.dot(d_Cn_d_alpha))) +
((1/2)*y.T.dot(Cn_inv).dot(d_Cn_d_alpha).dot(Cn_inv).dot(y))

    # Compute the gradient with respect to beta
    d_Cn_d_beta = (-np.identity(n) / beta**2)
    d_log_evidence_d_beta = ((-1/2) * np.trace(Cn_inv.dot(d_Cn_d_beta))) +
((1/2)*y.T.dot(Cn_inv).dot(d_Cn_d_beta).dot(Cn_inv).dot(y))

    # Compute the gradient with respect to s using the specified kernel
    dCn_ds = np.zeros((n, n))
    if kernel == rbf_kernel:
        for i in range(n):
            for j in range(n):
                if kernel == rbf_kernel:
                    term1 = kernel(X[i], X[j], s)
                    term2 = (np.linalg.norm(X[i] - X[j]) ** 2)/s**3
```

```
                    dCn_ds[i, j] = (1/alpha) * term1 * term2

    d_log_evidence_d_s = ((-1/2) * np.trace(Cn_inv.dot(dCn_ds))) + ((1/2) *
y.T.dot(Cn_inv).dot(dCn_ds).dot(Cn_inv).dot(y))

    return d_log_evidence_d_alpha, d_log_evidence_d_beta, d_log_evidence_d_s


def optimize_hyperparameters(X_train, y_train, alpha, beta, s, kernel,
learning_rate=0.01, max_iterations=1000,
                             tolerance=10e-5):
    iteration = 0
    a = log_evidence(X_train, y_train, alpha, beta, s, kernel)

    while iteration < max_iterations:
        # Compute gradients of log evidence with respect to alpha, beta, and s
        d_log_evidence_d_alpha, d_log_evidence_d_beta, d_log_evidence_d_s =
compute_gradients(X_train, y_train, alpha,

beta, s, kernel)

        alpha = np.exp(np.log(alpha) + learning_rate *
(d_log_evidence_d_alpha*alpha))
        beta = np.exp(np.log(beta) + learning_rate *
(d_log_evidence_d_beta*beta))
        s = np.exp(np.log(s) + learning_rate * (d_log_evidence_d_s*s))


        # Compute log evidence with updated hyperparameters
        b = log_evidence(X_train, y_train, alpha, beta, s, kernel)

        # Check for convergence
        c = (b - a) / abs(a)

        if c <= tolerance:
            break

        a = b
        iteration += 1

    return alpha, beta, s, iteration

def visualize_1d_results(X_train, y_train, X_test, mean, covariance,
true_function, k):
    # Sort X_test for better visualization
    X_test = X_test.flatten()
    sorted_indices = np.argsort(X_test)
    X_test_sorted = X_test[sorted_indices]
```

```
    mean_sorted = mean[sorted_indices]
    std_deviation_sorted = np.sqrt(np.diag(covariance))[sorted_indices]

    # Plot the true function
    plt.plot(X_test_sorted, true_function(X_test_sorted), label="True Function",
color="black")

    # Plot the mean of the predictive distribution
    plt.plot(X_test_sorted, mean_sorted, label="Mean Prediction", color="blue")

    # Plot uncertainty (±2 standard deviations around the mean)
    plt.fill_between(X_test_sorted, mean_sorted - 2 * std_deviation_sorted,
mean_sorted + 2 * std_deviation_sorted,
                        alpha=0.2, color="blue", label="Uncertainty")

    # Scatter plot of training data
    plt.scatter(X_train, y_train, color="red", marker="x", label="Training
Data")

    kernel = "polynomial kernel" if k == poly_kernel else "rbf kernel"

    plt.xlabel("X")
    plt.ylabel("Y")
    plt.title("GP Predictive Distribution - 1D Dataset with " + kernel + "
function")
    plt.legend()
    plt.show()


def calculate_mnll(true_labels, predictive_mean, predictive_variance):
    true_labels = np.array(true_labels)
    predictive_mean = np.array(predictive_mean)
    predictive_variance = np.array(predictive_variance)

    # Calculate MNLL
    log_likelihoods = (
        -0.5 * np.log(2 * np.pi * predictive_variance)
        - 0.5 * ((true_labels - predictive_mean) ** 2) / predictive_variance
    )
    mnll = -np.mean(log_likelihoods)

    return mnll


def calculate_mse(true_labels, predictive_mean):
    # Ensure true_labels and predictive_mean are NumPy arrays
    true_labels = np.array(true_labels)
    predictive_mean = np.array(predictive_mean)
```

```
    # Calculate MSE
    mse = np.mean((predictive_mean - true_labels) ** 2)

    return mse


def oneD_main():
    # Load your 1D dataset
    X_train, y_train, X_test, y_test, true_function = load_dataset("1D")
    kernels = [poly_kernel, rbf_kernel]

    for k in kernels:
        alpha, beta, s, iteration = optimize_hyperparameters(X_train, y_train,
1.0, 1.0, 0.1, k)
        print("alpha: "+str(alpha)+" beta: "+str(beta)+" s: "+str(s)+"
convergence iterations: "+str(iteration))

        mean, covariance = gp_predict(X_train, y_train, X_test, k, s, alpha,
beta)
        visualize_1d_results(X_train, y_train, X_test, mean, covariance,
true_function, k)


def evaluate_performance(X_train, y_train, X_test, y_test, true_function,
kernel, max_iterations=1000,
                        eval_interval=10):
    mnll_list = []
    mse_list = []
    iterations = []
    iter_val = 0

    alpha = 1.0
    beta = 1.0
    s = 5
    if X_train.shape == (30, 1):
        s = .1

    alpha, beta, s, iteration = optimize_hyperparameters(X_train, y_train,
alpha, beta, s, kernel, max_iterations=0)

    mean, covariance = gp_predict(X_train, y_train, X_test, kernel, s, alpha,
beta)
    mnll = calculate_mnll(y_test, mean, np.diag(covariance))
    mse = calculate_mse(y_test, mean)
    mnll_list.append(mnll)
    mse_list.append(mse)
    iterations.append(iter_val)

    while iter_val != max_iterations:
```

```
        alpha, beta, s, iteration = optimize_hyperparameters(X_train, y_train,
alpha, beta, s, kernel, max_iterations=10)

        if iteration != 0:
            mean, covariance = gp_predict(X_train, y_train, X_test, kernel, s,
alpha, beta)

            # Calculate MNLL and MSE
            mnll = calculate_mnll(y_test, mean, np.diag(covariance))
            mse = calculate_mse(y_test, mean)
            mnll_list.append(mnll)
            mse_list.append(mse)

            iter_val += iteration
            iterations.append(iter_val)

        if iteration != 10:
            break

    print("MSE error: "+str(mse_list[len(mse_list)-1])+" alpha: "+str(alpha)+"
beta: "+str(beta))
    return mnll_list, mse_list, iterations


def plot_performance_curves(datasets, kernels):
    for i, dataset in enumerate(datasets):
        fig, (axes_mnll, axes_mse) = plt.subplots(1, 2, figsize=(15, 5))

        fig.suptitle(f'Performance Comparison between Kernels - {dataset}',
fontsize=16)
        print('\n')
        print(dataset+" dataset:")
        mnll_legend_labels = []
        mse_legend_labels = []

        for j, kernel in enumerate(kernels):
            X_train, y_train, X_test, y_test, true_function =
load_dataset(dataset)
            mnll_list, mse_list, iterations = evaluate_performance(X_train,
y_train, X_test, y_test, true_function, kernel, max_iterations=1000,

eval_interval=10)

            # Plot MNLL
            axes_mnll.plot(iterations, mnll_list, label=f'{kernel.__name__}
Kernel')
            mnll_legend_labels.append(f'{kernel.__name__} Kernel')

            # Plot MSE
```

```
            axes_mse.plot(iterations, mse_list, label=f'{kernel.__name__}
Kernel')
            mse_legend_labels.append(f'{kernel.__name__} Kernel')

        # Set legend for MNLL plot
        axes_mnll.set_title('MNLL Comparison')
        axes_mnll.set_xlabel('Iterations')
        axes_mnll.set_ylabel('MNLL')
        axes_mnll.legend(mnll_legend_labels)

        # Set legend for MSE plot
        axes_mse.set_title('MSE Comparison')
        axes_mse.set_xlabel('Iterations')
        axes_mse.set_ylabel('MSE')
        axes_mse.legend(mse_legend_labels)

        plt.tight_layout(rect=[0, 0.03, 1, 0.95])
        plt.show()

if __name__ == "__main__":
    oneD_main()
    datasets = ['1D', 'housing', 'artsmall', 'crime']
    kernels = [poly_kernel, rbf_kernel]

    plot_performance_curves(datasets, kernels)
```